



UNIVERSITÀ⁹ DEGLI STUDI DI
NAPOLI FEDERICO II

Scuola Politecnica e delle Scienze di Base
Corso di Laurea Magistrale in Ingegneria Informatica

*Progetto finale di Architettura di Sistemi
Digitali*

Anno Accademico 2021/2022

Gruppo 23
Alfonso Conte, Daniele Fazzari, Francesco Iannaccone, Matteo Conti

Indice

1 Multiplexer	1
1.1 Traccia	2
1.2 Descrizione soluzione teorica	3
1.3 Componenti progettati	4
1.3.1 Multiplexer 2:1	4
1.3.2 Multiplexer 4:1	6
1.3.3 Multiplexer 16:1	8
1.3.4 Demultiplexer 1:4	10
1.3.5 Rete di interconnessione	11
1.4 Testing simulato	13
1.5 Caricamento sulla scheda	14
2 Encoder BCD	17
2.1 Traccia	17
2.2 Descrizione soluzione teorica	18
2.3 Schematici, Componenti ed Implementazione in VHDL	19
2.3.1 Arbitro di priorità	19
2.3.2 Encoder	21
2.3.3 PriorityEncoder	22
2.4 Testing simulato	23
2.5 Caricamento sulla scheda	23
2.6 Display a 7 segmenti	24
2.6.1 Visualizzatore a 7 segmenti	25
2.6.2 Priority Encoder BCD con display a 7 segmenti	28
2.6.3 Timing analysis	30
3 Riconoscitore di sequenze	32
3.1 Traccia	32
3.2 Descrizione soluzione teorica	33
3.3 Schematici	36

3.3.1	Riconoscitore	36
3.3.2	Button debouncer	37
3.4	Implementazione in VHDL	39
3.4.1	Top module	46
3.5	Testing simulato	48
3.6	Caricamento sulla scheda	49
4	Shift register	51
4.1	Traccia	51
4.2	Descrizione soluzione teorica	52
4.3	Approccio strutturale	53
4.3.1	Il flip-flop D	53
4.3.2	Top module	54
4.4	Approccio comportamentale	59
4.5	Testing simulato	61
4.6	Caricamento sulla scheda	63
5	Cronometro	64
5.1	Traccia	64
5.2	Descrizione soluzione teorica	65
5.3	Schematici	68
5.3.1	Contatore	68
5.3.2	Clock	68
5.3.3	BCD Converter	70
5.3.4	Memoria	71
5.3.5	Flip Flop T	72
5.3.6	Unità operativa	73
5.3.7	Unità di controllo	75
5.4	Implementazione in VHDL	77
5.4.1	Flip Flop T	77
5.4.2	Contatore	78
5.4.3	Clock	80
5.4.4	BCD Converter	82
5.4.5	Memoria	83
5.4.6	Unità operativa	84
5.4.7	Unità di controllo	88
5.4.8	Top module	92
5.5	Testing simulato	95
5.6	Caricamento sulla scheda	96

6 Sistema di testing	98
6.1 Traccia	98
6.2 Descrizione soluzione teorica	99
6.3 Schematici, componenti	99
6.4 Implementazione in VHDL	103
6.5 Testing simulato	113
6.6 Caricamento sulla scheda	114
7 Comunicazione con handshaking	115
7.1 Traccia	116
7.2 Descrizione soluzione teorica	116
7.3 Schematici, componenti	118
7.4 Implementazione in VHDL	123
7.5 Testing simulato	137
7.6 Caricamento sulla scheda	138
8 Il processore MIC-1	140
8.1 Traccia	140
8.2 La struttura del processore	141
8.3 Comandi analizzati	143
8.3.1 BIPUSH	143
8.3.2 IF_ICMPEQ	145
8.4 Modifiche effettuate	149
9 Interfaccia seriale	152
9.1 Traccia	152
9.2 Descrizione soluzione teorica	153
9.3 Schematici	155
9.3.1 UART Component	155
9.3.2 Nodo A	155
9.3.3 Nodo B	156
9.4 Implementazione in VHDL	158
9.4.1 Nodo A	158
9.4.2 Nodo B	162
9.4.3 Top module	165
9.5 Testing simulato	167
9.6 Caricamento sulla scheda	167

10 Switch multistato	169
10.1 Traccia	171
10.2 Descrizione soluzione teorica	171
10.3 Schematici, componenti	172
10.4 Implementazione in VHDL	175
10.5 Testing simulato	180
11 Il moltiplicatore di Robertson	182
11.1 Traccia	183
11.2 Descrizione soluzione teorica	183
11.3 Schematici, componenti	185
11.4 Implementazione VHDL	188
11.4.1 Il ripple carry adder	188
11.4.2 Unità di controllo	192
11.5 Testing simulato	196
11.6 Caricamento sulla scheda	197
12 Addizionatore in virgola mobile	198
12.1 Descrizione della soluzione	199
12.2 Schematici, componenti	200
12.2.1 Unità operativa	200
12.2.2 Unità di controllo	202
12.3 Implementazione in VHDL	203
12.3.1 Blocchi di conversione	203
12.3.2 Unità operativa	206
12.3.3 Unità di controllo	211
12.4 Testing simulato	214
12.5 Caricamento sulla scheda	218

Progetto 1

Multiplexer

In questo primo esercizio ci è stato richiesto di risolvere una problematica di interconnessione progettando e realizzando una delle reti combinatorie più ampiamente utilizzata, il multiplexer. Questa particolare rete combinatoria trova un'ampia gamma di applicazioni in tutti i campi relativi all'architettura dei sistemi digitali. Si tratta senza dubbio di uno dei "blocchi fondamentali", essendo, assieme al componente speculare, il demultiplexer, alla base di gran parte dei meccanismi di interconnessione tra sottosistemi che realizzando l'architettura di un sistema complesso. Il multiplexer consente infatti di collegare più segnali di sorgente ad un'unica destinazione, operando come selettore di dati controllato.

Questo dispositivo, nella sua versione più semplice, presenta tipicamente in ingresso M linee di selezione e $N = 2^M$ linee dato ed in uscita un singolo segnale. Il dispositivo opera collegando una specifica linea di ingresso, determinata tramite gli ingressi di selezione, all'unica linea di uscita. In ingresso alle linee di selezione è posta una stringa binaria che identifica una specifica linea di ingresso mediante un numero codificato in rappresentazione binaria posizionale.

Uno degli aspetti più interessanti relativi alla progettazione di questo dispositivo è la semplicità con la quale è possibile comporre multiplexer di dimensioni inferiori per produrre un unico multiplexer più grande consentendo uno sviluppo

modulare ed incentrato sul riuso dei componenti.

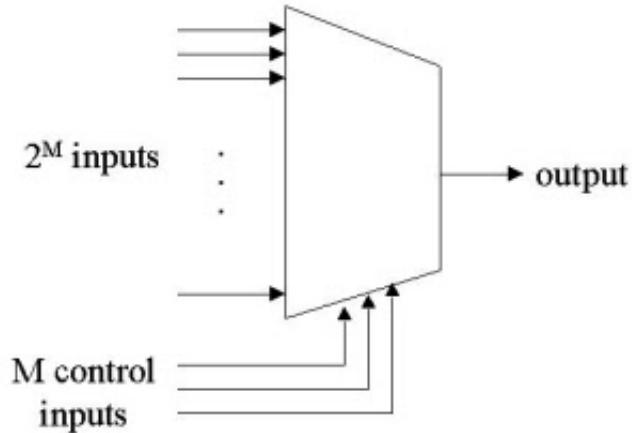


Figura 1.1: Schema ingressi ed uscite di un multiplexer $N = 2^M : 1$

1.1 Traccia

La traccia dell’elaborato, suddivisa in più punti è la seguente:

- 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.
- 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.
- Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi possono essere precaricati nel sistema oppure immessi anch’essi mediante switch, sviluppando in questo secondo caso un’apposita rete di controllo per l’acquisizione.

1.2 Descrizione soluzione teorica

Abbiamo affrontato la traccia in maniera incrementale, iniziando dalla progettazione e del *testing* dei componenti più semplici per poi comporli in dispositivi più complessi. Abbiamo quindi progettato inizialmente il componente più semplice, il multiplexer 2 : 1, e partendo dalle equazioni booleane che ne descrivono il comportamento, ne abbiamo ricavato una descrizione dataflow di basso livello. Il passo successivo è stato poi quello di comporre, come descritto nel dettaglio in seguito, tre mux 2 : 1 per ricavarne una descrizione strutturale del multiplexer 4 : 1. Componendo poi cinque mux 4 : 1, abbiamo ottenuto una descrizione strutturale del componente di interesse, il multiplexer 16 : 1.

Per il punto successivo della traccia, ovvero per la realizzazione di una rete di interconnessione a sedici sorgenti e quattro destinazioni, abbiamo progettato un secondo componente, un demultiplexer 1 : 4, un'altra macchina combinatoria notevole, la cui architettura è descritta nel dettaglio in seguito. La rete di interconnessione è stata realizzata connettendo l'uscita del multiplexer all'ingresso del demultiplexer. Le sedici destinazioni possono quindi, in questo modo, essere connesse alle quattro destinazioni a partire da sei ingressi di selezione, quattro per il mux e due per il demux.

L'implementazione su board è stato l'ultimo dei problemi che abbiamo dovuto affrontare. Ci siamo però trovati di fronte ad un vincolo tecnologico. Mentre infatti è stato semplice mappare le quattro destinazioni ad altrettanti led e le sei linee di selezione a degli *switch* della board, questa non dispone di un numero sufficiente di *switch* per coprire tutti i sedici possibili ingressi.

Per risolvere questa limitazione abbiamo deciso di arricchire il sistema, prima di caricarlo sulla scheda, con un componente aggiuntivo, una ROM, i cui dettagli sono approfonditi in seguito, in cui sono stati pre-caricati alcune possibili combinazioni dei sedici ingressi dato. L'idea è quindi quella di ridurre la dimensionalità

dell'ingresso non consentendo di selezionare un'arbitraria configurazione dei sedici ingressi dato, ma consentire di selezionare esclusivamente una delle configurazioni precaricate in memoria mediante degli appositi ingressi di selezione della ROM.

1.3 Componenti progettati

1.3.1 Multiplexer 2:1

Il dispositivo più semplice su cui si basa l'implementazione di tutti i componenti realizzati in seguito è il multiplexer 2 : 1.

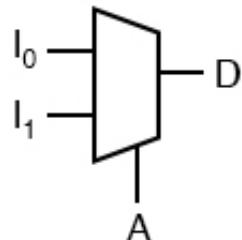


Figura 1.2: Interfaccia del multiplexer 2:1

Questo dispositivo presenta un'unica linea di uscita ed esclusivamente tre ingressi:

- un'unica linea di selezione
- due linee dato in ingresso

L'interfaccia, riportata in figura 1.2, può essere quindi facilmente descritta in VHDL come segue:

```
entity Mux2_1 is
  Port (
    X : in STD_LOGIC_VECTOR(1 downto 0);
    S : in STD_LOGIC;
    D : out STD_LOGIC);
```

```

Y : out STD_LOGIC
);
end Mux2_1;

```

Per l'implementazione di questo dispositivo abbiamo utilizzato un approccio *dataflow*, non è infatti difficile scrivere le equazioni circuitali che regolano il comportamento di questa semplice rete combinatoria, risulta infatti che l'uscita Y può essere calcolata a partire dagli ingresso con la seguente equazione booleana:

$$Y = (\bar{S} \cdot X_0) + (S \cdot x_1)$$

L'uscita Y è infatti pari all'ingresso X_0 quando la linea di selezione è bassa, è invece pari ad X_1 quando è alta. La descrizione della sua architettura in VHDL è quindi la seguente:

```

architecture Dataflow of Mux2_1 is
begin
    Y <= (X(0) AND (NOT(S))) OR (X(1) AND S);
end Dataflow;

```

L'equazione descrive quindi una rete combinatoria analoga a quella riportata in figura 1.3. L'implementazione su FPGA, ad ogni modo, non sarà realizzata a partire da combinazione di porte logiche ma tramite *look-up table* in cui sarà memorizzata la tabella di verità che regola il comportamento di questo oggetto.

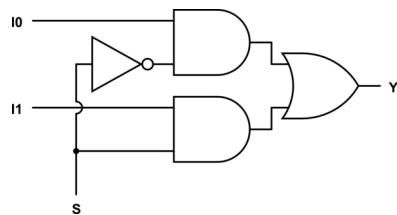


Figura 1.3: Realizzazione del mux 2:1 a partire da porte logiche.

1.3.2 Multiplexer 4:1

Il multiplexer 4:1 è un dispositivo che presenta sei ingressi di cui:

- due linee di selezione
- quattro linee di ingresso

ed un'unica linea di uscita. La sua interfaccia è quindi stata descritta in VHDL come segue:

```
entity Mux4_1 is
    Port(
        X : in STD_LOGIC_VECTOR(3 downto 0);
        S : in STD_LOGIC_VECTOR(1 downto 0);
        Y : out STD_LOGIC
    );
end Mux4_1;
```

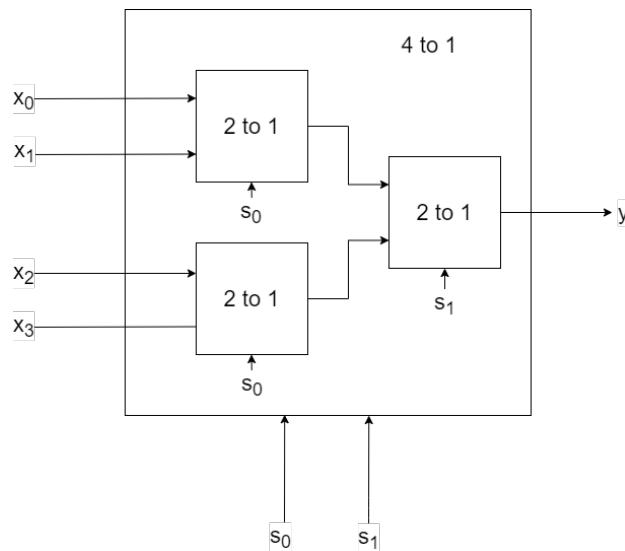


Figura 1.4: Interfaccia e composizione strutturale del Multiplexer 4:1.

L'architettura di questo componente è stata realizzata, seguendo una descrizione strutturale, per composizione di tre multiplexer 2:1 seguendo lo schema

rappresentato in figura 1.4. I primi due dispositivi, pilotati dalla linea di selezione s_0 operano una selezione tra x_0, x_1 e tra x_2, x_3 . Il terzo componente, pilotato da s_1 determina il risultato di quale dei due componenti dovrà essere portato in uscita.

In VHDL la descrizione strutturale è quindi la seguente:

```
architecture Structural of Mux4_1 is
COMPONENT Mux2_1 IS
    Port(
        X : in STD_LOGIC_VECTOR(1 downto 0);
        S : in STD_LOGIC;
        Y : out STD_LOGIC
    );
END COMPONENT;
signal U : STD_LOGIC_VECTOR(1 downto 0);
begin
    Mux0to2 : FOR i IN 0 TO 2 GENERATE
        M01: IF i < 2 GENERATE
            M : Mux2_1 PORT MAP (
                X => X(i*2+1 downto i*2),
                S => S(0),
                Y => U(i)
            );
        END GENERATE;
        M2: IF i = 2 GENERATE
            M : Mux2_1 PORT MAP (
                X => U,
                S => S(1),
                Y => Y
            );
        END GENERATE;
    END GENERATE;
end Structural;
```

dove per si è utilizzato il costrutto di sintesi *for generate* allo scopo di sintetizzare più mux 2:1.

1.3.3 Multiplexer 16:1

Il multiplexer 16:1 è un dispositivo che presenta nel complesso un'uscita e venti ingressi.

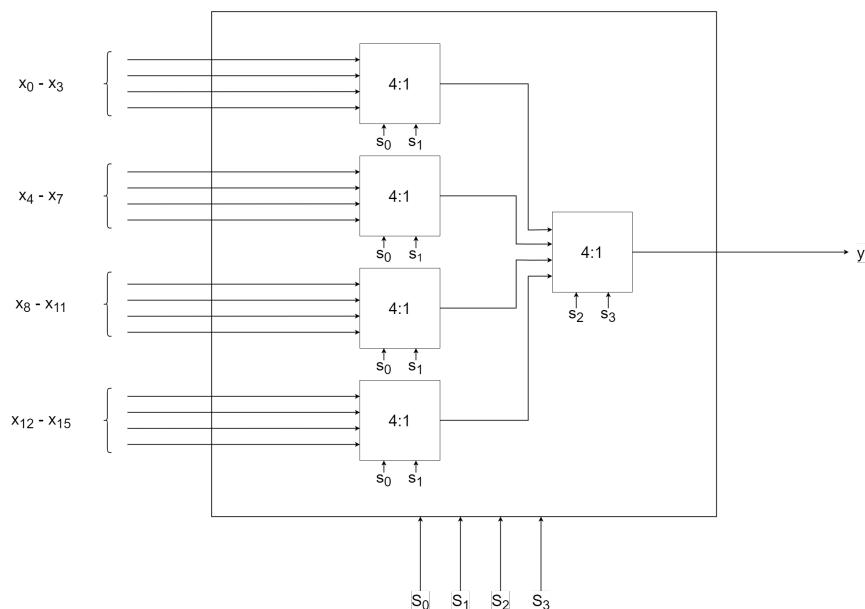


Figura 1.5: Interfaccia e composizione strutturale del Multiplexer 16:1.

Nello specifico è caratterizzato da :

- 16 ingressi dato,
- 4 ingressi di selezione,
- un'unica uscita

. La sua interfaccia, descritta in VHDL è quindi la seguente:

```
entity Mux16_1 is
  Port (
```

```

X : in STD_LOGIC_VECTOR(15 downto 0);
S : in STD_LOGIC_VECTOR(3 downto 0);
Y : out STD_LOGIC
);
end Mux16_1;

```

L'architettura di questo dispositivo è stata realizzata con un approccio strutturale componendo cinque multiplexer 4:1, seguendo lo schema rappresentato in figura 1.5. Non diversamente dal caso precedente, i multiplexer al primo livello operano una selezione tra sottoinsiemi contigui dell'ingresso, mentre il multiplexer posto al secondo livello opera la selezione tra quelli del primo livello, connettendo uno di questi all'uscita del dispositivo.

La descrizione strutturale del componente è la seguente:

```

architecture Structural of Mux16_1 is

COMPONENT Mux4_1 IS

Port (
    X : in STD_LOGIC_VECTOR(3 downto 0);
    S : in STD_LOGIC_VECTOR(1 downto 0);
    Y : out STD_LOGIC
);

END COMPONENT;

signal U : STD_LOGIC_VECTOR(3 downto 0);

begin

Mux0to4 : FOR i IN 0 TO 4 GENERATE
    M03: IF i < 4 GENERATE
        M : Mux4_1 PORT MAP (
            X => X(i*4+3 downto i*4),
            S => S(1 downto 0),
            Y => U(i)
        );

```

```
END GENERATE;

M4: IF i = 4 GENERATE
    M : Mux4_1 PORT MAP (
        X => U,
        S => S(3 downto 2),
        Y => Y
    );
END GENERATE;
END GENERATE;
end Structural;
```

In questo caso l'utilizzo del *for generate* diventa particolarmente importante allo scopo di garantire compattezza e leggibilità al codice, evitando di dover ripetere dichiarazioni analoghe e ridondanti.

1.3.4 Demultiplexer 1:4

Il demultiplexer è una macchina combinatoria notevole che consente di collegare un segnale di ingresso a più uscite, selezionando la specifica destinazione tramite degli ingressi di selezione analoghi a quelli descritti per il multiplexer.

Nel caso del demultiplexer 1:4 questo presenta un'interfaccia composta da:

- un ingresso dato,
- due ingressi di selezione,
- quattro uscite.

Per la realizzazione di questo componente abbiamo deciso di seguire un approccio semplice, realizzando il componente con una descrizione dataflow direttamente a partire dalla sua caratteristica.

Il codice VHDL che lo descrive è il seguente:

```
entity Demux1_4 is
    Port(
        X : in STD_LOGIC;
        S : in STD_LOGIC_VECTOR(1 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
    );
end Demux1_4;

architecture Dataflow of Demux1_4 is
begin
    with S select
        Y <= "000" & X when "00",
        "00" & X & "0" when "01",
        "0" & X & "00" when "10",
        X & "000" when "11",
        "----" when others;
end Dataflow;
```

In tal modo si assicura che tutte le uscite del dispositivo sono basse ad eccezione di quella selezionata tramite le apposite linee, il cui valore sarà direttamente pilotato dall'unico ingresso X .

Per la realizzazione del componente, oltre al tipico costrutto condizionale *select* è stato utilizzato l'utilissimo operatore di concatenazione tra stringhe $\&$.

1.3.5 Rete di interconnessione

Connettendo l'uscita del multiplexer 16:1 all'ingresso del demultiplexer 1:4 otteniamo in definitiva una rete di interconnessione 16:4 con sei linee di selezione, le

prime quattro permettono di determinare la sorgente e le ultime due, invece, la destinazione.

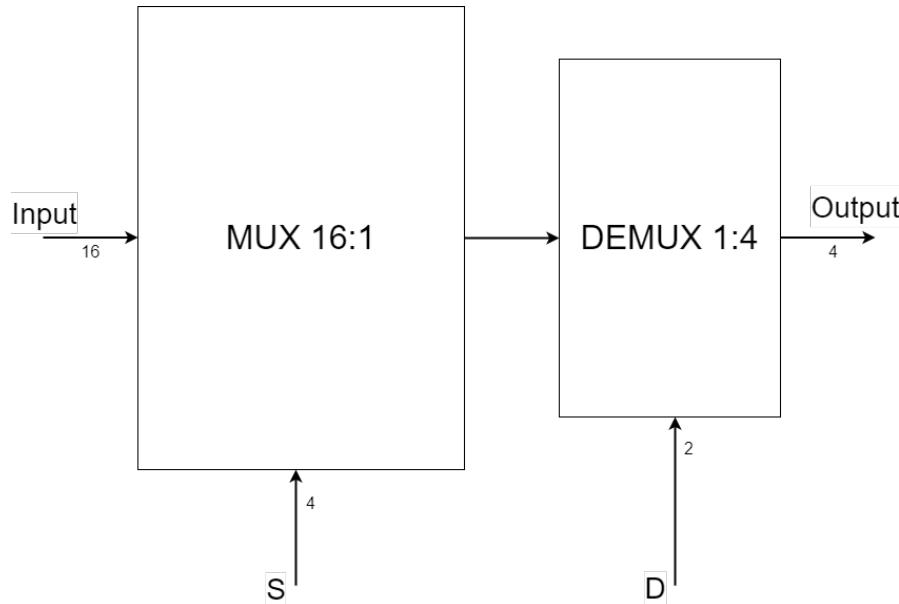


Figura 1.6: Schema della rete di connessione.

Facendo riferimento allo schema in figura 1.6, la descrizione dell’interfaccia di questa in VHDL è la seguente:

```

entity ConnectionNet is
    PORT (
        X : in STD_LOGIC_VECTOR(15 downto 0);
        S : in STD_LOGIC_VECTOR(3 downto 0);
        D : in STD_LOGIC_VECTOR(1 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
    );
end ConnectionNet;
    
```

La sua semplice implementazione, ottenuta per composizione delle reti fin qui discusse, è invece:

```
architecture Structural of ConnectionNet is
signal u : STD_LOGIC;

begin
    Mux : entity work.Mux16_1 PORT MAP (
        X => X, S => S, Y => U
    );
    Demux : entity work.Demux1_4 PORT MAP (
        X => u, S => D, Y => Y
    );
end Structural;
```

1.4 Testing simulato

In fase di sviluppo, dopo aver descritto ogni componente, ci siamo preoccupati di verificarne la corretta descrizione operando alcuni test simulati sfruttando l'ambiente di simulazione offerto dal software Vivado. Per brevità omettiamo in questa documentazione i codici VHDL utilizzati per la descrizione dei *testbench* che risultano sostanzialmente standard, ed i risultati, seppur positivi, del *testing* del multiplexer 2:1 e 4:1.

Si riporta invece il *testing* effettuato per il multiplexer 16:1 e per la rete di interconnessione composta da multiplexer e demultiplexer.

Come sottolineato dall'immagine 1.7 , il comportamento ottenuto in simulazione è, per tutti gli ingressi testati è quello atteso. E' importante sottolineare che non si tratta di una forma di testing esaustiva, ma, data la semplicità dei componenti realizzati, possiamo con un buon grado di sicurezza affermare che la descrizione VHDL fornisce un comportamento simulato adeguato alle specifiche.

Abbiamo poi effettuato il testing simulato di interconnessione, andando quindi a verificare la correttezza del comportamento della rete complessiva.

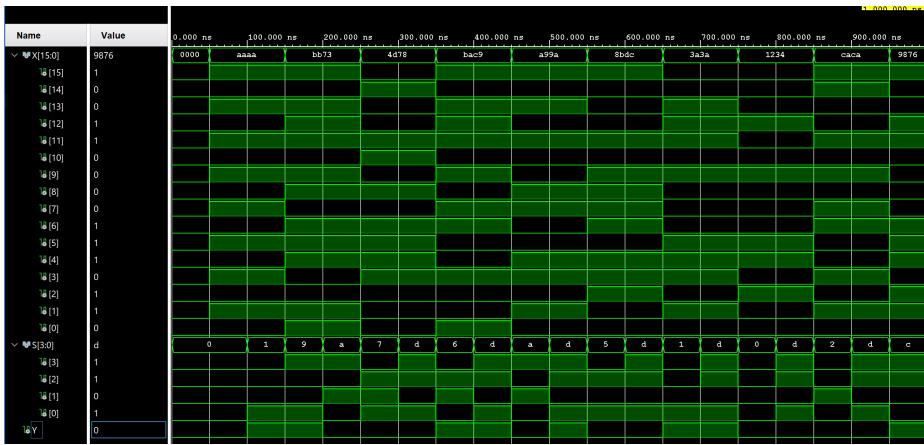


Figura 1.7: Test simulato del componente mux 16:1.

Anche in questo caso il comportamento del componente è quello atteso. Siamo quindi soddisfatti della descrizione VHDL della rete e pronti per la fase successiva di caricamento sulla scheda.

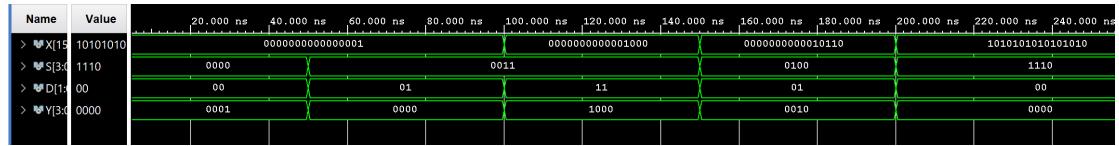


Figura 1.8: Test della rete complessiva.

1.5 Caricamento sulla scheda

Per poter caricare il progetto sulla scheda abbiamo prima dovuto, come descritto in precedenza, risolvere la problematica tecnologica associata con il numero insufficiente di switch presenti sulla scheda. Abbiamo deciso di adottare una metodologia semplice precaricando, come suggerito i valori delle sedici linee dato presenti della rete di connessione all'interno della scheda. Per far ciò abbiamo optato per l'utilizzo di un componente semplice ma estremamente utile e che sarà utilizzato più volte nei progetti presenti all'interno di questo elaborato, la ROM. Questo dispositivo, più dettagliatamente discusso nel capitolo sei, consente di memorizzare all'interno della scheda alcune stringhe binarie, permettendone l'accesso tramite apposite linee di selezione. Si riporta in figura 1.9 l'architettura complessiva.

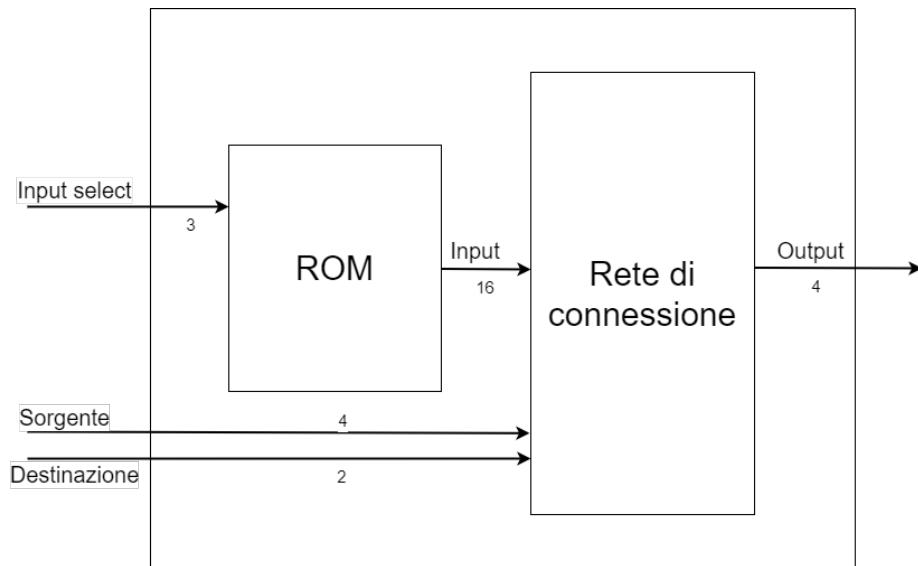


Figura 1.9: Schema del componente caricato sulla scheda.

Per il nostro problema abbiamo utilizzato una ROM 8x16, ovvero composta da otto locazioni da due byte l'una. Abbiamo poi connesso la memoria alla rete fin qui discussa.

Il codice VHDL realizzato per l'interconnessione è il seguente:

```

architecture Behavioral of ConnNet_on_board is

signal u : std_logic_vector(15 downto 0);
begin
  net :entity work.ConnectionNet port map(
    X=>u, S=>mux_sel, D => demux_sel, Y => leds
  );
  memory : entity work.ROM port map(
    addr => rom_sel, clk => clk, output => u
  );
end Behavioral;
  
```

Abbiamo quindi deciso di mappare gli ingressi e le uscite ai led e agli switch presenti sulla scheda. Sono stati dunque associati ai primi quattro switch della

PROGETTO 1. MULTIPLEXER

scheda gli input di selezione del multiplexer, ai successivi due quelli per la selezione del demultiplexer e, infine, gli ultimi tre switch sono stati utilizzati per prelevare l'indirizzo in ingresso alla ROM. Le quattro uscite della rete di connessione sono infine stati collegati ad altrettanti led della scheda.

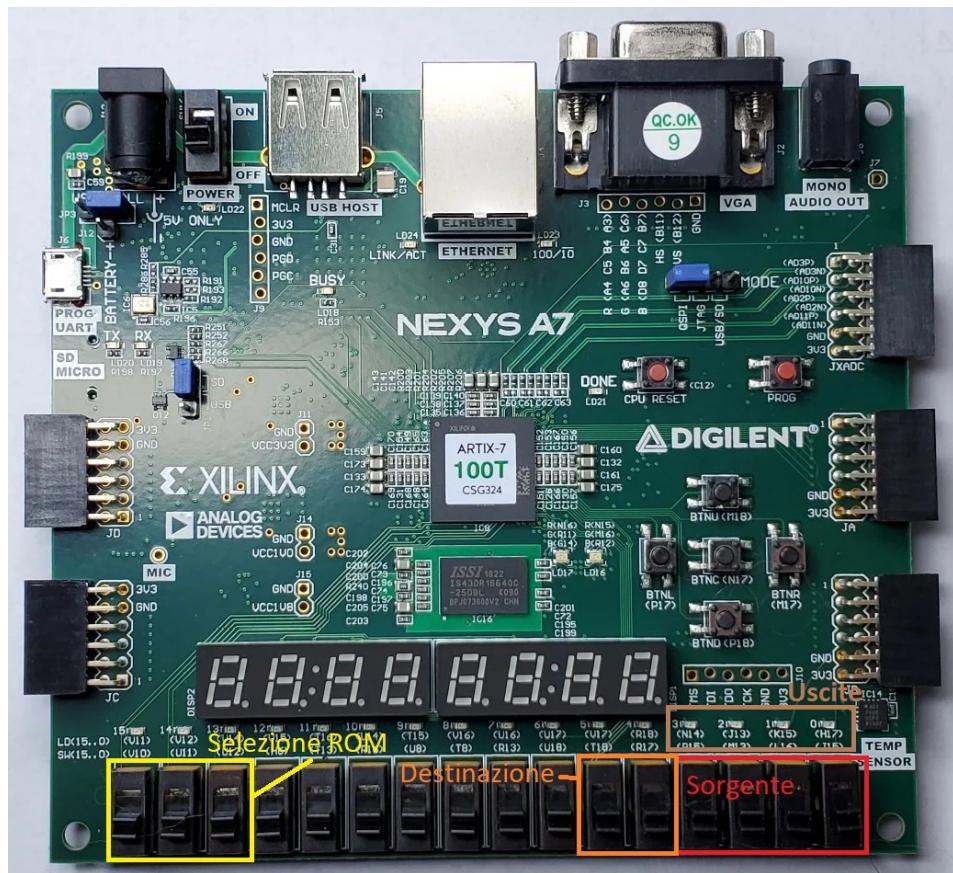


Figura 1.10: Mapping delle uscite sugli switch ed i led della scheda.

Progetto 2

Encoder BCD

Un codificatore (o encoder) è un circuito digitale combinatorio dotato di 2^n segnali di ingresso e di n segnali di uscita. Se viene attivata una delle linee di ingresso, in uscita viene prodotto il codice corrispondente. Un encoder che dispone di 10 ingressi e 4 uscite ($\lceil \log_2(10) \rceil$) BCD viene chiamato encoder Decimale-BCD. Un possibile utilizzo di questo circuito è per la realizzazione di tastierini numerici con 10 tasti corrispondenti ai numeri da 0 a 9: mediante l'encoder è possibile codificare la pressione di ogni tasto col codice binario corrispondente al numero del tasto premuto. Nel seguente svolgimento è stata anteposta al circuito codificatore una rete di priorità, in modo tale da realizzare complessivamente un codificatore a priorità BCD (priority encoder BCD): si tratta di un encoder che, in caso di attivazione di più di un ingresso, fornisce in uscita il numero binario corrispondente all'ingresso con priorità maggiore, cioè quello identificato dal numero decimale più alto.

2.1 Traccia

La traccia dell'esercizio è la seguente:

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).

Input: 0000000001 Output: 0000 (cifra 0)

Input: 0000000010 Output: 0001 (cifra 1)

Input: 0000000100 Output: 0010 (cifra 2)

....

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. Nel caso in cui si utilizzi una board dotata di soli 8 switch, è possibile sviluppare il progetto considerando X di soli 8 bit (la macchina sarà allora in grado di fornire in uscita la rappresentazione BCD delle cifre decimali da 0 a 7).

2.2 Descrizione soluzione teorica

Per risolvere l'esercizio è stato descritto con un livello di astrazione di tipo *data-flow* il componente di interesse, trattasi difatti di una rete combinatoria che ad un dato ingresso fa corrispondere una certa configurazione di uscita. Inoltre nella nostra soluzione è stato integrato un ulteriore componente utilizzando un approccio modulare, ovvero un arbitro tale da garantire che l'input fornito all'encoder presenti una codifica one-hot, presentando quindi un singolo bit alto.

2.3 Schematici, Componenti ed Implementazione in VHDL

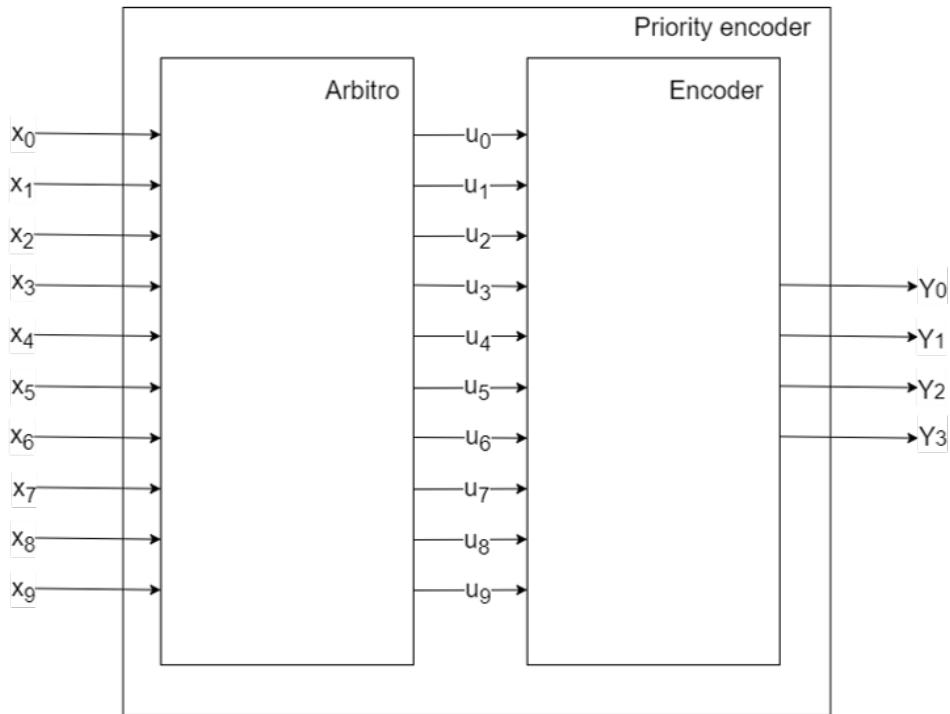


Figura 2.1: Schema Prioriy Encoder

Al primo stadio troviamo il componente Arbitro, una macchina combinatoria caratterizzata da 10 ingressi [X0...X9] e 10 uscite [U0...U9], delle quali sulla base degli ingressi una sola può essere pari ad 1. Il vettore in uscita dall'arbitro entra in ingresso al componente Encoder che si occupa di effettuare la codifica presentandola sul vettore di 4 bit in uscita [Y0...Y3].

2.3.1 Arbitro di priorità

Partiamo dall'implementazione dell'arbitro. Questo dispositivo presenta, come sottolineato in figura 2.1, dieci ingressi e dieci. La descrizione VHDL di questo componente è la seguente:

```
entity Arbiter is
  Port(
    X : in STD_LOGIC_VECTOR(9 downto 0);
    Y : out STD_LOGIC_VECTOR(9 downto 0)
  );
end Arbiter;

architecture Dataflow of Arbiter is
begin
  Y <= "1000000000" when X(9) = '1' else
    "0100000000" when X(8) = '1' else
    "0010000000" when X(7) = '1' else
    "0001000000" when X(6) = '1' else
    "0000100000" when X(5) = '1' else
    "0000010000" when X(4) = '1' else
    "0000001000" when X(3) = '1' else
    "0000000100" when X(2) = '1' else
    "0000000010" when X(1) = '1' else
    "0000000001" when X(0) = '1' else
    "-----";
end Dataflow;
```

Si tratta di una descrizione data-flow del componente. Viene posto nel vettore di uscita Y un singolo bit alto sulla base del valore decimale di peso maggiore posto in ingresso. Per fare ciò è stato utilizzato il costrutto *when* che dunque realizza un'assegnazione condizionata. Notiamo, inoltre, la presenza di un caso aggiuntivo di default, che copre gli altri possibili valori assumibili da X() secondo quanto stabilito dal tipo std_logic.

2.3.2 Encoder

Anche l'Encoder presenta nella definizione della sua interfaccia un vettore di 10 ingressi X[0...9] e un vettore di 4 uscite Y[0...3].

```
entity Encoder is
  Port(
    X : in STD_LOGIC_VECTOR(9 downto 0);
    Y : out STD_LOGIC_VECTOR(3 downto 0)
  );
end Encoder;

architecture Dataflow of Encoder is
begin
  with X select
    Y <= "0000" when "0000000001",
      "0001" when "0000000010",
      "0010" when "0000000100",
      "0011" when "0000001000",
      "0100" when "0000010000",
      "0101" when "0000100000",
      "0110" when "0001000000",
      "0111" when "0010000000",
      "1000" when "0100000000",
      "1001" when "1000000000",
      "----" when others;
end Dataflow;
```

Anche in questo caso è stata effettuata una descrizione a livello data-flow del componente sfruttando il costrutti **when** e codificando opportunamente le configurazione del vettore di ingresso sulle 4 cifre binarie di uscita.

2.3.3 PriorityEncoder

In maniera strutturale possiamo definire a questo punto il top-module, ovvero il PriorityEncoder. Partiamo dalla definizione dell'interfaccia dotata in questo caso di 10 ingressi e 4 uscite.

```
entity PriorityEncoder is
    Port (
        X : in STD_LOGIC_VECTOR(9 downto 0);
        Y : out STD_LOGIC_VECTOR(3 downto 0)
    );
end PriorityEncoder;
```

Per la descrizione strutturale del componente abbiamo mappato poi il vettore di ingresso sugli ingressi dell'arbitro e le uscite di quest'ultimo con gli ingressi dell'encoder. Per il collegamento dei due sottosistemi è stato utilizzato un segnale interno, denominato *u*.

```
signal u : STD_LOGIC_VECTOR(9 downto 0);
begin
    A : Arbiter
        PORT MAP (
            X => X, Y => u
        );
    E : Encoder
        PORT MAP (
            X => u, Y => Y
        );
end Structural;
```

2.4 Testing simulato

Si riporta di seguito il *testing* effettuato per l'encoder BCD. Il comportamento, come è possibile verificare dall'immagine 2.2, è quello atteso. In particolare è possibile constatare il corretto funzionamento della rete complessiva ottenuta dall'interconnessione dei componenti arbitro ed encoder: difatti in uscita (per ogni configurazione d'ingresso) è presente la codifica del valore decimale più grande, corrispondente al bit più prioritario alto. Non si tratta di una forma di testing esaustiva, ma con un certo livello di confidenza possiamo affermare che la descrizione VHDL fornisce, in ambiente simulato simulato, un comportamento coerente con le specifiche.



Figura 2.2: Testbench Encoder.

2.5 Caricamento sulla scheda

Effettuata la simulazione procediamo alla sintesi del progetto sulla board. Come indicato dalla traccia utilizziamo come ingressi gli switch, in particolare i primi 10

a partire da sinistra, ed infine il valore d'uscita viene mostrato tramite 4 dei led presenti (evidenziati in figura 2.3).

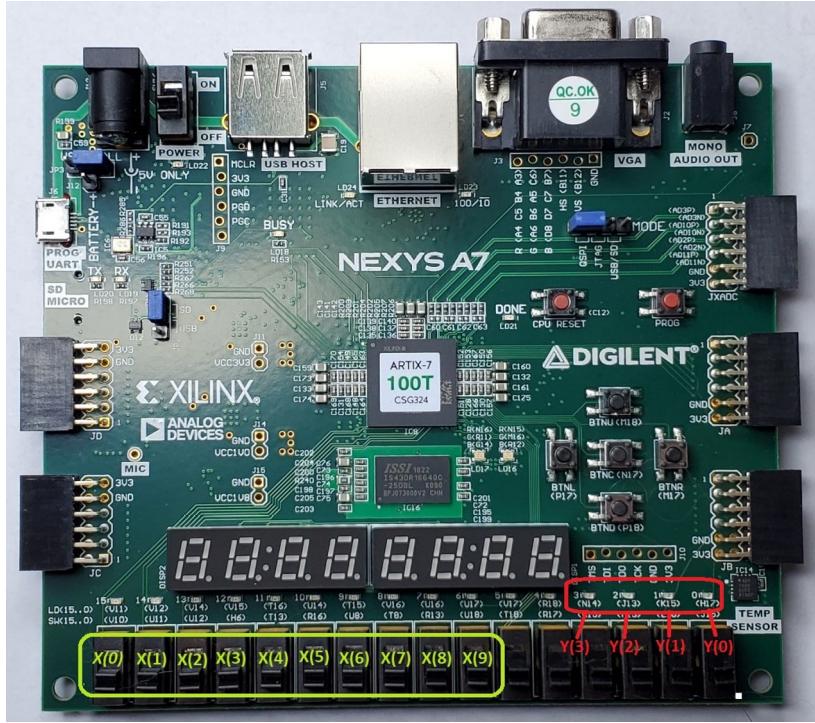


Figura 2.3: Corrispondenza ingressi/uscite

2.6 Display a 7 segmenti

L'esercizio ci richiede poi di utilizzare per l'uscita non più i leds ma il visore presente sulla board di sviluppo.

Sulla board è possibile notare la presenza di 2 display a 4 cifre (digits) a sette segmenti, i quali unitamente realizzano un unico display ad 8 cifre. Il pilotaggio di questo dispositivo però non è semplice come quella di altri componenti sulla scheda, non è infatti possibile, istante per istante, mostrare cifre distinte sui display. La gestione del display viene realizzata grazie ad un opportuno sistema di refreshing dei catodi che consente la visualizzazione, da parte dell'occhio umano, delle singole cifre distintamente. Difatti si sfrutta il limite umano nella percezione della frequenza dei cambiamenti. Se il refreshing avviene ad una frequenza

sufficientemente elevata, l'accensione selettiva delle singole cifre è percepita come simultanea.

2.6.1 Visualizzatore a 7 segmenti

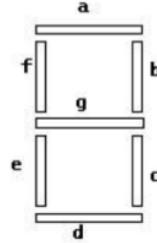


Figura 2.4: 7-segment display

I vari segmenti possono essere illuminati individualmente in modo da formare qualsiasi configurazione desiderata per la singola cifra. Gli anodi dei 7 LED (AN0,...,AN7, figura 2.5) che formano una cifra sono collegati insieme in un unico nodo circuitale (il quale funge da "anodo comune"). Tuttavia i catodi (CA,...,DP, figura 2.5) restano separati e sono responsabili della visualizzazione di una particolare configurazione. Essi sono disponibili come input del display a 8-cifre. Chiaramente per illuminare un segmento occorre "alzare" l'anodo e abbassare il catodo.

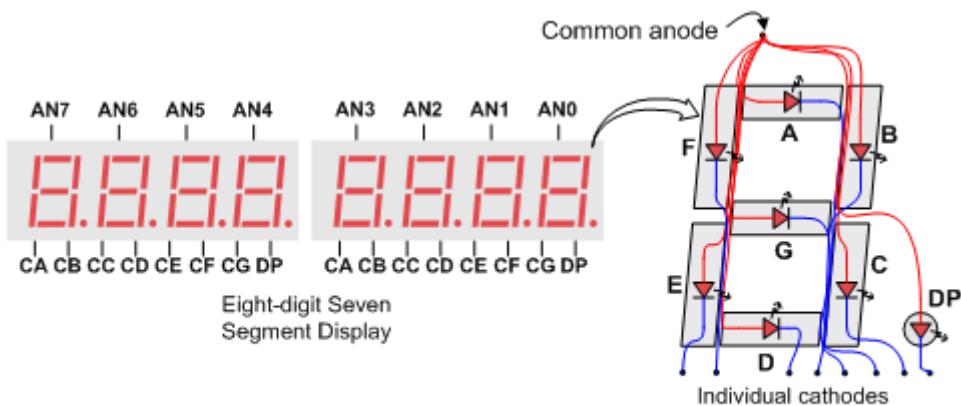


Figura 2.5: 7 segment display anodi e catodi

cathodes manager

```

entity display_seven_segments is
    Generic(
        CLKIN_freq : integer := 100000000;
        CLKOUT_freq : integer := 500
    );
    Port(
        CLK : in STD_LOGIC;
        RST : in STD_LOGIC;
        VALUE : in STD_LOGIC_VECTOR (31 downto 0);
        ENABLE : in STD_LOGIC_VECTOR (7 downto 0);
        DOTS : in STD_LOGIC_VECTOR (7 downto 0);
        ANODES : out STD_LOGIC_VECTOR (7 downto 0);
        CATHODES : out STD_LOGIC_VECTOR (7 downto 0));
    end display_seven_segments;

```

Gli anodi controllati tramite dei BJT pnp, dunque per alzare l'anodo occorre fornire un ingresso basso. Un circuito di controllo del display può essere utilizzato per mostrare un qualsiasi numero ad 8 cifre sul display. Tale circuito pilota i segnali di anodo e le corrispondenti configurazioni di catodi(di ogni singola cifra) periodicamente ad una frequenza di aggiornamento alta a tal punto che l'occhio umano non è in grado di rilevare lo scanning. Dunque, ogni cifra viene illuminata solo per $\frac{1}{8}$ del tempo di visualizzazione di tutte le 8 cifre, tuttavia, poichè l'occhio umano non può percepire lo spegnimento di una cifra prima che venga nuovamente illuminata, ogni cifra apparirà costantemente illuminata. Affinchè ciò avvenga, come indicato dai produttori della scheda, ogni cifra dovrebbe essere illuminata per un tempo compreso tra 1 e 16 millisecondi (ms), ottenendo dunque una frequenza di aggiornamento nel range 1Khz-60Hz. Il controllore dovrebbe abbassare i catodi con la corretta configurazione quando il corrispondente segnale di anodo viene "alzato".

Il dispositivo fornito ci dai docenti, che si occupa della gestione del display, è stato realizzato per composizione dei seguenti componenti:

- **counter**

```
entity counter_mod8 is
    Port ( clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        enable : in STD_LOGIC;
        counter : out STD_LOGIC_VECTOR (2 downto 0));
end counter_mod8;
```

Classico componente contatore, utilizzato per scorrere le singole cifre da illuminare, il segnale di enable viene fornito dal divisore di frequenza.

- **clock filter**

```
entity clock_filter is
    generic (
        CLKIN_freq : integer := 100000000;
        CLKOUT_freq : integer := 500
    );
    Port (
        clock_in : in STD_LOGIC;
        reset : in STD_LOGIC;
        clock_out : out STD_LOGIC
    );
end clock_filter;
```

Divisore di frequenza utilizzato per ottenere la frequenza desiderata di refreshing.

- **anodes manager**

```
entity anodes_manager is
    Port ( counter : in STD_LOGIC_VECTOR (2 downto 0);
```

```
enable_digit : in STD_LOGIC_VECTOR (7 downto 0);
anodes : out STD_LOGIC_VECTOR (7 downto 0)
);
end anodes_manager;
```

Effettua lo switching degli anodi per illuminare ad ogni colpo di counter una singola cifra.

- **cathodes manager**

```
entity cathodes_manager is
Port ( counter : in STD_LOGIC_VECTOR (2 downto 0);
value : in STD_LOGIC_VECTOR (31 downto 0);
dots : in STD_LOGIC_VECTOR (7 downto 0);
cathodes : out STD_LOGIC_VECTOR (7 downto 0));
end cathodes_manager;
```

Gestore dei catodi cui viene fornito il dato da mostrare sugli 8 display *value*, la configurazione dei punti da accendere *dots* e i 7 catodi più il punto *cathodes*. Viene definito in maniera comportamentale.

2.6.2 Priority Encoder BCD con display a 7 segmenti

Analizzato il comportamento del dispositivo di gestione del visore presente sulla scheda, ci siamo adoperati allo scopo di implementare l'ultimo punto della traccia:

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

Per il progetto complessivo è stata definita l'entity *Encoder_on_board* la cui architettura è stata descritta con approccio strutturale. In particolare, esso consta di due componenti: il *PriorityEncoder* da noi realizzato e il *display_seven_segment* fornитoci dai docenti durante il corso.

```

entity Encoder_on_board is Port(
    switches : in STD_LOGIC_VECTOR(9 downto 0);
    cathodes : out STD_LOGIC_VECTOR(7 downto 0);
    anodes : out STD_LOGIC_VECTOR(7 downto 0);
    clk : in STD_LOGIC
);
end Encoder_on_board;

architecture Structural of Encoder_on_board is

signal value_in : std_logic_vector(31 downto 0);
signal u : std_logic_vector(3 downto 0);

begin

value_in <= x"0000000" & u;

enc : entity work.PriorityEncoder port map(
    X => switches, Y => u
);

disp : entity work.display_seven_segments generic map(
    CLKIN_freq => 100000000, CLKOUT_freq => 500
) port map (
    clk => clk, RST => '0', value => value_in, enable => "00000001",
    dots => x"00", ANODES => ANODES, cathodes => cathodes
);

end Behavioral;

```

In particolare l'Encoder_on_board presenta come input i valori forniti tramite gli switch presenti sulla board (la cui mappatura è riportata nella figura 2.3) e

come uscita gli anodi e i catodi necessari al funzionamento del display. Vengono poi definiti due segnali interni: `value_in` e `u`. Il primo è un vettore necessario alla costruzione della sequenza di 32 bit rappresentativi delle cifre da visualizzare sul display, il secondo è un vettore sul quale viene mappata l'uscita del PriorityEncoder. Fondamentale è, inoltre, il mapping relativo al display in cui all'ingresso `value` viene assegnato il valore `value_in` precedentemente definito, mentre al vettore di `enable` viene assegnata una stringa binaria di cui un solo dei bit è alto in modo da far sì che solo uno degli 8 display risulti acceso. Ai `dots` viene assegnato zero dato che non sono di interesse per l'esercizio assegnatoci.

2.6.3 Timing analysis

Infine, abbiamo effettuato la timing analysis del sistema, valutandone i tempi di ritardo. Considerando che l'encoder BCD è una macchina combinatoria e, nel nostro caso, senza vincoli legati al tempo, abbiamo analizzato i tempi di commutazione per i percorsi senza vincoli.

Dalla figura 2.6, possiamo vedere come il caso peggiore in tempo di ritardo lo abbiamo nel path che va dall'ingresso `X(2)` all'uscita `Y(1)`. Chiaramente, questi sono parametri che dipendono da come è stato implementato il sistema in VHDL e da come il tool lo implementa, di conseguenza variano da implementazione ad implementazione.

Il Data Path Delay è stimato pari a 7.361ns e i livelli di logica previsti per il sistema sono 4. Inoltre, lo slack viene riportato ad un valore pari a "inf". Lo slack non è altro che la differenza di tempo tra l'arrivo previsto di un segnale e l'arrivo effettivo di un segnale. Fondamentalmente, un segnale dovrebbe raggiungere la sua destinazione prima del suo arrivo previsto. In questo caso, quindi, lo slack è pari ad un valore infinito in quanto non abbiamo requisiti di tempo, quindi il confronto sarà fatto su un requisito di tempo built-in del tool, che verosimilmente

PROGETTO 2. ENCODER BCD

è stato soddisfatto.

Max Delay Paths					
Slack:	inf	Path Group:	(none)	Path Type:	Max at Slow Process Corner
Source:	X[2]	Destination:	Y[1]	Data Path Delay:	7.361ns (logic 5.307ns (72.088%) route 2.055ns (27.912%))
Logic Levels:	4 (IBUF=1 LUT3=1 LUT6=1 OBUF=1)				
Location	Delay type	Incr(ns)	Path(ns)	Netlist	Resource(s)
U12	net (fo=0)	0.000	0.000	r X[2] (IN)	
U12	IBUF (Prop_ibuf_I_0)	1.523	1.523	r X_IBUF[2]_inst/0	
	net (fo=2, unplaced)	0.803	2.326	X_IBUF[2]	
	LUT6 (Prop_lut6_I3_0)	0.124	2.450	r Y_OBUF[1]_inst_i_2/0	
	net (fo=1, unplaced)	0.449	2.899	Y_OBUF[1]_inst_i_2_n_0	
	LUT3 (Prop_lut3_I1_0)	0.124	3.023	r Y_OBUF[1]_inst_i_1/0	
	net (fo=1, unplaced)	0.803	3.826	Y_OBUF[1]	
K15	OBUF (Propobuf_I_0)	3.535	7.361	r Y_OBUF[1]_inst/0	
	net (fo=0)	0.000	7.361	Y[1]	
K15			r Y[1] (OUT)		

Figura 2.6: Timing analysis del Priority Encoder BCD

Progetto 3

Riconoscitore di sequenze

Il terzo esercizio prevede la realizzazione di un riconoscitore di sequenza una notevole macchina sequenziale. Una macchina sequenziale è un sistema la cui uscita in un certo istante non dipende solo dal valore degli ingressi nello stesso istante, ma anche dagli ingressi precedenti. Viene quindi introdotto il concetto di stato, che può anche essere definito come memoria e che rappresenta le informazioni relative all'attività passata della macchina. In questo caso, il concetto di stato ci è utile in quanto chiaramente per riconoscere una sequenza di bit non basta conoscere l'ingresso in un istante, ma anche "ricordare" gli ingressi passati. Le scelte per quanto riguarda la modellazione della macchina sequenziale possono essere due: progettare una macchina di Mealy, in cui l'uscita è una funzione tanto dello stato corrente quanto dell'ingresso, o una macchina di Moore, in cui invece l'uscita è una funzione solo dello stato corrente e non c'è dipendenza dell'uscita rispetto all'ingresso corrente.

3.1 Traccia

La traccia dell'esercizio, suddivisa in due punti, è la seguente:

1. Progettare, implementare in VHDL e testare mediante simulazione una mac-

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

3.2 Descrizione soluzione teorica

Il comportamento del riconoscitore di sequenze è stato modellato, come tipicamente si procede nel caso della progettazione di macchine sequenziali, tramite un'automa a stati finiti (ASF).

L'automa a stati finiti del dispositivo progettato ha la particolarità del segnale di modo di funzionamento M . Esso è un ingresso aggiuntivo all'automa che fa cambiare il funzionamento della macchina, il cui comportamento può essere quindi modellato con 2 automi ben distinti e, nel caso in cui l'utente decidesse di far variare il modo, allora si passerà da un automa all'altro, passando allo stato base di uno dei due automi, $S0$ nel caso di $M=0$ e $S7$ nel caso di $M=1$.

PROGETTO 3. RICONOSCITORE DI SEQUENZE

Il comportamento di questa macchina sequenziale, specificato dalla traccia, ci permette di capire a fondo il funzionamento dei riconoscitori di sequenze. Infatti, quando $M=0$, il sistema deve riconoscere la sequenza a gruppi di 4 bit, il che implica un aumento considerevole del numero degli stati dell'automa in quanto ad ognuno dei 4 bit di un determinato gruppo (tranne l'ultimo, che porta sempre allo stato iniziale S_7) devono corrispondere due stati, uno che rappresenta la possibilità di sequenza corretta entro la fine del gruppo e uno che invece ne rappresenta la impossibilità, cioè che sicuramente il sistema non riconoscerà la sequenza target entro quel gruppo. Quando $M=1$, invece, l'automa è più semplice in quanto il riconoscimento dei bit avviene uno alla volta, senza il vincolo di dover valutare entro 4 bit la presenza della sequenza target **1001**. Infatti, se per esempio all'inizio ho una lunga sequenza di 0 in ingresso, in questo caso il nostro automa rimane fermo allo stato iniziale, invece nel primo caso mi sarei spostato di stato in stato per indicare che non verrà mai riconosciuto **1001** nella sequenza di 0 in ingresso.

Un aspetto altrettanto rilevante di questo progetto è la presenza del segnale di *clock* in ingresso al sistema. Infatti, il riconoscitore, essendo, come detto in precedenza, una macchina sequenziale, ha bisogno di un opportuno segnale di temporificazione, che permette alla macchina di evolvere da uno stato all'altro sul fronte di salita del segnale. Il comportamento della macchina sarà quindi **sincrono** col clock, che equivale ad una determinata scelta, come vedremo, nell'implementazione in VHDL.

Nel nostro progetto abbiamo deciso di implementare una macchina di Mealy in modo tale che venga visualizzata l'uscita, oltre all'ingresso, per ogni transizione di stato dell'automa. Questa scelta ci porta ad una semplificazione in termini di numero di stati presenti (già considerevole con Mealy) in quanto se avessimo implementato una macchina di Moore avremmo dovuto modellare due nuovi stati in cui l'uscita del riconoscitore era alta (uno per $M=0$ e uno per $M=1$).

Infine, per poter correttamente caricare su board il dispositivo progettato, c'è

PROGETTO 3. RICONOSCITORE DI SEQUENZE

da considerare il problema del **button bouncing** legato ai due bottoni utilizzati per acquisire l'input in sincronismo con il segnale di temporizzazione. Il button bouncing costituisce un problema quando il segnale associato al bottone viene utilizzato per far evolvere lo stato del sistema, come in questo caso, in quanto, essendo i bottoni delle abilitazioni a leggere l'input, questi sono fondamentali per l'evoluzione della macchina nel tempo.

Il problema alla base è che, in corrispondenza dell'appressione del pulsante, viene generata un'oscillazione che deve essere gestita per evitare comportamenti anomali da parte della macchina.

Introduciamo quindi il button debouncer, un semplice dispositivo il cui comportamento può essere modellato tramite un automa a stati finiti con solo due stati, NOT_PRESSED e PRESSSED, due ingressi, l'ingresso di clock (temporizzazione) e il bottone, e un'unica uscita, ovvero il segnale btn_cleared.

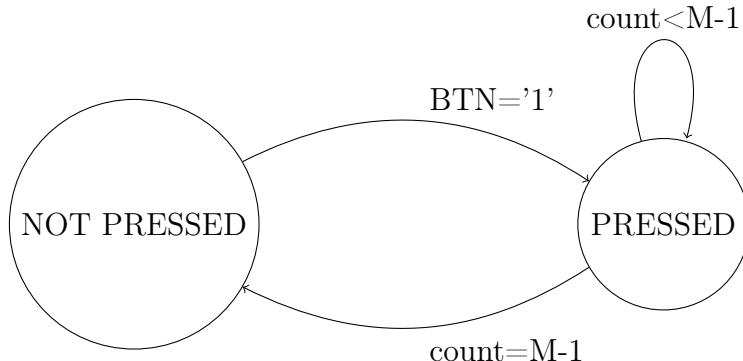


Figura 3.1: Automa a stati finiti del Button Debouncer

Il dispositivo rimane nello stato di idle NOT_PRESSED fin quando non viene premuto il bottone; appena viene premuto, passa allo stato di PRESSSED e la macchina permane per un certo numero M di colpi di clock; all' M -esimo colpo di clock viene alzato il segnale di output e la macchina torna allo stato NOT_PRESSED, dove viene azzerato count e abbassato il segnale di output. Di conseguenza, il segnale di output btn_cleared durerà per un solo colpo di clock, eliminando tutte le oscillazioni che disturbavano il comportamento della macchina. Si può dire che

il debouncer del bottone lavora come filtro poiché elimina le dinamiche oscillatorie alle alte frequenze indesiderate.

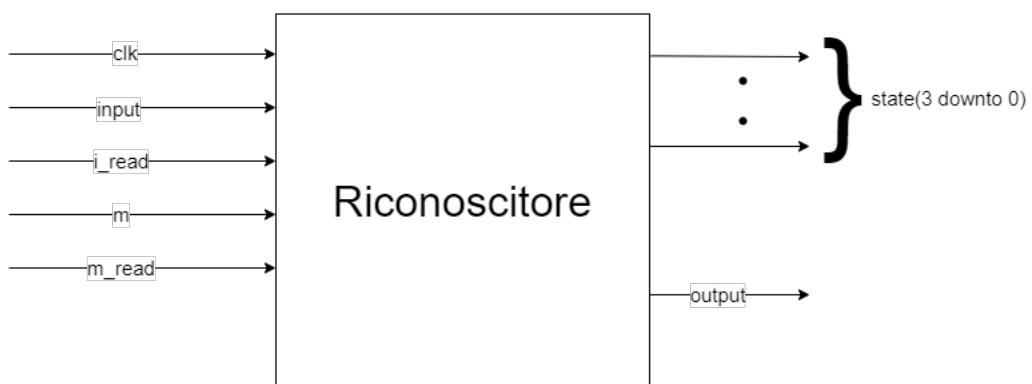
Per il debouncer abbiamo adoperato il componente fornito dai docenti, andando a modificare il valore del parametro M . Questo viene calcolato come il rapporto tra l'intervallo di tempo in cui si prevede l'oscillazione del bottone Δt , che in questo progetto assumiamo pari a 650000000 ns, e il periodo del clock, ovvero 10 ns. Abbiamo deciso di adoperare un intervallo di tempo Δt molto superiore a quello strettamente necessario per l'estinzione del transitorio allo scopo di evitare che una pressione prolungata del bottone venisse erroneamente letta come una sequenza di più ingressi e dunque causasse un'evoluzione indesiderata del sistema

3.3 Schematici

Il progetto si basa sui seguenti componenti fondamentali:

- Riconoscitore di sequenze;
- Button debouncer;

3.3.1 Riconoscitore



Il riconoscitore prende in ingresso un segnale di clock di temporizzazione, un segnale input, un segnale m di modo e i due corrispettivi segnali di abilitazione a

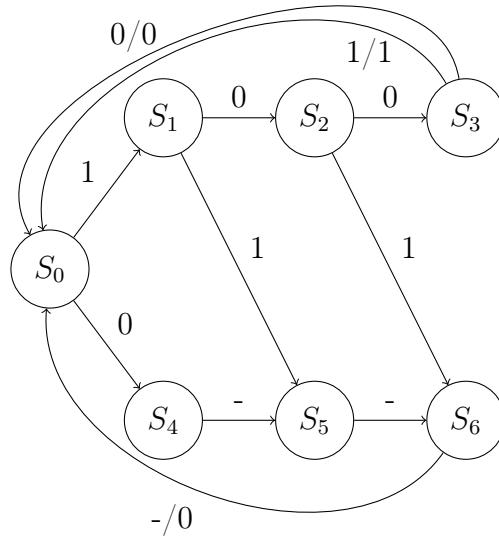
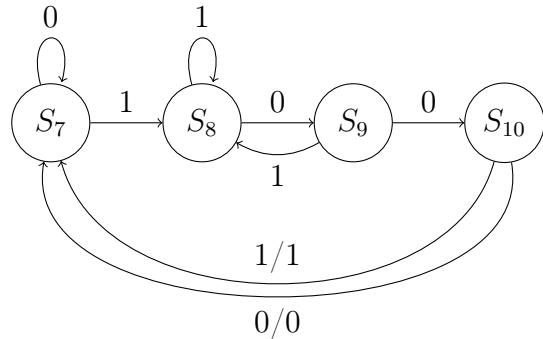
leggere rispettivamente l'input ed m, ovvero i_read e m_read. In uscita il componente fornisce un segnale che impostato ad uno 1 quando viene riconosciuta correttamente la sequenza 1001. Abbiamo inoltre previsto un'uscita addizionale, un vettore di 4 elementi che indica la codifica dello stato corrente, che seppur non strettamente necessaria, consente di verificare lo stato in cui si trova la macchina istante per istante. Il funzionamento del riconoscitore è modellato tramite un'automa a 11 stati, presentato di seguito come composizione di due automi, uno che modella il comportamento della macchina quando $m=0$ e l'altro quando $m=1$. Chiaramente, si sottintende per difficoltà di rappresentazione il fatto che i due automi sono in realtà collegati, da un qualsiasi stato è infatti possibile, cambiando il valore di m, passare allo stato iniziale dell'altro (S_0 o S_7). E' da notare il fatto che in linea di principio si sarebbero potute prendere altre scelte per la modellazione del comportamento del dispositivo. Infatti, si sarebbe potuto creare anche un solo stato base in comune ai due modi di funzionamento e solo successivamente specializzare i due diversi automi a seconda del valore di m. Chiaramente, in questo caso dopo il primo stato base, m sarebbe stato fisso e l'utente avrebbe potuto cambiarlo solo dopo il ritorno del dispositivo nello stato base.

In figura 3.2 è descritta la sequenza di stati che il riconoscitore attraversa quando $M=0$, cioè quando la macchina valuta i bit seriali in ingresso a gruppi di 4 bit. Si noti la presenza del simbolo di *don't care* '-'.

Invece, in figura 3.3, è presentato l'automa a stati finiti quando $M=1$, ovvero quando il sistema valuta i bit seriali in ingresso uno alla volta.

3.3.2 Button debouncer

Il button debouncer ha come ingressi il segnale di clock, un segnale di reset e il segnale btn, che diventa '1' quando il tasto corrispondente sulla scheda viene premuto. In uscita presenta il segnale cleared_btn, che dura quanto un colpo di


 Figura 3.2: Automa a stati finiti del riconoscitore di sequenze quando $m=0$

 Figura 3.3: Automa a stati finiti del riconoscitore di sequenze quando $m=1$

clock e risulta ripulito da tutte le oscillazioni transitorie del segnale btn. In questo progetto, il button debouncer ci servirà per entrambi i bottoni, sia `i_read` per l'abilitazione a leggere l'input, che `m_read` per l'abilitazione a leggere il segnale `m` di modo. Il suo funzionamento, con relativa descrizione dell'automa a stati finiti, è stato presentato nel paragrafo 3.2.

Si può quindi intuire che nel top module i due debouncer istanziati per i due bottoni verranno anteposti al riconoscitore, in modo tale che quest'ultimo abbia in ingresso i segnali di abilitazione puliti dal rumore che potrebbe causare

malfunzionamenti o anomalie.



3.4 Implementazione in VHDL

Abbiamo scelto di modellare il funzionamento del riconoscitore con un livello di astrazione comportamentale.

```

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity Riconoscitore is
    port(
        clk : in std_logic;
        i : in std_logic;
        i_read : in std_logic;
        o : out std_logic;
        m : in std_logic;
        m_read : in std_logic;
        state : out std_logic_vector(3 downto 0)
    );
end Riconoscitore;

```

Innanzitutto, abbiamo definito un tipo enumerativo, chiamato *stato*, con tutti i nomi degli stati da S_0 a S_{10} , e un segnale q di tipo stato inizializzato a S_0 .

PROGETTO 3. RICONOSCITORE DI SEQUENZE

Il processo *update* ha nella sensitivity list, che contiene gli identificativi dei segnali a cui il processo "reagisce", il segnale di clock. Quest'ultimo è fondamentale per il processo, dato che se ci troviamo sul suo fronte di salita, allora la macchina può evolvere e riconoscere la sequenza. Vengono quindi utilizzati dei costrutti IF...THEN dapprima per verificare di essere sul fronte di salita del clock, poi successivamente se viene alzato il segnale di abilitazione a leggere m (*m_read*) o il segnale di abilitazione a leggere l'input (*i_read*). In particolare, se *m_read* diventa uguale a '1', a seconda del valore di *m*, al segnale definito precedentemente *q* viene assegnato S_0 o S_7 ; invece, se *i_read* diventa uguale a '1', abbiamo usato un costrutto CASE...WHEN per modellare l'evoluzione degli stati, con il segnale *q* che indica lo stato corrente e che viene modificato coerentemente a com'è stato descritto l'automa a stati finiti nel paragrafo 3.3. Intuitivamente, l'uscita *o* rimane sempre '0' per tutti i casi tranne quando ci troviamo in S_3 e *i* è uguale a '1' e quando ci troviamo in S_{10} e *i* è uguale a '1'.

Oltre al processo, nell'architecture del riconoscitore abbiamo inserito anche una assegnazione selezionata riguardante il segnale *q*. Infatti, essa ci permette di assegnare un determinato valore (in esadecimale) al vettore state di 4 bit, a seconda del valore assunto dal segnale *q*. In questo modo, possiamo controllare il valore dello stato corrente ad ogni colpo di clock, per poi visualizzarlo su 4 LED della scheda. Essendo all'interno del architecture sia il processo che l'assegnazione selezionata, questi sono eseguiti in modo concorrente e quindi il vettore state viene aggiornato ogni volta che il segnale *q* è modificato.

```
architecture Behavioral of Riconoscitore is
    type stato is (S0,S1,S2,S3,S4,S5,S6,S7,S8,S9,S10);
    signal q : stato := S0;
begin
    update : process(clk) begin
        if(rising_edge(clk)) then
```

```

if(m_read='1') then
    if(m = '1' and
        (q = S0 or q = S1 or q = S2 or q = S3
        or q = S4 or q = S5 or q = S6)) then
            q <= S7;
            o <= '0';
    elsif(m = '0' and
        (q = S7 or q = S8 or q = S9 or q = S10)) then
            q <= S0;
            o <= '0';
    end if;

elsif(i_read = '1') then
    case q is
        when S0 =>
            if(i = '0') then
                q <= S4;
                o <= '0';
            elsif(i = '1') then
                q <= S1;
                o <= '0';
            end if;
        when S1 =>
            if(i = '0') then
                q <= S2;
                o <= '0';
            elsif(i = '1') then
                q <= S5;
                o <= '0';
            end if;
        when S2 =>
            if(i = '0') then

```

PROGETTO 3. RICONOSCITORE DI SEQUENZE

```
q <= S3;
o <= '0';

elsif(i = '1') then
    q <= S6;
    o <= '0';

end if;

when S3 =>
    if(i = '0') then
        q <= S0;
        o <= '0';
    elsif(i = '1') then
        q <= S0;
        o <= '1';
    end if;

when S4 =>
    q <= S5;
    o <= '0';

when S5 =>
    q <= S6;
    o <= '0';

when S6 =>
    q <= S0;
    o <= '0';

when S7 =>
    if(i = '0') then
        q <= S7;
        o <= '0';
    elsif(i = '1') then
        q <= S8;
        o <= '0';
    end if;

when S8 =>
    if(i = '0') then
```

PROGETTO 3. RICONOSCITORE DI SEQUENZE

```
q <= S9;
o <= '0';

elsif(i = '1') then
    q <= S8;
    o <= '0';

end if;

when S9 =>
    if(i = '0') then
        q <= S10;
        o <= '0';

    elsif(i = '1') then
        q <= S8;
        o <= '0';

    end if;

when S10 =>
    if(i = '0') then
        q <= S7;
        o <= '0';

    elsif(i = '1') then
        q <= S7;
        o <= '1';

    end if;

end case;

end if;

end if;

end process;

with q select
    state <= x"0" when S0,
    x"1" when S1,
    x"2" when S2,
    x"3" when S3,
    x"4" when S4,
```

PROGETTO 3. RICONOSCITORE DI SEQUENZE

```
        x"5" when S5,
        x"6" when S6,
        x"7" when S7,
        x"8" when S8,
        x"9" when S9,
        x"a" when S10,
        x"f" when others;
end Behavioral;
```

Il debouncer del bottone specifica nell'entity 2 generics, permettendo di parametrizzarne la descrizione: CLK_PERIOD è il periodo del clock, di tipo intero, inizializzato a 10; btn_noise_time è l'intervallo di tempo Δt del bottone, sempre di tipo intero, inizializzato a 6500000.

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

entity ButtonDebouncer is
    generic(
        CLK_PERIOD : integer := 10;
        btn_noise_time: integer :=6500000);
    port(
        clk : in std_logic;
        reset : in std_logic;
        btn : in std_logic;
        cleared_btn : out std_logic
    );
end ButtonDebouncer;
```

L'architecture del debouncer del bottone si basa sull'automa a stati finiti presentato nel paragrafo 3.2. Viene definito un tipo enumerativo stato, un segnale

PROGETTO 3. RICONOSCITORE DI SEQUENZE

btn_state di tipo stato, che rappresenta lo stato corrente della macchina, inizializzato a NOT_PRESSED e una costante M, di tipo intero, di valore pari al rapporto tra btn_noise_time e CLK_PERIOD.

Il processo deb, con il segnale di clock nella sensitivity list, nella parte dichiarativa specifica la variable count, di tipo intero, inizializzata a 0. Nel process body viene usato un costrutto IF...THEN per verificare che ci si trovi sul fronte di salita del clock. Se ciò avviene, innanzitutto si controlla se il reset è uguale a '1', il che porterebbe la macchina nello stato di NOT_PRESSED. Altrimenti, con un costrutto CASE...WHEN si implementa l'automa a stati finiti presentato nei paragrafi precedenti. Se ci troviamo nello stato NOT_PRESSED, l'uscita CLEARED_BTN deve essere bassa; invece, se la macchina è nello stato PRESSED e la variable count è uguale a M-1, l'uscita diventa alta e count viene riazzerrata; infine, se siamo sempre nello stato PRESSED ma count non è uguale a M-1, viene semplicemente incrementata la variabile, rimanenendo sempre nello stato PRESSED.

```
architecture Behavioral of ButtonDebouncer is
type stato is (NOT_PRESSED, PRESSED);
SIGNAL btn_state : stato := NOT_PRESSED;

constant M : integer := btn_noise_time/CLK_PERIOD;
begin
deb: process(CLK)
variable count: integer :=0;

begin
if rising_edge(CLK) then
  if(reset='1') then
    btn_state<=NOT_PRESSED;
    CLEARED_BTN <='0';
  else
    count:=count+1;
    if(count=M) then
      btn_state<=PRESSED;
      CLEARED_BTN <='1';
    else
      btn_state<=NOT_PRESSED;
    end if;
  end if;
end if;
end process;
end;
```

```

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 =M-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;

```

3.4.1 Top module

Il componente top module è, come detto in precedenza, formato da due button debouncer e il riconoscitore, descritto a livello comportamentale. Di conseguenza, la presentazione del top module sarà ad un livello di astrazione strutturale, con cui si può descrivere il funzionamento del sistema nella sua interezza, ovvero il suo legame fra i pin di ingresso e di uscita.

Nella parte dichiarativa dell'architecture vengono quindi definiti i tre componenti, che verranno usati per il nostro sistema, e due segnali interni cleared_i e cleared_m di tipo STD_LOGIC.

PROGETTO 3. RICONOSCITORE DI SEQUENZE

Nel corpo dell'architecture vengono istanziati i debouncer *debi* e *debm* per i due tasti, in particolare il reset viene associato a '0' in quanto nel nostro progetto non verrà utilizzato, e viene istanziato il riconoscitore con label *asf*.

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

entity Riconoscitore_on_board is
    port(
        clk : in std_logic;
        i : in std_logic;
        i_read : in std_logic;
        o : out std_logic;
        m : in std_logic;
        m_read : in std_logic;
        state : out std_logic_vector(3 downto 0)
    );
end Riconoscitore_on_board;

architecture Behavioral of Riconoscitore_on_board is

signal cleared_i : std_logic;
signal cleared_m : std_logic;

begin

    debi : entity work.ButtonDebouncer generic map(
        CLK_period => 10,      btn_noise_time => 650000000
    ) port map (
        reset => '0', clk => clk, BTN => i_read, cleared_btn => cleared_i
    );

    debm : entity work.ButtonDebouncer generic map(
        CLK_period => 10,      btn_noise_time => 650000000
    )
```

```

        ) port map (
            reset => '0', clk => clk, BTN => m_read , cleared_btn => cleared_m
        );

        asf : entity work.Riconoscitore port map(
            clk => clk, i => i, m => m, state => state, o => o,
            i_read => cleared_i, m_read => cleared_m
        );
    end Behavioral;

```

3.5 Testing simulato

Successivamente, abbiamo simulato il riconoscitore tramite un apposito test bench. Nella parte dichiarativa dell'architecture del file di test bench abbiamo definito il componente riconoscitore, che richiamiamo nel process body per l'istanziazione dell'*unit under testing* attraverso il costrutto PORT MAP.

Innanzitutto, abbiamo creato un process *tb_clock* che simula un andamento periodico del clock, come un'onda quadra di periodo 100 ns. Quindi abbiamo creato un altro process, chiamato *tb_inputs*, in cui simuliamo il comportamento del riconoscitore, fornendo una sequenza di ingressi *i*, preceduti chiaramente dal segnale di abilitazione *i_read*. Inoltre, abbiamo testato anche il cambiamento di modo, mettendo uguale a '1' prima l'abilitazione a leggere *m*, *m_read*, e poi *m* stesso. In figura 3.4, abbiamo testato il comportamento della macchina, quando *m*=0, questa arrivi a S_3 e ritorni a S_0 con output uguale a '1', il che significa che è stata riconosciuta correttamente la sequenza. Invece, in figura 3.5, abbiamo simulato il comportamento della macchina sequenziale quando il segnale di modo è uguale a '1': in questo caso, valutando singolarmente ogni bit, all'istante $t=850$ ns, l'output diventa uguale a '1' e lo stato stato proprio in quell'istante transita da S_{10} a S_7 .

PROGETTO 3. RICONOSCITORE DI SEQUENZE

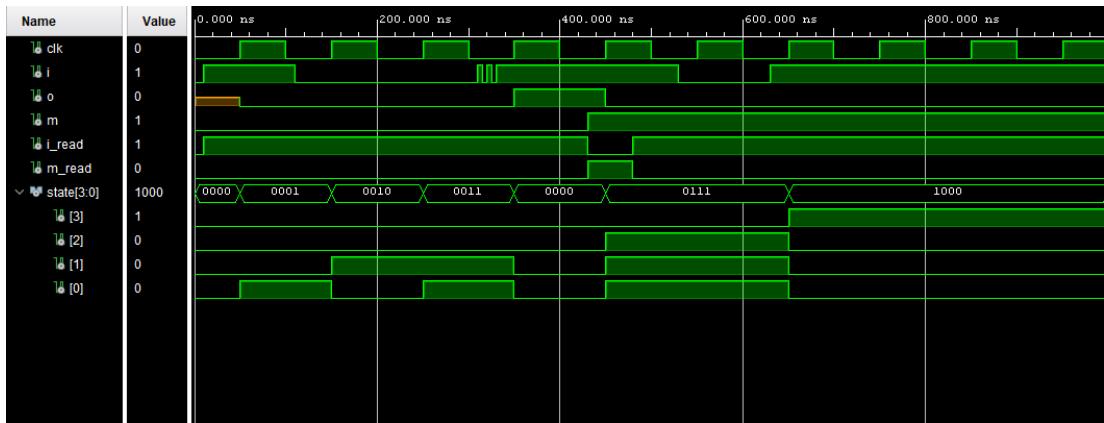


Figura 3.4: Simulazione del funzionamento del riconoscitore quando $m=0$

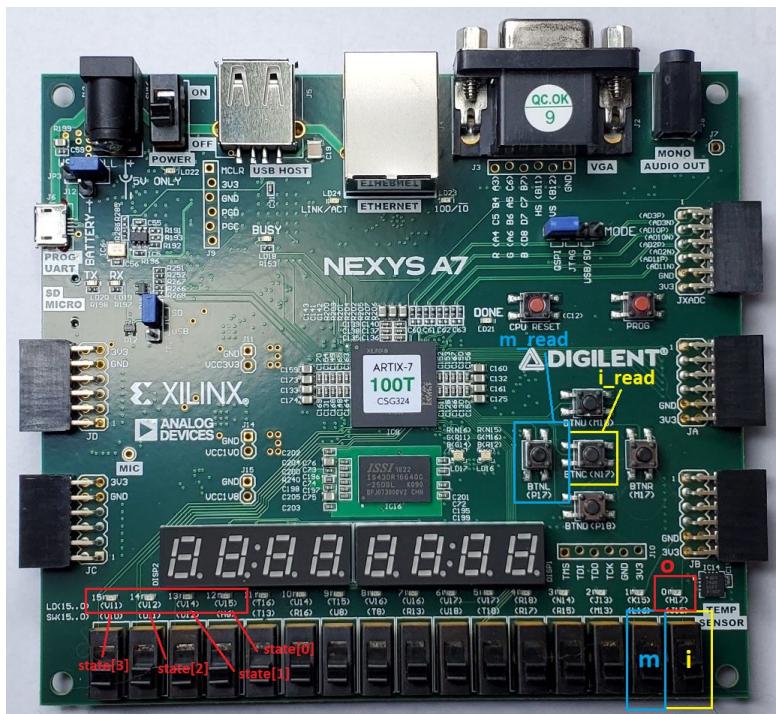


Figura 3.5: Simulazione del funzionamento del riconoscitore quando $m=1$

3.6 Caricamento sulla scheda

Infine, abbiamo sintetizzato e implementato su board il riconoscitore di sequenze. Gli ingressi i ed m sono stati assegnati a due switch, in particolare quelli in basso a destra. L'uscita o è stata collegata al led in basso a destra, mentre il vettore state in uscita al riconoscitore, che rappresenta lo stato corrente, è stato assegnato ai primi quattro led partendo da sinistra. In conclusione, per i due bottoni, abbiamo scelto il tasto N17, quello centrale, per l'ingresso i_read e il tasto P17, alla sua sinistra, per l'ingresso m_read.

PROGETTO 3. RICONOSCITORE DI SEQUENZE



Progetto 4

Shift register

Come quarto esercizio abbiamo realizzato un componente elementare fondamentale, lo shift register. Il funzionamento di base di questo dispositivo è semplice: in corrispondenza di un segnale di abilitazione i dati memorizzati al suo interno traslano di una posizione. Nella sua semplicità questo dispositivo è però alla base di molti dispositivi più complessi: può essere infatti impiegato all'interno di macchine aritmetiche per implementare la moltiplicazione o la divisione per una potenza di due, come contatore con codifica one-hot o per la generazione di segnali di controllo periodici nella sua versione circolare ed è inoltre alla base di tutte le comunicazioni seriali, in cui è necessario trasmettere un dato, inizialmente memorizzato in parallelo, un bit alla volta.

4.1 Traccia

Il testo della traccia di questo progetto è il seguente:

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 a) approccio comportamentale sia b)

approccio strutturale. E' quindi richiesto di realizzare una variante del componente classico che presenti un comportamento più particolare, consentendo di traslare i dati in esso memorizzati sia verso destra che verso sinistra di un numero variabile di posizioni.

La traccia richiede inoltre di descrivere il componente sia ad alto livello, mediante una implementazione comportamentale, che a basso livello, con un approccio strutturale.

4.2 Descrizione soluzione teorica

Per questo progetto abbiamo deciso di realizzare il componente con modalità circolare, consentendo, inoltre, il caricamento e la lettura parallela dei dati memorizzati.

Per quanto riguarda l'approccio comportamentale il problema risulta nel complesso di semplice risoluzione: siamo partiti dalle equazioni che regolano il componente per poi andarle ad implementare in un semplice *script* contenuto all'interno di un *process*. Il comportamento della macchina è in questo caso determinato da alcuni ingressi di modo che consentono di scegliere la direzione e l'ampiezza dello shift e che consentono, in corrispondenza di un ingresso, di precaricare un valore nel registro. Il componente è stato, inoltre, realizzato con un comportamento sincrono e quindi risulta abilitato esclusivamente in corrispondenza dei fronti di salita del clock.

L'approccio strutturale è invece stato più complesso da progettare. Siamo partiti dall'implementazione del singolo flip-flop D andando poi a connettere tutti i flip flop utilizzati mediante una rete di interconnessione composta da multiplexer, consentendo di specificare, mediante gli appositi ingressi di selezione, le particolari interconnessioni e quindi la direzione e l'ampiezza dello shift. Abbiamo poi dovuto risolvere un ulteriore problema per il caricamento parallelo dei dati, introducen-

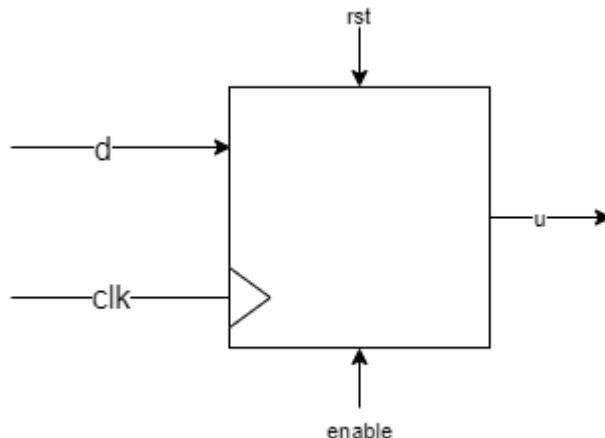
do un meccanismo addizionale di selezione descritto nel dettaglio al paragrafo successivo.

4.3 Approccio strutturale

4.3.1 Il flip-flop D

Il primo passo per la realizzazione strutturale di uno shift register è senza dubbio quella di andare a realizzare la cella elementare di memoria, il flip-flop D, che, nel nostro caso, è stato realizzato come sensibile sul fronte di salita del clock. Questo dispositivo è caratterizzato, oltre che dal segnale di abilitazione *enable*, dal clock e dal reset, che permette di azzerare l'uscita del flip-flop. In figura 4.1 è possibile osservare lo schematico del componente.

Figura 4.1: Schematico del flip-flop D



La descrizione VHDL dell'interfaccia di questo componente è quindi la seguente:

```
entity ff_d is
  port (
    clk : in std_logic;
    rst : in std_logic;
```

```
    enable : in std_logic;
    d : in std_logic;
    u : out std_logic
);
end ff_d;
```

E analogamente, la presenta la seguente descrizione comportamentale:

```
architecture Behavioral of ff_d is
signal q : std_logic;
begin
    u <= q;
    process(clk) begin
        if(rising_edge(clk) and enable = '1') then
            if(rst = '1') then
                q <= '0';
            else
                q <= d;
            end if;
        end if;
    end process;
end Behavioral;
```

4.3.2 Top module

Successivamente, abbiamo definito il top module dello shift register, ad un livello di astrazione chiaramente strutturale. Il registro a scorrimento è composto da tre tipi di componenti:

- Flip-flop D, spiegato nel paragrafo precedente;
- Multiplexer 2:1;
- Multiplexer 4:1.

L'entity dello shift register è stato implementato mediante un generic e 7 porte. Il generic, N, specificato con il valore di default pari a 16, indica il numero di bit salvati nel registro. Inoltre, lo shift register presenta in ingresso un segnale di clock, un segnale di reset, uno di enable e uno di parallel_load per abilitare la lettura in parallelo di un dato in ingresso formato da N bit, rappresentato dal vettore parallel_input. Infine, il sistema ha come ingresso anche un vettore *shifts* di 2 bit, che rappresenta le diverse modalità di shift implementate dal nostro registro a scorrimento. Infatti, esistono 4 diversi possibili modi per shiftare gli elementi di questo registro:

- quando *shifts*="00", i dati scorrono di una posizione da destra verso sinistra;
- quando *shifts*="01", i dati scorrono di due posizioni da destra verso sinistra;
- quando *shifts*="10", i dati scorrono di una posizione da sinistra verso destra;
- quando *shifts*="11", i dati scorrono di due posizioni da sinistra verso destra.

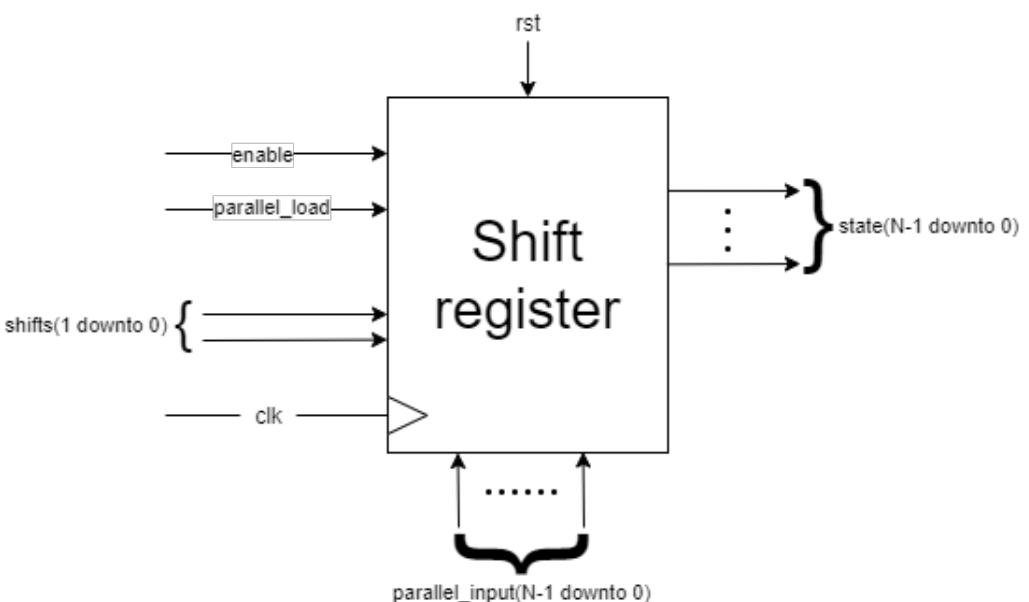


Figura 4.2: Schematico rappresentante gli input e gli output dello shift register

La macchina avrà come uscita un vettore *state* di N elementi, che rappresenta lo stato del registro e dei suoi N bit dato dopo aver eseguito la traslazione. In figura

4.2, è mostrato uno schematico rappresentante gli input e gli output del registro. L'unica particolarità di questo registro a scorrimento che abbiamo progettato è la sua circolarità: infatti, quando si tratta di shiftare i bit nelle posizioni intermedie, questi verranno spostati a destra o a sinistra di una o due posizioni in base al valore del vettore shifts; invece, per i bit che si trovano agli estremi del registro, ci si pone il problema di come gestirli ed eventualmente cosa inserire al loro posto quando viene effettuata la traslazione. Abbiamo quindi deciso di implementare una soluzione circolare, che, attraverso l'operatore di modulo, ci permette di traslare i dati nelle ultime posizioni alle prime in caso di shift verso destra e viceversa in caso di shift verso sinistra. L'implementazione in VHDL dell'entity è la seguente:

```
entity variable_shift_register is
    generic(
        N : integer := 16
    );
    port(
        clk : in std_logic;
        rst : in std_logic;
        enable : in std_logic;
        shifts : in std_logic_vector(1 downto 0);
        parallel_load : in std_logic;
        parallel_input : in std_logic_vector(N-1 downto 0);

        state : out std_logic_vector(N-1 downto 0)
    );
end variable_shift_register;
```

Come detto in precedenza, nella parte dichiarativa dell'architecture abbiamo definito le interfacce dei tre componenti che ci serviranno per costruire lo shift register. Inoltre, abbiamo definito anche tre vettori di N bit, *muxtoff*, *fftomux* e *muxtomux*, che rappresentano rispettivamente i segnali interni tra i multiplexer

2:1 e i flip flop, quelli tra i flip flop e i multiplexer 4:1 e, infine, quelli tra i multiplexer 4:1 e flip flop. Attraverso il costrutto FOR...GENERATE, nel corpo dell'architettura vengono istanziati N flip-flop D, N multiplexer 2:1 ed altrettanti multiplexer 4:1. In figura 4.3, è presentato uno schematico che descrive come sono collegati tra loro i vari componenti dello shift register, considerando N pari a 4.

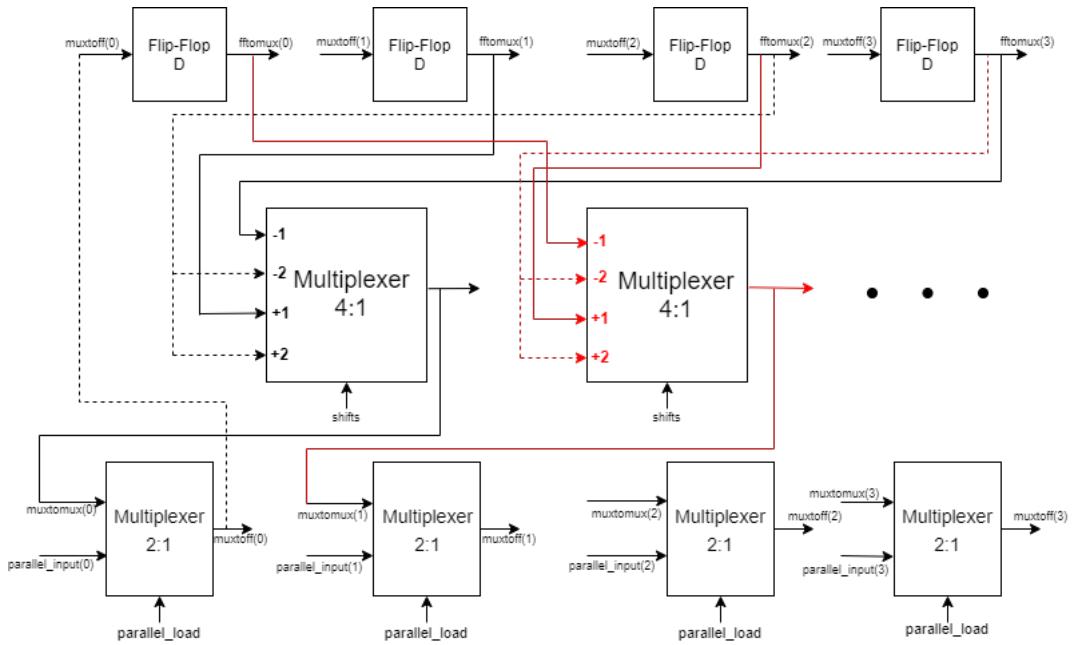


Figura 4.3: Descrizione illustrativa della struttura dello shift register al livello di astrazione strutturale. Per ragioni di spazio sono stati omessi il terzo e il quarto multiplexer 4:1

I multiplexer 4:1 permettono di effettuare effettivamente lo shift dei bit dato. Infatti, avendo come segnale di selezione il vettore di ingresso *shifts*, in base a questo segnale (e quindi in base all'ampiezza e alla direzione della traslazione) si assegna al flip-flop i-esimo, attraverso il vettore *muxtomux*, il nuovo dato, che verrà preso da uno dei flip-flop adiacenti con il segnale *fftomux*. Si noti che in questa istanziazione viene implementata la circolarità del registro descritta precedentemente, attraverso l'operatore **mod**.

I multiplexer 2:1, invece, hanno come segnale di selezione l'ingresso *parallel_load*. Quindi, se questo segnale è pari a '0', verrà selezionata l'uscita del multiplexer 4:1 *muxtomux* (che effettua lo shift dei dati) e verrà mandata in in-

gresso al flip-flop i-esimo corrispondente mediante il segnale *muxtoff*. Al contrario, se *parallel_load* è uguale a '1', il multiplexer prende in ingresso *parallel_input*, inserito dall'utente, e lo manda al flip-flop corrispondente sempre tramite *muxtoff*.

Infine, i flip-flop D servono chiaramente a memorizzare il bit di informazione salvato in quella i-esima posizione del registro a scorrimento. Essi prendono in ingresso l'uscita del multiplexer 2:1 *muxtoff* e hanno come uscita *fftomux*, che è indirizzata al multiplexer 4:1 in modo che quest'ultimo effettui lo shift. E' da notare, inoltre, che l'uscita *fftomux* è assegnata anche al segnale di uscita dello shift register *state*.

L'implementazione in VHDL dell'architecture è presentata di seguito:

```
architecture Structural of variable_shift_register is

signal muxtoff : std_logic_vector(N-1 downto 0);
signal fftomux : std_logic_vector(N-1 downto 0);
signal muxtomux : std_logic_vector(N-1 downto 0);

begin

    state <= fftomux;

    FFs : for i in 0 to N-1 generate
        FF : entity work.ff_d port map(
            clk => clk, rst => rst, enable => enable, d=> muxtoff(i),
            u => fftomux(i)
        );
    end generate;

    MUXs : entity work.mux2_1 port map(
        S => parallel_load, X(0) => muxtomux(i), X(1) => parallel_input(i),
        Y => muxtoff(i)
    );
    MUX : entity work.mux4_1 port map(
        S => shifts, Y => muxtomux(i), X(0) => fftomux((i-1) mod N),
        X(1) => fftomux((i-2) mod N),
        X(2) => fftomux((i-3) mod N),
        X(3) => fftomux((i-4) mod N)
    );
end;
```

```
X(1) => fftomux((i-2) mod N), X(2) => fftomux((i+1) mod N),  
X(3) => fftomux((i+2) mod N));  
end generate;  
end Structural;
```

4.4 Approccio comportamentale

Diversamente dal livello di astrazione strutturale, lo shift register che abbiamo progettato in maniera comportamentale permette di effettuare uno shift fino ad un numero di posizioni pari a 15 (in entrambi nelle direzioni). Il miglioramento drastico delle prestazioni è dovuto al fatto che, al livello di astrazione strutturale, per permettere uno shift di 2 posizioni abbiamo dovuto costruire N multiplexer 4:1, ergo per eseguire uno shift di 16 posizioni saremmo dovuti arrivare a progettare N multiplexer 32:1, rendendo lo schema circuitale del progetto sensibilmente più complesso. La maggiore efficienza viene pagata però col rischio di rendere la macchina non più sintetizzabile su FPGA, in quanto il suo comportamento è stato definito attraverso un process di natura altamente algoritmica.

L'entity dello shift register è molto simile a quella definita a livello strutturale, sia per quanto riguarda i generic che i ports. Le uniche differenze sono, come specificato, nel segnale di ingresso *shifts*, che sarà formato da 4 bit, e da un nuovo segnale di ingresso *lr*, che, quando pari a '0', indicherà che lo shift sarà fatto verso sinistra, altrimenti verso destra.

```
entity shift_register is  
generic(  
    N : integer := 16  
);  
port(  
    clk : in std_logic;
```

```
enable : in std_logic;  
rst : in std_logic;  
parallel_in : in std_logic_vector(N-1 downto 0);  
parallel_load : in std_logic;  
shifts : in std_logic_vector(3 downto 0);  
lr : in std_logic;  
state : out std_logic_vector(N-1 downto 0)  
);  
end shift_register;
```

L'architecture del registro a scorrimento definito a livello comportamentale presenta un process sensibile sul fronte di salita del clock. Se il segnale di reset *rst* è pari a '1', chiaramente tutti i bit del registro vengono settati a '0'; altrimenti, se *parallel_load*='1', allora i bit vengono settati in base al valore del segnale *parallel_in*.

Infine, viene gestito lo shift verso sinistra se *lr* è pari a '0' e lo shift verso destra se *lr* è pari a '1'. In particolare, nel caso di shift verso sinistra, i "primi" bit fino a N-1-shifts vengono traslati di *shifts* posizioni, considerando *shifts* in questo caso come numero intero, mentre in realtà alla macchina viene fornito come vettore di quattro bit. Invece, gli "ultimi" bit del registro, che vanno dalla posizione N-shifts a N-1, per la circolarità dello shift, non potendo essere traslati verso sinistra, vengono traslati nelle prime posizioni, lasciate libere dallo shift precedente. Un analogo ragionamento può essere fatto per lo shift di destra, considerando che in questo caso i "primi" bit, ovvero i bit meno significativi, dovranno spostarsi nelle ultime posizioni del registro.

```
architecture Behavioral of shift_register is  
  
signal q : std_logic_vector(N-1 downto 0);  
  
begin
```

```
state <= q;

process (clk) begin

    if(rising_edge(clk) and enable = '1') then

        if(rst = '1') then

            q <= (others => '0');

        elsif(parallel_load = '1') then

            q <= parallel_in;

        elsif(lr = '0') then

            q(N-1 downto conv_integer(shifts)) <= q(N-1-
conv_integer(shifts) downto 0);

            q(conv_integer(shifts) - 1 downto 0) <= q(N-1
downto N - conv_integer(shifts));

        elsif(lr = '1') then

            q(N-1-conv_integer(shifts) downto 0) <= q(N-1
downto conv_integer(shifts));

            q(N-1 downto N-conv_integer(shifts)) <=
q(conv_integer(shifts)-1 downto 0);

        end if;

    end if;

end process;

end Behavioral;
```

4.5 Testing simulato

Il sistema è stato verificato successivamente mediante simulazione. In figura 4.4 è mostrata una parte della simulazione dello shift register progettato con l'approccio strutturale. Considerando N pari a 8, abbiamo innanzitutto dato in ingresso, tramite il segnale di abilitazione parallel_load e il vettore di ingresso parallel_input, il valore "A0" in esadecimale. Impostando il valore di shifts pari a "00", che equivale ad uno shift verso sinistra di una posizione, possiamo notare che al fronte di salita successivo del clock il valore nel registro shifta a "41", sempre in esade-

cimale, verificando quindi il corretto funzionamento del sistema, sia per quanto riguarda la direzione che l'ampiezza della traslazione. Quindi, vengono eseguiti altri shift sempre di una posizione verso sinistra ad ogni fronte di salita del clock, per poi far variare il valore di shifts a "10", che equivale ad una traslazione sempre di una posizione ma nella direzione opposta, verso destra.

In figura 4.5, invece, si può notare una parte della simulazione dello shift register progettato con approccio comportamentale. Il funzionamento, come descritto precedentemente, è molto simile: in questo caso, a N è stato assegnato il valore di 16 e all'inizio al registro viene dato in ingresso il valore "0002" in esadecimale tramite il vettore parallel_input. Si noti che in questo caso per testare lo shift verso sinistra abbiamo messo lr uguale a '0' e il vettore *shifts* pari a "0001".

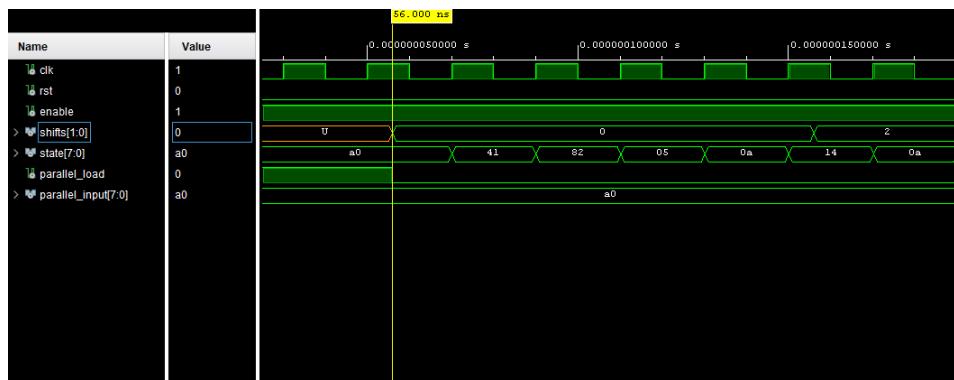


Figura 4.4: Testbench dello shift register definito a livello strutturale

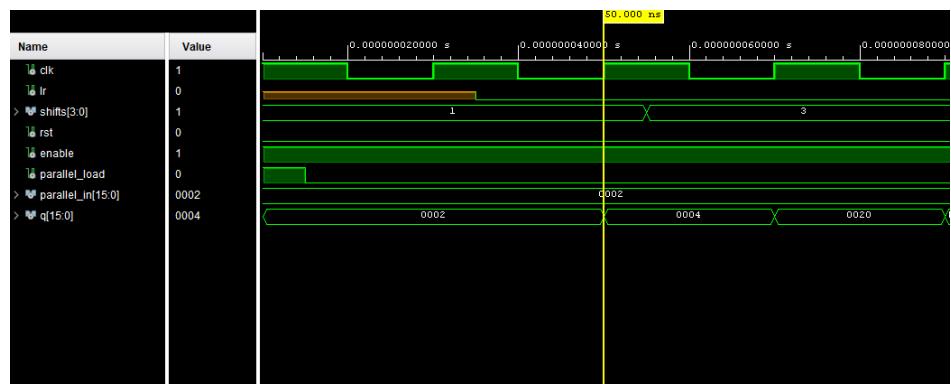


Figura 4.5: Simulazione dello shift register definito a livello comportamentale

4.6 Caricamento sulla scheda

Infine, abbiamo deciso di sintetizzare il sistema dello shift register progettato in maniera strutturale su FPGA. Inizialmente abbiamo creato un nuovo top module, chiamato ShiftRegisterOnBoard, formato dallo shift register descritto nell'approccio strutturale e da un button debouncer, che ci servirà per risolvere il problema delle oscillazioni associate alla pressione al tasto di enable.

Per ragioni di spazio abbiamo scelto N pari a 8. L'ingresso parallel_input, un vettore di 8 elementi, è stato assegnato ai primi 8 switch partendo da destra, con il segnale di parallel_load che è stato configurato sull'ultimo switch a sinistra e il vettore di due bit *shifts* sul terzo e il quarto switch da sinistra. Per quanto riguarda i tasti, abbiamo impostato il segnale di reset sul bottone in alto e il segnale di enable su quello in basso. Infine, gli 8 bit salvati nel registro a scorrimento saranno visualizzati sui primi 8 LED da sinistra, che rappresentano gli elementi del vettore di uscita state del sistema.

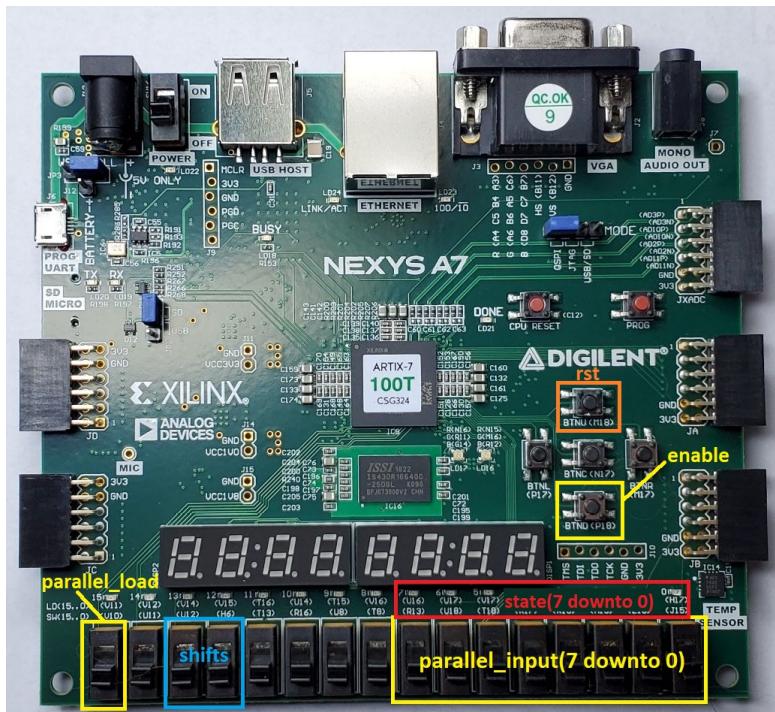


Figura 4.6: Schema illustrativo del mapping dei segnali

Progetto 5

Cronometro

Il cronometro è un sistema digitale il cui scopo è scandire il tempo sulla base di secondi, minuti e ore. Il suo utilizzo nella quotidianità è fortemente sottovalutato: qualsiasi sistema, anche non digitale, può aver bisogno del cronometro sia per misurare il tempo trascorso da una determinato evento sia per visualizzare l'ora attuale.

5.1 Traccia

La traccia è suddivisa nei seguenti tre punti:

- 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.

- 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).
- 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).

5.2 Descrizione soluzione teorica

L'approccio con il quale abbiamo deciso di progettare il cronometro è il tipico metodo di progettazione nell'ambito dei sistemi digitali: abbiamo realizzato un'unità di controllo (UC), che è la parte del sistema che ne definisce il comportamento, attraverso un'automa a stati finiti, e una parte operativa (UO), costituita dall'insieme dei componenti architettonici che implementano praticamente le operazioni richieste (flip-flop, memorie, multiplexer, ...), a cui sono impartiti opportuni comandi, detti segnali di controllo, dalla UC.

L'unità di controllo presenta in ingresso gli input che possono essere forniti dall'utente, in questo caso sono i tasti che possono essere premuti e che permettono di passare da uno stato all'altro. Consideriamo, quindi, le diverse modalità di funzionamento del nostro sistema:

- funzione di conteggio, attivata dalla pressione di un tasto, che permetterà all’utente di visualizzare sul display lo scorrere dei secondi, minuti e ore;
- funzione di orologio, attivata dal precaricamento tramite switch di un determinato valore di minuti e ore, a cui può essere combinata la funzione di conteggio in modo da far scorrere il tempo a partire da un certo orario;
- funzione di visualizzazione dei quattro intertempi memorizzati, attivata dalla pressione di un apposito tasto che permette all’utente di scorrere tra i vari intertempi (nel caso fossero meno di quattro, si visualizzerà una stringa nulla).

Inoltre, in questa UC, avremo anche un ingresso che terrà conto di quanti intertempi abbiamo salvato fino a quel momento.

Per quanto riguarda le uscite, esse saranno i cosiddetti segnali di controllo, che verranno impartiti in ingresso all’unità operativa. In generale, quindi, l’unità di controllo è stata progettata con un approccio comportamentale, così come si farebbe per ogni macchina sequenziale modellata con un automa a stati finiti.

La parte operativa, come detto, presenta in ingresso i segnali di controllo mandati dalla UC e due input utente che riguardano il pre-caricamento di un certo orario nel sistema. In relazione alle uscite, abbiamo quelle relative agli anodi e i catodi del display a 7 segmenti e una variabile contatore che tiene conto di quanti intertempi sono stati memorizzati.

A differenza della parte di controllo, l’unità operativa è stata progettata con un approccio strutturale. Quindi, sono stati definiti al suo interno tutti i componenti architetturali utili per permettere il funzionamento del sistema. I componenti, intuitivamente, saranno abilitati o disabilitati in base ai segnali di controllo inviati dalla UC.

Il componente fondamentale di tutto il sistema è il contatore modulo N. Esso, nel suo utilizzo più banale, sul fronte attivo del clock e col segnale di abilitazione

a contare alto, incrementa una variabile interna che viene poi restituita in output, fino a quando essa non raggiunge il valore N-1, in corrispondenza del quale viene azzerata, permettendo l'inizio di un nuovo ciclo di conteggio. E' da osservare, inoltre, che è stato aggiunto un segnale che diventa '1' ogni volta che la variabile interna raggiunge il valore N-1 e la possibilità di settare il valore della variabile interna in modo arbitrario, se il segnale di abilitazione a settare è attivo.

Il componente principale che compone l'architettura del cronometro è l'orologio centrale, un dispositivo composto per composizione di quattro diversi contatori. Abbiamo operato realizzando un componente fortemente sincrono, di conseguenza tutti e quattro i contatori sono sensibili allo stesso segnale di clock, fornito dalla scheda, ma incrementano il conteggio esclusivamente quando tutti i contatori precedenti hanno raggiunto l'ultimo valore ed hanno quindi il segnale di uscita alto. Il primo contatore opera sostanzialmente come base dei tempi, il suo compito fondamentale è quindi quello di operare come divisore di frequenza, restituendo, a partire dal segnale di clock della scheda a 100MHz un segnale di conteggio ad 1Hz. L'ingresso della base dei tempi è memorizzato all'interno di un flip-flop T, quando il valore memorizzato è alto la base dei tempi avanza generando il segnale di conteggio su cui tutto l'orologio si basa, quando è basso si arresta, fermando l'evoluzione dell'orologio. I contatori successivi sono due contatori modulo sessanta, rispettivamente il conteggio dei secondi e dei minuti, ed un contatore modulo ventiquattro, per il conteggio delle ore. Uno schema dettagliato dell'interconnessione dei contatori è riportato in figura 5.2.

Infine, per quanto riguarda la memorizzazione dei quattro intertempi, abbiamo creato una semplice memoria con 4 locazioni di memoria da 32 bit ciascuna in cui poter salvare gli intertempi. Per accedere a tale memoria c'è bisogno di un indirizzo, ottenuto come uscita di un multiplexer 2:1. Il multiplexer ha 2 ingressi che rappresentano rispettivamente il prossimo indirizzo dove salvare l'intertempo i -esimo e la prossima locazione da cui visualizzare l'intertempo memorizzato. In-

tuitivamente, il primo verrà selezionato quando si vorrà scrivere nella memoria e il secondo invece quando si vorrà leggere da essa.

5.3 Schematici

5.3.1 Contatore

Il contatore modulo N ha un ingresso *clk* di sincronizzazione, *rst*, che funziona da reset, *count*, che rappresenta il segnale di abilitazione a contare, e l'ingresso di abilitazione a precaricare un valore, chiamato *load*. Inoltre, ha un ingresso parallel_input, un vettore di $\lceil \log_2 N \rceil$ elementi, che rappresenta il valore da precaricare in binario. Le uscite sono *clk_out*, ovvero un segnale che diventa '1' quando il conteggio è arrivato a N, e *Y*, un vettore di $\lceil \log_2 N \rceil$ elementi, che rappresenta il conteggio vero e proprio.

Se siamo sul fronte di salita del clock e col segnale *count* pari a '1', il sistema incrementa l'uscita di 1 fino a quando non raggiunge il valore di N-1 e quindi viene riportata a 0. In aggiunta a ciò, se il segnale *rst* è pari a '1', il segnale viene azzerato, invece se *load* è attivo, l'uscita viene settata al valore di parallel_input.

5.3.2 Clock

Il clock (orologio) presenta, oltre all'ingresso di sincronizzazione *clk* e quello di reset *rst*, il segnale di abilitazione a contare *count* e uno di abilitazione a precaricare il dato *load*. Il dato da precaricare in ingresso è formato da 3 vettori:

- *sec_in*, un vettore di 6 bit, che rappresenta un numero da 0 a 60;
- *min_in*, un vettore di 6 bit per rappresentare un numero da 0 a 60;
- *hour_in*, di 5 bit per rappresentare un numero da 0 a 24.

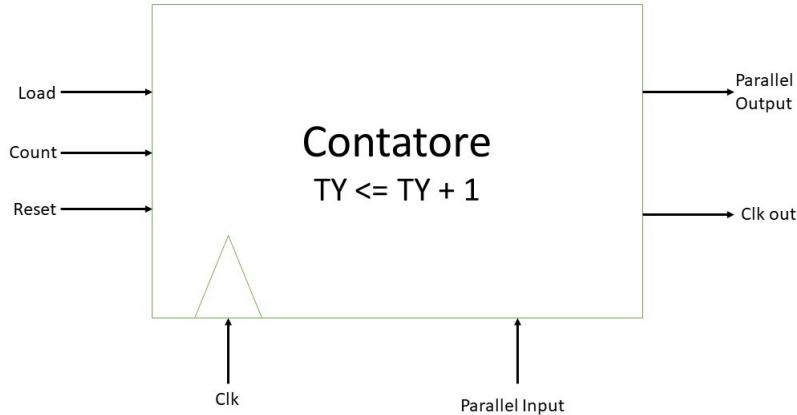


Figura 5.1: Schematico del componente contatore

Lo stesso discorso può essere applicato per l’uscita del clock, che sarà quindi formata da 3 vettori dello stesso tipo qui presentato.

L’approccio strutturale, già descritto nel paragrafo 5.2, con cui è stato progettato questo componente prevede la definizione di quattro contatori connessi in serie. Analizziamoli nel dettaglio:

1. la base dei tempi, con $N=100000000$, avrà la stessa abilitazione a contare del clock (di cui fa parte), che alla fine dei conti sarà l’uscita del flip-flop T collegato immediatamente prima di esso; inoltre, questo contatore non avrà il segnale di *load* e Y;
2. il primo contatore modulo 60, con il segnale di *clk_out* della base dei tempi come segnale di *count*; al segnale di *load* sarà assegnato lo stesso del clock e al segnale di *parallel_input* invece avremo *sec_in*;
3. il secondo contatore modulo 60, col segnale di *count* pari al risultato della AND tra il *clk_out* della base dei tempi e il primo contatore mod 60; per quanto riguarda *load* e *parallel_input* avremo rispettivamente il segnale di *load* del clock e *min_in*;

4. il contatore modulo 24, con il segnale di *count* pari al risultato della AND tra i *clk_out* dei tre contatori precedenti; per *load* e *parallel_input* avremo lo stesso ragionamento dei primi tre casi e quindi saranno assegnati rispettivamente al *load* del *clock* e ad *hour_in*.

Intuitivamente, le Y degli ultimi tre contatori presentati saranno le uscite del clock.

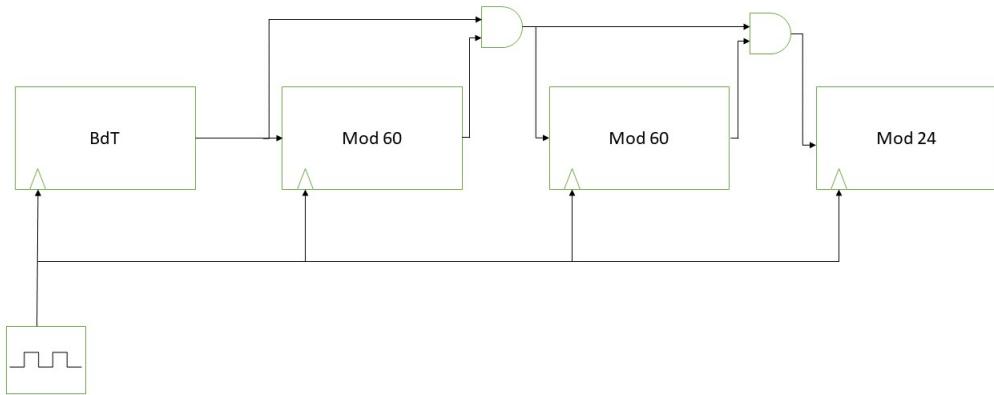


Figura 5.2: Schema illustrativo del clock del cronometro

5.3.3 BCD Converter

Il convertitore BCD prende in ingresso un vettore di 6 bit e ne restituisce uno da 8 bit. Semplicemente, il suo scopo è quello di trasformare il valore di secondi, minuti e ore (che è in binario) dapprima in decimale e successivamente trasformare le due cifre (unità e decine) separatamente in binario. Considerando che per ogni cifra ci possono essere al massimo 4 bit, per i valori di secondi, minuti e ore avremo bisogno di al massimo 8 bit per rappresentare le due cifre componenti. Ciò è stato fatto in quanto le cifre così trasformate saranno rappresentate sul display a 7 segmenti in una codifica BCD.

L'algoritmo che abbiamo implementato sfrutta il fatto che per ottenere la cifra delle unità di un intero basta fare l'operazione di modulo per 10 e che per ottenere la cifra delle decine basta dividere il numero intero per 10 ed eseguire di nuovo

l'operazione di modulo. Ottenute le due cifre, è bastato ri-trasformarle nel tipo `unsigned` e quindi assegnarle ai primi 4 bit dell'uscita per le unità e ai secondi 4 bit per le decine.

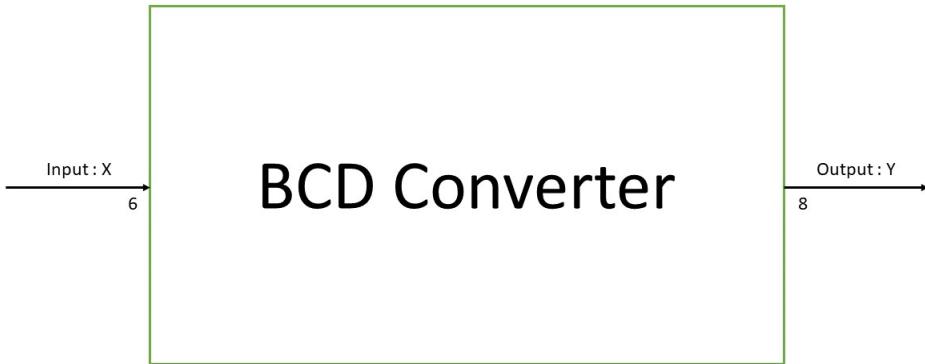


Figura 5.3: Schematico del BCD Converter

5.3.4 Memoria

La memoria nel nostro progetto è una memoria RWM e presenta due generics, `N` e `M`: il primo indica la grandezza di una singola cella di memoria, il secondo quante celle di memoria sono presenti. Oltre al segnale di ingresso di reset `rst` e al segnale di sincronizzazione `clk`, la macchina ha in ingresso un segnale di abilitazione a leggere `read` ed uno a scrivere `write`. Infatti, per accedere ad una locazione di memoria abbiamo in ingresso un segnale `address`, di $\lceil \log_2 M \rceil$ elementi, e un ingresso `input`, di `N` elementi, nel caso in cui volessimo accedere in modalità scrittura. Infine, se la memoria è in modalità lettura, abbiamo un'uscita `output` di `N` elementi.

Il funzionamento del sistema avviene sul fronte di salita del clock e per gestire le operazioni di lettura/scrittura è stato definito un tipo `mem`, ovvero un array di `M` elementi in cui ogni elemento è un vettore di `N` bit, e un segnale temporaneo `data` di tipo `mem`. Inoltre, è da osservare che quando il segnale `rst` è pari a '1', tutti gli `M` vettori del segnale `data` vengono settati a '0'.

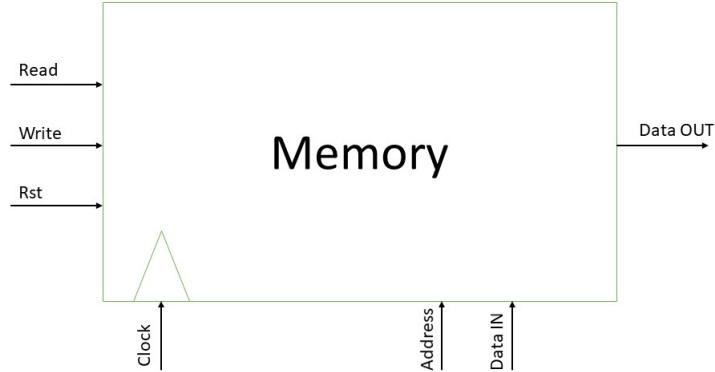


Figura 5.4: Schema rappresentativo della memoria

5.3.5 Flip Flop T

Il flip flop T ha tre ingressi e una uscita: T, uno di reset, uno di sincronizzazione e l'uscita q. Ha funzioni di memoria e di **toggle**, che consiste nella negazione del valore precedentemente memorizzato. La sua caratteristica, quindi, è che se $T=0$, l'uscita Q viene memorizzata e non subisce nessun cambiamento, mentre se $T=1$, Q viene negata, ottenendo in questo caso una frequenza dimezzata rispetto al clock.

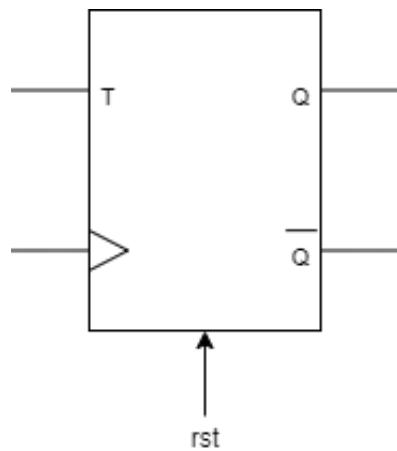


Figura 5.5: Schematico del flip flop T

5.3.6 Unità operativa

La parte operativa del cronometro, progettata con approccio strutturale, è quindi formata da:

- il clock;
- il display a 7 segmenti, con tutta la gestione di anodi e catodi ad esso collegati;
- un convertitore BCD per i secondi;
- un convertitore BCD per i minuti;
- un convertitore BCD per le ore;
- un flip flop T;
- una memoria;
- un contatore per il numero di intertempi memorizzati;
- un multiplexer 2:1 che seleziona l'uscita della memoria o l'orario in base ad un opportuno segnale di controllo;
- un multiplexer 2:1 che seleziona l'indirizzo della memoria nel caso si voglia scrivere un nuovo intertempo in essa oppure si voglia visualizzare i quattro intertempi

La struttura abbastanza complessa è dovuta alle molteplici modalità di funzionamento del nostro sistema. I multiplexer 2:1, infatti, hanno proprio questo scopo di selezionare una o l'altra modalità di funzionamento, assegnando un determinato ingresso alla memoria o al visore. Analizziamo, quindi, nel dettaglio i ports di ingresso e di uscita della UO della macchina:

- segnale di sincronizzazione clk ;

- ingressi di controllo: abilitazione a scrivere nella memoria *mem_write*, abilitazione a leggere dalla memoria *mem_read*, abilitazione a pre-caricare il dato nel clock *clock_load*, reset del clock *clock_rst*, abilitazione del contatore degli intertempi *incrementa_intertempi*, ingresso del flip flop T chiamato *counting*, segnale di selezione del multiplexer del visore *display_mode*, segnale di selezione del multiplexer in ingresso alla memoria *mem_mode* e l'indirizzo di uno dei quattro intertempi visualizzati dall'utente, chiamato *addr*, che è un vettore di 2 bit;
- ingressi utente: in questo caso abbiamo *min_in* per il pre-caricamento del valore dei minuti e *hour_in* per il valore delle ore nel clock, rispettivamente un vettore di 6 bit e un vettore di 5 bit;
- uscita di stato, ovvero l'uscita del contatore degli intertempi *n_intertempi*, un vettore di 2 bit;
- uscite del display, *anodes* e *cathodes*.

Come possiamo vedere anche dalla figura 5.6, le uscite dei secondi, minuti e ore del clock (rispettivamente *secEX*, *minEX*, *hourEX*) entrano direttamente in ingresso ai BCD converter, che le trasforma quindi in *secBCD*, *minBCD* e *hourBCD*.

Per quanto riguarda la memoria, assegniamo M=4 e N=32 in quanto ogni intertempo sarà salvato in una locazione di memoria da 32 bit (8 per i secondi, 8 per i minuti e 8 per le ore, più uno zero-padding di altri 8 bit). La memoria avrà, inoltre, come segnale di reset il segnale *clock_rst* e come output il segnale *mem_out*, un vettore chiaramente di 32 bit.

Per multiplexer 2:1 dedicato all'indirizzamento della memoria abbiamo assegnato il valore di N pari a 2; il primo ingresso sarà l'uscita del contatore degli intertempi *counter_intertempi*, la seconda invece il segnale di controllo *addr*. L'uscita, di conseguenza, sarà il segnale interno *mem_addr*, che è anche il segnale di address per la memoria.

Il multiplexer 2:1 anteposto al visore, invece, avrà $N=32$. I due ingressi saranno: l'uscita dei BCD converter zero-paddata fino a 32 bit e l'uscita della memoria *mem_out*. Il segnale di uscita *muxtovis* sarà poi lo stesso che sarà assegnato al segnale di ingresso del display.

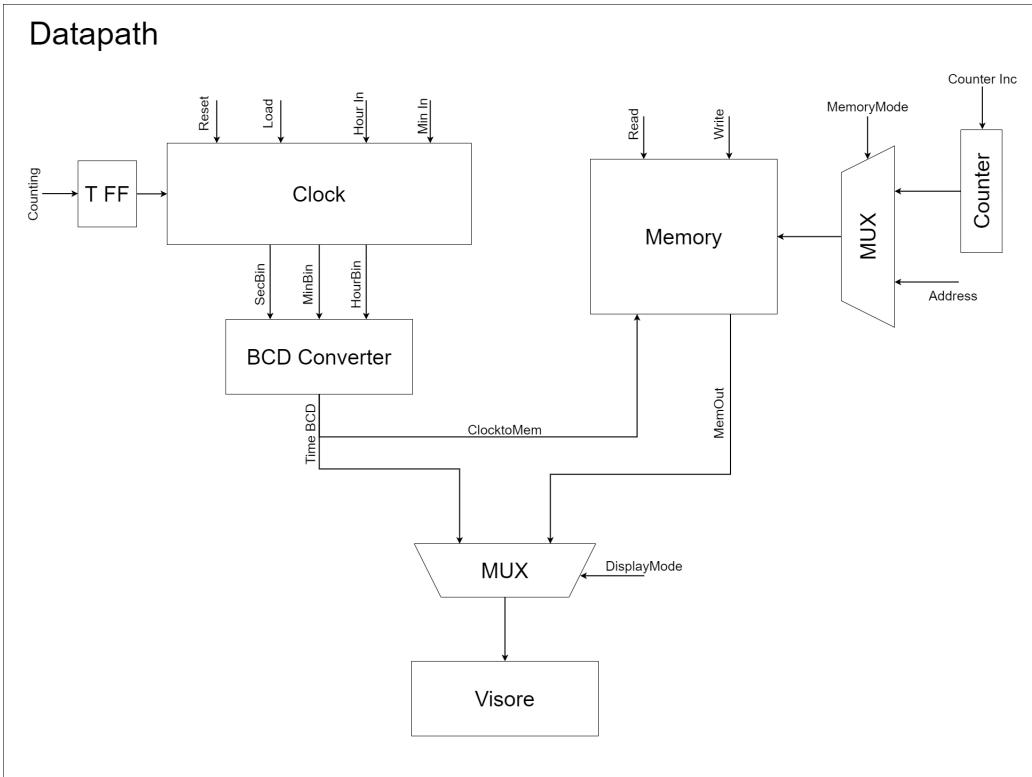


Figura 5.6: Schema rappresentativo della operating unit del sistema

5.3.7 Unità di controllo

L'unità di controllo del cronometro ha i seguenti ports:

- segnale di sincronizzazione *clk*;
- gli input utente, che permetteranno le transizioni da uno stato all'altro della macchina: *start_b* per iniziare il conteggio, *rst_b* per resettare il conteggio e la memorizzazione degli intertempi, *load_b* per pre-caricare un determinato orario nel cronometro, *mem_b* per memorizzare un intertempo ad un certo istante e *visual_b* per visualizzare i quattro intertempi;

- ingresso di stato, che nella UO era l'uscita di stato, ovvero l'uscita del contatore degli intertempi;
- i segnali di controllo, che sono esattamente gli stessi definiti per la UO con l'unica differenza che ora sono dei segnali di uscita e non di ingresso.

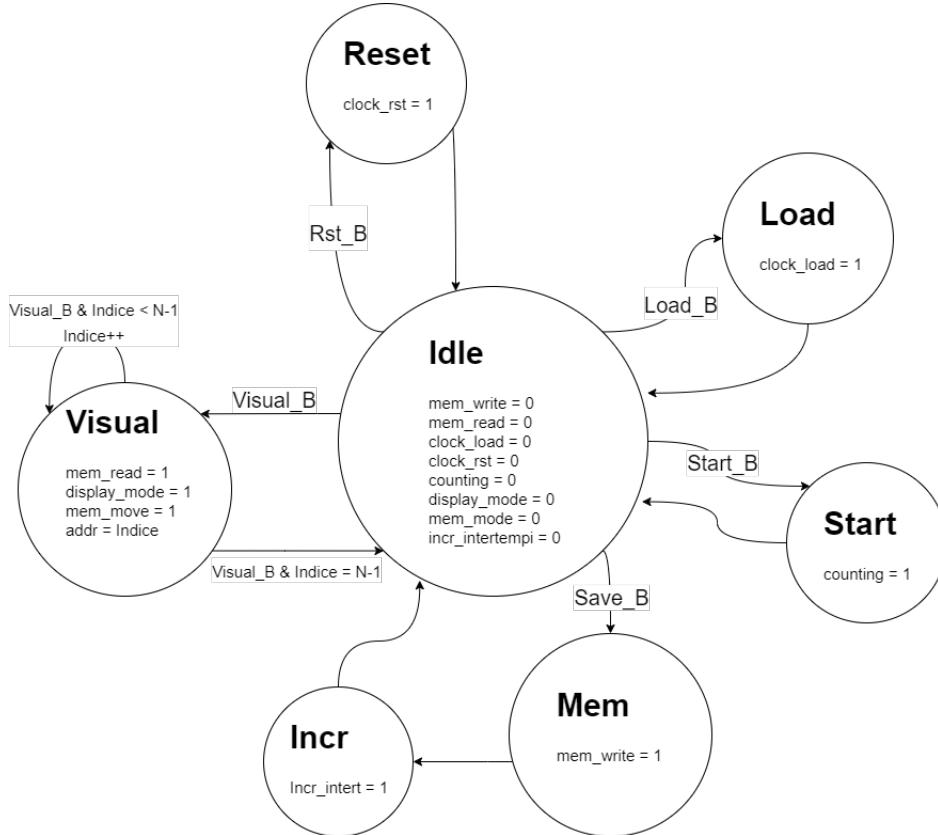


Figura 5.7: Automa a stati finiti della control unit

I possibili stati della UC sono: IDLE, START, MEM, INCR, RESET, LOAD e VISUAL. Come si può osservare dalla figura 5.7, abbiamo lo stato centrale IDLE, in cui sono messi a '0' tutti i segnali di controllo, e successivamente, in base all'input utente che diviene attivo, la macchina evolve in uno degli stati possibili, che intuitivamente descrivono le diverse modalità di funzionamento del nostro cronometro. E' da notare come la fase di memorizzazione di un intertempo è divisa in due stati elementari: MEM, in cui viene messa a '1' l'abilitazione a scrivere nella memoria, e INCR, in cui viene aggiornato il conteggio di intertempi

memorizzati. E' importante tenere traccia del numero di intertempi in quanto, nel caso venissero memorizzati più di 4 intertempi, si andrebbero a sovrascrivere delle locazioni di memoria già occupate. Inoltre, un'altra particolarità di questo automa è che una volta entrato nello stato VISUAL, si rimarrà in tale stato fino a quando tutti gli intertempi non sono stati visualizzati, per poi ritornare in IDLE.

5.4 Implementazione in VHDL

5.4.1 Flip Flop T

Il flip flop T è stato implementato con un approccio comportamentale, seguendo cioè la sua caratteristica di funzionamento:

```

entity ff_t is port(
    T : in std_logic;
    rst : in std_logic;
    clk: in std_logic;
    q: out std_logic
);
end ff_t;

architecture Behavioral of ff_t is
signal q_tmp : std_logic;
begin
    ff_proc : process(clk) begin
        if(rising_edge(clk)) then
            if(rst = '1') then
                q_tmp <= '0';
            elsif(T='1') then
                q_tmp <= not q_tmp;
            end if;
        end if;
    end process;
    q <= q_tmp;
end Behavioral;

```

```

    end if;

end process;

q <= q_tmp;
end Behavioral;
```

5.4.2 Contatore

Il componente contatore modulo M è stato definito con M generico, di tipo intero, assegnato ad un valore di default pari a 8. Si noti che per definire le cardinalità dell'ingresso parallel_input e Y sono stati usati i type di cast per i numeri interi e per i numeri reali in quanto il logaritmo prevede che l'argomento sia un numero reale, mentre per definire gli indici di uno std_logic_vector c'è bisogno di un numero intero.

```

use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;

entity counter is
    generic(
        M : integer := 8
    );
    port(
        clk, rst, count, load : in std_logic;
        parallel_input :
            in std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0);
        clk_out : out std_logic;
        Y : out std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)
    );
end counter;
```

```
architecture Behavioral of counter is

signal TY : std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0);

begin
    Y <= TY;

    output : process(TY) begin
        if(conv_integer(TY) = M-1) then
            clk_out <= '1';
        else
            clk_out <= '0';
        end if;
    end process;

    process(clk) begin
        if(rising_edge(clk)) then
            if(rst = '1') then
                TY <= (others => '0');
            elsif(load = '1') then
                TY <= parallel_input;
            elsif(count = '1') then
                if(conv_integer(TY) >= M-1) then
                    TY <= (others => '0');
                else
                    TY <= TY + "1";
                end if;
            end if;
        end if;
    end process;

end Behavioral;
```

5.4.3 Clock

Il clock è stato progettato con un approccio strutturale, seguendo la descrizione teorica fatta precedentemente:

```
use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;

entity orologio is
    port(
        clk, rst, load, count : in std_logic;
        sec_in : in std_logic_vector(5 downto 0);
        min_in : in std_logic_vector(5 downto 0);
        hour_in : in std_logic_vector(4 downto 0);

        sec : out std_logic_vector(5 downto 0);
        min : out std_logic_vector(5 downto 0);
        hour : out std_logic_vector(4 downto 0)
    );
end orologio;

architecture Structural of orologio is

    signal u : std_logic_vector(2 downto 0);
    signal enabs : std_logic_vector(1 downto 0);

begin

    enabs(0) <= u(0) and u(1);
    enabs(1) <= u(0) and u(1) and u(2);

    bdt : entity work.counter generic map(
```

```

M => 100000000
) port map(
    clk => clk, rst => rst, count => count,
    load => '0', parallel_input => (others => '0'),
    clk_out => u(0)
);

secs : entity work.counter generic map(
    M => 60
) port map(
    clk => clk, rst => rst,
    count => u(0), load => load,
    parallel_input => sec_in, Y => sec,
    clk_out => u(1)
);

mins : entity work.counter generic map(
    M => 60
) port map(
    clk => clk, rst => rst,
    count => enabs(0), load => load,
    parallel_input => min_in,
    Y => min, clk_out => u(2)
);

hours : counter generic map(
    M => 24
) port map(
    clk => clk, rst => rst,
    count => enabs(1), load => load,
    parallel_input => hour_in, Y => hour
);

```

```
end Structural;
```

5.4.4 BCD Converter

Il convertitore BCD, definito in maniera comportamentale, dichiara due variabili tmp e tmp2 nel process e nel body le assegna rispettivamente la cifra delle unità e quella delle decine dell'ingresso x, calcolate attraverso l'operazione di modulo. Si noti che per ri-trasformare tmp e tmp2 nel tipo unsigned viene usata la funzione to_unsigned:

```
use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;

entity BCDConverter is
    port(
        x : in std_logic_vector(5 downto 0);
        y : out std_logic_vector(7 downto 0));
end BCDConverter;

architecture Behavioral of BCDConverter is

begin
    process(x)
        variable tmp : integer;
        variable tmp2 : integer;
    begin
        tmp := conv_integer(x) mod 10;
        tmp2 := (conv_integer(x)/10) mod 10;
        y(3 downto 0) <= std_logic_vector(to_unsigned(tmp, 4));
        y(7 downto 4) <= std_logic_vector(to_unsigned(tmp2, 4));
    end process;
end Behavioral;
```

```
end process;  
end Behavioral;
```

5.4.5 Memoria

La memoria è stata implementata in VHDL con un approccio comportamentale e generico, per quanto riguarda i parametri N e M. Il funzionamento della memoria è sincrono col segnale di sincronizzazione *clk*, in particolare la macchina agisce sul suo fronte di salita. Di seguito l'implementazione dell'entity e dell'architecture.

```
use IEEE.NUMERIC_STD.ALL;  
use IEEE.math_real.all;  
use ieee.std_logic_unsigned.all;  
  
  
entity Memory is  
    generic(  
        N: integer := 32;  
        M: integer := 8  
    );  
    port(  
        rst : in std_logic;  
        read : in std_logic;  
        write : in std_logic;  
        clk : in std_logic;  
        input : in std_logic_vector(N-1 downto 0);  
        output : out std_logic_vector(N-1 downto 0);  
        address : in std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)  
    );  
end Memory;
```

```
architecture Behavioral of Memory is
type mem is array (M-1 downto 0) of std_logic_vector(N-1 downto 0);
signal data : mem := (others=>(others=>'0'));
begin
proc : process(clk) begin
    if(rising_edge(clk)) then
        if(rst='1') then
            data<=(others=>(others=>'0'));
        elsif(write='1') then
            data(conv_integer(address))<=input;
        elsif(read='1') then
            output <= data(conv_integer(address));
        else
            output <= (others=>'Z');
        end if;
    end if;
end process;

end Behavioral;
```

5.4.6 Unità operativa

La UO del cronometro presenta un interfaccia abbastanza complessa, con molti ingressi di controllo da gestire in input. Notiamo che nell'architecture body viene costruito il segnale di value_in come la concatenazione delle uscite dei tre BCD converter, a cui è aggiunto uno zero-padding di 8 zeri. Esso rappresenterà il valore dell'ora attuale, che può essere memorizzato come intertempo oppure essere visualizzato sul display a 7 segmenti.

Inoltre, nell'istanziazione del visore, si può osservare che vengono abilitati il terzo e il quinto punto partendo da destra, in modo da ottenere una visualizzazione

migliore dell'orario sul display.

```

use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;

entity DataPath is
    port(
        clk: in std_logic;
        -- Ingressi di controllo
        mem_write, mem_read : in std_logic;
        clock_load, clock_rst : in std_logic;
        incrementa_intertempi : in std_logic;
        counting : in std_logic;
        display_mode, mem_mode : in std_logic;
        addr : in std_logic_vector(1 downto 0);
        -- Ingressi utente
        min_in : in std_logic_vector(5 downto 0);
        hour_in : in std_logic_vector(4 downto 0);
        -- Stato
        n_intertempi : out std_logic_vector(1 downto 0);
        -- Segnali di uscita
        anodes :out std_logic_vector(7 downto 0);
        cathodes: out std_logic_vector(7 downto 0)
    );
end DataPath;

architecture Structural of DataPath is

signal mem_out : std_logic_vector(31 downto 0);
signal mem_addr: std_logic_vector(2-1 downto 0);
signal muxtovis : std_logic_vector(31 downto 0);

```

```

signal counter_intertempi : std_logic_vector(1 downto 0);

signal ff_out : std_logic;

signal value_in : std_logic_vector(31 downto 0);
signal secBCD : std_logic_vector(7 downto 0);
signal minBCD : std_logic_vector(7 downto 0);
signal hourBCD : std_logic_vector(7 downto 0);

signal secEX: std_logic_vector(5 downto 0);
signal minEX: std_logic_vector(5 downto 0);
signal hourEX : std_logic_vector(5 downto 0);

begin

    value_in <= x"00" & hourBCD & minBCD & secBCD;
    hourEX(5) <='0';

    n_intertempi <= counter_intertempi;

    Orol : entity work.orologio port map(
        clk=>clk, rst=>clock_rst, load => clock_load,
        count=>ff_out, sec_in=>(others=>'0'),
        min_in=>min_in, hour_in=>hour_in,
        sec=> secEX, min=>minEX, hour=>hourEX(4 downto 0)
    );

    Visore : entity work.display_seven_segments port map(
        CLK=>CLK, RST=>'0', VALUE=>muxtovis,
        DOTS=>"00010100", ENABLE=>"00111111",
        ANODES=>anodes, cathodes=>cathodes
    );

```

```

ConvSec : entity work.BCDConverter port map(
    x => secEX, y => secBCD
);

ConMin : entity work.BCDConverter port map(
    x => minEX, y => minBCD
);

ConvHour : entity work.BCDConverter port map(
    x => hourEX, y => hourBCD
);

FlipFlopT : entity work.ff_t port map(
    clk => clk, t => counting, rst => clock_rst , q=>ff_out
);

Mem : entity work.Memory generic map(
    M=>4, N=>32)
port map(
    read=>mem_read, rst=>clock_rst,
    clk=>clk, write=>mem_write,
    output=>mem_out, input=>value_in,
    address=>mem_addr
);

CounterIntertempi : entity work.Counter generic map(
    M=>4)
port map(
    clk=>clk, count=>incrementa_intertempi,
    y => counter_intertempi , rst=>clock_rst,
    load=>'0', parallel_input => (others=>'0')
);

```

```
) ;  
  
MuxDisplay : entity work.Multiplexer generic map(  
    N => 32  
) port map (  
    in1=>value_in, in2=>mem_out,  
    o=>muxtovis, s=>display_mode  
) ;  
  
MuxAddress : entity work.Multiplexer generic map(  
    N=>2)  
    port map (  
    in1=>counter_intertempi, in2=>addr,  
    s=>mem_mode, o=>mem_addr  
) ;  
  
end Structural;
```

5.4.7 Unità di controllo

L’unità di controllo è stata modellata in VHDL come tutti gli altri automi a stati finiti visti finora. L’unica osservazione da fare è che, mentre la UO lavora sul fronte di salita del clock, la UC funziona sul fronte di discesa. Ciò è stato fatto appositamente per fare in modo che le due parti non evolvano sullo stesso fronte del clock, il che potrebbe causare inconsistenze e, in generale, malfunzionamenti.

L’architecture body è formato da due processi: uno sensibile al clock, che descrive praticamente l’evoluzione tra gli stati in base al valore degli ingressi e del segnale temporaneo `stato_corrente`, inizializzato a IDLE; l’altro, invece, sensibile al segnale `stato_corrente`, che descrive come variano i segnali di controllo (che sono di uscita, in questo caso) nel passaggio da uno stato all’altro.

```
use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;
entity ControlUnit is
    port(
        clk : in std_logic;

        -- Input utente
        start_b : in std_logic;
        rst_b: in std_logic;
        load_b : in std_logic;
        mem_b : in std_logic;
        visual_b : in std_logic;

        -- Stato Datapath
        n_intertempi : in std_logic_vector(1 downto 0);

        -- Controllo
        mem_write : out std_logic;
        mem_read : out std_logic;
        clock_load : out std_logic;
        clock_RST : out std_logic;
        incrementa_intertempi : out std_logic;
        counting : out std_logic;
        display_mode : out std_logic;
        mem_mode : out std_logic;
        addr : out std_logic_vector(1 downto 0)

    );
end ControlUnit;

architecture Behavioral of ControlUnit is
```

```

type stato is (IDLE, START, MEM, INCR, RESET, LOAD, VISUAL);
signal stato_corrente : stato := IDLE;
signal IndiceVisualizzato : std_logic_vector(1 downto 0) := "00";
begin

state_prc : process(clk) begin

    if(falling_edge(clk)) then

        if(stato_corrente=IDLE) then

            if(start_b='1') then stato_corrente <= START;

            elsif(rst_b='1') then stato_corrente <= RESET;

            elsif(mem_b='1') then stato_corrente <= MEM;

            elsif(load_b='1') then stato_corrente <= LOAD;

            elsif(visual_b='1') then stato_corrente <= VISUAL;

            end if;

        elsif(stato_corrente=START) then stato_corrente <= IDLE;

        elsif(stato_corrente=RESET) then stato_corrente <= IDLE;

        elsif(stato_corrente=LOAD) then stato_corrente <= IDLE;

        elsif(stato_corrente=MEM) then stato_corrente <= INCR;

        elsif(stato_corrente=INCR) then stato_corrente <= IDLE;

        elsif(stato_corrente=VISUAL and visual_b='1') then

            if(IndiceVisualizzato="11") then

                IndiceVisualizzato<="00";

                stato_corrente <= IDLE;

            else

                IndiceVisualizzato<=IndiceVisualizzato+"01";

                stato_corrente <= VISUAL;

            end if;

        end if;

    end if;

end process;
```



```

out_prc : process(stato_corrente) begin

    mem_write <= '0';


```

```
mem_read <= '0';
clock_load <= '0';
clock_RST <= '0';
counting <= '0';
display_mode <= '0';
mem_mode <= '0';
incrementa_intertempi <='0';

if(stato_corrente=START) then
    counting <= '1';

elsif(stato_corrente=RESET) then
    clock_RST <= '1';

elsif(stato_corrente=LOAD) then
    clock_load <= '1';

elsif(stato_corrente=MEM) then
    mem_write <='1';

elsif(stato_corrente=INCRL) then
    incrementa_intertempi <= '1';

elsif(stato_corrente=VISUAL) then
    mem_read <= '1';
    display_mode <= '1';
    mem_mode <= '1';
    addr <= IndiceVisualizzato;

end if;
end process;
end Behavioral;
```

5.4.8 Top module

Il top module, chiamato Orologio_on_Board, è intuitivamente definito in maniera strutturale. Infatti, esso è formato dalla control unit, dalla parte operativa e da tre button debouncer che serviranno successivamente per la sintesi su FPGA. E' da notare che, rispetto alla UC e alla UO, il top module ha un interfaccia molto più semplice e facile da comprendere, in cui troviamo solo il segnale di sincronizzazione *clk*, gli input utente e le uscite di anodi e catodi per il display a 7 segmenti. Di seguito è presenta l'implementazione dell'entity e dell'architecture in VHDL.

```

use IEEE.NUMERIC_STD.ALL;
use IEEE.math_real.all;
use ieee.std_logic_unsigned.all;

entity Orologio_on_Board is
    port(
        clk: in std_logic;

        start_b : in std_logic;
        rst_b: in std_logic;
        load_b : in std_logic;
        mem_b : in std_logic;
        visual_b : in std_logic;

        min_in : in std_logic_vector(5 downto 0);
        hour_in : in std_logic_vector(4 downto 0);

        anodes :out std_logic_vector(7 downto 0);
        cathodes: out std_logic_vector(7 downto 0)
    );
end Orologio_on_Board;

```

```

architecture Structural of Orologio_on_Board is

--UC

signal clock_load : std_logic;
signal clock_RST : std_logic;
signal counting : std_logic;
signal mem_read : std_logic;
signal mem_write : std_logic;
signal mem_mode : std_logic;
signal display_mode : std_logic;
signal addr : std_logic_vector(1 downto 0);
signal incrementa_intertempi : std_logic;


--CONTATORE INTERTEMPI

signal n_intertempi : std_logic_vector(1 downto 0);


--BOTTONI

signal cleared_mem : std_logic;
signal cleared_start: std_logic;
signal cleared_visual : std_logic;


begin

CU : entity work.ControlUnit port map(
    clk=>clk, mem_write=>mem_write,
    mem_read=>mem_read, clock_load=>clock_load,
    clock_RST=>clock_RST, incrementa_intertempi=>incrementa_intertempi,
    counting=>counting, display_mode=>display_mode,
    mem_mode=>mem_mode, n_intertempi=>n_intertempi,
    start_b=>cleared_start, rst_b=>rst_b, load_b=>load_b,

```

PROGETTO 5. CRONOMETRO

```
mem_b=>cleared_mem, visual_b=>cleared_visual, addr=>addr
);

DP : entity work.DataPath port map(
    clk => clk, mem_write => mem_write,
    mem_read => mem_read, clock_load => clock_load,
    clock_rst => clock_rst, incrementa_intertempi => incrementa_intertempi,
    counting => counting, display_mode => display_mode, mem_mode => mem_mode,
    addr => addr, min_in => min_in, hour_in => hour_in,
    n_intertempi => n_intertempi,
    anodes => anodes, cathodes => cathodes
);

MemDebouncer : entity work.ButtonDebouncer generic map(
    CLK_period=>10, btn_noise_time=>325000000)
port map(
    rst=>'0', clk=>clk, btn=>mem_b, cleared_btn=>cleared_mem);

StartDebouncer : entity work.ButtonDebouncer generic map(
    CLK_period=>10, btn_noise_time=>325000000)
port map(
    rst=>'0', clk=>clk, btn=>start_b, cleared_btn=>cleared_start);

VisualDebouncer : entity work.ButtonDebouncer generic map(
    CLK_period=>10, btn_noise_time=>325000000)
port map(
    rst=>'0', clk=>clk, btn=>visual_b, cleared_btn=>cleared_visual);

end Structural;
```

5.5 Testing simulato

Abbiamo successivamente testato mediante simulazione il funzionamento del clock (orologio). E' da notare che, per simulare il conteggio, abbiamo assegnato un valore pari a 2 ad M del contatore della base dei tempi. Quindi, passando il valore di M da 100000000 a 2, il contatore avrà un ciclo di conteggio molto più veloce e questo ci consentirà di velocizzare la simulazione. Infatti, la generazione di una simulazione di durata di un secondo, il minimo che servirebbe per verificare il corretto funzionamento del clock, richiede molto tempo: nel nostro test bench, di conseguenza, un intervallo di un secondo verrà ristretto ad un intervallo di 40 ns in modo tale che esso possa essere visualizzato graficamente.

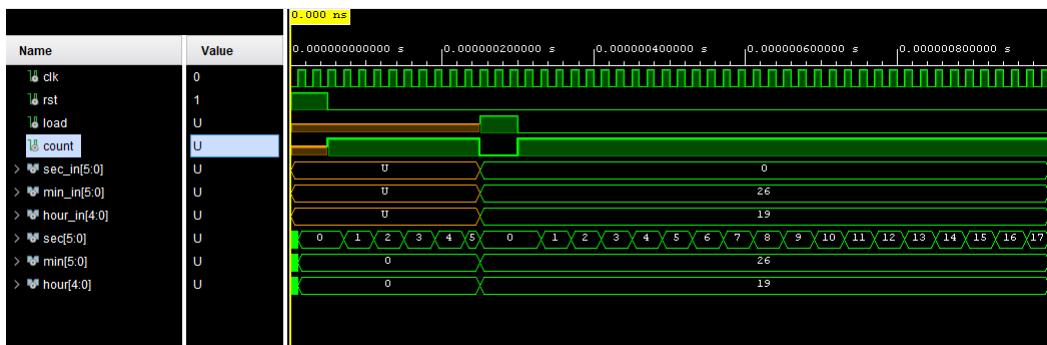


Figura 5.8: Test bench del componente clock

Inizialmente, alziamo il segnale di abilitazione a contare *count* e, come possiamo vedere dalla figura 5.8, l'uscita *sec* inizia a variare, mostrando lo scorrere dei secondi. Inoltre, abbiamo testato anche la funzionalità di pre-caricamento di un determinato orario, con un valore arbitrario di secondi, minuti e ore, e di conteggio a partire dall'orario pre-caricato. Infine, abbiamo alzato il segnale di reset *rst* per resettare il ciclo di conteggio.

5.6 Caricamento sulla scheda

Prima di poter caricare il progetto sulla scheda è stato necessario apportare alcune semplici modifiche. In particolar modo, abbiamo dovuto introdurre alcuni button debouncer in modo da evitare che le oscillazioni dei tasti start_b, mem_b e visual_b potessero portare l'unità di controllo in stati indesiderati.

Abbiamo quindi effettuato il mapping dei tasti start_b, mem_b, visual_b, reset_b e load_b, dei catodi e degli anodi del display forniti dal componente visore e degli switch tramite i quali l'utente può selezionare l'orario da pre-caricare all'interno del cronometro con il tasto load. Non disponendo di un numero sufficiente di switch per il pre-caricamento dell'orario per intero abbiamo deciso di prevedere esclusivamente la possibilità di selezionare l'ora e i minuti, in corrispondenza della pressione del tasto di load i secondi vengono invece resettati ed impostati a zero. Una possibile alternativa sarebbe potuta essere quella di prevedere un caricamento in più fasi, facendo in modo di utilizzare gli stessi switch per il caricamento dei secondi ma avrebbe complicato il comportamento della parte di controllo. Abbiamo inoltre connesso il clock a 100 MHz della scheda al nostro componente.

Lo schema complessivo degli ingressi utente sulla scheda è quindi il seguente:

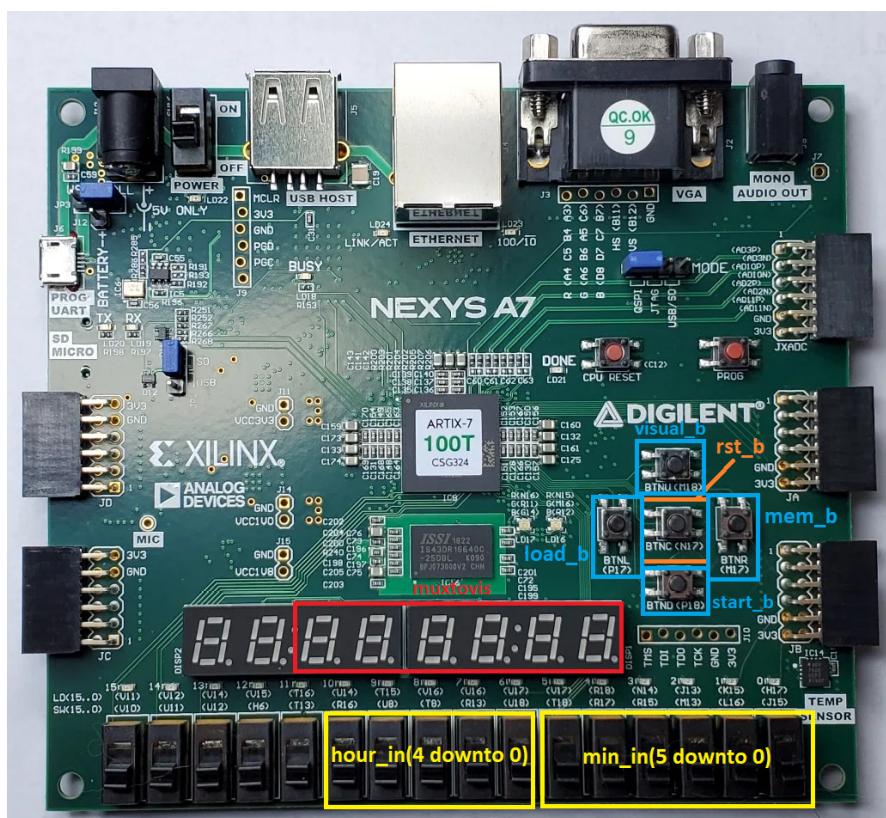


Figura 5.9: Mapping dei segnali del sistema su FPGA

Progetto 6

Sistema di testing

6.1 Traccia

1. 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 sottponendole 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.
2. Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due bottoni per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

6.2 Descrizione soluzione teorica

Per tale esercizio si è scelto di adottare un approccio strutturale che prevedesse il riuso di componenti già precedentemente progettati ed implementati. In particolare, è stato fatto utilizzo di una ROM in cui precaricare le sequenze di input per la macchina combinatoria da testare (N sequenze di input da 4 bit ciascuna) ed una memoria per il salvataggio delle sequenze di output corrispondenti (N sequenze di output da 3 bit ciascuna). E' stato chiaramente utilizzato un contatore per l'indirizzamento della ROM e della memoria di output.

La soluzione è stata progettualmente suddivisa in unità di controllo ed unità operativa. La prima fornisce i segnali di controllo che pilotano l'unità operativa quali l'abilitazione alla lettura della ROM, l'abilitazione del contatore per l'indirizzamento, il reset ed i segnali di lettura e scrittura per la memoria di output. Tali segnali provengono dall'esterno grazie all'utilizzo di appositi bottoni, uno per il reset, uno per fare iniziare il testing ed uno per la visualizzazione. In particolare, alla pressione del bottone di read il contenuto della ROM indirizzo per indirizzo (le sequenze di input) viene fornito alla macchina combinatoria la cui uscita viene poi riportata all'interno della memoria, dunque alla fine del processo di testing nella memoria saranno scritte tutte le sequenze elaborate. Invece il bottone di visualizzazione permette di indirizzare la memoria di output attraverso l'input proveniente dagli switch e di visualizzare il contenuto della locazione indirizzata tramite l'uso di appositi led.

6.3 Schematici, componenti

Il Sistema di testing prende in ingresso il segnale di temporizzazione (clock), un segnale di read, un segnale di reset ed un segnale di visualizzazione, tutti provenienti dalla pressione di un dato bottone. In uscita è in fine fornito il segnale per

l'accensione dei led opportuni.

L'interfaccia del componente è quindi la seguente:

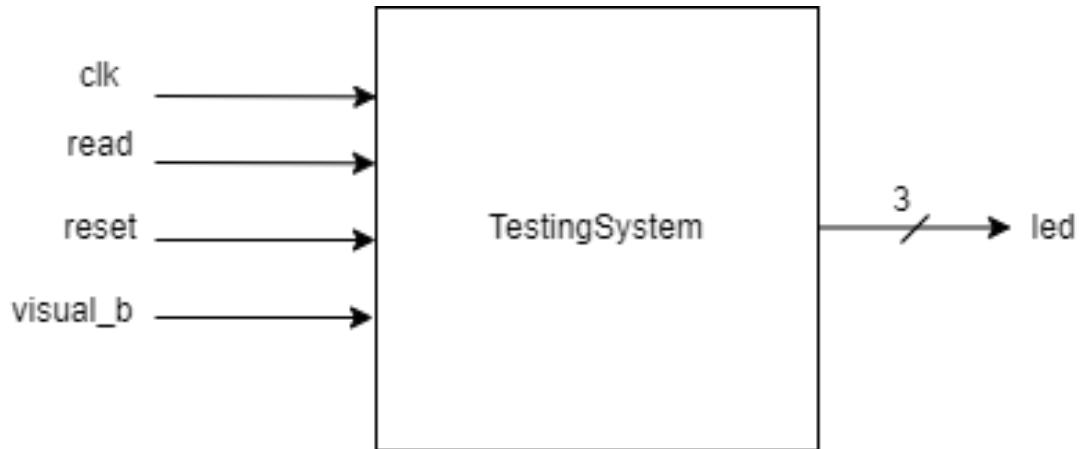


Figura 6.1: Schematico sistema di testing

In modalità di visualizzazione è inoltre previsto un ingresso addizionale che permette di indirizzare direttamente la memoria consentendo di ispezionarne i valori.

Tale sistema, come detto, è costituito da due macro-componenti: parte operativa e parte di controllo. A sua volta la parte operativa è stata progettata seguendo un approccio strutturale, mentre la parte di controllo può teoricamente essere descritta mediante un automa a stati finiti.

Di seguito vengono mostrati gli schematici relativi ai componenti che costituiscono l'unità operativa e la rappresentazione dell'unità di controllo.

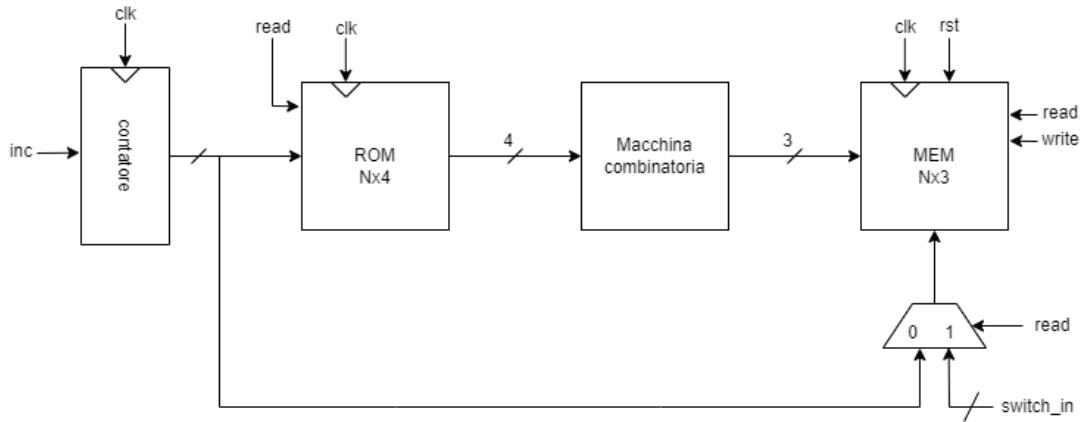


Figura 6.2: Schematico unità operativa

Come mostrato in figura 8.1 i componenti dell'unità operativa sono:

- Un contatore che riceve in ingresso un segnale di clock (clk) ed un segnale di abilitazione all'incremento (inc), la cui uscita serve ad indirizzare le due memorie.
- Una ROM che riceve in ingresso un segnale di clock (clk), un segnale di abilitazione alla lettura (read) della locazione indirizzata dal valore di conteggio a cui si trova il contatore. In particolare tale ROM, come da specifica, consta di N locazioni da 4 bit, dunque l'ingresso di indirizzamento sarà una parola di $\lceil \log_2(N) \rceil$ bit.
- Una macchina combinatoria con 4 ingressi e 3 uscite da testare.
- Una memoria che riceve in ingresso un segnale di clock (clk), un segnale di reset (rst), un segnale di read e di write sulla base dei quali è possibile leggere o scrivere alla locazione specificata.
- Un multiplexer, grazie al quale in corrispondenza del segnale di read la memoria è indirizzata tramite l'input degli switch, altrimenti se read è basso l'indirizzamento è proveniente dal contatore (che effettivamente indirizza la memoria se write è alto).

Per quanto riguarda l'unità di controllo, gestisce i segnali di comando della parte operativa implementando una logica definita sulla base di uno stato corrente. Si parte da uno stato di *START* e in corrispondenza della pressione del bottone di read si passa allo stato *LOAD* in cui viene abilitata la ROM, al successivo colpo di clock si arriva in uno stato di memorizzazione *MEM* leggendo dalla ROM e abilitando la scrittura nella MEM. Al colpo di clock seguente si arriva in uno stato *INC* in cui si incrementa il valore del contatore: se il counter_value è arrivato al numero di sequenze di testing allora l'automa torna allo stato iniziale di *START*, altrimenti al successivo colpo di clock l'automa passa nello stato di *LOAD* (e si ripete il ciclo fino a riempimento della MEM con tutte le sequenza elaborate). Nello stato di *START* inoltre, alla pressione del bottone di visualizzazione che abilita il segnale *visual_b*, l'automa transita nello stato *VISUAL* nel quale è alto il segnale di *mem_read*. Come abbiamo visto se il segnale di *mem_read* è alto allora in ingresso al multiplexer verso la MEM è selezionato l'input proveniente dagli switch.

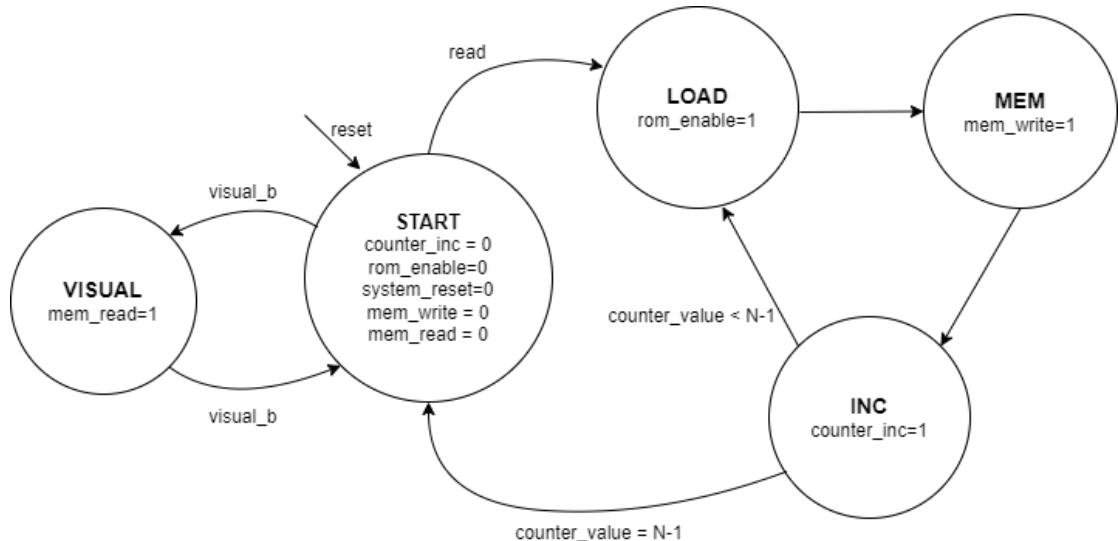


Figura 6.3: automa a stati finiti unità di controllo

6.4 Implementazione in VHDL

La memoria ROM (Read-Only-Memory) è un tipo di memoria non volatile nella quale i dati scritti (o memorizzati) non possono essere elettronicamente modificati dopo la sua inizializzazione. In questo esercizio è atta a mantenere le sequenze di input è stata modellata ad un livello di astrazione comportamentale. In particolare il process che la descrive è sensibile al rising edge del clock e quando il segnale di reset è alto l'uscita della ROM (data) è settata ad un valore di default (000) mentre se è alto il segnale di read allora in uscita è riportato il valore contenuto all'indirizzo ADDR.

```
entity ROM is
    port(
        clk : in std_logic;
        reset : in std_logic;
        read: in std_logic;
        addr : in std_logic_vector(2 downto 0);
        data : out std_logic_vector(3 downto 0));
    end ROM;

architecture Behavioral of ROM is
    type rom_type is array(7 downto 0) of std_logic_vector(3 downto 0);
    signal ROM : rom_type := (
        X"F",
        X"A",
        X"C",
        X"E",
        X"B",
        X"0",
        X"0",
        X"C");
begin
```

```

process (CLK) begin
    if rising_edge(CLK) then
        if (ReSeT='1') then
            DATA <= ROM(conv_integer("000")); --valore di default
        elsif(READ ='1') then
            DATA <= ROM(conv_integer(ADDR));
        end if;
    end if;
end process;

end Behavioral;

```

Anche la memoria di output è stata modellata con approccio comportamentale. In particolare è stata definita l'entità Memory come generica fornendo 32 e 8 come valori di default per N ed M (rispettivamente la grandezza del blocco di memoria e il numero di blocchi di memoria).

```

entity Memory is
    generic(
        N: integer := 32;
        M: integer := 8;
    );
    port(
        rst : in std_logic;
        read : in std_logic;
        write : in std_logic;
        clk : in std_logic;
        input : in std_logic_vector(N-1 downto 0);
        output : out std_logic_vector(N-1 downto 0);
        address : in std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)
    );
end Memory;

architecture Behavioral of Memory is

```

```

type mem is array (M-1 downto 0) of std_logic_vector(N-1 downto 0);
signal data : mem := (others=>(others=>'0'));
begin
proc : process(clk) begin
    if rising_edge(clk) then
        if(rst='1') then
            data<= (others=>(others=>'0'));
        elsif(write='1') then
            data(conv_integer(address))<=input;
        elsif(read='1') then
            output <= data(conv_integer(address));
        else
            output <= (others=>'Z');
        end if;
    end if;
end process;

end Behavioral;

```

La macchina combinatoria è stata realizzata a livello dataflow, essa realizza le semplici operazioni aritmetiche di and, or e xor tra i 4 bit di ingresso (fornendo 3 uscite) come mostrato nel codice seguente:

```

entity macchinaCombinatoria is
    port(
        i : in std_logic_vector(3 downto 0);
        o : out std_logic_vector(2 downto 0));
end macchinaCombinatoria;

architecture Dataflow of macchinaCombinatoria is
signal ris1 : std_logic;
begin
    o(0) <= i(3) and i(2) and i(1) and i(0);
    o(1) <= i(3) or i(2) or i(1) or i(0);

```

```
    o(2) <= i(3) xor i(2) xor i(1) xor i(0);  
  
end Dataflow;
```

Per la realizzazione del contatore di indirizzamento nelle memorie è stato poi fatto riuscito del componente counter (generico) precedentemente realizzato, del quale si riporta l'interfaccia di seguito. Tuttavia è da sottolineare che rispetto alla precedente implementazione è stata aggiunta una inizializzazione di default a valore nullo della variabile di conteggio.

```
entity counter is  
  generic(  
    M : integer := 8  
  );  
  port(  
    clk, rst, count, load : in std_logic;  
    parallel_input :  
      in std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0);  
    clk_out : out std_logic;  
    Y : out std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)  
  );  
end counter;
```

Per quanto riguarda l'architettura, essa è stata realizzata con astrazione comportamentale. In particolare si è fatto utilizzo di due process distinti. Il primo, sensibile al segnale TY, gestisce l'uscita clk_out (la quale assume valore alto quando TY risulta pari ad M-1). Il secondo, sensibile al clk, si occupa di gestire il valore del segnale TY in corrispondenza del rising edge del clk: Quando è alto il segnale di reset "rst" il segnale TY è settato a valore nullo, quando è alto il segnale di load allora TY è caricato al valore contenuto nel parallel_input, se count è alto allora TY è incrementato di una unità oppure resettato a valore nullo se risulta

maggiori o uguali di M-1. Infine all'esterno dei due process viene assegnato il segnale TY (gestito da i process stessi) all'uscita Y.

```

architecture Behavioral of counter is

signal TY : std_logic_vector(integer(ceil(log2(real(M))))-1 downto 0)
:= (others=>'0');

begin
    Y <= TY;

    output : process(TY) begin
        if(conv_integer(TY) = M-1) then
            clk_out <= '1';
        else
            clk_out <= '0';
        end if;
    end process;

    process(clk) begin
        if(rising_edge(clk)) then
            if(rst = '1') then
                TY <= (others => '0');
            elsif(load = '1') then
                TY <= parallel_input;
            elsif(count = '1') then
                if(conv_integer(TY) >= M-1) then
                    TY <= (others => '0');
                else
                    TY <= TY + "1";
                end if;
            end if;
        end if;
    end process;
end Behavioral;

```

```
end Behavioral;
```

Per quanto riguarda l'implementazione dell'unità di controllo, essa presenta come ingressi i segnali provenienti dall'unità operativa(counter_value), insieme a quelli provenienti dall'esterno(read,reset,visual_b) e come uscite i segnali di controllo che ne gestiscono l'evoluzione. Di seguito ne viene riportata l'interfaccia:

```
entity ControlUnit is
    port(
        read : in std_logic;
        clk : in std_logic;
        reset : in std_logic;
        counter_value : in std_logic_vector(2 downto 0);
        visual_b : in std_logic;
        counter_inc : out std_logic;
        rom_enable : out std_logic;
        system_reset : out std_logic;
        --uscita che vale come reset per la parte operativa
        mem_write : out std_logic;
        mem_read : out std_logic);
end ControlUnit;
```

Il comportamento viene definito in accordo alla sua rappresentazione come automa a stati finiti 8.1 tramite l'utilizzo di due process: il primo che gestisce l'evoluzione degli stati sulla base dello stato corrente e l'altro che si occupa, invece, di settare i segnali al valore corretto, nel momento corretto, per l'unità operativa.

```
architecture Behavioral of ControlUnit is
type stato is (START, LOAD, MEM, INC, VISUAL);
signal state : stato := START;
begin
    cu_proc_state: process(clk) begin
        if(rising_edge(clk)) then
```

```

if(reset='1') then state <= START;

elsif(state=START) then

    if(read='1') then state <= LOAD;

    elsif(visual_b='1') then state <= VISUAL;

    end if;

elsif(state=LOAD) then state <= MEM;

elsif(state=VISUAL and visual_b='1') then state<=START;

elsif(state=MEM) then state <= INC;

elsif(state=INC) then

    if(counter_value="111") then state <= START;

    else state <= LOAD;

    end if;

end if;

end if;

end process;


cu_proc_out : process(state) begin

    counter_inc <= '0';

    rom_enable <= '0';

    system_reset <= '0';

    mem_write <='0';

    mem_read <='0';

    if(state=LOAD) then

        rom_enable <= '1';

    elsif(state=MEM) then

        mem_write <='1';

    elsif(state=INC) then

        counter_inc <= '1';

    elsif(state=VISUAL) then

        mem_read <= '1';

    end if;

end process;

end Behavioral;

```

Per quanto riguarda l'unità operativa, essa riceve i segnali di controllo dalla control unit e presenta come uscita il valore del contatore per determinare la fine delle sequenze da testare e l'output vero e proprio come uscita del componente memoria, come riportato nella seguente definizione di interfaccia:

```
entity operatingUnit is
    port(
        counter_inc : in std_logic;
        rom_enable: in std_logic;
        switch_in : in std_logic_vector(2 downto 0);
        reset : in std_logic;
        mem_read : in std_logic;
        mem_write: in std_logic;
        clk : in std_logic;
        counter_value : out std_logic_vector(2 downto 0);
        output : out std_logic_vector(2 downto 0));
end operatingUnit;
```

Nella parte dichiarativa dell'architecture della operating unit vengono quindi istanziati i quattro componenti necessari (contatore, rom, macchina combinatoria e memoria di output) e quattro segnali interni: countertorom per collegare l'uscita del contatore alla ROM, romtocomb per collegare i 4 bit di uscita della memoria ROM con i 4 di ingresso della rete combinatoria utilizzata, combtomem per collegare i bit di uscita della rete con la memoria di output e mem_address per indirizzare la memoria attraverso l'input utente switch_in in fase di lettura o attraverso il contatore in fase di scrittura.

```
architecture Structural of operatingUnit is
    signal countertorom : std_logic_vector(2 downto 0);
    signal romtocomb : std_logic_vector(3 downto 0);
    signal combtomem : std_logic_vector(2 downto 0);
    signal mem_address : std_logic_vector(2 downto 0);
```

```

begin

    with mem_read select

        mem_address<=countertorom when '0',
            switch_in when '1',
            switch_in when others;


    cnt : entity work.counter generic map(
        M=>8)

    port map(
        clk=>clk, rst=>reset, count=>counter_inc,
        load=>'0', parallel_input=>(others=>'0'), Y=>countertorom
    );

    counter_value<=countertorom;

    readonlymemory : entity work.rom PORT map(
        clk=>clk, reset=>reset, read=>rom_enable,
        addr=> countertorom, data=>romtocomb);

    comb : entity work.macchinaCombinatoria PORT map(
        i=>romtocomb, o=>combtomem);

    mem : entity work.memory GENERIC MAP(
        N=>3, M=>8)

    port map(
        rst=>reset, read=>mem_read, write=>mem_write,
        clk=>clk, input=>combtomem,
        address=>mem_address, output =>output);

end Structural;

```

Infine il Sistema di testing viene realizzato secondo un approccio strutturale, istanziando come componenti la control unit, l'operating unit e i button debouncer (precedentemente definiti) e collegandoli opportunamente per la costruzione

dell'architettura complessiva tramite l'utilizzo di appositi segnali interni (quali counter_value, counter_inc, rom_enable, system_reset, mem_write, mem_read, cleared_read e cleared_visual).

```
entity TestingSystemOnBoard is
    port(
        read : in std_logic;
        clk : in std_logic;
        reset : in std_logic; --resetta la memoria
        visual_b : in std_logic;
        switch_in : in std_logic_vector(2 downto 0);
        led : out std_logic_vector(2 downto 0));
    end TestingSystemOnBoard;

architecture Structural of TestingSystemOnBoard is

    signal counter_value : std_logic_vector(2 downto 0);
    signal counter_inc : std_logic;
    signal rom_enable : std_logic;
    signal system_reset : std_logic;
    signal mem_write : std_logic;
    signal mem_read : std_logic;
    signal cleared_read : std_logic;
    signal cleared_visual : std_logic;
begin

    cu : entity work.controlunit port map(
        read=>cleared_read, clk=>clk, reset=>reset,
        counter_value=>counter_value, counter_inc=>counter_inc,
        rom_enable=>rom_enable, system_reset=>system_reset,
        mem_write=>mem_write, mem_read=>mem_read,
        visual_b=>cleared_visual);

    ou : entity work.operatingunit port map(
```

```
        counter_inc=>counter_inc, rom_enable=>rom_enable,
        reset=>system_reset, mem_read=>mem_read,
        mem_write=>mem_write, clk=>clk,
        counter_value=>counter_value,
        output=>led, switch_in=>switch_in);

debr : entity work.ButtonDebouncer generic map(
    CLK_PERIOD=>10, btn_noise_time=>650000000)
port map(
    clk=>clk, reset=>reset,
    btn=>read, cleared_btn=>cleared_read);

debv: entity work.ButtonDebouncer generic map(
    CLK_PERIOD=>10, btn_noise_time=>650000000)
port map(
    clk=>clk, reset=>reset,
    btn=>visual_b, cleared_btn=>cleared_visual);
end Structural;
```

6.5 Testing simulato

Successivamente abbiamo simulato il Sistema di Testing tramite un semplice Test-bench. E' stato innanzitutto definito il processo *clk_prc* che simula l'andamento di un clock con periodo pari a 10ns. In seguito, dopo aver istanziato la nostra unit under test, viene realizzato un ulteriore processo in cui dopo aver fornito un segnale di *reset*, si simula la pressione del bottone esterno ponendo ad '1' il segnale di *read*. A questo punto, dato l'automatismo di tale sistema, non ci resta che osservare i dati memorizzati nella memoria, tramite il segnale chiamato *data*. In particolare, il segnale di lettura viene alzato un numero di volte pari a quello delle stringhe pre-caricate all'interno della ROM. La simulazione viene mostrata in figura 6.4 dove si vede chiaramente la successione di stati dell'unità di controllo.

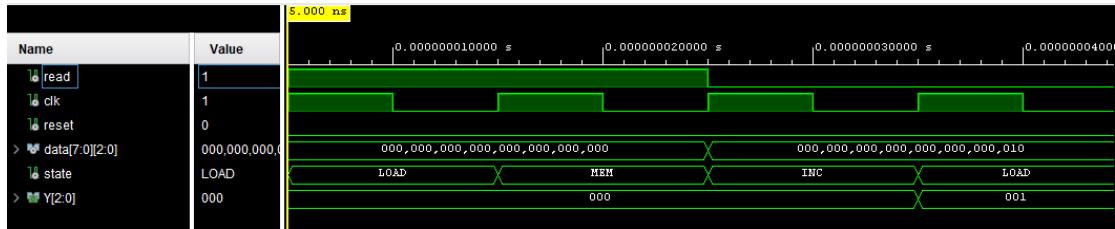


Figura 6.4: Test bench del sistema

6.6 Caricamento sulla scheda

Infine si è proceduti sintetizzando e implementando sulla scheda il TestingSystem, per validare il suo corretto funzionamento. I segnali di read e di reset sono stati rispettivamente assegnati a due buttoni (filtrati poi grazie al button debouncer) rispettivamente N17 ed M18. Per quanto riguarda la visualizzazione dell’uscita, come detto, sono stati utilizzati 3 dei led di interfaccia disponibili sulla board, in particolare H17, K15 e J13. Il tutto è mostrato nella figura 6.5:

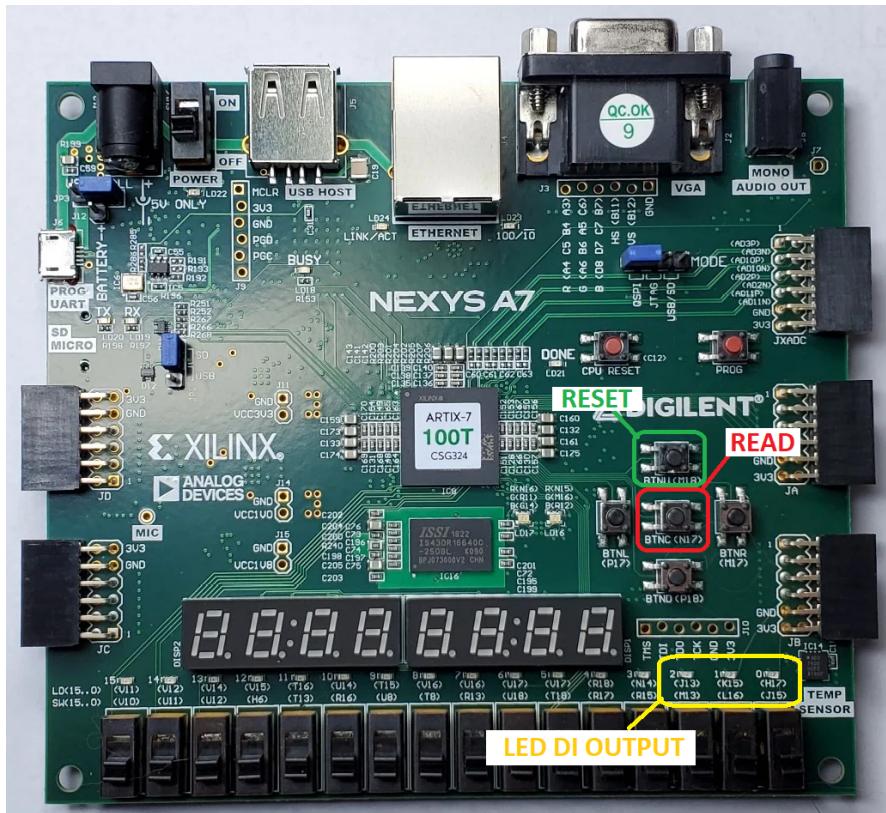


Figura 6.5

Progetto 7

Comunicazione con handshaking

Nella comunicazione tra entità differenti è necessario definire dei protocolli che regolino l'interazione tra i due dispositivi. Ciò risulta essere fondamentale tutte le volte in cui un componente viene inserito all'interno di un'architettura complessa in cui questo dovrà intraggire con altre parti del sistema.

I protocolli di comunicazione possono essere suddivisi in 3 tipologie:

- Sincroni
- Asincroni
- Semisincroni

Si parla di protocolli di tipo sincrono quando le due entità lavorano con lo stesso riferimento temporale, che risulta quindi condiviso. In tal caso il protocollo sfrutta il clock condiviso, la comunicazione in questo caso è iniziata da una fase di prologo.

I protocolli asincroni si basano invece sul concetto di evento, per cui le entità comunicano e scambiano i relativi messaggi al verificarsi di specifici eventi. Un esempio di protocollo asincrono è il protocollo di handshaking o di handshaking completo che si adattano perfettamente al caso di entità regolate da clock differenti e che quindi potrebbero non appartenere allo stesso sistema. I protocolli

semisincroni, invece, funzionano allo stesso modo di quelli sincroni prevedendo però per lo slave della comunicazione la possibilità di ritardare il messaggio tramite un apposito segnale(flag) notificato al master.

7.1 Traccia

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.

7.2 Descrizione soluzione teorica

La comunicazione tra i nodi A (master o mittente) e B (slave o ricevitore) è gestita tramite un protocollo di handshaking. Quest'ultimo fa uso di 2 segnali: un segnale di ready, gestito dal master, ed un segnale di ack, gestito dallo slave. In particolare il master procede innanzitutto alla trasmissione di una stringa di M bit sul bus che lo collega al nodo B, subito dopo alza il segnale di ready per comunicare al ricevitore che il dato è disponibile sul bus per essere prelevato. Lo slave, che era in attesa che il segnale di ready si alzasse, può a questo punto, subito dopo aver prelevato il dato, alzare il segnale di ack per comunicare al mittente che il dato

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

è stato prelevato. Il master, il quale era in tale frangente in attesa del segnale di ack con il segnale di ready alto, una volta che l'ha ricevuto procede ad abbassare la linea di ready e si pone in attesa che lo slave faccia lo stesso con la linea di ack. All'abbassarsi della linea ack la comunicazione è considerata definitivamente conclusa.

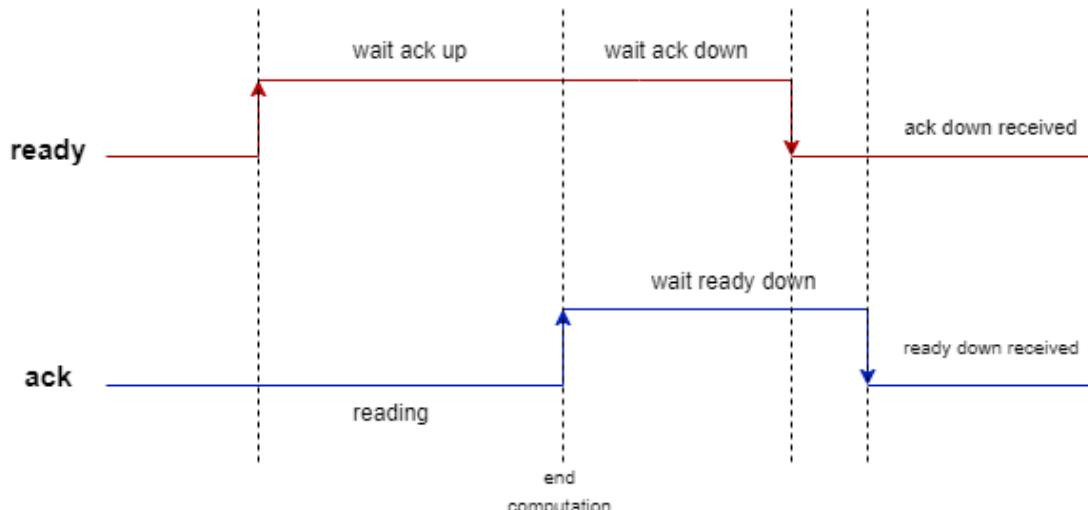


Figura 7.1

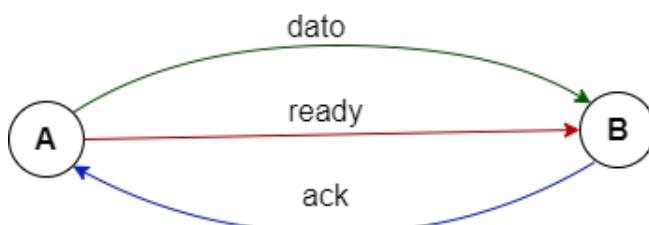


Figura 7.2

Passiamo ora ad esporre la nostra soluzione teorica per la realizzazione dei nodi di comunicazione. Entrambi si compongono di un datapath, che fa da unità operativa, e di un'unità di controllo che la gestisce; inoltre i due nodi si basano su segnali di clock a frequenza differente, generati tramite due apposite basi dei tempi. In particolare il Master lavora a frequenze più alte rispetto allo slave. Alla pressione di un pulsante che funge da start (inizio) della comunicazione, il nodo A preleva la prima stringa dalla propria ROM interna (precaricata con N stringhe

di M bit, con N=8 e M=16) grazie ad un indirizzamento gestito tramite apposito contatore, per memorizzarla all'interno di un registro tampone che fa da buffer per il bus parallelo. Tutto ciò è effettuato dal nodo A gestendo opportunamente i segnali necessari all'implementazione del meccanismo di handshaking. A questo punto il nodo B preleva sia la stringa ricevuta dal nodo A (da un apposito registro che funge da buffer in ricezione) sia quella contenuta nella propria memoria interna (anch'essa precaricata con 8 stringhe di 16 bit), per poi inviarle ad un componente che si occupa di effettuarne la somma. L'uscita di tale sommatore viene posta in un ulteriore registro prima di andare a sovrascrivere il contenuto della memoria di B all'indirizzo specificato da un opportuno contatore usato per il suo indirizzamento. In particolare per il nodo B è stata prevista un'ulteriore modalità di funzionamento, grazie all'utilizzo di un multiplex, secondo la quale si può decidere se continuare ad indirizzare la sua memoria tramite il contatore interno e dunque ricevere stringhe dal nodo A oppure procedere alla visualizzazione del suo contenuto tramite il visore presente sulla scheda, indirizzando in questo caso la memoria tramite gli switch della scheda. Il passaggio tra le due modalità viene realizzato alla pressione di un bottone di *visual*.

7.3 Schematici, componenti

Di seguito riportiamo lo schema black box del Sistema di Handshaking complesso:



Figura 7.3

esso accetta in ingresso il segnale di clock, il segnale di start proveniente dal pulsante posizionato sulla scheda, i segnali provenienti dagli switch che consentono di visualizzare uno specifico indirizzo della memoria di B. e il segnale visual_b, anch'esso generato grazie ad un bottone esterno che porta il sistema nello stato di visualizzazione; in uscita invece ritroviamo i vettori necessari alla visualizzazione del contenuto della memoria sul visore (anodi e catodi). Come detto in precedenza esso è costituito dai due nodi A e B i cui datapath vengono presentati di seguito:

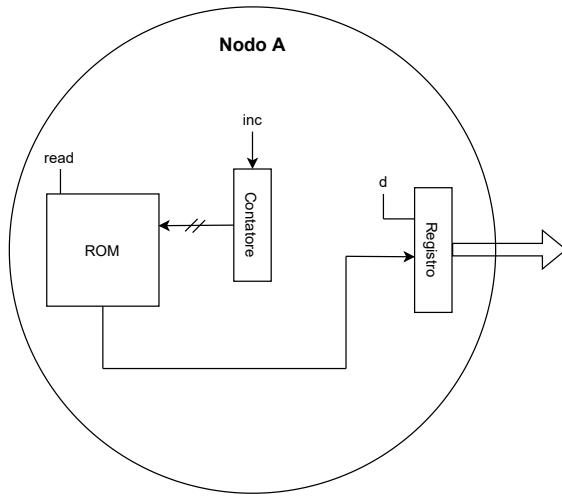


Figura 7.4: Unità operativa del trasmettitore.

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

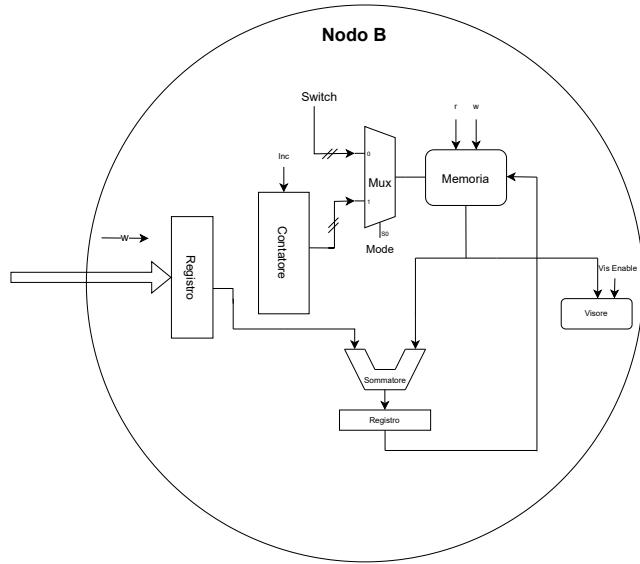


Figura 7.5: Unità operativa del ricevitore.

Le rispettive unità di controllo possono essere descritte mediante automi a stati finiti rispettivamente in figura 7.6 e 7.7.

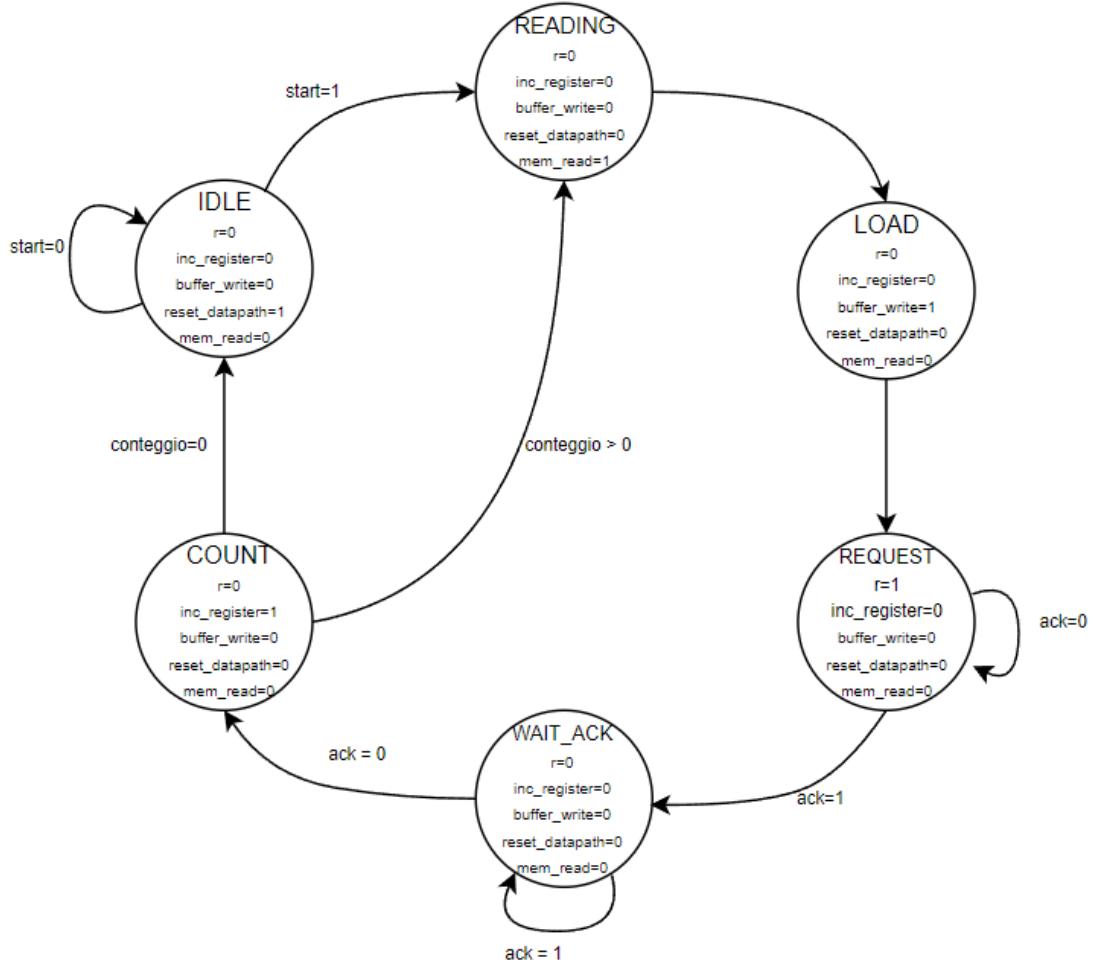


Figura 7.6: Unità di controllo del trasmettitore

Il trasmettitore A parte dallo stato di IDLE in cui è alto il segnale di reset dell'unità operativa (in particolare del contatore e del registro del nodo). Finché il segnale di start è basso l'automa permane nello stato di IDLE. Se al successivo rising edge del clock il segnale di start risulta alto, allora l'automa transita nello stato di lettura READING in cui sono bassi tutti i segnali di uscita tranne mem_read, ovvero è abilitata la lettura di una stringa di bit dalla memoria ROM, in particolare all'indirizzo (addr) determinato dal contatore (che è inizialmente resettato a valore nullo). Al successivo rising edge del clock l'automa passa direttamente nello stato di LOAD nel quale è alto unicamente il segnale buffer_write che determina l'abilitazione della scrittura sul registro buffer di interfaccia con il

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

bus: all'interno di tale registro sarà caricata dunque la stringa di bit in uscita dalla ROM abilitata ed indirizzata opportunamente nello stato precedente. Al successivo rising edge del clock l'automa transita nello stato di REQUEST, dunque, come specificato precedentemente, è necessario che sia alto il segnale ready (r) per comunicare al ricevitore che può prelevare il dato dal bus. In tale stato l'automa permane finché è basso il segnale di ack (il quale indica terminazione dell'elaborazione da parte dello slave). Se al successivo rising edge del clock ack è alto, l'automa transita in WAIT_ACK dove attende che l'ack ritorni nuovamente basso; a questo punto viene incrementato il contatore ponendo il segnale di controllo inc_register pari ad 1 ritornando poi al successivo colpo di clock o nello stato IDLE qualora le stringhe da inviare fossero terminate (counter ritorna a 0) oppure in quello di READING procedendo alla trasmissione della successiva stringa.

Per quanto riguarda il nodo ricevitore B, esso parte dallo stato di IDLE, alla pressione del pulsante visual_b transita nello stato VISUAL per la visualizzazione del contenuto della memoria interna, in cui viene abilitato il visore con enable_visore e si pone lo switch_mode pari ad 1 per consentire l'indirizzamento della stessa tramite gli switch configurati. Qualora nello stato IDLE, invece, il segnale ready venga posto uguale ad 1 (Inizio della comunicazione) il nodo transita nello stato di PRELIEVO in cui vengono prelevate le due stringhe contenute rispettivamente nel buffer di ricezione (msg_read) e nella memoria interna (mem_read). Al successivo colpo di clock B si troverà nello stato di CALC in cui a partire dalle stringhe appena prese provvede ad effettuare la somma. Nel prossimo colpo di clock, in cui lo stato diviene SAVE, viene effettuato il salvataggio della somma appena calcolata nella memoria (mem_write). A questo punto al successivo colpo di clock il nodo transita nello stato di INCR per incrementare il contatore necessario all'indirizzamento della memoria passando poi, al prossimo rising edge del clock, nello stato ACK_OUT in cui semplicemente si pone il segnale ack

ad 1 e qui si permane fino a quando il segnale di ready non viene posto pari a 0, per transitare infine nuovamente nello stato IDLE pronti a ricevere la stringa successiva.

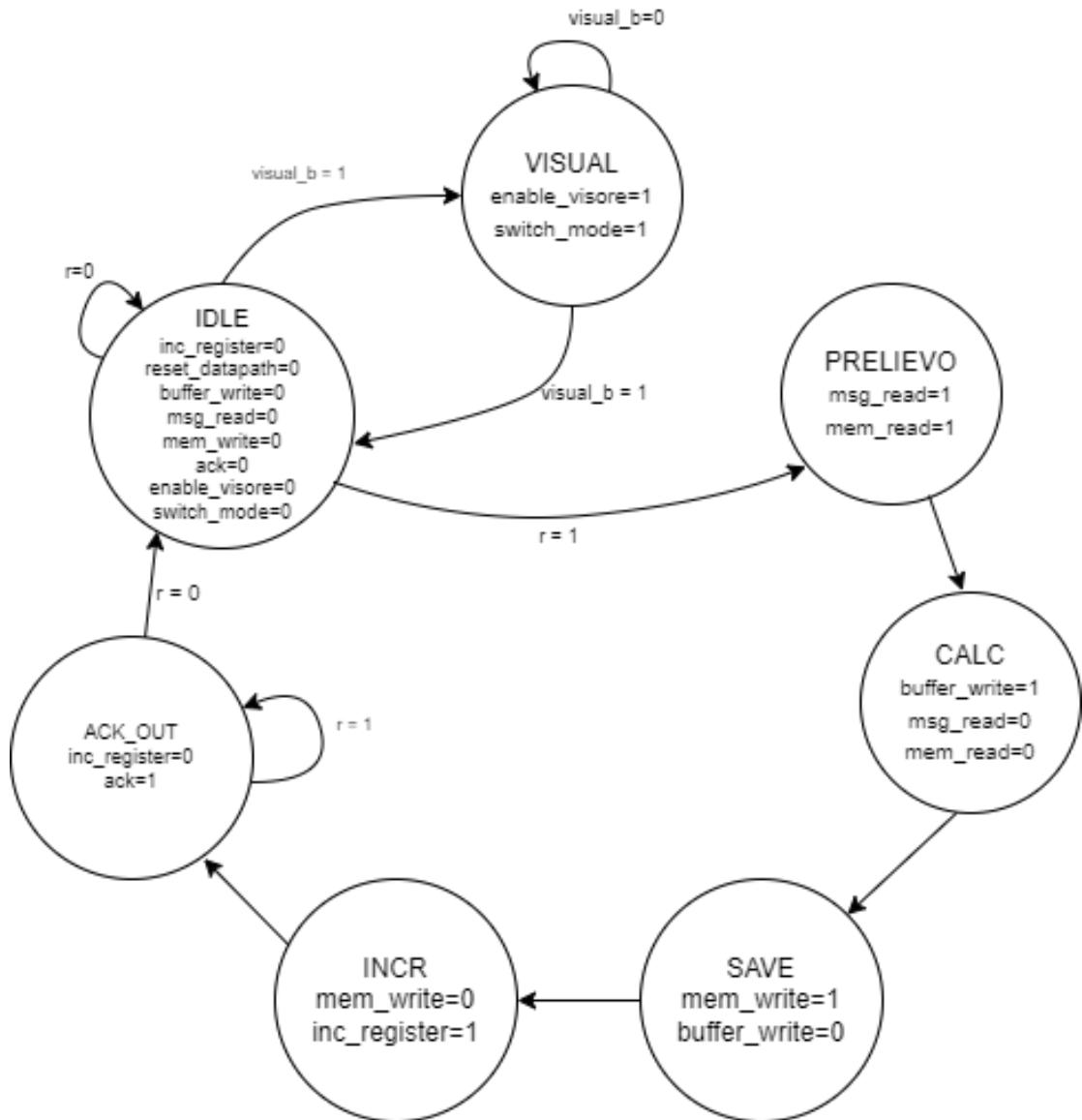


Figura 7.7: Unità di controllo del ricevitore

7.4 Implementazione in VHDL

Passiamo ora all'implementazione in VHDL di quanto descritto nelle sezioni precedenti. Si espone di seguito il top module: SistemaHandshaking. La sua interfaccia

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

è la seguente:

```
entity SistemaHandshaking is port(
    clk : in std_logic;
    start : in std_logic;
    in_switch : in std_logic_vector(2 downto 0);
    visual_b : in std_logic;
    anodes : out std_logic_vector(7 downto 0);
    cathodes : out std_logic_vector(7 downto 0));
end SistemaHandshaking;
```

Come detto si è provveduto alla sua implementazione seguendo un approccio strutturale utilizzando come componenti i due nodi A e B precedentemente presentati, oltre al Button Debouncer per "ripulire" il segnale start e visual_b e il contatore per la definizione delle 2 basi dei tempi necessarie alla temporizzazione dei due nodi.

```
architecture Structural of SistemaHandshaking is

    signal data_out : std_logic_vector(15 downto 0);
    signal r : std_logic;
    signal ack: std_logic;
    signal clk1 : std_logic;
    signal clk2 : std_logic;
    signal cleared_visual : std_logic;
    signal cleared_start : std_logic;

begin
    c1 : entity work.counter generic map(
        M=>2)
        port map(
            clk=>clk2, rst=>'0',
            count=>'1', load=>'0', clk_out=>clk1,
            parallel_input=>(others=>'0'));

```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
c2 : entity work.counter generic map(
    M=>3)

    port map(
        clk=>clk1, rst=>'0', count=>'1',
        load=>'0', clk_out=>clk2,
        parallel_input=>(others=>'0'));

debs : entity work.ButtonDebouncer generic map(
    CLK_PERIOD=>10, btn_noise_time=>650000000)
    port map(
        clk=>clk, reset=>'0',
        btn=>start, cleared_btn=>cleared_start);

debv : entity work.ButtonDebouncer generic map(
    CLK_PERIOD=>10, btn_noise_time=>650000000)
    port map(
        clk=>clk, reset=>'0',
        btn=>visual_b, cleared_btn=>cleared_visual);

a : entity work.nodoa port map(
    clk=>clk1, data_out=>data_out,
    start=>cleared_start, ack=>ack, r=>r);

b : entity work.nodob port map(
    clk=>clk2, msg=>data_out,
    ack=>ack, r=>r,
    visual_b=>cleared_visual, anodes=>anodes,
    cathodes=>cathodes, in_switch=>in_switch
);

end Structural;
```

Fondamentale è però il mapping effettuato per i due nodi. Infatti per il nodo A il segnale di clock viene mappato su clk1 (uscita di un contatore modulo 2), l'uscita

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

contenente la stringa da trasmettere è posta in un signal interno "data_out". Il segnale di ack viene anch'esso assegnato ad un segnale interno sul quale sarà successivamente mappato quello di uscita del nodo B. Per il nodo B, invece come segnale di clock viene utilizzato clk2 (uscita di un contatore modulo 3), sul "msg" di ingresso viene mappato il segnale "data_out" precedentemente assegnato, e come ingresso "r" il corrispondente segnale d'uscita del nodo A;

Il codice relativo all'interfaccia del nodo A è quello riportato di seguito:

```
entity NodoA is port (--MASTER
    clk : in std_logic;
    data_out : out std_logic_vector(15 downto 0);
    start : in std_logic;
    ack : in std_logic;
    r : out std_logic);
end NodoA;
```

In particolare vediamo che in ingresso presenta il segnale di clock clk per la temporizzazione, un segnale di start per l'inizio della comunicazione, un segnale di ack per la ricezione del segnale di completamento dell'elaborazione da parte del nodo slave. In uscita invece dispone di un segnale r di ready per comunicare al nodo slave la disponibilità del dato sul bus ed un vettore di 16 bit rappresentanti il dato da trasmettere. Anche in questo caso è stato seguito un approccio strutturale e, come è possibile osservare dal codice mostrato di seguito, esso risulta composto da un datapath e una unità di controllo.

```
architecture Structural of NodoA is
component datapath_a is port(
    clk : in std_logic;
    inc_register : in std_logic;
    rst : in std_logic;
    buffer_write : in std_logic;
    mem_read : in std_logic;
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
data_out : out std_logic_vector(15 downto 0);
conteggio : out std_logic_vector(2 downto 0));
end component;

component ControlUnit_A is port(
    clk : in std_logic;
    start : in std_logic;
    ack : in std_logic;
    r : out std_logic;
    conteggio : in std_logic_vector(2 downto 0);
    inc_register : out std_logic;
    buffer_write : out std_logic;
    mem_read : out std_logic;
    reset_datapath : out std_logic);
end component;

signal inc_register : std_logic;
signal buffer_write : std_logic;
signal mem_read : std_logic;
signal conteggio : std_logic_vector(2 downto 0);
signal reset_datapath : std_logic;

begin

a_controlunit : entity work.controlunit_a port map(
    clk=>clk, start=>start, ack=>ack, r=>r, conteggio=>conteggio,
    inc_register=>inc_register, buffer_write=>buffer_write,
    reset_datapath=>reset_datapath, mem_read=>mem_read);

a_datapath : entity work.Datapath_a port map(
    clk=>clk, inc_register=>inc_register,
    rst=>reset_datapath, buffer_write=>buffer_write,
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
    data_out=>data_out, mem_read=>mem_read, conteggio=>conteggio);  
  
end Structural;
```

In particolare è stato necessario definire i seguenti i segnali di controllo (signal) scambiati tra il datapath e la control unit : "inc_register" per gestire l'incremento del contatore, "buffer_write" per la scrittura all'interno del registro buffer di interfaccia con il bus, "mem_read" per la lettura della sequenza di bit da prelevare dalla memoria ROM, "conteggio" in ingresso all'unità di controllo per determinare la fine della trasmissione delle N stringhe tra i due nodi, reset_datapath per resettare i componenti interni.

Il codice relativo all'interfaccia del nodo B viene riportato di seguito:

```
entity NodoB is port(  
    clk : in std_logic;  
    msg : in std_logic_vector(15 downto 0);  
    ack : out std_logic;  
    r : in std_logic;  
    visual_b : in std_logic;  
    in_switch : in std_logic_vector(2 downto 0);  
    anodes : out std_logic_vector(7 downto 0);  
    cathodes : out std_logic_vector(7 downto 0));  
end NodoB;
```

Questo presenta il segnale di clock, il vettore di bit trasmesso dal nodo A, il segnale di ack per gestire la terminazione della comunicazione, l'ingresso "r" che segnala la presenza del dato sul bus, visual_b per il passaggio allo stato di visual, gli switch per indirizzare la memoria, gli anodi e i catodi per la visualizzazione del contenuto di questa sul componente visore. Ancora una volta l'approccio seguito prevede un'astrazione strutturale.

```
architecture Structural of NodoB is
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
signal inc_register : std_logic;
signal buffer_write : std_logic;
signal msg_read : std_logic;
signal mem_read : std_logic;
signal mem_write : std_logic;
signal enable_visore : std_logic;
signal switch_mode : std_logic;
```

I segnali interni sono gli stessi definiti e descritti già in precedenza per il nodo A ai quali però si aggiunge: msg_read per la lettura dal registro buffer, mem_write per la scrittura nella memoria interna del risultato di elaborazione, enable_visore per l'abilitazione del visore, switch_mode per selezionare o l'uscita del contatore o l'indirizzo impostato tramite gli switch.

```
begin
    datapath : entity work.datapath_B port map(
        clk=>clk, inc_register=>inc_register, rst=>'0',
        buffer_write=>buffer_write, msg_read=>msg_read,
        mem_read=>mem_read, mem_write=>mem_write,
        msg_in=>msg, anodes=>anodes, cathodes=>cathodes,
        in_switch=>in_switch, enable_visore=>enable_visore,
        switch_mode=>switch_mode);

    control : entity work.controlunit_b port map(
        r=>r, clk=>clk, inc_register=>inc_register,
        visual_b=>visual_b, buffer_write=>buffer_write,
        msg_read=>msg_read, mem_read=>mem_read,
        mem_write=>mem_write, ack=>ack,
        enable_visore=>enable_visore, switch_mode=>switch_mode);

end Structural;
```

Come presentato nel paragrafo 7.3 il datapath (figura7.4) del nodo A risulta costituito dai componenti illustrati di seguito. Per ciascuno di essi verrà presentata

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

esclusivamente l'istanziazione essendo componenti già progettati e descritti nei capitoli precedenti.

```
entity Datapath_A is port(
    clk : in std_logic;
    inc_register : in std_logic;
    rst : in std_logic;
    buffer_write : in std_logic;
    mem_read : in std_logic;
    data_out : out std_logic_vector(15 downto 0);
    conteggio : out std_logic_vector(2 downto 0)
);
end Datapath_A;

architecture Structural of Datapath_A is

signal addr : std_logic_vector(2 downto 0);
signal data : std_logic_vector(15 downto 0);

begin
    conteggio <= addr;
    a_rom : entity work.rom port map(
        clk=>clk, reset=>'0', read=>mem_read,
        addr=>addr, data=>data);

    a_counter : entity work.counter generic map(
        M=>8)
        port map(
            clk=>clk, rst=>rst, count=>inc_register, load=>'0',
            parallel_input=>(others=>'0'), Y=>addr);

    a_registro : entity work.registro generic map(
        N=>16)
        port map(
            clk=>clk, rst=>rst, write=>buffer_write,
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
    data_in=>data, data_out=>data_out);  
  
end Structural;
```

In particolare vengono utilizzati una memoria ROM, un contatore per il suo indirizzamento e un registro tampone per le stringhe da trasmettere.

Per quanto riguarda l'unità di controllo invece, viene realizzata grazie a due processi secondo quanto descritto dal FSM (7.6) presentato: Il primo process sensibile al clock per l'aggiornamento dello stato definito, il secondo sensibile allo stato che definisce il comportamento da implementare sulla base dello stato corrente e dei segnali di ingresso ricevuti.

```
entity ControlUnit_A is  
port(  
    clk : in std_logic;  
    start : in std_logic;  
    ack : in std_logic;  
    r : out std_logic;  
    conteggio : in std_logic_vector(2 downto 0);  
    inc_register : out std_logic;  
    mem_read : out std_logic;  
    buffer_write : out std_logic;  
    reset_datapath : out std_logic);  
end ControlUnit_A;  
  
architecture Behavioral of ControlUnit_A is  
type stato is (IDLE, READING, LOAD, REQUEST, WAIT_ACK, COUNT);  
signal state : stato := IDLE;  
  
begin  
cu_state : process(clk) begin  
    if(falling_edge(clk)) then  
        if(state=IDLE and start='1') then state<=READING;
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
    elsif(state= READING) then state <=LOAD;
    elsif(state=LOAD) then state<= REQUEST;
    elsif(state=REQUEST and ack='1') then state<=WAIT_ACK;
    elsif(state=WAIT_ACK and ack='0') then state <= COUNT;
    elsif(state=COUNT and conv_integer(conteggio)=0) then state<=IDLE;
    elsif(state=COUNT and conv_integer(conteggio)>0) then state<= READING;
    end if;
end if;
end process;

cu_out : process(state) begin
    r <= '0';
    inc_register <= '0';
    buffer_write <= '0';
    reset_datapath <= '0';
    mem_read <= '0';
case state is
    when IDLE =>
        reset_datapath <= '1';
    when READING =>
        mem_read <= '1';
    when LOAD =>
        buffer_write <= '1';
    when REQUEST =>
        r <= '1';
    when COUNT =>
        inc_register <= '1';
end case;
end process;

end Behavioral;
```

Anche per il datapath del nodo B 7.5 l'approccio seguito è stato strutturale.

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
entity Datapath_B is port(
    clk : in std_logic;
    inc_register : in std_logic;
    rst : in std_logic;
    buffer_write : in std_logic;
    msg_read : in std_logic;
    mem_read : in std_logic;
    mem_write : in std_logic;
    enable_visore : in std_logic;
    switch_mode : in std_logic;

    msg_in : in std_logic_vector(15 downto 0);
    in_switch : in std_logic_vector(2 downto 0);--ingresso dello switch

    anodes : out std_logic_vector(7 downto 0);
    cathodes : out std_logic_vector(7 downto 0));
end Datapath_B;

architecture Structural of Datapath_B is

    signal msg_out : std_logic_vector(15 downto 0);
    signal adder_out : std_logic_vector(15 downto 0);
    signal tamp_out : std_logic_vector(15 downto 0);
    signal mem_out : std_logic_vector(15 downto 0);
    signal count_out : std_logic_vector(2 downto 0);
    signal mux_out : std_logic_vector(2 downto 0);
    signal visore_data : std_logic_vector(31 downto 0);
    signal abilita_visore : std_logic_vector(7 downto 0);
begin
    msg_reg : entity work.registro generic map(
        N=>16)
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
port map(
    clk=>clk, rst=>rst, write=>msg_read,
    data_in=>msg_in, data_out=>msg_out);

tamp_reg: entity work.registro generic map(
    -registro tampone per salvare il risultato della addizione
    N=>16)
port map(
    clk=>clk, rst=>rst, write=>buffer_write,
    data_in=>adder_out, data_out=>tamp_out
);

addizionatore : entity work.adder port map(
    s1=>msg_out, s2=>mem_out, o=>adder_out);

mem : entity work.memory port map(
    rst=>rst, read=>mem_read, write=>mem_write,
    clk=>clk, input=>tamp_out, output=>mem_out,
    address=>mux_out);

contatore : entity work.counter generic map(
    M=>8)
port map(
    clk=>clk, rst=>rst, count=>inc_register,
    load=>'0', parallel_input=>(others=>'0'),
    y=>count_out);

mux : entity work.multiplexer generic map(
    N=>3)
port map(
    in1=>count_out, in2=>in_switch, o=>mux_out, s=>switch_mode);
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
visore_data<= x"0000"&mem_out;
```

Da notare che per far si che la visualizzazione del visore fosse corretta è stato necessario concatenare una stringa di zeri con mem_out.

```
with enable_visore select
  abilita_visore <= x"00" when '0',
                x"0F" when '1';

  display : display_seven_segments port map(
    clk=>clk, rst=>'0', value=>visore_data,
    enable=>abilita_visore, dots=>(others=>'0'),
    anodes=>anodes, cathodes=>cathodes);

end Structural;
```

I componenti utilizzati in questo caso sono un registro(msg_reg) per la ricezione della stringa dal nodo A, un addizionatore realizzato in maniera puramente comportamentale con un ulteriore registro tampone in uscita, una memoria atta a contenere i risultati dell'elaborazione, un contatore per l'indirizzamento della memoria stessa, un multiplexer che consente di selezionare la modalità di visualizzazione, e il componente visore per visualizzare il contenuto della memoria.

L'unità di controllo del nodo B viene definita sulla base della descrizione presentata nel paragrafo precedente (7.7), e ancora una volta tramite l'impiego di due process uno sensibile al clock e uno allo stato implementa il comportamento desiderato pilotando opportunamente la corrispondente unità operativa.

```
entity ControlUnit_B is port(
  r : in std_logic;
  clk: in std_logic;
  visual_b : in std_logic;

  inc_register : out std_logic;
  reset_datapath : out std_logic;
```

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

```
buffer_write : out std_logic;
msg_read : out std_logic;
mem_read : out std_logic;
mem_write : out std_logic;
enable_visore : out std_logic;
switch_mode : out std_logic;

ack : out std_logic);
end ControlUnit_B;

architecture Behavioral of ControlUnit_B is
type stato is (IDLE, PRELIEVO, CALC, SAVE, INCR, ACK_OUT, VISUAL);
signal state : stato := IDLE;
begin
cu_state : process(clk) begin
    if(falling_edge(clk)) then
        if(state=IDLE and r='1') THEN state<=PRELIEVO;
        elsif(state=IDLE and visual_b='1') then state <= VISUAL;
        elsif(state=VISUAL and visual_b='1') then state <= IDLE;
        elsif(state=PRELIEVO) then state <= CALC;
        elsif(state=CALC) then state <= SAVE;
        elsif(state=SAVE) then state <= INCR;
        elsif(state=INCR) THEN state <= ack_out;
        elsif(state=ACK_OUT and r='0') then state <= IDLE;
        end if;
    end if;
end process;

cu_out : process(state) begin
    inc_register <= '0';
    reset_datapath <= '0';
    buffer_write <= '0';
    msg_read <= '0';

```

```

        mem_read <= '0';
        mem_write <= '0';
        ack<='0';
        enable_visore <='0';
        switch_mode <='0';

case state is

    when VISUAL =>
        enable_visore <='1';
        switch_mode <='1';

    when PRELIEVO =>
        msg_read <= '1';
        mem_read <= '1';

    when CALC =>
        buffer_write <= '1';
        --scrivo il risultato nel tampone

    when SAVE=>
        mem_write <= '1';

    when INCR=>
        inc_register <= '1';

    when ACK_OUT=>
        ack <= '1';

end case;

end process;

end Behavioral;

```

7.5 Testing simulato

Riportiamo a questo punto il testing effettuato per il sistema complessivo, in cui si simula una interazione tra i due nodi A e B settando opportunamente all'interno del testbench i segnali che regolano la comunicazione. Il comportamento esibito è coerente con quanto progettato ed in particolare le unità di controllo dei due nodi procedono correttamente nella sequenza di stati definiti. All'interno della

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

simulazione si riporta, in pratica, la trasmissione di una singola stringa da A a B con il contenuto della memoria del nodo B per verificare che l'operazione definita sia stata effettuata correttamente e si mostra anche come alzando il segnale visual_b si passi nello stato di visualizzazione della memoria all'indirizzo specificato da in_switch.



Figura 7.8

7.6 Caricamento sulla scheda

Infine si è provveduto alla sintesi del Sistema di Handshake progettato sulla scheda. In particolare i segnali di start e visual_b vengono generati tramite due bottoni, rispettivamente N17 ed M18; mentre per quanto riguarda l'indirizzamento della memoria del nodo B, essendo costituita da 8 stringhe da 16 bit è necessario specificare $\log_2 8 = 3$ bit, e ciò è realizzato tramite i primi tre switch J15, L16 ed M13. Per quanto riguarda la visualizzazione sul Display è stato chiaramente necessario decommentare dal file di constraint le righe relative ad anodi e catodi.

PROGETTO 7. COMUNICAZIONE CON HANDSHAKING

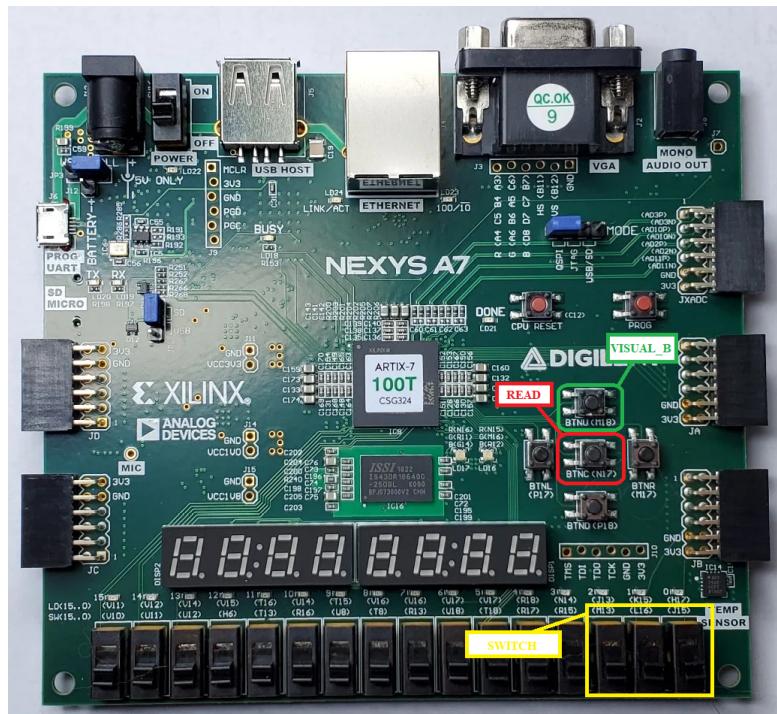


Figura 7.9

Progetto 8

Il processore MIC-1

Il processore MIC-1 è una architettura progettata dallo studioso Andrew S. Tanenbaum come esempio didattico all'interno del libro *Structured Computer Organization*. Questa architettura, seppur nel complesso presenti varie semplificazioni didattiche, fornisce un esempio completo e concreto del funzionamento di un semplice processore general-purpose. Questo mostra, inoltre, l'organizzazione interna di un sistema complesso, in cui la decomposizione funzionale in parte operativa e parte di controllo diventa una necessità assoluta in fase di progettazione.

8.1 Traccia

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

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

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

La traccia di questo esercizio richiede quindi una fase iniziale di analisi del componente allo scopo di comprenderne l'architettura ed il funzionamento ed il

focus relativo a due specifiche istruzioni, andando ad analizzare la sequenza dei segnali di controllo generati dall'unità di controllo a livello microarchitetturale,

8.2 La struttura del processore

Il MIC-1 utilizza un modello di processore a stack, non fornendo nel modello di programmazione registri generali, ma supportando esclusivamente istruzioni con operandi provenienti dallo stack. In questo modo è possibile semplificare notevolmente l'instruction set del processore supportando esclusivamente operazioni con operandi impliciti, ottenendo però maggiori inefficienze causate da continui accessi in memoria.

Il datapath di questa architettura è composto da due bus fondamentali, il B bus connesso alle uscite dei registri e il C bus, che connette l'output della ALU all'ingresso dei registri.

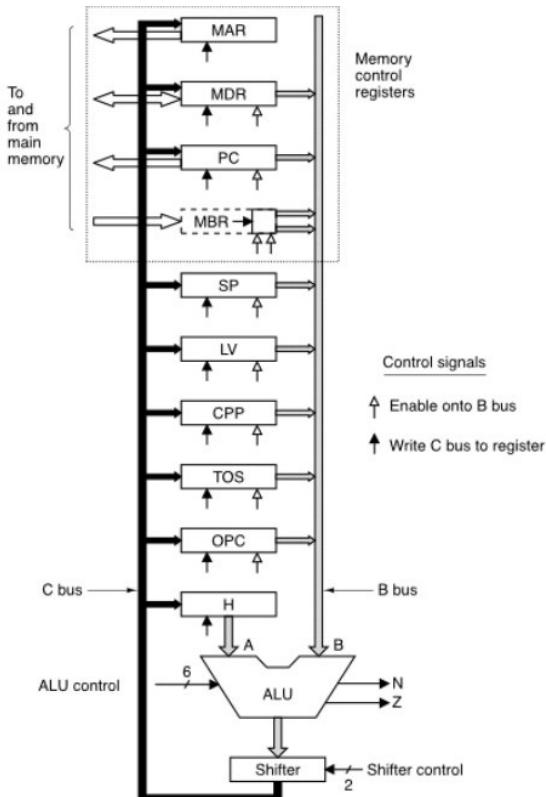


Figura 8.1: Datapath del MIC-1

Come precedentemente descritto, l'architettura non prevede registri generali. I registri più importanti che troviamo all'interno del datapath sono il MAR, l'MDR, il PC e il MBR, utilizzati per l'interfacciamento con la memoria dati e con la memoria istruzioni, lo stack pointer SP e il registro *top of stack* TOS, fondamentali per gestione dello stack, su cui sono pre-caricati gli operandi di tutte le operazioni.

La control unit è, invece, realizzata seguendo il modello **micropogrammato**: le sequenze di controllo sono memorizzate all'interno di una ROM 512x36 bit.

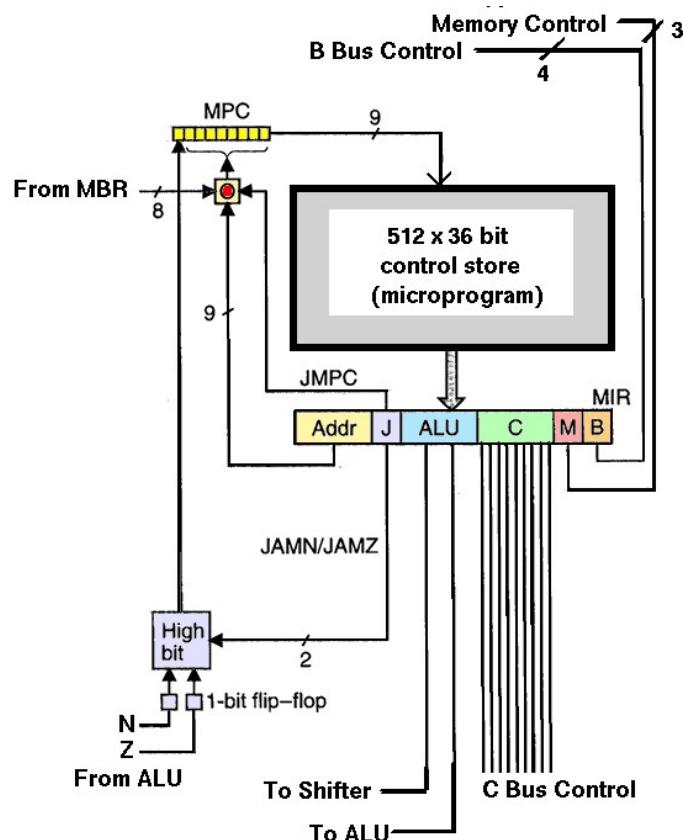


Figura 8.2: Control unit del MIC-1

Le microistruzioni, memorizzate nella control store, oltre ai segnali di controllo per il datapath, contengono il prossimo indirizzo da inserire all'interno del micro-program counter, alcuni bit utilizzati per gestire le condizioni di salto condizionato in corrispondenza dei segnali di stato Z e N forniti dall'ALU e il prelievo delle istruzioni dal MBR, che viene utilizzato, in questa architettura, come instruction

register.

Gli opcode utilizzati per identificare le specifiche istruzioni a livello architetture sono gli entry-point all'interno della control store della sequenza di microistruzioni che le implementano a livello micro-architetturale. All'interno della control unit troviamo, inoltre, della circuiteria addizionale utilizzata allo scopo di decodificare parte delle istruzioni. Sfruttando la proprietà fondamentale secondo cui è possibile per un sol registro alla volta scrivere sul bus B, si può ridurre il numero di bit necessari per memorizzare i segnali di controllo relativi alla scrittura sul bus.

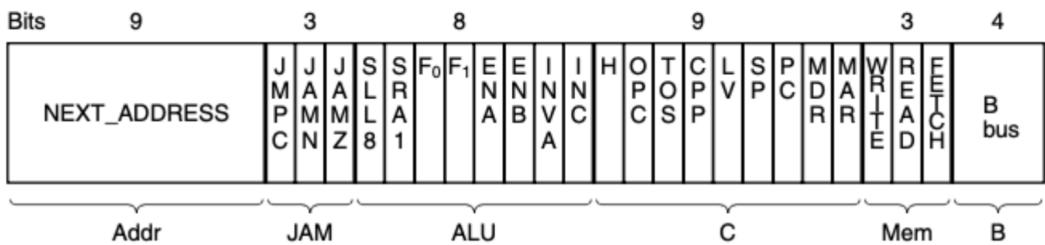


Figura 8.3: Struttura di una micro-istruzione MIC-1.

8.3 Comandi analizzati

8.3.1 BIPUSH

Per il primo punto della traccia abbiamo deciso di analizzare due codici operativi, il bipush e l'if_icmpeq.

Per quanto riguarda il primo, il bipush consente il caricamento di un byte sullo stack. Si tratta di un'istruzione fondamentale: il processore, infatti, seguendo il modello a stack, necessita, per l'esecuzione di moltissimi comandi, di precaricare gli operandi sullo stack. La sequenza di microistruzioni eseguite è la seguente:

```
bipush = 0x10:
```

```
SP = MAR = SP + 1
```

```

PC = PC + 1; fetch
MDR = TOS = MBR; wr; goto main

```

Questa istruzione ha una lunghezza complessiva di due byte: il primo corrisponde all'opcode del bipush, che per quanto detto in precedenza corrisponde all'indirizzo della control-store dove è presente l'entry-point del micro programma 0x10; il secondo, invece, è il valore da caricare sullo stack.

Per come è realizzata la sequenza di esecuzione del microprogramma "main", che implementa la fase di fetch del ciclo del processore, poiché questa anticipa la fetch dell'istruzione successiva, il valore da caricare sullo stack al momento dell'esecuzione del bipush sarà già disponibile all'interno del MDR.

Il valore dello stack pointer è inizialmente incrementato e caricato nel registro MAR. Successivamente, il PC, che attualmente punta all'indirizzo dell'operando dell'istruzione, viene incrementato e viene inizializzata l'operazione di lettura della prossima istruzione tramite il comando fetch, in modo tale che, per la prossima esecuzione di "main", il valore della prossima microistruzione da eseguire sarà correttamente pre-caricato nell'MDR. Il valore dell'operando da aggiungere allo stack, in questo momento salvato all'interno di MBR dalla precedente esecuzione del main, viene salvato in Top of Stack e in MDR. Infine, con l'operazione di wr (write), questo valore verrà memorizzato all'interno dell'indirizzo SP+1 caricato all'interno del MAR, completando l'esecuzione dell'istruzione. Il comando goto main, che troviamo alla fine di ogni micoprogramma, garantisce il corretto proseguimento del ciclo del processore e quindi la corretta esecuzione del programma caricato in memoria.

Abbiamo quindi analizzato il contenuto della RAM contenente un programma che esegue una bipush, in figura 8.10 è riportato la porzione di codice considerata. Poi, tramite l'ambiente di simulazione di Vivado, abbiamo verificato lo stato dei registri del datapath durante l'esecuzione di questa istruzione. L'andamento dei

segnali analizzati è riportato in figura 8.11.

```
-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000000000000000000000010000000",
1 => "00001110000100001000100010000000",
2 => "10100111000001100000000010100001",
3 => "00000000101001110000000000000000",
4 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);
```

Figura 8.4: In giallo i due byte dell’istruzione, in rosso l’opcode relativo 0x10, in verde il valore 0xA che vogliamo caricare sullo stack.

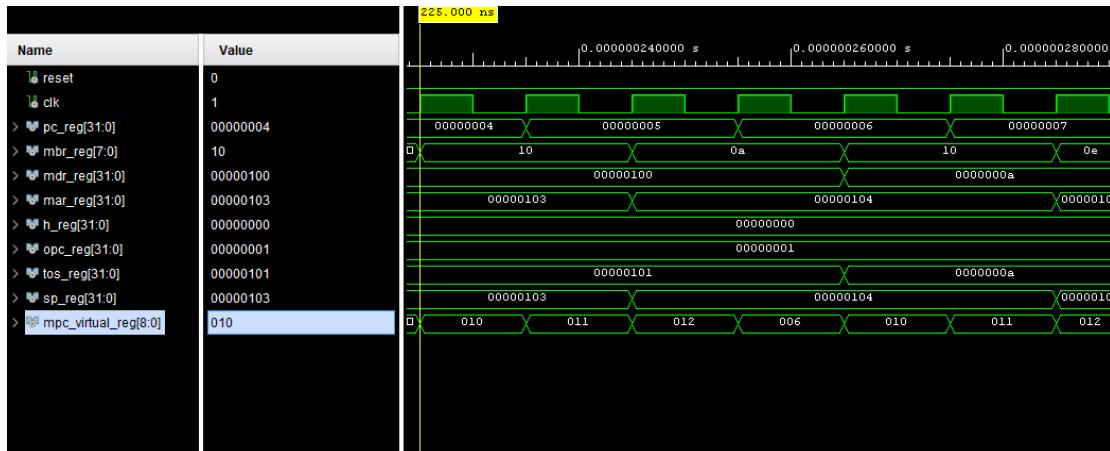


Figura 8.5: Andamento dello stato del datapath durante l’esecuzione della bipush. Osserviamo come il valore dello stack pointer venga incrementato e il valore del registro TOS sia assegnato al valore desiderato 0xA.

8.3.2 IF_ICMPEQ

Il secondo comando analizzato, l’ if_icmpeq, è invece più complesso. Si tratta di un comando di salto condizionato effettuato a fronte di un confronto tra due operandi pre-caricati sullo stack. La microistruzione nel complesso è composta da tre byte, il primo è l’opcode 0xA1 associato all’istruzione, i due byte successivi

sono utilizzati come offset rispetto al PC per calcolare il nuovo indirizzo a cui saltare. Le microistruzioni che implementano la procedura sono le seguenti:

```

if_icmpneq = 0xA1:
    MAR = SP = SP - 1; rd
    MAR = SP = SP - 1
    H = MDR; rd
    OPC = TOS
    TOS = MDR
    Z = OPC - H; if (Z) goto T; else goto F
T:
    OPC = PC - 1; goto goto_cont
F:
    PC = PC + 1
    PC = PC + 1; fetch
    goto main
goto_cont:
    PC = PC + 1; fetch
    H = MBR << 8
    H = MBRU OR H
    PC = OPC + H; fetch
    goto main
main:
    PC = PC + 1; fetch; goto (MBR)

```

Il micro-programma deve inizialmente portare i due operandi, caricati sullo stack, nei registri interni del processore in modo da effettuare la differenza tra questi. Il flag della ALU Z alto, a seguito della differenza, indica che questi risultano

uguali, e quindi bisogna effettuare il salto sovrascrivendo il PC, altrimenti, se resta basso, bisogna continuare con l'esecuzione.

Mentre il secondo operando OP2 è già presente nel processore nel registro TOS, il secondo deve essere caricato. Viene quindi decrementato lo stack pointer e viene letto tramite rd il valore del primo operando OP1. Viene poi decrementato ulteriormente l'SP e nuovamente assegnato il valore al MAR. Questa operazione ha lo scopo di eliminare i due operandi dallo stack e per aggiornare, a seguito dell'operazione, il registro TOS. Viene quindi memorizzato nel registro tampone H il valore letto in precedenza OP1 e si effettua la lettura dell'elemento posto sotto i due operandi che diventerà la nuova cima dello stack e che sarà quindi salvato, a seguito delle operazioni nel registro TOS. Si salva nel registro temporaneo OPC il contenuto attuale di Top of Stack (OP2). Il microprogramma procede con l'aggiornamento del TOS, assegnato al valore appena letto.

Viene poi effettuata la differenza tra il valore contenuto in OPC e quello in H, ovvero rispettivamente OP2 e OP1. Troviamo poi una operazione di salto a livello micro-architetturale: se il bit di stato Z dell'ALU è alto e, quindi, i due operandi sono uguali, si procede con il ramo T, altrimenti con F, il valore del micro-program counter sarà quindi aggiornato di conseguenza dalla logica di controllo esterna presente nell'UC.

Nel ramo T, in OPC viene salvato il valore di PC-1, che corrisponde all'indirizzo dell'istruzione correntemente in esecuzione (infatti la procedura main effettua un incremento anticipato del PC). L'offset è memorizzato su due byte, mentre la prima parte dell'incremento è già stata letta dalla precedente esecuzione di main, è necessario andare a leggere la seconda parte: il microprogramma procede quindi con l'incremento di PC e la corrispettiva fetch, procedendo al prelievo della seconda parte dell'offset. La prima parte di questo offset precedentemente caricata nell'MBR viene shiftata di 8 bit e memorizzata nel registro H. A questo punto, è fatta la OR tra il registro H e la seconda parte dell'offset viene a questo punto

caricata in MBR e il risultato, che corrisponde all'offset complessivo viene salvato in H. Il Program Counter è impostato quindi alla somma tra l'indirizzo dell'istruzione corrente e l'offset appena calcolato contenuto nel registro tampone. Inoltre, viene inizializzata la lettura del valore puntato dal PC in modo tale che questo sia pronto per la prossima esecuzione del main.

Nel ramo F, invece, non deve essere effettuato il salto e quindi il PC è incrementato due volte in modo tale che punti alla prossima istruzione, ignorando i due byte corrispondenti all'offset. Viene, infine, effettuata la fetch in modo da avere l'MBR pronto per la prossima esecuzione del main.

Anche per questo secondo programma abbiamo testato nell'ambiente simulato lo stato del datapath eseguendo un semplice programma che carica sullo stack due valori tramite due bipush e che va poi ad eseguire il comando if_cmpeq.

```
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000000000000000000000010000000",
1 => "0000001100010000000001100010000",
2 => "10100111000001100000010100001",
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
```

Figura 8.6: Il contenuto della RAM, in blu le due bipush, in giallo l'opcode della if_cmpeq, in arancione i due byte di offset.

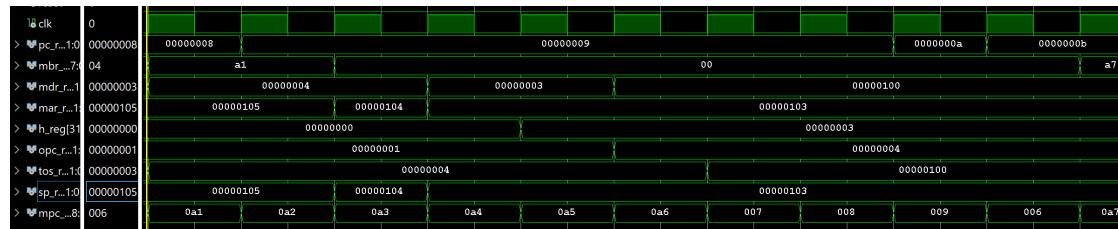


Figura 8.7: Registri del datapath durante l'esecuzione della if_icmpeq, gli operandi sono diversi, non viene effettuato il salto.

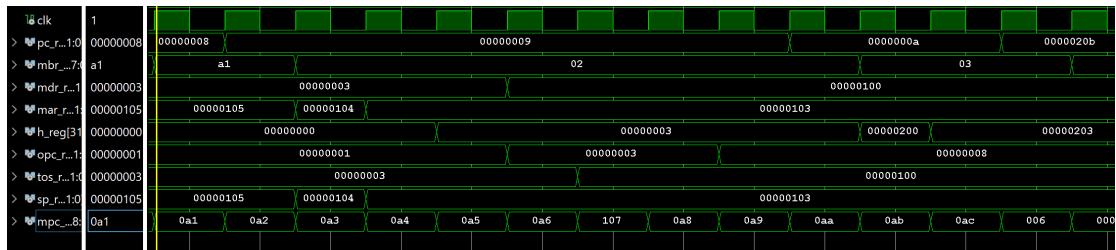


Figura 8.8: Registri del datapath durante l'esecuzione della if_icmpeq, gli operandi sono uguali, il program counter è quindi modificato.

8.4 Modifiche effettuate

Abbiamo deciso di implementare una nuova semplice istruzione, aggiungendo al processore una nuova funzionalità con l'introduzione di un nuovo codice operativo corrispondente ad una versione modificata della iadd, denominata itripadd, sostanzialmente una versione della add a 3 addendi presi dallo stack.

Prima di poter aggiungere un nuovo codice operativo ed il relativo microprogramma abbiamo dovuto però cercare, all'interno della micro-rom uno spazio libero inutilizzato sufficientemente grande da poter contenere il nuovo programma associato all'istruzione. Fortunatamente la control store del MIC-1 prevede alcune zone inutilizzate, abbiamo quindi selezionato il range che va dall'indirizzo 106 all'indirizzo 112.

L'entry point della itripadd, che corrisponde all'opcode utilizzato per la codifica, è quindi 0x6A.

L'istruzione è stata descritta nel linguaggio previsto per il livello micro-architetturale del MIC-1:

itripadd = 0x6A:

MAR = SP = SP - 1; rd

H = TOS

OPC = MDR

MAR = SP = SP - 1; rd

```

H = H + OPC
MDR = TOS = MDR + H; wr; goto main
    
```

Il microprogramma deve inizialmente prelevare i tre operandi dallo stack, ne deve effettuare la somma e deve poi memorizzare il dato calcolato nello stack.

Il valore del terzo operando OP3 è già presente all'intendo del registro TOS, viene quindi decrementato lo stack pointer e, dopo aver caricato con il nuovo valore dello SP il MAR, viene inizializzata l'operazione di lettura dell'operando due OP2. Il valore di OP3 viene quindi portato verso il registro buffer in ingresso all'ALU H. OP2, a questo punto caricato in MDR, viene salvato nel registro temporaneo OPC. Si effettua dunque un ulteriore decremento dello SP ed una conseguente lettura allo scopo di caricare in MDR l'operando OP1. Si effettua la somma tra OP3 ed OP2, posizionati nei registri H ed OPC, si procede poi ad effettuare la somma tra il risultato precedente ed OP1, ottenendo la somma complessiva, che sarà caricata in TOS e nel MDR affinché possa, tramite il comando wr (write), essere inserita nello stack.

E' stata infine effettuata un' operazione di ricompilazione della micro-rom mediante il micro-assemblatore, andando a tradurre queste istruzioni in bit di controllo. La nuova control store sarà quindi inserita all'interno del file .vhdl.

102 => "001100111000000101001000000000000000111",	102 => "001100111000000101001000000000000000111",
103 => "000000110000001111000010000101000000",	103 => "000000110000001111000010000101000000",
104 => "00",	104 => "00",
105 => "00",	105 => "00",
106 => "00",	106 => "00110101100000110110000010010100100",
107 => "00",	107 => "00110110000000101001000000000000111",
108 => "00",	109 => "00110110100000110110000010010100100",
109 => "00",	110 => "0011011110000011110010000000001000",
110 => "00",	111 => "000001000000111100001000010100000",
111 => "00",	112 => "00",
112 => "00",	113 => "00",
113 => "00",	114 => "00",
114 => "00",	
115 => "00",	

Figura 8.9: Il contenuto della micro-rom prima e dopo la compilazione. Il microprogramma, come previsto inizia alla locazione 0x6A (106 in decimale).

Al fine di testare l'istruzione siamo poi andati a modificare manualmente, senza ricorrere all'utilizzo dell'assemblatore i valori salvati nella ram affinché il proces-

PROGETTO 8. IL PROCESSORE MIC-1

sore eseguisse la nuova istruzione. Abbiamo dunque proceduto alla scrittura in un'apposita locazione di memoria dell'opcode della itripadd.

```
--BEGIN WORDS ENTRY
128 => "000000000000000000000000000000000000000000000000000000000000000",
0 => "00000000000000000000000000000000100000000",
1 => "00000010000100000000000011100010000",
2 => "1010011101101010000001100010000",
3 => "000000000000000000000000000000000000000000000000000000000000000",
others => (others => '0')
--END WORDS ENTRY
```

Figura 8.10: Codice del programma composto da tre bipush, in blu, ed una itripadd, in giallo.

Abbiamo poi verificato il funzionamento di questa in ambiente simulato, eseguendo un programma che pre-carica preliminarmente sullo stack 3 byte tramite tre bipush ed effettua poi la itripadd.

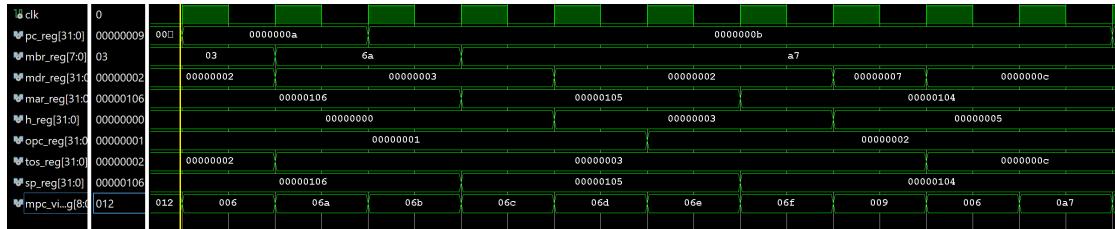


Figura 8.11: Andamento dello stato dei registri del processore durante dell'esecuzione della itripadd.

Progetto 9

Interfaccia seriale

L’interfaccia seriale stabilisce le modalità di comunicazione seriale, al livello più basso del modello ISO/OSI, fra due dispositivi. Come vedremo, la trasmissione seriale è una modalità di comunicazione tra due nodi distinti nella quale i dati sono trasferiti lungo un canale di comunicazione uno di seguito all’altro e giungono sequenzialmente al ricevente nello stesso ordine in cui li ha trasmessi il mittente.

9.1 Traccia

La traccia dell’esercizio, suddivisa in due punti, è la seguente:

- 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;
- come 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 Descrizione soluzione teorica

La trasmissione è caratterizzata da due nodi, un nodo trasmettitore e uno ricevitore, rispettivamente chiamati A e B.

Il nodo A, progettato con un approccio strutturale, è caratterizzato da: un contatore, una ROM, una Control Unit in logica cablata e un dispositivo UART. L'elemento fondamentale di questa struttura è, chiaramente, il dispositivo UART, che, controllato attraverso appositi segnali di controllo della Control Unit, permette la trasmissione dei dati. Nel nostro sistema, N sarà pari a 8: quindi la ROM conterrà 8 locazioni di memoria, ciascuna formata da un byte di dati.

Lo UART (Universal Asynchronous Receiver-Transmitter) è un dispositivo hardware che converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono o viceversa. Esso è un componente che può svolgere la funzione sia di trasmettitore che di ricevitore. Infatti, la sua interfaccia è abbastanza complessa in quanto presenta 7 input e 7 output. Il tipico frame dei dati che riceve/trasmette ha la particolarità di avere lo stato inattivo al livello logico alto, quindi per iniziare la comunicazione bisognerà avere un apposito start bit al livello logico basso, che segnale al ricevitore che sta arrivando un nuovo dato. Un' ulteriore considerazione da fare è sulla progettazione interna del componente, diviso in una parte che funge da trasmettitore e l'altra da ricevitore:

- il trasmettitore prende un byte di informazione parallela trovato sulla porta DBIN e lo converte in un byte di dati trasmessi serialmente sulla porta TXD; inoltre, è caratterizzato dal Write Strobe e dal segnale di underrun error.
- il ricevitore prende in ingresso un byte dei dati seriali trasmessi alla porta

RXD e lo converte in un byte di informazione parallela, posto sulla porta DBOUT; in più, abbiamo il Read Strobe, il segnale di handshaking Read Data Available e tre segnali aggiuntivi che verificano la presenza di errori.

Nel nostro caso, il dispositivo UART prenderà in ingresso sulla porta DBIN l'uscita della ROM, indirizzata dal contatore, e, quando verrà alzato il segnale di Write Strobe dall'unità di controllo, manderà il dato serialmente al dispositivo UART del nodo B attraverso la porta TXD. Inoltre, il segnale di underrun error (TBE) verrà sfruttato come segnale di stato dalla Control Unit, in quanto, quando attivo, indica che non rimane alcun dato da trasmettere.

Il nodo ricevitore B, definito anch'esso strutturalmente, è composto da: una memoria, un contatore, una Control Unit e un dispositivo UART. La memoria, intuitivamente, sarà di tipo read-write, in quanto potremmo accedervi sia nello stato di ricezione che nello stato di visualizzazione. Anche in questo caso sarà indirizzata da un contatore, che a sua volta riceverà il segnale di abilitazione a contare direttamente da un segnale di controllo della Control Unit. La CU del nodo B, in aggiunta a ciò, si occuperà direttamente di inviare i segnali di read e write alla memoria.

Il dispositivo UART nel nodo B si comporterà da ricevitore: non appena il segnale di RDA diventa attivo, prenderà in ingresso il dato serialmente dalla porta RXD e lo presenterà in parallelo sulla porta d'uscita DBOUT, che sarà, quindi, l'input della memoria.

Infine, è da notare la presenza di uno stato aggiuntivo LOAD nella Control Unit del nodo ricevitore, che permette al sistema di memorizzare in modo corretto il dato ricevuto serialmente e solo dopo visualizzarlo in uscita.

9.3 Schematici

9.3.1 UART Component

Il componente UART della Digilent presenta 7 input e 7 uscite, come mostrato in figura 9.1. Come già detto in precedenza, esso svolge sia la funzione di ricevitore che di trasmettitore quindi ci è molto utile per la trasmissione seriale, avendone inserito uno sia nel nodo A, dove riceve in parallelo e trasmette in serie, che nel nodo B, dove riceve in serie e trasmette in parallelo alla memoria. Da notare la presenza di alcuni segnali di errore in uscita: underrun error, sfruttato anche in questo progetto, overrun error, parity error e framing error.

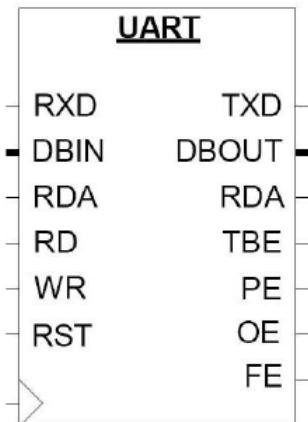


Figura 9.1: Digilent RS232 component

9.3.2 Nodo A

Control Unit

La CU del nodo A è stata sviluppata attraverso un automa a stati finiti. Gli stati possibili, come si può vedere dallo figura 9.2, sono IDLE, WRITE, WAIT_COM e INC. Dopo che il segnale di ingresso start diventa pari a '1', si entra nello stato WRITE, che setta il segnale WR a '1'. In questo stato, quindi, il dispositivo UART è abilitato a trasmettere attraverso la porta TXD. Successivamente, il sistema

dipende da TBE, in particolare dal momento in cui questo segnale diventa '1', che corrisponde al momento in cui non rimane alcun dato da trasmettere.

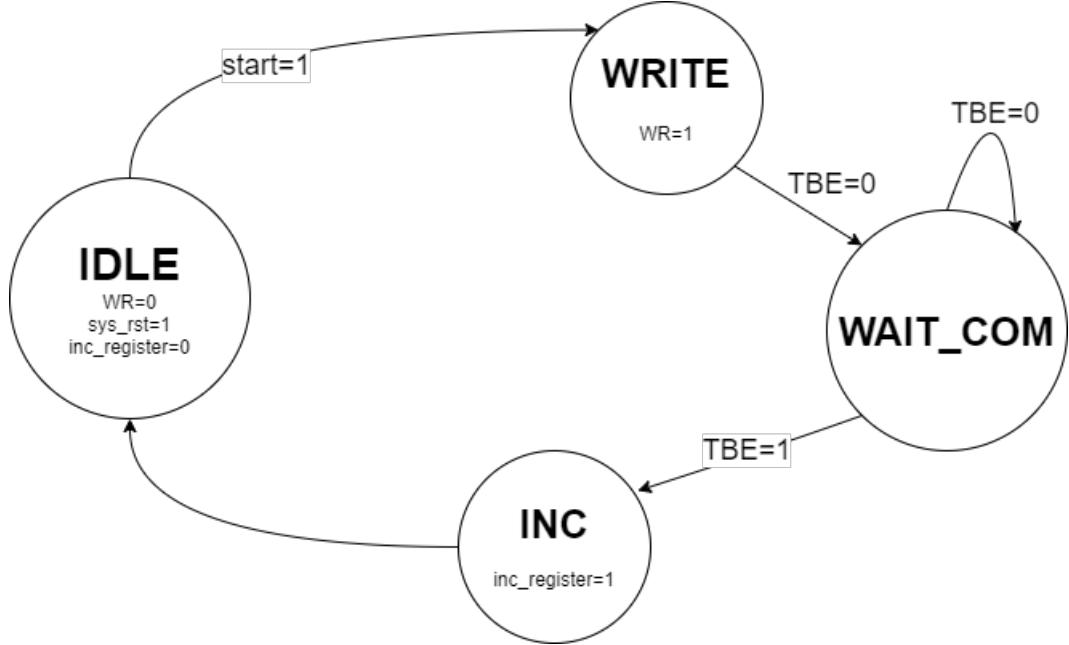


Figura 9.2: Automa della control unit del nodo A

Il nodo A presenta come ingressi un segnale di sincronizzazione clk, un segnale di reset rst e l'input utente, chiamato start, che dà inizio all'evoluzione della macchina. Il sistema presenta un'unica uscita, ovvero la linea dato, che corrisponde alla porta TXD, su cui vengono trasmessi gli 8 bit di una delle 8 stringhe serialmente.

9.3.3 Nodo B

Control Unit

La CU del nodo B prende in ingresso il segnale di sincronizzazione clk, il segnale di reset rst, RDA e start. In uscita presenta i seguenti segnali di controllo, che andranno in ingresso alla parte operativa: rst_usart e RD per il dispositivo UART, mem_in e mem_read per la memoria e, infine, inc_register per il contatore. Partendo dallo stato di IDLE, appena il dato è disponibile nel registro di holding di

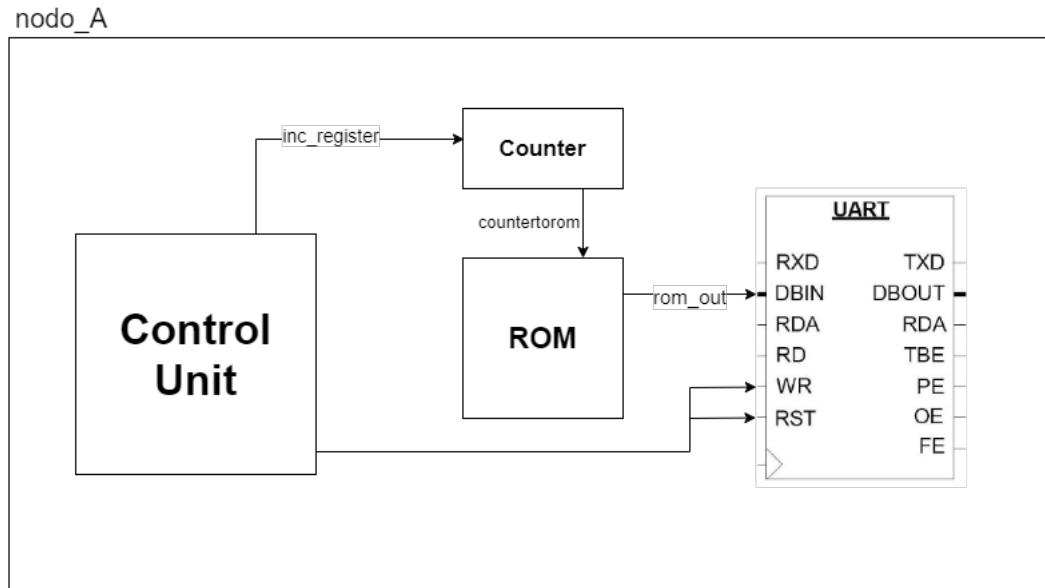


Figura 9.3: Schematico dell'architettura del nodo A

destinazione, si abilita l'accesso in scrittura alla memoria. Successivamente, viene abilitata la modalità in sola lettura per accedere al dato in memoria, indirizzata dal contatore. Infine, non appena viene settato nuovamente il segnale di start a '1' dall'utente, viene abilitato il segnale di count, ovvero inc_register, per il contatore che permetterà di accedere alla prossima locazione della memoria.

E' da osservare il fatto che, fin tanto che RDA è uguale a '0', questo automa rimane nello stato di IDLE. Discorso analogo può essere fatto per lo stato di VISUAL e l'ingresso utente start.

Nel nodo B abbiamo 4 ingressi: uno di sincronizzazione, uno di reset, l'input utente start e RXD, che rappresenta la linea dato seriale proveniente dal nodo A. In uscita avremo l'output, un vettore di 8 bit, chiamato dato. Infine, notiamo che la memoria è una memoria 8x8, costituita quindi da 64 bit in totale, e il contatore è stato implementato assegnando al generic N il valore 8.

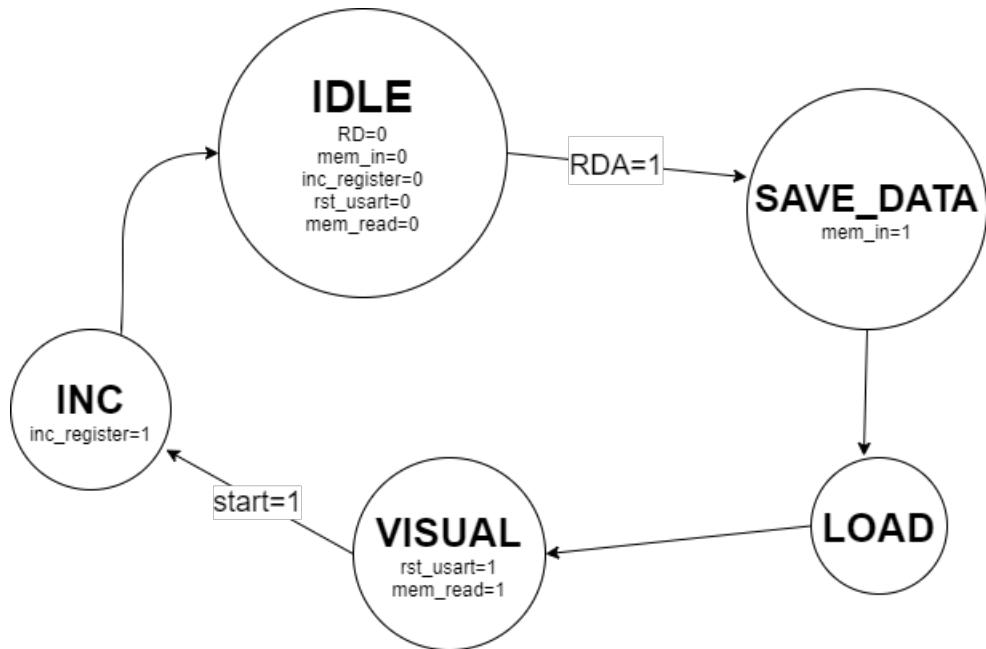


Figura 9.4: Automa della control unit del nodo B

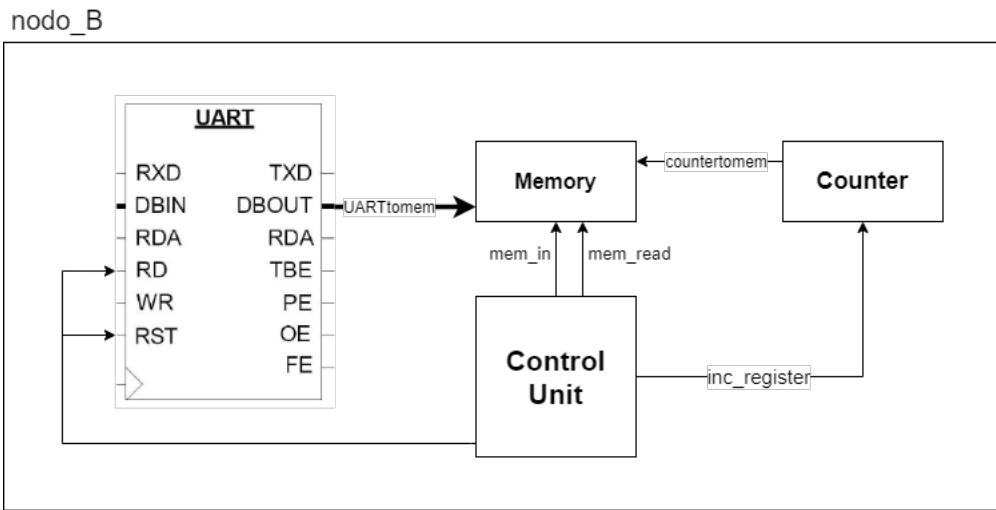


Figura 9.5: Schematico dell’architettura del nodo B

9.4 Implementazione in VHDL

9.4.1 Nodo A

Control Unit

Di seguito è presentata l’implementazione in VHDL della Control Unit del nodo A, sia dell’interfaccia entity che dell’architecture. Il livello di astrazione è chiaramente

comportamentale in quanto stiamo definendo il comportamento della macchina attraverso dei costrutti process:

```
entity A_CU is
    port(
        clk : in std_logic;
        start : in std_logic;
        reset : in std_logic;
        TBE : in std_logic;

        inc_register : out std_logic;
        WR : out std_logic;
        sys_rst : out std_logic
    );
end A_CU;

architecture Behavioral of A_CU is

type state is (IDLE, WRITE, WAIT_COM, INC);
signal stato : state := IDLE;

begin
    state_prc : process(clk) begin
        if(rising_edge(clk)) then
            if(reset = '1') then
                stato <= IDLE;
            elsif(stato = IDLE and start = '1') then
                stato <= WRITE;
            elsif(stato = WRITE and TBE = '0') then
                stato <= WAIT_COM;
            elsif(stato = WAIT_COM and TBE = '1') then
                stato <= INC;
            elsif(stato = INC) then
                stato<=IDLE;
            end if;
        end if;
    end process;
end;
```

```
        end if;
    end if;
end process;

out_prc : process(stato) begin
    WR <= '0';
    sys_rst <= '0';
    inc_register <= '0';
    if(stato = IDLE) then
        sys_rst <= '1';
    elsif(stato = WRITE) then
        WR <= '1';
    elsif(stato=INC) then
        inc_register <= '1';
    end if;
end process;

end Behavioral;
```

nodo_A

Successivamente, abbiamo implementato ad un livello di astrazione strutturale il nodo trasmettitore A. Utilizzando il dispositivo UART della Digilent, abbiamo sfruttato il principio di modularità e riuso per progettare in VHDL il sistema.

```
entity nodo_A is
    port(
        clk : in std_logic;
        rst : in std_logic;
        start : in std_logic;
        TXD : out std_logic
```

```

    );
end nodo_A;

architecture Structural of nodo_A is

signal WR : std_logic;
signal sys_rst : std_logic;
signal rom_out : std_logic_vector(7 downto 0);
signal countertorom : std_logic_vector(2 downto 0);
signal inc_register : std_logic;
signal TBE : std_logic;

begin
    contA : entity work.counter generic map(M=>8)
        port map(
            clk=>clk, rst=>rst, count=>inc_register, load=>'0',
            parallel_input=>(others=>'0'), Y=>countertorom);

    romA : entity work.rom port map(
        clk=>clk, reset=>rst, read=>'1',
        addr=>countertorom, data=>rom_out);

    cu : entity work.A_CU port map(
        clk => clk, start => start, reset => rst,
        TBE => TBE, WR => WR,
        sys_rst=>sys_rst, inc_register=>inc_register
    );

    dp : entity work.UARTcomponent port map (
        TXD=>TXD, RXD=>'1', CLK => CLK,
        DBIN=>rom_out, TBE=> TBE, RD =>'1', WR=>WR, RST=>sys_rst
    );
end Structural;

```

9.4.2 Nodo B

Control Unit

L'implementazione della Control Unit di B è molto simile a quella fatta per la CU di A. Infatti, sono stati sviluppati due process, uno per l'evoluzione degli stati in base allo stato corrente e all'ingresso e uno per le uscite, che dipendono dallo stato in cui si trova il sistema in quel momento.

```
entity B_CU is
    port (
        clk : in std_logic;
        RDA : in std_logic;
        rst : in std_logic;
        start : in std_logic;

        rst_usart : out std_logic;
        RD : out std_logic;
        mem_in : out std_logic;
        mem_read : out std_logic;
        inc_register : out std_logic
    );
end B_CU;

architecture Behavioral of B_CU is

type state is (IDLE, SAVE_DATA, VISUAL, LOAD, INC);
signal stato : state:= IDLE;

begin
    state_prc : process(clk) begin
        if(rising_edge(clk)) then
```

```
if(rst = '1') then
    stato <= IDLE;
elsif(stato = IDLE and RDA = '1') then
    stato <= SAVE_DATA;
elsif(stato = SAVE_DATA) then
    stato <= LOAD;
elsif(statostato= LOAD) then
    stato <= VISUAL;
elsif(stato = VISUAL and start = '1') then
    stato <= INC;
elsif(statostato=INC) then
    stato <= IDLE;
end if;
end if;
end process;

out_prc : process(stato) begin
RD <= '0';
mem_in <= '0';
inc_register <= '0';
rst_usart <= '0';
mem_read <= '0';
if(stato = IDLE) then
    RD <= '0';
elsif(stato = SAVE_DATA) then
    mem_in <= '1';
elsif(stato=VISUAL) then
    rst_usart <= '1';
    mem_read <= '1';
elsif(statostato=INC) then
    inc_register <='1';
end if;
end process;
```

```
end Behavioral;
```

nodo_B

L'implementazione in VHDL, invece, del nodo ricevitore B è la seguente. E' da osservare il fatto che nella istanziazione del dispositivo UART, a RD viene assegnato '0', in quanto è un segnale 0-attivo. Infatti, quando è pari a '0', significa che il dispositivo deve leggere, in questo caso dalla porta RXD:

```
entity nodo_B is
    port (
        clk : in std_logic;
        rst : in std_logic;
        start : in std_logic;
        RXD : in std_logic;
        dato : out std_logic_vector(7 downto 0)
    );
end nodo_B;

architecture Structural of nodo_B is

    signal UARTtomem : std_logic_vector(7 downto 0);
    signal RD : std_logic;
    signal RDA : std_logic;
    signal mem_in : std_logic;
    signal rst_usart : std_logic;
    signal countertomem : std_logic_vector(2 downto 0);
    signal inc_register : std_logic;
    signal mem_read : std_logic;

begin
```

```

mem : entity work.memory generic map(N=>8, M=>8)
      port map(
        clk=>clk, address=>countertomem, write=>mem_in,
        read=>mem_read, input=>UARTtomem,
        output=>dato, rst=>rst);

cont : entity work.counter generic map(M=>8)
      port map(
        clk=>clk, rst=>rst,
        count=>inc_register, load=>'0',
        parallel_input=>(others=>'0'), Y=>countertomem );

uc : entity work.B_CU port map(
  clk => clk, RDA => RDA, rst => rst,
  RD=>RD, mem_in => mem_in,
  rst_usart=>rst_usart, start=>start,
  mem_read=>mem_read, inc_register=>inc_register
);

uart : entity work.UARTcomponent port map(
  RXD => RXD, CLK => CLK, DBIN => (others => '0'),
  DBOUT => UARTtomem, RD => '0', WR => '0',
  RST=>rst_usart, RDA=>RDA
);

end Structural;

```

9.4.3 Top module

Il top module, definito in maniera strutturale, è composto da: nodo trasmettitore A, nodo ricevitore B e un button debouncer associato al segnale d'ingresso

start, per evitare che la pressione del tasto provochi malfunzionamenti o anche trasmissioni troppo rapide, senza che esse possano essere visualizzate correttamente dall'utente.

```
entity uart_on_board is
    port(
        start : in std_logic;
        reset : in std_logic;
        clk : in std_logic;
        leds : out std_logic_vector(7 downto 0)
    );
end uart_on_board;

architecture Structural of uart_on_board is

signal data_line : std_logic;
signal start_b : std_logic;
begin

    startdeb : entity work.ButtonDebouncer generic map(
        clk_period=>10, btn_noise_time=>650000000)
        port map(
            clk=>clk, reset=>'0',
            btn=>start, cleared_btn=>start_b
        );
    n_A : entity work.nodo_A port map(
        clk => clk, rst => reset,
        TXD => data_line, start => start_b
    );
    n_B : entity work.nodo_B port map(
        clk => clk, rst => reset,
        RXD => data_line, dato => leds,
        start=>start_b
    );

```

```

) ;

end Structural;

```

9.5 Testing simulato

Successivamente, abbiamo testato il sistema mediante simulazione. In figura 9.6 possiamo vedere un esempio in cui poniamo start uguale a '1' e di conseguenza gli stati delle Control Unit di A e di B iniziano ad evolvere. Si noti che l'uscita leds viene settata nello stato VISUAL del ricevitore, in cui esso permane fino a quando start non diverrà nuovamente uguale a '1'.

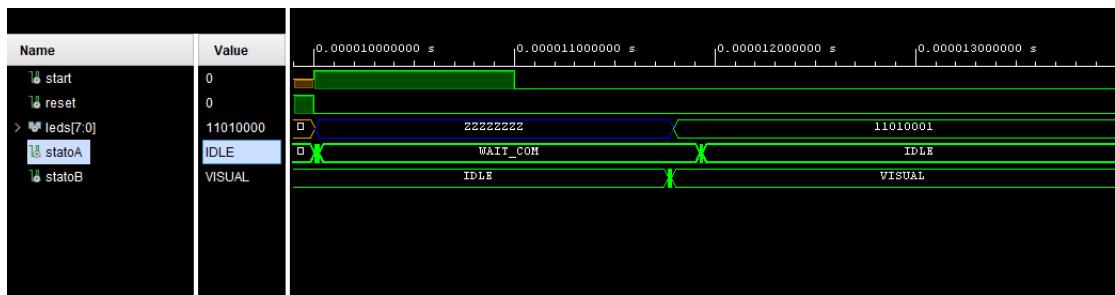


Figura 9.6: Test bench dell'interfaccia seriale

9.6 Caricamento sulla scheda

Infine, abbiamo sintetizzato il progetto su FPGA. Come si vede in figura 9.7, l'ingresso start è stato mappato sul bottone centrale, il reset sul bottone centrale più in alto e i led di uscita, rappresentanti il byte della memoria di B, sui primi 8 led da destra.

PROGETTO 9. INTERFACCIA SERIALE

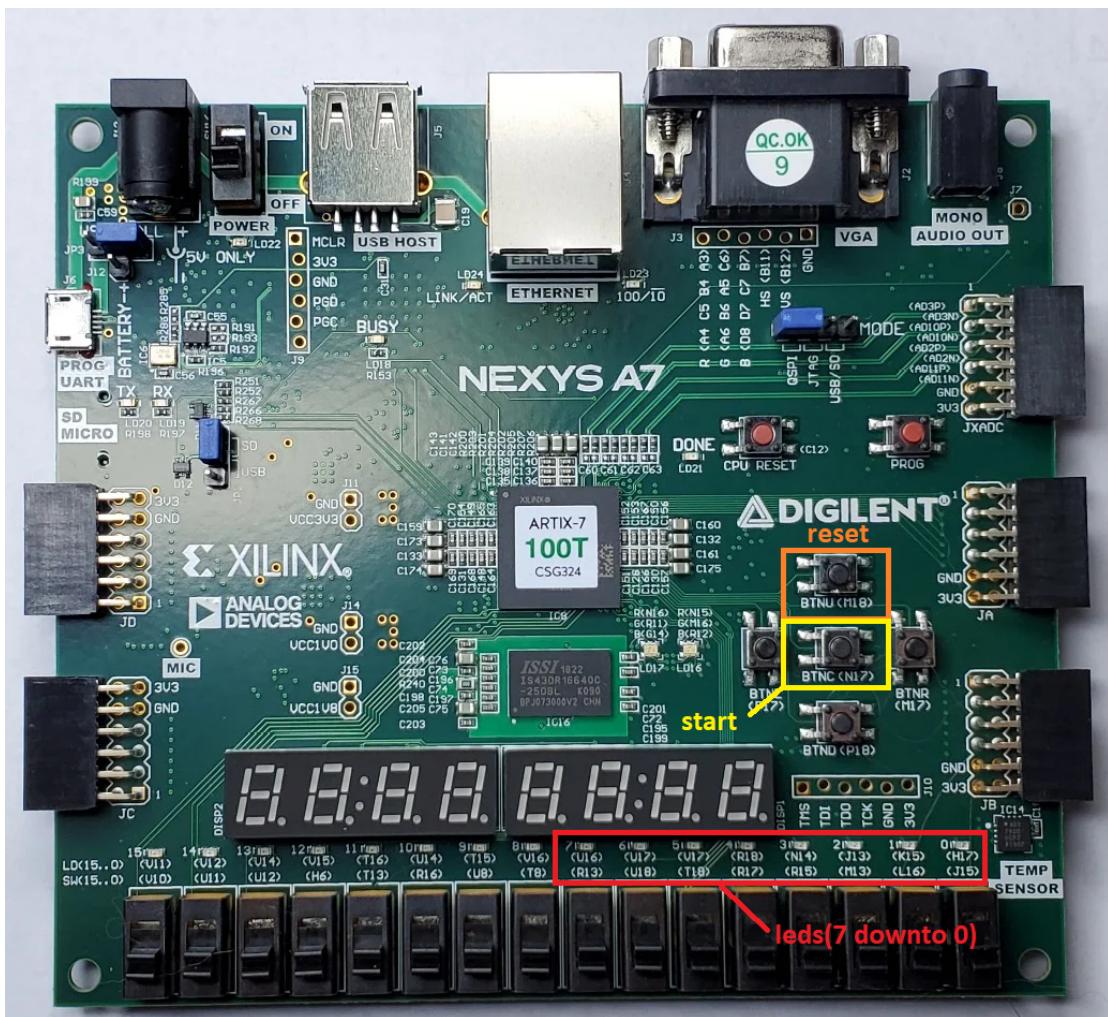


Figura 9.7: Mapping dei segnali su FPGA

Progetto 10

Switch multistato

Uno switch multistadio è un dispositivo utilizzato per connettere N sorgenti con N destinazioni. In particolare è definito multistadio poichè è realizzato con un'architettura composta da più switch a singolo stadio o diretti (vedi figura 10.1).

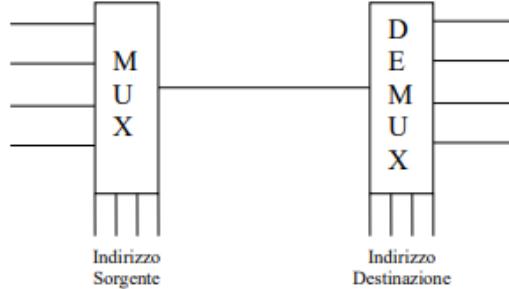


Figura 10.1: Switch a connessione diretta

Tuttavia l'inconveniente隐式的是 che si realizza una mutua esclusione tra i nodi dovuta al fatto che un solo collegamento per volta può essere attivo, infatti in tale schema ad un solo porto per volta è concesso comunicare ed è inoltre difficile avere multiplexer (o demultiplexer) con un numero elevato di ingressi. Per questo si può pensare di utilizzare uno switch che si basi sull'utilizzo di più stadi intermedi. Il modello Omega Network è un tipo di rete di interconnessione multistadio la cui connessione tra i porti si basa sulla tecnica nota come perfect shuffling, la quale fa riferimento al modo con cui delle carte di un mazzo possono

essere "perfettamente" mischiate. In particolare si divide il mazzo in due gruppi uguali e si mischiano le carte in modo tale che la i-esima carta della prima metà viene accoppiata con la i-esima della seconda metà, e così via per $\log_2(N)$ volte fino a quando non si ripristina l'ordine originario. Per la realizzazione di questa tipologia di switch multistadio dunque si utilizzano $\log_2(N)$ stadi di $N/2$ switch a connessione diretta (quindi con 2 ingressi e 2 uscite, come mostrato in figura 10.2)

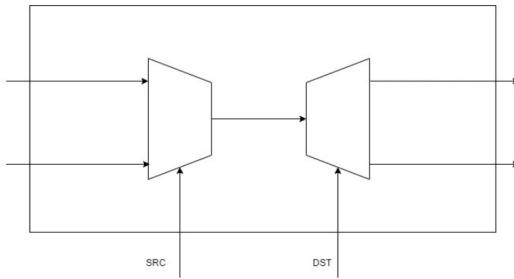


Figura 10.2: Singolo switch in una Omega Network

In questo modo si risolvono i problemi di bassa produttività e difficile realizzabilità del modello con un solo mux/demux, ma non viene risolto il problema dei conflitti: se due nodi di ingresso vogliono parlare con lo stesso nodo di uscita simultaneamente generano un conflitto, ma la contesa può avvenire anche negli stadi intermedi dovuta a nodi diversi che non vogliono comunicare con lo stesso nodo ma che si scontrano durante il percorso.

Le soluzioni possibili per la gestione delle collisioni sono 4:

- Prevenire il conflitto tramite l'assegnazione di una priorità. Il problema di tale soluzione consiste nell'impossibilità di comunicazione simultanea tra due nodi differenti anche quando una loro trasmissione non genererebbe una reale collisione.
- Una seconda soluzione consiste nella perdita del messaggio, ovvero uno dei due segnali viene perso, mentre l'altro procede verso la destinazione.

- Si potrebbe pensare di seguire strade alternative al verificarsi di un conflitto, ma in realtà ciò non è realizzabile anche perché l'algoritmo di routing prevede un percorso obbligatorio;
- All'atto di un conflitto è possibile memorizzare in un buffer il messaggio che l'ha causato, per poi inoltrarlo appena il nodo è libero. Tuttavia ciò complica l'architettura del nodo.

10.1 Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch progettato deve operare come segue:

- a) Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in un 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).
- b) (Opzionale) rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).
- c) (Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.2 Descrizione soluzione teorica

Anche in questo caso per la risoluzione dell'esercizio abbiamo scelto la decomposizione in Unità operativa (10.6) e Unità di controllo (10.7). L'approccio adottato è

ancora strutturale a partire dalla realizzazione del singolo switch come composizione di mux e demux necessari all’indirizzamento. In particolare, viene implementato uno schema a priorità fissa tramite la realizzazione, all’interno dell’unità di controllo, di un arbitro secondo cui uno solo dei nodi (quello a valore più elevato) è abilitato alla trasmissione del messaggio. Nella nostra soluzione è stato previsto che il messaggio trasmesso contenesse al contempo anche l’indirizzo del mittente e del destinatario estrapolati proprio dalla parte di controllo. In questo caso, data la scelta di realizzare un rete a 4 sorgenti e 4 destinazioni, ciascun indirizzo risulta chiaramente composto da soli 2 bit, mentre il messaggio trasmesso è di 4 bit.

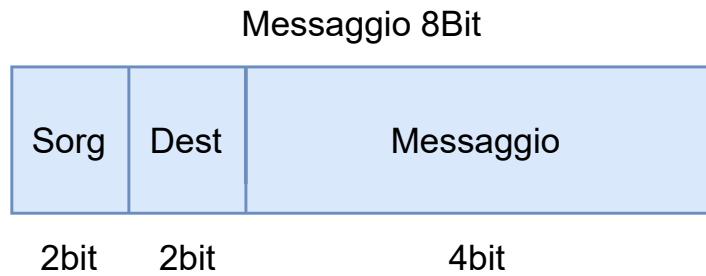


Figura 10.3

10.3 Schematici, componenti

Di seguito riportiamo lo schema complessivo dello switch multistadio:

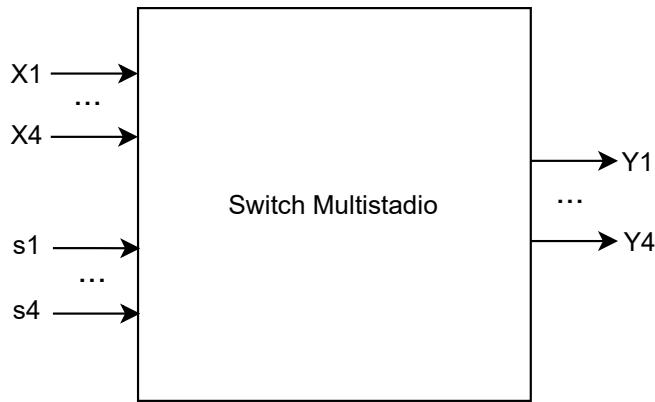


Figura 10.4

Esso accetta in ingresso i messaggi provenienti dalle 4 sorgenti con i segnali di selezione che identificano la sorgente a maggior priorità alla quale è concessa la trasmissione. In uscita riportiamo il segnale trasmesso alla destinazione. Come detto precedentemente esso viene realizzato come la composizione di unità operativa e unità di controllo.

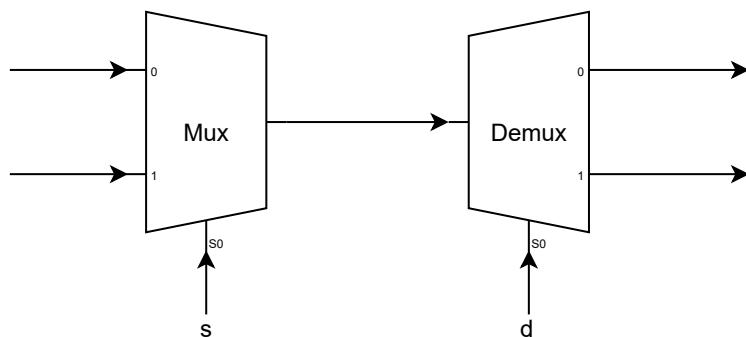


Figura 10.5: Switch

La parte operativa è costituita da 4 switch (figura 10.5) opportunamente interconnessi secondo la tecnica del perfect shuffling.

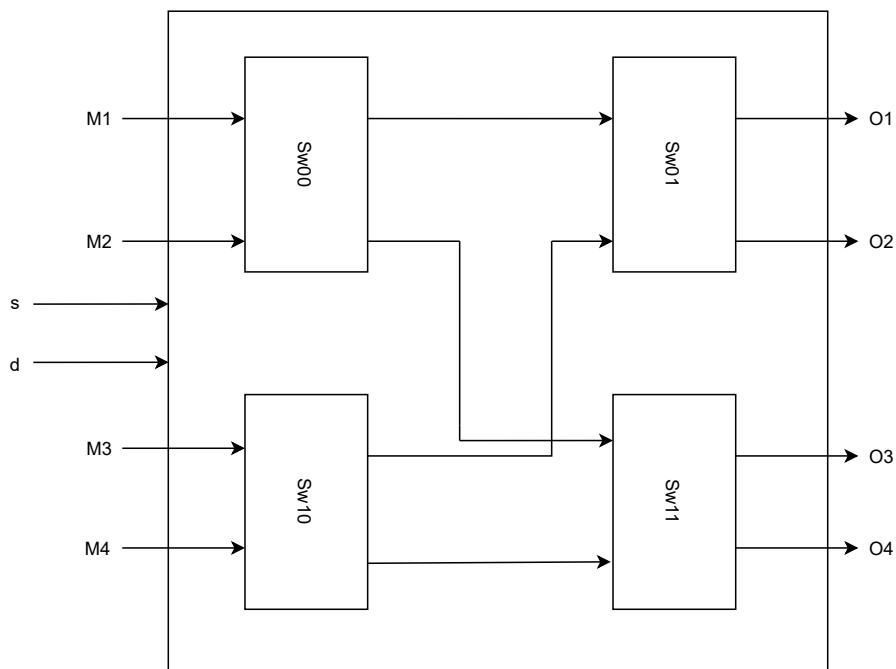


Figura 10.6: Parte Operativa

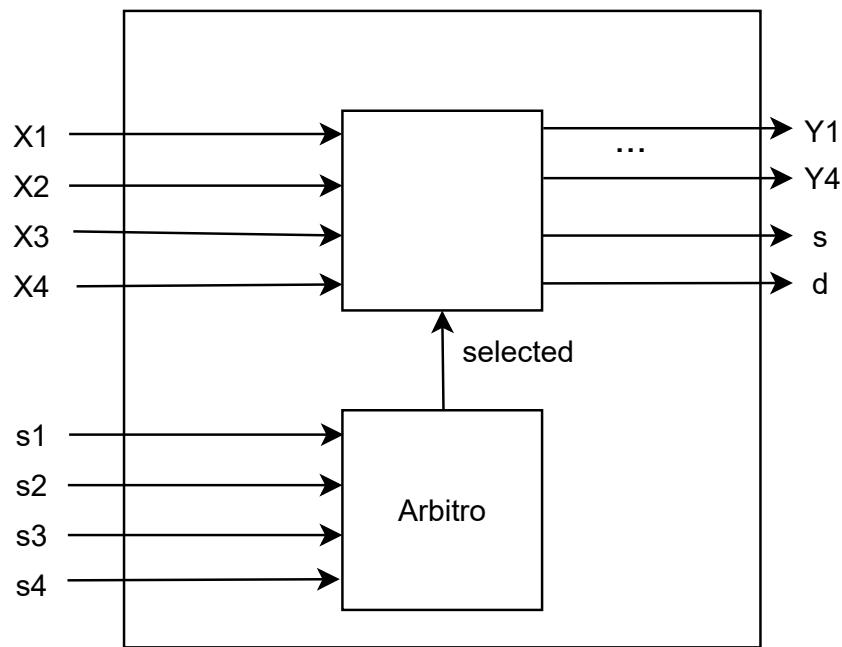


Figura 10.7: Parte Controllo

10.4 Implementazione in VHDL

Procediamo a questo punto all'implementazione dello switch multistadio in VHDL. Partiamo dal blocco elementare, ovvero lo switch, realizzato come composizione di un mux 2:1 e un demux 1:2, come riportato nel codice seguente:

```
entity Switch is
    port(
        in1 : in std_logic_vector(7 downto 0);
        in2 : in std_logic_vector(7 downto 0);

        s : in std_logic;
        d : in std_logic;

        out1 : out std_logic_vector(7 downto 0);
        out2 : out std_logic_vector(7 downto 0)
    );
end Switch;

architecture Structural of Switch is

signal muxtodemux : std_logic_vector(7 downto 0);

begin
    mux : entity work.Mux generic map(
        N => 8
    ) port map(
        in1 => in1, in2 => in2, s => s, o => muxtodemux
    );

    demux : entity work.Demux generic map(
        N => 8
    ) port map(
        ins => muxtodemux, d => d, o1 => out1, o2 => out2
    );

```

```

    );
end Structural;

```

I singoli switch vengono poi composti per realizzare l'unità operativa con approccio strutturale (figura: 10.6), come riportato nel codice seguente:

```

entity UnitaOperativa is
    port (
        s, d : in std_logic_vector(1 downto 0);
        M1, M2, M3, M4 : in std_logic_vector(7 downto 0);
        O1, O2, O3, O4 : out std_logic_vector(7 downto 0)
    );
end UnitaOperativa;

architecture Structural of UnitaOperativa is

signal conn00, conn01, conn10, conn11 : std_logic_vector(7 downto 0);

begin
    sw00 : entity work.Switch port map(
        in1 => M1, in2=> M2, out1=>conn00, out2=>conn01, s=>s(0), d=>d(1)
    );
    sw10 : entity work.Switch port map(
        in1 => M3, in2=> M4, out1=>conn10, out2=>conn11, s=>s(0), d=>d(1)
    );
    sw01 : entity work.Switch port map(
        in1 => conn00, in2=> conn10, out1=>O1, out2=>O2, s=>s(1), d=>d(0)
    );
    sw11 : entity work.Switch port map(
        in1 => conn01, in2=> conn11, out1=>O3, out2=>O4, s=>s(1), d=>d(0)
    );
end Structural;

```

Per la determinazione del percorso da uno specifico nodo sorgente a uno destinazione si valuta ad ogni stadio il singolo bit i-esimo associato all'indirizzo

di destinazione, e viene selezionato il collegamento da percorrere utilizzando il seguente criterio:

- Se il valore del bit è 0, si procede con il ramo di uscita superiore del blocco considerato
- Se il valore del bit è 1, si procede con il ramo di uscita inferiore del blocco considerato

Per realizzare tale criterio è necessario considerare come bit di selezione dei mux e demux relativi al primo stadio rispettivamente il primo bit del vettore sorgente e il secondo di quello destinazione; per gli switch al secondo stadio invece viene considerato il secondo bit del vettore "s" e il primo bit di quello destinazione.

L'arbitro è stato invece descritto a livello dataflow, esso accetta in ingresso 4 bit di selezione e restituisce in uscita un vettore di 2 bit che codifica il porto abilitato, nel nostro caso la massima priorità è data al nodo con valore maggiore, ovvero il nodo 4;

```

entity Arbitro is
    port (
        s1,s2,s3,s4 : in std_logic;
        selected : out std_logic_vector(1 downto 0)
    );
end Arbitro;

architecture Dataflow of Arbitro is
begin
    selected <= "11" when s4 = '1' else
                "10" when s3 = '1' else
                "01" when s2 = '1' else
                "00" when s1 = '1' else

```

```

        " -- ";
end Dataflow;

```

L'unità di controllo invece è costituita semplicemente dall'arbitro precedentemente descritto e un'opportuna logica. Essa si occupa, infatti, non solo di realizzare una rete a priorità tale che uno solo dei nodi trasmetta il messaggio posto in ingresso ma presenta anche il compito di estrapolare dal messaggio di ingresso i relativi indirizzi sorgente e destinazione (rispettivamente i primi 2 bit per la sorgente e i successivi 2 bit per la destinazione) poi trasmessi all'unità operativa per un corretto indirizzamento.

```

entity ControlUnit is
    port (
        X1,X2,X3,X4 : in std_logic_vector(7 downto 0);
        s1,s2,s3,s4 : in std_logic;
        Y1,Y2,Y3,Y4 : out std_logic_vector(7 downto 0);
        s,d : out std_logic_vector(1 downto 0)
    );
end ControlUnit;

architecture Structural of ControlUnit is

signal selected : std_logic_vector(1 downto 0);

begin
    arb : entity work.Arbitro port map(
        s1 => s1, s2 => s2, s3 => s3, s4 => s4, selected => selected
    );

    Y4 <= X4 when selected = "11" else
        (others=>'0');
    Y3 <= X3 when selected = "10" else

```

```

        (others=>'0') ;

Y2 <= X2 when selected = "01" else
        (others=>'0') ;

Y1 <= X1 when selected = "00" else
        (others=>'0') ;

with selected select
    s <= X4(7 downto 6) when "11",
    X3(7 downto 6) when "10",
    X2(7 downto 6) when "01",
    X1(7 downto 6) when "00",
    "—" when others;

with selected select
    d <= X4(5 downto 4) when "11",
    X3(5 downto 4) when "10",
    X2(5 downto 4) when "01",
    X1(5 downto 4) when "00",
    "—" when others;

end Structural;

```

Infine il blocco complessivo SwitchMultistadio viene realizzato, come detto, seguendo un approccio strutturale componendo unità di controllo e unità operativa.

```

entity SwitchMultistadio is
    port(
        X1,X2,X3,X4 : in std_logic_vector(7 downto 0);
        s1,s2,s3,s4 : in std_logic;
        Y1,Y2,Y3,Y4 : out std_logic_vector(7 downto 0)
    );
end SwitchMultistadio;

```

```
architecture Structural of SwitchMultistadio is
```

```

signal s, d : std_logic_vector(1 downto 0);
signal M1,M2,M3,M4 : std_logic_vector(7 downto 0);

begin
    cu : entity work.ControlUnit port map(
        X1=>X1, X2=>X2, X3=>X3, X4=>X4,
        S1=>s1, S2=>s2, S3=>s3, S4=>s4,
        Y1=>M1, Y2=>M2, Y3=>M3, Y4=>M4,
        S=>s, D=>d
    );
    dp : entity work.UnitaOperativa port map(
        S=>s, D=>d,
        M1=>M1, M2=>M2, M3=>M3, M4=>M4,
        O1=>Y1, O2=>Y2, O3=>Y3, O4=>Y4
    );
end Structural;

```

10.5 Testing simulato

Infine riportiamo i test effettuati per i vari componenti realizzati. Consideriamo dapprima il singolo switch: date le selezioni poste in ingresso l'uscita o2 riporta correttamente il valore del primo ingresso.

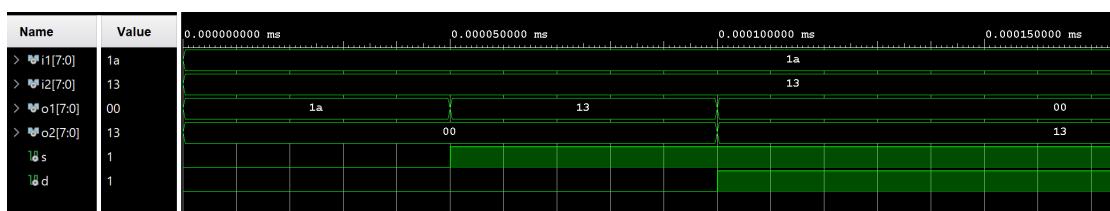


Figura 10.8: Switch_TB

Per quanto riguarda l'Omega Network è stato simulato uno scenario di comunicazione che prevede come sorgente il porto 4 e come destinazione il porto 3

PROGETTO 10. SWITCH MULTISTATO

come si vede chiaramente dalla figura 10.9 in cui l'uscita O3 coincide proprio col messaggio di ingresso del nodo 4.

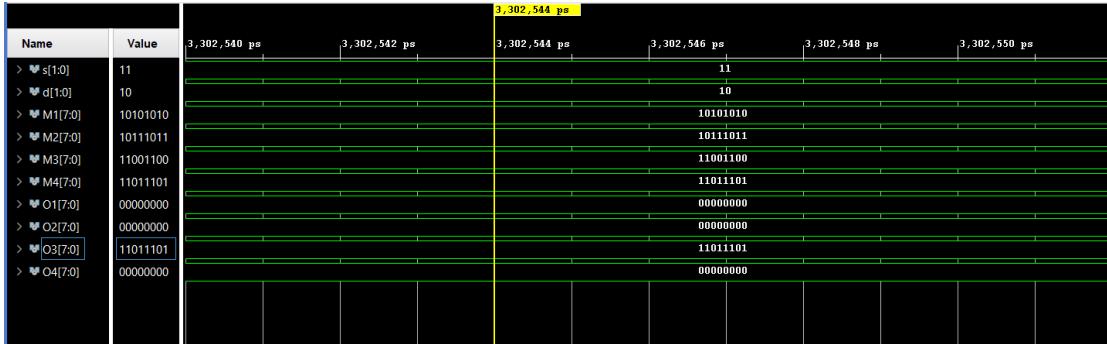


Figura 10.9: OmegaNet_TB

Per il sistema complessivo viene simulato anche lo scenario in cui a voler comunicare siano 2 nodi in concorrenza, ovvero il nodo 00 e 11, chiaramente la rete a priorità risolve la competizione consentendo solo ad uno di essi, ovvero quello a maggiore priorità (nodo 11), di comunicare con la relativa destinazione.

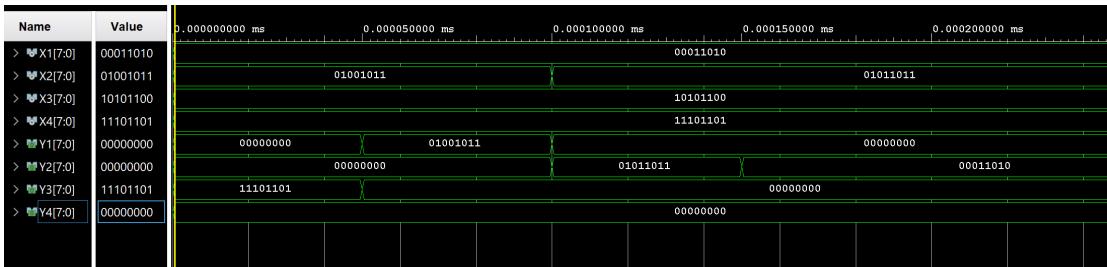


Figura 10.10

Progetto 11

Il moltiplicatore di Robertson

Le macchine aritmetiche sono senza dubbio tra i dispositivi più importanti che troviamo all'interno di un sistema digitale. Consentono, infatti, di effettuare operazioni aritmetiche, requisito fondamentale alla base di ogni calcolatore elettronico.

Le operazioni aritmetiche in generale possono essere realizzate secondo due approcci fondamentali:

- Combinatorio: in questo caso la macchina aritmetica è realizzata come un componente combinatorio. Questo approccio produce tipicamente dispositivi estremamente veloci ma complessi e costosi in termini di componenti utilizzati.
- Sequenziale: la macchina aritmetica è decomposta funzionalmente in parte operativa e di controllo. La parte di controllo gestisce quella operativa allo scopo di implementare un algoritmo di calcolo. Le macchine aritmetiche sequenziali, dovendo lavorare su più fronti di clock, sono tipicamente più lente della loro controparte combinatoria, ma risultano più semplici in termini di componenti utilizzati.

11.1 Traccia

Progettare ed implementare in VHDL una macchina aritmetica sequenziale a scelta fra le seguenti:

- moltiplicatore di Robertson, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
- moltiplicatore di Booth, per effettuare il prodotto di 2 stringhe A e B da 8 bit ciascuna;
- divisore non-restoring, per effettuare la divisione intera fra due stringhe A e B di 4 bit ciascuna;
- divisore restoring, per effettuare la divisione intera fra due stringhe A e B di 4 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.

La traccia richiede di realizzare una tra quattro macchine aritmetiche: per il nostro progetto abbiamo optato per il moltiplicatore di Robertson.

11.2 Descrizione soluzione teorica

Il moltiplicatore di Robertson è una macchina aritmetica sequenziale che realizza l'operazione di moltiplicazione tra due interi, eventualmente relativi, seguendo un algoritmo di calcolo ispirato a quello manuale.

Il risultato di una moltiplicazione può essere infatti facilmente calcolato andando a moltiplicare il moltiplicando per il moltiplicatore, una cifra alla volta, e

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

andando a sommare i risultati correttamente pesati. Nel caso di operandi binari, questo algoritmo può essere facilmente espresso come:

$$P_0 = 0$$

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

Dove, il prodotto con 2^i corrisponde ad uno shift a sinistra di i posizioni ed il prodotto tra Y ed x_i può essere pari a zero se $x_i = 0$ o a Y se $x_i = 1$.

Questo algoritmo, seppur in linea di principio implementabile, non è semplice nella realizzazione in quanto necessita di effettuare shift a sinistra di un numero di posizioni variabili. Modificando l'ordine con il quale si effettuano le operazioni, ad ogni modo, è possibile riorganizzare l'algoritmo in modo da avere ad ogni step uno shift costante a destra:

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

Per come descritto fino a questo punto l'algoritmo funziona esclusivamente per i numeri positivi. Sfruttando però le proprietà della rappresentazione in complementi a due è possibile garantirne il funzionamento anche nel caso di operandi negativi. Per far ciò si sfrutta la proprietà fondamentale della rappresentazione in complementi per la quale la cifra più significativa ha un peso negativo e risulta quindi che:

$$X = 2^0 x_0 + 2^1 x_1 + \cdots + 2^{n-2} x_{n-2} - 2^{n-1} x_{n-1}$$

E' possibile quindi effettuare una moltiplicazione tra numeri relativi moltiplicando le prime $n - 1$ cifre ignorando il segno e procedendo con un passo di correzione in cui si effettua una sottrazione all'ultimo step di calcolo. Viene inoltre utilizzato un flipflop nel quale è memorizzato il valore da porre in ingresso allo

shift register allo scopo di gestire il caso di moltiplicatore negativo.

11.3 Schematici, componenti

Per la progettazione del componente abbiamo iniziato dalla descrizione della parte operativa, che, considerando l'algoritmo da implementare sarà composta da uno shift register di 16 bit (o, come nel nostro caso due shift register da 8 bit in cascata), un multiplexer utilizzato per moltiplicare il bit x_i del moltiplicando con il moltiplicatore Y , un flipflop per la memorizzazione del segno, di un contatore per scandire le fasi dell'algoritmo ed un addizionatore in grado di effettuare somma o differenza su due operandi di 8 bit.

Lo schema complessivo della parte operativa dispositivo è quindi il seguente:

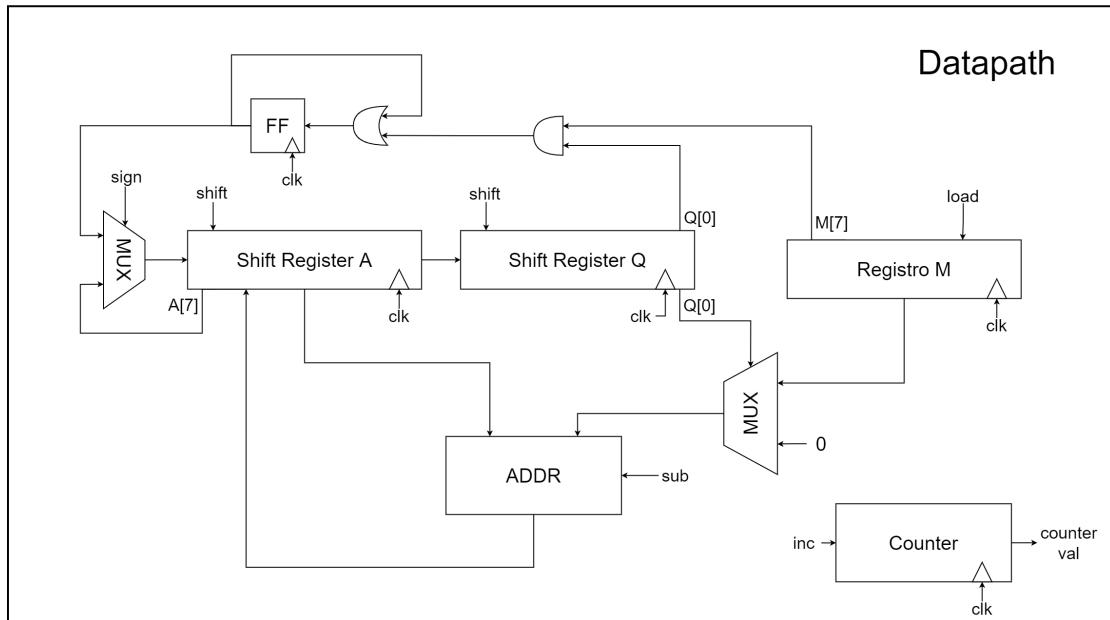


Figura 11.1: Parte operativa del moltiplicatore di Robertson

Il datapath di questo componente è quindi realizzato per la gran parte da componenti semplici e già utilizzati nei progetti precedenti. E' stato quindi possibile riusare, in fase di implementazione, i dispositivi già progettati e descritti in precedenza.

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

L'addizionatore è stato realizzato seguendo il modello del ripple carry adder, realizzato strutturalmente a partire da 8 full adder. Per supportare l'operazione di sottrazione è stato inoltre previsto una logica addizionale, che, in corrispondenza di un segnale di controllo "sub", nega il secondo operando e imposta a 1 il carry in ingresso.

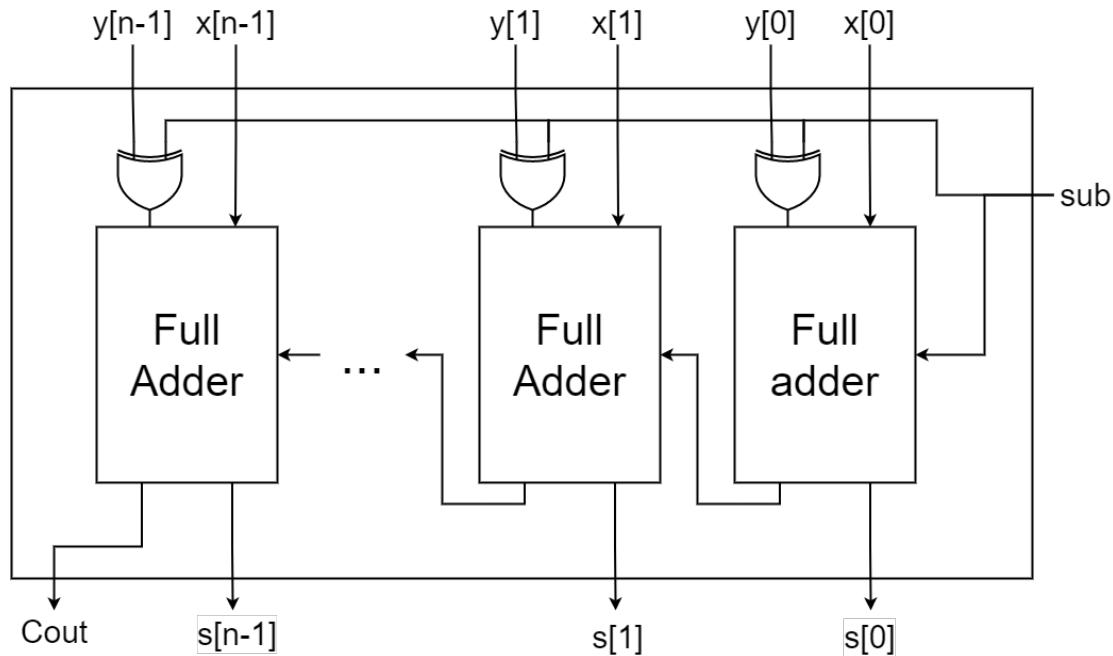


Figura 11.2: Struttura del ripple carry adder.

I segnali di controllo sono quindi i segnali di load e reset dei tre registri, il segnale di shift dei due shift register, il segnale di sub per l'addizionatore, i segnali di selezione per i due multiplexer e quello di incremento del contatore.

La control unit deve pilotare i segnali di controllo del datapath allo scopo di implementare il seguente algoritmo di somma:

BEGIN :

```
A:=0, COUNT:=0, F:=0,
```

INPUT :

```
M:=INBUS; Q:=INBUS;
```

ADD :

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

```
A[7:0]:= A[7:0] + M [7:0] x Q[0],  
F:= (M[7] and Q[0]) or F;
```

RSHIFT:

```
A[7]:= F,      A[6:0].Q:= A.Q[7:1],
```

INCREMENT:

```
COUNT:=COUNT+1
```

TEST:

```
if COUNT<7 then go to ADD;
```

SUBTRACT:

```
A[7:0]:=A[7:0]-M[7:0]xQ[0];
```

RSHIFT:

```
A[7]:= A[7],      A[6:0].Q:= A.Q[7:1];
```

OUTPUT:

```
OUTBUS:=Q; OUTBUS:=A;
```

Essendo un comportamento nel complesso semplice e ripetitivo, abbiamo optato, anche in questo caso, per l'approccio cablato, descrivendo la control unit con un automa a stati finiti ed implementandola tramite una rete combinatoria e degli elementi di memorizzazione per lo stato.

L'automa che descrive il comportamento desiderato è mostrato in figura 11.3.

Lo stato init è quello iniziale in cui la macchina si trova a riposo; a seguito della pressione del tasto di reset la macchina torna in questo stato. In corrispondenza della pressione del tasto di start, invece, si passa allo stato init, dove il contenuto dei tre registri e del flip-flop è resettato. Successivamente, in fase di op_ass sono caricati nei registri M e Q il moltiplicando ed il moltiplicatore, forniti in ingresso alla parte operativa dell'utente. A questo punto per sette volte si alternano gli stati add, dove x_iY è addizionato al contenuto del registro A, shift, in cui viene effettuato lo shift a destra dei due registri A e Q, ed inc in cui si effettua l'incremento

del contatore di stato. Quando il contatore arriva al valore 6 viene effettuato il passo di correzione in cui la control unit alza i segnali di sub, nello stato subtract, e di selezione del multiplexer in modo da preservare il segno del risultato, nello stato end_shift. Terminata la fase di correzione, la macchina torna nello stato idle.

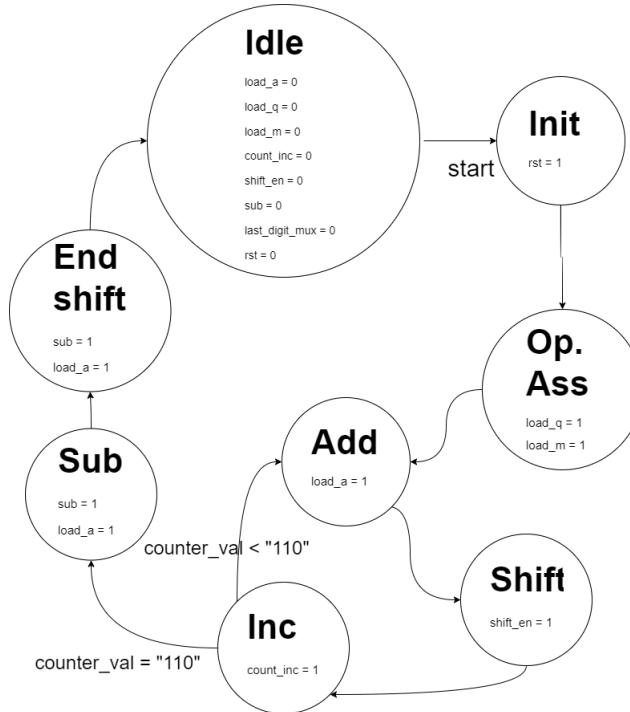


Figura 11.3: Automa della control unit

11.4 Implementazione VHDL

11.4.1 Il ripple carry adder

Il primo componente che abbiamo realizzato è l'addizionatore, iniziando dal full adder.

Questo semplice dispositivo ha tre ingressi, i due operandi ed il resto in ingresso, e due uscite, il risultato della somma, calcolata come xor degli ingressi, ed il resto

in uscita, calcolato come $(a \cdot b) + (c_{in} \cdot (a + b))$ L'implementazione dataflow di questo componente è quindi la seguente:

```

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

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

```

A partire dal componente full adder, abbiamo poi realizzato, mediante una descrizione strutturale il ripple carry adder, la cui struttura è riportata in figura 11.2.

L'implementazione strutturale in VHDL di questo componente è quindi la seguente:

```

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

        X : in std_logic_vector(7 downto 0);
        Y : in std_logic_vector(7 downto 0);

        load_a : in std_logic;

```

```

load_q : in std_logic;
load_m : in std_logic;
count_inc : in std_logic;
shift_enable : in std_logic;
sub : in std_logic;
last_digit_mux : in std_logic;

counter_val : out std_logic_vector(2 downto 0);
resoult : out std_logic_vector(15 downto 0)

);

end datapath;

architecture Structural of datapath is

signal ff_out : std_logic;
signal adder_out : std_logic_vector(7 downto 0);
signal a_parallel_out : std_logic_vector(7 downto 0);
signal a_serial_out : std_logic;
signal q_parallel_out : std_logic_vector(7 downto 0);
signal q_serial_out : std_logic;
signal m_parallel_out : std_logic_vector(7 downto 0);
signal mux_out : std_logic_vector(7 downto 0);
signal a_in : std_logic;

begin

resoult <= a_parallel_out & q_parallel_out;
a_in <= ff_out when last_digit_mux = '0' else
    a_parallel_out(7) when last_digit_mux = '1';

A : entity work.shift_register port map(
    clk=>clk, shift => shift_enable, reset => rst,
    load => load_a, serial_in=>a_in,
    parallel_in => adder_out, p
    arallel_out => a_parallel_out,

```

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

```
    serial_out => a_serial_out
  );
Q : entity work.shift_register port map(
  clk=>clk, shift => shift_enable, reset => rst,
  load => load_q, serial_in=>a_serial_out,
  parallel_in => X, parallel_out => q_parallel_out,
  serial_out => q_serial_out
);
M : entity work.registro port map(
  clk => clk, reset => rst, load => load_m,
  parallel_in => Y, parallel_out => m_parallel_out
);
addr : entity work.ripple_carry_adder port map(
  a => a_parallel_out, b => mux_out,
  sub => sub , res => adder_out
);
cnt : entity work.counter generic map(
  M => 8
) port map(
  clk => clk, rst => rst,
  count => count_inc, load => '0',
  y => counter_val,
  parallel_input => (others => '0')
);
mu : entity work.mux port map(
  b => m_parallel_out, a => (others=>'0'),
  s=> q_serial_out, out_d => mux_out
);
ff : entity work.ff_sign port map(
  q0 => q_serial_out, m7 => m_parallel_out(7),
  clk => clk, rst => rst, data => ff_out
);
end Structural;
```

11.4.2 Unità di controllo

A questo punto ci siamo occupati di implementare un dispositivo che realizza-
se l'automa descritto in figura 11.3. Abbiamo quindi realizzato una descrizione
comportamentale della control unit suddivisa in due process, uno per il calcolo e
l'update dello stato, attivo sul fronte di salita del clock, ed uno, puramente combi-
natorio, che in base allo stato computa il valore dei segnali di controllo desiderati.

La descrizione VHDL del componente è la seguente:

```

entity ControlUnit is

    port (
        clk : in std_logic;
        counter_val : in std_logic_vector(2 downto 0);
        start : in std_logic;
        reset : in std_logic;

        load_a : out std_logic;
        load_q : out std_logic;
        load_m : out std_logic;
        count_inc : out std_logic;
        shift_enable : out std_logic;
        sub : out std_logic;
        last_digit_mux : out std_logic;
        rst : out std_logic
    );
end ControlUnit;

architecture Behavioral of ControlUnit is

    type stato is (idle, init, op_ass, add, shift, inc, subtract, end_shift);
    signal state : stato := idle;

begin
    state_prc : process(clk) begin

```

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

```
if(rising_edge(clk)) then
    if(reset = '1') then
        state <= idle;
    elsif(state = idle and start = '1') then
        state <= init;
    elsif(state = init) then
        state <= op_ass;
    elsif(state = op_ass) then
        state <= add;
    elsif(state = add) then
        state <= shift;
    elsif(state = shift) then
        state <= inc;
    elsif(state = inc and counter_val = "110") then
        state <= subtract;
    elsif(state = inc) then
        state <= add;
    elsif(state = subtract) then
        state <= end_shift;
    elsif(state = end_shift) then
        state <= idle;
    end if;
end if;
end process;
out_prc : process(state) begin
--default values
load_a <= '0';
load_q <= '0';
load_m <= '0';
count_inc <= '0';
shift_enable <= '0';
sub <= '0';
rst <= '0';
```

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

```
last_digit_mux <= '0';

if(state = init) then
    rst <= '1';
elsif(state = op_ass) then
    load_q <= '1';
    load_m <= '1';
elsif(state = add) then
    load_a <= '1';
elsif(state = shift) then
    shift_enable <= '1';
elsif(state = inc) then
    count_inc <= '1';
elsif(state = subtract) then
    sub <= '1';
    load_a <= '1';
elsif(state = end_shift) then
    shift_enable <= '1';
    last_digit_mux <= '1';
end if;
end process;
end Behavioral;
```

Abbiamo quindi connesso parte operativa e parte di controllo ultimando la fase di implementazione:

```
entity moltiplicatore_robertson is
    port(
        X : in std_logic_vector(7 downto 0);
        Y : in std_logic_vector(7 downto 0);
        rst : in std_logic;
        start : in std_logic;
        clk : in std_logic;
        RES : out std_logic_vector(15 downto 0)
```

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

```
) ;

end moltiplicatore_robertson;

architecture structural of moltiplicatore_robertson is

    signal counter_val : std_logic_vector(2 downto 0);
    signal load_a : std_logic;
    signal load_q : std_logic;
    signal load_m : std_logic;
    signal count_inc : std_logic;
    signal shift_enable : std_logic;
    signal sub : std_logic;
    signal reset_datapath : std_logic;
    signal last_digit_mux : std_logic;

begin

    cu : entity work.ControlUnit port map(
        clk => clk, counter_val => counter_val,
        start=>start, reset => rst,
        sub=>sub, load_a => load_a,
        load_q => load_q, load_m => load_m,
        count_inc => count_inc, shift_enable => shift_enable,
        rst => reset_datapath,
        last_digit_mux => last_digit_mux
    );

    dp : entity work.datapath port map(
        clk => clk, X => X, Y => Y,
        load_a => load_a, load_q => load_q,
        load_m => load_m, count_inc => count_inc,
        shift_enable => shift_enable,
    );

```

```

        sub => sub, rst => reset_datapath,
        counter_val => counter_val, result => res,
        last_digit_mux => last_digit_mux
    );
}

end structural;

```

11.5 Testing simulato

Durante l'implementazione del moltiplicatore abbiamo effettuato alcuni test nell'ambiente simulato allo scopo di garantirne la corretta descrizione.

Il primo componente che abbiamo testato è l'adder, per il quale abbiamo verificato la correttezza delle somme effettuate sia tra numeri positivi che negativi. Abbiamo, inoltre, testato il corretto funzionamento della modalità di sottrazione e del segnale di sub.

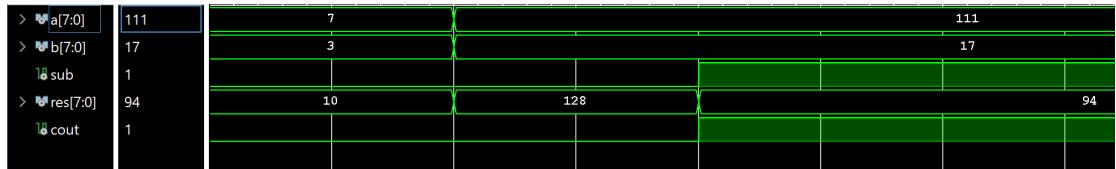


Figura 11.4: Testing simulato dell'ripple carry adder.

Una volta implementato il componente connettendo parte operativa ed unità di controllo, abbiamo testato l'effettiva correttezza dei calcoli da questo realizzati.

Abbiamo quindi verificato la corretta esecuzione dell'algoritmo di calcolo evidenziando, all'interno del testbench, lo stato dell'unità di controllo.

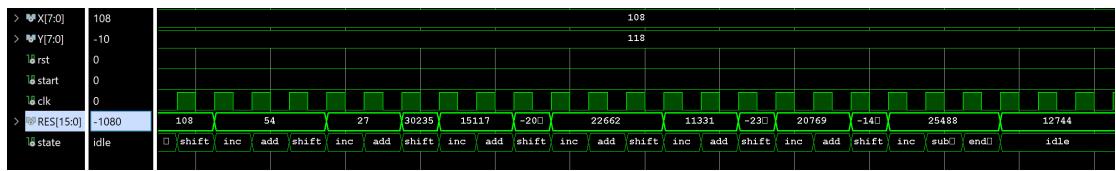


Figura 11.5: Moltiplicazione tra due numeri positivi.

PROGETTO 11. IL MOLTIPLICATORE DI ROBERTSON

Abbiamo inoltre verificato la corretta esecuzione del passo di correzione nel caso di moltiplicazione tra numeri negativi.



Figura 11.6: Moltiplicazione tra negativi.

I risultati sono, per tutti i componenti testati, soddisfacenti; siamo quindi passati alla fase di implementazione su board.

11.6 Caricamento sulla scheda

Per concludere il progetto abbiamo caricato il moltiplicatore di Robertson su scheda. Dovendo ricevere in ingresso due operandi da otto bit e dovendo mostrare, in uscita, il risultato su sedici bit abbiamo deciso di mappare i due ingressi a due gruppi di otto switch e l'uscita ai sedici led presenti sulla board. Abbiamo inoltre mappato i tasti di start e reset a due bottoni della scheda. In figura 12.8 è riportato il mapping effettuato.



Figura 11.7: Moltiplicazione tra negativi.

Progetto 12

Addizionatore in virgola mobile

Le macchine aritmetiche trattate fino a questo punto sono in grado di operare esclusivamente su numeri interi. Per molte applicazioni di simulazione e di calcolo è però necessario estendere l'insieme dei numeri su cui è possibile operare, fornendo la possibilità di effettuare calcoli su numeri decimali. I calcolatori elettronici rappresentano in forma approssimata i numeri reali tramite la rappresentazione in virgola mobile che consente di riprodurre una vasta gamma di numeri decimali.

Un numero in virgola mobile è composto da due parti fondamentali:

- un campo mantissa, M
- un campo esponente, e .

Un generico numero reale, fissata una specifica base b , può essere quindi espresso, in termini di mantissa ed esponente come $x = M \times b^e$. Per garantire l'univocità della rappresentazione si impone una condizione di normalizzazione.

Lo standard di rappresentazione che abbiamo adottato per la rappresentazione dei numeri in virgola mobile è quello indicato dallo standard IEEE754, che prevede, per i numeri in virgola mobile in singola precisione, l'utilizzo di 32 bit. La mantissa è rappresentata in modulo, su 23 bit, e segno, rappresentato sul bit iniziale. L'esponente è invece rappresentato su 8 bit ed è espresso in eccesso 127.

Il valore numerico rappresentato da un numero in virgola mobile è pari a:

$$(-1)^s \times 2^{(e-127)} \times (1.M)$$

12.1 Descrizione della soluzione

L'operazione di somma per numeri rappresentati in virgola mobile, a differenza di quelle di prodotto e rapporto, risulta piuttosto complessa. Prima di poter effettuare la somma algebrica tra due numeri espressi in termini di mantissa ed esponente, bisogna, anzitutto, effettuare un passo di pre-normalizzazione, necessario per allineare le mantisse nel caso di esponenti differenti. L'approccio tipicamente utilizzato allo scopo di minimizzare l'errore di approssimazione è quello di allineare la mantissa dell'operando minore a quello maggiore. Si effettua quindi il calcolo della differenza tra gli esponenti: in base al segno di questa, si determina l'addendo minore e, selezionata la sua mantissa, si effettuano shift a destra pari alla differenza tra gli esponenti precedentemente calcolata.

Effettuato questo step preliminare, è poi possibile sommare le due mantisse, utilizzando ad esempio un ripple carry adder. Prima di poter effettuare la somma tra i due numeri è però necessario complementare gli operandi negativi, il cui bit di segno è alto.

Una volta effettuata la somma è, successivamente, necessario rinormalizzare il risultato in modo da garantire l'aderenza allo standard del risultato. Bisogna, quindi, effettuare un certo numero di shift a destra o a sinistra e, allo stesso tempo, correggere il valore dell'esponente, in modo da portare la mantissa nella forma $1.M$.

Analizzando l'algoritmo di somma, abbiamo deciso di optare per l'utilizzo di

alcuni componenti notevoli, tra cui shift register per le operazioni di normalizzazione, addizionatori e sottrattori e un dispositivo combinatorio utilizzato per la complementazione, che andranno a comporre l'unità operativa. L'unità di controllo invece dovrà pilotare i segnali di controllo di questa architettura allo scopo di eseguire correttamente la somma.

12.2 Schematici, componenti

12.2.1 Unità operativa

Lo schema complessivo dell'architettura progettata è il seguente:

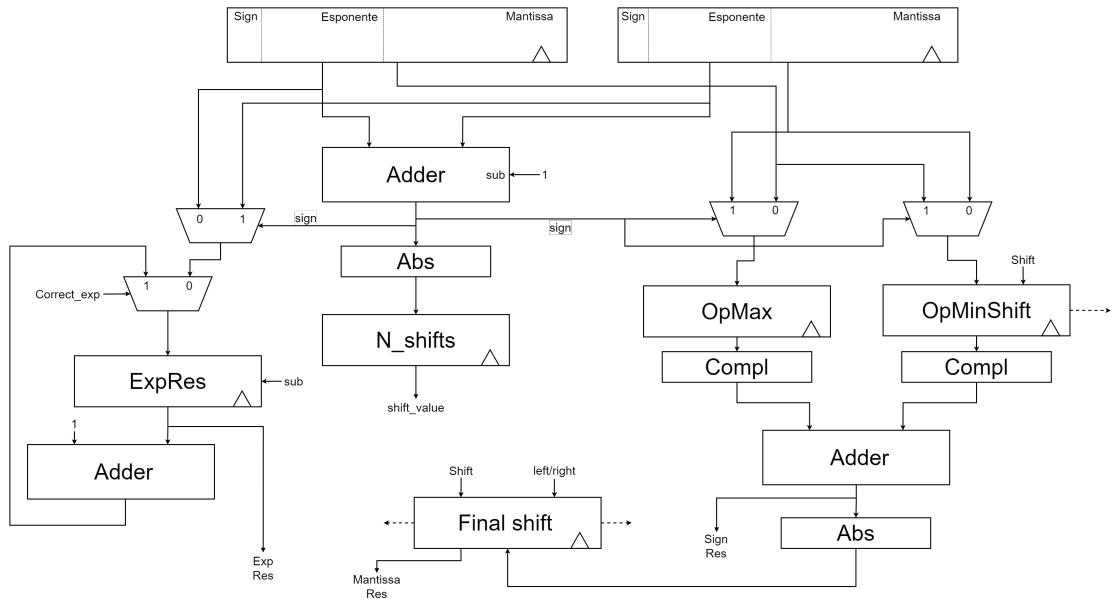


Figura 12.1: Schema complessivo dell'unità operativa.

Gran parte dei componenti utilizzati nel datapath sono componenti notevoli già trattati e descritti nei progetti precedenti. Troviamo un primo addizionatore utilizzato per il calcolo della differenza ed il confronto tra i due esponenti. Il risultato di questa somma sarà utilizzato dalla parte di controllo allo scopo di determinare il numero di shift necessari per allineare le due mantisse. Sono poi presenti alcuni multiplexer il cui segnale di selezione corrisponde al segno della

differenza tra i due esponenti. Questi consentono di memorizzare all'interno del registro ExpRes il valore dell'esponente maggiore e di portare all'interno dello shift register OpMinShift la mantissa dell'addendo con l'esponente minore. All'interno di questo shift register saranno effettuati un numero di shift a destra pari alla differenza dei due esponenti, valore posto all'interno del registro N_shifts. In modo da garantire che il numero di shift sia positivo (nel caso in cui l'esponente del secondo addendo è maggiore del primo l'adder restituisce un risultato negativo), il segnale è precedentemente elaborato da un semplice blocco combinatorio che complementa il risultato ed aggiunge a questo un 1, nel caso in cui questo risulti negativo, allo scopo di calcolarne il valore assoluto.

Le due mantisse, una volta allineate, prima di poter essere addizionate, sono ulteriormente elaborate da una apposita rete combinatoria indicata nello schema come Compl. Questa, la cui descrizione dataflow è riportata nel paragrafo successivo, complementa l'operando in ingresso se il bit di segno è alto e lo lascia inalterato in caso contrario. Questo blocco è necessario per poter effettuare la somma tramite un addizionatore ripple carry nonostante le due mantisse siano rappresentate in modulo e segno. L'ultimo blocco abs, precedentemente descritto, consente di portare il risultato dell'addizione in rappresentazione modulo e segno.

L'ultima fase dell'algoritmo di somma prevede di effettuare shift a destra o sinistra sulla mantissa calcolata allo scopo di rinormalizzare il risultato e, in corrispondenza di questi, correggere il valore dell'esponente memorizzato nel registro ExpRes. E' quindi necessario uno shift register che consente di shiftare la mantissa a destra o a sinistra, nello schema indicato come Final_shift, ma anche di un addizionatore utilizzato allo scopo di aggiungere uno, in corrispondenza di un right shift, o di sottrarre uno, in corrispondenza di ogni left shift, al valore memorizzato nel registro ExpRes.

12.2.2 Unità di controllo

L'automa che descrive il funzionamento dell'unità di controllo è il seguente:

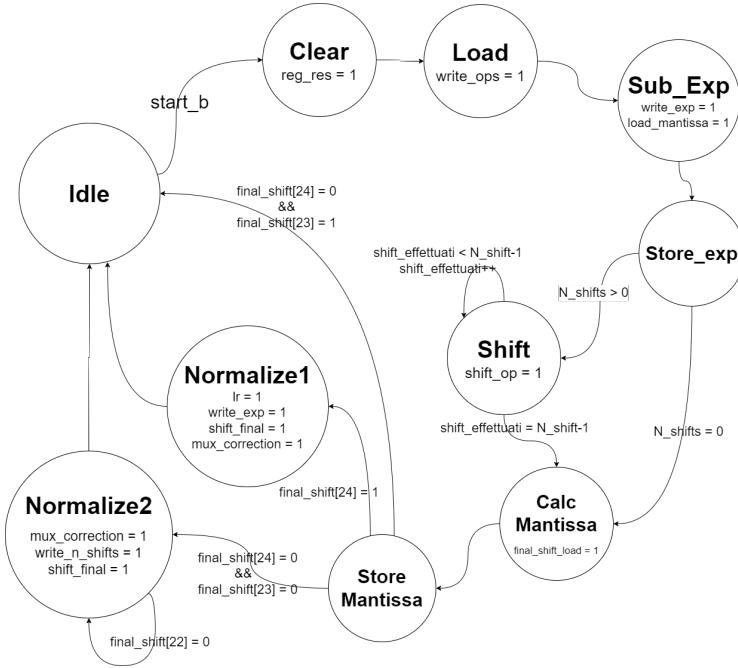


Figura 12.2: Unità di controllo.

Lo stato iniziale di riposo è quello di **idle**. Alla pressione del tasto **start_b** la macchina si porta prima allo stato di **clear**, in cui sono inizializzati tutti i registri interni, poi allo stato di **load**, in cui i due operandi, forniti in ingresso alla macchina, sono memorizzati all'interno dei due appositi registri. Nello stato successivo, **sub_exp**, la differenza tra i due operandi è caricata all'interno del registro **n_shifts**, sono inoltre caricati i registri **exp_val** con il risultato massimo tra i due operandi e le due mantisse nei registri **OpMax** e **OpMinShift**, in base al valore del segno della differenza. Nello stato di **store_exp** viene poi valutato il contenuto del registro **n_shifts**. Se pari a zero non sono necessarie operazioni di shift per l'allineamento delle mantisse e si passa allo stato successivo. In caso contrario, si passa allo stato **shift**, dove si effettuano, un numero di shift a sx sull'operando minore pari al numero caricato nell'apposito registro. Nello stato **calc_mantissa** è caricato il risultato della differenza tra le mantisse all'interno del

registro final_shift. Nel passo successivo si valuta il risultato ottenuto: se questo è nella forma $01.M$ risulta essere normalizzato e si torna quindi nello stato iniziale. Se si trova nella forma $1 - .M$ è necessaria una operazione di rinormalizzazione shiftando il risultato a destra ed incrementando il valore di exp_val. Se la mantissa è invece nella forma $00.M$, è necessario effettuare un numero di shift variabile fino a portare la mantissa nella forma desiderata, procedendo a decrementare il valore dell'esponente.

Visto la relativa semplicità dell'unità di controllo e considerando la ripetitività delle sequenze di controllo, abbiamo, anche in questo caso, optato per la realizzazione del componente in logica cablata.

12.3 Implementazione in VHDL

12.3.1 Blocchi di conversione

Il primo blocco di conversione utilizzato calcola il valore assoluto dell'operando in ingresso. L'implementazione è semplice: utilizzando le proprietà della rappresentazione in complementi a due, se il bit più significativo dell'operando è alto questo sarà negativo ed occorre complementarlo ed aggiungere uno. Questa operazione può essere facilmente realizzata facendo ricorso ad un ripple carry adder ed una porta not. In caso contrario, invece, l'operando è positivo e va restituito inalterato in uscita.

```
entity abs_value is generic(
    N : integer := 8
);
port (
    X : in std_logic_vector(N-1 downto 0);
    Y : out std_logic_vector(N-1 downto 0)
);

```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
end abs_value;

architecture Structural of abs_value is

signal not_X : std_logic_vector(N-1 downto 0);
signal zeros : std_logic_vector(N-2 downto 0);
signal b_port : std_logic_vector(N-1 downto 0);
signal neg_plus_one : std_logic_vector(N-1 downto 0);

begin

zeros <= (others => '0');
b_port <= zeros & '1';
not_X <= not X;

addr : entity work.ripple_carry_adder generic map(
N=>N
) port map (
a => not_X, b => b_port, sub => '0', res => neg_plus_one
);

with X(N-1) select
Y <= X when '0',
neg_plus_one when '1',
X when others;
end Structural;
```

Il secondo blocco di conversione è invece utilizzato allo scopo di convertire le mantisse, rappresentate inizialmente in modulo e segno, ottenendo una rappresentazione in complementi a due più adeguato per eseguire la somma.

Se il bit di segno, posto in ingresso alla macchina oltre al valore del modulo, è alto, l'uscita risulta inalterata anche in questo caso. Viene infatti esclusivamente aggiunto un bit 0 come MSB.

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

Se il bit di segno è invece alto, la macchina complementa il modulo ed aggiunge uno. Il MSB dell'uscita è, in questo caso, impostato a 1.

```
entity converter is generic(
    N : integer := 8
);
port (
    X : in std_logic_vector(N-1 downto 0);
    sign : in std_logic;
    Y : out std_logic_vector(N downto 0)
);
end converter;

architecture Structural of converter is

signal not_X : std_logic_vector(N-1 downto 0);
signal zeros : std_logic_vector(N-2 downto 0);
signal b_port : std_logic_vector(N-1 downto 0);
signal neg_plus_one : std_logic_vector(N-1 downto 0);

begin
    zeros <= (others => '0');
    b_port <= zeros & '1';
    not_X <= not X;

    addr : entity work.ripple_carry_adder generic map(
        N=>N
    ) port map (
        a => not_X, b => b_port, sub => '0', res => neg_plus_one
    );
    with sign select
        Y <= '0' & X when '0',
        '1' & neg_plus_one when '1',

```

```

'0' & X when others;
end Structural;

```

12.3.2 Unità operativa

Per l'implementazione del dispositivo in VHDL abbiamo utilizzato un approccio strutturale, seguendo lo schema precedentemente commentato e riportato in figura 12.1.

Per la realizzazione, oltre alla progettazione dei blocchi di conversione, abbiamo riutilizzato molti componenti progettati in precedenza: gli addizionatori utilizzati sono derivati dal ripple carry adder utilizzato nel progetto undici, i registri a scorrimento ricavati dall'esercizio 4 ed i multiplexer dal'esercizio 1.

La descrizione in VHDL è quindi la seguente:

```

entity UnitaOperativa is
  port(
    clk : in std_logic;
    op1, op2 : in std_logic_vector(31 downto 0);
    reg_reset : in std_logic;
    write_ops, write_n_shifts : in std_logic;
    load_mantissa, shift_op : in std_logic;
    shift_final, lr_final, final_shift_load, mux_correction: in std_logic;
    res : out std_logic_vector(31 downto 0)
  );
end UnitaOperativa;

architecture Structural of UnitaOperativa is

  signal regA, regB : std_logic_vector(31 downto 0);
  signal expA, expB, diff_exp, abs_diff_exp : std_logic_vector(8 downto 0);
  --8 bit + segno
  signal mantissaA, mantissaB : std_logic_vector(22 downto 0);
  signal signA, signB : std_logic;

```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
signal sign_mantissaA, sign_mantissaB : std_logic_vector(23 downto 0);
signal M : std_logic;
signal minMantissa, maxMantissa : std_logic_vector(23 downto 0);
signal exp_max_mux, exp_max : std_logic_vector(7 downto 0);
signal muxMintoreg, muxMaxtoreg : std_logic_vector(24 downto 0);
signal shift_reg_min_out, reg_max_out : std_logic_vector(24 downto 0);
signal add1, add2, res_addizione,
mantissa_da_normalizzare : std_logic_vector(25 downto 0);
signal mantissa_finale : std_logic_vector(24 downto 0);
signal mux_correct_out, adder_exp_out : std_logic_vector(7 downto 0);
signal not_lr : std_logic;

begin
    expA <= '0' & regA(30 downto 23);
    expB <= '0' & regB(30 downto 23);
    signA <= regA(31);
    signB <= regB(31);
    mantissaA <= regA(22 downto 0);
    mantissaB <= regB(22 downto 0);
    sign_mantissaA <= signA & mantissaA;
    sign_mantissaB <= signB & mantissaB;

    A : entity work.registro generic map(
        N => 32
    ) port map (
        clk=>clk, rst => reg_reset, write => write_ops,
        data_in => op1, data_out => regA
    );
    B : entity work.registro generic map(
        N => 32
    ) port map (
        clk=>clk, rst => reg_reset, write => write_ops,
        data_in => op2, data_out => regB
    );
end;
```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
) ;

sub_exp : entity work.ripple_carry_adder generic map(
    N=>9
) port map (
    A=>expA, B=>expB, res=>diff_exp, sub=>'1'
);

abs_sub_exp : entity work.abs_value generic map(
    N=>9
) port map (
    X=>diff_exp, Y => abs_diff_exp
);

M <= diff_exp(8);

n_shifts : entity work.registro generic map(
    N=>8
) port map (
    clk=>clk, rst=> reg_reset,
    data_in=>abs_diff_exp(7 downto 0), write=>write_n_shifts
);

mux_exp_max : entity work.mux generic map(
    N=>8
) port map(
    a=>expA(7 downto 0), b=>expB(7 downto 0), s=>M, out_d=>exp_max_mux
);

mux_correct_exp : entity work.mux generic map(
    N=>8
) port map (
    a=>exp_max_mux, b=>adder_exp_out,
    s=>mux_correction, out_d=> mux_correct_out
);

not_lr <= not lr_final;

adder_correct_exp : entity work.ripple_carry_adder generic map(
```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
N=>8

) port map (
    a=>exp_max, b=>"00000001", sub => not_lr, res => adder_exp_out
);

exp_res : entity work.registro generic map(
    N=>8

) port map (
    clk=>clk, rst=>reg_reset, write=>write_n_shifts,
    data_in=>mux_correct_out, data_out=>exp_max
);

mux_mantissa_min : entity work.mux generic map(
    N=>24 --dim mantissa + bit segno

) port map (
    a=>sign_mantissaB, b=>sign_mantissaA, s=>M, out_d =>minMantissa
);

mux_mantissa_max : entity work.mux generic map(
    N=>24 --dim mantissa + bit segno

) port map (
    a=>sign_mantissaA, b=>sign_mantissaB, s=>M, out_d =>maxMantissa
);

muxmaxtoreg <= "01" & maxMantissa(22 downto 0);
muxmintoreg <= "01" & minMantissa(22 downto 0);

shift_register_min : entity work.shift_register generic map(
    N => 25

) port map (
    clk=>clk, shift=>shift_op, lr=>'1',
    reset=>reg_reset, load=>load_mantissa,
    serial_in => '0', parallel_in => muxmintoreg,
    parallel_out => shift_reg_min_out
);

reg_max : entity work.registro generic map(
    N =>25
```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
) port map (
    clk=>clk, rst=>reg_reset,
    write=>load_mantissa, data_in => muxmaxtoreg,
    data_out=>reg_max_out
);
convMin : entity work.converter generic map(
    N => 25
) port map (
    X=>shift_reg_min_out, sign => minMantissa(23), Y=>add1
);
convMax : entity work.converter generic map(
    N => 25
) port map (
    X=>reg_max_out, sign => maxMantissa(23), Y=>add2
);
add_final : entity work.ripple_carry_adder generic map(
    N => 26
) port map(
    A => add1, B => add2, sub => '0', res=>res_addizione
);
abs_res : entity work.abs_value generic map(
    N=>26
) port map (
    X=>res_addizione, Y=>mantissa_da_normalizzare
);
final_shift : entity work.shift_register generic map(
    N=>25
) port map (
    clk=>clk, shift=>shift_final, lr=>lr_final,
    reset=>reg_reset, load=>final_shift_load,
    parallel_in => mantissa_da_normalizzare(24 downto 0),
    serial_in => '0', parallel_out => mantissa_finale
);
```

```

res <= res_addizione(25) & exp_max & mantissa_finale(22 downto 0);

end Structural;

```

12.3.3 Unità di controllo

L’unità di controllo è stata implementata seguendo la logica cablata, realizzando un dispositivo che implementa l’automa descritto in precedenza come combinazione di una rete combinatoria che calcola stato prossimo ed uscita e di un elemento di memorizzazione utilizzato per lo stato corrente.

In questo caso, per la realizzazione dell’automa in forma di Moore, abbiamo optato per l’utilizzo di due processi, uno sensibile al fronte del clock, che aggiorna lo stato, ed il secondo, puramente combinatorio e sensibile allo stato, che calcola l’uscita.

Le uscite di controllo fornite dalla UC sono:

- write_ops e write_n_shifts, utilizzati per la scrittura nei registri operando, nel registro n_shifts ed ExpRes.
- load_mantissa, utilizzato per il caricamento delle mantisse nei registri dove saranno allineate ed addizionate.
- shift_op, segnale di shift a destra per l’operazione di allineamento della mantissa corrispondente all’operando con esponente minore.
- final_shift_load, shift_final e lr_final, utilizzati per il controllo dello shift register utilizzato per la rinormalizzazione della mantissa dopo l’addizione
- mux_correction, per l’abilitazione del mux utilizzato per la correzione del valore ExpRes.

```

entity ControlUnit is port(
    start_b : in std_logic;

```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
clk : in std_logic;
N_shifts : in std_logic_vector(7 downto 0);
final_shift : in std_logic_vector(24 downto 0);
reg_reset : out std_logic;
write_ops, write_n_shifts : out std_logic;
load_mantissa, shift_op : out std_logic;
shift_final, lr_final, final_shift_load,
mux_correction: out std_logic);

end ControlUnit;

architecture Behavioral of ControlUnit is

type stato is (IDLE, START, SUB_EXP, STORE_EXP,
SHIFT, CALC_MANTISSA, STORE_MANTISSA,
NORMALIZE_1, NORMALIZE_2, CLEAR);

signal stato_corrente : stato := IDLE;
signal shift_effettuati : std_logic_vector(7 downto 0) := "00000000";
begin

proc_state : process(clk) begin
    if(rising_edge(clk)) then
        if(stato_corrente=IDLE and start_b='1') then stato_corrente <= CLEAR;
        elsif(stato_corrente=CLEAR) then stato_corrente <= START;
        elsif(stato_corrente=START) then stato_corrente <= SUB_EXP;
        elsif(stato_corrente=SUB_EXP) then stato_corrente <= STORE_EXP;
        elsif(stato_corrente=STORE_EXP) then
            if(N_shifts=x"00") then stato_corrente <= CALC_MANTISSA;
            else stato_corrente <= SHIFT;
            end if;
        elsif(stato_corrente=SHIFT and shift_effettuati=N_shifts-1) then
            stato_corrente <= CALC_MANTISSA;
            shift_effettuati <= (others=>'0');
        elsif(stato_corrente=SHIFT) then
            stato_corrente <= SHIFT;
```

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

```
    shift_effettuati<=shift_effettuati+1;

    elsif(statorettore=CALC_MANTISSA) then
        statorettore<=STORE_MANTISSA;

    elsif(statorettore=STORE_MANTISSA) then
        if(final_shift=x"000000"&"0")
            then statorettore <= IDLE;
        elsif(final_shift(24)='1')
            then statorettore <= NORMALIZE_1;
        elsif(final_shift(24)='0' and final_shift(23)='1')
            then statorettore <= IDLE;
        elsif(final_shift(24)='0' and final_shift(23)='0')
            then statorettore <= NORMALIZE_2;
        end if;

    elsif(statorettore = NORMALIZE_1) then statorettore <= IDLE;
    elsif(statorettore = NORMALIZE_2) then
        if(final_shift(22)='0') then statorettore <= NORMALIZE_2;
        else statorettore <= IDLE;
        end if;
    end if;

    end if;
end process;

out_proc : process(statorettore) begin
    reg_reset <= '0';
    write_ops <= '0';
    write_n_shifts <= '0';
    load_mantissa <= '0';
    shift_op <= '0';
    shift_final <= '0';
    lr_final <= '0';
    final_shift_load <= '0';
    mux_correction <= '0';
```

```

if(stato_corrente=START) then
    write_ops <= '1';
elsif(stato_corrente=CLEAR) then
    reg_reset <= '1';
elsif(stato_corrente=SUB_EXP) then
    write_n_shifts <= '1';
    load_mantissa <= '1';
elsif(stato_corrente=SHIFT) then
    shift_op <= '1';
elsif(stato_corrente=CALC_MANTISSA) then
    final_shift_load <= '1';
elsif(stato_corrente=NORMALIZE_1) then
    lr_final <= '1';
    write_n_shifts<='1';
    shift_final <= '1';
    mux_correction <= '1';
elsif(stato_corrente=NORMALIZE_2) then
    mux_correction <= '1';
    write_n_shifts <= '1';
    shift_final <= '1';
end if;
end process;
end Behavioral;

```

Dall'implementazione si può notare come, allo scopo di garantire il corretto numero di shift in fase di pre-normalizzazione, il componente adopera un contatore, descritto in maniera puramente comportamentale all'interno dell'unità di controllo.

12.4 Testing simulato

La fase successiva è quella di testing. Abbiamo effettuato per questo dispositivo il testing sia dei singoli componenti introdotti ad hoc, sia dell'unità di operativa

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

nel complesso per verificare la corretta composizione di questa. Abbiamo poi effettuato i test di integrazione connettendo l'unità operativa e quella di controllo ed effettuando la somma tra alcuni operandi.

Abbiamo testato inizialmente i blocchi di conversione. Data la similitudine delle due architetture, si riporta, per brevità, esclusivamente il risultato del testing relativo al componente che calcola il valore assoluto.



Figura 12.3: Testing del componente che calcola il valore assoluto.

Il risultato ottenuto è quello desiderato: i numeri positivi sono restituiti in uscita invariati mentre a quelli negativi è associato correttamente il modulo.

Allo scopo di testare l'unità operativa nella sua interezza, abbiamo realizzato tramite un testbench tutti le operazioni necessarie all'esecuzione dell'algoritmo di calcolo. Tutte le funzionalità fondamentali tra cui la selezione dell'esponente maggiore, il calcolo del numero di shift, lo shift della mantissa con esponente minore, il calcolo della somma e la rinormalizzazione sono eseguite correttamente.



Figura 12.4: Testing delle funzionalità dell'unità operativa.

Nell'ultima fase abbiamo testato l'unità operativa ed il sistema complessivo, verificando la correttezza dei risultati prodotti a partire da vari operandi.

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

La prima operazione che abbiamo effettuato è la seguente:

$$0\ 10000000\ 00000010100011110101110 = 2.02 +$$

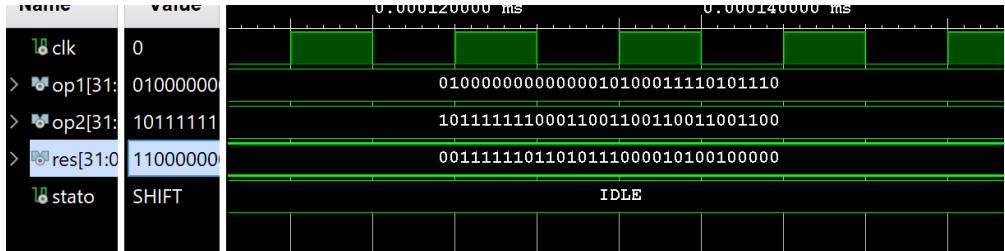
$$1\ 01111111\ 00011001100110011001100 = -1.1$$

Stiamo, quindi, effettuando la somma algebrica tra due numeri relativi. Il ciclo di esecuzione dell'operazione è il seguente:



Figura 12.5: Stati di esecuzione dell'operazione $2.02 + (-1.1)$.

Il calcolo porta, quindi, al seguente risultato:



Si ottiene di conseguenza che il risultato dell'operazione è:

$$0\ 01111110\ 11010111000010100100000 = 0.920$$

Il risultato di questo primo test è corretto: otteniamo, infatti, il numero previsto.

Il test successivo che abbiamo effettuato ha lo scopo di verificare la corretta esecuzione della fase di normalizzazione eseguita a seguito della somma tra le mantisse precedentemente allineate.

I due operandi sono in questo caso:

$$0\ 10000101\ 10011001010111000010100 = 102.34 +$$

PROGETTO 12. ADDIZIONATORE IN VIRGOLA MOBILE

$$1\ 10000101\ 10010001010111000010100 = -100.34$$

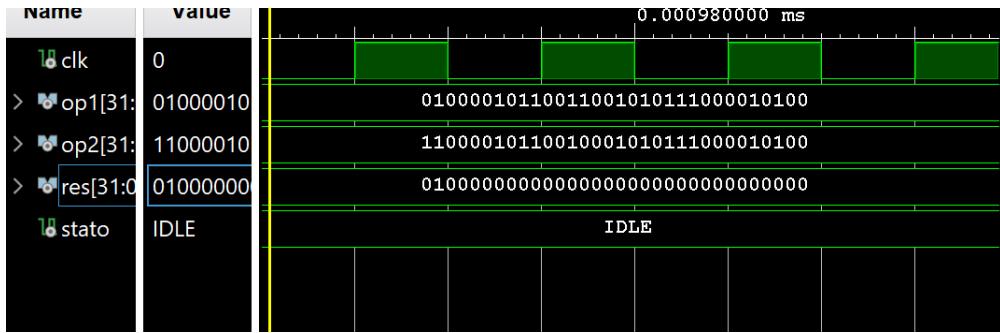


Figura 12.6: Secondo calcolo effettuato.

il risultato ottenuto è come previsto pari a:

$$0\ 10000000\ 00000000000000000000000000000000 = 2$$

Il terzo ed ultimo scenario che riportiamo richiede sia una operazione di allineamento iniziale che di normalizzazione finale. Gli addendi in questo caso sono:

$$0\ 10000100\ 01010011010111000010100 = 42.42 +$$

$$0\ 10000011\ 01110011110101110000101 = 23.24$$



Figura 12.7: Terzo caso di test analizzato

Il risultato ottenuto è quindi pari a:

$$0\ 10000101\ 00000110101000111101011 = 65.6599$$

Il risultato è quindi corretto, nonostante la limitata precisione della rappresentazione in virgola mobile che causa un piccolo errore di approssimazione sulla mantissa.

12.5 Caricamento sulla scheda

Abbiamo poi sintetizzato su FPGA l'addizionatore. Poiché gli operandi previsti sono a 32 bit, ci è stato impossibile caricarli in maniera immediata tramite le porte fornite sulla scheda. Abbiamo quindi optato per il pre-caricamento di due operandi in fase di implementazione, fornendo in ingresso al dispositivo i due addendi visti all'esempio precedente. L'uscita dell'addizionatore, anch'essa rappresentata su 32 bit, è stata mostrata utilizzando le otto cifre esadecimali del visore. Abbiamo, inoltre, mappato il segnale di start su uno dei bottoni della scheda.

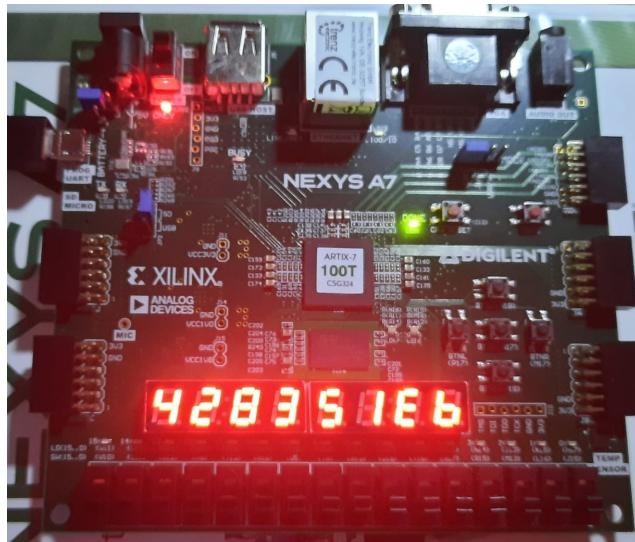


Figura 12.8: Fotografia del risultato ottenuto

Il dispositivo, implementato su FPGA, ha dimostrato di funzionare correttamente. Il risultato mostrato in esadecimale sul visore corrisponde, infatti, al risultato previsto, 0 10000101 00000110101000111101011, precedentemente discusso.