

**UNIVERSITÀ DEGLI STUDI DI NAPOLI
FEDERICO II**



Corso di Laurea Magistrale in Ingegneria Informatica

DIPARTIMENTO DI INGEGNERIA ELETTRICA E DELLE TECNOLOGIE
DELL'INFORMAZIONE

TESINA DEL CORSO DI

ARCHITETTURA DEI SISTEMI DIGITALI

Professori:

Nicola Mazzocca
Alessandra De Benedictis

GRUPPO 12 - Candidati:

Benfenati Domenico
Fusco Camilla
Banchini Federico
Iannuzzi Gianluca

ANNO ACCADEMICO 2021/2022

Primo Semestre

Indice

| | |
|---|-----------|
| 1 Esercizio 1 | 11 |
| 1.1 Traccia | 11 |
| 1.2 Soluzioni | 11 |
| 1.3 Schematici | 11 |
| 1.4 Codice | 14 |
| 1.5 Simulazione | 18 |
| 2 Esercizio 2.1 | 21 |
| 2.1 Traccia | 21 |
| 2.2 Soluzioni | 21 |
| 2.3 Schematici | 22 |
| 2.4 Codice | 25 |
| 2.4.1 Soluzione a : descrizione comportamentale | 25 |
| 2.4.2 Soluzione b : descrizione strutturale | 27 |
| 2.5 Simulazione | 33 |
| 3 Esercizio 2.2 | 36 |
| 3.1 Traccia | 36 |
| 3.2 Soluzioni | 36 |
| 3.3 Schematici | 37 |
| 3.4 Codice | 39 |
| 3.4.1 Soluzione a : descrizione comportamentale | 39 |
| 3.4.2 Soluzione b : descrizione ibrida | 41 |
| 3.5 Simulazione | 44 |
| 4 Esercizio 3 | 45 |
| 4.1 Traccia | 45 |
| 4.2 Soluzione | 45 |
| 4.3 Schematici | 46 |
| 4.4 Codice | 47 |

| | | |
|----------|---|------------|
| 4.5 | Simulazione | 51 |
| 4.6 | Sintesi su FPGA | 53 |
| 5 | Esercizio 4 | 59 |
| 5.1 | Traccia | 59 |
| 5.2 | Soluzioni | 59 |
| 5.3 | Schematici | 60 |
| 5.4 | Codice | 63 |
| 5.4.1 | Soluzione comportamentale | 63 |
| 5.4.2 | Soluzione strutturale | 64 |
| 5.5 | Simulazione | 68 |
| 6 | Esercizio 5 | 72 |
| 6.1 | Traccia | 72 |
| 6.2 | Soluzioni | 72 |
| 6.3 | Schematici | 73 |
| 6.3.1 | Unità operativa | 74 |
| 6.3.2 | Sistema Esterno | 75 |
| 6.3.3 | Unità di controllo | 75 |
| 6.4 | Codice | 77 |
| 6.4.1 | Unità di controllo in Logica cablata | 86 |
| 6.4.2 | Unità di controllo in Logica microprogrammata | 88 |
| 6.5 | Simulazione | 92 |
| 7 | Esercizio 6 | 95 |
| 7.1 | Traccia | 95 |
| 7.2 | Soluzione | 95 |
| 7.3 | Schematici | 95 |
| 7.3.1 | Automi | 97 |
| 7.4 | Codice | 99 |
| 7.5 | Simulazione | 110 |
| 8 | Esercizio 7 | 113 |
| 8.1 | Traccia | 113 |
| 8.2 | Soluzione | 113 |
| 8.2.1 | Unità di controllo | 114 |
| 8.2.2 | Unità operativa | 114 |
| 8.3 | Schematici | 114 |
| 8.3.1 | Unità operativa | 115 |

| | | |
|-----------|---|------------|
| 8.3.2 | Unità di controllo | 118 |
| 8.4 | Codice | 119 |
| 8.4.1 | Unità Operativa | 119 |
| 8.4.2 | Unità di Controllo | 130 |
| 8.4.3 | Top Module | 132 |
| 8.5 | Simulazione | 134 |
| 9 | Esercizio 8 | 142 |
| 9.1 | Traccia | 142 |
| 9.2 | Introduzione | 142 |
| 9.2.1 | Architettura a stack | 142 |
| 9.2.2 | Datapath | 143 |
| 9.2.3 | Accesso alla memoria | 144 |
| 9.2.4 | Comunicazione tra registri | 146 |
| 9.2.5 | Unità ALU | 148 |
| 9.2.6 | Unità di controllo | 149 |
| 9.2.7 | Architettura completa | 154 |
| 9.2.8 | IJVM | 155 |
| 9.3 | Analisi del Codice | 158 |
| 9.3.1 | datapath.vhd | 158 |
| 9.3.2 | alu.vhd | 162 |
| 9.3.3 | control_store.vhd | 163 |
| 9.3.4 | control_unit.vhd | 164 |
| 9.4 | Studio di due istruzioni a scelta | 165 |
| 9.4.1 | Istruzione 1 : IADD | 165 |
| 9.4.2 | Istruzione 2 : BIPUSH | 166 |
| 9.5 | Simulazione | 168 |
| 9.5.1 | IADD | 168 |
| 9.5.2 | BIPUSH | 169 |
| 9.6 | Modifica di un istruzione | 170 |
| 10 | Esercizio 9 | 171 |
| 10.1 | Traccia | 171 |
| 10.2 | Cenni teorici | 172 |
| 10.2.1 | Ricezione | 175 |
| 10.2.2 | Trasmissione | 177 |
| 10.3 | Soluzione | 179 |
| 10.4 | Schematici | 180 |

| | |
|--|------------|
| 10.5 Codice | 181 |
| 10.5.1 UART Tappo | 181 |
| 10.5.2 2 UART | 182 |
| 10.6 Simulazione | 183 |
| 10.7 Sintesi su FPGA | 184 |
| 11 Esercizio 10 | 186 |
| 11.1 Traccia | 186 |
| 11.2 Soluzione 1 | 186 |
| 11.3 Schematici 1 | 188 |
| 11.3.1 Gestione della priorità | 189 |
| 11.4 Codice 1 | 189 |
| 11.4.1 Unità operativa | 189 |
| 11.4.2 Unità di controllo | 192 |
| 11.5 Simulazione 1 | 193 |
| 11.6 Soluzione 2 | 196 |
| 11.7 Schematici 2 | 196 |
| 11.8 Codice 2 | 198 |
| 11.8.1 Switch 2:2 | 198 |
| 11.8.2 Switch 8:8 | 200 |
| 11.9 Simulazione 2 | 204 |
| 11.10 Soluzione 3 | 206 |
| 11.11 Schematici 3 | 208 |
| 11.12 Codice 3 | 209 |
| 11.12.1 Gestore Handshake | 209 |
| 11.12.2 Switch Omega Handshaking | 211 |
| 11.13 Simulazione 3 | 215 |
| 12 Esercizio 11 | 218 |
| 12.1 Traccia | 218 |
| 12.2 Soluzione | 218 |
| 12.2.1 Cenni Teorici | 218 |
| 12.3 Schematici | 221 |
| 12.4 Codice | 222 |
| 12.4.1 Unità operativa | 222 |
| 12.4.2 Unità di controllo | 226 |
| 12.4.3 Contatore | 228 |
| 12.4.4 Gestore Button | 228 |

| | |
|--------------------------------------|-----|
| 12.4.5 Componente Generale | 230 |
| 12.5 Simulazione | 233 |
| 12.6 Sintesi su FPGA | 236 |

Elenco delle figure

| | | |
|-----|--|----|
| 1.1 | Soluzione ad albero | 12 |
| 1.2 | Soluzione a semiselezione | 13 |
| 1.3 | Simulazione Decoder 4:16 | 20 |
| 2.1 | Modello di Huffman di una macchina sequenziale | 22 |
| 2.2 | Diagramma degli stati | 23 |
| 2.3 | Architettura di Huffman del riconoscitore | 23 |
| 2.4 | Codifica degli stati e tabella di transizione di stato | 24 |
| 2.5 | Tabella di verità delle uscite | 24 |
| 2.6 | Mappe di Karnaugh delle uscite dei flip-flop e della macchina combinatoria | 25 |
| 2.7 | Simulazione del riconoscitore 1-1 | 35 |
| 3.1 | Diagramma degli stati | 37 |
| 3.2 | Tabella di transizione e codifica degli stati | 37 |
| 3.3 | Tabella di verità della memoria di stato e dell'uscita | 38 |
| 3.4 | Mappe di Karnaugh della macchina combinatoria | 38 |
| 3.5 | Simulazione del riconoscitore 1-10 | 44 |
| 4.1 | Schema strutturale dell'orologio | 46 |
| 4.2 | Simulazione dell'orologio | 53 |
| 4.3 | Reset fisico dell'orologio su scheda | 55 |
| 4.4 | Set delle ore su scheda | 55 |
| 4.5 | Set dei minuti su scheda | 56 |
| 4.6 | Set dei secondi su scheda | 56 |
| 4.7 | Avanzamento del tempo su scheda | 57 |
| 4.8 | Settaggio delle ore 23:59:59 su scheda | 57 |
| 4.9 | Reset automatico dell'orologio su scheda | 58 |
| 5.1 | Struttura generale del registro a scorrimento | 61 |
| 5.2 | Shift a sinistra di un bit pari a 0 | 61 |

| | | |
|------|---|-----|
| 5.3 | Shift a destra di un bit pari a 0 | 62 |
| 5.4 | Shift circolare verso sinistra | 62 |
| 5.5 | Shift a sinistra di un bit pari a X | 62 |
| 5.6 | Simulazione del registro a scorrimento (1) | 70 |
| 5.7 | Simulazione del registro a scorrimento (2) | 70 |
| 5.8 | Simulazione del registro a scorrimento con uscita seriale | 71 |
| 6.1 | Funzionamento generale del protocollo di handshake | 73 |
| 6.2 | Architettura generale del sistema | 74 |
| 6.3 | Architettura dell'unità operativa | 74 |
| 6.4 | Architettura del sistema esterno | 75 |
| 6.5 | Automa di funzionamento del sistema esterno | 75 |
| 6.6 | Automa dell'unità di controllo | 76 |
| 6.7 | Architettura dell'unità di controllo in logica cablata | 76 |
| 6.8 | Architettura dell'unità di controllo in logica microprogrammata | 77 |
| 6.9 | Simulazione della macchina AmodB in logica cablata | 94 |
| 6.10 | Simulazione della macchina AmodB in logica microprogrammata | 94 |
| 7.1 | Architettura generica del sistema di comunicazione | 96 |
| 7.2 | Architettura di dettaglio del sistema di comunicazione | 96 |
| 7.3 | Diagramma degli stati del sistema A | 97 |
| 7.4 | Diagramma degli stati del sistema B | 97 |
| 7.5 | Diagramma degli stati dell'unità FSM-A | 98 |
| 7.6 | Diagramma degli stati dell'unità FSM-B | 99 |
| 7.7 | Simulazione della comunicazione tramite buffer | 112 |
| 8.1 | Architettura generica della macchina Prodotto Scalare | 115 |
| 8.2 | Architettura generica dell'unità operativa | 115 |
| 8.3 | Architettura del componente Vector | 116 |
| 8.4 | Algoritmo di Booth per la moltiplicazione con segno | 117 |
| 8.5 | Architettura interna dell'unità aritmetico-logica | 118 |
| 8.6 | Diagramma degli stati dell'unità di controllo | 118 |
| 8.7 | Simulazione del componente Vector | 136 |
| 8.8 | Simulazione del moltiplicatore di Booth | 138 |
| 8.9 | Simulazione dell'unità Aritmetica | 139 |
| 8.10 | Simulazione della macchina Prodotto Scalare | 141 |
| 9.1 | Datapath del processore MIC-1 | 143 |
| 9.2 | Indirizzamento a parole del registro MAR | 145 |

| | |
|---|-----|
| 9.3 Ciclo di clock del MIC-1 | 146 |
| 9.4 Disponibilità dei dati nei registri di interfaccia con la memoria | 147 |
| 9.5 Architettura interna dell'ALU | 148 |
| 9.6 Unità di controllo nel processore MIC-1 | 150 |
| 9.7 Struttura di una Control Word della micro-ROM nel MIC-1 | 151 |
| 9.8 Struttura di codifica dei bit di attivazione del bus B | 151 |
| 9.9 Circuito e mappa di Karnaugh per la definizione di High-Bit | 153 |
| 9.10 Circuito di funzionamento di JMPC | 153 |
| 9.11 Architettura generale del processore MIC-1 | 154 |
| 9.12 Esempio di gestione dello stack nel MIC-1 | 155 |
| 9.13 Esempio di utilizzo dello stack con l'operazione di somma | 157 |
| 9.14 Simulazione dell'istruzione IADD tramite GTK-WAVE | 168 |
| 9.15 Simulazione dell'istruzione BIPUSH tramite GTK-WAVE | 169 |
| 9.16 Simulazione dell'istruzione IADD modificata tramite GTK-WAVE | 170 |
| 10.1 Frame dati trasmessi tramite un UART | 172 |
| 10.2 Schema di una comunicazione | 174 |
| 10.3 Segnali di ingresso e uscita dell'UART | 174 |
| 10.4 Schema di parte di trasmissione e ricezione di un UART | 175 |
| 10.5 Schema di funzionamento della ricezione sull'UART | 176 |
| 10.6 Diagramma a stati in ricezione dell'UART | 176 |
| 10.7 Schema di funzionamento della trasmissione sull'UART | 178 |
| 10.8 Diagramma a stati in trasmissione dell'UART | 178 |
| 10.9 Configurazione a tappo | 180 |
| 10.10Configurazione con due UART | 180 |
| 10.11Simulazione UART a tappo e con due UART | 184 |
| 10.12Sintesi su scheda dell'UART | 185 |
| 11.1 Tecnica del perfect faro shuffle | 187 |
| 11.2 Architettura interna dello switch 2:2 | 188 |
| 11.3 Architettura delle reti Omega | 188 |
| 11.4 Architettura del sistema con priorità | 189 |
| 11.5 Simulazione Switch multistadio versione 1 | 195 |
| 11.6 Schema di bit di un ingresso dei componenti | 196 |
| 11.7 Architettura interna dello switch 2:2 - soluzione 2 | 197 |
| 11.8 Architettura dello switch 8:8 | 197 |
| 11.9 Simulazione Switch multistadio versione 2 | 206 |
| 11.10Diagramma a stati finiti del Gestore Handshake | 207 |

| | | |
|-------|---|-----|
| 11.11 | Architettura del top level Switch Handshaking | 208 |
| 11.12 | Architettura interna del sistema Switch Omega con Handshake | 209 |
| 11.13 | Simulazione Switch multistadio versione 3 | 217 |
| 12.1 | Architettura interna del moltiplicatore di Booth | 221 |
| 12.2 | Automa dell'algoritmo di Booth | 222 |
| 12.3 | Simulazione del moltiplicatore di Booth - in0 positivo e in1 negativo | 234 |
| 12.4 | Simulazione del moltiplicatore di Booth - in0 negativo e in1 negativo | 235 |
| 12.5 | Simulazione del moltiplicatore di Booth - in0 positivo e in1 negativo | 235 |
| 12.6 | Sintesi su scheda del moltiplicatore di Booth - positivo per negativo | 237 |
| 12.7 | Sintesi su scheda del moltiplicatore di Booth - negativo per negativo | 238 |
| 12.8 | Sintesi su scheda del moltiplicatore di Booth - positivo per positivo | 238 |

Elenco delle tavelle

| | | |
|------|---|-----|
| 6.1 | Combinazione di bit di una control word | 77 |
| 6.2 | Tabella di verità di un Full-Adder | 79 |
| 7.1 | Tabella di trasmissione dei bit | 112 |
| 9.1 | Tabella di codifica delle operazioni dell'ALU | 149 |
| 9.2 | Tabella di codifica dei segnali di abilitazione del bus B | 151 |
| 9.3 | Codici operativi e istruzioni in IJVM | 156 |
| 11.1 | Tabella di codifica dei bit di priorità | 187 |
| 12.1 | Esempio di codifica di Booth di un numero | 219 |
| 12.2 | Tabella di codifica di Booth-2 | 220 |

Capitolo 1

Esercizio 1

1.1 Traccia

Si progetti un decoder 4:16 utilizzando componenti decoder 2:4 opportunamente interconnessi mediante le strutture:

- ad albero;
- a semiselezione.

1.2 Soluzioni

L'approccio utilizzato per la risoluzione di tale esercizio è di tipo modulare, in cui la macchina da implementare viene scomposta in componenti più piccoli.

In particolare, la struttura ad albero è composta da cinque decoder separati in due livelli, il primo del quale è composto da un solo decodificatore, le cui uscite forniscono il segnale di abilitazione al livello successivo, che contiene i decodificatori rimanenti, le cui uscite rappresentano l'uscita dell'intero sistema. La struttura a semiselezione è composta da due decoder, le cui uscite vengono opportunamente poste in AND a due a due per formare l'uscita dell'intero sistema. I due decodificatori vengono abilitati dallo stesso segnale in ingresso all'intero sistema complessivo.

1.3 Schematici

Il Decoder 2:4 è una macchina combinatoria notevole, che riceve in ingresso una parola codice su n bit e presenta in uscita la sua rappresentazione decodificata su $m = 2^n$ bit. Opzionalmente, può ricevere anche un ingresso di abilitazione e , tale che quando $e = 0$ tutte le uscite del decoder sono nulle.

Il Decoder 4:16, analogamente, è una macchina combinatoria che fa corrispondere ad un codice di

4 bit in ingresso un'uscita decodificata di 16 bit, in cui al più uno dei bit è alto, se l'abilitazione è alta e se è stato inserito un codice valido in ingresso.

Sia per la soluzione ad albero che per la soluzione a semiselezione, la progettazione è stata fatta tramite la pratica della **composizione di macchine**, dividendo il codice in ingresso su n bit in sottocodici formati da un numero di bit inferiore. In particolare, si è scelto di esprimere i 4 bit in ingresso al decoder 4:16 come combinazione di due gruppi di 2 bit, i quali rappresentano gli ingressi dei decoder 2:4 usati per la composizione.

Per quel che riguarda la struttura ad albero, l'ingresso **x3x2x1x0** del decoder 4:16 è stato suddiviso nei due codici **x3x3** e **x1x0**, che rispettivamente piloteranno il primo e il secondo livello dell'architettura risultante, rappresentati da un decoder al primo livello le cui uscite vanno ad abilitare uno dei quattro decoder presenti al secondo livello dell'architettura. Il primo livello dell'architettura riceve in ingresso i due bit più significativi, mentre gli altri due bit sono dati in ingresso ai decoder del secondo livello.

In Figura 1.1 è rappresentata la struttura proposta.

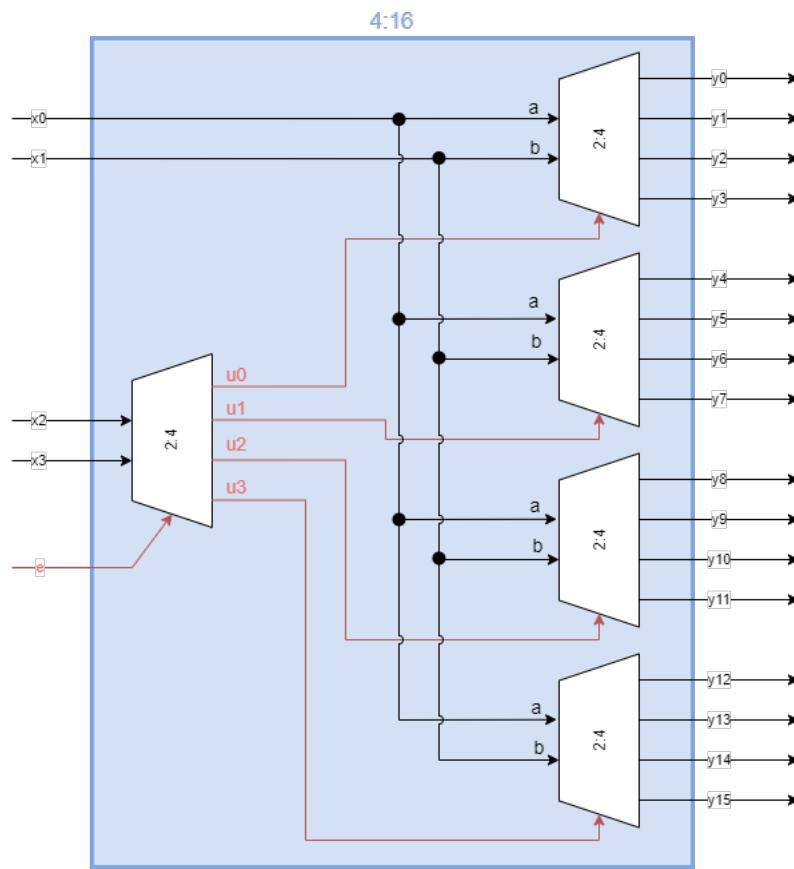


Figura 1.1: Soluzione ad albero

Per quel che riguarda invece la struttura a semiselezione, si è scelto di utilizzare due decoder, abilitati da uno stesso ingresso di abilitazione e . Le uscite di tali decoder vanno ad essere combinate seguendo uno schema matriciale, come si nota in Figura 1.2, e poste opportunamente in ingresso

ad n porte logiche AND, dove n è il numero di uscite.

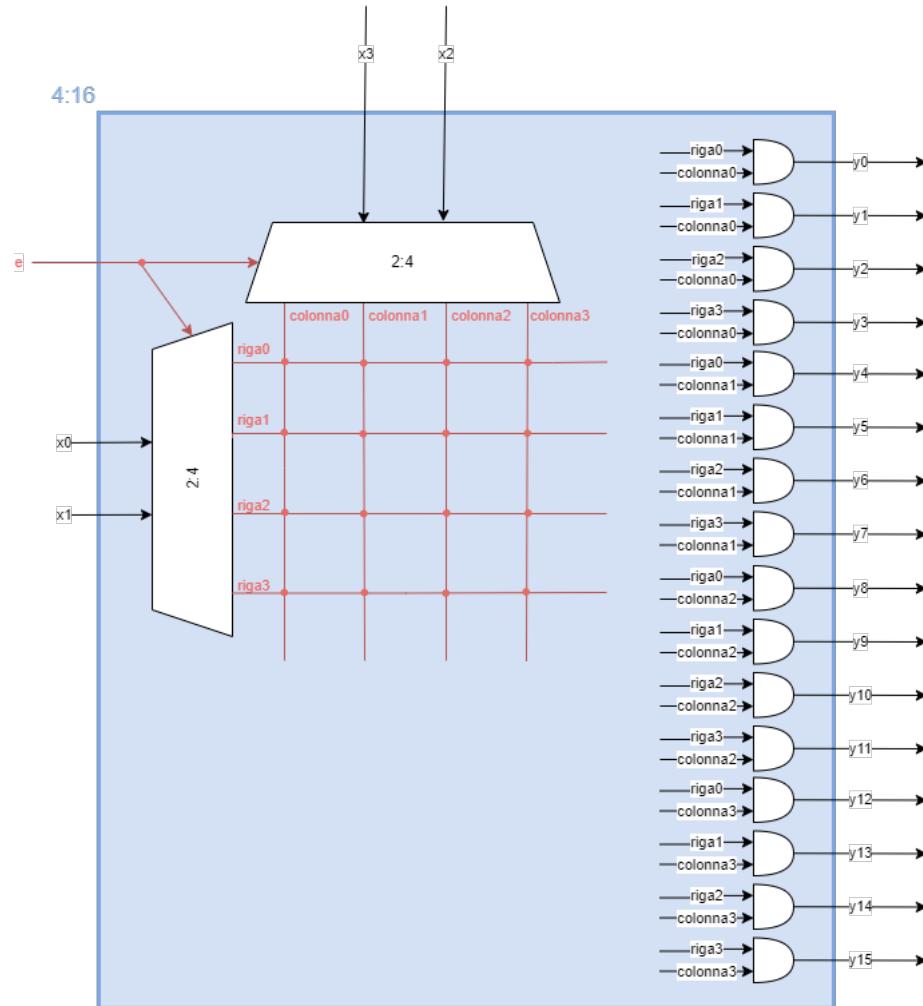


Figura 1.2: Soluzione a semiselezione

1.4 Codice

Si è scelto di partire dall'implementazione del decoder 2:4: come prima cosa si è definito l'entity `decoder_2_4`, che rappresenta l'interfaccia del componente. Al suo interno sono stati dichiarati i 3 ingressi (`a`,`b`,`e`) e le 4 uscite (`o`). L'entità è stata implementata attraverso un process, con una descrizione **comportamentale**.

Come si può vedere, se il segnale di abilitazione è alto, in base alla combinazione dei due bit in ingresso, in uscita verrà alzato solo uno dei 4 bit, quello corrispondente, in maniera posizionale partendo da destra verso sinistra, alla decodifica del valore in ingresso.

```
entity decoder_2_4 is
    Port ( a : in STD_LOGIC;
            b : in STD_LOGIC;
            e : in STD_LOGIC;
            o : out STD_LOGIC_VECTOR (3 downto 0));
end decoder_2_4;

architecture Behavioral of decoder_2_4 is

begin
    process(a,b,e)
    begin
        if (e='1') then
            if (a='0' AND b='0') then
                o<="0001";
            elsif (a='1' AND b='0') then
                o<="0010";
            elsif (a='0' AND b='1') then
                o<="0100";
            elsif (a='1' AND b='1') then
                o<="1000";
            end if;
        else
            o<="0000";
        end if;
    end process;
end Behavioral;
```

Per l'implementazione del decoder 4:16 sia per quanto riguarda la decomposizione ad albero, sia per quella a semiselezione, si è utilizzato la stessa entity, per la quale sono state poi definite due architetture differenti tramite il costrutto `architecture`. Inoltre, in entrambi i casi, per descrivere il sistema come composizione di macchine, ci siamo posti a livello di astrazione strutturale, facendo uso del costrutto `component`, che consente di dichiarare i componenti utilizzati da un design,

mentre con il port map è possibile istanziare il componente nella parte di descrizione funzionale di un'architettura. Di seguito l'implementazione della soluzione ad albero.

```

entity decoder_4_16 is
    Port ( x : in std_logic_vector (3 downto 0);
           e : in std_logic;
           y : out std_logic_vector (15 downto 0)
    );
end decoder_4_16;

architecture Structural of decoder_4_16 is
    signal enable0 : STD_LOGIC := '0';
    signal enable1 : STD_LOGIC := '0';
    signal enable2 : STD_LOGIC := '0';
    signal enable3 : STD_LOGIC := '0';

    component decoder_2_4 is
        Port ( a : in STD_LOGIC;
               b : in STD_LOGIC;
               e : in STD_LOGIC;
               o : out STD_LOGIC_VECTOR (3 downto 0));
    end component;

begin
    decoder0: decoder_2_4
    port map(
        e => e,
        a => x(3),
        b => x(2),
        o(0) => enable0,
        o(1) => enable1,
        o(2) => enable2,
        o(3) => enable3
    );

    decoder1: decoder_2_4
    port map(
        e => enable0,
        a => x(1),
        b => x(0),
        o(0) => y(0),
        o(1) => y(1),
        o(2) => y(2),
        o(3) => y(3)
    );

    decoder2: decoder_2_4
    port map(

```

```

        e => enable0,
        a => x(1),
        b => x(0),
        o(0) => y(4),
        o(1) => y(5),
        o(2) => y(6),
        o(3) => y(7)
    );
}

decoder3: decoder_2_4
port map(
    e => enable0,
    a => x(1),
    b => x(0),
    o(0) => y(8),
    o(1) => y(9),
    o(2) => y(10),
    o(3) => y(11)
);
}

decoder4: decoder_2_4
port map(
    e => enable0,
    a => x(1),
    b => x(0),
    o(0) => y(12),
    o(1) => y(13),
    o(2) => y(14),
    o(3) => y(15)
);
}

end Structural;

```

L'implementazione della soluzione a semiselezione segue la stessa filosofia della precedente, ovviamente variandone la logica. Di seguito si riporta il codice di tale schema.

```

entity decoder_4_16 is
    Port ( x : in std_logic_vector (3 downto 0);
           e : in std_logic;
           y : out std_logic_vector (15 downto 0)
    );
end decoder_4_16;

architecture Structural of decoder_4_16 is

    signal riga : std_logic_vector(3 downto 0);
    signal colonna : std_logic_vector(3 downto 0);

```

```

component decoder_2_4
port (
    a : in std_logic;
    b : in std_logic;
    e : in std_logic;
    o : out std_logic_vector(3 downto 0)
);
end component;

component operatore_and
port (
    a : in std_logic;
    b : in std_logic;
    z : out std_logic
);
end component;

begin
decoder_riga: decoder_2_4
port map (
    a => x(2),
    b => x(3),
    e => e,
    o => riga
);
decoder_colonna: decoder_2_4
port map (
    a => x(0),
    b => x(1),
    e => e,
    o => colonna
);
and_riga : for i in 0 to 3 generate
    and_colonna : for j in 0 to 3 generate
        uscita : operatore_and
        port map (
            a=> riga(i),
            b=> colonna(j),
            z=> y(j+i*4)
        );
    end generate;
end generate;
end Structural;

```

Si noti che, per quel che riguarda la struttura a semiselezione, è stato necessario definire anche il componente `operatore_and`, che descrive il funzionamento di una porta logica AND tramite descrizione comportamentale. Di seguito si riporta tale implementazione.

```

entity operatore_and is
  Port ( a : in STD_LOGIC;
         b : in STD_LOGIC;
         z : out STD_LOGIC);
end operatore_and;

architecture dataflow of operatore_and is

begin
  z <= a AND b;

end dataflow;

```

1.5 Simulazione

Al fine di verificare il corretto funzionamento di tali strutture, si è scelto di definire un file di testbench ad-hoc per la simulazione di tali architetture. Il file è stato utilizzato univocamente per entrambe le strutture, ed è definito di seguito.

```

entity decoder_4_16_tb is
end decoder_4_16_tb;

architecture testbench of decoder_4_16_tb is

component decoder_4_16
  Port ( x : in std_logic_vector (3 downto 0);
         e : in std_logic;
         y : out std_logic_vector (15 downto 0)
        );
end component;

signal x: std_logic_vector (3 downto 0);
signal e: std_logic:= '1';
signal y: std_logic_vector (15 downto 0) ;

begin

uut: decoder_4_16 port map ( x => x,
                           e => e,
                           y => y );

stimulus: process
begin

  x <= "0000";
  wait for 10 ns;

```

```

assert y="0000000000000001";
x <= "0001";
wait for 10 ns;
assert y="0000000000000010";
x <= "0010";
wait for 10 ns;
assert y="00000000000000100";
x <= "0011";
wait for 10 ns;
assert y="000000000000001000";
x <= "0100";
wait for 10 ns;
assert y="0000000000000010000";
x <= "0101";
wait for 10 ns;
assert y="00000000000000100000";
x <= "0110";
wait for 10 ns;
assert y="000000000000001000000";
x <= "0111";
wait for 10 ns;
assert y="0000000000000010000000";
x <= "1000";
wait for 10 ns;
assert y="0000000010000000";
x <= "1001";
wait for 10 ns;
assert y="0000000100000000";
x <= "1010";
wait for 10 ns;
assert y="0000001000000000";
x <= "1011";
wait for 10 ns;
assert y="0000010000000000";
x <= "1100";
wait for 10 ns;
assert y="0001000000000000";
x <= "1101";
wait for 10 ns;
assert y="0010000000000000";
x <= "1110";
wait for 10 ns;
assert y="0100000000000000";
x <= "1111";
wait for 10 ns;
assert y="1000000000000000";
wait;

```

```
end process;
```

```
end testbench;
```

A valle della fase di simulazione, sono stati ottenuti gli schemi riportati in basso, identici per entrambe le architetture.

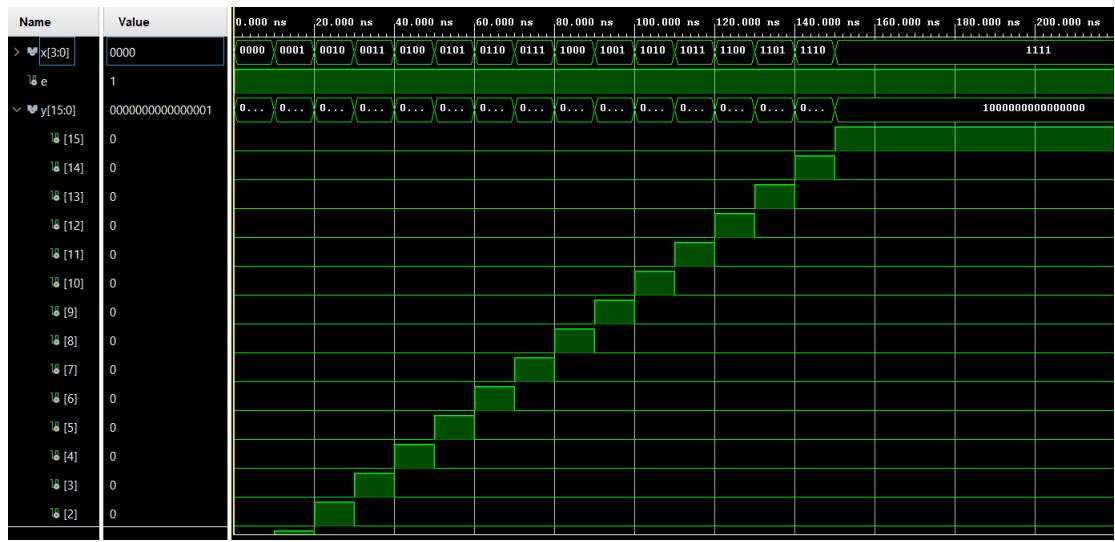


Figura 1.3: Simulazione Decoder 4:16

Capitolo 2

Esercizio 2.1

2.1 Traccia

Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna. La macchina riceve attraverso un ingresso seriale stringhe di 3 bit e, alla ricezione del terzo bit di ciascuna stringa, fornisce uscita alta se la sequenza ricevuta è **1-1**. Si disegni l'automa e si proceda alla sintesi utilizzando flip-flop D. Implementare la macchina in VHDL utilizzando:

- a) una descrizione di tipo comportamentale che faccia uso di un unico processo;
- b) una descrizione strutturale in cui vengano evidenziati tutti i componenti risultanti dalla sintesi (porte logiche e flip-flop) e le loro interconnessioni.

NOTA: per risolvere il punto b) è richiesto l'utilizzo di componenti realizzati ad-hoc che implementano le porte AND e OR. L'implementazione del flip-flop D può essere fatta utilizzando una descrizione comportamentale.

2.2 Soluzioni

Per quanto riguarda la soluzione al **punto a**, è stato necessario definire l'automa a stati finiti che caratterizza tale riconoscitore, per poter mappare all'interno del codice le operazioni con formalismi matematici, sfruttando un unico `process` come richiesto. La sintesi è stata effettuata a valle dell'operazione di codifica della macchina tramite mappe di Karnaugh. Per quel che invece riguarda il **punto b**, è stato necessario definire due componenti: la **memoria**, composta dall'adeguato numero di flip-flop di tipo D per la codifica dei relativi stati dell'automa, e la **rete combinatoria**, che comprende la logica elaborativa della macchina, espressa mediante combinazione di porte AND, OR e NOT, a seconda della sintesi effettuata sfruttando le mappe di Karnaugh.

2.3 Schematici

Le macchine sequenziali sono macchine in cui l'uscita, oltre a dipendere dall'ingresso, dipende anche dallo stato in cui si trova la macchina. Esso può essere definito come la memoria interna del sistema, in base alla quale esso reagisce con una determinata “**uscita**” ad un determinato “**ingresso**”. Una macchina sincrona è una macchina in cui lo stato corrente dipende dalla durata del segnale di ingresso e dai tempi di reazione della macchina. Il modello di macchina sincrona più semplice è quello delle **macchine a sincronizzazione esterna**, dove gli ingressi vengono campionati solo in corrispondenza di un segnale di sincronizzazione esterno impulsivo. Poiché le transizioni avvengono in generale tra stati non stabili, è necessario congelare lo stato prossimo calcolato in base al valore dello stato corrente e dell'ingresso in un dato istante (identificato dal segnale di sincronismo) all'interno di una memoria, da cui esso viene letto al successivo segnale di sincronismo per calcolare la nuova transizione di stato e la nuova uscita. Ciò è realizzato tramite il **modello di Huffman**, schematizzato in Figura 2.1. Il comportamento di una macchina può

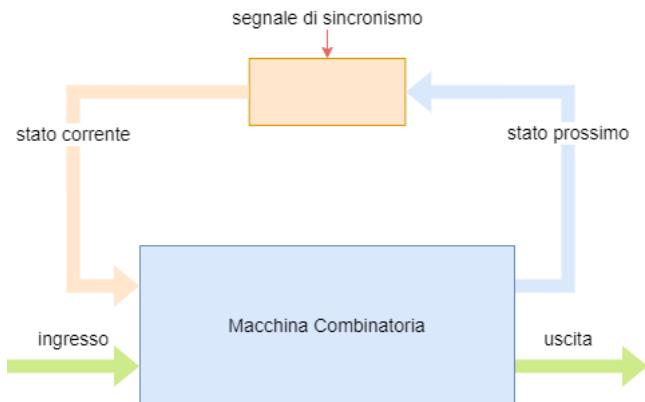


Figura 2.1: Modello di Huffman di una macchina sequenziale

essere rappresentato tramite un automa a stati finiti. Abbiamo due modelli fondamentali di ASF:

- Modello di Mealy: si rappresenta la variazione dell'uscita U in funzione di ingresso I e stato Q $w : Q \times I \rightarrow U$
- Modello di Moore: l'uscita U dipende solo dallo stato Q , il quale è modificato tramite l'ingresso $w : Q \rightarrow U$

Si è quindi proceduto a progettare la versione a grafo di tale automa, tramite il **diagramma degli stati**, che esprime gli stati come nodi e le transizioni di stato come archi del grafo, etichettati con i valori di ingresso e uscita che si ottengono a valle di tale transizione. In Figura 2.2 è espresso tale diagramma, analogo sia per la soluzione a che per la soluzione b.

Ottenuto l'automa, per riuscire a codificare quest'ultimo per il punto b della traccia, si è proceduto alla scrittura delle mappe di Karnaugh delle funzioni di uscita dei flip-flop che compongono la memoria, e dell'uscita della macchina combinatoria complessiva, tenendo a mente lo schema

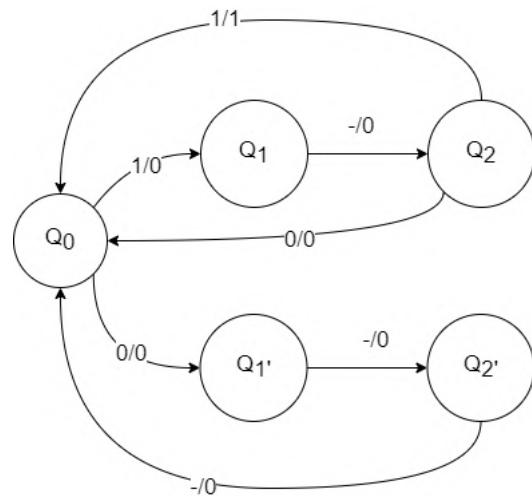


Figura 2.2: Diagramma degli stati

riportato in Figura 2.3 che adatta il modello di Huffman al caso in esame.

Per fare ciò è necessario definire la codifica degli stati dell'automa, e mappare le transizioni di stato in una tabella, riportate in Figura 2.4.

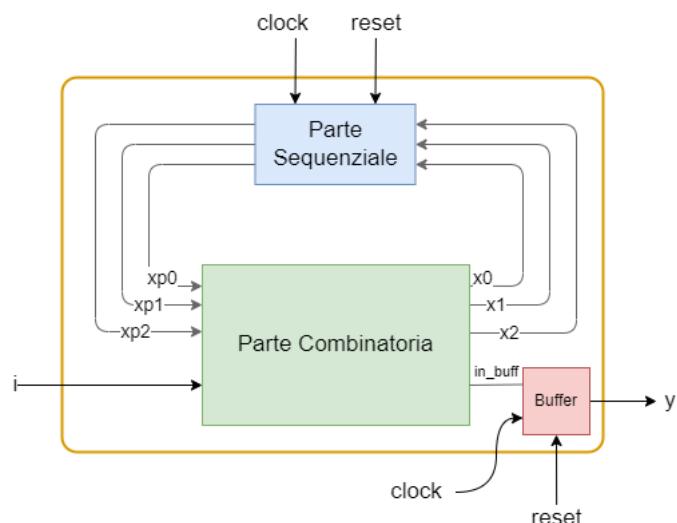


Figura 2.3: Architettura di Huffman del riconoscitore

Codificati gli stati, per rendere più agevole la scrittura delle mappe di Karnaugh, è stato ritenuto opportuno stilare la tabella di verità di ognuna delle uscite da codificare, al variare dei bit di stato e dell'ingresso esterno i . Tale tabella è riportata in Figura 2.5

| | INGRESSO | | | |
|-------|----------|------|-------|----------|
| STATO | 0 | 1 | STATO | CODIFICA |
| Q0 | Q1' | Q1 | Q0 | 000 |
| Q1 | Q2 | Q2 | Q1 | 001 |
| Q2 | Q0/0 | Q0/1 | Q2 | 010 |
| Q1' | Q2' | Q2' | Q1' | 011 |
| Q2' | Q0/0 | Q0/0 | Q2' | 100 |

Figura 2.4: Codifica degli stati e tabella di transizione di stato

| STATO | BIT DI STATO | | | IN | USCITE FLIP-FLOP | | | OUT |
|-------|--------------|----|----|----|------------------|-----|-----|-----|
| | x2 | x1 | x0 | | i | FF2 | FF1 | |
| Q0 | 0 | 0 | 0 | 0 | 0 | 1 | 1 | 0 |
| Q0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Q1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Q1 | 0 | 0 | 1 | 1 | 0 | 1 | 0 | 0 |
| Q2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Q2 | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 1 |
| Q1' | 0 | 1 | 1 | 0 | 1 | 0 | 0 | 0 |
| Q1' | 0 | 1 | 1 | 1 | 1 | 0 | 0 | 0 |
| Q2' | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Q2' | 1 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| | 1 | 0 | 1 | 0 | - | - | - | - |
| | 1 | 0 | 1 | 1 | - | - | - | - |
| | 1 | 1 | 0 | 0 | - | - | - | - |
| | 1 | 1 | 0 | 1 | - | - | - | - |
| | 1 | 1 | 1 | 0 | - | - | - | - |
| | 1 | 1 | 1 | 1 | - | - | - | - |

Figura 2.5: Tabella di verità delle uscite

Prodotta la tabella di verità delle uscite è immediata la stesura delle mappe di Karnaugh per l'automa realizzato, riportate in Figura 2.6.

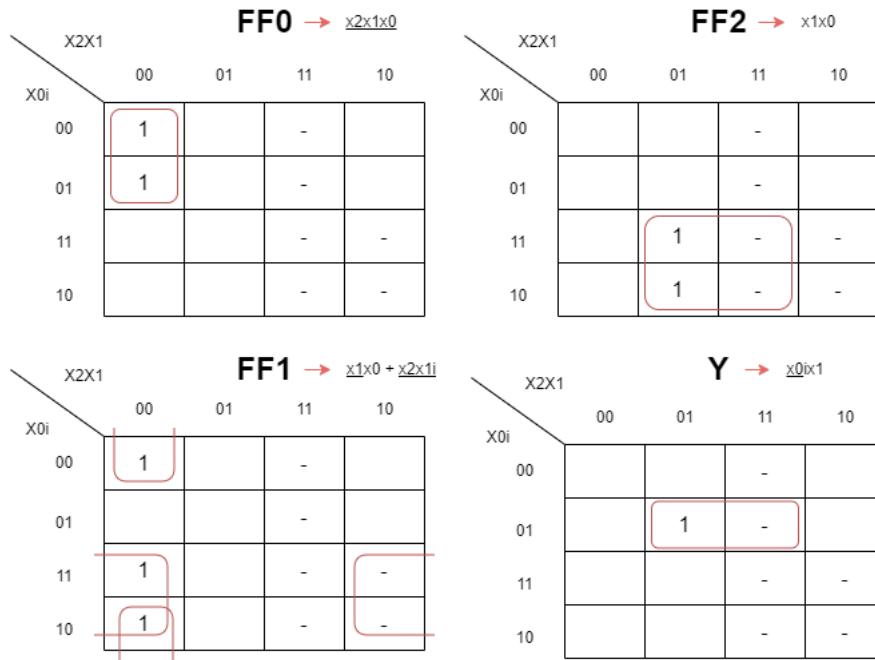


Figura 2.6: Mappe di Karnaugh delle uscite dei flip-flop e della macchina combinatoria

Dalle mappe si ricavano quindi i seguenti implicant:

- $FF0 = \bar{x}_2 \text{ AND } \bar{x}_1 \text{ AND } \bar{x}_0$
- $FF1 = (\bar{x}_1 \text{ AND } x_0) \text{ OR } (\bar{x}_2 \text{ AND } \bar{x}_1 \text{ AND } \bar{i})$
- $FF2 = x_1 \text{ AND } x_0$
- $Y = \bar{x}_0 \text{ AND } i \text{ AND } x_1$

2.4 Codice

2.4.1 Soluzione a : descrizione comportamentale

Per la soluzione a singolo process, ci si è basati unicamente sull'automa, dal momento che la logica della parte sequenziale non era prevista. Di seguito il codice implementato.

```

entity riconoscitore_beh is
    Port ( i : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           y : out STD_LOGIC);
end riconoscitore_beh;

architecture Behavioral of riconoscitore_beh is
    type stato is (Q0,Q1,Q2,Q1p,Q2p);
    signal stato_corrente: stato := Q0;

```

```

begin
    stato_uscita_mem : process(clock)
begin
    if (clock'event and clock='1') then
        if (reset = '1') then
            stato_corrente <= Q0;
            y<= '0';
        else
            case stato_corrente is
                when Q0=>
                    if (i='0') then
                        stato_corrente<=Q1p;
                        y<= '0';
                    else
                        stato_corrente<=Q1;
                        y<='0';
                    end if;
                when q1=>
                    if (i='0') then
                        stato_corrente<=Q2;
                        y<= '0';
                    else
                        stato_corrente<=Q2;
                        y<='0';
                    end if;
                when Q2=>
                    if (i='0') then
                        stato_corrente<=Q0;
                        y<= '0';
                    else
                        stato_corrente<=Q0;
                        y<='1';
                    end if;
                when Q1p=>
                    if (i='0') then
                        stato_corrente<=Q2p;
                        y<= '0';
                    else
                        stato_corrente<=Q2p;
                        y<='0';
                    end if;
                when Q2p=>
                    if (i='0') then
                        stato_corrente<=Q0;
                        y<= '0';
                    else
                        stato_corrente<=Q0;

```

```

        y<='0';
    end if;
when others =>
    stato_corrente<=Q0;
    y<='0';
end case;
end if;
end if;
end process;

end Behavioral;
```

2.4.2 Soluzione b : descrizione strutturale

Al fine di realizzare la scatola esterna, si è scelto di implementare separatamente due componenti: il primo rappresenta la **memoria di stato** della macchina, composta da tre flip-flop di tipo D, uno per ogni bit che codifica lo stato, e il secondo rappresenta la **parte combinatoria** della macchina, che è formata dalla combinazione di porte AND, NOT e OR così come vengono definite dalle mappe di Karnaugh nella fase di sintesi.

Flip-Flop D

```

entity flipflopD is
    Port ( d : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           o : out STD_LOGIC);
end flipflopD;

architecture Behavioral of flipflopD is

begin
    process(clock)
    begin
        if (clock'event AND clock='1') then
            if (reset = '1') then
                o<='0';
            else
                o<=D;
            end if;
        end if;
    end process;

end Behavioral;
```

Operatori AND, OR e NOT

```
entity op_AND is
    Generic ( n : positive := 2);
    Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
           c : out STD_LOGIC);
end op_AND;

architecture Behavioral of op_AND is

begin
    and_f : process(a)
        variable r: std_logic;
    begin
        r:='1';
        for i in 0 to n-1 loop
            r:=r and a(i);
        end loop;
        c<=r;
    end process;

end Behavioral;

entity op_NOT is
    Port ( a : in STD_LOGIC;
           c : out STD_LOGIC);
end op_NOT;

architecture Behavioral of op_NOT is

begin
    c <= not a;

end Behavioral;

entity op_OR is
    Port ( a : in STD_LOGIC;
           b : in STD_LOGIC;
           c : out STD_LOGIC);
end op_OR;

architecture Behavioral of op_OR is

begin
    c <= a or b;

end Behavioral;
```

Memoria

```
entity memoria is
    Port ( x : in STD_LOGIC_VECTOR (2 downto 0);
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           xp : out STD_LOGIC_VECTOR (2 downto 0));
end memoria;

architecture structural of memoria is

component flipflopD is
    Port ( d : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           o : out STD_LOGIC);
end component;

begin
    FFO : flipflopD
        port map(
            d => x(0),
            clock => clock,
            reset => reset,
            o => xp(0)
        );
    FF1 : flipflopD
        port map(
            d => x(1),
            clock => clock,
            reset => reset,
            o => xp(1)
        );
    FF2 : flipflopD
        port map(
            d => x(2),
            clock => clock,
            reset => reset,
            o => xp(2)
        );
end structural;
```

Rete combinatoria

```
entity rete_comb is
    Port ( x : out STD_LOGIC_VECTOR (2 downto 0);
           i : in STD_LOGIC;
```

```

y : out STD_LOGIC;
xp : in STD_LOGIC_VECTOR (2 downto 0));
end rete_comb;

architecture Structural of rete_comb is
component op_AND is
Generic ( n : positive := 2);
Port ( a : in STD_LOGIC_VECTOR(n-1 downto 0);
c : out STD_LOGIC);
end component;
component op_NOT is
Port ( a : in STD_LOGIC;
c : out STD_LOGIC);
end component;
component op_OR is
Port ( a : in STD_LOGIC;
b : in STD_LOGIC;
c : out STD_LOGIC);
end component;
begin
not_x0 : op_NOT
Port map(
a => xp(0),
c => not_xp(0)
);
not_x1 : op_NOT
Port map(
a => xp(1),
c => not_xp(1)
);
not_x2 : op_NOT
Port map(
a => xp(2),
c => not_xp(2)
);
not_ing : op_NOT
Port map(
a => i,
c => not_i
);
FF0 : op_AND generic map(3)

```

```

Port map(
    a(0) => not_xp(0),
    a(1) => not_xp(1),
    a(2) => not_xp(2),
    c => x(0)
);
FF1_temp1 : op_AND generic map(2)
Port map(
    a(0) => not_xp(1),
    a(1) => xp(0),
    c => temp1
);
FF1_temp2 : op_AND generic map(3)
Port map(
    a(0) => not_xp(2),
    a(1) => not_xp(1),
    a(2) => not_i,
    c => temp2
);
FF1 : op_OR
Port map(
    a => temp1,
    b => temp2,
    c => x(1)
);
FF2 : op_AND generic map(2)
Port map(
    a(0) => xp(0),
    a(1) => xp(1),
    c => x(2)
);
Uscita : op_AND generic map(3)
Port map(
    a(0) => not_xp(0),
    a(1) => xp(1),
    a(2) => i,
    c => y
);
end Structural;

```

Riconoscitore

```

entity riconoscitore is
  Port( i : in std_logic;
        clock : in std_logic;
        y : out std_logic;
        reset : in std_logic

```

```

    );
end riconoscitore;

architecture structural of riconoscitore is
begin
    component memoria is
        Port ( x : in STD_LOGIC_VECTOR (2 downto 0);
               clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               xp : out STD_LOGIC_VECTOR (2 downto 0));
    end component;

    component rete_comb is
        Port ( x : out STD_LOGIC_VECTOR (2 downto 0);
               i : in STD_LOGIC;
               y : out STD_LOGIC;
               xp : in STD_LOGIC_VECTOR (2 downto 0));
    end component;

    component flipflopD is
        Port ( d : in STD_LOGIC;
               clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               o : out STD_LOGIC);
    end component;

    signal x_comb : std_logic_vector(2 downto 0);
    signal xp_in : std_logic_vector(2 downto 0);
    signal in_buff : std_logic := '0';

    RC : rete_comb
        port map(
            x => xp_in,
            xp => x_comb,
            i => i,
            y => in_buff
        );
    MEM : memoria
        port map(
            clock => clock,
            reset => reset,
            x => xp_in,
            xp => x_comb
        );
    BUFF : flipflopD
        port map(
            clock => clock,

```

```

        reset => reset,
        d => in_buff,
        o => y
    );
}

end structural;

```

2.5 Simulazione

Per la simulazione di tale codice di entrambe le soluzioni, è stato creato un unico testbench apposito, che ha prodotto lo schema dei segnali in Figura 2.7, identico in entrambe le versioni come ci si aspettava.

```

entity riconoscitore_tb is
end riconoscitore_tb;

architecture Behavioral of riconoscitore_tb is
component riconoscitore_beh is
    Port( i : in std_logic;
          clock : in std_logic;
          y : out std_logic;
          reset : in std_logic
    );
end component;

signal x : std_logic := '0';
signal y : std_logic := '0';
signal clock : std_logic := '0';
signal reset : std_logic := '0';

constant clk_per : time := 10 ns;

begin
    uut: riconoscitore_beh port
        map (
            i=> x,
            clock => clock,
            reset => reset,
            y => y
        );

    clk_process : process
    begin
        clock <= '0';
        wait for clk_per/2;

```

```

    clock <= '1';
    wait for clk_per/2;
end process;

process1 : process
begin
    reset <= '1';
    wait for 20 ns;
    reset <= '0';

    --uscita alta
    x <= '1';
    wait for clk_per;
    x <= '0';
    wait for clk_per;
    x <= '1';
    assert y ='1';

    x <= '1';
    wait for clk_per;
    x <= '1';
    wait for clk_per;
    x <= '1';
    assert y ='1';

    --uscita bassa
    --001
    x <= '0';
    wait for clk_per;
    x <= '0';
    wait for clk_per;
    x <= '1';
    assert y ='1';

    --011
    x <= '0';
    wait for clk_per;
    x <= '1';
    wait for clk_per;
    x <= '1';
    assert y ='1';

    --010
    x <= '0';
    wait for clk_per;
    x <= '1';
    wait for clk_per;

```

```

x <= '0';
assert y ='1';

--000
x <= '0';
wait for clk_per;
x <= '0';
wait for clk_per;
x <= '0';
assert y ='1';

end process;
end Behavioral;

```

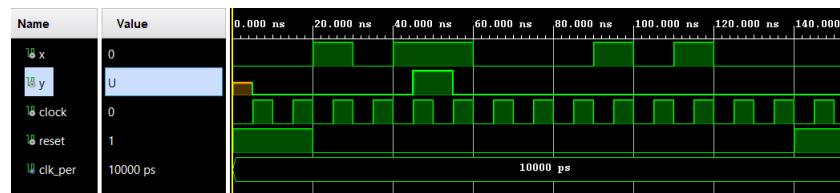


Figura 2.7: Simulazione del riconoscitore 1-1

Capitolo 3

Esercizio 2.2

3.1 Traccia

Si vuole progettare un riconoscitore di sequenza come macchina sincrona a sincronizzazione esterna: la macchina fornisce uscita alta quando viene riconosciuta la sequenza **1-10**, e le sequenze possono sovrapporsi (esempio: la sequenza 11101010 produrrebbe un'uscita alta in corrispondenza del quarto, sesto e ottavo bit). Si disegni l'automa e si progetti la macchina utilizzando **flip-flop D**. Implementare la macchina in VHDL utilizzando

- a) una descrizione di tipo comportamentale che faccia uso di due processi, uno che realizza la funzione di uscita e di transizione e l'altro che rappresenta la memoria di stato,
- b) una descrizione ibrida in cui le funzioni di uscita/transizione vengano realizzate mediante un modello di astrazione di tipo dataflow e la memoria di stato (i flip-flop) sia realizzata mediante una descrizione comportamentale.

3.2 Soluzioni

La soluzione a è stata implementata seguendo lo stesso procedimento della parte a dell'esercizio precedente. Si è andato a definire l'automa del riconoscitore per definire le transizioni di stato e i valori dell'uscita, e si è quindi proceduto a definire in due processi separati l'esecuzione di tale automa.

La soluzione b, invece, è stata definita a partire dalla descrizione delle componenti interne, come i flip-flop di tipo D, la memoria di stato da essi composta, e la rete combinatoria, che ingloba le porte AND, OR e NOT opportunamente collegate.

3.3 Schematici

Per quel che riguarda la soluzione di tipo a, in Figura 3.1 è possibile vedere l'automa per il riconoscitore in esame. A valle della definizione dell'automa, si è proceduto a codificare gli stati

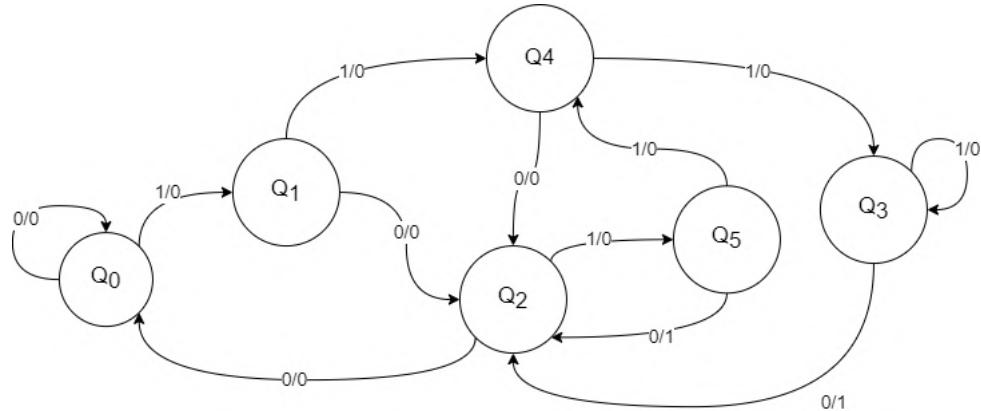


Figura 3.1: Diagramma degli stati

con i necessari 3 bit di stato, e a produrre la relativa tabella di transizione, visibile in Figura 3.2. Successivamente, da tali tabelle si è proceduto a definire la tabella di verità dei vari flip-flop che

| | INGRESSO | | | |
|-------|----------|------|-------|----------|
| STATO | 0 | 1 | STATO | CODIFICA |
| Q0 | Q0 | Q1 | Q0 | 000 |
| Q1 | Q2 | Q4 | Q1 | 001 |
| Q2 | Q0 | Q5 | Q2 | 010 |
| Q3 | Q2/1 | Q3/0 | Q3 | 011 |
| Q4 | Q2 | Q3 | Q4 | 100 |
| Q5 | Q2/1 | Q4/0 | Q5 | 101 |

Figura 3.2: Tabella di transizione e codifica degli stati

compongono la memoria di stato e dell'uscita, producendo la tabella, indicata in Figura 3.3. Da tale tabella è possibile facilmente estrarre le mappe di Karnaugh, necessario per la definizione delle uscite di memoria e macchina combinatoria in termini di implicanti.

Tali mappe sono espresse in Figura 3.4.

| | BIT DI STATO | | | IN | USCITE FLIP-FLOP | | | OUT |
|-------|--------------|----|----|----|------------------|-----|-----|-----|
| STATO | x2 | x1 | x0 | i | FF2 | FF1 | FF0 | Y |
| Q0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Q0 | 0 | 0 | 0 | 1 | 0 | 0 | 1 | 0 |
| Q1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |
| Q1 | 0 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| Q2 | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 |
| Q2 | 0 | 1 | 0 | 1 | 1 | 0 | 1 | 0 |
| Q3 | 0 | 1 | 1 | 0 | 0 | 1 | 0 | 1 |
| Q3 | 0 | 1 | 1 | 1 | 0 | 1 | 1 | 0 |
| Q4 | 1 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| Q4 | 1 | 0 | 0 | 1 | 0 | 1 | 1 | 0 |
| Q5 | 1 | 0 | 1 | 0 | 0 | 1 | 0 | 1 |
| Q5 | 1 | 0 | 1 | 1 | 1 | 0 | 0 | 0 |
| | 1 | 1 | 0 | 0 | - | - | - | - |
| | 1 | 1 | 0 | 1 | - | - | - | - |
| | 1 | 1 | 1 | 0 | - | - | - | - |
| | 1 | 1 | 1 | 1 | - | - | - | - |

Figura 3.3: Tabella di verità della memoria di stato e dell'uscita

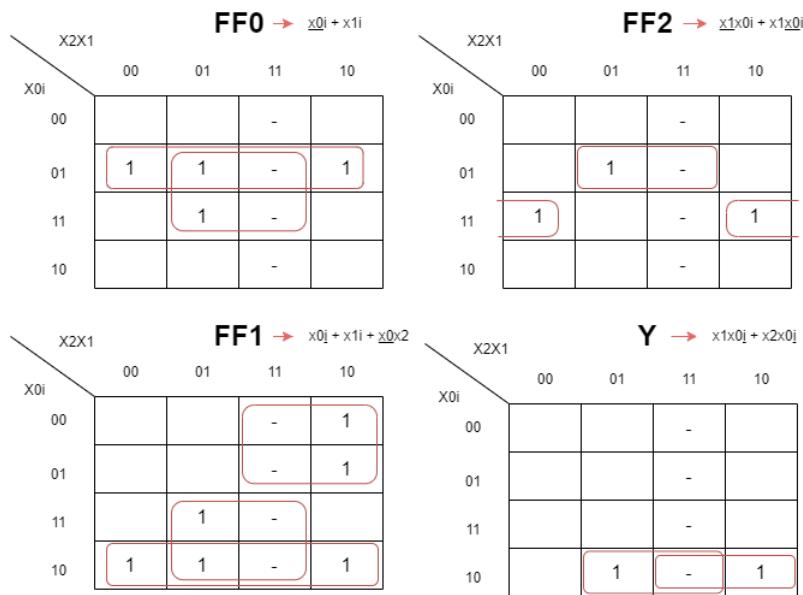


Figura 3.4: Mappe di Karnaugh della macchina combinatoria

Dalle mappe si ricavano quindi i seguenti implicanti:

- $FF0 = (\bar{x}_0 \text{ AND } i) \text{ OR } (x_1 \text{ AND } i)$

- $FF1 = (x_0 \text{ AND } \bar{i}) \text{ OR } (x_1 \text{ AND } i) \text{ OR } (\bar{x}_0 \text{ AND } x_2)$
- $FF2 = (\bar{x}_1 \text{ AND } x_0 \text{ AND } i) \text{ OR } (x_1 \text{ AND } \bar{x}_0 \text{ AND } i)$
- $Y = (x_1 \text{ AND } x_0 \text{ AND } \bar{i}) \text{ OR } (x_2 \text{ AND } x_0 \text{ AND } \bar{i})$

3.4 Codice

3.4.1 Soluzione a : descrizione comportamentale

```

entity riconoscitore_1proc is
    Port ( i : in STD_LOGIC;
           clock : in STD_LOGIC;
           y : out STD_LOGIC;
           reset : in STD_LOGIC);
end riconoscitore_1proc;

architecture Behavioral of riconoscitore_1proc is

    type stato is (Q0,Q1,Q2,Q3,Q4,Q5);
    signal stato_corrente: stato := Q0;
    signal stato_prossimo: stato;
    signal output: STD_LOGIC;

begin
    stati : process(stato_corrente, i)
    begin
        case stato_corrente is
            when Q0 =>
                if (i = '0') then
                    stato_prossimo <= Q0;
                    output <= '0';
                else
                    stato_prossimo <= Q1;
                    output <= '0';
                end if;
            when Q1 =>
                if (i = '0') then
                    stato_prossimo <= Q2;
                    output <= '0';
                else
                    stato_prossimo <= Q4;
                    output <= '0';
                end if;
            when Q2 =>
                if (i = '0') then
                    stato_prossimo <= Q0;

```

```

        output <= '0';
    else
        stato_prossimo <= Q5;
        output <= '0';
    end if;
when Q3 =>
    if (i = '0') then
        stato_prossimo <= Q2;
        output <= '1';
    else
        stato_prossimo <= Q3;
        output <= '0';
    end if;
when Q4 =>
    if (i = '0') then
        stato_prossimo <= Q2;
        output <= '0';
    else
        stato_prossimo <= Q3;
        output <= '0';
    end if;
when Q5 =>
    if (i = '0') then
        stato_prossimo <= Q2;
        output <= '1';
    else
        stato_prossimo <= Q4;
        output <= '0';
    end if;
when others =>
    stato_prossimo <= Q0;
    output <= '0';
end case;
end process;

transizione : process(clock)
begin
    if(clock'event AND clock='1') then
        if (reset='1') then
            stato_corrente<=Q0;
            y<='0';
        else
            stato_corrente<=stato_prossimo;
            y<=output;
        end if;
    end if;
end process;

```

```
end Behavioral;
```

3.4.2 Soluzione b : descrizione ibrida

Flip-Flop D

```
entity flipflopD is
    Port ( d : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           o : out STD_LOGIC);
end flipflopD;

architecture Behavioral of flipflopD is

begin
    process(clock)
    begin
        if (clock'event AND clock='1') then
            if (reset = '1') then
                o<='0';
            else
                o<=d;
            end if;
        end if;
    end process;
end Behavioral;
```

Memoria

```
entity memoria is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           x : in STD_LOGIC_VECTOR (2 downto 0);
           xp : out STD_LOGIC_VECTOR (2 downto 0));
end memoria;

architecture structural of memoria is
    component flipflopD is
        Port ( d : in STD_LOGIC;
               clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               o : out STD_LOGIC);
    end component;

begin
```

```
FF0 : flipflopD
```

```
  port map(
    d => x(0),
    clock => clock,
    reset => reset,
    o => xp(0)
  );
```

```
FF1 : flipflopD
```

```
  port map(
    d => x(1),
    clock => clock,
    reset => reset,
    o => xp(1)
  );
```

```
FF2 : flipflopD
```

```
  port map(
    d => x(2),
    clock => clock,
    reset => reset,
    o => xp(2)
  );
```

```
end structural;
```

Rete Combinatoria

```
entity rete_comb is
  Port ( i : in STD_LOGIC;
         xp : in std_logic_vector(2 downto 0);
         x : out std_logic_vector(2 downto 0);
         y : out STD_LOGIC);
end rete_comb;

architecture dataflow of rete_comb is

begin
  x(0) <= ((not xp(0)) AND i) OR (xp(1) AND i);
  x(1) <= (xp(0) AND (not i)) OR (xp(1) AND i) OR ((not xp(0)) AND xp(2));
  x(2) <= ((not xp(1)) AND xp(0) AND i) OR (xp(1) AND (not xp(0)) AND i);
  y <= (xp(1) AND xp(0) AND (not i)) OR (xp(2) AND xp(0) AND (not i));

end dataflow;
```

Riconoscitore

```
entity riconoscitore_hyb is
  Port ( i : in STD_LOGIC;
         clock : in STD_LOGIC;
         reset : in STD_LOGIC;
```

```

        y : out STD_LOGIC);
end riconoscitore_hyb;

architecture Structural of riconoscitore_hyb is

    component rete_comb is
        Port ( i : in STD_LOGIC;
               xp : in std_logic_vector(2 downto 0);
               x : out std_logic_vector(2 downto 0);
               y : out STD_LOGIC);
    end component;

    component memoria is
        Port ( clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               x : in STD_LOGIC_VECTOR (2 downto 0);
               xp : out STD_LOGIC_VECTOR (2 downto 0));
    end component;

    component flipflopD is
        Port ( d : in STD_LOGIC;
               clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               o : out STD_LOGIC);
    end component;

    signal x_in_mem : std_logic_vector(2 downto 0);
    signal x_out_mem : std_logic_vector(2 downto 0);
    signal temp_out : std_logic := '0';

begin
    MEM : memoria
        port map(
            clock => clock,
            reset => reset,
            x => x_in_mem,
            xp => x_out_mem
        );

    RC : rete_comb
        port map(
            i => i,
            y => temp_out,
            xp => x_out_mem,
            x => x_in_mem
        );

```

```

FFout : flipflopD
port map(
    d => temp_out,
    clock => clock,
    reset => reset,
    o => y
);
end Structural;

```

3.5 Simulazione

Per la simulazione di entrambe le soluzioni, è stato utilizzato lo stesso file di test, che ha prodotto le medesime forme d'onda nella simulazione, visibili in Figura 3.5.

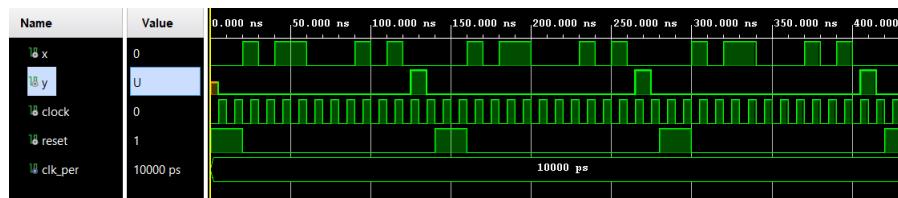


Figura 3.5: Simulazione del riconoscitore 1-10

Capitolo 4

Esercizio 3

4.1 Traccia

Progettare e implementare in VHDL un orologio che, a partire da un clock di riferimento di 50MHz che opera da base dei tempi, generi, mediante uso di contatori, il secondo, il minuto e l'ora. Utilizzare un approccio strutturale collegando opportunamente i contatori secondo uno schema a scelta. Il progetto deve prevedere la possibilità di inizializzare l'orologio con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set (l'ingresso di set può corrispondere ad un unico segnale oppure a tre segnali differenti, a scelta dello studente) e deve prevedere un ingresso di reset per azzerare il tempo.

Opzionale: il sistema deve acquisire un insieme di al massimo N intertempi in corrispondenza di un ingresso di stop. Ogni intertempo, nella forma ora—minuto—secondo, deve essere memorizzato in una memoria interna (registri).

4.2 Soluzione

L'orologio è stato realizzato attraverso l'utilizzo di un approccio strutturale: in particolare, sono stati utilizzati tre contatori in cascata. Il primo contatore conta i secondi, il secondo conta i minuti ed il terzo conta le ore. Tutti e tre contatori sono stati implementati secondo approccio comportamentale.

Per realizzare un clock di riferimento adeguato alla generazione dei secondi, a partire dal clock di riferimento di 50MHz, è stato introdotto nel progetto un divisore di frequenza (anch'esso descritto in maniera comportamentale), in modo tale da garantire un clock che oscilla a frequenza 1Hz, ideale per il conteggio dei secondi.

Con riferimento allo schema generale di funzionamento di tale struttura, il clock in uscita dal divisore di frequenza va quindi in ingresso al contatore dei secondi, mentre i contatori successivi ricevono il clock dai contatori precedenti.

4.3 Schematici

La struttura dell'orologio si compone di tre contatori, uno modulo 32 per il conteggio delle ore e due modulo 64 per il conteggio dei minuti, opportunamente resettati all'arrivo dei valori di ore, minuti e secondi massimi.

In aggiunta a tali contatori, collegati secondo una configurazione a cascata, in modo da far variare il conteggio di uno solo a valle del reset del contatore precedente, vi è un **divisore di frequenza** necessario affinché le commutazioni del contatore dei secondi avvengano ogni secondo. Il meccanismo di tale divisore di frequenza è molto semplice: ponendovi in ingresso un segnale di 50 MHz, come quello in esame, se fosse necessario generare un segnale che invece sia da 1 Hz, e che cioè esprima un periodo di 1 secondo, basterebbe contare il numero di fronti di salita del clock in ingresso e opportunamente variare l'uscita di tale divisore a valle del raggiungimento del numero di conteggi necessario.

Il divisore infatti sfrutta la seguente formula:

$$\text{conteggioMAX} = \frac{\text{freqIN}}{\text{freqOUT}} - 1 \quad (4.1)$$

dove *freqIN* rappresenta la frequenza del segnale che si vuole modificare, e *freqOUT* la frequenza desiderata per il segnale di uscita.

Si consideri però che è necessario che il segnale in uscita sia alto fino al raggiungimento della metà di tale conteggio, dal momento che il segnale da ottenere deve avere periodo pari ad 1.

La struttura di tale architettura è riportata in Figura 4.1.

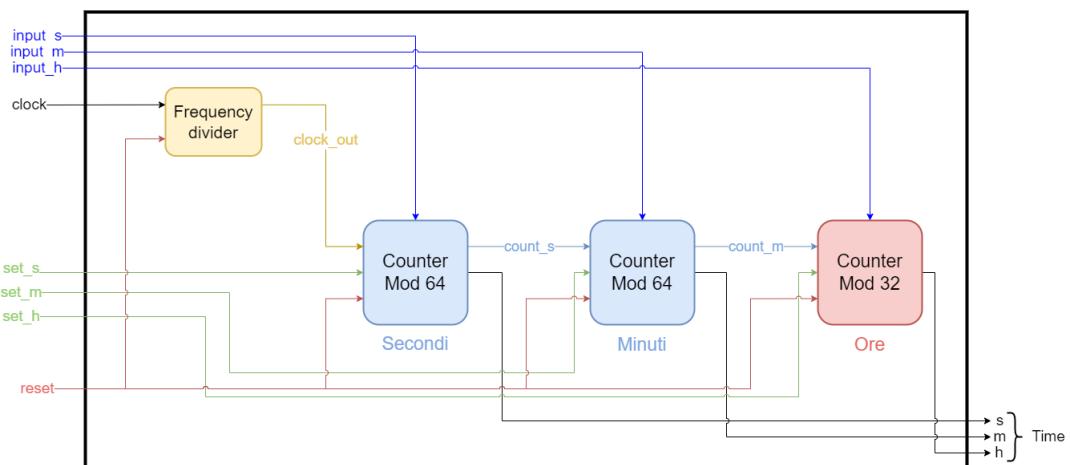


Figura 4.1: Schema strutturale dell'orologio

4.4 Codice

Divisore di frequenza

```
entity div_frequenza is
    generic(
        clock_frequency_in : integer;
        clock_frequency_out : integer);
    Port (
        clock_in : in STD_LOGIC;
        reset : in STD_LOGIC;
        clock_out : out STD_LOGIC);
end div_frequenza;

architecture Behavioral of div_frequenza is
    constant count_max_value : integer := clock_frequency_in/(clock_frequency_out)-1;
    signal clockfx : std_logic := '0';
    signal counter : integer range 0 to count_max_value := 0;
begin
    clock_out <= clockfx;
    count_for_division: process(clock_in, reset)
    begin
        if (reset = '1') then
            counter <= 0;
            clockfx <= '0';
        elsif (clock_in'event and clock_in = '1') then
            --metto la divisione della costante per 2 per poter inserire i
            --valori reali di clock nella zona generic quando viene usato
            --il componente
            if counter = count_max_value/2 then
                clockfx <= '1';
                counter <= 0;
            else
                clockfx <= '0';
                counter <= counter + 1;
            end if;
        end if;
    end process;
end Behavioral;
```

Contatore modulo M

```
entity cont_mod_M is
    generic(
        M : integer := 60 ;
        N : integer := 6
    );
```

```

Port (
    clock : in STD_LOGIC;
    set : in STD_LOGIC;
    input : in std_logic_vector(N-1 downto 0);
    reset : in STD_LOGIC;
    count : out STD_LOGIC;
    y : out std_logic_vector(N-1 downto 0)
);
end cont_mod_M;

architecture Behavioral of cont_mod_M is

signal internal_count : std_logic := '0';
signal internal_y : std_logic_vector(N-1 downto 0);
constant max : integer := M;

begin
    countProcess: process(clock, reset, set)
    begin
        -- il segnale di reset pone a 0 il contatore
        if (reset = '1') then
            internal_y<=(others => '0');
            internal_count<= '0';
        -- il segnale di set predisponde il contatore ad impostarsi su un valore in ingresso
        elsif (set='1') then
            if(conv_integer(input) > M-1) then --se il segnale di input è maggiore del conteggio massimo
                -- non posso superare il massimo valore di conteggio
                internal_y <= std_logic_vector(to_unsigned(M-1, internal_y' length));
            else
                --altrimenti pongo il segnale in ingresso al contatore
                internal_y <= input; --set assegna all'uscita il valore in input
            end if;
            internal_count <='0';
        elsif(clock 'event and clock ='1') then --se ho ricevuto un fronte di salita
            if(internal_y = std_logic_vector(to_unsigned(M-1, internal_y' length))) then
                internal_y <= (others => '0');
                internal_count <= '1';
            else
                internal_y <= internal_y + 1; --incremento l'uscita di conteggio
                internal_count <= '0';
            end if;
        end if;
    end process;
    y <= internal_y;
    count <= internal_count;
end Behavioral;

```

Orologio

```
entity orologio is
  Port (
    clock : in STD_LOGIC;
    set_s : in STD_LOGIC;
    set_m : in STD_LOGIC;
    set_h : in STD_LOGIC;
    input_s : in STD_LOGIC_VECTOR(5 downto 0);
    input_m : in STD_LOGIC_VECTOR(5 downto 0);
    input_h : in STD_LOGIC_VECTOR(4 downto 0);
    reset : in STD_LOGIC;
    y_time : out STD_LOGIC_VECTOR(16 downto 0)
  );
end orologio;

architecture structural of orologio is

  signal clock_out : std_logic := '0';
  signal count_s, count_m, count_h : std_logic := '0';
  signal secondi : std_logic_vector(5 downto 0) := (others =>'0');
  signal minuti : std_logic_vector(5 downto 0) := (others =>'0');
  signal ore : std_logic_vector(4 downto 0) := (others =>'0');

  component div_frequenza is
    generic(
      clock_frequency_in : integer := 50000000;
      clock_frequency_out : integer := 1
    );
    Port (
      clock_in : in STD_LOGIC;
      reset: in STD_LOGIC;
      clock_out : out STD_LOGIC
    );
  end component;

  component cont_mod_M is
    generic (
      M: integer;
      N: integer
    );
    Port (
      clock: in std_logic;
      set: in std_logic;
      input: in std_logic_vector(N-1 downto 0);
      reset: in std_logic;
      count: out std_logic;
      y: out std_logic_vector(N-1 downto 0)
    );
  end component;
```

```

end component;

begin
    divisore_di_frequenza: div_frequenza
        generic map(
            clock_frequency_in => 50000000,
            clock_frequency_out => 1
        )
        Port map(
            clock_in => clock,
            reset => reset,
            clock_out => clock_out
        );
    contatore_secondi : cont_mod_M
        generic map (
            M => 60,
            N => 6
        )
        Port map (
            clock => clock_out,
            set => set_s,
            input => input_s,
            reset => reset,
            count => count_s,
            y=> secondi
        );
    contatore_minuti : cont_mod_M
        generic map (
            M => 60,
            N => 6
        )
        Port map (
            clock => count_s,
            set => set_m,
            input => input_m,
            reset => reset,
            count => count_m,
            y=> minuti
        );
    contatore_ore : cont_mod_M
        generic map (
            M => 24,
            N => 5
        )
        Port map (
            clock => count_m,
            set => set_h,

```

```

        input => input_h,
        reset => reset,
        count => count_h,
        y=> ore
    );

y_time <= ore & minuti & secondi;

end structural;

```

4.5 Simulazione

Per la simulazione dell'orologio, è stato prodotto un testbench ad-hoc, riportato di seguito, dal quale sono state generate le forme d'onda visibili in Figura 4.2.

Al fine di visualizzare correttamente il funzionamento dell'orologio e il reset dei contatori al suo interno, si è deciso di verificare che, partendo dalle ore 23:59:59, tutti i contatori si resettassero così come voluto, e dalla simulazione si vede effettivamente che il valore registrato dai contatori al secondo successivo è proprio 00:00:00. Inoltre, il cambiamento di stato dell'orologio avviene ogni secondo, anche se il segnale di clock generato è ad una frequenza differente, a testimonianza del fatto che anche il divisore di frequenza lavora correttamente.

```

entity orologio_tb is
end orologio_tb;

architecture testbench of orologio_tb is

component orologio is
    Port (
        clock : in STD_LOGIC;
        set_s : in STD_LOGIC;
        set_m : in STD_LOGIC;
        set_h : in STD_LOGIC;
        input_s : in STD_LOGIC_VECTOR(5 downto 0);
        input_m : in STD_LOGIC_VECTOR(5 downto 0);
        input_h : in STD_LOGIC_VECTOR(4 downto 0);
        reset : in STD_LOGIC;
        y_time : out STD_LOGIC_VECTOR(16 downto 0)
    );
end component;

signal clock : STD_LOGIC;
signal set_s : STD_LOGIC;
signal set_m : STD_LOGIC;
signal set_h : STD_LOGIC;
signal input_s : STD_LOGIC_VECTOR(5 downto 0);

```

```

signal input_m : STD_LOGIC_VECTOR(5 downto 0);
signal input_h : STD_LOGIC_VECTOR(4 downto 0);
signal reset : STD_LOGIC;
signal y_time : STD_LOGIC_VECTOR(16 downto 0);
constant clk_period : time := 20000 ps;

begin

uut: orologio
port map(
    clock => clock,
    set_s => set_s,
    set_m => set_m,
    set_h => set_h,
    input_s => input_s,
    input_m => input_m,
    input_h => input_h,
    reset => reset,
    y_time => y_time
);

clock_proc : process
begin
    clock <= '0';
    wait for clk_period/2;
    clock <= '1';
    wait for clk_period/2;
end process;

stim_proc : process
begin
    wait for clk_period*300;
    set_s <= '1';
    set_m <= '1';
    set_h <= '1';
    input_s <= "111011";
    input_m <= "111011";
    input_h <= "10111";
    wait for clk_period*300;
    set_s <= '0';
    set_m <= '0';
    set_h <= '0';
    wait;
end process;

end testbench;

```

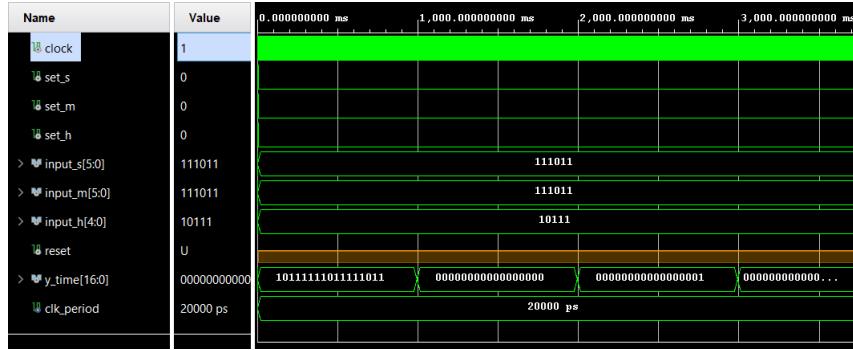


Figura 4.2: Simulazione dell’orologio

4.6 Sintesi su FPGA

Al fine di visualizzare gli output dell’orologio sui display della board, è necessario andare a modificare il progetto al fine di effettuare il corretto mapping tra segnali interni della board e variabili dei componenti costruiti. Per fare ciò si sono dovuti implementare dei moduli aggiuntivi, volti alla gestione dei catodi e degli anodi dei due display, e per l’utilizzo degli switch come dato in ingresso all’orologio.

Successivamente si è andati a modificare il file **constraints** per la board a disposizione, la Nexys A7-100T, al fine di codificare i segnali con l’hardware di tale scheda. In particolare sono stati necessari i seguenti componenti:

- **1 pin** per la generazione del segnale di clock;
- **6 switches** per il settaggio dei valori in secondi, minuti e ore;
- **6 catodi e anodi** per l’accensione e lo spegnimento dei display a 7 segmenti;
- **4 pulsanti** rispettivamente per il reset della macchina, e per i tre segnali di set di secondi, minuti e ore.

Di seguito vengono riportate le configurazioni all’interno del file di constraints.

```
## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMS3 } [get_ports { clock }];
create_clock -add -name sys_clk_pin -period 10.00 -waveform {0 5} [get_ports {clock}];

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMS3 } [get_ports { VALUE[0] }];
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMS3 } [get_ports { VALUE[1] }];
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMS3 } [get_ports { VALUE[2] }];
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMS3 } [get_ports { VALUE[3] }];
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMS3 } [get_ports { VALUE[4] }];
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMS3 } [get_ports { VALUE[5] }];
```

```

##7 segment display

set_property -dict { PACKAGE_PIN T10    IOSTANDARD LVCMOS33 } [get_ports { cat[0] }];
set_property -dict { PACKAGE_PIN R10    IOSTANDARD LVCMOS33 } [get_ports { cat[1] }];
set_property -dict { PACKAGE_PIN K16    IOSTANDARD LVCMOS33 } [get_ports { cat[2] }];
set_property -dict { PACKAGE_PIN K13    IOSTANDARD LVCMOS33 } [get_ports { cat[3] }];
set_property -dict { PACKAGE_PIN P15    IOSTANDARD LVCMOS33 } [get_ports { cat[4] }];
set_property -dict { PACKAGE_PIN T11    IOSTANDARD LVCMOS33 } [get_ports { cat[5] }];
set_property -dict { PACKAGE_PIN L18    IOSTANDARD LVCMOS33 } [get_ports { cat[6] }];
set_property -dict { PACKAGE_PIN H15    IOSTANDARD LVCMOS33 } [get_ports { cat[7] }];
set_property -dict { PACKAGE_PIN J17    IOSTANDARD LVCMOS33 } [get_ports { an[0] }];
set_property -dict { PACKAGE_PIN J18    IOSTANDARD LVCMOS33 } [get_ports { an[1] }];
set_property -dict { PACKAGE_PIN T9     IOSTANDARD LVCMOS33 } [get_ports { an[2] }];
set_property -dict { PACKAGE_PIN J14    IOSTANDARD LVCMOS33 } [get_ports { an[3] }];
set_property -dict { PACKAGE_PIN P14    IOSTANDARD LVCMOS33 } [get_ports { an[4] }];
set_property -dict { PACKAGE_PIN T14    IOSTANDARD LVCMOS33 } [get_ports { an[5] }];
set_property -dict { PACKAGE_PIN K2     IOSTANDARD LVCMOS33 } [get_ports { an[6] }];
set_property -dict { PACKAGE_PIN U13    IOSTANDARD LVCMOS33 } [get_ports { an[7] }];

##Buttons

set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 } [get_ports { reset }];
set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 } [get_ports { set_s }];
set_property -dict { PACKAGE_PIN M18    IOSTANDARD LVCMOS33 } [get_ports { set_m }];
set_property -dict { PACKAGE_PIN M17    IOSTANDARD LVCMOS33 } [get_ports { set_h }];

```

Il test del funzionamento del programma su board è stato fatto su differenti fasi:

Fase 1 : reset della macchina

Per effettuare il reset dell'orologio è stato utilizzato quindi il tasto **C12**. La Figura 4.3 mostra che a valle della pressione del tasto tutti i display vengono impostati a 0.

Fase 2 : set delle ore

Per effettuare il set delle ore, a valle dell'impostazione tramite gli switch da J15 a T18 del numero da impostare, è stato utilizzato il tasto **M17** per rendere effettivo tale settaggio. La Figura 4.4 mostra che a valle della pressione del tasto i display vengono impostati al numero desiderato.

Fase 3 : set dei minuti

Per effettuare il set dei minuti, a valle dell'impostazione tramite gli switch da J15 a T18 del numero da impostare, è stato utilizzato il tasto **M18** per rendere effettivo tale settaggio. La Figura 4.5 mostra che a valle della pressione del tasto i display vengono impostati al numero desiderato.

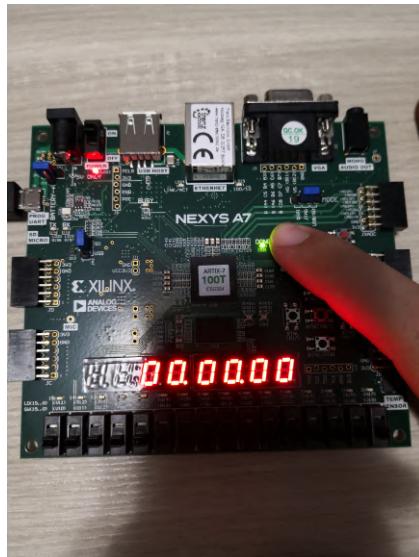


Figura 4.3: Reset fisico dell'orologio su scheda

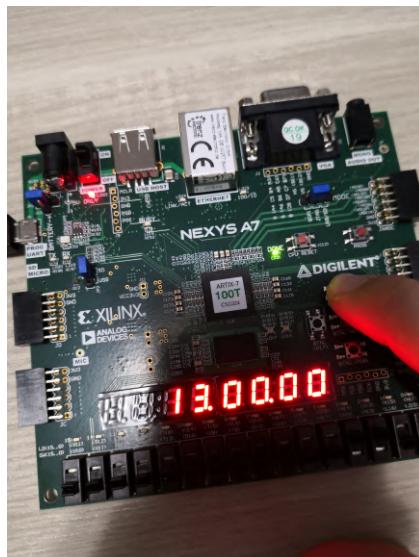


Figura 4.4: Set delle ore su scheda

Fase 4 : set dei secondi

Per effettuare il set dei secondi, a valle dell'impostazione tramite gli switch da J15 a T18 del numero da impostare, è stato utilizzato il tasto **N17** per rendere effettivo tale settaggio. La Figura 4.6 mostra che a valle della pressione del tasto i display vengono impostati al numero desiderato.

Fase 5 : avanzamento automatico

Se non viene effettuata alcuna pressione su nessuno dei bottoni abilitati, l'orologio procede a scandire secondi, minuti ed ore in maniera automatica. La Figura 4.7 mostra che, attendendo il

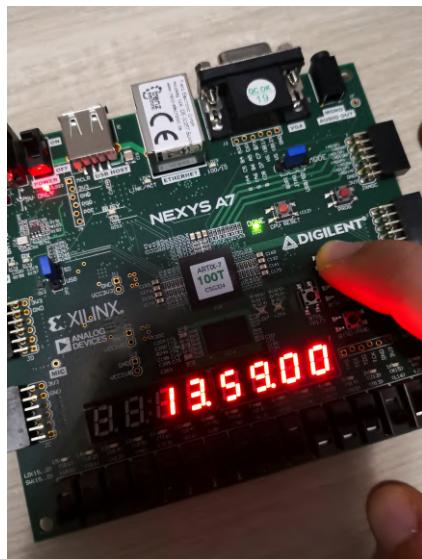


Figura 4.5: Set dei minuti su scheda

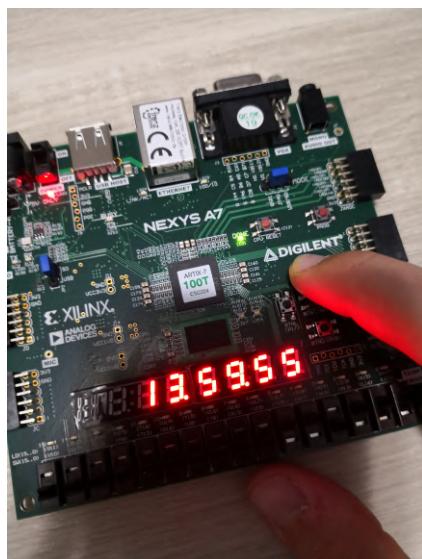


Figura 4.6: Set dei secondi su scheda

tempo necessario, l'avanzamento del tempo avviene in maniera corretta; infatti, avendo impostato come orario le 13:59:55, l'orologio si porta nello stato 14:00:05.

Fase 6 : reset automatico

Per mostrare l'effettivo reset di tutta la macchina, è stato impostato l'orario 23:59:59, come mostrato in Figura 4.8. Al secondo successivo si nota come la macchina ritorni nello stato 00:00:00 per poi proseguire la sua evoluzione in maniera automatica. La Figura 4.9 mostra che a valle della pressione del tasto i display vengono impostati al numero desiderato.



Figura 4.7: Avanzamento del tempo su scheda

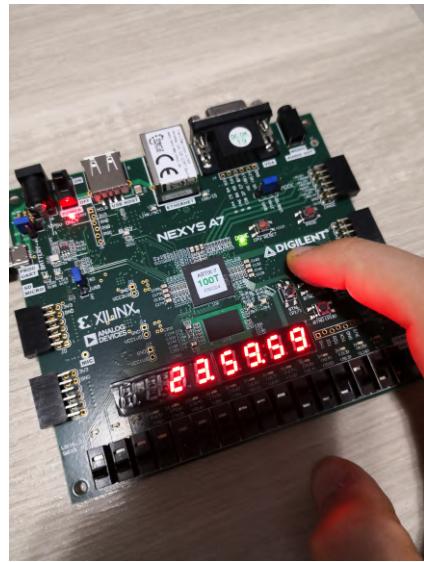


Figura 4.8: Settaggio delle ore 23:59:59 su scheda



Figura 4.9: Reset automatico dell'orologio su scheda

Capitolo 5

Esercizio 4

5.1 Traccia

Progettare un registro a scorrimento di 4 bit in grado di operare, in base ad una selezione, in 4 diverse modalità:

1. scorrimento **a sinistra** con caricamento seriale di un bit pari a 0;
2. scorrimento **a destra** con caricamento seriale di un bit pari a 0;
3. scorrimento **circolare** verso sinistra;
4. scorrimento **a sinistra** con caricamento seriale di un bit **x**.

Il valore iniziale del registro può essere configurato mediante un segnale di reset oppure tramite il caricamento parallelo di un valore $A_3A_2A_1A_0$ fornito dall'esterno, inserito grazie ad un segnale di load. Un segnale di shift regola lo scorrimento del registro. Si progetti e implementi il registro utilizzando un approccio:

- a) strutturale;
- b) comportamentale;

5.2 Soluzioni

Il problema di realizzazione di un registro a scorrimento dipende principalmente dal numero di bit in gioco: a seconda della lunghezza del registro è necessario utilizzare il giusto numero di Flip-Flop. In questo caso, dati i 4 bit in ingresso, è stato necessario definire quattro flip-flop di tipo D per la memorizzazione dei dati.

Inoltre, la particolarità di tale esercizio è la definizione delle differenti modalità di funzionamento del registro, date dal valore di un ingresso di selezione opportunamente codificato per la definizione

di tutte le modalità. Visto che le modalità di utilizzo sono 4, è stato necessario definire un ingresso di selezione di soli 2 bit, definendo come segue le varie modalità:

- Un ingresso di selezione pari a **00** definisce la modalità di **scorrimento a sinistra**;
- Un ingresso di selezione pari a **01** definisce la modalità di **scorrimento a destra**;
- Un ingresso di selezione pari a **10** definisce la modalità di **scorrimento circolare**;
- Un ingresso di selezione pari a **11** definisce la modalità di **scorrimento a sinistra di un bit x**;

All'interno dell'architettura del registro, è quindi necessario istanziare dei multiplexer per la definizione della modalità di funzionamento del registro, abilitati dall'ingresso di selezione definito in precedenza. Oltre a tali multiplexer, è stato necessario inserire all'interno dell'architettura dei multiplexer per la scelta del segnale di abilitazione dei flip-flop, dal momento che il registro prevede la modalità di caricamento del dato in parallelo all'interno del registro. Si è scelto quindi di definire l'abilitazione dei flip-flop a valle di una porta OR, tra i segnale di shift del registro e quello di load che abilita il caricamento parallelo.

Infine, per definire l'uscita a seconda della tipologia di funzionamento del registro, è stato ritenuto necessario andare ad inserire un multiplexer in uscita che selezionasse l'uscita al primo bit se il registro lavora con scorrimento a sinistra, o all'ultimo bit nel caso il tipo di scorrimento selezionato sia quello a destra.

5.3 Schematici

Come è stato descritto in precedenza, in Figura 5.1 è rappresentata la struttura generale del registro a scorrimento.

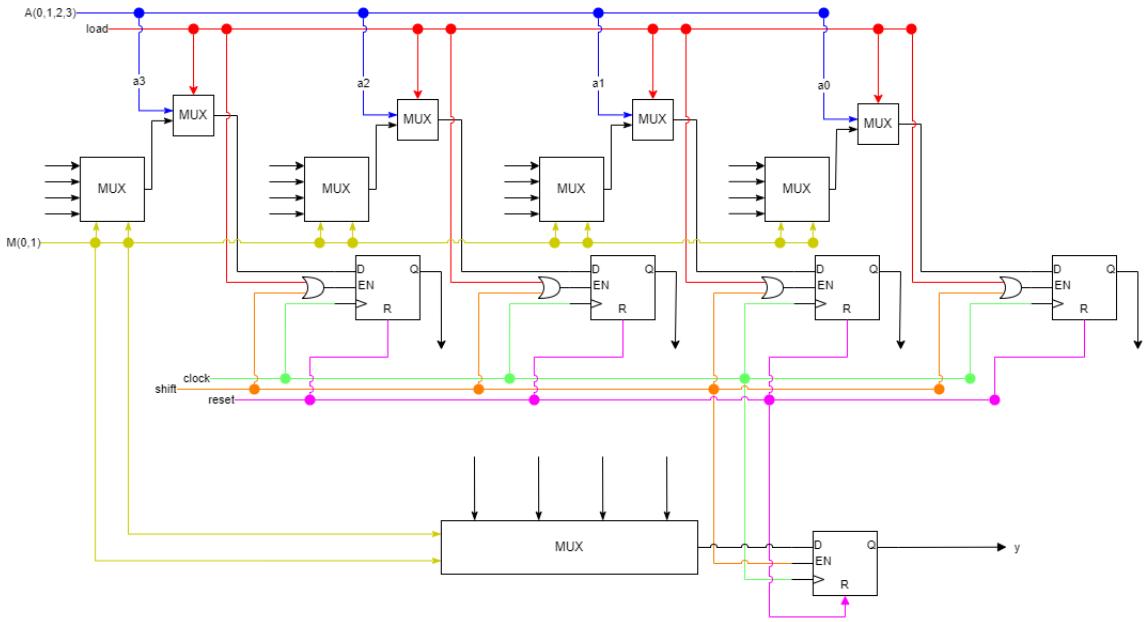


Figura 5.1: Struttura generale del registro a scorrimento

Vista la complessità dello schema, si è scelto di definire i sottoschemi dei singoli meccanismi di funzionamento del registro, visibili nelle Figure 5.2, 5.3, 5.4, 5.5.

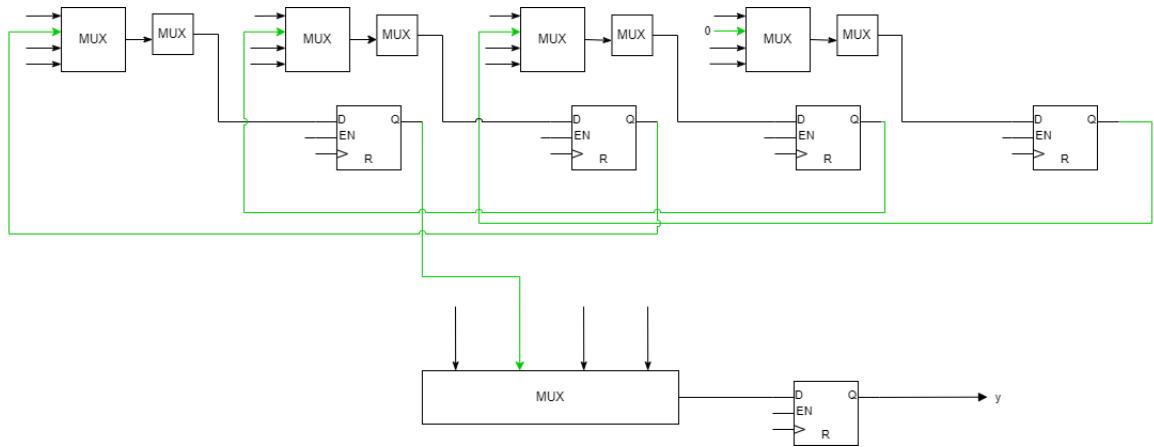


Figura 5.2: Shift a sinistra di un bit pari a 0

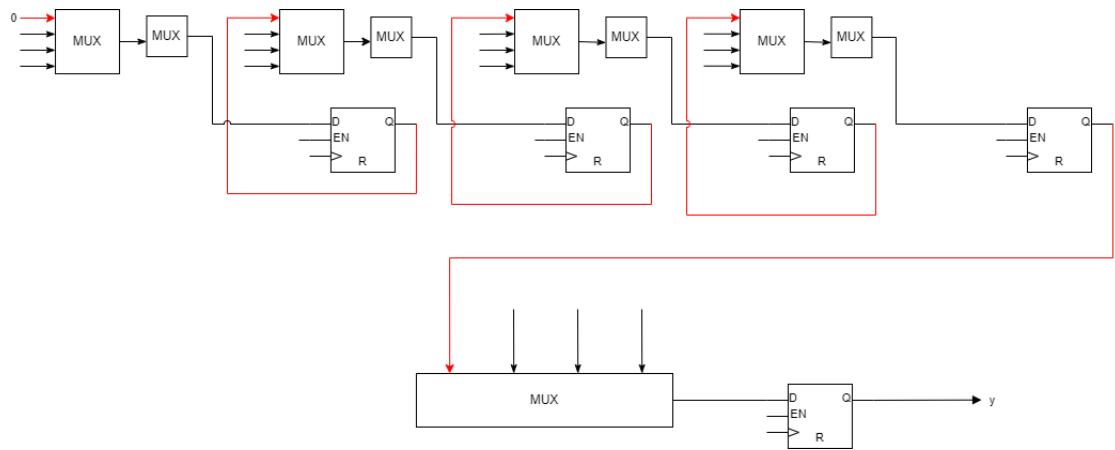


Figura 5.3: Shift a destra di un bit pari a 0

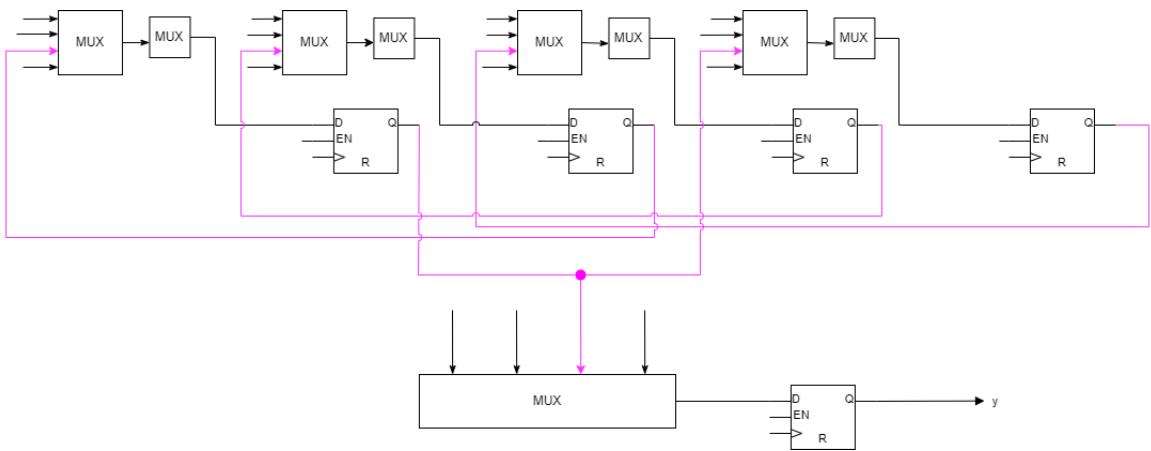


Figura 5.4: Shift circolare verso sinistra

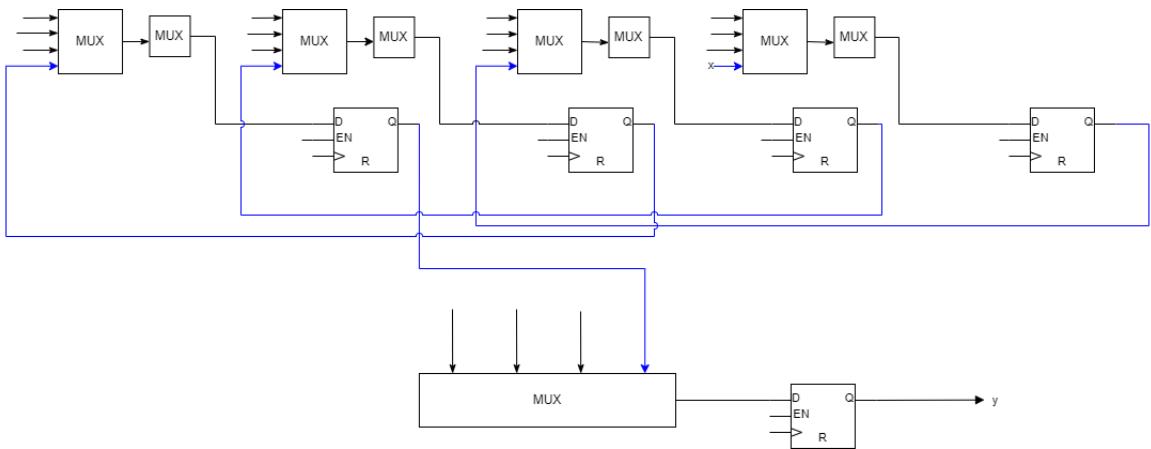


Figura 5.5: Shift a sinistra di un bit pari a X

5.4 Codice

5.4.1 Soluzione comportamentale

```
entity reg_scorrimento_beh is
    Port ( selection : in STD_LOGIC_VECTOR (0 to 1); --selezione della modalità
           x : in STD_LOGIC; --bit da caricare nella modalità 4
           clock : in STD_LOGIC; --segnale di temporizzazione
           reset : in STD_LOGIC; --reset dei FF
           load : in STD_LOGIC; --segnale di abilitazione del caricamento
           A : in STD_LOGIC_VECTOR (0 to 3); --dato da caricare
           y : out STD_LOGIC_VECTOR (0 to 3)); --segnale di output
end reg_scorrimento_beh;

architecture Behavioral of reg_scorrimento_beh is

    signal output : std_logic_vector(0 to 3);

begin
    process(clock, reset)
    begin
        if(reset = '1') then
            output <= "0000";
        elsif(clock'event and clock='0') then
            if(load = '1') then
                output <= A;
                -- '00' indica lo scorrimento a sinistra
            elsif(load = '0' and selection = "00") then
                output(3) <= '0';
                output(0 to 2) <= output(1 to 3);
                -- '01' indica lo scorrimento a destra
            elsif(load = '0' and selection = "01") then
                output(0) <= '0';
                output(1 to 3) <= output(0 to 2);
                -- '10' indica lo scorrimento circolare
            elsif(load = '0' and selection = "10") then
                output(3) <= output(0);
                output(0 to 2) <= output(1 to 3);
                -- '11' indica lo scorrimento di X a destra
            elsif(load = '0' and selection = "11") then
                output(3) <= x;
                output(0 to 2) <= output(1 to 3);
            else
                output <= "XXXX";
            end if;
        end if;
    end process;
```

```

y <= output;

end Behavioral;

```

5.4.2 Soluzione strutturale

La soluzione strutturale si compone di tre differenti componenti:

- Multiplexer 2:1 per la selezione del segnale di abilitazione dei flip-flop D tra segnale di shift e segnale di load;
- Multiplexer 4:1 per la scelta del dato in ingresso ai flip-flop D a seconda del tipo di modalità di scorrimento del registro;
- Flip-Flop D per la memorizzazione dei bit nel registro;
- Multiplexer 4:1 aggiuntivo a valle dell'architettura per la scelta del bit da mostrare in uscita al registro, al variare della modalità di scorrimento;
- Flip-Flop D in uscita per la sincronizzazione dell'uscita con il clock.

Multiplexer generic

```

entity multiplexer is
  generic(
    n : positive := 2
  );
  Port ( i : in STD_LOGIC_VECTOR (2**n downto 0);
         sel : in STD_LOGIC_VECTOR(n-1 downto 0);
         o : out STD_LOGIC
  );
end multiplexer;

architecture dataflow of multiplexer is

begin
  o <= i(2**n - to_integer(unsigned(sel)));
end dataflow;

```

Flip-Flop D

Rispetto al componente flip-flop utilizzato negli esercizi precedenti, tale implementazione presenta l'aggiunta di un segnale di enable.

```

entity flip_flop_D is
  Port ( clk : in STD_LOGIC;
         reset : in STD_LOGIC;

```

```

        en : in STD_LOGIC;
        data : in STD_LOGIC;
        output : out STD_LOGIC);
end flip_flop_D;

architecture Behavioral of flip_flop_D is

begin
    process(clk)
    begin
        if(reset = '1') then
            output <= '0';
        elsif(clk'event and clk = '0' and en ='1') then
            output <= data;
        end if;
    end process;

end Behavioral;

```

Registro a Scorrimento

```

entity registro_scorrimento is
    Port ( selection : in std_logic_vector (1 downto 0);
           x : in std_logic;
           clock : in std_logic;
           sh : in std_logic;
           reset : in std_logic;
           load : in std_logic;
           A : in std_logic_vector (3 downto 0);
           y : in std_logic);
end registro_scorrimento;

architecture structural of registro_scorrimento is
--Flip-Flop di tipo D
component flip_flop_D is
    Port ( clk : in std_logic;
           reset : in std_logic;
           en : in std_logic;
           data : in std_logic;
           output : out std_logic);
end component;
--Multiplexer generic
component multiplexer is
    generic(
        n : positive := 2
    );
    Port ( i : in std_logic_vector (2**n downto 0);

```

```

    sel : in std_logic_vector(n-1 downto 0);
    o : out std_logic
  );
end component;
--Segnali interni
signal s_ff_out : std_logic_vector(3 downto 0); --uscite dei flip-flop
signal s_ff_in : std_logic_vector(3 downto 0); --ingressi dei flip-flop
signal s_mux4_out : std_logic_vector(3 downto 0); --uscita del mux 4:1
signal s_y: std_logic; --segna le uscite temporaneo
signal en_or : std_logic; --segna l'abilitazione dei flip-flop (OR tra share load)
begin

  -- Multiplexer 4:1 per il Flip-Flop 3
  mux_4_1_3 : multiplexer generic map(2)
    port map(
      i(4) => s_ff_out(2),
      i(3) => '0',
      i(2) => s_ff_out(2),
      i(1) => s_ff_out(2),
      sel => selection,
      o => s_mux4_out(3)
    );
  -- Multiplexer 4:1 per il Flip-Flop 2
  mux_4_1_2 : multiplexer generic map(2)
    port map(
      i(4) => s_ff_out(1),
      i(3) => s_ff_out(3),
      i(2) => s_ff_out(1),
      i(1) => s_ff_out(1),
      sel => selection,
      o => s_mux4_out(2)
    );
  -- Multiplexer 4:1 per il Flip-Flop 1
  mux_4_1_1 : multiplexer generic map(2)
    port map(
      i(4) => s_ff_out(0),
      i(3) => s_ff_out(2),
      i(2) => s_ff_out(0),
      i(1) => s_ff_out(0),
      sel => selection,
      o => s_mux4_out(1)
    );
  -- Multiplexer 4:1 per il Flip-Flop 0
  mux_4_1_0 : multiplexer generic map(2)
    port map(
      i(4) => '0',
      i(3) => s_ff_out(1),

```

```

        i(2) => s_ff_out(3),
        i(1) => x,
        sel => selection,
        o => s_mux4_out(0)
    );
-- Ciclo for per la creazione dei Mux 2:1
mux_2_1_ciclo : for i in 0 to 3 generate
    mux_2_1_i : multiplexer generic map(1)
        port map(
            i(2) => s_mux4_out(i),
            i(1) => a(i),
            sel(0) => load,
            o => s_ff_in(i)
        );
    end generate;
-- Multiplexer 4:1 per l'uscita dei flip-flop
mux_4_1_out : multiplexer generic map(2)
    port map(
        i(4) => s_ff_out(3),
        i(3) => s_ff_out(0),
        i(2) => s_ff_out(3),
        i(1) => s_ff_out(3),
        sel => selection,
        o => s_y
    );
-- Generazione del segnale di enable dei flip-flop concorrente
en_or <= sh or load;
-- Ciclo per la creazione dei flip-flop del registro
ff_interni : for i in 0 to 3 generate
    ff_i : flip_flop_D
        port map(
            data => s_ff_in(i),
            clk => clock,
            reset => reset,
            en => en_or,
            output => s_ff_out(i)
        );
    end generate;
-- Flip-Flop per la sincronizzazione dell'uscita
ff_out : flip_flop_D
    port map(
        data => s_y,
        clk => clock,
        reset => reset,
        en => sh,
        output => y
    );

```

```
end structural;
```

5.5 Simulazione

Per la verifica del corretto funzionamento del registro, sia nella sua versione comportamentale che nella sua versione strutturale, è stato prodotto il testbench ad-hoc riportato di seguito. L'esecuzione della simulazione ha prodotto le forme d'onda riportate nelle Figure 5.6 e 5.7. Come è possibile vedere dall'implementazione della soluzione strutturale, si è scelto di definire un'uscita a singolo bit, per portare alla luce due differenti tipologie di uscita, seriale o parallela. In Figura 5.8 la simulazione ottenuta con la versione strutturale del registro.

```
entity reg_scorrimento_tb is
end reg_scorrimento_tb;

architecture testbench of reg_scorrimento_tb is

component reg_scorrimento_beh is
    Port ( selection : in STD_LOGIC_VECTOR (0 to 1);
           x : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           load : in STD_LOGIC;
           A : in STD_LOGIC_VECTOR (0 to 3);
           y : out STD_LOGIC_VECTOR (0 to 3));
end component;

signal selection : STD_LOGIC_VECTOR (0 to 1);
signal x : STD_LOGIC;
signal clock : STD_LOGIC;
signal reset : STD_LOGIC;
signal load : STD_LOGIC;
signal A : STD_LOGIC_VECTOR (0 to 3);
signal y : STD_LOGIC_VECTOR (0 to 3);

constant clk_period : time := 10 ns;

begin
    uut : reg_scorrimento_beh
        port map(
            selection => selection,
            x => x,
            clock => clock,
            reset => reset,
            load => load,
            A => A,
            y => y

```

```

);

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

stimulus : process
begin
    wait for clk_period;
    reset <='1';
    wait for clk_period;
    reset <='0';

    --Prova Modalita 1
    wait for clk_period;
    A<="1001";
    load <='1';
    wait for clk_period;
    load <='0';

    --Scorrimento a sinistra 00
    selection<="00";
    wait for 5*clk_period;

    --Prova Modalita 2
    wait for clk_period;
    A<="1010";
    load <='1';

    wait for clk_period;
    load <='0';
    --Scorrimento a destra 01
    selection<="01";
    wait for 5*clk_period;

    --Prova Modalita 3
    wait for clk_period;
    A<="1110";
    load <='1';
    wait for clk_period;
    load <='0';

    --Scorrimento circolare a sinistra 10

```

```

selection<="10";
wait for 5*clk_period;

--Prova Modalita 4
wait for clk_period;
A<="1000";
load <='1';
wait for clk_period;
load <='0';

--Scorrimento a sinistra con x 11
x<='1';
selection<="11";
wait for 5*clk_period;
reset <= '1';
wait for clk_period;
reset <='0';
wait;
end process;

end testbench;

```

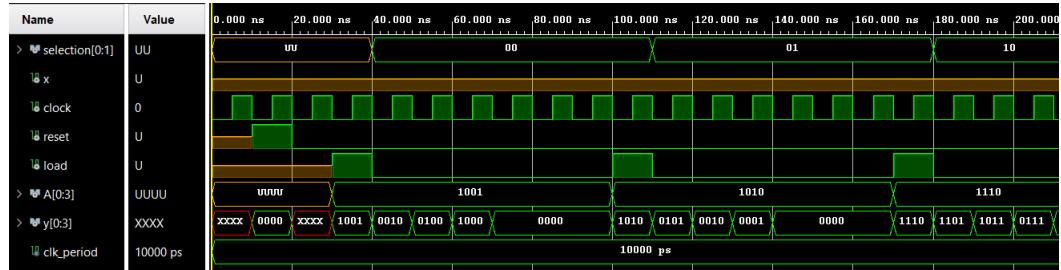


Figura 5.6: Simulazione del registro a scorrimento (1)

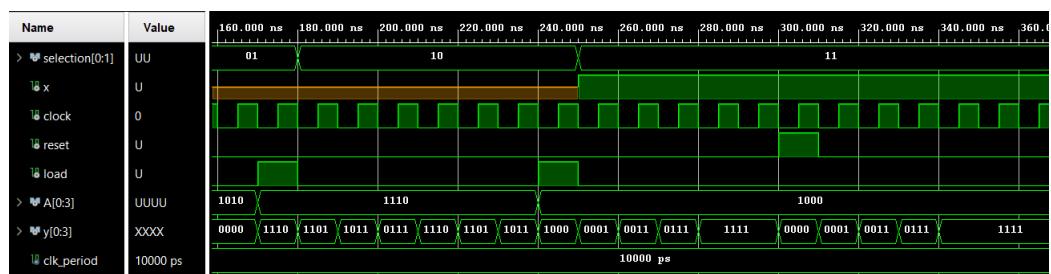


Figura 5.7: Simulazione del registro a scorrimento (2)

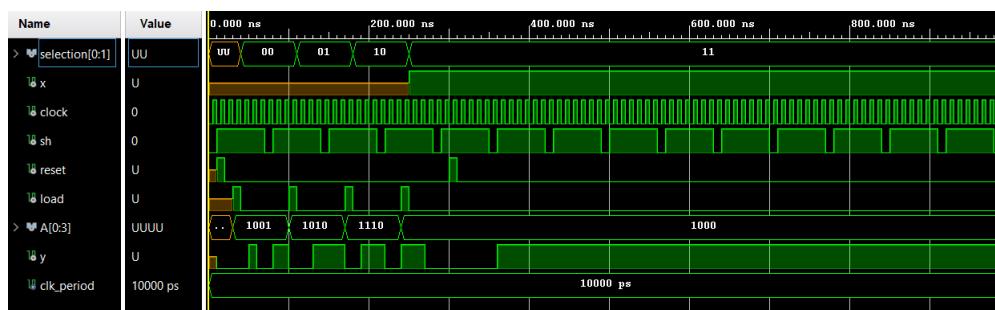


Figura 5.8: Simulazione del registro a scorrimento con uscita seriale

Capitolo 6

Esercizio 5

6.1 Traccia

Progettare e implementare in VHDL un sistema che, date due stringhe binarie A e B di 8 bit ciascuna acquisite mediante handshaking, calcoli il valore **A mod B**.

Il sistema deve essere progettato utilizzando un approccio modulare basato sull'individuazione della *parte operativa* e della *parte di controllo*, e la parte di controllo deve essere realizzata mediante (a) **logica cablata** e (b) **logica microprogrammata**.

Con riferimento alle modalità di acquisizione delle stringhe in input mediante handshaking, si discutano due diverse soluzioni possibili.

6.2 Soluzioni

Il sistema è stato realizzato effettuandone una decomposizione di funzionalità, pensando ad un modello che prevede la realizzazione di un'unità complessa composta da due sottosistemi: **unità di controllo (UC)** e **unità operativa (UO)**.

L'**unità operativa** è il sottosistema che si occupa di elaborare i dati in ingresso, sulla base degli ingressi di controllo forniti dall'unità di controllo, e fornisce in uscita uno stato che sarà utile all'unità di controllo per scandire le prossime operazioni da comandare al blocco di elaborazione.

L'unità operativa deve essere progettata prima dell'unità di controllo, in quanto la UC genera i segnali in ingresso alla UO.

L'**unità di controllo** è invece la parte del sistema che si occupa di coordinare le operazioni che deve compiere l'unità operativa tramite dei segnali di abilitazione. La logica di controllo dipende unicamente dal tipo di operazioni da effettuare. Si è scelto in questo caso, di modificare quindi unicamente l'unità di controllo, e rendere essa in grado di comunicare con l'unità esterna che fornisce i dati tramite il protocollo di **handshaking**. Il **protocollo di handshake** tra due sistemi necessita di due segnali per il suo corretto funzionamento: un segnale di richiesta del dato (REQ)

e un segnale di acknowledge per la terminazione delle operazioni.

Il ciclo di handshake funziona quindi in questo modo: il sistema A asserisce REQ per richiedere al sistema B un'informazione. Vedendo asserita la linea REQ, il sistema B elabora la richiesta ricevuta e, non appena termina il processo, asserisce la linea ACK, e in questa maniera il sistema A sa che il dato sul bus è valido e può campionarlo. Contemporaneamente, il sistema A deasserisce REQ e allora il sistema B rilascia il controllo del bus. In Figura 6.1 i segnali scambiati dai due sistemi.

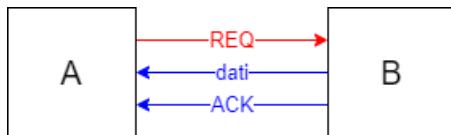


Figura 6.1: Funzionamento generale del protocollo di handshake

Il tipo di handshake realizzato viene detto **handshake non completo** ed **asincrono**, dal momento che non vi è alcun segnale di ack di ritorno dal sistema A al sistema B per la corretta ricezione delle informazioni, e dal momento che non è necessario che i due sistemi ricevano lo stesso segnale di clock. Nel caso in esame, l'handshaking si sarebbe potuto realizzare in due modi:

- Handshaking tra unità di controllo e sistema esterno: l'unità di controllo richiede al sistema esterno il dato da inoltrare all'unità operativa, e attende che gli venga notificata la terminazione dell'operazione da parte dell'unità esterna.
- Handshaking tra unità operativa e sistema esterno: l'unità esterna invia i dati all'unità operativa, e procede a notificare all'unità di controllo il corretto completamento dell'operazione, che a sua volta si occupa di notificarne la ricezione al sistema esterno.

Come definito in precedenza, si è scelto di procedere con la prima soluzione, in quanto è l'unica che richiede la modifica della sola logica di controllo, e permette di lasciare intatta la logica operativa del sistema.

6.3 Schematici

Si è scelto di mostrare dapprima la versione generica dell'architettura del sistema, composto da sistema esterno e sistema AmdB, formato da unità operativa e di controllo. In Figura 6.2 l'architettura del top module.

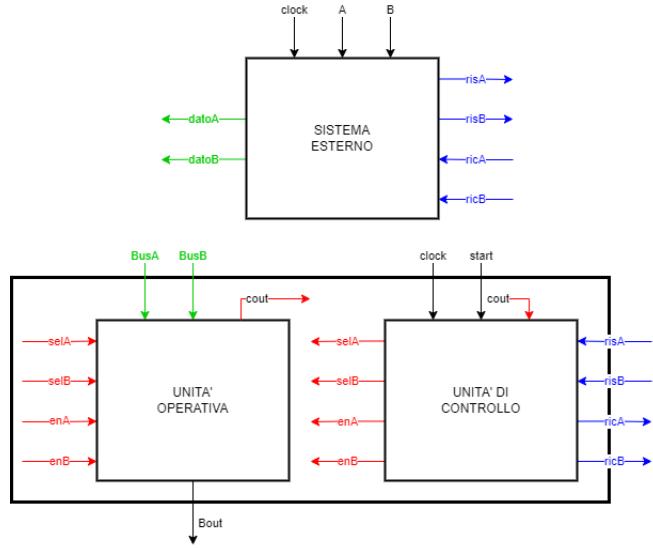


Figura 6.2: Architettura generale del sistema

6.3.1 Unità operativa

Visto che non è necessario la modifica della logica operativa, si è scelto di partire con il definire la struttura di quest'ultima: essa è composta da 2 multiplexer 2:1, abilitati da due segnali **selA** e **selB**, che servono per scegliere il dato in ingresso ai due registri A e B, che contengono i bit provenienti dal sistema esterno. A valle di due segnali di abilitazione, **enA** e **enB**, i dati inseriti all'interno dei due registri vengono inviati, con B negato, all'addizionatore, che si occupa di effettuare la somma tra A e (-B), producendo un uscita che viene riportata in ingresso al multiplexer del lato A finché tale uscita non sia negativa. In Figura 6.3 lo schema di funzionamento appena descritto.

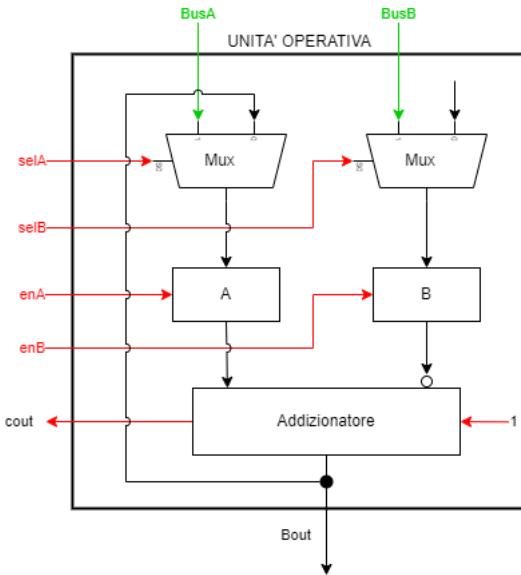


Figura 6.3: Architettura dell'unità operativa

6.3.2 Sistema Esterno

Per quel che riguarda il sistema esterno, esso è composto dai due registri contenenti i dati da inviare al sistema, e da un unità di controllo esterna, componente che gestisce l'handshaking con l'unità di controllo del sistema interno. L'unità di controllo esterna si occupa di abilitare la trasmissione dei dati nei registri, comunicando con essi tramite dei segnali di abilitazione. In Figura 6.4 l'architettura del sistema esterno.

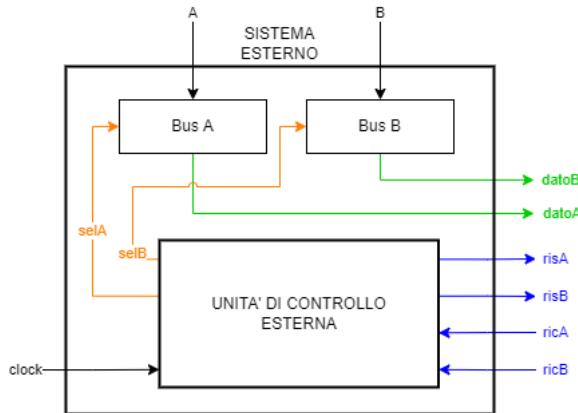


Figura 6.4: Architettura del sistema esterno

Il funzionamento dell'unità esterna può essere descritto dall'automa in Figura 6.5.

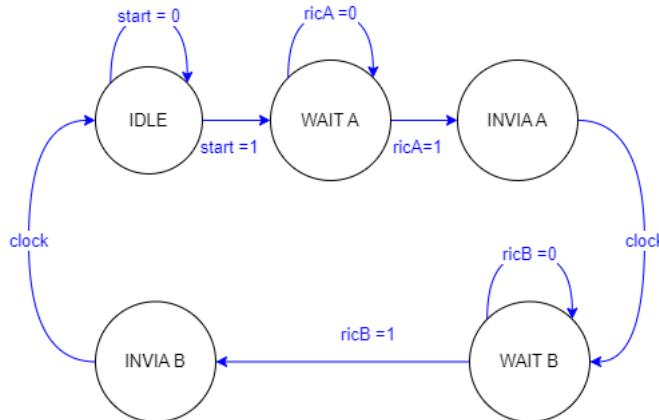


Figura 6.5: Automa di funzionamento del sistema esterno

6.3.3 Unità di controllo

Versione in logica cablata

L'unità di controllo è stata realizzata in due versioni, come richiesto dalla traccia: per la configurazione in logica cablata sono stati utilizzati due tipologie di segnali per la gestione dell'handshaking, chiamati **risA, risB, ricA** e **ricB**, rispettivamente utilizzati per la risposta dei registri A e B e per la richiesta di invio da parte del sistema esterno di A e di B.

L'architettura dell'unità di controllo è presentata in Figura 6.7.

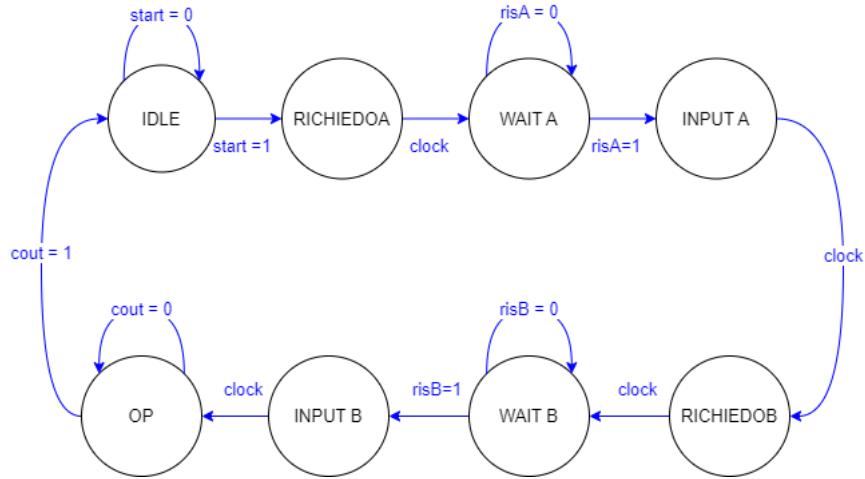


Figura 6.6: Automa dell'unità di controllo

L'unità di controllo è stata codificata tramite l'automa raffigurato in Figura 6.6.

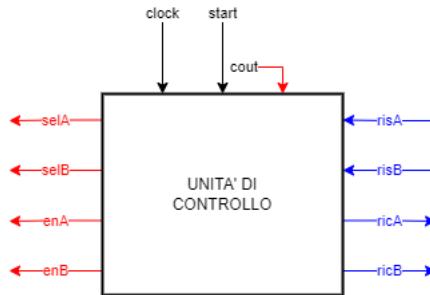


Figura 6.7: Architettura dell'unità di controllo in logica cablata

Versione in logica microprogrammata

L'unità di controllo in logica microprogrammata si compone di due sottosistemi: parte di controllo e micro-ROM. La parte di controllo, tramite un registro chiamato Program Counter, si occupa di selezionare la cella di memoria che contiene la giusta **control word** per la definizione dei segnali di uscita. La memoria serve quindi a conservare le differenti configurazioni di segnali di uscita per tutte le possibili combinazioni dei segnali in ingresso alla logica di controllo. Il program counter quindi conserva l'indirizzo di memoria della control word attiva. Un segnale di PC_next comunica l'indirizzo della prossima microistruzione alla parte di controllo.

Il vantaggio di un'unità di controllo sviluppata in logica microprogrammata è la forte adattabilità al numero di stati della macchina: qualora dovesse entrare in gioco uno stato nuovo per la macchina, basterebbe infatti aggiungere una control word all'interno della micro-ROM, piuttosto che cambiare il funzionamento dell'intera unità di controllo.

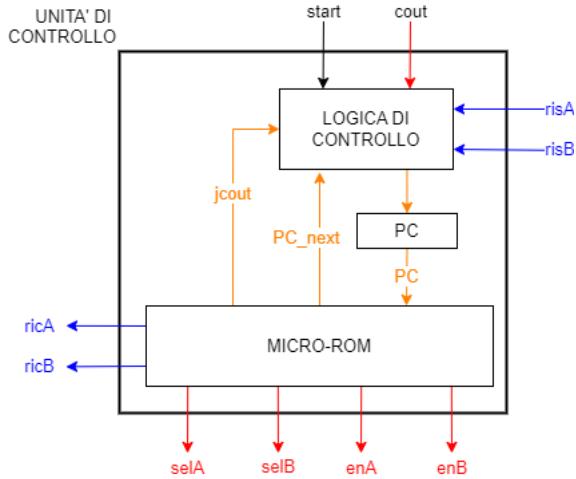


Figura 6.8: Architettura dell'unità di controllo in logica microprogrammata

L'architettura dell'unità di controllo così composta è presentata in Figura 6.8. La micro-ROM è una struttura a blocchi, composta da registri contenenti le combinazioni di bit adatte ad ogni situazione di funzionamento dell'unità di controllo. In particolare, la micro-Rom progettata presenta delle control word con la disposizione di bit indicata in Tabella 6.1.

| PC_next (3 bit) | JCout (1 bit) | enA (1 bit) | enB (1 bit) | selA (1 bit) | selB (1 bit) | ricA (1 bit) | ricB (1 bit) |
|--------------------|------------------|----------------|----------------|-----------------|-----------------|-----------------|-----------------|
| | | | | | | | |

Tabella 6.1: Combinazione di bit di una control word

6.4 Codice

Si riporta di seguito il codice delle componenti che non differiscono dal tipo di logica, quali unità operativa e sistema esterno, mentre successivamente si riportano le differenti implementazioni delle unità di controllo.

Multiplexer 2:1

```

entity mux_2_1 is
  Port ( x0 : in STD_LOGIC_VECTOR (7 downto 0);
         x1 : in STD_LOGIC_VECTOR (7 downto 0);
         load : in STD_LOGIC;
         y : out STD_LOGIC_VECTOR (7 downto 0));
end mux_2_1;

architecture dataflow of mux_2_1 is

begin

```

```

y <= x0 when load = '0' else
    x1 when load = '1';

end dataflow;

```

Registro 8 bit

```

entity registro is
    Port ( d : in STD_LOGIC_VECTOR (7 downto 0);
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in STD_LOGIC;
           o : out STD_LOGIC_VECTOR (7 downto 0));
end registro;

architecture Behavioral of registro is

signal y_conc : std_logic_vector(7 downto 0);

begin
    process(clock, reset)
    begin
        if (reset = '1') then
            y_conc <= (others => '0');
        elsif (clock'event and clock = '1') then
            if(en = '1') then
                y_conc <= d;
            end if;
        end if;
    end process;
    o <= y_conc;
end Behavioral;

```

Full-Adder

Per definire l'uscita del full-adder tramite il modello di programmazione dataflow, si è codificata la tabella di verità di un full-adder a due ingressi. Tale tabella di verità viene riportata in Tabella 6.2.

Il full-adder implementato è dotato di due bit di ingresso, un riporto **c_in** eventuale da conteggiare, un riporto **c_out** in uscita e dal valore **s** di somma tra i due bit.

```

entity fullAdder is
    Port ( A : in STD_LOGIC;
           B : in STD_LOGIC;
           c_in : in STD_LOGIC;
           c_out : out STD_LOGIC;
           s : out STD_LOGIC);

```

| X | Y | C | SUM | R |
|---|---|---|-----|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

Tabella 6.2: Tabella di verità di un Full-Adder

```
end fullAdder;

architecture dataflow of fullAdder is

begin
    s <= A xor B xor c_in;
    c_out <= (A and B) or (c_in and (A xor B));
end dataflow;
```

Sottrattore

Il Sottrattore, chiamato così per coerenza con la sua funzione di somma tra positivo e negativo, è un addizionatore composto da n Full-Adder, dove n è il numero di bit dei numeri di cui esso si occupa di effettuare la somma.

```
entity sottrattore is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           carry : in STD_LOGIC;
           ris : out STD_LOGIC_VECTOR (7 downto 0);
           cout : out STD_LOGIC);
end sottrattore;

architecture structural of sottrattore is

    signal c_i: std_logic_vector(0 to 6) := (others => '0');
    signal B_comp: std_logic_vector(7 downto 0) := (others =>'0');
    component fullAdder is
        Port (
            A: in std_logic;
            B: in std_logic;
            c_in: in std_logic;
            c_out:out std_logic;
            s: out std_logic
```

```

    );
end component;

begin
  B_comp <= not B; -- mi serve il negato per fare il complemento a 2
  --istanziazione del primo full adder
  F_Adder_0: fullAdder
    port map(
      A=>A(0),
      B=>B_comp(0),
      c_in=>carry,
      c_out=>c_i(0),
      s=>ris(0)
    );
  --istanziazione dei 6 full adder interni
  FA: for i in 1 to 6 generate
    F_Adder_I: fullAdder
      port map(
        A => A(i),
        B => B_comp(i),
        c_in => c_i(i-1),
        c_out => c_i(i),
        s => ris(i)
      );
  end generate;
  --istanziazione del full adder finale
  F_Adder_7: fullAdder
    port map(
      A=>A(7),
      B=>B_comp(7),
      c_in=>c_i(6),
      c_out=>cout,
      s=>ris(7)
    );
end structural;

```

Unità operativa

L'unità operativa è composta da 2 mux_2_1, 2 registri da 8 bit e da un Sottrattore. L'architettura è stata definita strutturale.

```

entity unita_op is
  Port ( binA : in STD_LOGIC_VECTOR (7 downto 0);
         binB : in STD_LOGIC_VECTOR (7 downto 0);
         selA : in STD_LOGIC := '0';
         selB : in STD_LOGIC := '0';
         enA : in STD_LOGIC := '0';

```

```

    enB : in STD_LOGIC := '0';
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    Bout : out STD_LOGIC_VECTOR (7 downto 0);
    cout : out STD_LOGIC);
end unita_op;

architecture structural of unita_op is
-- Componenti
component mux_2_1 is
port (
    x1 : in STD_LOGIC_VECTOR (7 downto 0);
    x0 : in STD_LOGIC_VECTOR (7 downto 0);
    load : in STD_LOGIC;
    y : out STD_LOGIC_VECTOR(7 downto 0)
);
end component;

component registro is
port (
    D : in STD_LOGIC_VECTOR(7 downto 0);
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    en : in STD_LOGIC;
    o : out STD_LOGIC_VECTOR(7 downto 0)
);
end component;

component Sottrattore is
port (
    A: in std_logic_vector(7 downto 0);
    B: in std_logic_vector(7 downto 0);
    carry: in std_logic:='1';
    ris: out std_logic_vector(7 downto 0);
    cout: out std_logic
);
end component;

-- Segnali interni
signal out_muxA : std_logic_vector (7 downto 0) := (others=>'0');
signal out_muxB : std_logic_vector (7 downto 0) := (others=>'0');
signal out_regA : std_logic_vector (7 downto 0) := (others=>'0');
signal out_regB : std_logic_vector (7 downto 0) := (others=>'0');
signal out_add : std_logic_vector (7 downto 0) := (others=>'0');
signal carry_s : std_logic := '1';

begin
-- multiplexer di ingresso ai registri

```

```

mux2_1_A : mux_2_1
port map (
    x1 => BinA,
    x0 => out_add,
    load => selA,
    y => out_muxA
);
mux2_1_B : mux_2_1
port map (
    x1 => BinB,
    x0 => (others=>'0'),
    load => selB,
    y => out_muxB
);
-- registri A e B
registro_A : registro
port map (
    D => out_muxA,
    clock => clock,
    reset => reset,
    en => enA,
    o => out_regA
);
registro_B : registro
port map (
    D => out_muxB,
    clock => clock,
    reset => reset,
    en => enB,
    o => out_regB
);
-- Sottrattore
sottrazione : Sottrattore
port map (
    A => out_regA,
    B => out_regB,
    carry => carry_S,
    ris => out_add,
    cout => cout
);
-- uscita concorrente
Bout<=out_add;

end structural;

```

Unità di controllo esterna

L'**unità di controllo esterna** è responsabile del coordinamento con l'unità di controllo interna al sistema tramite handshaking: le due unità di controllo si scambiano i segnali di controllo per l'unità operativa.

```
entity unita_ctrl_ext is
    Port ( ricA : in STD_LOGIC;
            ricB : in STD_LOGIC;
            clock : in STD_LOGIC;
            start : in STD_LOGIC;
            enA : out STD_LOGIC;
            enB : out STD_LOGIC;
            risA : out STD_LOGIC;
            risB : out STD_LOGIC);
end unita_ctrl_ext;

architecture Behavioral of unita_ctrl_ext is

    type state is (idle,waitA,sendA,waitB,sendB);
    signal stato_corrente, stato_prossimo : state;

begin
    reg_state: PROCESS(clock) is
    begin
        if (clock'event and clock='1') then
            if (start='1') then
                stato_corrente<=waitA;
            else
                stato_corrente<=stato_prossimo;
            end if;
        end if;
    end process;

    combinatorio: process(stato_corrente,ricA,ricB) is
    begin
        -- Inizializzazione
        enA<='0';
        enB<='0';
        risA<='0';
        risB<='0';
        stato_prossimo<=idle;
        CASE stato_corrente IS
            WHEN idle =>
                stato_prossimo<=idle;
            WHEN waitA =>
                if (ricA='0') then
                    stato_prossimo<=waitA;
                end if;
            WHEN sendA =>
                if (ricA='1') then
                    stato_prossimo<=sendA;
                end if;
            WHEN waitB =>
                if (ricB='0') then
                    stato_prossimo<=waitB;
                end if;
            WHEN sendB =>
                if (ricB='1') then
                    stato_prossimo<=sendB;
                end if;
        END CASE;
    end process;
end;
```

```

        elsif (ricA='1') then --se mi arriva una richiesta del dato A
            stato_prossimo<=sendA; --procedo ad inviare il dato
        end if;
    WHEN sendA =>
        enA<='1'; --abilito A
        risA<='1'; --e invio il segnale di ack alla UC interna
        stato_prossimo<=waitB; --attendo che mi venga richiesto B
    WHEN waitB =>
        if (ricB='0') then
            stato_prossimo<=waitB;
        elsif (ricB='1') then --se mi arriva una richiesta del dato B
            stato_prossimo<=sendB; --procedo ad inviare il dato
        end if;
    WHEN sendB =>
        enB<='1'; --abilito B
        risB<='1'; --e invio il segnale di ack alla UC interna
        stato_prossimo<=idle;
    end CASE;
end process;

end Behavioral;

```

Sistema esterno

Il sistema esterno è composto dall'unità esterna di controllo per l'handshaking e dai due registri contenenti il dato da inviare al sistema.

```

entity sys_ext is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           clock : in STD_LOGIC;
           start : in STD_LOGIC;
           ricA : in STD_LOGIC;
           ricB : in STD_LOGIC;
           datoA : out STD_LOGIC_VECTOR (7 downto 0);
           datoB : out STD_LOGIC_VECTOR (7 downto 0);
           risA : out STD_LOGIC;
           risB : out STD_LOGIC;
           reset : in STD_LOGIC);
end sys_ext;

architecture structural of sys_ext is
    signal enA : std_logic := '0';
    signal enB : std_logic := '0';

    component registro is
        port (
            D : in STD_LOGIC_VECTOR(7 downto 0);

```

```

    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    en : in STD_LOGIC;
    o : out STD_LOGIC_VECTOR(7 downto 0)
);
end component;

component unita_ctrl_ext is
port (
    ricA : in STD_LOGIC;
    ricB : in STD_LOGIC;
    clock : in STD_LOGIC;
    start : in STD_LOGIC;
    enA : out STD_LOGIC;
    enB : out STD_LOGIC;
    risA : out STD_LOGIC;
    risB : out STD_LOGIC
);
end component;

begin
    registro_A : registro
    port map (
        D=>A,
        clock=>clock,
        reset=>reset,
        en=>enA,
        o=>datoA
    );
    registro_B : registro
    port map (
        D=>B,
        clock=>clock,
        reset=>reset,
        en=>enB,
        o=>datoB
    );
    unita_controllo : unita_ctrl_ext
    port map (
        ricA=> ricA,
        ricB=> ricB,
        clock=> clock,
        start=> start,
        enA => enA,
        enB=> enB,
        risA=> risA,
        risB=> risB
    );

```

```
end structural;
```

6.4.1 Unità di controllo in Logica cablata

L'unità di controllo in logica cablata è stata sviluppata in un unico process, facendo fede all'automa di funzionamento descritto nei paragrafi precedenti.

```
entity unita_ctrl is
    Port ( cout : in STD_LOGIC;
            start : in STD_LOGIC;
            risA : in STD_LOGIC;
            risB : in STD_LOGIC;
            clock : in STD_LOGIC;
            selA : out STD_LOGIC;
            selB : out STD_LOGIC;
            enA : out STD_LOGIC;
            enB : out STD_LOGIC;
            ricA : out STD_LOGIC;
            ricB : out STD_LOGIC);
end unita_ctrl;

architecture Behavioral of unita_ctrl is
    type state is (
        idle,
        ric_A,
        waitA,
        inA,
        ric_B,
        waitB,
        inB,
        op
    );
    signal stato_corrente : state;
    signal stato_prossimo : state;

begin
    reg_state: process(clock)
    begin
        if (clock'event and clock='1') then
            if (start='1') then
                stato_corrente<=ric_A;
            else
                stato_corrente<=stato_prossimo;
            end if;
        end if;
    end process;
end Behavioral;
```

```

end process reg_state;

combinatorio : process(stato_corrente, cout, risA, risB)
begin
    -- Inizializzazione
    selA<='0';
    selB<='0';
    enA<='0';
    enB<='0';
    ricA<='0';
    ricB<='0';
    stato_prossimo<=idle;
    -- Codifica dell'automa
    case stato_corrente is
        when idle =>
            stato_prossimo<=idle; --sono in IDLE finchè non ho lo start
        when ric_A =>
            ricA<='1'; -- e' stato richiesto il dato A
            stato_prossimo<=waitA;
        when waitA =>
            if (risA='0') then --se non ho ricevuto l'ack di risposta di A
                stato_prossimo<=waitA; --attendo ancora
            elsif (risA='1') then
                stato_prossimo<=inA; -- altrimenti passo a comunicare con l'UO
            end if;
        when inA =>
            selA<='1'; --alzo il bit di selezione di A
            enA<='1'; -- e abilito il registro A all'interno dell'UO
            stato_prossimo<=ric_B; --e passo a richiedere il dato B
        when ric_B =>
            ricB<='1'; -- e' stato richiesto il dato B
            stato_prossimo<=waitB;
        when waitB =>
            if (risB='0') then --se non ho ricevuto l'ack di risposta di B
                stato_prossimo<=waitB; --attendo ancora
            elsif (risB='1') then
                stato_prossimo<=inB; -- altrimenti passo a comunicare con l'UO
            end if;
        when inB =>
            selB<='1'; --alzo il bit di selezione di B
            enB<='1'; -- e abilito il registro B all'interno dell'UO
            stato_prossimo<=op; -- e passo a far partire le operazioni
        when op =>
            if (cout='0') then --se l'UO non ha ancora terminato
                enA<='1'; -- prendi ancora il dato in A
                stato_prossimo<=op; --ed esegui nuovamente le operazioni
            elsif (cout='1') then -- se invece ho terminato

```

```

        enB<='1'; --prendi il dato in B
        stato_prossimo<=idle; -- e termina
    end if;
end case;
end process;

end Behavioral;
```

6.4.2 Unità di controllo in Logica microprogrammata

Micro-ROM

La micro-ROM si compone di costanti di tipo record, definite a priori come combinazione di bit di stato.

```

entity microRom is
  port(
    PC: in unsigned(2 downto 0);
    PC_next: out unsigned (2 downto 0);
    JCount: out std_logic;
    enA: out std_logic;
    selA: out std_logic;
    enB: out std_logic;
    selB: out std_logic;
    ricA: out std_logic;
    ricB: out std_logic
  );
end microRom;

architecture synth of microRom is
  type controllo_type is record
    PC_next: unsigned (2 downto 0);
    JCount: std_logic;
    enA: std_logic;
    selA: std_logic;
    enB: std_logic;
    selB: std_logic;
    ricA: std_logic;
    ricB: std_logic;
  end record;

  constant idle : controllo_type :=(
    PC_next=>"000",
    JCount=>'0',
    enA=>'0',
    selA=>'0',
    enB=>'0',
    selB=>'0',
    ricA=>'0',
```

```

    ricB=>'0'
);

constant richiestaA: controllo_type:=(

    PC_next=>"010",
    JCount=>'0',
    enA=>'0',
    selA=>'0',
    enB=>'0',
    selB=>'0',
    ricA=>'1',
    ricB=>'0'

);

constant waitA: controllo_type:=(

    PC_next=>"010",
    JCount=>'0',
    enA=>'0',
    selA=>'0',
    enB=>'0',
    selB=>'0',
    ricA=>'0',
    ricB=>'0'

);

constant inA: controllo_type:=(

    PC_next=>"100",
    JCount=>'0',
    enA=>'1',
    selA=>'1',
    enB=>'0',
    selB=>'0',
    ricA=>'0',
    ricB=>'0'

);

constant richiestaB: controllo_type:=(

    PC_next=>"101",
    JCount=>'0',
    enA=>'0',
    selA=>'0',
    enB=>'0',
    selB=>'0',
    ricA=>'0',
    ricB=>'1'

);

constant waitB: controllo_type:=(

    PC_next=>"101",

```

```

JCount=>'0',
enA=>'0',
selA=>'0',
enB=>'0',
selB=>'0',
ricA=>'0',
ricB=>'0'
);

constant inB: controllo_type:=(

PC_next=>"111",
JCount=>'0',
enA=>'0',
selA=>'0',
enB=>'1',
selB=>'1',
ricA=>'0',
ricB=>'0'
);

constant op : controllo_type:=(

PC_next=>"111",
JCount=>'1',
enA=>'1',
selA=>'0',
enB=>'1',
selB=>'0',
ricA=>'0',
ricB=>'0'
);

type ROM_TYPE is array (0 to 7) of controllo_type;
constant ROM: ROM_type:=(

0=>idle,
1=>richiestaA,
2=>waitA,
3=>inA,
4=>richiestaB,
5=>waitB,
6=>inB,
7=>op
);

signal controllo : controllo_type;

begin
controllo <= ROM(to_integer(PC));
PC_next<= controllo.PC_next;
JCount<=controllo.JCount;
enA<= controllo.enA;
selA<= controllo.selA;

```

```

    enB<= controllo.enB;
    selB<= controllo.selB;
    ricA<= controllo.ricA;
    ricB<= controllo.ricB;
end synth;
```

Unità di controllo

L'unità di controllo microprogrammata si compone di una micro-rom, e di una struttura di controllo che gestisce i segnali.

```

entity unita_ctrl_microprog is
  port(
    clock: in std_logic;
    start: in std_logic;
    cout: in std_logic;
    selA: out std_logic;
    enA: out std_logic;
    enB: out std_logic;
    selB: out std_logic;
    risA: in std_logic;
    risB: in std_logic;
    ricA: out std_logic;
    ricB: out std_logic
  );
end unita_ctrl_microprog;

architecture microprogramming of unita_ctrl_microprog is

  component microRom is
    port(
      PC: in unsigned(2 downto 0);
      PC_next: out unsigned (2 downto 0);
      JCount: out std_logic;
      enA: out std_logic;
      selA: out std_logic;
      enB: out std_logic;
      selB: out std_logic;
      ricA: out std_logic;
      ricB: out std_logic
    );
  end component;

  signal PC_next,PC : unsigned(2 downto 0);
  signal enA_temp,enB_temp,JCount: std_logic;

begin
  rom: MicroRom
```

```

port map(
    PC=>PC,
    PC_next=>PC_next,
    JCount=>JCount,
    enA=>enA_temp,
    selA=>selA,
    enB=>enB_temp,
    selB=>selB,
    ricA=>ricA,
    ricB=>ricB
);

reg_PC: PROCESS(clock)
begin
    if(clock'event and clock='1') then
        if(start='1') then
            PC<="001";
        elsif(risA='1')then
            PC<="011";
        elsif(risB='1')then
            PC<="110";
        elsif(JCount='0')then
            PC<=PC_next;
        else
            if(cout='0')then
                PC<="000";
            end if;
        end if;
    end if;
end process;

enA<=enA_temp when JCount='0' else cout;
enB<=enB_temp when JCount='0' else not(cout);

end microprogramming;

```

6.5 Simulazione

Per simulare le due versioni dell’architettura si è scelto di definire un testbench ad-hoc, che faccia uso di due differenti process: uno per la generazione del segnale di clock e uno per il set degli ingressi. Una particolarità è quella che nel caso della simulazione della logica cablata, sono stati definiti due numeri più piccoli per fare sì che nell’immagine si siano potuti inserire gli stati dell’unità esterna e di controllo in modo da verificare il corretto funzionamento dell’handshaking tra le due entità. In particolare, in Figura 6.9 in rosso sono riportati gli stati dell’unità di controllo e in azzurro quelli

dell'unità esterna. In Figura 6.10 la simulazione dell'architettura in logica microprogrammata, con i numeri 125 e 12.

```

entity top_module_tb is
end top_module_tb;

architecture Behavioral of top_module_tb is

component top_module is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           AmodeB : out STD_LOGIC_VECTOR (7 downto 0);
           start : in STD_LOGIC);
end component;

signal A : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
signal B : STD_LOGIC_VECTOR (7 downto 0) := "00000000";
signal clock : STD_LOGIC;
signal reset : STD_LOGIC;
signal AmodeB : STD_LOGIC_VECTOR (7 downto 0);
signal start : STD_LOGIC := '0';
constant clock_period : time := 10 ns;

begin
    uut : top_module
        port map(
            A => A,
            B=> B,
            clock => clock,
            reset => reset,
            AmodeB => AmodeB,
            start => start
        );

    clock_process :process
    begin
        clock <= '1';
        wait for clock_period/2;
        clock <= '0';
        wait for clock_period/2;
    end process;

    stim_proc: process
    begin
        reset <= '1';
        wait for 100 ns;

```

```

    reset <= '0';
    A<="01111101"; --125
    --A<="00100111" -- 39 (logica cablata)
    B<="000001100"; --12
    wait for clock_period;
    start<='1';
    wait for clock_period;
    start<='0';
    wait;
end process;

end Behavioral;

```

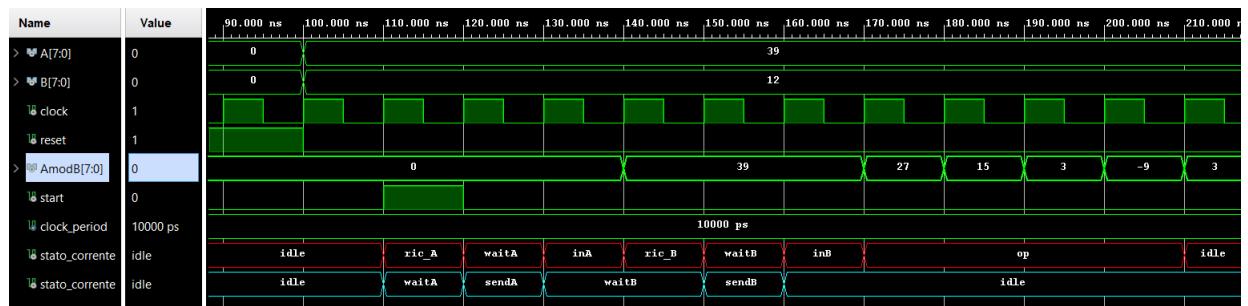


Figura 6.9: Simulazione della macchina AmodB in logica cablata

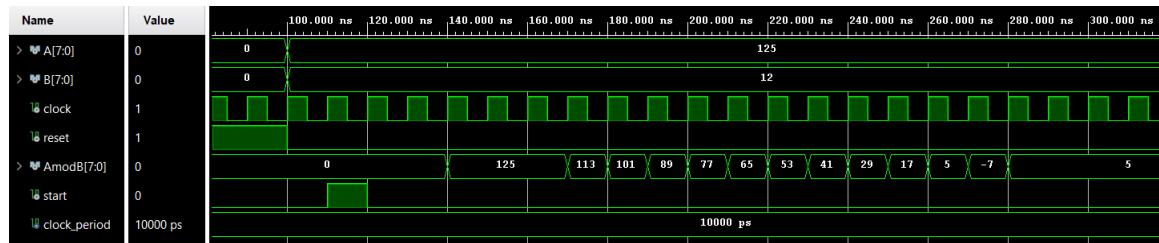


Figura 6.10: Simulazione della macchina AmodB in logica microprogrammata.

Capitolo 7

Esercizio 6

7.1 Traccia

Progettare un sistema per inviare due parole da 16 bit ciascuna da una unità A ad una unità B. Le due unità non sono dotate di un collegamento diretto costituito da un bus di 16 bit, ma l'unità A possiede un bus di 4 bit in uscita e l'unità B possiede un bus di 8 bit in ingresso. Per questa ragione, il trasferimento deve avvenire in più passi facendo uso di un buffer. A questo scopo, una unità di controllo si occupa di trasferire ognuna delle due parole di 16 bit da A al buffer in 4 blocchi successivi da 4 bit ciascuno, e successivamente dal buffer a B in 2 blocchi successivi da 8 bit ciascuno.

7.2 Soluzione

Per la risoluzione dell'esercizio sono stati implementati 3 sistemi, le due unità richieste dalla traccia e un unità di controllo centrale che contiene il buffer per il trasferimento dei dati.

Le due unità comunicano con il sistema di controllo centrale per la trasmissione dei bit e la successiva ricezione, sfruttando due protocolli di handshaking, uno a monte e uno a valle dell'unità di controllo centrale. Il sistema A attende che il buffer sia pronto a ricevere i dati in ingresso, e procede a riempire il buffer. Il sistema B provvede a prelevare i dati finché il buffer è pieno, e attende che il buffer sia nuovamente pieno per procedere al prelievo successivo. L'unità di controllo centrale provvede ad informare i due sistemi circa lo stato del buffer tramite appositi segnali di riempimento e prelievo all'interno del registro buffer.

7.3 Schematici

Si è scelto di procedere ad analizzare in primis l'architettura generale per poi riportare alcuni diagrammi di dettaglio dei sottosistemi e della loro interazione.

In Figura 7.1 si nota l'interazione tra i due sistemi A e B mediante l'unità di controllo:



Figura 7.1: Architettura generica del sistema di comunicazione

- Il sistema A comunica i dati al sistema centrale tramite un bus di 4 bit. Comunica al sistema centrale che i bit all'interno del bus sono pronti tramite il segnale **A_ready**, e attende la conferma di ricezione dal sistema centrale tramite il segnale **A_received**.
- Il sistema centrale attende l'invio dei dati da parte del sistema A, ne notifica la ricezione tramite il segnale **A_received**, e comunica al sistema B l'abilitazione al prelievo tramite il segnale **B_ready**.
- Il sistema B preleva i dati dal sistema centrale tramite un bus di 8 bit. Attende l'invio dei bit da parte del sistema centrale tramite il segnale **B_ready**, e comunica al sistema centrale l'esito del prelievo tramite il segnale **B_received**.

Si mostra quindi, in Figura 7.2 lo schema interno dei sistemi.

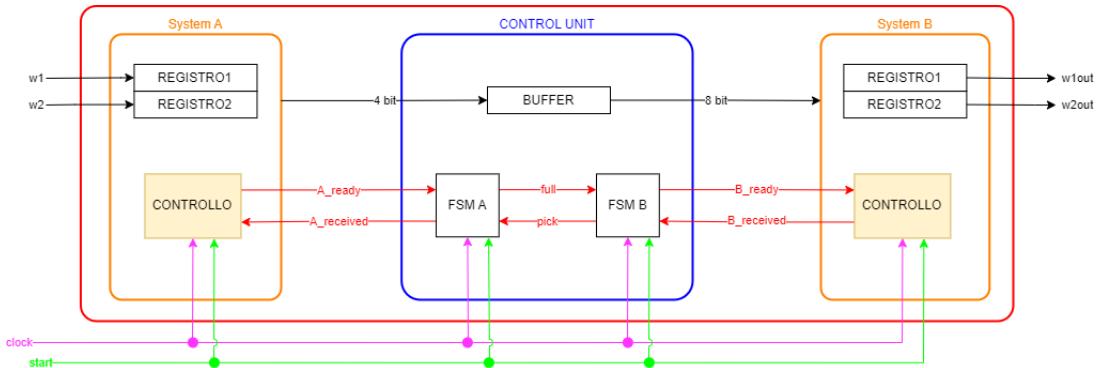


Figura 7.2: Architettura di dettaglio del sistema di comunicazione

Sia il sistema A, che il sistema B, che anche l'unità di controllo centrale, sfruttano dei registri a 16 bit per la gestione dei dati. La differenza è che, ovviamente, l'unità centrale presenta un solo registro buffer, mentre i sistemi esterni conservano le word trasmesse in due differenti registri. La comunicazione tramite protocollo di handshaking viene gestita tramite due sotto-unità all'interno dell'unità di controllo centrale: l'unità **FSM-A** si occupa della gestione dell'handshaking con il sistema A, mentre l'unità **FSM-B** di quello con il sistema B. Le due sotto-unità comunicano tra loro tramite i segnali **full** e **picked**, rispettivamente utilizzati per il riempimento del buffer da parte di A, e di prelievo del buffer da parte di B.

7.3.1 Automi

Per rendere chiaro il processo delle due FSM e del loro dialogo con i sistemi A e B sono stati prodotti degli automi a stati finiti per descrivere il funzionamento di tutte le unità in gioco all'interno del sistema di comunicazione.

Sistema A

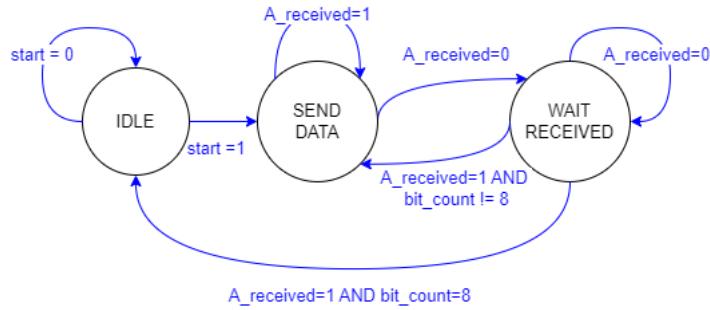


Figura 7.3: Diagramma degli stati del sistema A

- Il sistema A resta in `idle` finchè non riceve il segnale di `start`.
- All'atto della ricezione di `start`, il sistema A comunica i bit tramite il bus al buffer centrale, finchè il sistema centrale comunica di aver ricevuto tali bit.
- Il sistema A resta in attesa del segnale di ricezione da parte dell'unità di controllo centrale, prima di provvedere ad inviare nuovamente i bit sul bus.

Sistema B

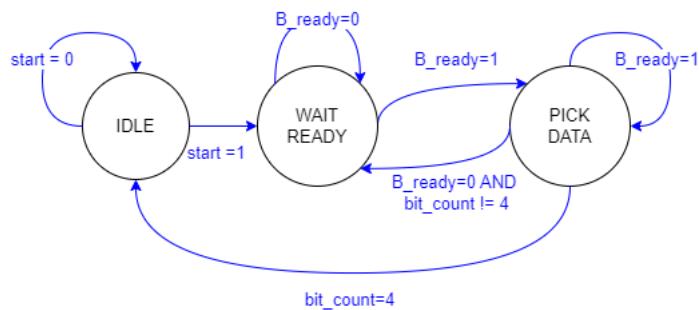


Figura 7.4: Diagramma degli stati del sistema B

- Il sistema B resta in `idle` finchè non riceve il segnale di `start`.
- All'atto della ricezione di `start`, il sistema B resta in attesa del segnale di `ready` da parte dell'unità di controllo centrale, prima di provvedere a prelevare i bit dal buffer tramite il bus.

- Il sistema A preleva i bit tramite il bus dal buffer centrale, e ne comunica la ricezione al sistema centrale.

FSM-A

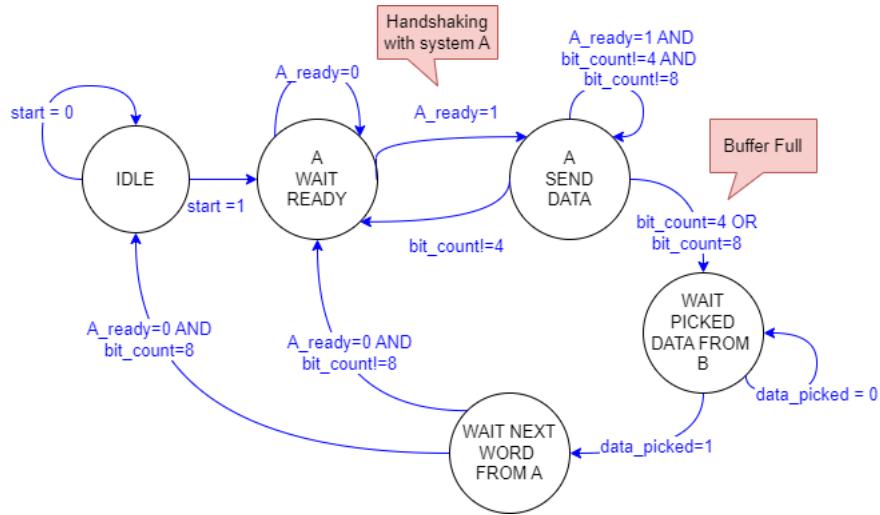


Figura 7.5: Diagramma degli stati dell'unità FSM-A

- L'unità FSM-A resta in `idle` finché non riceve il segnale di `start`.
- All'atto della ricezione dello `start`, l'unità FSM-A attende che i dati all'interno del buffer siano pronti, e comunica tramite handshaking con il sistema A affinché esso possa riempire il buffer.
- All'atto della commutazione di un contatore, `bit_count`, che conta il numero di gruppi di bit inseriti da A all'interno del buffer, se quest'ultimo eccede la dimensione del buffer l'unità attende che il buffer venga svuotato.
- Se il buffer è stato svuotato, è possibile nuovamente per il sistema A procedere ad un nuovo caricamento, a patto che il conteggio di bit non sia già arrivato al numero massimo di gruppi di bit trasmissibili da A. In tal caso l'unità passa nuovamente nello stato di `idle`.

FSM-B

- L'unità FSM-B resta in `idle` finché non riceve il segnale di `start`.
- All'atto della ricezione dello `start`, l'unità FSM-B attende che il buffer sia pieno, e procede a notificare il sistema B di tale condizione afinché esso possa prelevare i dati.
- All'atto della commutazione di un contatore, `bit_count`, che conta il numero di gruppi di bit prelevati dal buffer dal sistema B, se quest'ultimo eccede il valore di 2, ci si trova nella condizione di buffer vuoto, ed è necessario attendere una nuova trasmissione.

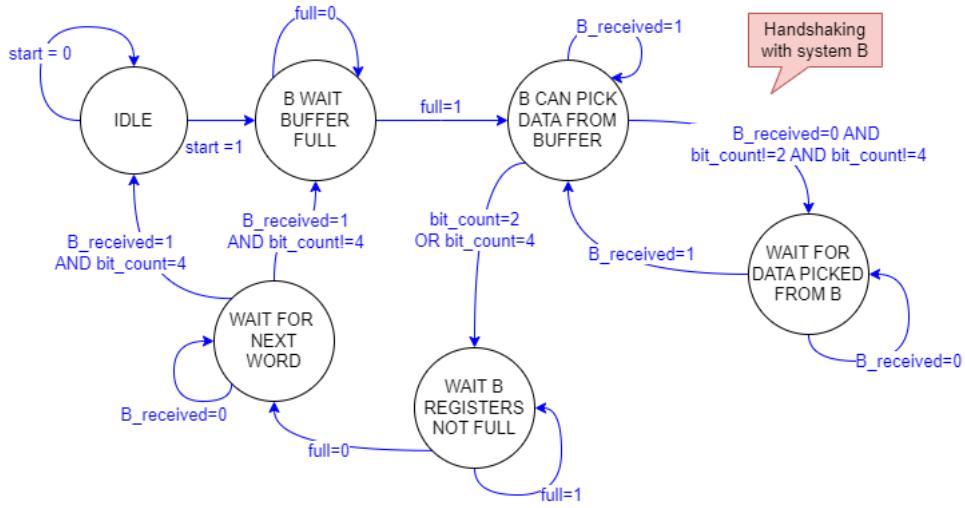


Figura 7.6: Diagramma degli stati dell'unità FSM-B

- Se il buffer è stato nuovamente riempito, è possibile per il sistema B procedere con un nuovo prelievo, a patto che il conteggio di bit non sia già arrivato al numero massimo di gruppi di bit che il sistema B può contenere. In tal caso l'unità passa nuovamente nello stato di `idle`.

7.4 Codice

Registro

```

entity registro is
  Port (
    D : in STD_LOGIC_VECTOR(15 downto 0);
    clk : in STD_LOGIC;
    rst : in STD_LOGIC;
    en : in STD_LOGIC;
    o: out STD_LOGIC_VECTOR(15 downto 0));
end registro;
architecture Behavioral of registro is
  signal y_temp: std_logic_vector(15 downto 0) := (others=>'0');
begin
  begin
    o<=y_temp;
    output: process(clk,rst)
    begin
      if (rst = '1') then
        y_temp<=(others=>'0');
      elsif(clk'event AND clk='1') then
        if (en='1') then
          y_temp<=D;
        end if;
      end if;
    end process;
  end;
end;

```

```

        end process;
end Behavioral;

SystemA

entity SystemA is
  Port (
    clock : in STD_LOGIC;
    start : in STD_LOGIC;
    reset : in STD_LOGIC;
    A_received : in STD_LOGIC;
    A_ready : out STD_LOGIC;
    w1 : in STD_LOGIC_VECTOR (15 downto 0);
    w2 : in STD_LOGIC_VECTOR (15 downto 0);
    o : out STD_LOGIC_VECTOR (3 downto 0)
  );
end SystemA;

architecture structural of SystemA is
  -- Stati del sistema A
  type state is (idle, send_data, wait_received);
  signal stato_corrente : state;
  -- Segnali interni
  signal out_temp1, out_temp2 : std_logic_vector(15 downto 0);
  signal out_data_temp : std_logic_vector(3 downto 0) := (others=>'0');
  -- Componenti
  component registro is
    port (
      D : in STD_LOGIC_VECTOR(15 downto 0);
      clk : in STD_LOGIC;
      rst : in STD_LOGIC;
      en : in STD_LOGIC;
      o : out STD_LOGIC_VECTOR(15 downto 0)
    );
  end component;
begin
  begin
    o<=out_data_temp;

    registro_1 : registro
      port map (
        D=>w1,
        clk=>clock,
        rst=>reset,
        en=>'1',
        o=>out_temp1
      );
  end;

```

```

registro_2 : registro
port map (
    D=>w2,
    clk=>clock,
    rst=>reset,
    en=>'1',
    o=>out_temp2
);
UC_A : process(clock,start,reset)
variable bit_count : integer range 0 to 8 := 1;
begin
    if (reset='1') then
        bit_count:=1;
        stato_corrente<=idle;
    elsif (start='1') then
        stato_corrente<=send_data;
    elsif (clock'event and clock='1') then
        case stato_corrente is
            when idle =>
                stato_corrente<=idle;
                A_ready<='0';
            when send_data =>
                A_ready<='0';
                if (A_received='0')then
                    case bit_count is
                        when 0 => NULL;
                        when 1 => out_data_temp<=out_temp1(3 downto 0);
                        when 2 => out_data_temp<=out_temp1(7 downto 4);
                        when 3 => out_data_temp<=out_temp1(11 downto 8);
                        when 4 => out_data_temp<=out_temp1(15 downto 12);
                        when 5 => out_data_temp<=out_temp2(3 downto 0);
                        when 6 => out_data_temp<=out_temp2(7 downto 4);
                        when 7 => out_data_temp<=out_temp2(11 downto 8);
                        when 8 => out_data_temp<=out_temp2(15 downto 12);
                    end case;
                    stato_corrente<=wait_received;
                    A_ready<='1';
                else
                    stato_corrente<=send_data;
                end if;
            when wait_received =>
                if (A_received='1' AND bit_count/=8) then
                    stato_corrente<=send_data;
                    bit_count:=bit_count+1;
                elsif (A_received='1' AND bit_count=8) then
                    stato_corrente<=idle;
                else

```

```

        stat0_corrente<=wait_received;
    end if;
end case;
end if;
end process ;
end structural;

```

SystemB

```

entity SystemB is
    Port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        start : in STD_LOGIC;
        input : in STD_LOGIC_VECTOR(7 downto 0);
        B_ready : in STD_LOGIC;
        B_received : out STD_LOGIC;
        w1 : out STD_LOGIC_VECTOR(15 downto 0);
        w2 : out STD_LOGIC_VECTOR(15 downto 0) );
end SystemB;

architecture structural of SystemB is
    -- Stati del sistema B
    type state is (idle, wait_ready, pick_data);
    signal stat0_corrente : state := idle;
    -- Segnali interni
    signal y_temp : std_logic_vector(15 downto 0) := (others=>'0');
    signal input1, input2 : std_logic_vector(15 downto 0) := (others=>'0');
    signal input_temp1, input_temp2 :std_logic_vector(15 downto 0) := (others=>'0');
    -- Componenti
    component registro is
        port (
            D : in STD_LOGIC_VECTOR(15 downto 0);
            clk : in STD_LOGIC;
            rst : in STD_LOGIC;
            en : in STD_LOGIC;
            o : out STD_LOGIC_VECTOR(15 downto 0)
        );
    end component;
begin
    input1<=input_temp1;
    input2<=input_temp2;
    registro_1 : registro
        port map (
            D=>input1,
            clk=>clock,
            rst=>reset,

```

```

        en=>'1',
        o=>w1
    );
registro_2 : registro
port map (
    D=>input2,
    clk=>clock,
    rst=>reset,
    en=>'1',
    o=>w2
);
UC_B : process(clock, start, reset)
variable bit_count : integer range 0 to 4 := 0;
begin
    B_received<='0';
    if (reset='1') then
        bit_count:=1;
        stato_corrente<=idle;
    elsif (start='1') then
        stato_corrente<=wait_ready;
    elsif (clock'event and clock='1') then
        case stato_corrente is
            when idle =>
                stato_corrente<=idle;
            when wait_ready =>
                if (B_ready='0') then
                    stato_corrente<=wait_ready;
                elsif (B_ready='1') then
                    stato_corrente<=pick_data;
                    bit_count:=bit_count+1;
                end if;
            when pick_data =>
                case bit_count is
                    when 0 => NULL;
                    when 1 => input_temp1(7 downto 0)<=input;
                    when 2 => input_temp1 (15 downto 8)<=input;
                    when 3 => input_temp2(7 downto 0)<=input;
                    when 4 => input_temp2(15 downto 8)<=input;
                end case;
                B_received<='1';
                if (bit_count=4) then
                    stato_corrente<=idle;
                    B_received<='0';
                elsif (B_ready='0') then
                    stato_corrente<=wait_ready;
                else
                    stato_corrente<=pick_data;
                end if;
            end if;
        end if;
    end process;
end;

```

```

        end if;
    end case;
end if;
end process;
end Structural;
```

ControlUnit

```

entity ControlUnit is
  Port (
    clock : in STD_LOGIC;
    reset : in STD_LOGIC;
    start : in STD_LOGIC;
    inputA : in STD_LOGIC_VECTOR(3 downto 0);
    outputB : out STD_LOGIC_VECTOR(7 downto 0);
    A_ready : in STD_LOGIC;
    A_received : out STD_LOGIC;
    B_ready : out STD_LOGIC;
    B_received : in STD_LOGIC);
end ControlUnit;

architecture structural of ControlUnit is
  -- Stati dell'unità FSM - A
  type statoA is (idle, A_wait_ready, A_send_data, A_wait_picked_data_from_B, wait_next_word_from_A);
  signal stato_correnteA : statoA := idle;
  -- Stati dell'unità FSM - B
  type statoB is (idle, B_wait_buffer_full, B_can_pick_data_from_buffer, wait_B_registers_not_full,
                  wait_for_data_picked_from_B, wait_for_next_word);
  signal stato_correnteB : statoB := idle;
  -- Segnali interni
  signal full,picked : std_logic := '0';
  signal in_buffer : std_logic_vector(15 downto 0) := (others=>'0');
  signal out_buffer : std_logic_vector(15 downto 0) := (others=>'0');
  signal out_temp : std_logic_vector(7 downto 0) := (others=>'0');
  -- Componenti
  component registro is
    port (
      D : in STD_LOGIC_VECTOR(15 downto 0);
      clk : in STD_LOGIC;
      rst : in STD_LOGIC;
      en: in STD_LOGIC;
      o : out STD_LOGIC_VECTOR(15 downto 0)
    );
  end component;
begin
  outputB<=out_temp;
  buffer_C : registro
```

```

port map (
    D=>in_buffer,
    clk=>clock,
    rst=>reset,
    en=>'1',
    o=>out_buffer
);
fsm_A : process (clock, reset, start)
variable bit_count : integer range 0 to 8 := 0;
begin
    if (reset='1') then
        bit_count:=1;
        stato_correnteA<=idle;
    elsif (start='1') then
        stato_correnteA<=A_wait_ready;
    elsif (clock'event and clock='0') then
        full<='0';
        case stato_correnteA is
            when idle =>
                stato_correnteA<=idle;
                A_received<='0';
            when A_wait_ready =>
                A_received<='0';
                if (A_ready='1') then
                    bit_count:=bit_count+1;
                    stato_correnteA<=A_send_data;
                else
                    stato_correnteA<=A_wait_ready;
                end if;
            when A_send_data =>
                case bit_count is
                    when 0 => NULL;
                    -- primo messaggio
                    when 1 => in_buffer(3 downto 0)<=inputA;
                    when 2 => in_buffer(7 downto 4)<=inputA;
                    when 3 => in_buffer(11 downto 8)<=inputA;
                    when 4 => in_buffer(15 downto 12)<=inputA;
                    -- secondo messaggio
                    when 5 => in_buffer(3 downto 0)<=inputA;
                    when 6 => in_buffer(7 downto 4)<=inputA;
                    when 7 => in_buffer(11 downto 8)<=inputA;
                    when 8 => in_buffer(15 downto 12)<=inputA;
                end case;
                if (bit_count=4 or bit_count=8) then
                    full<='1';
                    A_received<='0';
                    stato_correnteA<=A_wait_picked_data_from_B;
                end if;
            end if;
        end if;
    end if;
end process;

```

```

        else
            A_received<='1';
            if (A_ready='0') then
                stato_correnteA<=A_wait_ready;
            else
                stato_correnteA<=A_send_data;
            end if;
        end if;
    when A_wait_picked_data_from_B =>
        full<='1';
        if (picked='0') then
            stato_correnteA<=A_wait_picked_data_from_B;
        elsif (picked='1') then
            stato_correnteA<=wait_next_word_from_A;
            full<='0';
        end if;
    when wait_next_word_from_A =>
        full<='0';
        A_received<='1';
        if (A_ready='0') then
            if(bit_count=8) then
                stato_correnteA <= idle;
                A_received <= '0';
            else
                stato_correnteA <= A_wait_ready;
                A_received <= '0';
            end if;
        else
            stato_correnteA <= wait_next_word_from_A;
        end if;
    end case;
end if;
end process;

fsm_B : process (clock, reset, start)
variable bit_count : integer range 0 to 4 := 0;
begin
    if (reset='1') then
        bit_count:=1;
        stato_correnteB<=idle;
    elsif (start='1') then
        stato_correnteB<=B_wait_buffer_full;
    elsif (clock'event and clock='0') then
        case stato_correnteB is
            when idle =>
                stato_correnteB<=idle;
                B_ready<='0';

```

```

when B_wait_buffer_full =>
    B_ready<='0';
    if (full='1') then
        stato_correnteB=B_can_pick_data_from_buffer;
    else
        stato_correnteB=B_wait_buffer_full;
    end if;
when B_can_pick_data_from_buffer =>
    B_ready<='0';
    if (B_received='0') then
        bit_count:=bit_count+1;
        case bit_count is
            when 0 => NULL;
            -- primo messaggio
            when 1 => out_temp<=out_buffer(7 downto 0);
            when 2 => out_temp<=out_buffer(15 downto 8);
            -- secondo messaggio
            when 3 => out_temp<=out_buffer(7 downto 0);
            when 4 => out_temp<=out_buffer(15 downto 8);
        end case;
        if (bit_count=2 or bit_count=4) then
            stato_correnteB=wait_B_registers_not_full;
            picked<='1';
        else
            stato_correnteB=wait_for_data_picked_from_B;
            B_ready<='1';
        end if;
    else
        stato_correnteB=B_can_pick_data_from_buffer;
    end if;
when wait_for_data_picked_from_B =>
    if (B_received='1') then
        stato_correnteB=B_can_pick_data_from_buffer;
    else
        stato_correnteB=wait_for_data_picked_from_B;
    end if;
when wait_B_registers_not_full =>
    picked<='1';
    if (full='0') then
        stato_correnteB=wait_for_next_word;
        picked<='0';
        B_ready<='1';
    else
        stato_correnteB=wait_B_registers_not_full;
    end if;
when wait_for_next_word =>
    B_ready<='1';

```

```

        if (B_received='0') then
            stato_correnteB<=wait_for_next_word;
        else
            if (bit_count=4) then
                stato_correnteB<=idle;
                B_ready<='0';
                picked<='0';
            else
                stato_correnteB<=B_wait_buffer_full;
                B_ready<='0';
                picked<='0';
            end if;
        end if;
    end case;
end if;
end process;

end structural;

```

TopModule

```

entity TopModule is
    Port (
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        start : in STD_LOGIC;
        input1 : in STD_LOGIC_VECTOR(15 downto 0);
        input2 : in STD_LOGIC_VECTOR(15 downto 0);
        output1 : out STD_LOGIC_VECTOR(15 downto 0);
        output2 : out STD_LOGIC_VECTOR(15 downto 0));
end TopModule;

architecture structural of TopModule is
    -- Componenti
    component SystemA is
        port(
            clock : in STD_LOGIC;
            start : in STD_LOGIC;
            reset : in STD_LOGIC;
            A_received : in STD_LOGIC;
            A_ready : out STD_LOGIC;
            w1 : in STD_LOGIC_VECTOR (15 downto 0);
            w2 : in STD_LOGIC_VECTOR (15 downto 0);
            o : out STD_LOGIC_VECTOR (3 downto 0)
        );
    end component;

```

```

component SystemB is
    port(
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        start : in STD_LOGIC;
        input : in STD_LOGIC_VECTOR(7 downto 0);
        B_ready : in STD_LOGIC;
        B_received : out STD_LOGIC;
        w1 : out STD_LOGIC_VECTOR(15 downto 0);
        w2 : out STD_LOGIC_VECTOR(15 downto 0)
    );
end component;
component ControlUnit is
    port(
        clock : in STD_LOGIC;
        reset : in STD_LOGIC;
        start : in STD_LOGIC;
        inputA : in STD_LOGIC_VECTOR(3 downto 0);
        outputB : out STD_LOGIC_VECTOR(7 downto 0);
        A_ready : in STD_LOGIC;
        A_received : out STD_LOGIC;
        B_ready : out STD_LOGIC;
        B_received : in STD_LOGIC
    );
end component;
-- Segnali interni
signal A_ready,A_received : std_logic := '0';
signal B_ready,B_received: std_logic := '0';
signal outputA : std_logic_vector(3 downto 0):= (others=>'0');
signal inputB : std_logic_vector(7 downto 0):= (others=>'0');

begin
    UC_sysA : SystemA
        port map (
            clock => clock,
            start => start,
            reset => reset,
            A_received => A_received,
            A_ready => A_ready,
            w1 => input1,
            w2 => input2,
            o => outputA
        );
    UC_sysB : SystemB
        port map (
            clock => clock,

```

```

        reset => reset,
        start => start,
        input => inputB,
        B_ready => B_ready,
        B_received => B_received,
        w1 => output1,
        w2 => output2
    );
UC_ctrl_unit : ControlUnit
port map (
    clock => clock,
    reset => reset,
    start => start,
    inputA => outputA,
    outputB => inputB,
    A_ready => A_ready,
    A_received => A_received,
    B_ready => B_ready,
    B_received => B_received
);

```

end structural;

7.5 Simulazione

Al fine di verificare il corretto funzionamento del meccanismo di comunicazione, è stato sviluppato il testbench riportato di seguito, il quale ha prodotto le forme d'onda raffigurate in Figura 7.7.

```

entity testbench is
end testbench;

architecture Behavioral of testbench is
component TopModule
port(
    clock : in std_logic;
    reset : in std_logic;
    start : in std_logic;
    input1 : in std_logic_vector(15 downto 0);
    input2 : in std_logic_vector(15 downto 0);
    output1 : out std_logic_vector(15 downto 0);
    output2 : out std_logic_vector(15 downto 0)
);
end component;

--Inputs
signal clock : std_logic := '0';

```

```

signal reset : std_logic := '0';
signal start : std_logic := '0';
signal A_input1 : std_logic_vector(15 downto 0) := "0011110011000011" ;
signal A_input2 : std_logic_vector(15 downto 0) := "1010101001010101" ;

--Outputs
signal B_output1 : std_logic_vector(15 downto 0);
signal B_output2 : std_logic_vector(15 downto 0);

-- Clock period definitions
constant clock_period : time := 10 ns;

begin

-- Instantiate the Unit Under Test (UUT)
uut: TopModule port map (
    clock => clock,
    reset => reset,
    start => start,
    input1 => A_input1,
    input2 => A_input2,
    output1 => B_output1,
    output2 => B_output2
);

clock_process :process
begin
    clock <= '0';
    wait for clock_period/2;
    clock <= '1';
    wait for clock_period/2;
end process;

stim_proc: process
begin
    wait for 100 ns;
    start<='1';
    wait for clock_period;
    start<='0';
    wait for clock_period;
    wait;
end process;

end Behavioral;

```

Come si vede dalla simulazione, la trasmissione avviene correttamente. Infatti, tramite la

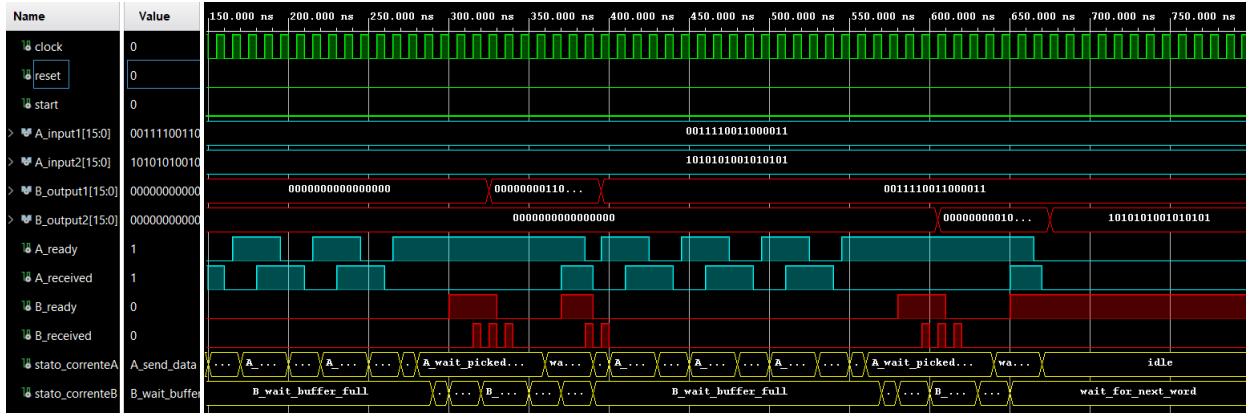


Figura 7.7: Simulazione della comunicazione tramite buffer

Tabella 7.1, è possibile notare come siano effettivamente trasmessi i bit corretti al momento giusto: nella prima comunicazione vengono trasmessi i gruppi di bit **rossi** e **azzurri**, che vengono poi prelevati in ordine dal sistema B, mentre nella seconda trasmissione vengono inviati i gruppi di bit **verdi** e **gialli**, prelevati nuovamente dal sistema B in seconda battuta.

| | Trasmitted bits | | | | Received bits | | |
|----|-----------------|------|------|------|---------------|-----------|-----------|
| | A1 | 0011 | 1100 | 1100 | 0011 | B1 | 0011-1100 |
| A2 | 1010 | 1010 | 0101 | 0101 | B2 | 1010-1010 | 0101-0101 |

Tabella 7.1: Tabella di trasmissione dei bit

Capitolo 8

Esercizio 7

8.1 Traccia

Progettare un sistema in grado di calcolare il prodotto scalare fra 2 vettori A e B di M elementi, ciascuno codificato su N bit (M ed N a scelta dello studente), definito dalla formula:

$$\sum_{i=0}^{M-1} A(i) \cdot B(i)$$

Il sistema deve essere alimentato con k coppie di vettori A e B diversi (cioè $[A_0, B_0]$, $[A_1, B_1]$, ..., $[A_k, B_k]$), forniti in uno dei modi seguenti (a scelta dello studente):

1. Tutti i vettori A_j e B_j ($j=1,\dots,k$) sono precaricati in una ROM, e ciascuna coppia è fornita alla macchina in parallelo;
2. Tutti i vettori sono precaricati, e la macchina riceve serialmente gli elementi di ciascuna coppia di vettori tramite l'ausilio di registri a scorrimento (es., nel caso di $M = 3$, vengono forniti in sequenza $[A_0(0), B_0(0)]$, poi $[A_0(1), B_0(1)]$, e poi $[A_0(2), B_0(2)]$; successivamente, vengono forniti $[A_1(0), B_1(0)]$, $[A_1(1), B_1(1)]$ e $[A_1(2), B_1(2)]$, e così via);
3. Ciascuna coppia di vettori viene ricevuta da un'entità produttore mediante handshaking (e gestita in modalità parallela o seriale a seconda dell'architettura scelta);

Lo studente, inoltre, può scegliere di realizzare un datapath pipelined o meno, e di utilizzare la logica cablata o micropogrammata per l'unità di controllo.

8.2 Soluzione

Per la risoluzione dell'esercizio si è scelto di precaricare la coppia di vettori all'interno di una ROM, inviando alla macchina ciascuna coppia di componenti in parallelo (**soluzione 1**).

L'unità di controllo è stata realizzata in logica cablata, sfruttando un datapath di tipo **pipelined**. L'unità operativa, invece, è stata realizzata tramite il livello di astrazione strutturale, come combinazione di sottocomponenti.

8.2.1 Unità di controllo

L'**unità di controllo** si occupa della gestione delle operazioni; in particolare, genera i segnali necessari ad abilitare le fasi di lettura delle componenti dai registri e di coordinare le operazioni di moltiplicazione ed addizione al giusto tempo.

Inoltre, a valle della ricezione di un segnale di terminazione del conteggio, l'unità di controllo si occupa di abilitare il salvataggio del risultato finale all'interno di un apposito registro.

8.2.2 Unità operativa

L'**unità operativa** è responsabile dell'effettiva operazione di prodotto scalare. Tramite un **moltiplicatore** tale unità effettua la moltiplicazione tra le componenti dei vettori presenti in ROM, e di effettuare la somma tra i risultati parziali tramite un **addizionatore**.

8.3 Schematici

Per mostrare il funzionamento generale dei componenti, è stata definita dapprima l'architettura di interazione tra parte operativa e parte di controllo, per poi passare alla progettazione e alla descrizione architettonale delle due unità nello specifico.

Il dialogo tra unità operativa di controllo consta di 5 particolari segnali:

- **A/B**: tali segnali indicano il precaricamento delle componenti dei due vettori all'interno dell'unità operativa;
- **read_en**: il segnale è inviato dall'unità di controllo, ed abilita la lettura da parte dell'unità operativa dei registri che contengono le componenti;
- **start**: il segnale viene inviato dall'unità di controllo, ed è responsabile dell'avvio del processo di prodotto scalare;
- **acc_en**: il segnale, in uscita dall'unità di controllo, abilita un registro responsabile dello storage del risultato parziale;
- **result_en**: segnale dall'unità di controllo che abilita il salvataggio del risultato finale all'interno di un registro;
- **end**: segnale in ingresso all'unità di controllo che serve a notificare l'effettiva terminazione dell'operazione di prodotto tra le componenti.

La struttura dei segnali così descritti è visualizzabile in Figura 8.1.

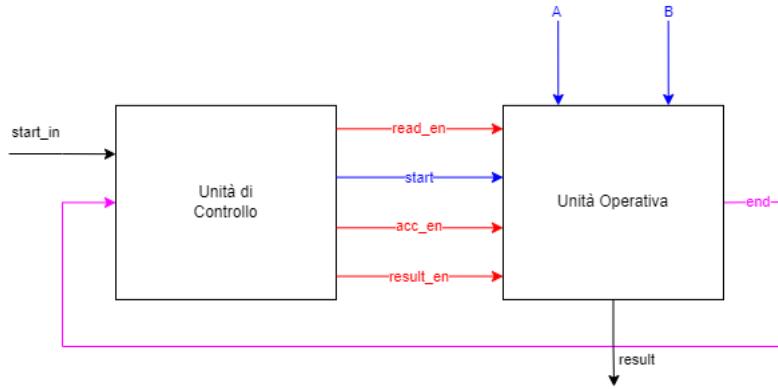


Figura 8.1: Architettura generica della macchina Prodotto Scalare

8.3.1 Unità operativa

L'unità operativa è costituita da differenti componenti: due componenti **VectorA** e **VectorB** si occupano della trasmissione delle componenti all'unità **ALU**, che è il componente responsabile del calcolo del prodotto e della somma tra le componenti dei vettori. Un **ResultRegister** si occupa di conservare il risultato dell'operazione di prodotto scalare.

L'unità operativa ha quindi la struttura rappresentata in Figura 8.2.

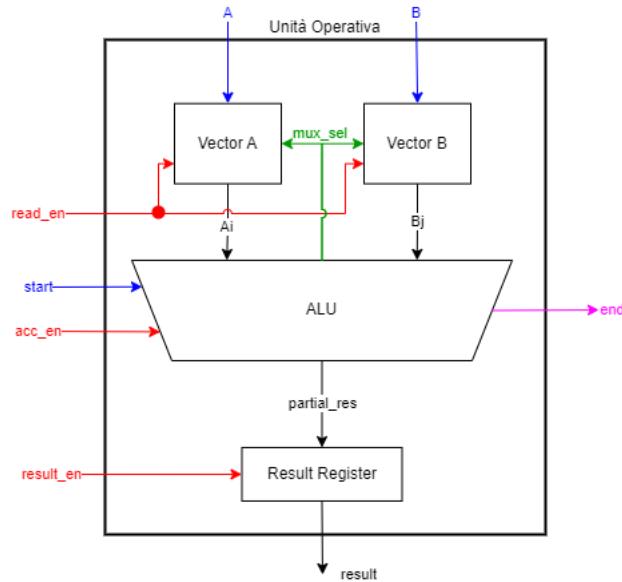


Figura 8.2: Architettura generica dell'unità operativa

Vector

Il componente **Vector** è composto da un decoder, che serve a selezionare il registro all'interno del quale caricare la componente del vettore, da un banco di registri, in numero pari al numero di componenti dei vettori in ingresso, e da un multiplexer, abilitato dal segnale `mux_sel` che l'**ALU**

da ai due componenti al termine di ogni prodotto, che serve a selezionare la componente corretta da moltiplicare.

L'architettura del componente Vector è mostrata in Figura 8.3.

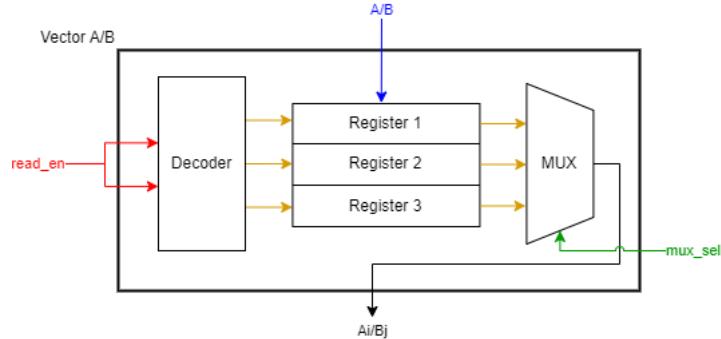


Figura 8.3: Architettura del componente Vector

ALU

Il cuore dell’unità operativa è senza dubbio il componente **ALU**. Tale componente si occupa di effettuare l’operazione di prodotto scalare, effettuando in prima battuta le moltiplicazioni, e poi sommare i risultati parziali, tramite un addizionatore di tipo **Ripple Carry Adder**, un tipo di addizionatore che tiene conto del riporto in caso di overflow dei bit. A valle di tale componente è presente un registro, chiamato **accumulator**, che si occupa di conservare le somme parziali delle componenti moltiplicate, per poi riporle in ingresso all’adder. Tale registro viene abilitato solo nel caso in cui il moltiplicatore abbia terminato di effettuare le sue operazioni, questo per evitare problemi di dati inconsistenti durante l’esecuzione delle operazioni di prodotto scalare.

Il meccanismo di moltiplicazione è affidato al componente **Moltiplicatore di Booth**, che esegue l'algoritmo di Booth per il calcolo della moltiplicazione di due numeri in complemento a due; tale algoritmo è rappresentato in Figura 8.4.

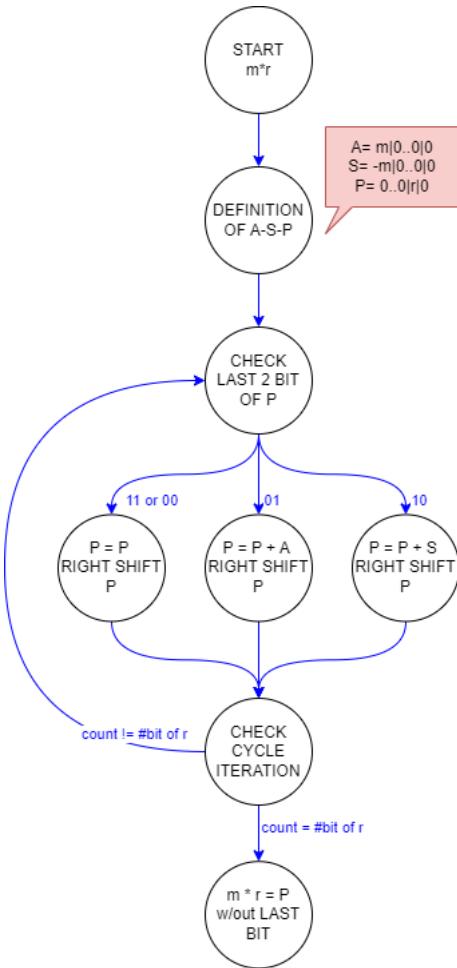


Figura 8.4: Algoritmo di Booth per la moltiplicazione con segno

Una particolarità di tale algoritmo è il tipo di shift eseguito a valle della eventuale rivalutazione del numero P. Si tratta infatti di uno **shift aritmetico**: lo shift aritmetico è una particolare operazione di shift, talvolta definito come *shift con segno* (anche se non è limitato agli operandi con segno). I due tipi fondamentali sono lo **shift aritmetico a sinistra** e lo **shift aritmetico a destra**, quello applicato nell'algoritmo di Booth. Per i numeri binari tale operazione è di tipo bitweise, che sposta tutti i bit del suo operando: ogni bit dell'operando viene semplicemente spostato di un determinato numero di posizioni e le posizioni libere vengono riempite. Invece di essere riempito con tutti 0, come nel caso dello spostamento logico, quando si sposta a destra, il bit più a sinistra (di solito il bit di segno nelle rappresentazioni di interi firmati) viene replicato per riempire tutte le posizioni vacanti (si tratta di una sorta di estensione del segno).

Infine, un elemento **contatore mod3** si occupa di conteggiare il numero di moltiplicazioni effettuate tra le componenti dei vettori, e di dare quindi un segnale di uscita all'unità di controllo per notificare l'effettiva terminazione delle operazioni non appena si raggiunge il numero di componenti moltiplicate. Anche il contatore viene incrementato solo a valle della terminazione effettiva delle operazioni di moltiplicazione, per evitare incrementi fantasma del contatore che causerebbero

malfunzionamenti nell'intero processo. Il modulo del conteggio è pari al numero di componenti dei vettori, nel caso in esame 3.

L'architettura interna dell'ALU è raffigurata in Figura 8.5.

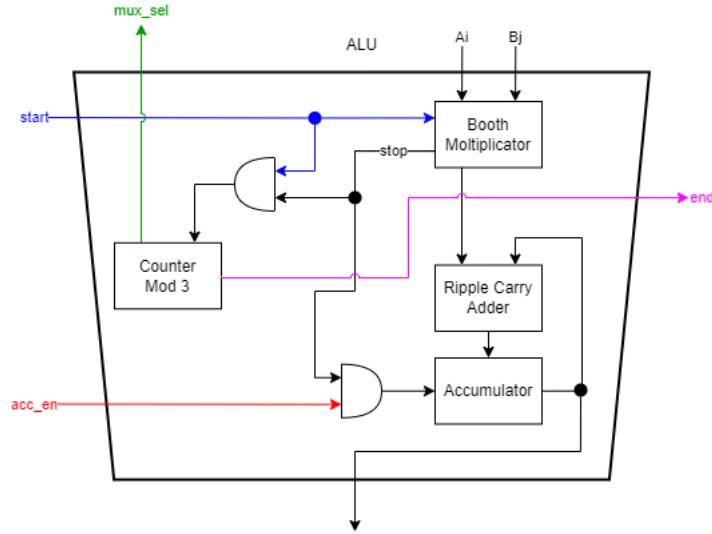


Figura 8.5: Architettura interna dell'unità aritmetico-logica

8.3.2 Unità di controllo

L'unità di controllo è descrivibile tramite un diagramma degli stati:

- Il sistema resta in IDLE fino alla ricezione del segnale di **start**.
- Successivamente al segnale di **start**, il sistema procede in 3 stati, sequenzialmente, durante i quali vengono lette le componenti dei vettori su cui effettuare le operazioni.
- A valle della lettura dei dati, vengono eseguite le operazioni di moltiplicazione e addizione, fino alla ricezione di un segnale di **end** da parte dell'unità operativa.

L'automa così descritto è rappresentato in Figura 8.6.

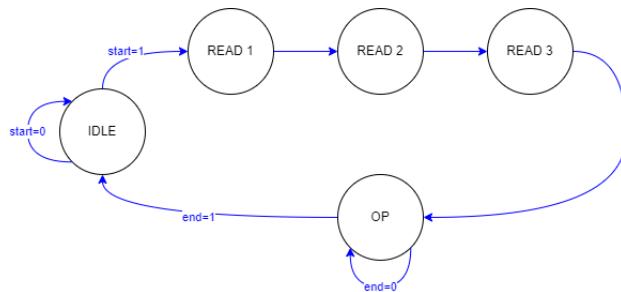


Figura 8.6: Diagramma degli stati dell'unità di controllo

8.4 Codice

8.4.1 Unità Operativa

È buona norma procedere prima con la codifica dell'unità operativa e poi definire il comportamento di quella di controllo. L'unità operativa è formata da differenti componenti:

- 2 componenti Vector per l'immissione delle componenti all'interno dell'ALU;
- 1 unità aritmetica che effettua le moltiplicazioni e le addizioni tra le componenti in ingresso;
- 1 registro finale abilitato che contiene il risultato finale dell'operazione di prodotto scalare.

Di seguito vengono indicate le componenti suddette, procedendo mediante un approccio bottom-up.

Registro Generic

Si è scelto di definire un componente Registro di tipo generic, visto che è necessario istanziare registri sia da 8 che da 16 bit, in base a se essi contengono il risultato o le componenti.

```
entity registro is
    generic(
        N : positive := 8 );
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            en : in STD_LOGIC;
            D : in STD_LOGIC_VECTOR (N-1 downto 0);
            D_out : out STD_LOGIC_VECTOR (N-1 downto 0));
end registro;

architecture Behavioral of registro is

    signal out_temp : STD_LOGIC_VECTOR (N-1 downto 0) := (others=>'0');

begin
    --Processo di funzionamento del registro
    process(clock, reset)
    begin
        if (reset = '1') then
            out_temp<=(others=>'0');
        elsif(clock'event AND clock='1') then
            if (en='1') then
                out_temp<=D;
            end if;
        end if;
    end process;

    --Uscita concorrente
end;
```

```
D_out <= out_temp;
```

```
end Behavioral;
```

Vector

Il componente Vector si compone di:

- 1 decoder per abilitare il registro in scrittura;
- un numero di registri da 8 bit pari al numero di componenti da cui sono composti i vettori in ingresso, in questo caso 3;
- 1 multiplexer per la lettura del registro abilitato.

I comportamenti del decoder e del multiplexer sono stati integrati all'interno dell'architettura del componente, definendone il funzionamento sottoforma di process, mentre i registri sono degli appositi componenti. L'architettura del componente Vector così definita è di tipo ibrido.

```
entity Vector is
    Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
           read_en : in STD_LOGIC_VECTOR (1 downto 0);
           mux_sel : in STD_LOGIC_VECTOR (1 downto 0);
           data_out : out STD_LOGIC_VECTOR (7 downto 0);
           clock : in STD_LOGIC;
           reset : in STD_LOGIC);
end Vector;

architecture hybrid of Vector is
    --Componenti utili
    -- REGISTRO
    component registro is
        generic(
            N : positive := 8 );
        Port ( clock : in STD_LOGIC;
               reset : in STD_LOGIC;
               en : in STD_LOGIC;
               D : in STD_LOGIC_VECTOR (N-1 downto 0);
               D_out : out STD_LOGIC_VECTOR (N-1 downto 0));
    end component;
    --Segnali interni
    type words is array(2 downto 0) of std_logic_vector(7 downto 0);
    signal data_out_reg : words;
    signal reg_en : std_logic_vector(2 downto 0);

begin
    --Istanziazione Registri (M=8, N=3, numero di registri=3 da 8)
    Reg : for i in 0 to 2 generate
```

```

reg_i : registro generic map(8)
port map(
    clock => clock,
    reset => reset,
    en => reg_en(i),
    D => data_in,
    D_out => data_out_reg(i)
);
end generate;

-- Process per il funzionamento del DECODER
process(read_en)
begin
    case read_en is
        when "00" => reg_en<="000";
        when "01" => reg_en<="001";
        when "10" => reg_en<="010";
        when "11" => reg_en<="100";
        when others => reg_en<= "000";
    end case;
end process;

-- Process per il funzionamento del MULTIPLEXER
process(data_out_reg,mux_sel)
begin
    case mux_sel is
        when "00" => data_out<=(others=>'0');
        when "01" => data_out<=data_out_reg(0);
        when "10" => data_out<=data_out_reg(1);
        when "11" => data_out<=data_out_reg(2);
        when others => data_out<=(others=>'0');
    end case;
end process;
end hybrid;

```

Moltiplicatore di Booth

L'unità che si occupa di effettuare la moltiplicazione delle componenti è il moltiplicatore di Booth. Per tale architettura si è scelto di descriverne il comportamento, e quindi definire il componente di tipo behavioral, andando a creare il componente di tipo generic per rendere l'architettura più generalizzabile possibile, e andando a definire il process codificando l'algoritmo di Booth descritto nei paragrafi precedenti.

```

entity booth_multiplier is
    -- Il componente esegue il prodotto m * r
    generic (
        x : positive := 8; --numero di bit di m

```

```

y : positive := 8); --numero di bit di r

Port(clock : in std_logic;
      reset : in std_logic;
      start : in std_logic;
      m : in std_logic_vector(x - 1 downto 0);
      r : in std_logic_vector(y - 1 downto 0);
      result : out std_logic_vector(x + y - 1 downto 0);
      stop : out std_logic);

end booth_multiplier;

architecture Behavioral of booth_multiplier is

begin
  process(clock, reset)
    constant X_ZEROS : std_logic_vector(x - 1 downto 0) := (others => '0');
    constant Y_ZEROS : std_logic_vector(y - 1 downto 0) := (others => '0');

    -- le variabili a, s e p servono per l'esecuzione dell'algoritmo di Booth
    variable a, s, p : std_logic_vector(x + y + 1 downto 0);
    variable mn      : std_logic_vector(x - 1 downto 0);

    begin
      if (reset = '1') then
        result <= (others=>'0');
        stop <= '0';
      end if;
      if(clock'event and clock = '1') then
        if (start = '1') then
          --Inizializzazione di a, s e p
          a := (others => '0');
          s := (others => '0');
          p := (others => '0');

          -- Se m o r sono zero, il risultato è nullo in ogni caso
          if (m /= X_ZEROS and r /= Y_ZEROS) then

            --Settaggio di a nel formato m/0...0/0
            a(x + y downto y + 1) := m;
            a(x + y + 1) := m(x - 1);

            --Settaggio di s nel formato -m/0...0/0
            mn := (not m) + 1;
            s(x + y downto y + 1) := mn;
            s(x + y + 1) := not(m(x - 1));

```

```

--Setteggia di p nel formato 0...0/r/0
p(y downto 1) := r;

--Check degli ultimi due bit di p
for i in 1 to y loop
    -- è necessario ridefinire p solo se non sono '00' o '11'
    if (p(1 downto 0) = "01") then
        p := p + a;
    elsif (p(1 downto 0) = "10") then
        p := p + s;
    end if;

    -- Shift aritmetico a destra
    -- (fatto in ogni caso)
    p(x + y downto 0) := p(x + y + 1 downto 1);
end loop;
end if;
--Scrittura del risultato (si esclude l'ultimo bit di p)
result <= p(x + y downto 1);
--Segnalazione della terminazione dell'operazione
stop <= '1';
end if;
end if;
end process;

end Behavioral;

```

Ripple Carry Adder

Per effettuare le somme tra le componenti moltiplicate, si è scelto un componente RippleCarryAdder, che esegue la somma bit a bit di due sequenze di bit in ingresso. Tale architettura sfrutta il componente FullAdder, di tipo dataflow, descritto nei capitoli precedenti. L'architettura del Ripple Carry Adder è quindi di tipo strutturale.

```

entity RippleCarryAdder is
    Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
           B : in STD_LOGIC_VECTOR (15 downto 0);
           carry : in STD_LOGIC;
           ris : out STD_LOGIC_VECTOR (15 downto 0);
           cout : out STD_LOGIC);
end RippleCarryAdder;

architecture structural of RippleCarryAdder is
    --Carry interni
    signal c_i: std_logic_vector(0 to 14) := (others => '0');
    -- Full Adder interni
    component fullAdder is

```

```

Port (
    A: in std_logic;
    B: in std_logic;
    c_in: in std_logic;
    c_out:out std_logic;
    s: out std_logic
);
end component;

begin
--INSTANZIAZIONE FULL ADDER
-- Full Adder 0
F_Adder_0: fullAdder
port map(
    A=>A(0),
    B=>B(0),
    c_in=>carry,
    c_out=>c_i(0),
    s=>ris(0)
);
-- Full Adder 1-14
F_Adder : for i in 1 to 14 generate
    f_add_i : fullAdder
    port map(
        A=>A(i),
        B=>B(i),
        c_in=>c_i(i-1),
        c_out=>c_i(i),
        s=>ris(i)
    );
end generate;
-- Full Adder 15
F_Adder_15 : fullAdder
port map(
    A=>A(15),
    B=>B(15),
    c_in=>c_i(14),
    c_out=>cout,
    s=>ris(15)
);
end structural;

```

Contatore Modulo 3

Per segnalare la terminazione e per definire la selezione dei registri in ingresso all'unità operativa, si è reso necessario l'utilizzo di un contatore, modulo 3 in questo caso, per lo scorrimento e per la notifica all'unità di controllo del termine delle operazioni. Il contatore è stato definito di tipo

comportamentale, convertendo in binario un contatore intero all'interno del process di definizione del funzionamento dello stesso contatore.

```

entity counter_mod3 is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           mux_sel : out STD_LOGIC_VECTOR (1 downto 0);
           s_end : out STD_LOGIC;
           start_stop : in STD_LOGIC);
end counter_mod3;

architecture Behavioral of counter_mod3 is

    signal counter : integer range 0 to 3 :=1;

begin
    countProcess: process(clock,reset)
    begin
        if (clock'event and clock ='0') then
            if (reset = '1') then
                counter<=0;
            elsif (start_stop='1') then
                if (counter<3) then
                    counter<=counter+1;
                    s_end<='0';
                else
                    counter<=0;
                    s_end<='1';
                end if;
            end if;
            mux_sel<=conv_std_logic_vector(counter,2);
        end process;
    end Behavioral;

```

ALU

L'unità ALU si compone di differenti elementi:

- 1 moltiplicatore di Booth per la moltiplicazione;
- 1 Ripple Carry Adder per la somma delle componenti moltiplicate;
- 1 contatore per lo scorrimento dei vettori;
- 1 registro da $8+8 = 16$ bit contenente il risultato finale di prodotto.

L'architettura definita è di tipo strutturale, e fa uso delle componenti descritte in precedenza.

```

entity ALU is
    Port ( comp_A : in STD_LOGIC_VECTOR (7 downto 0);
           comp_B : in STD_LOGIC_VECTOR (7 downto 0);
           start_ALU : in STD_LOGIC;
           acc_en : in STD_LOGIC;
           s_end : out STD_LOGIC;
           clock : in STD_LOGIC;
           mux_sel : out STD_LOGIC_VECTOR(1 downto 0);
           reset : in STD_LOGIC;
           partial_res : out STD_LOGIC_VECTOR (15 downto 0));
end ALU;

architecture structural of ALU is

--MOLTIPLICATORE DI BOOTH (BoothMultiplier)
component booth_multiplier is
    generic (
        x : positive := 8;
        y : positive := 8);

    Port(clock : in std_logic;
          reset : in std_logic;
          start : in std_logic;
          m : in std_logic_vector(x - 1 downto 0);
          r : in std_logic_vector(y - 1 downto 0);
          result : out std_logic_vector(x + y - 1 downto 0);
          stop : out std_logic);
end component;

-- CONTATORE MODULO 3 (CounterMod3)
component counter_mod3 is
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           mux_sel : out STD_LOGIC_VECTOR (1 downto 0);
           s_end : out STD_LOGIC;
           start_stop : in STD_LOGIC);
end component;

--ADDER (Ripple Carry Adder)
component RippleCarryAdder is
    Port ( A : in STD_LOGIC_VECTOR (15 downto 0);
           B : in STD_LOGIC_VECTOR (15 downto 0);
           carry : in STD_LOGIC;
           ris : out STD_LOGIC_VECTOR (15 downto 0);
           cout : out STD_LOGIC);
end component;

```

```

--REGISTRO 16 Bit (Accumulator)
component registro is
    generic(
        N : positive := 8 );
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            en : in STD_LOGIC;
            D : in STD_LOGIC_VECTOR (N-1 downto 0);
            D_out : out STD_LOGIC_VECTOR (N-1 downto 0));
end component;

-- SEGNALI INTERNI
signal stop_start : std_logic := '0';
signal booth_res : std_logic_vector(15 downto 0) := (others=>'0');
signal stop_booth : std_logic := '0';
signal cout : std_logic := '0';
signal out_RCA : std_logic_vector(15 downto 0) := (others=>'0');
signal start_acc_en : std_logic := '0';
signal ris_inout : std_logic_vector(15 downto 0) := (others=>'0');

begin

    --Processi di definizione dei segnali interni
    start_acc_en <= start_ALU and acc_en;
    stop_start <= start_ALU and stop_booth;
    partial_res <= ris_inout;

    --Definizione strutturale dei componenti interni
    BOOTH : booth_multiplier generic map(8,8)
        port map(
            clock => clock,
            reset => reset,
            start => start_ALU,
            m => comp_A,
            r => comp_B,
            result => booth_res,
            stop => stop_booth
        );

    RCA : RippleCarryAdder
        port map(
            A => booth_res,
            B => ris_inout,
            carry => '0',
            ris => out_RCA,
            cout => cout
        );

```

```

ACCUMULATOR : registro generic map(16)
port map(
    clock => clock,
    reset => reset,
    en => start_acc_en,
    D => out_RCA,
    D_out => ris_inout
);
COUNTER : counter_mod3
port map(
    clock => clock,
    reset => reset,
    start_stop => stop_start,
    mux_sel => mux_sel,
    s_end => s_end
);
end structural;

```

UnitàOperativa

Il componente finale UnitàOperativa viene descritto di tipo strutturale, e fa uso delle componenti già precedentemente descritte, secondo lo schema raffigurato nel paragrafi precedenti.

```

entity UnitàOperativa is
  Port(
    A : in std_logic_vector(7 downto 0);
    B : in std_logic_vector(7 downto 0);
    clock : in std_logic;
    reset : in std_logic;
    read_en : in std_logic_vector(1 downto 0);
    start : in std_logic;
    acc_en : in std_logic;
    result_en : in std_logic;
    s_end : out std_logic;
    result : out std_logic_vector(15 downto 0)
  );
end UnitàOperativa;

architecture structural of UnitàOperativa is
-- COMPONENTI UTILI: Vettore
component Vector is
  Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
         read_en : in STD_LOGIC_VECTOR (1 downto 0);
         mux_sel : in STD_LOGIC_VECTOR (1 downto 0);
         data_out : out STD_LOGIC_VECTOR (7 downto 0);

```

```

    clock : in STD_LOGIC;
    reset : in STD_LOGIC);
end component;

-- COMPONENTI UTILI: Unità Aritmetica

component ALU is
    Port ( comp_A : in STD_LOGIC_VECTOR (7 downto 0);
           comp_B : in STD_LOGIC_VECTOR (7 downto 0);
           start_ALU : in STD_LOGIC;
           acc_en : in STD_LOGIC;
           s_end : out STD_LOGIC;
           clock : in STD_LOGIC;
           mux_sel : out STD_LOGIC_VECTOR(1 downto 0);
           reset : in STD_LOGIC;
           partial_res : out STD_LOGIC_VECTOR (15 downto 0));
end component;

-- COMPONENTI UTILI: Registro 16 bit

component registro is
    generic(
        N : positive := 8 );
    Port ( clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           en : in STD_LOGIC;
           D : in STD_LOGIC_VECTOR (N-1 downto 0);
           D_out : out STD_LOGIC_VECTOR (N-1 downto 0));
end component;

-- SEGNALI INTERNI

signal mux_sel : std_logic_vector(1 downto 0) := (others=>'0');
signal compA : std_logic_vector(7 downto 0) := (others=>'0');
signal compB : std_logic_vector(7 downto 0) := (others=>'0');
signal partial_res : std_logic_vector(15 downto 0) := (others=>'0');

begin

VettoreA : Vector
    port map(
        data_in => A,
        read_en => read_en,
        mux_sel => mux_sel,
        clock => clock,
        reset => reset,
        data_out => compA
    );
VettoreB : Vector
    port map(
        data_in => B,
        read_en => read_en,
        mux_sel => mux_sel,
        clock => clock,

```

```

        reset => reset,
        data_out => compB
    );
UnitaAritmetica : ALU
    port map(
        comp_A => compA,
        comp_B => compB,
        start_ALU => start,
        acc_en => acc_en,
        s_end => s_end,
        clock => clock,
        mux_sel => mux_sel,
        reset => reset,
        partial_res => partial_res
    );
ResultRegister : registro generic map(16)
    port map(
        clock => clock,
        reset => reset,
        en => result_en,
        D => partial_res,
        D_out => result
    );
end structural;

```

8.4.2 Unità di Controllo

L'unità di controllo è stata definita tramite process, codificandone l'automa descritto in precedenza attraverso i suoi stati.

```

entity UnitaControllo is
    Port ( clock : in STD_LOGIC;
            reset : in STD_LOGIC;
            start_in : in STD_LOGIC;
            start : out STD_LOGIC;
            s_end : in STD_LOGIC;
            acc_en : out STD_LOGIC;
            result_en : out STD_LOGIC;
            read_en : out STD_LOGIC_VECTOR (1 downto 0));
end UnitaControllo;

architecture Behavioral of UnitaControllo is

    type stato is (idle, read1, read2, read3, op);
    signal stato_corrente, stato_prossimo : stato := idle;

begin

```

```

state : process (clock, reset)
begin
  if (clock'event and clock ='1') then
    if (stato_corrente = idle and reset ='1') then
      stato_corrente <= idle;
    else
      stato_corrente <= stato_prossimo;
    end if;
  end if;
end process;

fsm : process(stato_corrente,stato_prossimo, clock)
begin
  start <= '0';
  read_en <= "00";
  acc_en <= '0';
  result_en <= '0';
  if (reset = '1') then
    stato_prossimo <= idle;
  end if;

  case stato_corrente is
    when idle =>
      start <= '0';
      read_en <= "00";
      acc_en <= '0';
      result_en <= '0';
      if (start_in = '1') then
        stato_prossimo <= read1;
      else
        stato_prossimo <= idle;
      end if;

    when read1 =>
      start <= '0';
      read_en <= "01";
      acc_en <= '1';
      result_en <= '0';
      stato_prossimo <= read2;

    when read2 =>
      start <= '0';
      read_en <= "10";
      acc_en <= '1';
      result_en <= '0';
      stato_prossimo <= read3;
  end case;
end process;

```

```

when read3 =>
    start <= '0';
    read_en <= "11";
    acc_en <= '1';
    result_en <= '0';
    stato_prossimo <= op;

when op =>
    start <= '1';
    read_en <= "00";
    acc_en <= '1';
    if(s_end ='1') then
        result_en <= '1';
        stato_prossimo <= idle;
    else
        stato_prossimo <= op;
    end if;

when others =>
    stato_prossimo <= idle;

end case;
end process;

end Behavioral;

```

8.4.3 Top Module

Il modulo `TopModule` comprende l'unità operativa e quella di controllo, opportunamente collegate, e viene descritto in maniera strutturale, come si vede dal codice di seguito.

```

entity topModule is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           start_in : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           result : out STD_LOGIC_VECTOR (15 downto 0));
end topModule;

architecture structural of topModule is

-- COMPONENTI
component UnitaOperativa is
    Port(

```

```

A : in std_logic_vector(7 downto 0); --
B : in std_logic_vector(7 downto 0); --
clock : in std_logic; --
reset : in std_logic; --
read_en : in std_logic_vector(1 downto 0);
start : in std_logic;
acc_en : in std_logic;
result_en : in std_logic;
s_end : out std_logic;
result : out std_logic_vector(15 downto 0) --
);
end component;

component UnitaControllo is
Port ( clock : in STD_LOGIC; --
reset : in STD_LOGIC; --
start_in : in STD_LOGIC; --
start : out STD_LOGIC;
s_end : in STD_LOGIC;
acc_en : out STD_LOGIC;
result_en : out STD_LOGIC;
read_en : out STD_LOGIC_VECTOR (1 downto 0));
end component;

-- SEGNALI INTERNI
signal read_en_int : std_logic_vector(1 downto 0);
signal start_int : std_logic;
signal acc_en_int : std_logic;
signal result_en_int : std_logic;
signal s_end_int : std_logic;

begin
UO : UnitaOperativa
port map(
A => A,
B => B,
clock => clock,
reset => reset,
read_en => read_en_int,
start => start_int,
acc_en => acc_en_int,
result_en => result_en_int,
s_end => s_end_int,
result => result
);
UC : UnitaControllo

```

```

port map(
    clock => clock,
    reset => reset,
    start_in => start_in,
    start => start_int,
    s_end => s_end_int,
    acc_en => acc_en_int,
    result_en => result_en_int,
    read_en => read_en_int
);

end structural;

```

8.5 Simulazione

Per tale architettura si è deciso dapprima di testare alcune componenti singolarmente, per verificarne il funzionamento, prima di testare l'intero processo di funzionamento della macchina. Si è scelto di effettuare dapprima un test del componente vettore, tramite il seguente testbench.

```

entity Vector_tb is
end Vector_tb;

architecture Behavioral of Vector_tb is
component Vector is
    Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
           read_en : in STD_LOGIC_VECTOR (1 downto 0);
           mux_sel : in STD_LOGIC_VECTOR (1 downto 0);
           data_out : out STD_LOGIC_VECTOR (7 downto 0);
           clock : in STD_LOGIC;
           reset : in STD_LOGIC);
end component;

signal data_in : STD_LOGIC_VECTOR (7 downto 0);
signal read_en : STD_LOGIC_VECTOR (1 downto 0);
signal mux_sel : STD_LOGIC_VECTOR (1 downto 0);
signal data_out : STD_LOGIC_VECTOR (7 downto 0);
signal clock : STD_LOGIC;
signal reset : STD_LOGIC;

constant clk_per : TIME := 10 ns;
begin
    uut : Vector
    port map(
        data_in => data_in,
        read_en => read_en,

```

```

    mux_sel => mux_sel,
    data_out => data_out,
    clock => clock,
    reset => reset
);

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

test : process
begin
    reset <= '1';
    wait for 2*clk_per;
    reset <= '0';

    data_in <= "01001101";
    read_en <= "11";
    wait for clk_per;
    mux_sel <= "11";
    wait for clk_per;

    data_in <= "10010011";
    read_en <= "10";
    wait for clk_per;
    mux_sel <= "10";
    wait for clk_per;

    data_in <= "00110101";
    read_en <= "01";
    wait for clk_per;
    mux_sel <= "01";
    wait for 2*clk_per;
    reset <= '1';
    data_in <= "00000000";
    read_en <= "00";
    mux_sel <= "00";
    wait;
end process;

end Behavioral;

```

Tale testbench ha prodotto la simulazione raffigurata in Figura 8.7. Come si nota dalla simulazione

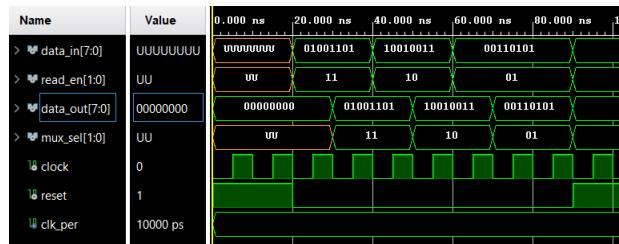


Figura 8.7: Simulazione del componente Vector

di tale unità, il segnale `read_en` regola la fase di scrittura del componente, mentre il segnale `mux_sel` gestisce la fase di lettura dai registri.

Successivamente, vista la particolarità del moltiplicatore di Booth, è stato ritenuto opportuno testare anche il suo funzionamento in maniera autonoma, tramite l'apposito testbench di seguito.

```

entity booth_multiplier_tb is
end booth_multiplier_tb;

architecture Behavioral of booth_multiplier_tb is

component booth_multiplier is
generic (
    x : integer := 8; --numero di bit di m
    y : integer := 8); --numero di bit di r

    Port(clock : in std_logic;
        reset : in std_logic;
        start : in std_logic;
        m : in std_logic_vector(x - 1 downto 0);
        r : in std_logic_vector(y - 1 downto 0);
        result : out std_logic_vector(x + y - 1 downto 0);
        stop : out std_logic);

end component;

begin
    uut : booth_multiplier generic map(3,3)

```

```

port map(
    clock => clock,
    reset => reset,
    start => start,
    m => m,
    r => r,
    result => result,
    stop => stop);

clk: process
begin
    clock <= '0';
    wait for clk_per;
    clock <= '1';
    wait for clk_per;
end process;

test: process
begin
    m <= "010";
    r <= "100";
    wait for clk_per;
    start <= '1';
    wait for clk_per;

    m <= "011";
    r <= "010";
    wait for clk_per;
    start <= '1';
    wait for clk_per;

    m <= "110";
    r <= "101";
    wait for clk_per;
    start <= '1';
    wait for clk_per;

    m <= "000";
    r <= "000";
    wait for clk_per;
    start <= '1';
    wait;
end process;
end Behavioral;

```

Tale testbench ha prodotto la simulazione mostrata in Figura 8.8.



Figura 8.8: Simulazione del moltiplicatore di Booth

Stesso trattamento è stato riservato all'Unità Aritmetico-Logica; si è quindi generato un test-bench apposito per il test di funzionamento delle operazioni di moltiplicazione e somma, riportato di seguito.

```

entity ALU_tb is
end ALU_tb;

architecture Behavioral of ALU_tb is

-- COMPONENTE DA TESTARE
component ALU is
    Port ( comp_A : in STD_LOGIC_VECTOR (7 downto 0);
           comp_B : in STD_LOGIC_VECTOR (7 downto 0);
           start_ALU : in STD_LOGIC;
           acc_en : in STD_LOGIC;
           s_end : out STD_LOGIC;
           clock : in STD_LOGIC;
           mux_sel : out STD_LOGIC_VECTOR(1 downto 0);
           reset : in STD_LOGIC;
           partial_res : out STD_LOGIC_VECTOR (15 downto 0));
end component;

-- SEGNALI INTERNI
signal comp_A : STD_LOGIC_VECTOR (7 downto 0);
signal comp_B : STD_LOGIC_VECTOR (7 downto 0);
signal start_ALU : STD_LOGIC;
signal acc_en : STD_LOGIC;
signal s_end : STD_LOGIC;
signal clock : STD_LOGIC;
signal mux_sel : STD_LOGIC_VECTOR(1 downto 0);
signal reset : STD_LOGIC;
signal partial_res : STD_LOGIC_VECTOR (15 downto 0);

-- COSTANTI
constant clk_per : TIME := 10 ns;

```

```

begin
    uut : ALU
    port map(
        comp_A => comp_A,
        comp_B => comp_B,
        start_ALU => start_ALU,
        acc_en => acc_en,
        s_end => s_end,
        clock => clock,
        mux_sel => mux_sel,
        reset => reset,
        partial_res => partial_res
    );
end Behavioral;

```

```

clk: process
begin
    clock <= '0';
    wait for clk_per;
    clock <= '1';
    wait for clk_per;
end process;

test : process
begin
    wait for 100ns;
    comp_A <= "01001101";
    comp_B <= "11011010";
    wait for 4*clk_per;
    start_ALU <= '1';
    wait for 4*clk_per;
    acc_en <= '1';
end process;

```

La simulazione mediante tale testbench ha prodotto le forme d'onda riportate in Figura 8.9.

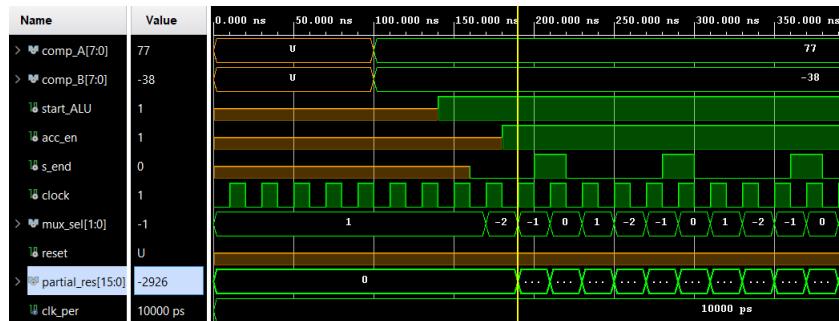


Figura 8.9: Simulazione dell'unità Aritmetica

Infine, a valle del test delle varie componenti, si è proceduto al test del top module, che racchiude la logica operativa e di controllo. È stato quindi prodotto un testbench apposito per testarne il funzionamento, ottenendo la simulazione rappresentata in Figura 8.10.

```

entity topModule_tb is
end topModule_tb;

architecture Behavioral of topModule_tb is

-- COMPONENTE DA TESTARE
component topModule is
    Port ( A : in STD_LOGIC_VECTOR (7 downto 0);
           B : in STD_LOGIC_VECTOR (7 downto 0);
           start_in : in STD_LOGIC;
           clock : in STD_LOGIC;
           reset : in STD_LOGIC;
           result : out STD_LOGIC_VECTOR (15 downto 0));
end component;

-- SEGNALI DI TEST
signal A : STD_LOGIC_VECTOR (7 downto 0) :=(others=>'0');
signal B : STD_LOGIC_VECTOR (7 downto 0):=(others=>'0');
signal start_in : STD_LOGIC :='0';
signal clock : STD_LOGIC :='0';
signal reset : STD_LOGIC :='0';
signal result : STD_LOGIC_VECTOR (15 downto 0):=(others=>'0');

-- COSTANTI
constant clk_per : TIME := 10ns;

begin

uut : topModule
port map(
    A=> A,
    B=> B,
    start_in => start_in,
    clock => clock,
    reset => reset,
    result => result
);

clk : process
begin
    clock <= '0';
    wait for clk_per/2;
    clock <= '1';

```

```

        wait for clk_per/2;
    end process;

test : process
begin
    reset <= '1';
    wait for 5*clk_per;
    reset <= '0';
    wait for clk_per;

    start_in <= '1';
    wait for clk_per;
    start_in <= '0';
    A <= "00000010";
    B <= "00000110";
    wait for clk_per;
    A <= "00000111";
    B <= "00000011";
    wait for clk_per;
    A <= "00000010";
    B <= "00000101";
    wait for clk_per;
    A <= "00000000";
    B <= "00000000";

    wait;
end process;

```

```
end Behavioral;
```

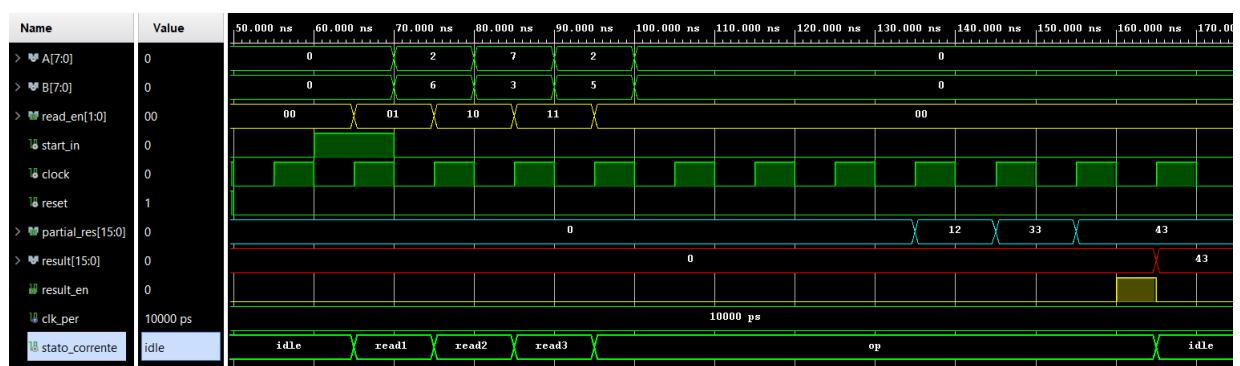


Figura 8.10: Simulazione della macchina Prodotto Scalare

Capitolo 9

Esercizio 8

9.1 Traccia

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

- a) si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta,
- b) si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate,
- c) (opzionale) si descriva il funzionamento del processore in merito alle istruzioni di input/output,
- d) (solo ove possibile) si sintetizzi il processore su FPGA.

9.2 Introduzione

Il **MIC-1** è l'architettura di un processore ideata da Tanenbaum.

Tale processore nasce come interprete hardware del bytecode, generato da un programma Java e consiste di un'unità di controllo molto semplice che esegue microcodice da una memoria di 512 parole. Il linguaggio microcodice **MAL** (*Micro Assembly Language*) è realizzato per consentire la scrittura di un'interprete **IJVM** (*Integer Java Virtual Machine*) con semplicità.

Il processore MIC-1 presenta un'architettura a stack ed una logica microprogrammata.

Nelle seguenti sezioni approfondiremo nel dettaglio tali aspetti.

9.2.1 Architettura a stack

Il processore a stack vede la memoria suddivisa in diverse aree. In una di essa vengono memorizzate le costanti ed in un'altra le variabili locali. Tutte le operazioni di calcolo fanno implicitamente riferimento alla testa e al secondo elemento presente nello stack. Ad esempio, l'operazione di

addizione in questa architettura non ha bisogno di operandi, perché essi sono implicitamente i primi elementi dello stack. Questo consente di godere del vantaggio di codici operativi molto brevi, ma dall'altro richiede che tutte le variabili soggetto di calcolo debbano essere poste in testa allo stack. Tale sistema è spesso utilizzato nelle calcolatrici programmabili.

Detto ciò, è inevitabile che per poter lavorare in questo modo è necessario dover disporre dei giusti registri per poterla indirizzare e tener traccia dell'elemento in testa. Per tale compito il processore MIC-1 prevede due registri, lo **stack pointer** (*SP*) e il **top of stack** (*TOS*). Il primo è necessario che sia tracciato con coerenza, in maniera tale che possa sempre puntare all'ultimo elemento spinto nella struttura dati.

9.2.2 Datapath

La parte operativa del processore di Tanenbaum è il **datapath**, riportato in Figura 9.1.

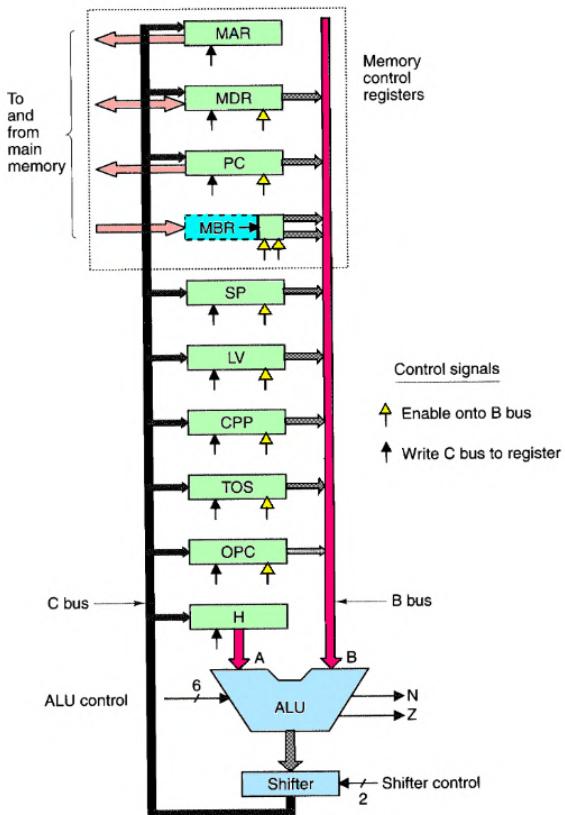


Figura 9.1: Datapath del processore MIC-1

Il datapath è il cuore della CPU, ovvero quella parte che contiene dieci registri e due bus tramite i quali essi possono scambiare dati dopo che ne sia stata abilitata la scrittura o la lettura. In particolare, il bus **B** ha la funzione di **multiplexer**, mentre il bus **C** di **demultiplexer**. Il primo viene utilizzato in lettura mentre il secondo in scrittura.

Dato che l'intento della macchina non è solo lo scambio di dati è presente anche un'ALU che

consente di fare operazioni logico/matematiche di base. Al di sotto dell'ALU troviamo anche uno shift register, utile in alcune operazioni matematiche, quali moltiplicazioni e divisioni binarie.

I registri utili alla comunicazione con la memoria sono:

- **MAR** (*Memory Address Register*);
- **MDR** (*Memory Data Register*);
- **PC** (*Program Counter*);
- **MBR** (*Memory Byte Register*).

Notiamo che si tratta di due coppie di registri, una dedicata alla lettura dell'istruzioni, l'altra coppia invece alla lettura e scrittura dei dati. Risulta quindi evidente come area dati (MAR ed MDR) ed area istruzioni (PC ed MBR) in questo modello siano separate.

Analizzando ulteriormente lo schema si può notare come vi siano due segnali di enable:

- Il primo che abilita la scrittura del contenuto sul bus B;
- Il secondo che abilita la scrittura del contenuto sul bus C.

Di seguito viene descritta la funzionalità degli ulteriori registri:

- **CPP** (*Constant Pool Pointer*) che punta all'area dove sono memorizzate le costanti;
- **LV** (*Local Variable Frame*) che punta alla base dello stack dove sono presenti le variabili locali dei metodi richiamati, oltre che ai parametri con i quali sono stati invocati;
- **SP** (*Stack Pointer*) che punta alla testa dello stack e tiene conto degli operandi inseriti o rimossi;
- **H** (*Holding*) che è un registro tampone utile a memorizzare gli operandi dell'ALU;
- **TOS** (*Top of Stack*) che contiene la copia del valore in memoria della testa dello stack (a differenza di SP, TOS non rappresenta un indirizzo);
- **OPC**, un registro di appoggio di dati utili durante l'esecuzione della microistruzione.

9.2.3 Accesso alla memoria

La memoria del processore è una memoria costituita da parole di 8 bit, univocamente identificate da indirizzi di 32 bit. Dunque, possiamo indirizzare all'interno di tale memoria 2^{32} parole. La CPU ha due modi per comunicare con la memoria:

- Una porta da **32 bit** per la **lettura/scrittura dei dati** del livello **ISA** (*Instruction Set Architecture*). Tale porta viene controllata dal registro **MAR** che specifica l'indirizzo di memoria in cui si desidera leggere o scrivere una parola e il registro **MDR** che ospita la parola (32 bit) che sarà letta o scritta all'indirizzo di memoria specificato da MAR;

- Una porta da **8 bit** per leggere il programma eseguibile (**fetch delle istruzioni ISA**). Anche questa porta è controllata da 2 registri:
 - il **PC** ovvero un registro a **32 bit** che indica l'indirizzo di memoria della prossima istruzione ISA da caricare;
 - il registro **MBR** contiene il byte letto dalla memoria durante il fetch. Esso è in realtà un registro a **32 bit**, pertanto il byte letto viene memorizzato negli **8 bit meno significativi**.

Le due porte, quindi, indirizzano la memoria a parole (32 bit) e a byte (8 bit). Essendo tutte le operazioni aritmetiche riferenti allo stack, se ne deduce che con 8 bit si riesce a codificare la maggior parte delle istruzioni.

Un'importante conseguenza di tutto ciò è per gli indirizzi memorizzati in MAR che sono espressi in termini di parole. Ogni valore letto al suo interno comporta quindi una moltiplicazione per 4, data la differenza da parola e byte. Ciò si risolve non valutando i primi due bit di una parola, ma gli ultimi pari a 00. Infatti, ad esempio, la word 0 la codifco come 00-00, 1 come 01-00 (pari a 4) e 2 come 10-00 (pari ad 8).

Indirizzando invece PC la memoria in termini di byte, la lettura ad esempio dell'indirizzo 2 causa il trasferimento negli 8 bit meno significativi di MBR del byte di memoria 2. Ciò che potrebbe sembrare una complicazione, semplifica in realtà il funzionamento interno in quanto il fetch delle istruzioni può avvenire un byte alla volta mentre risulta possibile leggere o scrivere in memoria una parola in un unico ciclo. Fisicamente la memoria è realizzata come un unico spazio lineare organizzato in byte. È possibile notare come collegando MAR sul bus indirizzi sfalsato di due posizioni, ovvero non utilizzando i due bit più significativi (impostati a 0) e collegando il bit 0 di MAR con il bit 2 degli indirizzi e così via, si potrà indirizzare con MAR la memoria in termini di parole senza bisogno di introdurre alcun circuito. In Figura 9.2 si nota come viene eseguito tale sfasamento di bit.

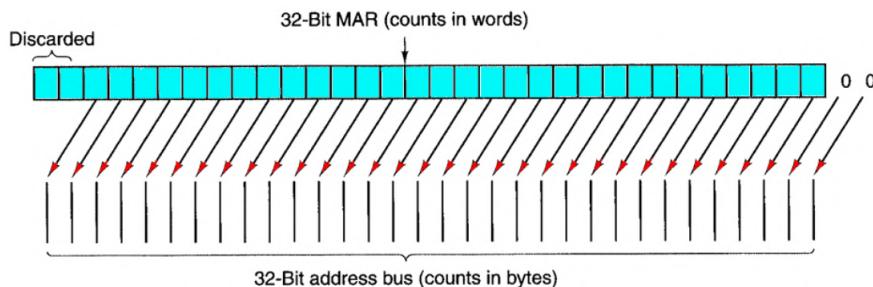


Figura 9.2: Indirizzamento a parole del registro MAR

9.2.4 Comunicazione tra registri

Per quanto riguarda la comunicazione tra i registri, il bus B permette di caricare su di esso il valore di un registro, mentre il bus C permette di scrivere su un registro il contenuto presente su di esso. Ricordiamo inoltre che il primo ha la funzione di MUX mentre il secondo di DEMUX.

Nell'architettura in esame, a prescindere se devo eseguire un'operazione aritmetica o meno, il dato da leggere deve passare sempre per l'ALU. Qualora non dovessi svolgere alcun calcolo addizionerò semplicemente zero. Ogni fase del processore si compone di quattro step (tra due colpi di clock successivi):

1. In un primo quanto di tempo Δw , si preleva l'istruzione dalla memoria di controllo, contenente i bit che pilotano il datapath;
2. Nel quanto di tempo successivo Δx , si seleziona il registro e si scrive sul bus B;
3. Nel quanto di tempo successivo Δy , avvengono le operazioni sull'ALU e vi è un eventuale shift;
4. Nell'ultimo quanto di tempo Δz , si scrive sul bus C.

La Figura 9.3 mostra nel dettaglio le fasi descritte in precedenza.

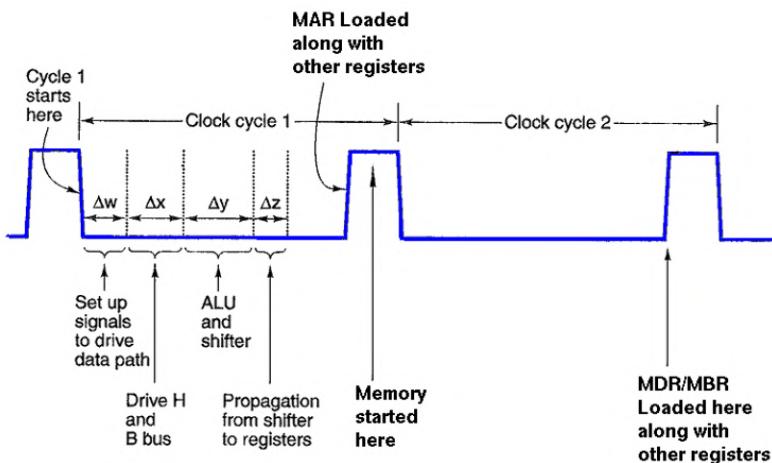


Figura 9.3: Ciclo di clock del MIC-1

Tra due clock occorre dare la possibilità che tutti i segnali si assestino sul bus. Questo tempo di clock è dato dal tempo richiesto dal leggere l'istruzione successiva, dalla selezione del bus B e dal tempo di calcolo richiesto dall'ALU, dalla propagazione dello shift register ed infine, dal caricamento del valore su C.

Questo permette di dimensionare il duty cycle ovvero la distanza tra un fronte di discesa ed un fronte di salita del clock. La somma dei tempi deve garantire che tutti i sistemi funzionino. Se la frequenza del clock è troppo veloce e l'implementazione del FPGA non è efficiente, questo sistema potrebbe causare dei problemi.

Questo sistema è quindi sincrono, si muove su due colpi di clock senza dover aspettare che l'ALU comunichi di aver terminato le proprie operazioni. Inoltre, è un modello transfer register, in cui i dati viaggiano tra i registri tramite due bus e vengono modificati esclusivamente dall'ALU.

È importante notare che sebbene nello stesso ciclo di datapath sia possibile eseguire più operazioni, in un ciclo non è possibile completare la lettura o la scrittura di parole in memoria. Infatti, le memorie non sono in grado di far fronte istantaneamente ad una rischiesta di lettura o scrittura, che non possono quindi essere concluse nello stesso ciclo di clock nel quale è stata inoltrata la richiesta. Nella micro-architettura in esame se viene attivato un segnale di lettura dalla memoria dati (MAR/MDR), l'operazione di lettura ha inizio al termine del ciclo del datapath, dopo aver caricato in MAR l'indirizzo. I dati quindi sono disponibili al termine del ciclo seguente in MDR, e quindi possono essere utilizzati solo due cicli dopo. In altre parole, una lettura che ha inizio al ciclo k , fornisce dati alla fine del ciclo $k + 1$ e quindi potranno essere utilizzati solo al ciclo $k + 2$, come descritto in Figura 9.4. Nel ciclo $k + 1$, la CPU non deve necessariamente rimanere inattiva aspet-

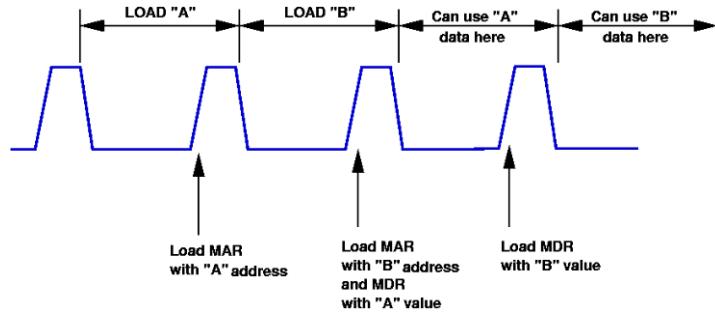


Figura 9.4: Disponibilità dei dati nei registri di interfaccia con la memoria

tando la memoria, ma può eseguire un ciclo di datapath che non necessita del dato in corso di lettura. Per il fetch delle istruzioni le cose sono diverse, il ritardo sarà pari ad un sol ciclo.

Come rappresentato in figura, per controllare il datapath del MIC-1 occorrono **29 segnali**. Essi sono raggruppati nelle seguenti cinque tipologie:

- **9 segnali** per effettuare la *selezione del bus C*;
- **9 segnali** per effettuare la *selezione del bus B*;
- **8 segnali** per la *gestione dell'ALU*:
 - **6** per l'**ALU**;
 - **2** per lo **Shift Register** a valle dell'ALU;
- **2 segnali** di read/write per specificare se si vuole *leggere o scrivere dalla memoria* tramite la coppia **MAR/MDR**;
- **1 segnale** per indicare il *memory fetch* del valore presente in **PC/MBR**.

I valori di tali segnali determinano le operazioni da eseguire durante un ciclo del processore. Notiamo che i registri presenti sono 10, eppure servono 9 segnali per controllare i bus B e C. Ciò è dovuto al fatto che non tutte le combinazioni sono effettivamente significative.

9.2.5 Unità ALU

L'ALU ha 6 ingressi, quindi è in grado di effettuare 2^6 operazioni differenti. Per pilotare l'ALU ho un segnale di **shifter control** costituito da due bit:

- **SLL8** - *Shift Left Logical*, sposta il contenuto dell'ALU a sinistra di un byte, aggiungendo zeri agli 8 bit di ordine inferiore.
- **SRA1** - *Shift Right Arithmetic*, sposta il contenuto dell'ALU di un bit a destra, lasciando però invariato il byte più significativo (il bit di segno).

La definizione dell'operazioni che viene effettuata dall'ALU è fatta tramite gli altri 6 segnali:

- **F0 e F1** - selezionano la funzione ALU: AND, OR, NOT o ADD
- **INVA** - complemento dell'ingresso A (complemento a uno)
- **ENA** - abilita l'ingresso A (altrimenti zero)
- **ENB** - abilita l'ingresso B (altrimenti zero)
- **INC** - bit di riporto di ordine inferiore (consente di aggiungere 1)

L'architettura dell'ALU viene riportata in Figura 9.5.

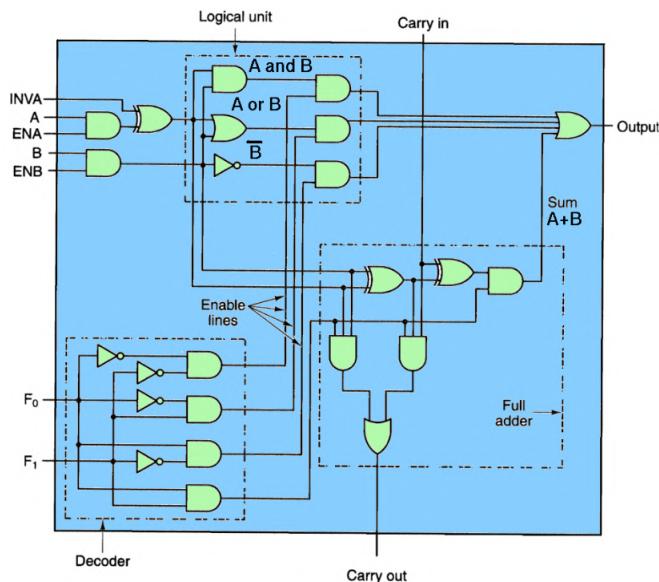


Figura 9.5: Architettura interna dell'ALU

Per ogni combinazione dei segnali di gestione dell'ALU, viene definita una determinata operazione

che l'ALU effettua.

Le differenti combinazioni di bit sono riportate in Tabella 9.1.

| F0 | F1 | ENA | ENB | INVA | INC | Function |
|----|----|-----|-----|------|-----|------------------|
| 0 | 1 | 1 | 0 | 0 | 0 | A |
| 0 | 1 | 0 | 1 | 0 | 0 | B |
| 0 | 1 | 1 | 0 | 1 | 0 | A' |
| 1 | 0 | 0 | 1 | 0 | 0 | B' |
| 1 | 1 | 1 | 1 | 0 | 0 | A + B |
| 1 | 1 | 1 | 1 | 0 | 1 | A + B + 1 |
| 1 | 1 | 1 | 0 | 0 | 1 | A + 1 |
| 1 | 1 | 0 | 1 | 0 | 1 | B + 1 |
| 1 | 1 | 1 | 1 | 1 | 1 | B - A |
| 1 | 1 | 0 | 1 | 1 | 0 | B - 1 |
| 1 | 1 | 1 | 0 | 1 | 1 | -A |
| 0 | 0 | 1 | 1 | 0 | 0 | A AND B |
| 0 | 1 | 1 | 1 | 0 | 0 | A OR B |
| 0 | 1 | 0 | 0 | 0 | 0 | O |
| 1 | 1 | 0 | 0 | 0 | 1 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 | -1 |

Tabella 9.1: Tabella di codifica delle operazioni dell'ALU

9.2.6 Unità di controllo

Come già descritto nei capitoli precedenti, una volta progettata l'unità operativa, esistono due modi per realizzare l'unità di controllo:

- **Logica cablata**, dove viene progettata l'unità di controllo in termini di porte logiche;
- **Logica micro-programmata**, dove l'unità di controllo si costituisce di una micro-ROM e di una logica che la controlli. In tale ROM sono mantenute le microistruzioni necessarie allo svolgimento di ciascun codice operativo, dunque i segnali necessari all'unità operativa per ciascuna operazione.

Dal punto di vista dell'unità operativa è indifferente, in entrambi i casi l'unità di controllo provvederà a generare i segnali di abilitazione verso di essa, ciò che cambia è solo come ciò accade.

Nel caso di logica cablata avremo una macchina sequenziale o una macchina combinatoria, realizzata tramite porte logiche, che a fronte di determinati input (o stati nel caso di macchina sequenziale) produrrà in output i segnali di abilitazione, calcolandoli istante per istante. Tutto ciò che accade è definito dall'architettura dell'unità di controllo.

Nel caso di logica micro-programmata, invece, l'unità di controllo andrà a prelevare la sequenza dei segnali di abilitazione memorizzati nella **micro-ROM** in corrispondenza del codice operativo

da eseguire. Tutto ciò che accade è, quindi, contenuto nella micro-ROM.

Ciò significa che laddove fosse necessario effettuare alcune modifiche alla logica di controllo, nel caso cablato bisognerebbe riprogettare l'unità da zero, nel caso micropogrammato invece sarà sufficiente andare a modificare le sequenze memorizzate nella ROM, dunque, il modello risulta essere maggiormente versatile.

L'unità di controllo del processore MIC-1 è realizzata in logica micro-programmata, ed è rappresentata in Figura 9.6. Nell'unità di controllo, come raffigurato, è quindi presente:

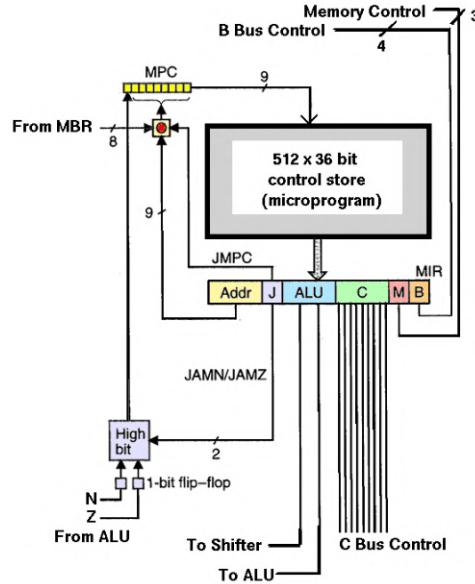


Figura 9.6: Unità di controllo nel processore MIC-1

- un **generatore degli indirizzi** di partenza, ovvero l'indirizzo della micro-ROM a cui si accede per eseguire una data operazione;
- la **memoria di controllo**, ovvero la micro-ROM stessa;
- un **Program Counter** della micro-ROM (**MPC**).

L'unità di controllo deve generare una sequenza dei segnali di controllo necessari a pilotare l'unità operativa, ad ogni colpo di clock. Nel caso del processore MIC-1 sono necessari 29 segnali. Tuttavia, questi bit servono a determinare il percorso dei dati per un singolo colpo di clock, viene dunque aggiunta una parte necessaria a determinare cosa effettuare al ciclo successivo:

- il campo **Addr**, che si compone di **9 bit** ed è necessario ad *individuare la microistruzione da eseguire* al successivo colpo di clock;
- il campo **JAM**, che si compone di **3 bit** che serve a gestire le *operazioni di salto*.

La struttura della Control Word risultante è quella rappresentata in Figura 9.7. Notiamo che non sarà a 41 bit ma a 36 bit. Ciò è dovuto al fatto che il segnale di controllo relativo al bus B viene

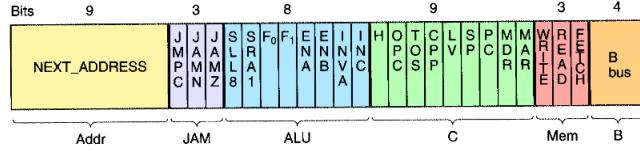


Figura 9.7: Struttura di una Control Word della micro-ROM nel MIC-1

codificato, dunque espresso su 4 bit anziché su 9, come mostrato in Figura 9.8. Tale codifica implica

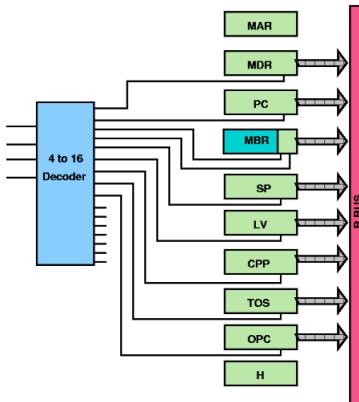


Figura 9.8: Struttura di codifica dei bit di attivazione del bus B

l'aggiunta di un decoder necessario a far arrivare l'informazione su 9 bit al bus B, col vantaggio però di non dover far viaggiare 5 fili aggiuntivi in tutta la memoria ma solo a partire da un certo punto in poi. Le attivazioni dei registri a seguito di tale decoder vengono descritte in Tabella 9.2.

| Binary Value | Register Enabled |
|--------------|------------------|
| 0 | MDR |
| 1 | PC |
| 2 | MBR |
| 3 | MBRU |
| 4 | SP |
| 5 | LV |
| 6 | CPP |
| 7 | TOS |
| 8 | OPC |
| 9 - 15 | none |

Tabella 9.2: Tabella di codifica dei segnali di abilitazione del bus B

Il motivo per cui si sceglie di codificare l'informazione di B e non di C è che il primo bus rappresenta un MUX, i cui segnali di selezione devono essere mutuamente esclusivi, dunque sono codificabili, di contro il secondo rappresenta un DEMUX, i cui segnali non sono necessariamente esclusivi e ciò comporterebbe di dover individuare i gruppi di bit che lo sono, codificarli e poi decodificarli con un hardware maggiormente complesso.

Il campo **Next-Address** indica l'indirizzo della prossima istruzione, ovvero il prossimo codice operativo, il quale è codificato con l'indirizzo della microistruzione della micro-ROM da cui partire, le successive micro-istruzioni saranno calcolate a partire da questa.

Anche la memoria di controllo prevede la presenza di un registro **MA** (*Memory Address*) e di un **MB** (*Memory Buffer*) per il prelievo delle istruzioni. Anch'essa necessita di un PC, chiamato **MPC** (*Micro Program Counter*), invece nel registro **MIR** (*Micro Instruction Register*) è memorizzata la microistruzione in esecuzione, i cui bit determinano i segnali di controllo che guidano il datapath. Se le operazioni che dobbiamo effettuare sono in sequenza, il campo Next-Address indicherà la successiva istruzione da eseguire. Dunque, tale campo fornisce lo stato prossimo, grazie al quale non è necessaria la presenza di un contatore, per incrementare di volta in volta il PC. Quando invece è necessario effettuare un salto, il valore del Next-Address viene opportunamente modificato, prima di essere inserito nel PC.

Per fare ciò si utilizzano i bit del campo JAM e i bit N e Z provenienti dall'ALU e vengono salvati in due flip-flop per non avere problemi di instabilità. A seconda di tali valori, nel registro MPC copieremo il Next-Address, diversamente alterato:

- Se il campo JAM vale 000, allora il Next-Address viene copiato senza modifiche sull'MPC;
- Se JAMN è alto, se ne calcola l'OR con il flag N e si pone il risultato nel bit più significativo di MPC;
- Se JAMZ è alto, se ne calcola l'OR con il flag Z e si pone il risultato nel bit più significativo di MPC;
- Se JAMN e JAMZ sono entrambi alti, si calcola l'OR rispetto a entrambi i flag;
- Se è alto il bit JMPC si effettua l'OR tra gli 8 bit di MBR e gli 8 bit meno significativi di Next-Address ed il risultato viene posto in MPC.

Tramite queste tecniche, è possibile realizzare eventuali diramazioni, necessarie in caso di salti.

Per comprendere meglio tali configurazioni, in Figura 9.9 è presente il circuito e la mappa di Karnaugh per identificare che cosa fare all'interno del campo Next-Address con i bit JAMZ e JAMN, mentre in Figura 9.10 viene rappresentato il funzionamento del circuito analizzando il bit JMPC.

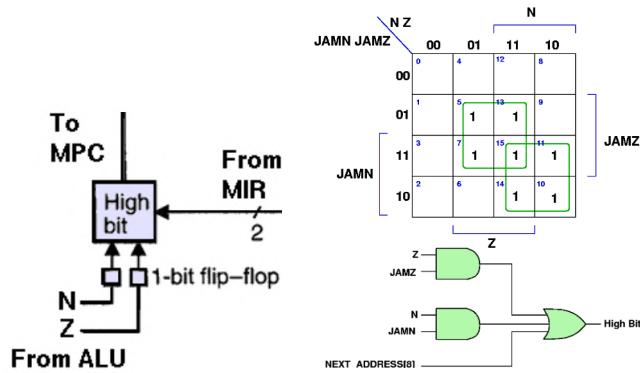


Figura 9.9: Circuito e mappa di Karnaugh per la definizione di High-Bit

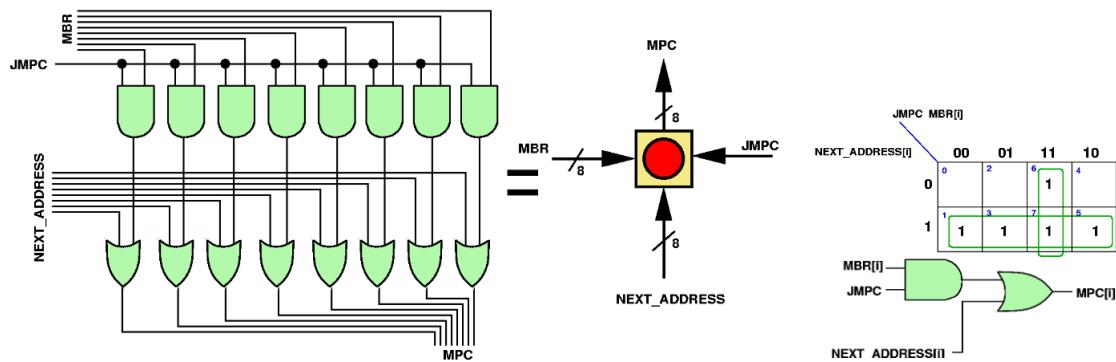


Figura 9.10: Circuito di funzionamento di JMPC

9.2.7 Architettura completa

Dopo aver descritto componente per componente, in Figura 9.11 viene mostrata l'architettura nel suo complesso, unendo opportunamente i segnali descritti ai passi precedenti.

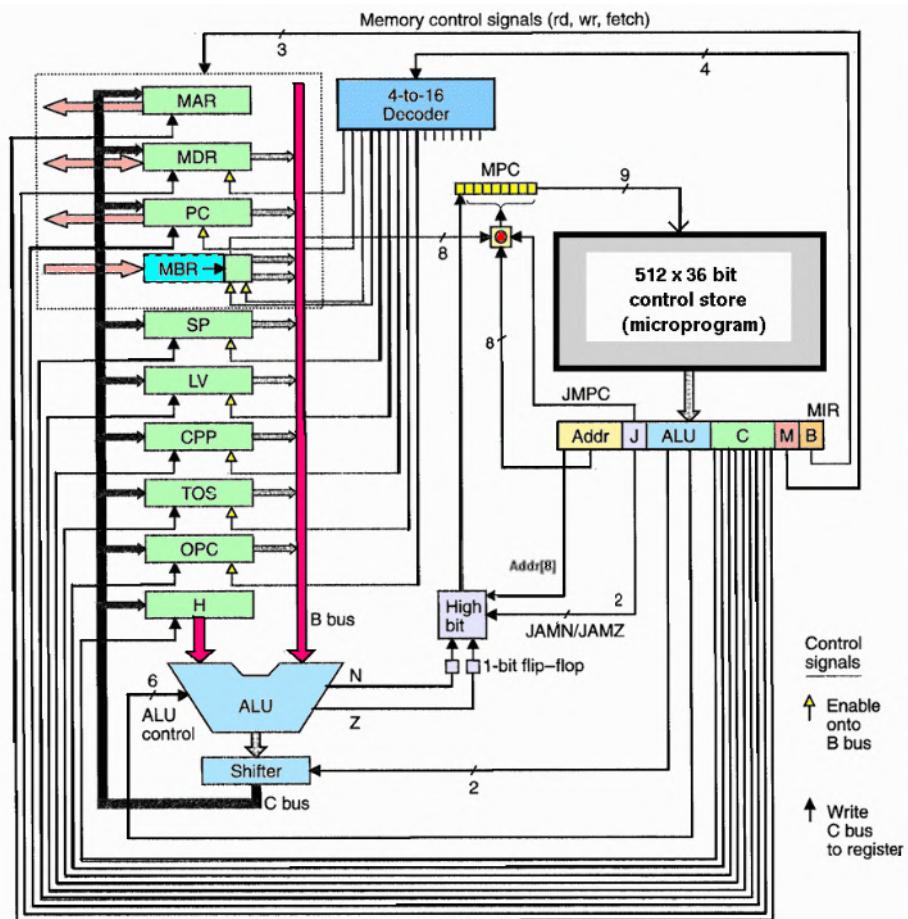


Figura 9.11: Architettura generale del processore MIC-1

9.2.8 IJVM

IJVM è un **linguaggio assembly** creato da Andrew Stuart Tanenbaum implementato sulla micro-architettura MIC-1. IJVM utilizza la memoria in un modo abbastanza particolare e fa largo uso dello **stack**. Per questo motivo, il processore di Tanenbaum è avvantaggiato nella gestione del sistema delle funzioni, dato che lo scambio dei parametri già avviene nello stack, che è una struttura dinamica e non statica. Quindi la gestione delle variabili locali alle procedure risulta essere più semplice.

Lo **stack** è una parte della memoria dove i dati vengono *impilati* e possono essere inseriti o prelevati solo dalla relativa cima:

- Una scrittura (*push*) sullo stack causa la *crescita* in altezza della pila di dati;
- Una lettura (*pop*) dallo stack causa un *accorciamento*.

In IJVM lo stack è utilizzato per memorizzare variabili locali e per eseguire calcoli aritmetici. Vengono mantenuti due puntatori a indirizzi di memoria al suo interno:

- Un puntatore alla base attuale dello stack – **LV** (*Local Variable*);
- Un puntatore alla cima dello stack – **SP** (*Stack Pointer*).

Un esempio di gestione dello stack è riportato in Figura 9.12. Nell'esempio:

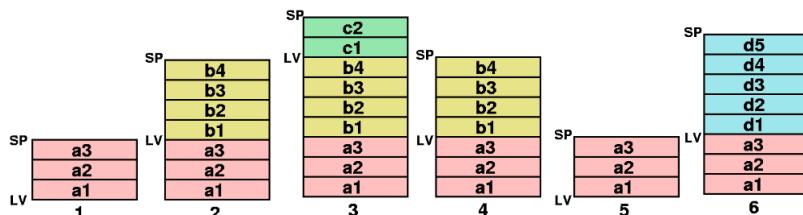


Figura 9.12: Esempio di gestione dello stack nel MIC-1

1. metodo '**a**' chiamato, **3** variabili locali;
2. '**a**' chiama '**b**' che ha **4** variabili locali;
3. '**b**' chiama '**c**' che ha **2** variabili locali;
4. '**c**' termina;
5. '**b**' termina;
6. '**a**' chiama '**d**' che ha **5** variabili locali.

Nel processore di Tanenbaum la gestione delle variabili locali alle procedure è già memorizzata all'interno dello stack. Oltre a quest'ultimo, dove IJVM memorizza le variabili locali ed esegue operazioni aritmetiche, IJVM utilizza altre due regioni di memoria:

- **Constant Pool**, dove vengono memorizzate le *costanti*. La base di tale area è puntata dal registro **CPP**;
- **Method Area**, dove è memorizzato il *codice* e quindi i programmi IJVM. I programmi JAVA prendono il nome di **metodi** ed il puntatore all'istruzione corrente nella method area è **PC**.

In Tabella 9.3 la lista di codici operativi e istruzioni IJVM.

| Codice op (Hex) | Assembly Language Mnemonic | Operandi | Descrizione |
|--------------------|----------------------------------|----------------------------|---|
| 0x10 | BIPUSH | byte | Push di un byte sullo stack |
| 0x59 | DUP | N/A | Copia dello top word sullo stack e push sullo stack |
| 0xA7 | GOTO | etichetta | Salto incondizionato |
| 0x60 | IADD | N/A | Pop di due word dallo stack; push della loro somma |
| 0x7E | IAND | N/A | Pop di due word dallo stack; push della loro AND |
| 0x99 | IFEQ | etichetta | Pop di una word dallo stack e branch se è zero |
| 0x9B | IFLT | etichetta | Pop di una word dallo stack e branch se è minore di zero |
| 0x9F | IF_ICMPEQ | etichetta | Pop di due word dallo stack e branch se sono uguali |
| 0x84 | IINC | nome della variabile, byte | Add di un valore costante ad una variabile locale |
| 0x15 | ILOAD | nome della variabile | Push di una variabile locale sullo stack |
| 0xB6 | INVOKEVIRTUAL | nome del metodo | Chiamata di un metodo |
| 0x80 | IOR | N/A | Pop di due word dallo stack; push della loro OR |
| 0xAC | IRETURN | N/A | Return da un metodo con valore intero |
| 0x36 | ISTORE | nome della variabile | Pop di una word dallo stack e store in una variabile locale |
| 0x64 | ISUB | N/A | Pop di due word dallo stack; push della loro differenza |
| 0x13 | LDC_W | nome della costante | Push di una costante dal constant pool sullo stack |
| 0x00 | NOP | N/A | Nessuna operazione |
| 0x57 | POP | N/A | Delete di una word dalla cima dello stack |
| 0x5F | SWAP | N/A | Swap delle due top word sullo stack |
| 0xC4 | WIDE | N/A | Istruzione prefisso; l'istruzione successiva ha un indice di 16 bit |
| N/A | ERR | N/A | Print un messaggio di errore e blocca il simulatore |
| N/A | HALT | N/A | Blocco del simulatore |
| N/A | IN | N/A | Lettura di un character dal buffer di tastiera e push sullo stack. Se nessun carattere è disponibile, push di uno 0 |
| N/A | OUT | N/A | Pop di una word non nello stack e stampa con standard out |

Tabella 9.3: Codici operativi e istruzioni in IJVM

Fisicamente la memoria è una sola ed il considerarla divisa in 3 regioni è soltanto una semplificazione. Avrò infatti tre puntatori CPP, LV, PC ed a seconda di dove devo prelevare gli elementi avrò un codice operativo differente, con il quale individuerò nella ROM una micro-sequenza con la quale effettuerò delle sequenze opportune.

Per rendere più chiaro il funzionamento dello stack, si è scelto di utilizzare come esempio l'operazione $a_1 = a_2 + a_3$. Le operazioni che vengono effettuate sono le seguenti:

- Push di a_2 sullo stack;
- Push di a_3 sullo stack;
- Rimozione degli operandi dallo stack, somma degli operandi e push del risultato sullo stack;
- Pop dello stack e inserimento del risultato in a_1 .

In Figura 9.13 le operazioni appena descritte.

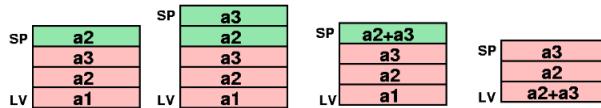


Figura 9.13: Esempio di utilizzo dello stack con l'operazione di somma

Traduzione di un istruzione Java

Prendendo come esempio lo snippet di codice descritto di seguito:

```
i = j + k ;
if ( i == 3 ){
    k = 0 ;
}
else{
    j = j - 1 ;
}
```

tale codice in linguaggio IJVM viene codificato come segue:

| --LINGUAGGIO IJVM | CODICE OPERATIVO |
|-------------------|------------------|
| ILOAD j | 0x15 0x02 |
| ILOAD k | 0x15 0x03 |
| IADD | 0x60 |
| ISTORE i | 0x36 0x01 |
| ILOAD i | 0x15 0x01 |
| BIPUSH 3 | 0x10 0x03 |
| IF_ICMPEQ L1 | 0x9F 0x00 0x0D |
| ILOAD j | 0x15 0x02 |
| BIPUSH 1 | 0x10 0x01 |
| ISUB | 0x64 |
| ISTORE j | 0x36 0x02 |
| GOTO L2 | 0xA7 0x00 0x07 |
| L1: BIPUSH 0 | 0x10 0x00 |
| ISTORE k | 0x36 0x03 |
| L2: | |

9.3 Analisi del Codice

Si è scelto quindi di passare in rassegna il codice implementativo del processore MIC-1.

9.3.1 datapath.vhd

```
entity datapath is
  port (
    clk           : in  std_logic; --! Clock
    reset         : in  std_logic; --! Synchronous active-high reset
    alu_control   : in  alu_ctrl_type; --! Control signals for the ALU
    c_to_reg_control : in  c_ctrl_type; --! Control signals for the C bus
    mem_control   : in  mem_ctrl_type; --! Control signals for memory operations
    reg_to_b_control : in  b_ctrl_type; --! Control signals for the B bus
    mbr_reg_out   : out mbr_data_type; --! Content of the MBR register
    alu_n_flag    : out std_logic; --! ALU negative flag
    alu_z_flag    : out std_logic; --! ALU zero flag
    mem_data_we   : out std_logic; --! Memory data write enable
    mem_data_in   : in  reg_data_type; --! Port for memory data read
    mem_data_out  : out reg_data_type; --! Port for memory data write
    mem_data_addr : out reg_data_type; --! Memory address for memory data operations
    mem_instr_in  : in  mbr_data_type; --! Port for memory instruction read
    mem_instr_addr : out reg_data_type --! Memory address for memory instruction fetch
  );
end entity datapath;
```

Da `alu_control` fino ad `alu_z_flag` vi sono i segnali che interfacciano l'unità operativa (i segnali in input) con l'unità di controllo (i segnali in output). I successivi segnali si interfacciano con la memoria: vi è un `write_enable`, bus di indirizzo e un bus dati. I tipi `reg_data_type` e `bus_data_type` sono degli **STD_LOGIC_VECTOR** con una dimensione comune (in genere pari a 32 bit).

All'interno del file di descrizione del datapath vengono anche dichiarati i segnali relativi ai bus, che sono dei semplici vettori, come definito nella sezione di codice del file **common_defs.vhd** riportata di seguito.

```
--! C bus control width
constant c_ctrl_width : positive := 9;
[...]
--! C bus control type
subtype c_ctrl_type is std_logic_vector(c_ctrl_width - 1
                                         ,! downto 0);
[...]
--! C control MAR bit
constant c_ctrl_mar : natural := 0;
--! C control MDR bit
constant c_ctrl_mdr : natural := 1;
```

```
--! C control PC bit
constant c_ctrl_pc : natural := 2;
--! C control SP bit
constant c_ctrl_sp : natural := 3;
[...]
```

L'elemento critico di un BUS è l'accesso ed esistono diversi modi per farlo. La descrizione del registro da controllare tramite bus è controllata tramite un costrutto **with-select**. Tutte le parti sono elaborazioni in evoluzione concorrente. Il bus C è pilotato sempre e solo dallo shifter. Quindi più sorgenti non accederanno ad esso in contemporanea. Il bus C è un signal connesso da un lato allo shifter e dall'altro in ingresso agli altri registri. Di seguito mostrato il codice di ciò che è stato appena descritto.

```
signal a_bus : reg_data_type;
signal b_bus : reg_data_type;
signal c_bus : reg_data_type;
[...]
with reg_to_b_control select b_bus <=
  mdr_reg      when b_ctrl_mdr,
  pc_reg       when b_ctrl_pc,
  mbr_s        when b_ctrl_mbr,
  mbr_u        when b_ctrl_mbru,
  sp_reg       when b_ctrl_sp,
  lv_reg       when b_ctrl_lv,
  cpp_reg      when b_ctrl_cpp,
  tos_reg      when b_ctrl_tos,
  opc_reg      when b_ctrl_opc,
  (others => '0') when others;
```

I registri, come facilmente si nota nel codice riportato, sono inseriti tramite un solo process, con una certa logica di reset. Sul fronte di clock se il reset è alto resetta tutti i registri, dopodichè c'è una logica di scrittura che avviene tramite il bus B. Questa logica presente nel ramo else del process ci suggerisce che “*se è alto il valore di controllo del bus C e se è alto quello relativo al registro MAR allora scrivi nel registro MAR il contenuto del BUS C. Questa logica si ripete anche per gli altri registri*”.

Il BUS C può ad ogni ciclo scrivere un qualsiasi numero di registri del datapath, anche tutti. Vi sono però delle convenzioni da rispettare sui registri. Sono pochi quelli che hanno uno scopo ben specifico, ovvero quelli che si interfacciano con la memoria ed il registro **H**. Tutti gli altri registri sono equivalenti ed hanno dei nomi perché convenzionalmente vengono usati all'interno del microprogramma sempre per lo stesso scopo, ma a livello di implementazione non differiscono l'un con l'altro.

Prossimo step è analizzare il protocollo con la memoria.

Esso richiede due registri **MAR** e **MDR**. Quando occorre scrivere bisogna alzare il segnale **WE** (*write enable*), che è pilotato dall'unità di controllo. Esso deve essere alto allo stesso fronte di

salita del clock in cui sono disponibili sia il dato da scrivere nel registro MDR sia l'indirizzo in cui scriverlo nell'indirizzo MAR. Viceversa, quando si opera in lettura dobbiamo porre l'indirizzo da cui dobbiamo andare a leggere nell'indirizzo MAR e al fronte di clock successivo il dato sarà disponibile nel registro MDR. Questo comportamento è insolito perché ha poco latenza nella lettura, a differenza di altri processori.

Il protocollo è sincrono di tipo **CACHE**. In lettura dobbiamo porre l'indirizzo da cui dobbiamo leggere nel registro MAR e al fronte di clock successivo sarà disponibile il dato nel MDR. Anche i registri della memoria sono dei registri del datapath, quindi implementeranno una stessa logica, con l'unica differenza che il registro MDR può essere pilotato non solo dal bus C ma anche dalla memoria. Per tale motivo si utilizza un flip flop che porta un valore di “*read*” che risulta essere alto se è stato alto al ciclo precedente il medesimo segnale proveniente dall'unità di controllo.

La prima condizione ci dice che se non c'è una lettura in corso scrivi i registri data con il bus C, se c'è una lettura in corso scrivilo con il dato che viene dalla memoria.

L'interfaccia che viene utilizzata per il fetch delle istruzioni è praticamente identica anche se il registro equivalente al registro di indirizzo è chiamato *Program Counter* ed il registro dati è chiamato *Memory Byte Register*.

L'unica differenza è che **MBR** rappresenta il dato su **8 bit**. Ci sarà un segnale di *fetch* che è equivalente a quello di read dell'interfaccia con la memoria dati, ma non vi è un segnale analogo per la scrittura. La memoria è unica anche se vi è un'interfaccia dati e istruzioni, come lo spazio di indirizzamento. L'unica peculiarità è avere due porte una di scrittura e una di lettura che viene definita **“dual port memory”**. Affinché l'unità di controllo possa pilotare l'unità operativa abbiamo bisogno di segnali di controllo. Per questo motivo, abbiamo bisogno di un certo numero di segnali che ci indicano se ad ogni colpo di clock quello che si trova sul BUS C deve essere scritto o meno su un registro. Per i 9 segnali del BUS C l'architettura è orizzontale mentre per i 4 segnali del BUS B l'architettura è verticale. Se i 9 segnali sono tutti 0 non scrivo su alcun registro, di contro, se son tutti 1 scrivo su tutti i registri. Il BUS B invece può esser controllato da una sola sorgente. Quindi, avrò bisogno di nove possibili registri.

```
-- Registers
signal sp_reg    : reg_data_type;
signal lv_reg    : reg_data_type;
signal cpp_reg   : reg_data_type;
signal tos_reg   : reg_data_type;
signal opc_reg   : reg_data_type;
signal h_reg     : reg_data_type;
signal mar_reg   : reg_data_type;
signal mdr_reg   : reg_data_type;
signal pc_reg    : reg_data_type;
signal mbr_reg   : mbr_data_type;
signal rd_ff     : std_logic;
signal fetch_ff  : std_logic;
```

```

signal wr_ff      : std_logic;
[...]
-- Processor registers
reg_proc : process(clk) is
begin
  if rising_edge(clk) then
    if reset = '1' then
      sp_reg  <= x"00000101";
      lv_reg  <= x"00000100";
      cpp_reg <= x"00000080";
      tos_reg <= (others => '0');
      opc_reg <= (others => '0');
      h_reg   <= (others => '0');
      mar_reg <= (others => '0');
      mdr_reg <= (others => '0');
      pc_reg  <= (others => '0');
      mbr_reg <= (others => '0');

      rd_ff    <= '0';
      fetch_ff <= '0';
      wr_ff    <= '0';
    else
      if c_to_reg_control(c_ctrl_mar) = '1' then
        mar_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_pc) = '1' then
        pc_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_sp) = '1' then
        sp_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_lv) = '1' then
        lv_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_tos) = '1' then
        tos_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_cpp) = '1' then
        cpp_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_h) = '1' then
        h_reg <= c_bus;
      end if;
      if c_to_reg_control(c_ctrl_opc) = '1' then
        opc_reg <= c_bus;
      end if;
    end if;
  end if;
end process;

```

```

-- MDR can also receive data from memory
if c_to_reg_control(c_ctrl_mdr) = '1' then
    mdr_reg <= c_bus;
elsif rd_ff = '1' then
    mdr_reg <= mem_data_in;
end if;

-- MBR can't be written from C bus
if fetch_ff = '1' then
    mbr_reg <= mem_instr_in;
end if;

-- Effects on regs on next clock cycle
rd_ff     <= mem_control(mem_ctrl_read);
fetch_ff  <= mem_control(mem_ctrl_fetch);
wr_ff     <= mem_control(mem_ctrl_write);
end if;
end if;
end process reg_proc;
[...]
-- Output
mbr_reg_out     <= mbr_reg;
mem_data_out    <= mdr_reg;
mem_data_addr   <= mar_reg;
mem_data_we     <= wr_ff;
mem_instr_addr <= pc_reg;

```

9.3.2 alu.vhd

```

entity alu is
port (
    --! ALU control
    control : in alu_ctrl_type;
    --! ALU operand A
    operand_a : in reg_data_type;
    --! ALU operand B
    operand_b : in reg_data_type;
    --! ALU result
    sh_result : out reg_data_type;
    --! Negative flag
    negative_flag : out std_logic;
    --! Zero flag
    zero_flag : out std_logic
);
end entity alu;

```

L'**ALU** avrà un process che associa l'uscita direttamente agli operandi, in base ai segnali di con-

trollo in ingresso. L'implementazione utilizzata è di tipo comportamentale. Di seguito il process di funzionamento dell'ALU.

```
-- Inputs
t_operand_a      <= operand_a      when en_a = '1'  else (others => '0');
t_operand_a_inv <= not t_operand_a when inv_a = '1'  else t_operand_a;
t_operand_b      <= operand_b      when en_b = '1'  else (others => '0');
t_inc(0)         <= inc;
t_u_sum          <= unsigned(t_operand_a_inv) + unsigned(t_operand_b) + unsigned(t_inc);

-- ALU function
t_and    <= t_operand_a_inv and t_operand_b when fn = alu_fn_and   else (others => '0');
t_or     <= t_operand_a_inv or t_operand_b  when fn = alu_fn_or    else (others => '0');
t_not_b <= not t_operand_b           when fn = alu_fn_not_b else (others => '0');
t_sum    <= reg_data_type(t_u_sum)       when fn = alu_fn_sum  else (others => '0');

with fn select t_result <=
  t_and  when alu_fn_and,
  t_or   when alu_fn_or,
  t_not_b when alu_fn_not_b,
  t_sum   when others;

-- ALU flags
negative_flag <= t_result(31);
zero_flag      <= '1' when t_result = x"00000000" else '0';

-- Shifter
with sh select sh_result <=
  t_result(23 downto 0) & x"00"      when alu_sh_sll8,
  t_result(31) & t_result(31 downto 1) when alu_sh_srai,
  t_result           when others;
```

9.3.3 control_store.vhd

```
entity control_store is
  port (
    --! Address of the desired word
    address : in ctrl_str_addr_type;
    --! Content of the addressed word
    word    : out ctrl_str_word_type
  );
end entity control_store;
[...]
--! Dataflow architecture for the control store
architecture dataflow of control_store is
  -- Constants
  constant words : ctrl_str_type := (
    --BEGIN_WORDS_ENTRY
```

```

0 => "000000110000000000000000000000001001",
1 => "010111000000000000000000000000001001",
2 => "0000000000000000000000000000000000000000",
3 => "0000000000000000000000000000000000000000",
4 => "0000000000000000000000000000000000000000",
[...]
others => (others => '0')
--END_WORDS_ENTRY
);
begin -- architecture dataflow
word <= words(to_integer(unsigned(address)));
end architecture dataflow;

```

Il **Control Store** presenta un’interfaccia molto semplice: essendo una **ROM** è letteralmente una *constant* che contiene le word che sono nel programma. C’è poi una logica di controllo che riguarda la codifica dei segnali di controllo per il bus B e la logica inerente ai salti.

9.3.4 control_unit.vhd

```

-- MPC virtual register
ctrl_nxt_addr_no_msb <= mir_reg(ctrl_nxt_addr_no_msb_type'range);
jmpc_addr           <= ctrl_nxt_addr_no_msb or mbr_reg_in
when mir_reg(ctrl_jmpc) = '1' else ctrl_nxt_addr_no_msb;
high_bit            <= (alu_n_flag and mir_reg(ctrl_jamn)) or (alu_z_flag and mir_reg(ctrl_jamz));
mpc_virtual_reg <= (mir_reg(ctrl_nxt_addr_msb) or high_bit) & jmpc_addr;

-- B_BUS control decoder
reg_to_b_decoder : process(mir_reg(ctrl_b'range)) is
begin
reg_to_b_decoder_out <= (others => '0');

if unsigned(mir_reg(ctrl_b'range)) < b_ctrl_width then
    reg_to_b_decoder_out(to_integer(unsigned(mir_reg(ctrl_b'range)))) <= '1';
end if;
end process reg_to_b_decoder;

```

L’implementazione della control unit è molto semplice: un process che aggiorna il registro MIR con la micro-istruzione corrente, i cui bit del campo ADDR, opportunamente mascherati, costituiscono il registro virtuale MPC. Il decoder per la selezione del registro che controlla il bus B è implementato come un process puramente combinatorio.

La definizione delle variabili è sempre descritta all’interno della seguente sezione di codice del file **common_defs.vhd**, ed è riportata di seguito.

```

--! Control store address width
constant ctrl_str_addr_width : positive := 9;
--! Control store word width

```

```

constant ctrl_str_word_width : positive := 36;
[...]
--! Control store word
subtype ctrl_str_word_type is std_logic_vector(ctrl_str_word_width - 1 downto 0);
[...]
--! Control store content
type ctrl_str_type is array (ctrl_str_words - 1 downto 0) of ctrl_str_word_type;

```

9.4 Studio di due istruzioni a scelta

9.4.1 Istruzione 1 : IADD

Ad ogni ciclo di istruzioni del processore viene effettuata la seguente operazione:

main:

```

PC = PC + 1;
FETCH;
GOTO (MBR);

```

Avremo un salto non condizionato di tipo **GOTO** a MBR, ovvero all'indirizzo all'interno del *control store* della micro-procedura relativa all'istruzione prelevata.

Il codice **IADD** si troverà alla locazione in valore esadecimale **0X85**. Una volta finita la sequenza, ripartirà con un goto main, corrispondente al prelievo del program counter dell'istruzione successiva. Prelevata l'istruzione, il *program counter* viene incrementato e ci si ritroverà nella fase di *fetch*. La prossima microistruzione è il salto a MBR, i cui primi 8 bit corrispondono alla codifica di IADD.

In linguaggio MAL il codice corrispondente è il seguente:

```

IADD = 0x65:
      MAR = SP = SP - 1; rd
      H = TOS
      MDR = TOS = MDR + H; wr; GOTO main

```

L'istruzione IADD dice al processore di prelevare le ultime due word in cima allo stack, eseguirne la somma trattandole come interi con segno su 32bit ed eseguire la push del risultato sullo stack. La precondizione di IADD sarà quella di aver eseguito il push di due interi sullo stack, mentre la post-condizione è che questi due elementi vengano eliminati dallo stack e sostituiti dalla loro somma. Non occorre dare molta attenzione ai campi ADDR e JAM per l'istruzione in esame, potremo infatti spiegarla riportando i segnali relativi all'unità operativa.

La seguente istruzione

```
MAR = SP = SP - 1; rd
```

rende esplicito il comportamento richiesto all'unità operativa in un ciclo. Si presenta come uno spostamento tra registri ed il tutto avviene in una fase di clock:

1. Nella prima fase il valore di SP viene scritto nel bus B;
2. Nella seconda fase il contenuto del bus B viene decrementato di 1;
3. Nella terza fase il segnale viene caricato nel MAR e si alza il segnale di read. Ci troviamo in una fase di prelievo dell'operando dove il valore decrementato dell'istruzione estratta si trova in MDR;
4. Nel quarto passo tramite il bus C, il valore letto si troverà in H senza aver subito alterazioni;
5. All'ultimo passo il valore somma si troverà in TOS, mentre in valore letto si troverà in MDR e potrà farne la somma. Calcolato il risultato, quest'ultimo non solo sarà salvato in TOS, ma contemporaneamente grazie al segnale WRITE, verrà scritto nel registro MDR con spiazzamento H.

I risultati saranno disponibili dopo un certo numero di cicli dalla fine della microistruzione e la scrittura è completata dopo due cicli rispetto all'ultima operazione effettuata.

I gruppi di bit specificati dalla prima microistruzione saranno:

- I sei bit di controllo **110110** per l'ALU;
- I nove bit di controllo **000001001** per il bus C;
- I tre bit di controllo **010** per le operazioni in memoria;
- I quattro bit di controllo **0100** per i bit codificati del bus B.

9.4.2 Istruzione 2 : BIPUSH

In linguaggio MAL il codice corrispondente all'operazione di **BIPUSH** è il seguente:

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

L'istruzione BIPUSH consente di inserire un byte in testa allo stack. Prima di eseguire questa operazione vi deve essere una fase di predisposizione per l'inserimento di un nuovo elemento in memoria, per cui lo stack pointer viene incrementato e l'indirizzo viene inserito nel registro MAR, al fine di assicurare che il dato sia memorizzato fisicamente.

Successivamente il program counter viene incrementato per leggere l'operando all'interno dell'istruzione. A questo punto, tale operando viene memorizzato all'interno del registro MBR, copiato

nel TOS e nel registro MDR, infine avviene la scrittura di tale byte in memoria, operazione indicata dalla presenza della parola chiave **wr**. Le stringhe di bit dell’istruzione sono:

- I sei bit di controllo **110101** per l’ALU;
- I nove bit di controllo **000001001** per il bus C;
- I tre bit di controllo **000** per le operazioni in memoria;
- I quattro bit di controllo **0100** per i bit codificati del bus B.

In linguaggio Assembly le due istruzioni di BIPUSH e IADD vengono codificate come segue:

```
.main
BIPUSH 0x56
BIPUSH 0x0A
IADD
HALT
.endmethod
```

9.5 Simulazione

9.5.1 IADD

Tramite il tool GTK-WAVE in ambiente Linux viene mostrato in Figura 9.14 il comportamento dell'istruzione IADD. Come si evince, l'andamento è proprio quello descritto in precedenza. In

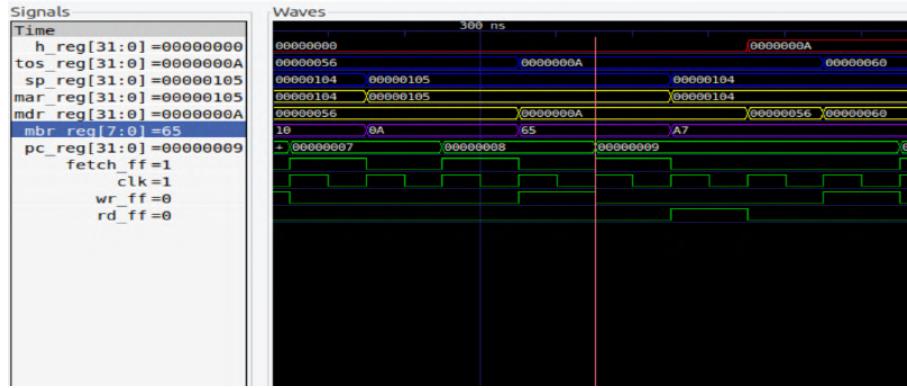


Figura 9.14: Simulazione dell'istruzione IADD tramite GTK-WAVE

corrispondenza della barra rossa avviene il fetch della microistruzione all'indirizzo 0x65 della control store e quindi al prossimo colpo di clock saranno caricate le micro-operazioni della IADD:

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

In questa micro-operazione avviene una lettura dalla memoria necessaria al prelievo del secondo operando dell'addizione. Quest'ultimo si troverà in seconda posizione, sarà quindi necessario decrementare lo stack pointer e passare l'indirizzo al MAR per il prelievo di questo operando dalla memoria. Come presente in figura, il registro **sp_reg** passa dal valore 105 al valore 104 e viene passato al registro **mar_reg**:

- **H=TOS**

In questa micro-operazione il valore contenuto nel registro TOS viene assegnato al registro tampone H al fine di memorizzare il primo operando da sommare. Dopo due colpi di clock il valore **0000000A**, rappresentante il primo operando, presente nel registro **tos_reg**, viene copiato nel registro di holding **h_reg**.

- **MDR = TOS = MDR + H; wr; GOTO main**

Al termine della micro-procedura, viene effettuata la somma dei due operandi presenti rispettivamente in MDR e H. La somma viene inserita in TOS sovrascrivendo il valore precedente. In seguito, quest'ultimo valore viene inserito nel registro MDR per renderlo disponibile all'operazione di scrittura sul bus C. Infine, avviene un salto condizionato al main. Dalla figura è possibile avere un riscontro di quanto appena illustrato verificando che la somma tra **0x0A + 0x56 = 0x60**. Tale valore viene inserito contemporaneamente in **tos_reg** e **mdr_reg**.

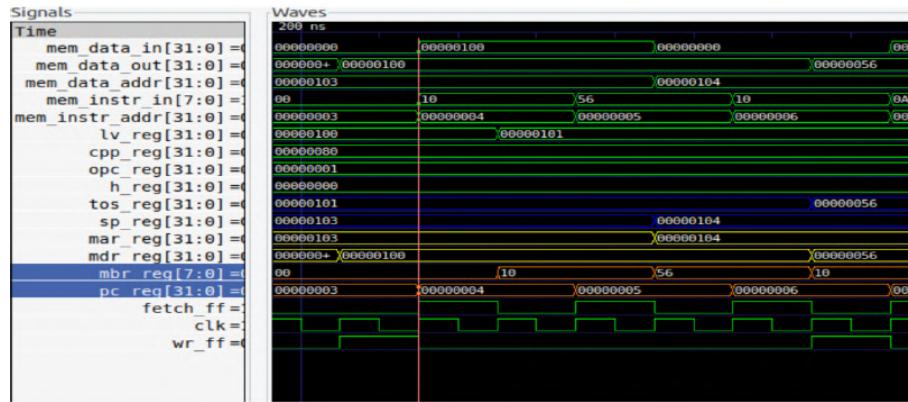


Figura 9.15: Simulazione dell'istruzione BIPUSH tramite GTK-WAVE

9.5.2 BIPUSH

Tramite il tool GTK-WAVE in ambiente Linux viene mostrato in Figura 9.15 il comportamento dell'istruzione BIPUSH. Come si evince, l'andamento è quello precedentemente descritto.

La barra rossa indica, infatti, il prelievo che consente di caricare nel registro MBR il codice operativo che accede alla parola del control store e avvierà la prima microistruzione della micro-procedura relativa a BIPUSH. Affinché l'istruzione venga resa disponibile occorre attendere un ciclo. Nel momento in cui essa diventa disponibile, stiamo ancora eseguendo la micro-istruzione seguente:

```
main:
    PC = PC + 1;
    FETCH;
    GOTO (MBR);
```

Per cui, la microistruzione presente nel registro MBR verrà eseguita nel successivo colpo di clock. Nel frattempo, verrà svolta la fetch e l'incremento del program counter. Al colpo di clock successivo, verranno eseguite le micro-operazioni della BIPUSH:

- **SP = MAR = SP + 1**

Questa micro-istruzione, incrementa lo stack pointer di 1. In seguito, tale indirizzo viene memorizzato nel registro MAR e rappresenterà l'elemento effettivo che conterrà lo stack pointer. Come si può vedere in figura, **sp_reg** passerà dal valore 00000103 a 00000104 e verrà inserito in **mar_reg**.

- **PC = PC + 1 ; FETCH**

Viene effettuato il prelievo dei prossimi 8 bit in memoria, corrispondenti all'operando della micro-istruzione (0x56 nella simulazione riportata in figura) e viene effettuato l'incremento del program counter (**pc_reg** da 00000005 a 00000006).

- **MDR = TOS = MBR; wr ; goto main**

Nell'ultima micro-istruzione vi è il ritorno nello stato di idle, seguito dalla scrittura del valore

aggiunto allo stack in memoria e nei registri TOS e MDR. In figura sarà possibile ritrovare il valore 00000056 all'interno di **mem_data_out**, **tos_reg** e **mdr_reg**.

9.6 Modifica di un istruzione

È possibile modificare il comportamento di una data istruzione sfruttando la traduzione che svolge l'assemblatore tra le istruzioni ad alto livello e quelle micro-architetturali.

Nel nostro caso si è deciso di modificare il codice di un'istruzione di IADD, rendendola una sottrazione.

Il codice dell'istruzione IADD non modificato risulta essere il seguente:

```
IADD = 0x65:
```

```
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR + H; wr; GOTO main
```

Applicando la modifica suddetta, il micro-codice diventa il seguente:

```
IADD = 0x65:
```

```
    MAR = SP = SP - 1; rd
    H = TOS
    MDR = TOS = MDR - H; wr; GOTO main
```

A valle di tale modifica il programma assembly corrispondente a tale istruzione resta inalterato. Ci si aspetta che al termine della micro-procedura venga effettuata la sottrazione tra i due operandi presenti rispettivamente in **mdr_reg** (00000060) e **h_reg** (0000000A). La differenza (00000056) verrà inserita in **tos_reg** sovrascrivendo il valore precedente e nel registro **mdr_reg**.

La simulazione del codice modificato è riportata in Figura 9.16.

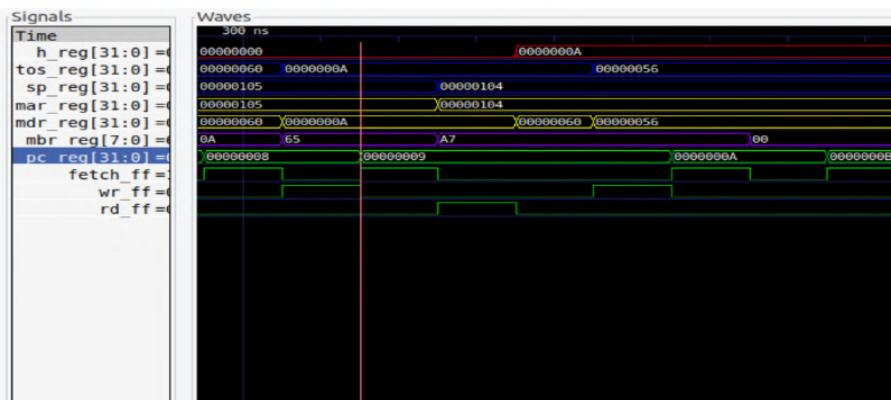


Figura 9.16: Simulazione dell'istruzione IADD modificata tramite GTK-WAVE

Capitolo 10

Esercizio 9

10.1 Traccia

Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare ed implementare in VHDL i seguenti componenti:

- **UART_TAPPO:** il componente acquisisce una stringa di 8 bit (fornita attraverso gli switch della board di sviluppo) e la serializza tramite la sezione di trasmissione del dispositivo UART; l'output seriale della UART viene re-invito in ingresso alla sezione di ricezione dello stesso dispositivo (configurazione a tappo), e il dato deserializzato viene visualizzato sui led della board di sviluppo.
- **2_UART:** il componente acquisisce una stringa di 8 bit (fornita dall'utente tramite gli switch della board di sviluppo), la serializza tramite la sezione di trasmissione di un primo dispositivo UART, la deserializza tramite la sezione di ricezione di un secondo dispositivo UART collegato a valle del primo, e mostra le stringa led della board di sviluppo.
- **UART_PC (*facoltativo*):** il componente realizza la comunicazione fra la board di sviluppo e un terminale seriale in esecuzione su PC (es. Termite), previa connessione di PC e board tramite dispositivo fisico RS232 (uno degli endpoint di comunicazione è rappresentato dal PC). Il componente deve poter acquisire una stringa di 8 bit che rappresenta un carattere in codifica ASCII (fornita attraverso gli switch della board di sviluppo), ed inviarla tramite il dispositivo UART al terminale in esecuzione sul PC, in cui il carattere viene visualizzato. Allo stesso modo, il componente deve essere in grado di ricevere attraverso lo stesso dispositivo UART (oppure una seconda UART) un carattere trasmesso dal terminale e mostrarlo sui led.

10.2 Cenni teorici

L'UART, il cui acronimo sta ad indicare "ricevitore-trasmettitore asincrono universale", è un dispositivo hardware utilizzato per le comunicazioni seriali; esso converte flussi di bit di dati da un formato parallelo a un formato seriale asincrono.

Tale comunicazione è caratterizzata da due entità, il **trasmettitore**, che prende informazioni in parallelo e le trasmette in serie, ed il **ricevitore**, il quale prende le informazioni in serie e le memorizza in parallelo.

Ogni UART contiene un registro a scorrimento, che è il metodo fondamentale di conversione tra forme seriali e parallele.

Esistono tre modalità di trasmissione:

- **Simplex**: la comunicazione è unidirezionale, per cui solo il trasmettitore comunica con il ricevitore;
- **Half-Duplex**: trasmettitore e ricevitore comunicano sullo stesso canale, ma solo uno alla volta può parlare;
- **Full-Duplex**: trasmettitore e ricevitore comunicano sullo stesso canale e possono parlare contemporaneamente.

La board di sviluppo Nexys A7 presenta una periferica UART in modalità Full-Duplex: in questa modalità di trasmissione, avendo la necessità di trasmettere e ricevere contemporaneamente, le strutture dati vengono duplicate.

Il frame dei dati usato per la trasmissione tramite UART viene mostrato in Figura 10.1 . I dispositivi, trasmettitore e ricevitore, devono essere a conoscenza della struttura del pacchetto trasmesso.

Il frame è costituito da 10 bit, un bit per lo **start**, un bit per lo **stop** e i restanti 8 per i bit **dati**.

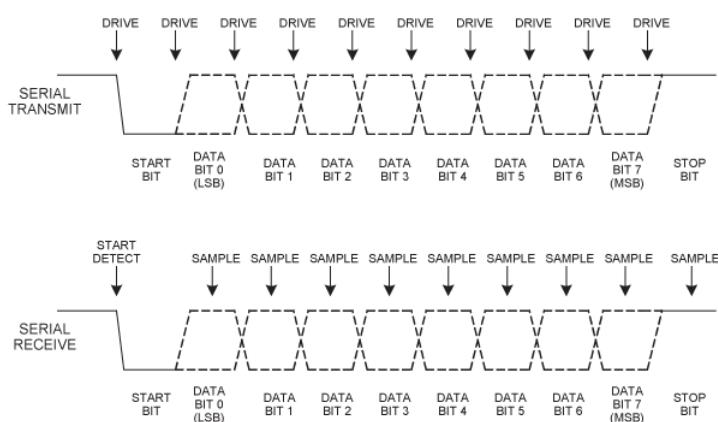


Figura 10.1: Frame dati trasmessi tramite un UART

Lo start bit, con livello logico basso, segnala al ricevitore che sta arrivando un nuovo carattere; lo stop bit, con livello logico alto, segnala al destinatario che il dato è completo.

Grazie al fatto che il bit di start ha un livello logico basso, ed il bit di stop ha un livello logico alto, ci sono sempre almeno due cambi di segnale garantiti tra i caratteri. Lo stato inattivo, ha il livello logico alto, ovvero la linea è tenuta alta per mostrare che la linea stessa e il trasmettitore non sono danneggiati, questo perchè, se lo stato inattivo fosse basso, il ricevitore non saprebbe distinguere se la linea è inattiva perchè il trasmettitore non sta trasmettendo nulla oppure perchè la linea è rotta.

UART risiedenti su dispositivi che comunicano tra loro non hanno un clock condiviso, esse risincronizzano i loro clock interni ad ogni carattere trasmesso.

Per garantirne il corretto funzionamento i dispositivi di trasmissione e ricezione devono accordarsi su:

- baud rate: numero di transizioni che avvengono sulla linea;
- lunghezza dei caratteri;
- numero di bit di parità;
- numero di stop bit.

Il ricevitore lavora con una frequenza multipla (fattore 8 o 16) di quella del trasmettitore per poter sovraccampionare la linea. Se lo start bit dura almeno la metà del periodo di campionamento, esso è valido e segnala l'inizio di un nuovo carattere. Quindi, il ricevitore verifica che lo start bit si abbassi e non appena si abbassa, passa in stato di ricezione, ovvero campiona i primi 8 campioni e si posiziona a centro bit. Da quel momento inizia a campionare ogni 16 per poter rimanere sempre a centro bit. Tenersi a centro bit consente al ricevitore di evitare il framing error, ovvero evitare di deviare il campionamento verso bit che non sono quelli che si aspetta di campionare. Nel caso in cui l'UART ricevente rileva una configurazione errata può impostare un bit di flag noto come errore di framing.

Il trasmettitore è più semplice in quanto non è necessario determinare la temporizzazione dallo stato della linea, ne è vincolata a intervalli di temporizzazione fissi.

Nel momento in cui il trasmettitore deposita un carattere nel registro a scorrimento, l'UART (lato trasmissione):

- genera uno start bit;
- in base al frame dati concordato sposta un certo numero di bit sulla linea (in questo caso 8 bit);
- genera e invia un bit di parità, se previsto (in questo caso no);
- invia i/gli stop bit.

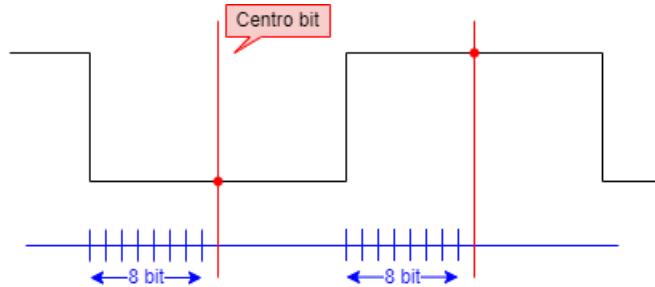


Figura 10.2: Schema di una comunicazione

Quando la comunicazione termina il trasmettitore alza il flag **Text Buffer Empty**, per indicare che il buffer è stato svuotato, mentre il ricevitore alza il flag **Read Data Available**, ciò è la disponibilità in lettura delle informazioni ricevute.

Le soluzioni proposte sono state realizzate attraverso il dispositivo RS232RefComp2.vhd della Digilent. In Figura 10.3 si riporta il componente UART e i segnali di cui esso è dotato. L'UART component

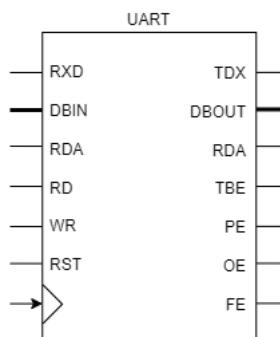


Figura 10.3: Segnali di ingresso e uscita dell'UART

è caratterizzato in ingresso da un segnale di **CLK**, il quale consente la temporizzazione della macchina, un segnale di reset **RST**, che consente alla macchina di ripartire da uno stato noto, un segnale **WR**, per far partire il trasmettitore. Inoltre, si ha un segnale di ingresso **DBIN** ed un segnale di uscita **DBOUT** entrambi ad 8 bit. I segnali **RXD-TXD**, il primo di ingresso e il secondo di uscita, rappresentano il canale di trasmissione. In particolare, **TXD** è inizializzato ad 1 affinché la linea sia sempre in stato inattiva alta, a meno che non si voglia iniziare una trasmissione, questo perchè non si vuole iniziare una ricezione spuria (non prevista). Altri segnali sono:

- *UnderRun Error (TBE)*, indica che il trasmettitore UART ha completato l'invio di un carattere e il buffer di trasmissione è vuoto;
- *Errore di parità (PE)*, si verifica quando la parità dei bit dati non corrisponde col bit di parità;
- *Overrun error (OE)*, indica che il destinatario non è in grado di elaborare il carattere appena arrivato prima che arrivi quello successivo;

- *Framing error (FE)*, si verifica quando la linea di dati non si trova nello stato previsto (alto) quando è previsto il bit di stop (in base al numero di bit di dati e di parità per i quali è impostato l'UART).

Lo UART ha due blocchi funzionali principali che possiamo vedere come due circuiti separati, integrati all'interno di un unico componente: un circuito per la ricezione di informazioni seriali e un circuito per la trasmissione di informazioni seriali. I segnali sono descritti in Figura 10.4.

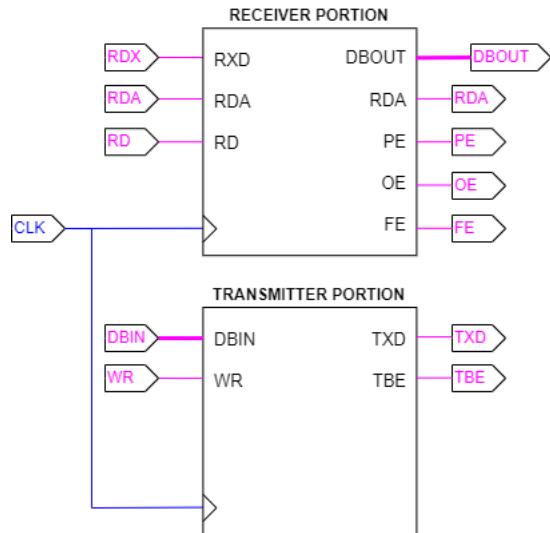


Figura 10.4: Schema di parte di trasmissione e ricezione di un UART

10.2.1 Ricezione

Il ricevitore, tramite la porta RXD di ingresso, riceve un byte di informazioni e lo converte in parallelo, il byte convertito viene mostrato sull'uscita DBOUT. Il trasmettitore, invece, prende un byte di informazioni parallele da DBIN (quando si alza il bit di word) e lo trasmette in serie sulla porta TXD.

Il blocco presentato in Figura 10.5 è costituito da un controller dei dati seriali, due contatori (necessari per la sincronizzazione con la trasmissione visto che i dati trasferiti sulla porta RXD arrivano ad una velocità di trasmissione specifica), il Data Counter conta i bit che stanno arrivando e Counter conta il sistema di frequenza; poi un controller dei bit di errore ed un registro a scorrimento, utilizzato per la memorizzazione dei dati in arrivo al pin di ingresso RXD.

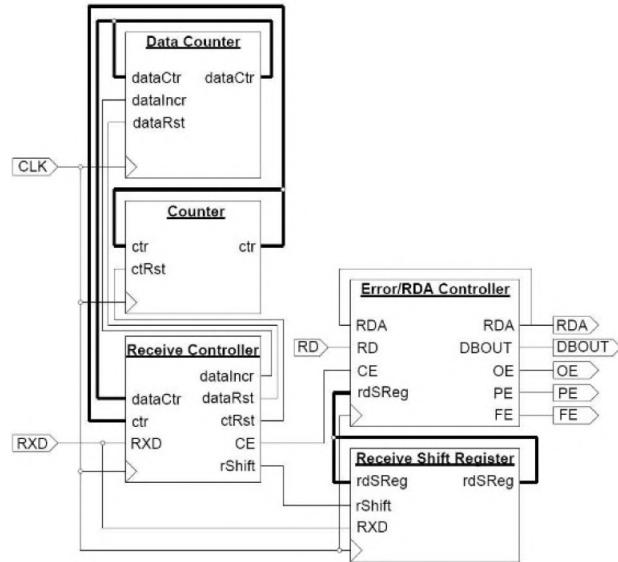


Figura 10.5: Schema di funzionamento della ricezione sull'UART

L'automa che rappresenta la macchina a stati del ricevitore viene mostrata in Figura 10.6.

L'automa si compone dei seguenti stati:

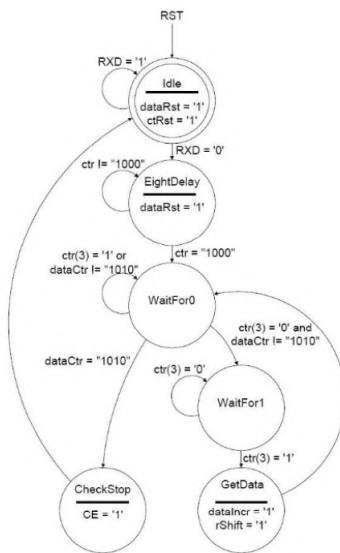


Figura 10.6: Diagramma a stati in ricezione dell'UART

- uno stato **idle**: quando non c'è trasmissione in corso, il trasmittitore non genera lo start bit, quindi il segnale TXD è tenuto alto e di conseguenza il segnale RXD, in ingresso al ricevitore, sarà mantenuto alto. La macchina permane in tale stato fintanto che la porta RXD non si abbassa, cioè fino a quando non inizia la ricezione dello start bit. Quando RXD='0', allora si passa nello stato immediatamente successivo;

- uno stato **EightDelay**: il seguente stato ritarda la macchina per otto cicli, cioè per otto colpi di clock, affinchè il ricevitore si posizioni a centro bit, affinchè il segnale RXD sia letto correttamente. Quindi la macchina permane in questo stato finchè il conteggio non è pari a 8; una volta che il conteggio è arrivato ad 8 avviene la transizione allo stato successivo;
- uno stato **WaitFor0**: è uno stato di ritardo in quanto nel caso in cui non sono stati ancora trasferiti tutti i dati seriali questo stato ha il compito di ritardare il ricevitore. La comunicazione UART prevede la trasmissione di 10 bit, 8 di dato, quindi se il **dataCtr** non è uguale a 10, ovvero 1010 vuol dire che ancora dobbiamo ricevere altri bit. In particolare, se il quarto bit di **ctr** è uguale ad 1, si permane in questo stato. Quando il quarto bit di **ctr** si azzera, vuol dire che è stato raggiunto metà del ritardo (si entra in questo stato quando **ctr** vale "1000", cioè dopo 8 conteggi, per cui **ctr** si azzera dopo altri 8 conteggi, cioè dopo che sono avvenute le comunicazioni 1001-1010-1011-1100-1101-1110-1111-0000) e quindi si transita nel secondo stato di ritardo **Waitfor1**. Se invece il **dataCtr** è pari a 10, vuol dire che tutti i dati necessari sono stati acquisiti, quindi si passa nello stato di **CheckStop**, per verificare la presenza di errori e ripristinare la macchina di ricezione;
- uno stato di **WaitFor1**: questo stato si raggiunge quando il quarto bit di **ctr** è 0. E molto simile allo stato **WaitFor0** eccetto per il fatto che aspetta che il quarto bit di **ctr** sia pari ad 1, ovvero che siano stati raggiunti altri 8 conteggi. Quando **ctr(3)** è alto, allora avviene la transizione allo stato **GetData**, altrimenti continua a rimanere nello stato corrente, in attesa degli 8 cicli di clock;
- uno stato **GetData**: in questo stato, vengono tenuti alti per un colpo di clock i segnali di **rShift** e di **dataIncr**, in modo tale che il registro a scorrimento possa effettuare uno shift e che il numero di bit letti venga incrementato di uno. Questo stato acquisisce i dati in arrivo sul pin **RXD** nel registro a scorrimento **rdSreg**. Si ritorna poi nello stato di **WaitFor0** che avvia i due stati di ritardo (wait for 0 e wait for 1) necessari per l'acquisizione dei dati;
- uno stato **CheckStop**: si giunge in questo stato quando sono stati acquisiti tutti e 10 bit di dati e serve ad avviare il processo che effettua il controllo degli errori, infatti, viene alzato il flag **CE**. Da questo stato il ricevitore transita nuovamente in **idle**.

10.2.2 Trasmissione

Come già detto precedentemente, il trasmettitore trasmette un byte di dati seriali sulla porta TXD: per trasmettere il byte in ingresso dalla porta DBIN, la parte di trasmissione di UART deve contenere un controller di trasferimento, due contatori per la sincronizzazione e un registro di spostamento del trasferimento, come mostrato in Figura 10.7. Il controller di trasferimento utilizza i due contatori di sincronizzazione per controllare la velocità con cui il byte di dati deve essere trasmesso attraverso la porta TXD e il numero di bit trasmessi. Un contatore viene utilizzato per

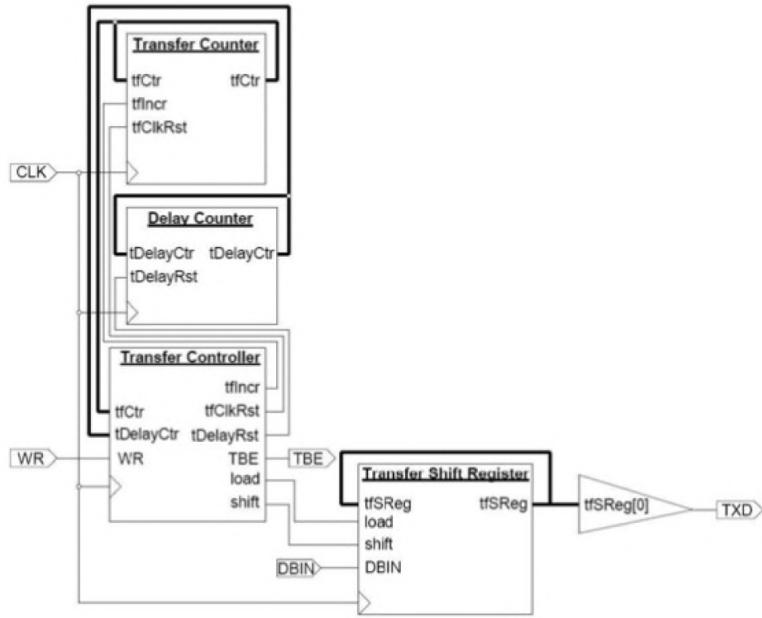


Figura 10.7: Schema di funzionamento della trasmissione sull'UART

ritardare il controller di trasferimento tra le trasmissioni, mentre l'altro contatore viene utilizzato per tenere traccia di quante trasmissioni sono state inviate. La porta TXD è impostata uguale al bit meno significativo del registro a scorrimento, in questo modo la trasmissione dei dati risulta essere semplice, si effettua lo shift a destra di una posizione ad ogni trasmissione. L'automa che rappresenta la macchina a stati del trasmettitore viene mostrata in Figura 10.8.

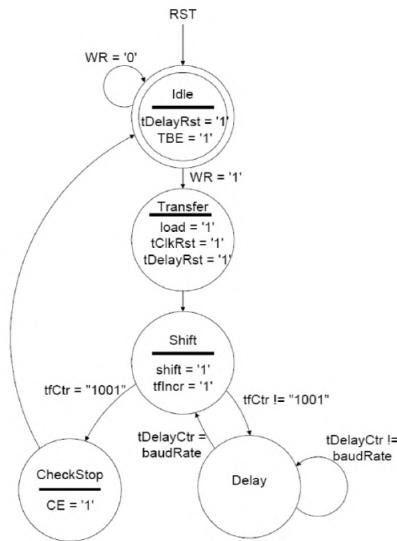


Figura 10.8: Diagramma a stati in trasmissione dell'UART

L'automa si compone dei seguenti stati:

- uno stato **idle**: rappresenta la fase di inattività e di avvio della macchina, la quale permane

in questo stato fino a quando il segnale di **WR** (write) rimane basso. Il seguente stato imposta i segnali **load**, **shift**, **tClkRST** e **tfIncr** bassi; il segnale **TBE** invece è alto per indicare che il buffer di trasmissione non è attualmente in uso. Infine, viene alzato il bit di **tDelayRst**, che rappresenta il segnale di reset del contatore **tDelayCtr**. Quando si alza il segnale di write avviene la transizione al prossimo stato, ovvero stato **Transfer**.

- uno stato **Transfer**: il seguente stato imposta i segnali **load**, **tClkRST** e **tDelayRst** alti, mentre imposta il segnale **TBE** su basso. Il segnale di **load** viene utilizzato per caricare il registro a scorrimento (di trasferimento) con i dati appropriati, mentre i segnali di **tClkRST** e **tDelayRst** vengono utilizzati per resettare **tfCtr** e **tDelayCtr**, ovvero i due contatori di sincronizzazione (conteggio di bit trasferiti e divisore di frequenza). La macchina passa poi nello stato successivo di **Delay**;
- uno stato **Delay**: realizza il ritardo della macchina tra le trasmissioni. Tutti i segnali vengono tenuti bassi: se il **tDelayCtr** è uguale al **baudRate** avviene la transizione allo stato **Shift**, altrimenti il trasmettitore continua a rimanere in questo stato;
- uno stato **Shift**: in questo stato, i segnali di abilitazione **shift** e **tfIncr** vengono alzati, mentre tutti gli altri sono tenuti bassi. Inoltre, viene controllato il **tfCtr**, per vedere quanti bit sono stati trasmessi. Con il segnale di **shift** alto si effettua uno shift a destra di un bit ad ogni colpo di clock, ed ogni volta il **tfCtr**, conteggio di bit trasmessi, viene incrementato di uno grazie al segnale di **tfIncr** abilitato. Fintanto che **tfCtr** non arriva a "1010", cioè a 10, non sono stati trasmessi tutti i bit, quindi la macchina ritorna allo stato di **Delay**. Invece, nel caso in cui **tfCtr** arriva a "1010", allora viene caricato lo stato finale, ossia **WaitWrite**;
- uno stato **WaitWrite**: si occupa di controllare che il segnale di **WR**, inizialmente attivato dalla macchina, sia stato riportato a 0. Senza questo stato, il segnale di scrittura sarebbe tenuto alto per lungo tempo, comportando trasmissioni multiple. Quando il segnale di **write** si abbassa, viene caricato lo stato di **idle** che riporta il trasmettitore allo stato di partenza.

10.3 Soluzione

Per la variante a tappo, il top module è stato implementato con un livello di astrazione Structural in cui è stato dichiarato ed istanziato un unico componente **UART**. Nell'entity sono presenti i seguenti segnali: il **clock**, che gestisce la temporizzazione della macchina, il segnale **dato_in**, che prende in ingresso un byte di informazioni parallele, un segnale **DBOUT**, il quale fornisce in uscita un byte di informazioni parallele, un segnale **reset** ed un segnale **start**, che consente di avviare la trasmissione delle informazioni quando si alza. Per ottenere una configurazione a tappo le porte **TXD** e **RXD** sono collegate mediante un segnale interno (**internal**).

Per la variante a due **UART**, invece, per implementare la macchina richiesta sono stati utilizzati due **UartComponent**, dove uno opera da trasmettitore e l'altro da ricevitore. Anche qui, come avviene nell'esercizio precedente, la lettura dei dati dal buffer è ritenuta immediata. Il seguente componente è stato realizzato con un approccio di tipo Structural con la dichiarazione e l'istanziazione di due **UARTcomponent**. L'interfaccia, nell'entity **UART 2**, ha quattro segnali di ingresso (**clock**, **dato_in**, **start**, **reset**) e un solo segnale d'uscita (**DBOUT**). Il trasmettitore trasmette un byte di informazioni seriali sulla porta **TXD**, quindi viene affidato ad un segnale interno (**internal**) che trasmette un bit sulla porta **RXD** del ricevitore.

10.4 Schematici

In Figura 10.9 la struttura della configurazione UART a tappo.

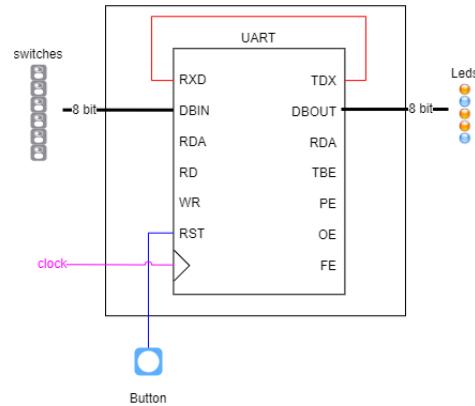


Figura 10.9: Configurazione a tappo

In Figura 10.10 la struttura della configurazione a due UART.

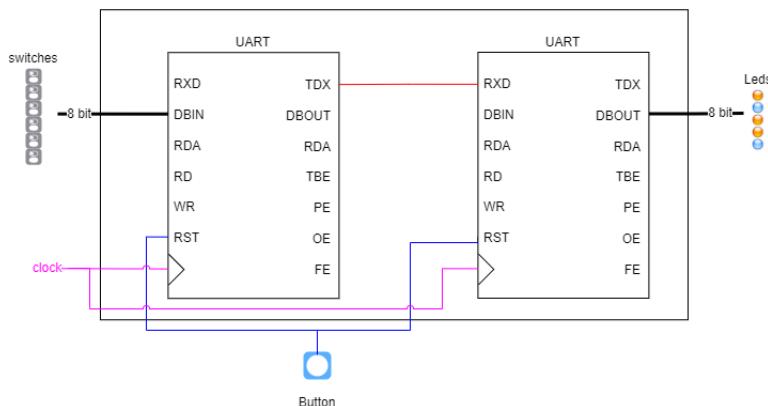


Figura 10.10: Configurazione con due UART

10.5 Codice

10.5.1 UART Tappo

Per la dichiarazione dei segnali, è stata utilizzata una keyword OPEN, la quale indica il legame con un pin non collegato o non utilizzato. In questo caso, i segnali TBE e RDA, dell'UARTcomponent, assumono il valore "open".

```
entity UART_tappo is
    Port (
        clock: in std_logic;
        reset: in std_logic;
        start: in std_logic;
        dato_in: in std_logic_vector(7 downto 0);
        DBOUT: out std_logic_vector(7 downto 0)
    );
end UART_tappo;

architecture structural of UART_tappo is

    component UARTcomponent is
        Generic (
            BAUD_DIVIDE_G : integer := 14; --115200 baud
            BAUD_RATE_G : integer := 231
        );
        Port (
            TXD : out std_logic := '1';-- Transmitted serial data output
            RXD : in std_logic; -- Received serial data input
            CLK : in std_logic; -- Clock signal
            DBIN : in std_logic_vector (7 downto 0); -- Input parallel data to be transmitted
            DBOUT : out std_logic_vector (7 downto 0); -- Recevived parallel data output
            RDA : inout std_logic; -- Read Data Available
            TBE : out std_logic := '1'; -- Transfer Buffer Emty
            RD : in std_logic; -- Read Strobe
            WR : in std_logic; -- Write Strobe
            PE : out std_logic; -- Parity error
            FE : out std_logic; -- Frame error
            OE : out std_logic; -- Overwrite error
            RST : in std_logic := '0'); -- Reset signal
    end component;

    -- SEGNALI INTERNI
    signal internal: std_logic;

begin
    UART: UARTcomponent
```

```

port map(
    TXD=>internal,
    RXD=>internal,
    CLK=>clock,
    DBIN=>dato_in,
    DBOUT=>DBOUT,
    RDA=>OPEN,
    TBE=>OPEN,
    RD=>'0',
    WR=>start,
    RST=>reset
);
end structural;

```

10.5.2 2 UART

```

entity UART2 is
  Port (
    clock: in std_logic;
    reset: in std_logic;
    start: in std_logic;
    dato_in: in std_logic_vector(7 downto 0);
    DBOUT: out std_logic_vector(7 downto 0)
  );
end UART2;

architecture structural of UART2 is

  component UARTcomponent is
    Generic (
      BAUD_DIVIDE_G : integer := 14; --115200 baud
      BAUD_RATE_G : integer := 231
    );
    Port (
      TXD : out std_logic := '1';-- Transmitted serial data output
      RXD : in std_logic; -- Received serial data input
      CLK : in std_logic; -- Clock signal
      DBIN : in std_logic_vector (7 downto 0); -- Input parallel data to be transmitted
      DBOUT : out std_logic_vector (7 downto 0); -- Recevived parallel data output
      RDA : inout std_logic; -- Read Data Available
      TBE : out std_logic := '1'; -- Transfer Buffer Emty
      RD : in std_logic; -- Read Strobe
      WR : in std_logic; -- Write Strobe
      PE : out std_logic; -- Parity error
      FE : out std_logic; -- Frame error
      OE : out std_logic; -- Overwrite error
    );
  end component;
end;

```

```

    RST : in std_logic := '0'); -- Reset signal
end component;

-- SEGNALI INTERNI
signal internal: std_logic;
signal uscita_temp: std_logic_vector(7 downto 0);
signal temp1,temp2,temp3 : std_logic := '0';

begin

UART_TX: UARTcomponent
port map(
    TXD=>internal,
    RXD=>'0',
    CLK=>clock,
    DBIN=>dato_in,
    DBOUT=>uscita_temp,
    RDA=>temp1,
    RD=>'0',
    WR=>start,
    RST=>reset
);

UART_RX: UARTcomponent
port map(
    TXD=>temp3,
    RXD=>internal,
    CLK=>clock,
    DBIN=>(others=>'0'),
    DBOUT=>DBOUT,
    RDA=>temp2,
    RD=>'0',
    WR=>'0',
    RST=>reset
);

end structural;

```

10.6 Simulazione

Si è scelto quindi di eseguire anche una simulazione di tale codice, prima di passare ad implementare l'architettura su scheda, in modo da verificarne il funzionamento. Si riporta, in Figura 10.11 la simulazione ottenuta, che è la medesima di entrambe le configurazioni.



Figura 10.11: Simulazione UART a tappo e con due UART

10.7 Sintesi su FPGA

Per la sintesi sull'FPGA, si è scelto di mappare gli switches come porte di ingresso, un bottone per il reset, e 8 LED per la visualizzazione dell'output. Di seguito il file dei constraints modificato.

```

## Clock signal
set_property -dict { PACKAGE_PIN E3      IOSTANDARD LVCMS3 } [get_ports { clock }];

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMS3 } [get_ports { data_in[0] }];
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMS3 } [get_ports { data_in[1] }];
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMS3 } [get_ports { data_in[2] }];
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMS3 } [get_ports { data_in[3] }];
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMS3 } [get_ports { data_in[4] }];
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMS3 } [get_ports { data_in[5] }];
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMS3 } [get_ports { data_in[6] }];
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMS3 } [get_ports { data_in[7] }];
[...]
set_property -dict { PACKAGE_PIN V10      IOSTANDARD LVCMS3 } [get_ports { start }];

## LEDs
[...]
set_property -dict { PACKAGE_PIN V16      IOSTANDARD LVCMS3 } [get_ports { DBOUT[0] }];
set_property -dict { PACKAGE_PIN T15      IOSTANDARD LVCMS3 } [get_ports { DBOUT[1] }];
set_property -dict { PACKAGE_PIN U14      IOSTANDARD LVCMS3 } [get_ports { DBOUT[2] }];
set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMS3 } [get_ports { DBOUT[3] }];
set_property -dict { PACKAGE_PIN V15      IOSTANDARD LVCMS3 } [get_ports { DBOUT[4] }];
set_property -dict { PACKAGE_PIN V14      IOSTANDARD LVCMS3 } [get_ports { DBOUT[5] }];
set_property -dict { PACKAGE_PIN V12      IOSTANDARD LVCMS3 } [get_ports { DBOUT[6] }];
set_property -dict { PACKAGE_PIN V11      IOSTANDARD LVCMS3 } [get_ports { DBOUT[7] }];

##Buttons
set_property -dict { PACKAGE_PIN C12      IOSTANDARD LVCMS3 } [get_ports { reset }];

```

A valle di tale codice, al settaggio del pin di start, vengono letti i valori immessi tramite switch, e vengono accesi i corrispettivi led sulla board. In Figura 10.12 è mostrato il funzionamento di tali componenti sulla board, sia nella configurazione a tappo che con due UART: in particolare, in ingresso vi è posto "01010101" tramite switch, e vengono accesi i led corretti per la visualizzazione a valle dell'attivazione dello switch di **start**.

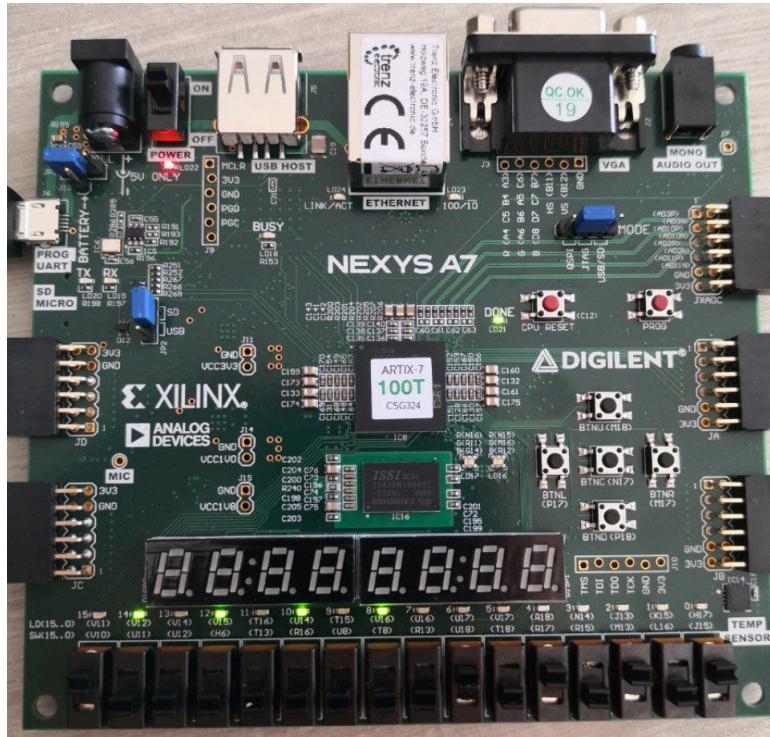


Figura 10.12: Sintesi su scheda dell'UART

Capitolo 11

Esercizio 10

11.1 Traccia

Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network.

Lo switch progettato deve operare come segue:

- 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 (es. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).
- (*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).
- (*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.

11.2 Soluzione 1

La soluzione proposta si compone da una unità di controllo che implementa un meccanismo a priorità e una rete di **switch** (che funge quindi da unità operativa) realizzata con la tecnica del **perfect shuffling**.

Tale tecnica, detta anche **faro shuffle**, è una tecnica che deriva dal mescolamento di un mazzo di carte: il mazzo viene diviso in due parti perfettamente identiche, e le carte dei due sotto mazzi vengono accoppiate pescando dalla testa di tali mazzetti. Il mescolamento termina con un ordine identico a quello di partenza. Si riporta in Figura 11.1 un esempio di tale shuffle.

| Step | Top Card | 2 | 3 | 4 | 5 | Bottom Card |
|-------|----------|---|---|---|---|-------------|
| Start | | | | | | |
| 1 | | | | | | |
| 2 | | | | | | |
| 3 | | | | | | |

Figura 11.1: Tecnica del perfect faro shuffle

La rete a priorità riceve in ingresso i quattro bit di abilitazione dagli switch e gli indirizzi di destinazione di ogni ingresso e li elabora attraverso le seguenti equazioni logiche:

$$\begin{aligned} q_0 &= i_0 \\ q_1 &= \overline{i_0} \cdot i_1 \\ q_2 &= \overline{i_0} \cdot \overline{i_1} \cdot i_2 \\ q_3 &= \overline{i_0} \cdot \overline{i_1} \cdot \overline{i_2} \cdot i_3 \end{aligned}$$

Per stabilire le uscite della rete di codifica a valle della rete a priorità così gestita, è necessario definire delle codifiche per tali bit. Si è scelto di seguire la codifica in Tabella 11.1.

| q_0 | q_1 | q_2 | q_3 | buff_1 | buff_0 |
|-----|-----|-----|-----|--------|--------|
| 0 | 0 | 0 | 1 | 0 | 0 |
| 0 | 0 | 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 0 | 0 | 1 | 1 |

Tabella 11.1: Tabella di codifica dei bit di priorità

Tali uscite rispettano quindi le seguenti relazioni:

$$\begin{aligned} buff_1 &= (\overline{q_0} + \overline{q_1} + \overline{q_2} + q_3) \cdot (\overline{q_0} + \overline{q_1} + q_2 + \overline{q_3}) \\ buff_0 &= (\overline{q_0} + \overline{q_1} + \overline{q_2} + q_3) \cdot (\overline{q_0} + q_1 + \overline{q_2} + \overline{q_3}) \end{aligned}$$

A seconda quindi della combinazione dei due bit $buff_1$ e $buff_0$ vengono selezionate le sorgenti che devono inviare i dati alla rete di switching. La rete di switching è composta da un numero di switch 2:2 pari al numero di segnali in ingresso, nel caso in esame 4, opportunamente collegati tra loro. La rete di controllo quindi si occupa di definire il percorso corretto da attivare per il collegamento tra sorgente e destinazione, attraverso i segnali definiti in precedenza.

11.3 Schematici 1

Lo **switch 2:2** è l'unità fondamentale dell'architettura della rete, e quindi della parte operativa di tale esercizio. Quest'ultimo è stato realizzato mediante composizione di un mux 2:1 ed un demux 1:2, come raffigurato in Figura 11.2. Come si nota, entrambi i componenti prevedono un

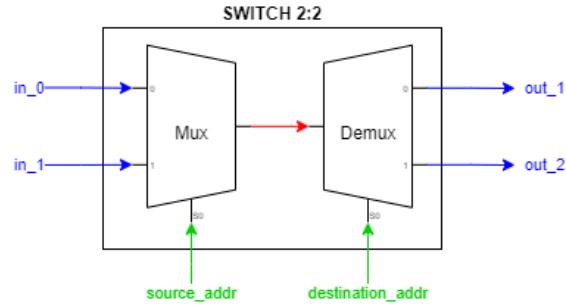


Figura 11.2: Architettura interna dello switch 2:2

bit di selezione, chiamati **source_addr** e **destination_addr**, utile a definire il percorso che deve intraprendere il dato all'interno di mux e demux. Ad esempio, se **source_addr** è pari ad 0, viene inviato il valore corrispondente all'ingresso **in_0** del multiplexer; se **destination_addr** è pari a 1 viene selezionato il valore in uscita **out_2** del demultiplexer.

Seguendo le tecniche di shuffling e progettando la rete secondo il modello omega, si è definita l'unità operativa come composta da un numero di livelli pari a $\log_2 M$ dove M rappresenta il numero di switch elementari presenti nella rete. In tal caso, la rete si compone di due livelli, ognuno comprendente due switch 2:2, collegati come mostrato in Figura 11.3.

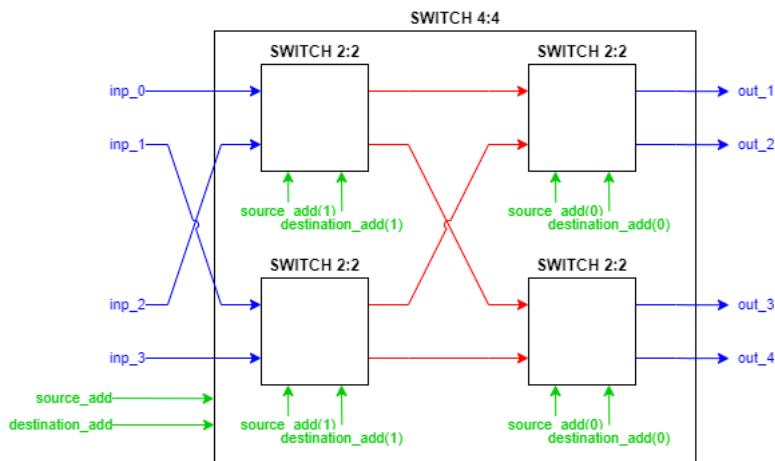


Figura 11.3: Architettura delle reti Omega

11.3.1 Gestione della priorità

Per gestire la priorità si è scelto di definire una rete di controllo a monte dell'unità operativa, rappresentata unicamente dalla rete Omega, che definisce i segnali di abilitazione dei mux e demux elementari presenti all'interno degli switch 2:2, secondo le priorità stabilite a priori e descritte in precedenza. L'architettura completa di tale sistema è riportata quindi in Figura 11.4. All'interno

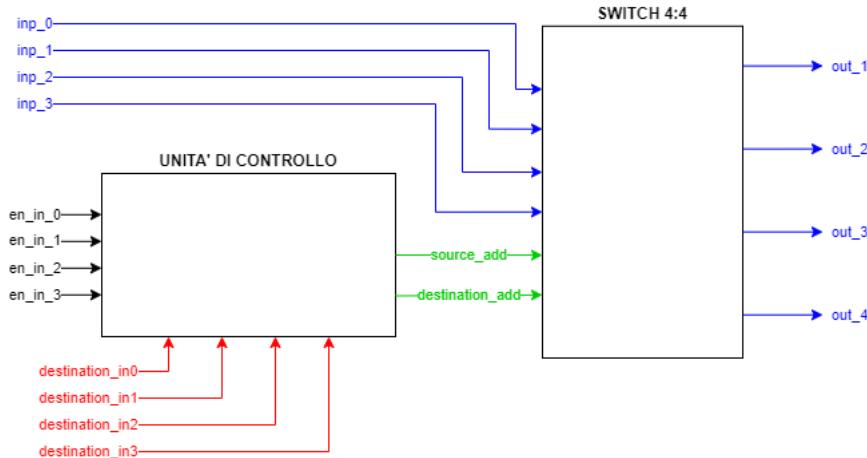


Figura 11.4: Architettura del sistema con priorità

di tale architettura, i segnali `destination_in(i)` vengono scelti tramite i due bit `source_add`, codificati attraverso il meccanismo di priorità definito nel paragrafo precedente sui segnali `en_in_i`.

11.4 Codice 1

11.4.1 Unità operativa

Mux 2:1

```
entity mux_2_1 is
Port(
    a0 :in std_logic_vector(1 downto 0);
    a1 :in std_logic_vector(1 downto 0);
    s :in std_logic;
    y :out std_logic_vector(1 downto 0)
);
end mux_2_1;

architecture dataflow of mux_2_1 is
begin
    y <= a0 when s='0' else a1 when s='1' else "UU";
end dataflow;
```

Demux 2:1

```
entity demux_2_1 is
  Port (
    a: in std_logic_vector(1 downto 0);
    s: in std_logic;
    y0: out std_logic_vector(1 downto 0);
    y1: out std_logic_vector(1 downto 0)
  );
end demux_2_1;

architecture dataflow of demux_2_1 is
begin
  y0<= a when s='0' else "XX";
  y1<= a when s='1' else "XX";
end dataflow;
```

Switch 2:2

```
entity switch_2_2 is
  Port(
    in_0: in STD_LOGIC_VECTOR(1 downto 0);
    in_1: in STD_LOGIC_VECTOR(1 downto 0);
    source_addr: in STD_LOGIC;
    destination_addr: in STD_LOGIC;
    out_1: out STD_LOGIC_VECTOR(1 downto 0);
    out_2: out STD_LOGIC_VECTOR(1 downto 0)
  );
end switch_2_2;

architecture Structural of switch_2_2 is
  component mux_2_1 is
    Port(
      a0: in std_logic_vector (1 downto 0);
      a1: in std_logic_vector (1 downto 0);
      s: in std_logic;
      y: out std_logic_vector (1 downto 0)
    );
  end component;
  component demux_2_1 is
    Port (
      a: in std_logic_vector(1 downto 0);
      s: in std_logic;
      y0: out std_logic_vector(1 downto 0);
      y1: out std_logic_vector(1 downto 0)
    );
  end component;
  signal k: STD_LOGIC_VECTOR(1 downto 0);
```

```

begin
    mux: mux_2_1
        port map(
            a0 => in_0,
            a1 => in_1,
            s => source_addr,
            y => k
        );
    demux: demux_2_1
        port map(
            a => k,
            s => destination_addr,
            y0 => out_1,
            y1 => out_2
        );
end Structural;

```

Switch 4:4

```

entity switch_4_4 is
    Port (
        inp_0: in STD_LOGIC_VECTOR(1 downto 0);
        inp_1: in STD_LOGIC_VECTOR(1 downto 0);
        inp_2: in STD_LOGIC_VECTOR(1 downto 0);
        inp_3: in STD_LOGIC_VECTOR(1 downto 0);
        source_add: in STD_LOGIC_VECTOR(1 downto 0);
        destination_add: in STD_LOGIC_VECTOR(1 downto 0);
        out_1: out STD_LOGIC_VECTOR(1 downto 0);
        out_2: out STD_LOGIC_VECTOR(1 downto 0);
        out_3: out STD_LOGIC_VECTOR(1 downto 0);
        out_4: out STD_LOGIC_VECTOR(1 downto 0)
    );
end switch_4_4;

architecture Structural of switch_4_4 is
    component switch_2_2 is
        Port(
            in_0: in STD_LOGIC_VECTOR(1 downto 0);
            in_1: in STD_LOGIC_VECTOR(1 downto 0);
            source_addr: in STD_LOGIC;
            destination_addr: in STD_LOGIC;
            out_1: out STD_LOGIC_VECTOR(1 downto 0);
            out_2: out STD_LOGIC_VECTOR(1 downto 0)
        );
    end component;
    type intermedio is array(0 to 3) of STD_LOGIC_VECTOR(1 downto 0);
    signal buff: intermedio;

```

```

begin
    switch_0: switch_2_2
        port map(
            in_0 => inp_0,
            in_1 => inp_2,
            source_addr => source_add(1),
            destination_addr => destination_add(1),
            out_1 => buff(0),
            out_2 => buff(1)
        );
    switch_1: switch_2_2
        port map(
            in_0 => inp_1,
            in_1 => inp_3,
            source_addr => source_add(1),
            destination_addr => destination_add(1),
            out_1 => buff(2),
            out_2 => buff(3)
        );
    switch_2: switch_2_2
        port map(
            in_0 => buff(0),
            in_1 => buff(2),
            source_addr => source_add(0),
            destination_addr => destination_add(0),
            out_1 => out_1,
            out_2 => out_2
        );
    switch_3: switch_2_2
        port map(
            in_0 => buff(1),
            in_1 => buff(3),
            source_addr => source_add(0),
            destination_addr => destination_add(0),
            out_1 => out_3,
            out_2 => out_4
        );
end Structural;

```

11.4.2 Unità di controllo

```

entity unita_controllo is
    Port (
        en_in_0: in STD_LOGIC;
        en_in_1: in STD_LOGIC;
        en_in_2: in STD_LOGIC;
        en_in_3: in STD_LOGIC;

```

```

        destination_in_0: in STD_LOGIC_VECTOR(1 downto 0);
        destination_in_1: in STD_LOGIC_VECTOR(1 downto 0);
        destination_in_2: in STD_LOGIC_VECTOR(1 downto 0);
        destination_in_3: in STD_LOGIC_VECTOR(1 downto 0);
        source_addr: out STD_LOGIC_VECTOR(1 downto 0);
        destination_addr: out STD_LOGIC_VECTOR(1 downto 0)
    );
end unita_controllo;

architecture Dataflow of unita_controllo is
    signal q0, q1, q2, q3: STD_LOGIC;
    signal buff: STD_LOGIC_VECTOR(1 downto 0);
begin
    begin
        --1. Determino la priorità
        q0 <= en_in_0;
        q1 <= (not en_in_0) and en_in_1;
        q2 <= (not en_in_0) and (not en_in_1) and en_in_2;
        q3 <= (not en_in_0) and (not en_in_1) and (not en_in_2) and en_in_3;
        --2. Codifico l'indirizzo sorgente
        buff(1) <= ((not q0) and (not q1) and (not q2) and q3) or ((not q0) and (not q1) and q2 and (not q3));
        buff(0) <= ((not q0) and (not q1) and (not q2) and q3) or ((not q0) and q1 and (not q2) and (not q3));
        source_addr(1) <= buff(1);
        source_addr(0) <= buff(0);
        --3. Determino l'indirizzo di destinazione
        with buff select
            destination_addr <=
                destination_in_0 when "00",
                destination_in_1 when "01",
                destination_in_2 when "10",
                destination_in_3 when "11",
                "UU" when others;
    end Dataflow;

```

11.5 Simulazione 1

Per simulare il funzionamento di tale struttura, è stato generato l'apposito testbench riportato di seguito. L'esecuzione di tale bench ha prodotto le forme d'onda riportate in Figura 11.5.

```

entity switch_omega_network_tb is
end switch_omega_network_tb;

architecture Behavioral of switch_omega_network_tb is
    component switch_omega_network is
        Port (
            en_in_0: in STD_LOGIC;
            en_in_1: in STD_LOGIC;
            en_in_2: in STD_LOGIC;

```

```

    en_in_3: in STD_LOGIC;
    destination_in_0: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_1: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_2: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_3: in STD_LOGIC_VECTOR(1 downto 0);
    inp_0: in STD_LOGIC_VECTOR(1 downto 0);
    inp_1: in STD_LOGIC_VECTOR(1 downto 0);
    inp_2: in STD_LOGIC_VECTOR(1 downto 0);
    inp_3: in STD_LOGIC_VECTOR(1 downto 0);
    out_0: out STD_LOGIC_VECTOR(1 downto 0);
    out_1: out STD_LOGIC_VECTOR(1 downto 0);
    out_2: out STD_LOGIC_VECTOR(1 downto 0);
    out_3: out STD_LOGIC_VECTOR(1 downto 0)

);

end component;

--Inputs
signal en_in_0: STD_LOGIC := '0';
signal en_in_1: STD_LOGIC := '0';
signal en_in_2: STD_LOGIC := '0';
signal en_in_3: STD_LOGIC := '0';
signal destination_in_0: STD_LOGIC_VECTOR(1 downto 0);
signal destination_in_1: STD_LOGIC_VECTOR(1 downto 0);
signal destination_in_2: STD_LOGIC_VECTOR(1 downto 0);
signal destination_in_3: STD_LOGIC_VECTOR(1 downto 0);
signal inp_0: STD_LOGIC_VECTOR(1 downto 0);
signal inp_1: STD_LOGIC_VECTOR(1 downto 0);
signal inp_2: STD_LOGIC_VECTOR(1 downto 0);
signal inp_3: STD_LOGIC_VECTOR(1 downto 0);

--Outputs
signal out_0: STD_LOGIC_VECTOR(1 downto 0);
signal out_1: STD_LOGIC_VECTOR(1 downto 0);
signal out_2: STD_LOGIC_VECTOR(1 downto 0);
signal out_3: STD_LOGIC_VECTOR(1 downto 0);

begin
uut: entity work.switch_omega_network(Structural)
port map(
    en_in_0 => en_in_0,
    en_in_1 => en_in_1,
    en_in_2 => en_in_2,
    en_in_3 => en_in_3,
    destination_in_0 => destination_in_0,
    destination_in_1 => destination_in_1,
    destination_in_2 => destination_in_2,
    destination_in_3 => destination_in_3,
    inp_0 => inp_0,
    inp_1 => inp_1,

```

```

    inp_2 => inp_2,
    inp_3 => inp_3,
    out_0 => out_0,
    out_1 => out_1,
    out_2 => out_2,
    out_3 => out_3
);

stim_proc: process
begin
    wait for 10ns;
    inp_0 <= "10";
    inp_1 <= "11";
    inp_2 <= "11";
    inp_3 <= "00";
    en_in_0 <= '1';
    en_in_1 <= '1';
    en_in_2 <= '1';
    en_in_3 <= '1';
    destination_in_0 <= "01";
    destination_in_1 <= "11";
    destination_in_2 <= "00";
    destination_in_3 <= "10";
    wait for 10ns;
    en_in_0 <= '0';
    wait for 10ns;
    en_in_1 <= '0';
    wait for 10ns;
    en_in_2 <= '0';
    wait;
end process;
end Behavioral;

```

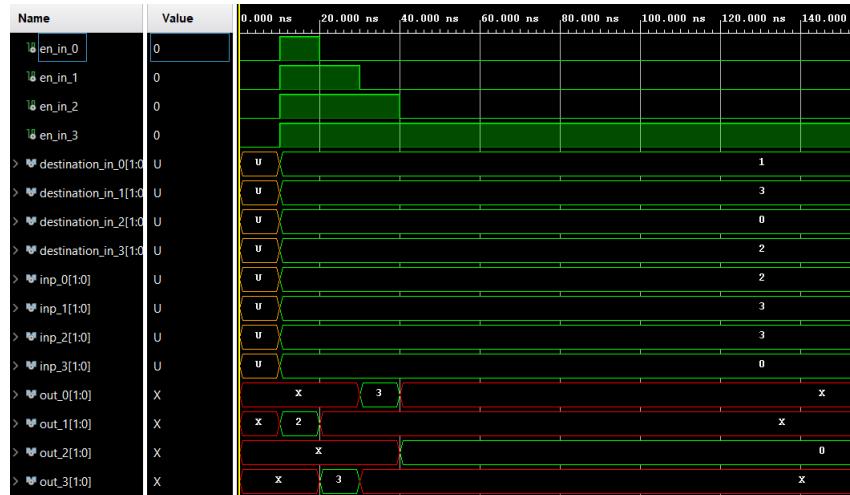


Figura 11.5: Simulazione Switch multistadio versione 1

11.6 Soluzione 2

Per poter risolvere questo quesito è stato necessario riprogettare tutto il sistema a partire dal componente Switch 2:2 poichè non in grado di gestire le eventuali collisioni.

Questo componente adesso è costituito da una unità operativa e da una unità di controllo. L'unità operativa adesso è costituita da due multiplexer 2:1 ognuno dei quali pilota una uscita potendo scegliere tra entrambi gli ingressi. Per poter avere una implementazione più efficiente ogni ingresso del componente è costituito da sei bit di cui:

- il **più significativo** rappresenta l'abilitazione dell'ingresso stesso utilizzando la convenzione delle comunicazioni seriali (1 linea in idle, 0 comunicazione attiva),
- i successivi **tre bit** (dovendo raggiungere 8 destinazioni diverse) rappresentano l'indirizzo di destinazione,
- gli **ultimi due** costituiscono il messaggio da trasmettere.

Il messaggio è rappresentato in Figura 11.6.



Figura 11.6: Schema di bit di un ingresso dei componenti

Come nel caso precedente la rete è stata realizzata mediante la tecnica del perfect shuffling. Per quanto riguarda la gestione delle collisioni, invece, si è optato per un meccanismo a perdita di messaggio.

11.7 Schematici 2

Come precedentemente descritto, il componente Switch 2:2 è stato realizzato dividendo parte operativa e parte di controllo. L'unità operativa è composta da due MUX, che ricevono entrambi i dati in ingresso e, attraverso i due bit di selezione del segnale **sel** si va a stabilire il porto di uscita di tale componente. La logica di gestione di tali segnali, ai fini dell'eliminazione e della gestione dei conflitti viene demandata all'unità di controllo.

Lo schema di uno Switch 2:2 è presentato in Figura 11.7.

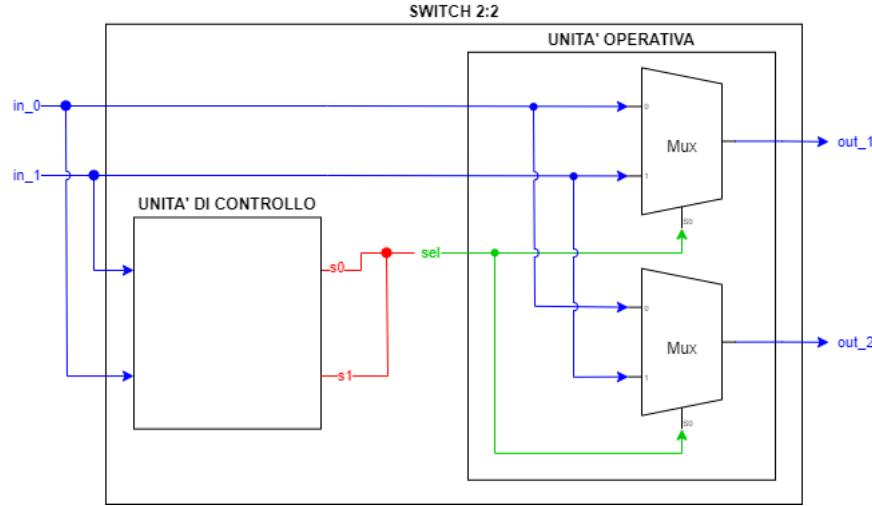


Figura 11.7: Architettura interna dello switch 2:2 - soluzione 2

Avendo spostato la logica di controllo dall'esterno, all'interno degli switch elementari, l'architettura dello Switch 8:8 consta del solo collegamento opportuno tra i singoli switch elementari così composti. Tale architettura è presentata in Figura 11.8.

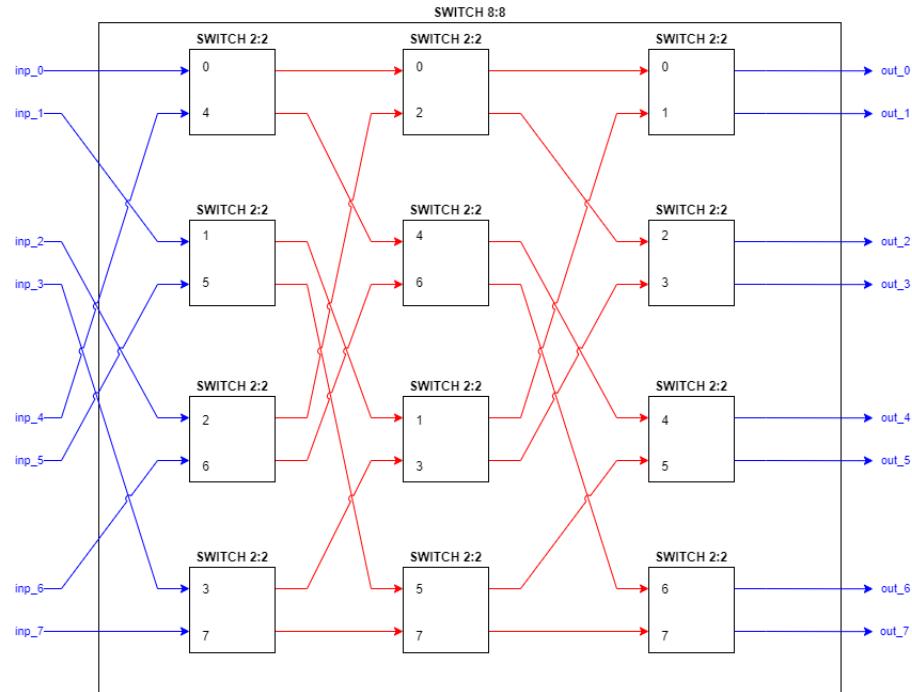


Figura 11.8: Architettura dello switch 8:8

11.8 Codice 2

11.8.1 Switch 2:2

Per quel che riguarda i componenti MUX all'interno dell'unità operativa, si è fatto uso dello stesso codice disponibile alla soluzione 1.

Unità operativa

```
entity unita_operativa is
    Port (
        in_0: in STD_LOGIC_VECTOR(5 downto 0);
        in_1: in STD_LOGIC_VECTOR(5 downto 0);
        sel: in STD_LOGIC_VECTOR(1 downto 0);
        out_0: out STD_LOGIC_VECTOR(5 downto 0);
        out_1: out STD_LOGIC_VECTOR(5 downto 0)
    );
end unita_operativa;

architecture Structural of unita_operativa is
    component mux is
        Port(
            a0 :in std_logic_vector(5 downto 0);
            a1 :in std_logic_vector(5 downto 0);
            s :in std_logic;
            y :out std_logic_vector(5 downto 0)
        );
    end component;
begin
    mux_0: mux
        port map(
            a0 => in_0,
            a1 => in_1,
            s => sel(0),
            y => out_0
        );
    mux_1: mux
        port map(
            a0 => in_0,
            a1 => in_1,
            s => sel(1),
            y => out_1
        );
end Structural;
```

Unità di controllo

Per gestire i vari livelli della rete, si è fatto uso del costrutto **generic**, creando una variabile chiamata **indice**, che fosse in grado di discriminare il livello a cui appartiene l'unità, e poter quindi smistare correttamente i messaggi al livello successivo.

```
entity unita_controllo is
    Generic( indice: integer := 0 );
    Port (
        in_0: in STD_LOGIC_VECTOR(3 downto 0);
        in_1: in STD_LOGIC_VECTOR(3 downto 0);
        s0: out STD_LOGIC;
        s1: out STD_LOGIC
    );
end unita_controllo;

architecture Dataflow of unita_controllo is
begin
    s0 <= '1' when (in_0(3)='1' and in_1(3)='0' and in_1(indice)='0') OR
                (in_0(indice)='1' and in_1(3)='0' and in_1(indice)='0') else
                '0' when (in_0(3)='0' and in_0(indice)='0') else
                'X';
    s1 <= '1' when (in_0(3)='1' and in_1(3)='0' and in_1(indice)='1') OR
                (in_0(indice)='0' and in_1(3)='0' and in_1(indice)='1') else
                '0' when (in_0(3)='0' and in_0(indice)='1') else
                'X';
end Dataflow;
```

Componente totale

```
entity switch_2_2 is
    Generic ( indice: integer :=0 );
    Port(
        in_0: in STD_LOGIC_VECTOR(5 downto 0);
        in_1: in STD_LOGIC_VECTOR(5 downto 0);
        out_0: out STD_LOGIC_VECTOR(5 downto 0);
        out_1: out STD_LOGIC_VECTOR(5 downto 0)
    );
end switch_2_2;

architecture Structural of switch_2_2 is
    component unita_operativa is
        Port (
            in_0: in STD_LOGIC_VECTOR(5 downto 0);
            in_1: in STD_LOGIC_VECTOR(5 downto 0);
            sel: in STD_LOGIC_VECTOR(1 downto 0);
            out_0: out STD_LOGIC_VECTOR(5 downto 0);
            out_1: out STD_LOGIC_VECTOR(5 downto 0)
        );
    end component;
```

```

);
end component;
component unita_controllo is
  Generic( indice: integer := 0 );
  Port (
    in_0: in STD_LOGIC_VECTOR(3 downto 0);
    in_1: in STD_LOGIC_VECTOR(3 downto 0);
    s0: out STD_LOGIC;
    s1: out STD_LOGIC
  );
end component;
signal selector: STD_LOGIC_VECTOR(1 downto 0);
signal k0: STD_LOGIC_VECTOR(3 downto 0);
signal k1: STD_LOGIC_VECTOR(3 downto 0);
signal sel0: STD_LOGIC;
signal sel1: STD_LOGIC;
begin
  uc: unita_controllo
    generic map(indice => indice)
    port map(
      in_0 => in_0(5 downto 2),
      in_1 => in_1(5 downto 2),
      s0 => sel0,
      s1 => sel1
    );
  selector <= sel1 & sel0;
  uo: unita_operativa
    port map(
      in_0 => in_0,
      in_1 => in_1,
      sel => selector,
      out_0 => out_0,
      out_1 => out_1
    );
end Structural;

```

11.8.2 Switch 8:8

```

entity switch_omega_network is
  Port (
    inp_0: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_1: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_2: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_3: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');

```

```

    inp_4: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_5: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_6: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    inp_7: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
    out_0: out STD_LOGIC_VECTOR(1 downto 0);
    out_1: out STD_LOGIC_VECTOR(1 downto 0);
    out_2: out STD_LOGIC_VECTOR(1 downto 0);
    out_3: out STD_LOGIC_VECTOR(1 downto 0);
    out_4: out STD_LOGIC_VECTOR(1 downto 0);
    out_5: out STD_LOGIC_VECTOR(1 downto 0);
    out_6: out STD_LOGIC_VECTOR(1 downto 0);
    out_7: out STD_LOGIC_VECTOR(1 downto 0)
);

end switch_omega_network;

architecture Structural of switch_omega_network is
component switch_2_2 is
  Generic ( indice: integer :=0 );
  Port(
    in_0: in STD_LOGIC_VECTOR(5 downto 0);
    in_1: in STD_LOGIC_VECTOR(5 downto 0);
    out_0: out STD_LOGIC_VECTOR(5 downto 0);
    out_1: out STD_LOGIC_VECTOR(5 downto 0)
  );
end component;
type collegamenti is ARRAY(0 to 7) of STD_LOGIC_VECTOR(5 downto 0);
signal stadio1: collegamenti;
signal stadio2: collegamenti;
signal buffer_uscita: collegamenti;
begin
  --SWITCH INTERNI
  s0: switch_2_2
    generic map(indice=>2)
    port map(
      in_0 => inp_0,
      in_1 => inp_4,
      out_0 => stadio1(0),
      out_1 => stadio1(1)
    );
  s1: switch_2_2
    generic map(indice=>2)
    port map(
      in_0 => inp_1,
      in_1 => inp_5,
      out_0 => stadio1(2),
      out_1 => stadio1(3)

```

```

    );
s2: switch_2_2
    generic map(indice=>2)
    port map(
        in_0 => inp_2,
        in_1 => inp_6,
        out_0 => stadio1(4),
        out_1 => stadio1(5)
    );
s3: switch_2_2
    generic map(indice=>2)
    port map(
        in_0 => inp_3,
        in_1 => inp_7,
        out_0 => stadio1(6),
        out_1 => stadio1(7)
    );
s4: switch_2_2
    generic map(indice=>1)
    port map(
        in_0 => stadio1(0),
        in_1 => stadio1(4),
        out_0 => stadio2(0),
        out_1 => stadio2(2)
    );
s5: switch_2_2
    generic map(indice=>1)
    port map(
        in_0 => stadio1(1),
        in_1 => stadio1(5),
        out_0 => stadio2(4),
        out_1 => stadio2(6)
    );
s6: switch_2_2
    generic map(indice=>1)
    port map(
        in_0 => stadio1(2),
        in_1 => stadio1(6),
        out_0 => stadio2(1),
        out_1 => stadio2(3)
    );
s7: switch_2_2
    generic map(indice=>1)
    port map(
        in_0 => stadio1(3),
        in_1 => stadio1(7),
        out_0 => stadio2(5),

```

```

        out_1 => stadio2(7)
    );
s8: switch_2_2
generic map(indice=>0)
port map(
    in_0 => stadio2(0),
    in_1 => stadio2(1),
    out_0 => buffer_uscita(0),
    out_1 => buffer_uscita(1)
);
s9: switch_2_2
generic map(indice=>0)
port map(
    in_0 => stadio2(2),
    in_1 => stadio2(3),
    out_0 => buffer_uscita(2),
    out_1 => buffer_uscita(3)
);
s10: switch_2_2
generic map(indice=>0)
port map(
    in_0 => stadio2(4),
    in_1 => stadio2(5),
    out_0 => buffer_uscita(4),
    out_1 => buffer_uscita(5)
);
s11: switch_2_2
generic map(indice=>0)
port map(
    in_0 => stadio2(6),
    in_1 => stadio2(7),
    out_0 => buffer_uscita(6),
    out_1 => buffer_uscita(7)
);

--USCITE
out_0<=buffer_uscita(0)(1 downto 0);
out_1<=buffer_uscita(1)(1 downto 0);
out_2<=buffer_uscita(2)(1 downto 0);
out_3<=buffer_uscita(3)(1 downto 0);
out_4<=buffer_uscita(4)(1 downto 0);
out_5<=buffer_uscita(5)(1 downto 0);
out_6<=buffer_uscita(6)(1 downto 0);
out_7<=buffer_uscita(7)(1 downto 0);

end Structural;

```

11.9 Simulazione 2

Per testare il funzionamento di tali componenti, si è scelto di creare il seguente testbench, che ha prodotto le forme d'onda in Figura 11.9.

```
entity switch_omega_network_tb is
end switch_omega_network_tb;

architecture Behavioral of switch_omega_network_tb is
component switch_omega_network is
    Port (
        inp_0: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_1: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_2: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_3: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_4: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_5: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_6: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        inp_7: in STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
        out_0: out STD_LOGIC_VECTOR(1 downto 0);
        out_1: out STD_LOGIC_VECTOR(1 downto 0);
        out_2: out STD_LOGIC_VECTOR(1 downto 0);
        out_3: out STD_LOGIC_VECTOR(1 downto 0);
        out_4: out STD_LOGIC_VECTOR(1 downto 0);
        out_5: out STD_LOGIC_VECTOR(1 downto 0);
        out_6: out STD_LOGIC_VECTOR(1 downto 0);
        out_7: out STD_LOGIC_VECTOR(1 downto 0)
    );
end component;

--Inputs
signal inp_0: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_1: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_2: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_3: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_4: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_5: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_6: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');
signal inp_7: STD_LOGIC_VECTOR(5 downto 0) := (others=>'1');

--Outputs
signal out_0: STD_LOGIC_VECTOR(1 downto 0);
signal out_1: STD_LOGIC_VECTOR(1 downto 0);
signal out_2: STD_LOGIC_VECTOR(1 downto 0);
signal out_3: STD_LOGIC_VECTOR(1 downto 0);
signal out_4: STD_LOGIC_VECTOR(1 downto 0);
signal out_5: STD_LOGIC_VECTOR(1 downto 0);
signal out_6: STD_LOGIC_VECTOR(1 downto 0);
signal out_7: STD_LOGIC_VECTOR(1 downto 0);
```

```

begin
  uut: switch_omega_network
    port map(
      inp_0 => inp_0,
      inp_1 => inp_1,
      inp_2 => inp_2,
      inp_3 => inp_3,
      inp_4 => inp_4,
      inp_5 => inp_5,
      inp_6 => inp_6,
      inp_7 => inp_7,
      out_0 => out_0,
      out_1 => out_1,
      out_2 => out_2,
      out_3 => out_3,
      out_4 => out_4,
      out_5 => out_5,
      out_6 => out_6,
      out_7 => out_7
    );
  stim_proc: process
  begin
    wait for 10 ns;
    inp_0<="011111"; --vado in out7 MSG: "11"
    inp_1<="010101";
    inp_2<="110000";
    inp_3<="100110";
    inp_4<="011111";
    inp_5<="000010";
    inp_6<="111111";
    inp_7<="010000";
    wait;
  end process;
end Behavioral;

```

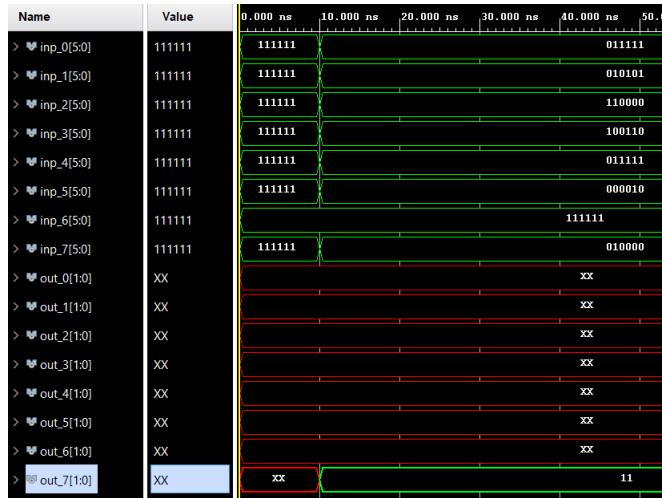


Figura 11.9: Simulazione Switch multistadio versione 2

11.10 Soluzione 3

Per progettare tale soluzione si è utilizzato lo switch multistadio implementato nella soluzione 1 opportunamente modificato per gestire un messaggio in ingresso a 8 bit.

Infatti, tale messaggio viene composto come segue:

- i primi due bit rappresentano lo stato dell’handshaking;
- i successivi bit rappresentano il dato e la codifica di sorgente e destinazione.

Le combinazioni sono quindi le seguenti:

- 00XXYY00 → stato di **IDLE**;
- 01XXYY00 → stato di **PRONTO_AD_INVIARE**;
- 10XXYY00 → stato di **PRONTO_A_RICEVERE**;
- 11XXYYDD → stato di **DATO**;

Pertanto la comunicazione si compone come segue:

1. Un messaggio dalla sorgente alla destinazione con i primi 2 bit posti a “01” per indicare la richiesta di invio;
2. Un messaggio dalla destinazione alla sorgente con i primi 2 bit posti a “10” per segnalare l'avvenuta predisposizione alla ricezione dei dati;
3. Il terzo messaggio dalla sorgente alla destinazione con i primi bit pari a “11” per indicare la trasmissione del messaggio di informazione.

Per gestire la comunicazione tramite handshaking tra i nodi si è scelto di implementare un componente **Gestore Handshake**, caratterizzato dai seguenti segnali:

- **Mex**: segnale a 4 bit che contiene la destinazione e il dato che si vogliono trasmettere;
- **En**: segnale di abilitazione per trasmettere il messaggio “Pronto a trasmettere”;
- **SRC**: segnale utilizzato per la codifica del nodo sorgente;
- **RX**: linea a 8 bit utilizzata per la ricezione attraverso lo switch multistadio;
- **TX**: linea a 8 bit utilizzata per la trasmissione attraverso lo switch multistadio;
- **TX_en**: segnale che indica l’intenzione di voler trasmettere all’interno dello switch (esso sarà in input alla rete di priorità).
- **dato_ricevuto**: segnale a due bit che rappresenta il dato “utile” ricevuto attraverso la rete (“11XXYYDD”).
- **clock** e **reset**: segnali necessari al corretto funzionamento e al ripristino di uno stato noto della macchina.

Il gestore Handshake è stato realizzato a partire dall’automa a stati finiti e siccome può gestire sia la trasmissione che la ricezione distinguiamo i due casi. Un gestore handhsake può implementare sia la trasmissione che la ricezione di un nodo.

L’automa è rappresentato in Figura 11.10.

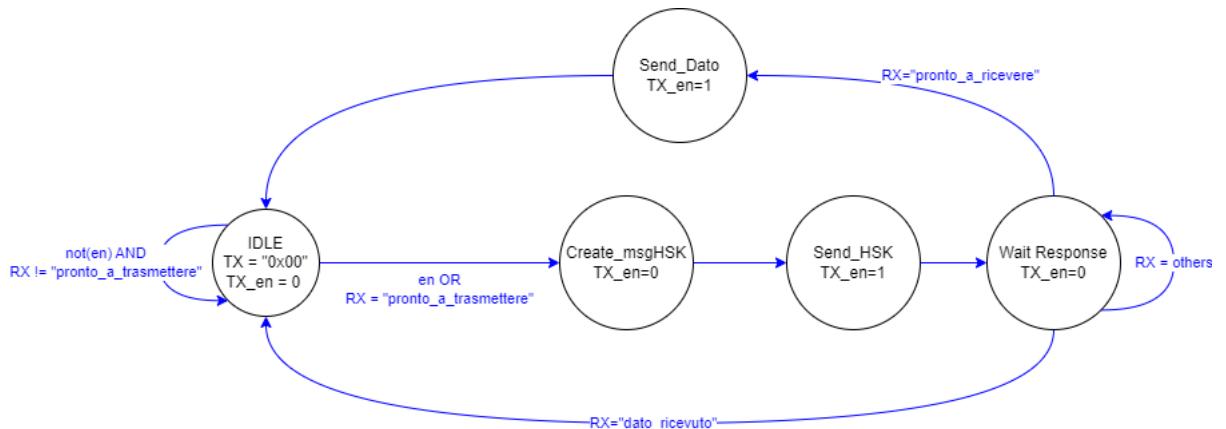


Figura 11.10: Diagramma a stati finiti del Gestore Handshake

Di seguito vengono descritti gli stati di tale automa:

1. Se un nodo non deve fare nulla e non deve trasmettere, rimane in **IDLE**.
2. Se si verifica la condizione, deve trasmettere o riceve un ”*pronto a trasmettere*”. Occorre adesso creare il messaggio. Nel caso in cui si vuole trasmettere deve generare il messaggio

di “*Pronto a trasmettere*” e, nel caso in cui riceve il messaggio “*Pronto a trasmettere*”, deve avvisare il nodo tramite la creazione del messaggio “*Pronto a ricevere*”.

3. Lo stato **Send_HSK** viene creato come stato per non complicare troppo l'automa; esso potrebbe essere accorpato per tanto anche allo stato “**Create_msgHSK**”. In questo stato ci si occupa all'abilitare la trasmissione all'interno della rete.
4. La macchina successivamente passa nello stato **Wait_Response** nel quale si divide il comportamento in trasmissione e ricezione. Se si riceve il messaggio di ”*Pronto a ricevere*” si crea il messaggio dati da trasmettere. Ed in questo caso si entra nello stato **Send_Data** che è identico allo stato di **Send_HSK**. Invece, nel caso in cui si riceve, il dato si pone in output sulla linea Rx e si ritorna allo stato di IDLE.

11.11 Schematici 3

Per rappresentare l'architettura, si è utilizzata la stessa architettura dello Switch 4:4, opportunamente modificata in modo tale che ogni Switch 2:2 interno riceva i segnali di ingresso e uscita del proprio Gestore Handshake.

In Figura 11.11 è mostrata l'architettura dello switch generale, dove vengono esplicitati i segnali di ingresso e di uscita dal componente top level.

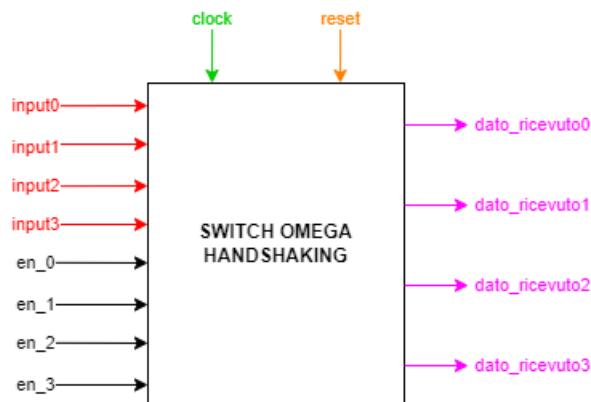


Figura 11.11: Architettura del top level Switch Handshaking

L'architettura si compone di differenti elementi rispetto a quella proposta nella soluzione 1. Infatti, per ogni Switch 2:2 è presente un Gestore Handshake che regola l'ingresso e l'uscita di ogni Switch 2:2, tramite il protocollo descritto in precedenza.

I segnali di ingresso $input_i$ vengono posti in input a tali gestori che, leggendo solo i relativi bit di handshake, regolano l'uscita e tempificano quest'ultima all'interno degli Switch 2:2.

L'architettura nello specifico viene presentata in Figura 11.12.

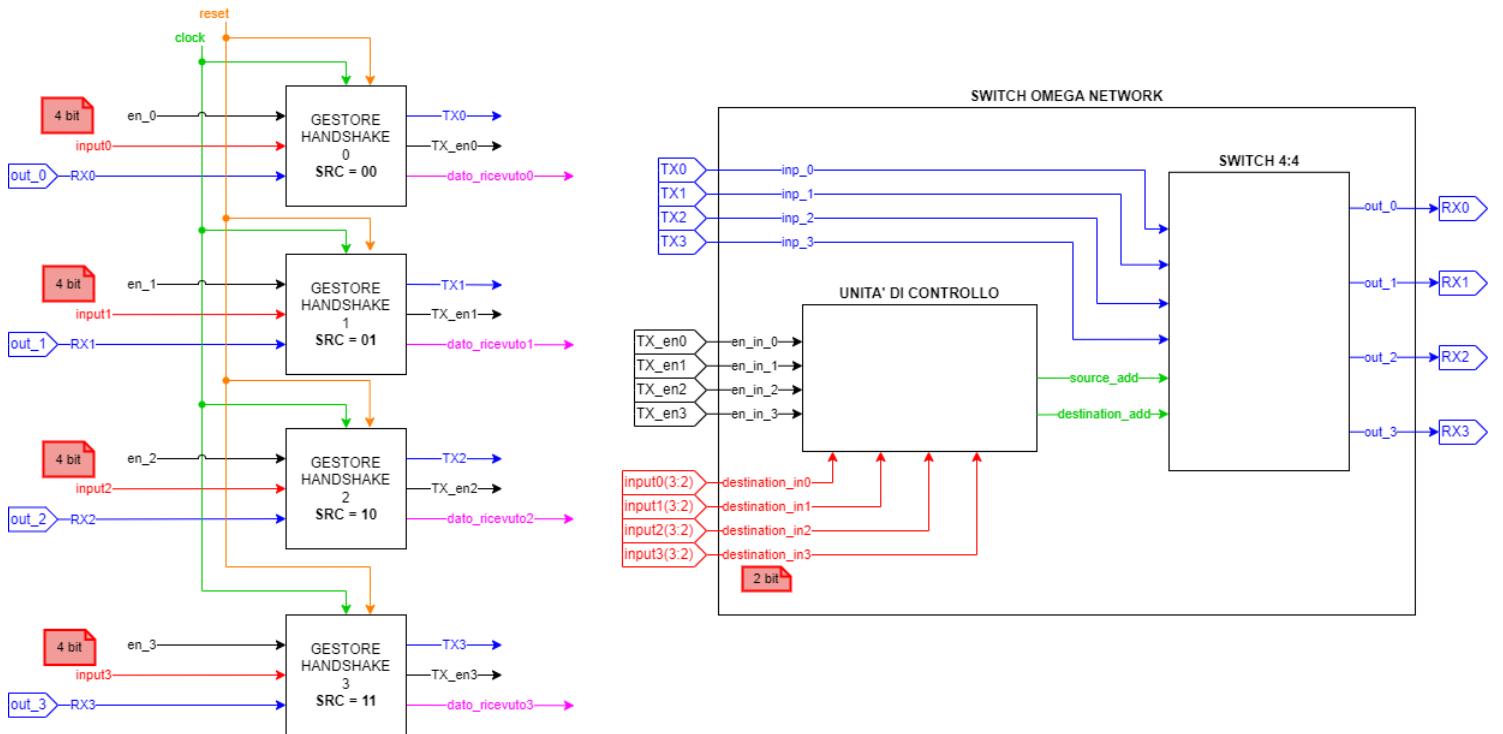


Figura 11.12: Architettura interna del sistema Switch Omega con Handshake

11.12 Codice 3

11.12.1 Gestore Handshake

L'implementazione del gestore Handshake è stata realizzata mediante traduzione dell'automa a stati finiti, e quindi sfruttando un processo comportamentale.

```

entity gestore_handshaking is
  Port (
    CLK: in STD_LOGIC;
    RST: in STD_LOGIC;
    SRC: in STD_LOGIC_VECTOR(1 downto 0);
    MEX: in STD_LOGIC_VECTOR(3 downto 0);
    EN: in STD_LOGIC;
    RX: in STD_LOGIC_VECTOR(7 downto 0);
    TX_EN: out STD_LOGIC;
    TX: out STD_LOGIC_VECTOR(7 downto 0);
    dato_ricevuto: out STD_LOGIC_VECTOR(1 downto 0)
  );
end gestore_handshaking;

architecture Behavioral of gestore_handshaking is
  signal source: STD_LOGIC_VECTOR(1 downto 0);

```

```

signal destination: STD_LOGIC_VECTOR(1 downto 0);
signal dato: STD_LOGIC_VECTOR(1 downto 0);

type stato is (IDLE, create_msgHand, send_handshake, wait_response, send_dato);
signal stato_corrente: stato := IDLE;
signal stato_prossimo: stato;

begin
    source <= SRC;
    destination <= MEX(3 downto 2);
    dato <= MEX(1 downto 0);

    f_stato_uscita: process(stato_corrente, CLK)
    begin
        case stato_corrente is
            when IDLE =>
                --Se deve trasmettere o se deve essere pronto a trasmettere
                --evolve i due bit più significativi indicando il tipo di messaggio
                --Pronto a trasmettere <= "10"
                if(EN='1' or RX(7 downto 6)="10") then
                    stato_prossimo <= create_msgHand;
                else
                    stato_prossimo <= IDLE;
                end if;
                TX <= (others=>'0');
                TX_en <= '0';
                --Nel caso voglia trasmettere deve generare il segnale pronto a trasmettere
            when create_msgHand =>
                --Richiesta di trasmissione, devo verificare quale condizione è vera
                if(EN='1') then
                    TX <= "10" & source & destination & dato;
                    --Richiesta di trasmissione accettata
                elsif(RX(7 downto 6)="10") then
                    --Nodo pronto a ricevere
                    TX <= "01" & SRC & RX(5 downto 4) & RX(1 downto 0);
                end if;
                stato_prossimo <= send_handshake;
            when send_handshake =>
                --Viene abilitata la trasmissione nella rete
                TX_EN <= '1';
                stato_prossimo <= wait_response;
            when wait_response =>
                TX_en <= '0';
                --Se riceve pronto a ricevere può inviare i dati
                if(RX(7 downto 6)="01") then
                    --Invia il dato
                    TX <= "11" & source & destination & dato;
                    stato_prossimo <= send_dato;

```

```

--Se riceve i dati, li mette in output
elsif(RX(7 downto 6)="11") then
    --Resetta TX per una nuova comunicazione
    TX <= (others=>'-' );
    dato_ricevuto <= RX(1 downto 0);
    stato_prossimo <= IDLE;
elsif(RX(7 downto 6)="10") then
    TX <= (others => '-');
    dato_ricevuto <= RX(1 downto 0);
    stato_prossimo <= IDLE;
else
    stato_prossimo <= wait_response;
end if;
when send_dato =>
    TX_en <= '1';
    stato_prossimo <= IDLE;
end case;
end process;

memoria: process(CLK, RST)
begin
if(CLK'event and CLK='1') then
    if(RST='1') then
        stato_corrente <= IDLE;
    else
        stato_corrente <= stato_prossimo;
    end if;
end if;
end process;
end Behavioral;

```

11.12.2 Switch Omega Handshaking

Come mostra il codice, in ingresso ai bit utili per selezionare la destinazione dell'omega network vengono forniti i bit più significativi dell'ingresso, corrispondenti alla destinazione scelta del dato nodo.

```

entity switch_omega_handshaking is
Port (
    CLK: in STD_LOGIC;
    RST: in STD_LOGIC;
    inp_0: in STD_LOGIC_VECTOR(3 downto 0);
    inp_1: in STD_LOGIC_VECTOR(3 downto 0);
    inp_2: in STD_LOGIC_VECTOR(3 downto 0);
    inp_3: in STD_LOGIC_VECTOR(3 downto 0);
    en_0: in STD_LOGIC;
    en_1: in STD_LOGIC;

```

```

en_2: in STD_LOGIC;
en_3: in STD_LOGIC;
out_0: out STD_LOGIC_VECTOR(1 downto 0);
out_1: out STD_LOGIC_VECTOR(1 downto 0);
out_2: out STD_LOGIC_VECTOR(1 downto 0);
out_3: out STD_LOGIC_VECTOR(1 downto 0)
);
end switch_omega_handshaking;

architecture Structural of switch_omega_handshaking is
component gestore_handshaking is
Port (
    CLK: in STD_LOGIC;
    RST: in STD_LOGIC;
    SRC: in STD_LOGIC_VECTOR(1 downto 0);
    MEX: in STD_LOGIC_VECTOR(3 downto 0);
    EN: in STD_LOGIC;
    RX: in STD_LOGIC_VECTOR(7 downto 0);
    TX_EN: out STD_LOGIC;
    TX: out STD_LOGIC_VECTOR(7 downto 0);
    dato_ricevuto: out STD_LOGIC_VECTOR(1 downto 0)
);
end component;

component switch_omega_network is
Port (
    en_in_0: in STD_LOGIC;
    en_in_1: in STD_LOGIC;
    en_in_2: in STD_LOGIC;
    en_in_3: in STD_LOGIC;
    destination_in_0: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_1: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_2: in STD_LOGIC_VECTOR(1 downto 0);
    destination_in_3: in STD_LOGIC_VECTOR(1 downto 0);
    inp_0: in STD_LOGIC_VECTOR(7 downto 0);
    inp_1: in STD_LOGIC_VECTOR(7 downto 0);
    inp_2: in STD_LOGIC_VECTOR(7 downto 0);
    inp_3: in STD_LOGIC_VECTOR(7 downto 0);
    out_0: out STD_LOGIC_VECTOR(7 downto 0);
    out_1: out STD_LOGIC_VECTOR(7 downto 0);
    out_2: out STD_LOGIC_VECTOR(7 downto 0);
    out_3: out STD_LOGIC_VECTOR(7 downto 0)
);
end component;

signal TX_OK_0: STD_LOGIC;
signal TX_OK_1: STD_LOGIC;

```

```

signal TX_OK_2: STD_LOGIC;
signal TX_OK_3: STD_LOGIC;
signal TX_0: STD_LOGIC_VECTOR(7 downto 0);
signal TX_1: STD_LOGIC_VECTOR(7 downto 0);
signal TX_2: STD_LOGIC_VECTOR(7 downto 0);
signal TX_3: STD_LOGIC_VECTOR(7 downto 0);
signal RX_0: STD_LOGIC_VECTOR(7 downto 0);
signal RX_1: STD_LOGIC_VECTOR(7 downto 0);
signal RX_2: STD_LOGIC_VECTOR(7 downto 0);
signal RX_3: STD_LOGIC_VECTOR(7 downto 0);

signal dest0 : std_logic_vector(1 downto 0);
signal dest1 : std_logic_vector(1 downto 0);
signal dest2 : std_logic_vector(1 downto 0);
signal dest3 : std_logic_vector(1 downto 0);

begin
    dest0 <= inp_0(3 downto 2);
    dest1 <= inp_1(3 downto 2);
    dest2 <= inp_2(3 downto 2);
    dest3 <= inp_3(3 downto 2);

gestore_switch_0: gestore_handshaking
port map(
    CLK => CLK,
    RST => RST,
    SRC => "00",
    MEX => inp_0,
    EN => EN_0,
    RX => RX_0,
    TX_en => TX_OK_0,
    TX => TX_0,
    dato_ricevuto => out_0
);
gestore_switch_1: gestore_handshaking
port map(
    CLK => CLK,
    RST => RST,
    SRC => "01",
    MEX => inp_1,
    EN => EN_1,
    RX => RX_1,
    TX_en => TX_OK_1,
    TX => TX_1,
    dato_ricevuto => out_1
);
gestore_switch_2: gestore_handshaking
port map(

```

```

CLK => CLK,
RST => RST,
SRC => "10",
MEX => inp_2,
EN => EN_2,
RX => RX_2,
TX_en => TX_OK_2,
TX => TX_2,
dato_ricevuto => out_2
);
gestore_switch_3: gestore_handshaking
port map(
CLK => CLK,
RST => RST,
SRC => "11",
MEX => inp_3,
EN => EN_3,
RX => RX_3,
TX_en => TX_OK_3,
TX => TX_3,
dato_ricevuto => out_3
);
omega: switch_omega_network
port map(
en_in_0 => TX_OK_0,
en_in_1 => TX_OK_1,
en_in_2 => TX_OK_2,
en_in_3 => TX_OK_3,
destination_in_0 => dest0,
destination_in_1 => dest1,
destination_in_2 => dest2,
destination_in_3 => dest3,
inp_0 => TX_0,
inp_1 => TX_1,
inp_2 => TX_2,
inp_3 => TX_3,
out_0 => RX_0,
out_1 => RX_1,
out_2 => RX_2,
out_3 => RX_3
);
end Structural;

```

11.13 Simulazione 3

La simulazione mostra il comportamento atteso in Figura 11.13. La scelta dei componenti a cui dare la possibilità di comunicare viene sempre fatta attraverso la rete di priorità, mentre l'instradamento dei vari pacchetti inviati avviene tramite l'handshaking descritto.

```
entity switch_omega_handshaking_tb is
end switch_omega_handshaking_tb;

architecture Behavioral of switch_omega_handshaking_tb is
component switch_omega_handshaking is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        inp_0: in STD_LOGIC_VECTOR(3 downto 0);
        inp_1: in STD_LOGIC_VECTOR(3 downto 0);
        inp_2: in STD_LOGIC_VECTOR(3 downto 0);
        inp_3: in STD_LOGIC_VECTOR(3 downto 0);
        en_0: in STD_LOGIC;
        en_1: in STD_LOGIC;
        en_2: in STD_LOGIC;
        en_3: in STD_LOGIC;
        out_0: out STD_LOGIC_VECTOR(1 downto 0);
        out_1: out STD_LOGIC_VECTOR(1 downto 0);
        out_2: out STD_LOGIC_VECTOR(1 downto 0);
        out_3: out STD_LOGIC_VECTOR(1 downto 0)
    );
end component;

--Inputs
signal CLK : std_logic := '0';
signal RST : std_logic := '0';
signal Input0 : std_logic_vector(3 downto 0) := (others => '0');
signal Input1 : std_logic_vector(3 downto 0) := (others => '0');
signal Input2 : std_logic_vector(3 downto 0) := (others => '0');
signal Input3 : std_logic_vector(3 downto 0) := (others => '0');
signal En0 : std_logic := '0';
signal En1 : std_logic := '0';
signal En2 : std_logic := '0';
signal En3 : std_logic := '0';

--Outputs
signal Output0 : std_logic_vector(1 downto 0);
signal Output1 : std_logic_vector(1 downto 0);
signal Output2 : std_logic_vector(1 downto 0);
signal Output3 : std_logic_vector(1 downto 0);
```

```

--Clock period definitions
constant CLK_period: time := 10ns;

begin
  uut: switch_omega_handshaking
    port map (
      CLK => CLK,
      RST => RST,
      inp_0 => Input0,
      inp_1 => Input1,
      inp_2 => Input2,
      inp_3 => Input3,
      en_0 => En0,
      en_1 => En1,
      en_2 => En2,
      en_3 => En3,
      out_0 => Output0,
      out_1 => Output1,
      out_2 => Output2,
      out_3 => Output3
    );
  );

CLK_process: process
begin
  CLK <= '0';
  wait for CLK_period/2;
  CLK <= '1';
  wait for CLK_period/2;
end process;

stim_proc: process
begin
  wait for 100ns;
  Input3 <= "0001"; --Switch 3 invia a switch 0 il dato "01"
  Input1 <= "1000"; --Switch 1 invia allo switch 2 il dato "00"
  wait for CLK_period;
  En3 <= '1';
  En1 <= '1';
  wait for 10*CLK_period;
  Input2 <= "1111"; --Switch 2 invia allo switch 3 il dato "11"
  Input0 <= "0110"; --Switch 0 invia allo switch 1 il dato "10"
  wait for CLK_period;
  En0 <= '1';
  En2 <= '1';
  wait for 10*CLK_period;
  En0 <= '0';
  En1 <= '0';
  En2 <= '0';
end process;

```

```

    En3 <= '0';
    wait;
end process;
end Behavioral;

```

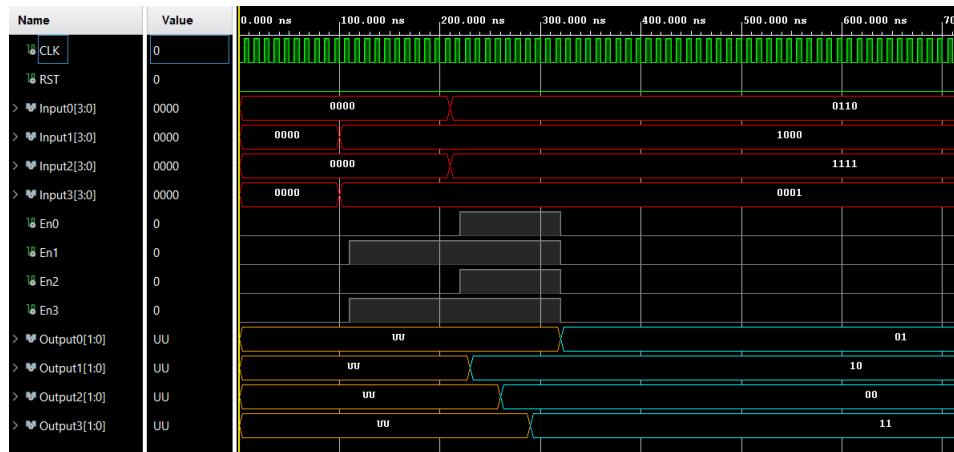


Figura 11.13: Simulazione Switch multistadio versione 3

Capitolo 12

Esercizio 11

12.1 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.

12.2 Soluzione

Per tale traccia si è scelto di implemetare il **Moltiplicatore di Booth**.

12.2.1 Cenni Teorici

L'**Algoritmo di Booth** è una procedura per il calcolo di una moltiplicazione tra due numeri binari con segno, codificati in complemento a 2.

La scoperta di tale algoritmo è dovuta all'omonimo matematico che, durante i suoi studi sulla cristallografia, effettuava dei calcoli con una calcolatrice lenta, ma molto performante nell'eseguire degli shift.

Alla base della teoria di Booth c'è l'omonima **codifica di Booth**: tale rappresentazione può essere facilmente ottenuta sostituendo ciascuna coppia di bit adiacenti della stringa $X_{j-1}X_j$ con un valore $\{1, 0, 1\}$ secondo la seguente logica:

- $X_{j-1}X_j = 00 \rightarrow 0$
- $X_{j-1}X_j = 11 \rightarrow 0$
- $X_{j-1}X_j = 01 \rightarrow 1$
- $X_{j-1}X_j = 10 \rightarrow -1$

Per capire da dove deriva tale codifica si propone un esempio: si consideri un valore X la cui rappresentazione in complemento a due sia $x_nx_{n-1}\dots x_1x_0$. Si va a definire un operando Y definito come segue:

- $y_0 = -x_0$
- $y_1 = -x_1 + x_0$
- $y_2 = -x_2 + x_1$
- ...
- $y_{n-1} = -x_{n-1} + x_{n-2}$

Se si suppone di moltiplicare per 2^0 il valore y_0 , per 2^1 il valore y_1 e così via, si ottiene la seguente espressione:

$$y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \dots + y_0 \cdot 2^0 = -x_{n-1} \cdot x^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_0 \cdot 2^0$$

Gli elementi del vettore Y appartengono all'insieme $\{-1; 0; 1\}$ e vanno a comporre la **rappresentazione di Booth-1** del valore X . Tale rappresentazione può essere ottenuta andando a valutare le coppie di bit adiacenti di X ottenendo quindi la logica precedentemente descritta.

Volendo fare un esempio numerico, in Tabella 12.1 è riportata la codifica del numero 10110011(0).

| | | | | | | | | | |
|-------------------|----|----|---|----|---|----|---|----|---|
| X | 1 | 0 | 1 | 1 | 0 | 0 | 1 | 1 | 0 |
| codifica di Booth | -1 | +1 | 0 | -1 | 0 | +1 | 0 | -1 | |
| | | | | | | | | | |

Tabella 12.1: Esempio di codifica di Booth di un numero

La codifica appena presentata è detta **Booth-1**, perchè i valori codificati dipendono da coppie di

2 bit. Un altro tipo di codifica viene detta **Booth-2**: tale codifica si basa sul raggruppamento di 3 bit alla volta, costruendo il numero codificato come segue:

$$Y = (-2^n) \cdot y_n + (2^{n-1}) \cdot y_{n-1} + \dots + (2^0) \cdot y_0$$

Volendo fare un esempio, se si suppone $n = 6$, si raggruppano i bit di Y a tre a tre, partendo dai bit più significativi, con l'ultimo termine in comune fra una terna e la successiva. Per rispettare la suddivisione in terne è necessario aggiungere, come nel caso di Booth-1, un bit pari a zero a destra della codifica.

A valle di tale raggruppamento, il vettore $Y = [y_5 y_4 y_3 y_2 y_1 y_0]$ può essere riscritto come segue:

$$\begin{aligned} Y &= [-2^5 y_5 + 2^4 y_4 + 2^3 y_3] + [-2^3 y_3 + 2^2 y_2 + 2^1 y_1] + [-2^1 y_1 + 2^0 y_0 + 0] \\ Y &= 2^4[-2y_5 + y_4 + y_3] + 2^2[-2y_3 + y_2 + y_1] + 2^0[-2y_1 + y_0 + 0] \end{aligned}$$

Le combinazioni di bit all'intendo dei gruppi da tre danno luogo alla codifica di Booth-2; in Tabella 12.2 è presente la codifica di tali bit secondo Booth-2. Si nota quindi che il vettore Y può essere

| y_{i-1} | y_i | y_{i+1} | Codifica Booth-2 |
|-----------|-------|-----------|------------------|
| 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | +1 |
| 0 | 1 | 0 | +1 |
| 0 | 1 | 1 | +2 |
| 1 | 0 | 0 | -2 |
| 1 | 0 | 1 | -1 |
| 1 | 1 | 0 | -1 |
| 1 | 1 | 1 | 0 |

Tabella 12.2: Tabella di codifica di Booth-2

scritto, identificando $C = \{-2; -1; 0; +1; +2\}$ come possibili codifiche del numero, come segue:

$$Y = 2^4 C_4 + 2^2 C_2 + 2^0 C_0$$

Tale codifica produce una versione di Y a tre bit piuttosto che a sei. A valle di tale codifica, il prodotto $Z = X \cdot Y$ è esprimibile tramite la seguente espressione:

$$Z = x \cdot Y = 2^4 X C_4 + 2^2 X C_2 + 2^0 X C_0$$

Il numero di prodotti parziali da sommare fra loro si è ridotto della metà. Il calcolo dei prodotti parziali è tuttavia più complesso (non è una semplice AND), perché dipende dal tipo di codifica che si è ottenuta sull'operando Y :

- $XC_i = 2X \rightarrow$ shift a sinistra di X di un bit ed aggiunta di uno 0 come bit meno significativo;

- $XC_i = X \rightarrow$ banale;
- $XC_i = 0 \rightarrow$ banale;
- $XC_i = -X \rightarrow$ si complementano i bit di X e si somma un 1 meno significativo;
- $XC_i = -2X \rightarrow$ si complementano i bit di X, si somma un 1 meno significativo, si effettua uno shift a sinistra del risultato, si aggiunge uno 0 come bit meno significativo.

12.3 Schematici

L'architettura del moltiplicatore di Booth è mostrata in Figura 12.1. L'unità operativa di tale

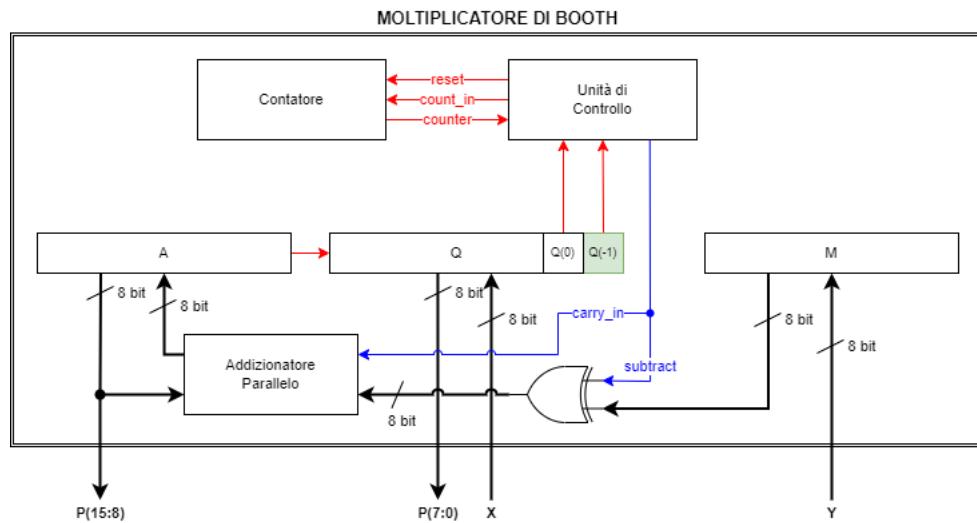


Figura 12.1: Architettura interna del moltiplicatore di Booth

macchina si compone di differenti elementi elencati di seguito:

- Gli operandi X e Y vengono conservati in dei **registri a scorrimento**, rispettivamente indicati come Q ed M all'interno dell'architettura. L'architettura necessita di un ulteriore registro, chiamato A , per il calcolo delle somme parziali.
- Un **Adder parallelo** si occupa di effettuare le somme parziali tra l'operando M e la somma parziale al passo precedente. Tale addizionatore riceve un carry che è in XOR con un segnale proveniente dall'unità di controllo, alto nel caso in cui sia necessario effettuare una sottrazione piuttosto che una somma.
- Un **contatore** si occupa di temporizzare l'unità di controllo affinché sia in grado di sapere quale sia l'indice di somma o sottrazione da eseguire.

L'**unità di controllo** regola le operazioni parziali, discriminando tra somma o differenza inviando il corretto bit di **subtract**.

Tale architettura è in grado di eseguire pienamente l'algoritmo di Booth, già precedentemente descritto nell'automa in Figura 8.4.

Traducendo tale algoritmo in un diagramma a stati finiti, che andrà successivamente codificato affinchè si possa descrivere il comportamento dell'unità di controllo, si ottiene l'automa in Figura 12.2.

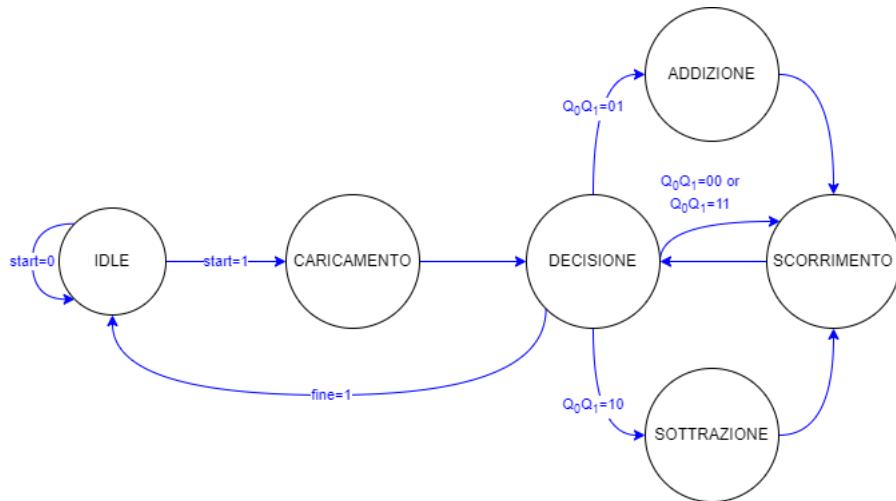


Figura 12.2: Automa dell'algoritmo di Booth

12.4 Codice

12.4.1 Unità operativa

Shift Register

Per l'implementazione dello shift register si è scelto di definire un unico componente che si occupasse di entrambi i registri contenenti i due operandi.

```

entity shift_register is
  Port (
    CLK: in STD_LOGIC;
    RST: in STD_LOGIC;
    SFT: in STD_LOGIC;
    LOAD_in_0: in STD_LOGIC;
    LOAD_in_1: in STD_LOGIC;
    in_0: in STD_LOGIC_VECTOR(7 downto 0);
    in_1: in STD_LOGIC_VECTOR(7 downto 0);
    check_0: out STD_LOGIC;
    check_1: out STD_LOGIC;
    out_0: out STD_LOGIC_VECTOR(7 downto 0);
    out_1: out STD_LOGIC_VECTOR(7 downto 0)
  );
  
```

```

end shift_register;

architecture Behavioral of shift_register is
    signal T: STD_LOGIC_VECTOR(16 downto 0);
begin
    reg: process(CLK, RST)
    begin
        if(RST='1') then
            T <= (others => '0');
        elsif(CLK'event and CLK='0') then
            if(SFT='1') then
                T(16) <= T(16);
                for i in 16 downto 1 loop
                    T(i-1) <= T(i);
                end loop;
            end if;
            if(LOAD_in_0='1') then
                T(16 downto 9) <= in_0(7 downto 0);
            end if;
            if(LOAD_in_1='1') then
                T(8 downto 1) <= in_1(7 downto 0);
            end if;
        end if;
    end process;

    out_0 <= T(16 downto 9);
    out_1 <= T(8 downto 1);
    check_0 <= T(0);
    check_1 <= T(1);
end Behavioral;

```

Full Adder

```

entity full_adder is
    Port(
        in_0: in STD_LOGIC;
        in_1: in STD_LOGIC;
        c_in: in STD_LOGIC;
        c_out: out STD_LOGIC;
        sum: out STD_LOGIC
    );
end full_adder;

architecture Dataflow of full_adder is
begin
    sum <= in_0 XOR in_1 XOR c_in;
    c_out <= (in_0 AND in_1) OR (c_in AND (in_0 XOR in_1));

```

```
end Dataflow;
```

Ripple Carry Adder con subtract

```
entity ripple_carry_adder is
  Port (
    in_0: in STD_LOGIC_VECTOR(7 downto 0);
    in_1: in STD_LOGIC_VECTOR(7 downto 0);
    CIN: in STD_LOGIC;
    SUM: out STD_LOGIC_VECTOR(7 downto 0)
  );
end ripple_carry_adder;

architecture Structural of ripple_carry_adder is
  component full_adder
    Port(
      in_0: in STD_LOGIC;
      in_1: in STD_LOGIC;
      c_in: in STD_LOGIC;
      c_out: out STD_LOGIC;
      sum: out STD_LOGIC
    );
  end component;

  signal complemento: STD_LOGIC_VECTOR(7 downto 0);
  signal carry: STD_LOGIC_VECTOR(7 downto 0);
begin
  complemento_y: FOR i in 0 to 7 GENERATE
    complemento(i) <= in_1(i) XOR CIN;
  END GENERATE;
  first: full_adder
    port map(
      in_0 => in_0(0),
      in_1 => complemento(0),
      c_in => CIN,
      c_out => carry(0),
      sum => SUM(0)
    );
  addizionatore: FOR i in 1 to 7 GENERATE
    fa_1_7: full_adder
      port map(
        in_0 => in_0(i),
        in_1 => complemento(i),
        c_in => carry(i-1),
        c_out => carry(i),
        sum => SUM(i)
      );
  
```

```

        END GENERATE;
end Structural;

```

Componente Unità operativa totale

```

entity unita_operativa is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        SFT: in STD_LOGIC;
        CIN: in STD_LOGIC;
        in_0: in STD_LOGIC_VECTOR(7 downto 0);
        in_1: in STD_LOGIC_VECTOR(7 downto 0);
        LOAD_in_0: in STD_LOGIC;
        LOAD_in_1: in STD_LOGIC;
        output: out STD_LOGIC_VECTOR(15 downto 0);
        check_0: out STD_LOGIC;
        check_1: out STD_LOGIC
    );
end unita_operativa;

architecture Structural of unita_operativa is
    component ripple_carry_adder is
        Port (
            in_0: in STD_LOGIC_VECTOR(7 downto 0);
            in_1: in STD_LOGIC_VECTOR(7 downto 0);
            CIN: in STD_LOGIC;
            SUM: out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;
    component shift_register is
        Port (
            CLK: in STD_LOGIC;
            RST: in STD_LOGIC;
            SFT: in STD_LOGIC;
            LOAD_in_0: in STD_LOGIC;
            LOAD_in_1: in STD_LOGIC;
            in_0: in STD_LOGIC_VECTOR(7 downto 0);
            in_1: in STD_LOGIC_VECTOR(7 downto 0);
            check_0: out STD_LOGIC;
            check_1: out STD_LOGIC;
            out_0: out STD_LOGIC_VECTOR(7 downto 0);
            out_1: out STD_LOGIC_VECTOR(7 downto 0)
        );
    end component;
    signal somma: STD_LOGIC_VECTOR(7 downto 0);
    signal prodotto_0: STD_LOGIC_VECTOR(7 downto 0);

```

```

    signal prodotto_1: STD_LOGIC_VECTOR(7 downto 0);
begin
    sr: shift_register
        port map (
            CLK => CLK,
            RST => RST,
            SFT => SFT,
            LOAD_in_0 => LOAD_in_0,
            LOAD_in_1 => LOAD_in_1,
            in_0 => somma,
            in_1 => in_1,
            check_0 => check_0,
            check_1 => check_1,
            out_0 => prodotto_0,
            out_1 => prodotto_1
        );
    rca: ripple_carry_adder
        port map(
            in_0 => prodotto_0,
            in_1 => in_0,
            CIN => CIN,
            SUM => somma
        );
    output <= prodotto_0 & prodotto_1;
end Structural;

```

12.4.2 Unità di controllo

```

entity unita_controllo is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        START: in STD_LOGIC;
        FINE: in STD_LOGIC;
        Q0: in STD_LOGIC;
        Q1: in STD_LOGIC;
        SFT: out STD_LOGIC;
        SUBTRACT: out STD_LOGIC;
        LOAD_in_0: out STD_LOGIC;
        LOAD_in_1: out STD_LOGIC;
        COUNT: out STD_LOGIC
    );
end unita_controllo;

architecture Behavioral of unita_controllo is
    type stato is (IDLE, CARIMENTO, DECISIONE, SOTTRAZIONE, ADDIZIONE, SCORRIMENTO);
    signal stato_corrente: stato;

```

```

begin
    controllo: process(CLK, RST)
    begin
        if(RST='1') then
            stato_corrente <= IDLE;
        elsif(CLK'event and CLK='1') then
            case stato_corrente is
                when IDLE =>
                    SUBTRACT <= '0';
                    LOAD_in_0 <= '0';
                    LOAD_in_1 <= '0';
                    SFT <= '0';
                    COUNT <= '0';
                    if(START='1') then
                        stato_corrente <= CARICAMENTO;
                    else
                        stato_corrente <= IDLE;
                    end if;
                when CARICAMENTO =>
                    LOAD_in_1 <= '1';
                    stato_corrente <= DECISIONE;
                when DECISIONE =>
                    COUNT <= '0';
                    SFT <= '0';
                    LOAD_in_1 <= '0';
                    if(FINE='1') then
                        stato_corrente <= IDLE;
                    elsif(Q0='0' and Q1='1') then
                        stato_corrente <= SOTTRAZIONE;
                    elsif(Q0='1' and Q1='0') then
                        stato_corrente <= ADDIZIONE;
                    else
                        stato_corrente <= SCORRIMENTO;
                    end if;
                when SOTTRAZIONE =>
                    LOAD_in_0 <= '1';
                    SUBTRACT <= '1';
                    stato_corrente <= SCORRIMENTO;
                when ADDIZIONE =>
                    LOAD_in_0 <= '1';
                    SUBTRACT <= '0';
                    stato_corrente <= SCORRIMENTO;
                when SCORRIMENTO =>
                    LOAD_in_0 <= '0';
                    SUBTRACT <= '0';
                    COUNT <= '1';
                    SFT <= '1';
            end case;
        end if;
    end process;
end;

```

```

        stato_corrente <= DECISIONE;
    end case;
end if;
end process;
end Behavioral;
```

12.4.3 Contatore

```

entity contatore is
  Port (
    CLK: in STD_LOGIC;
    RST: in STD_LOGIC;
    start_count: in STD_LOGIC;
    end_count: out STD_LOGIC
  );
end contatore;

architecture Behavioral of contatore is
  signal k: STD_LOGIC_VECTOR(2 downto 0);
begin
  contatore_modulo_8: process(CLK, RST)
  begin
    if(RST='1') then
      k <= (others => '0');
      end_count <= '0';
    elsif(CLK'event and CLK='0' and start_count='1') then
      k <= STD_LOGIC_VECTOR(unsigned(k)+1);
      if(k="111") then
        end_count <= '1';
      else
        end_count <= '0';
      end if;
    end if;
  end process;
end Behavioral;
```

12.4.4 Gestore Button

Di seguito si riporta anche il componente **ButtonDebouncer**, utilizzato durante la fase di sintesi su scheda. Tale componente si occupa di effettuare una "pulizia" del segnale in ingresso ad esso, al fine di evitare che la pressione del tasto provochi malfunzionamenti o anche semplicemente trasmissioni troppo rapide senza un effettivo riscontro visivo in output da parte dell'utente del sistema.

```

entity ButtonDebouncer is
  generic (
```

```

CLK_period:  integer := 10;  -- periodo del clock della board 10 nanosecondi
btn_noise_time:  integer := 650000000 --intervallo di tempo in cui si ha l'oscillazione del bottone
--assumo che duri 6.5ms=6500microsec=6500000ns
);

Port (
    RST : in STD_LOGIC;
    CLK : in STD_LOGIC;
    BTN : in STD_LOGIC;
    CLEARED_BTN : out STD_LOGIC
);

end ButtonDebouncer;

architecture Behavioral of ButtonDebouncer is

type stato is (NOT_PRESSED, PRESSED);
signal BTN_state : stato := NOT_PRESSED;
constant max_count : integer := btn_noise_time/CLK_period; -- 650000000/10= conto 650000 colpi di clock

begin
deb: process (CLK)
variable count: integer := 0;
begin
if rising_edge(CLK) then
    if( RST = '1') then
        BTN_state <= NOT_PRESSED;
        CLEARED_BTN <= '0';
    else
        case BTN_state is
            when NOT_PRESSED =>
                CLEARED_BTN<= '0';
                if( BTN = '1' ) then
                    BTN_state <= PRESSED;
                else
                    BTN_state <= NOT_PRESSED;
                end if;
            when PRESSED =>
                if(count = max_count -1) then
                    count:=0;
                    CLEARED_BTN <= '1';
                    BTN_state <= NOT_PRESSED;
                else
                    count:= count+1;
                    BTN_state <= PRESSED;
                end if;
            when others =>
                BTN_state <= NOT_PRESSED;
        end case;
    end if;
end process;
end;

```

```

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

12.4.5 Componente Generale

```

entity moltiplicatore_booth is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        START: in STD_LOGIC;
        in_0: in STD_LOGIC_VECTOR(7 downto 0);
        in_1: in STD_LOGIC_VECTOR(7 downto 0);
        output: out STD_LOGIC_VECTOR(15 downto 0)
    );
end moltiplicatore_booth;

architecture Structural of moltiplicatore_booth is
component unita_controllo is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        START: in STD_LOGIC;
        FINE: in STD_LOGIC;
        Q0: in STD_LOGIC;
        Q1: in STD_LOGIC;
        SFT: out STD_LOGIC;
        SUBTRACT: out STD_LOGIC;
        LOAD_in_0: out STD_LOGIC;
        LOAD_in_1: out STD_LOGIC;
        COUNT: out STD_LOGIC
    );
end component;
component unita_operativa is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        SFT: in STD_LOGIC;
        CIN: in STD_LOGIC;
        in_0: in STD_LOGIC_VECTOR(7 downto 0);
        in_1: in STD_LOGIC_VECTOR(7 downto 0);
        LOAD_in_0: in STD_LOGIC;
        LOAD_in_1: in STD_LOGIC;
        output: out STD_LOGIC_VECTOR(15 downto 0);
        check_0: out STD_LOGIC;
        check_1: out STD_LOGIC
    );
end component;
```

```

    );
end component;
component contatore is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        start_count: in STD_LOGIC;
        end_count: out STD_LOGIC
    );
end component;

-- -----
-- COMPONENTE PER LA SINTESI SU SCHEDA
-- -----
-- component ButtonDebouncer is
--     Port (
--         RST : in STD_LOGIC;
--         CLK : in STD_LOGIC;
--         BTN : in STD_LOGIC;
--         CLEARED_BTN : out STD_LOGIC
--     );
-- end component;
-- -----


signal end_count: STD_LOGIC;
signal k0: STD_LOGIC;
signal k1: STD_LOGIC;
signal sft: STD_LOGIC;
signal subtract: STD_LOGIC;
signal start_count: STD_LOGIC;
signal load_in_0: STD_LOGIC;
signal load_in_1: STD_LOGIC;

-- -----
-- SEGNALI INTERNI PER LA SINTESI SU SCHEDA
-- -----
-- signal cleared_input: STD_LOGIC;
-- signal reset_in: STD_LOGIC;
-- -----


begin

    --reset_in <= not RST;  (SCHEDA)
    --
    -- -----
    -- COMPONENTE PER LA SINTESI SU SCHEDA
    -- -----
    -- cleared_in: ButtonDebouncer
    --     port map (

```

```

--          RST => reset_in,
--          CLK => CLK,
--          BTN => START,
--          CLEARED_BTN => cleared_input
--      );
-- -----
UC: unita_controllo
port map(
    CLK => CLK,
    --RST => reset_in, (SCHEDA)
    RST => RST, --(SIMULAZIONE)
    --START => cleared_input, (SCHEDA)
    START => START, --(SIMULAZIONE)
    FINE => end_count,
    Q0 => k0,
    Q1 => k1,
    SFT => sft,
    SUBTRACT => subtract,
    LOAD_in_0 => load_in_0,
    LOAD_in_1 => load_in_1,
    COUNT => start_count
);
C: contatore
port map(
    CLK => CLK,
    --RST => reset_in, (SCHEDA)
    RST => RST, --(SIMULAZIONE)
    start_count => start_count,
    end_count => end_count
);
UO: unita_operativa
port map(
    CLK => CLK,
    --RST => reset_in, (SCHEDA)
    RST => RST, --(SIMULAZIONE)
    SFT => sft,
    CIN => subtract,
    in_0 => in_0,
    in_1 => in_1,
    LOAD_in_0 => load_in_0,
    LOAD_in_1 => load_in_1,
    output => output,
    check_0 => k0,
    check_1 => k1
);
end Structural;

```

12.5 Simulazione

Per effettuare la simulazione funzionale di tale componente è stato creato il testbench riportato di seguito che, a valle della sua esecuzione, ha portato alla definizione delle forme d'onda successivamente.

```
entity moltiplicatore_booth_tb is
end moltiplicatore_booth_tb;

architecture Behavioral of moltiplicatore_booth_tb is
component moltiplicatore_booth is
    Port (
        CLK: in STD_LOGIC;
        RST: in STD_LOGIC;
        START: in STD_LOGIC;
        in_0: in STD_LOGIC_VECTOR(7 downto 0);
        in_1: in STD_LOGIC_VECTOR(7 downto 0);
        output: out STD_LOGIC_VECTOR(15 downto 0)
    );
end component;
--Inputs
signal CLK : STD_LOGIC := '0';
signal RST : STD_LOGIC := '0';
signal START : STD_LOGIC := '0';
signal in_0 : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
signal in_1 : STD_LOGIC_VECTOR(7 downto 0) := (others => '0');
--Outputs
signal output: STD_LOGIC_VECTOR(15 downto 0) := (others => '0');

constant CLK_period: time := 10ns;
begin
    uut: moltiplicatore_booth
    port map(
        CLK => CLK,
        RST => RST,
        START => START,
        in_0 => in_0,
        in_1 => in_1,
        output => output
    );
    CLK_process: process
    begin
        CLK <= '0';
        wait for CLK_period/2;
        CLK <= '1';
        wait for CLK_period/2;
    end process;
```

```

stim_proc: process
begin
    wait for 100ns;
    RST <= '1';
    wait for 50ns;
    RST <= '0';
    wait for 50ns;
    START <= '1';
    in_0 <= "00000101"; -- +5
    -- in_0 <= "10001010"; -- -118
    -- in_0 <= "00001011"; -- +11
    in_1 <= "10000101"; -- -123
    -- in_1 <= "10010010"; -- -110
    -- in_1 <= "00010101"; -- +21
    wait for 50ns;
    START <= '0';
    wait;
end process;
end Behavioral;

```

In Figura 12.3 si nota la simulazione della moltiplicazione tra un numero positivo ed un numero negativo. In particolare si ha $+5 \cdot (-123) = -615$.

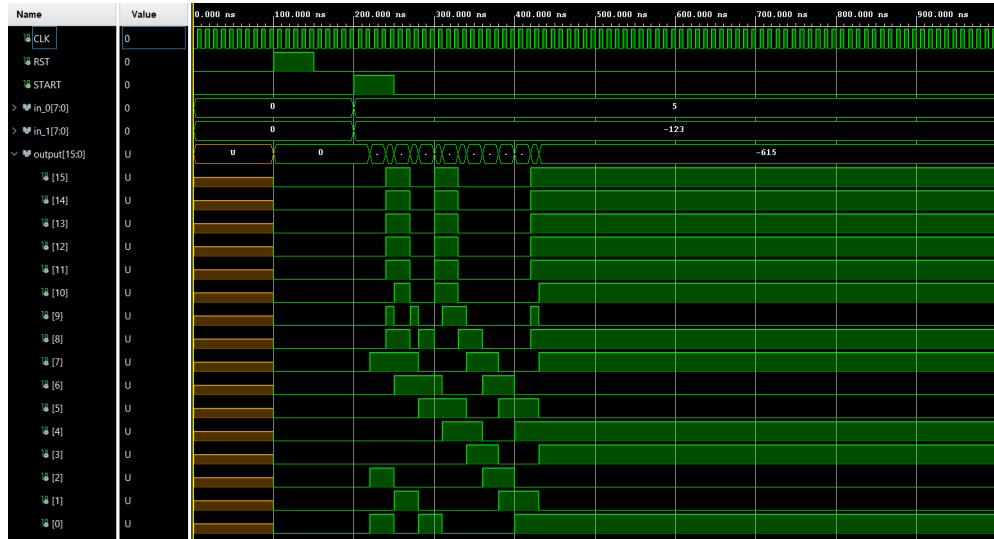


Figura 12.3: Simulazione del moltiplicatore di Booth - in0 positivo e in1 negativo

In Figura 12.4 si nota la simulazione della moltiplicazione tra un numero negativo ed un numero negativo. In particolare si ha $(-118) \cdot (-110) = +12980$.



Figura 12.4: Simulazione del moltiplicatore di Booth - in0 negativo e in1 negativo

In Figura 12.5 si nota la simulazione della moltiplicazione tra un numero positivo ed un numero positivo. In particolare si ha $+11 \cdot +21 = +231$.

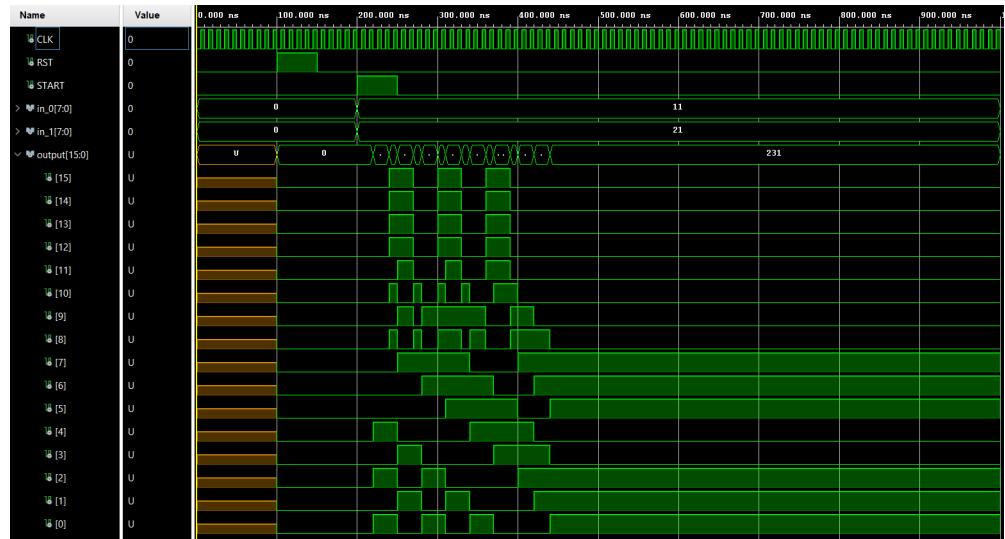


Figura 12.5: Simulazione del moltiplicatore di Booth - in0 positivo e in1 negativo

12.6 Sintesi su FPGA

Per la sintesi su scheda, si è scelto di mappare i due operandi della moltiplicazione con gli switch presenti sulla scheda, e di mostrare il risultato in output tramite i LED. Di seguito viene riportato il mapping effettuato tramite il file di constraints della scheda in dotazione (Nexys A7-100T).

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

##Switches
set_property -dict { PACKAGE_PIN J15      IOSTANDARD LVCMOS33 } [get_ports { in_1[0] }];
set_property -dict { PACKAGE_PIN L16      IOSTANDARD LVCMOS33 } [get_ports { in_1[1] }];
set_property -dict { PACKAGE_PIN M13      IOSTANDARD LVCMOS33 } [get_ports { in_1[2] }];
set_property -dict { PACKAGE_PIN R15      IOSTANDARD LVCMOS33 } [get_ports { in_1[3] }];
set_property -dict { PACKAGE_PIN R17      IOSTANDARD LVCMOS33 } [get_ports { in_1[4] }];
set_property -dict { PACKAGE_PIN T18      IOSTANDARD LVCMOS33 } [get_ports { in_1[5] }];
set_property -dict { PACKAGE_PIN U18      IOSTANDARD LVCMOS33 } [get_ports { in_1[6] }];
set_property -dict { PACKAGE_PIN R13      IOSTANDARD LVCMOS33 } [get_ports { in_1[7] }];
set_property -dict { PACKAGE_PIN T8       IOSTANDARD LVCMOS33 } [get_ports { in_0[0] }];
set_property -dict { PACKAGE_PIN U8       IOSTANDARD LVCMOS33 } [get_ports { in_0[1] }];
set_property -dict { PACKAGE_PIN R16      IOSTANDARD LVCMOS33 } [get_ports { in_0[2] }];
set_property -dict { PACKAGE_PIN T13      IOSTANDARD LVCMOS33 } [get_ports { in_0[3] }];
set_property -dict { PACKAGE_PIN H6       IOSTANDARD LVCMOS33 } [get_ports { in_0[4] }];
set_property -dict { PACKAGE_PIN U12      IOSTANDARD LVCMOS33 } [get_ports { in_0[5] }];
set_property -dict { PACKAGE_PIN U11      IOSTANDARD LVCMOS33 } [get_ports { in_0[6] }];
set_property -dict { PACKAGE_PIN V10     IOSTANDARD LVCMOS33 } [get_ports { in_0[7] }];

## LEDs
set_property -dict { PACKAGE_PIN H17      IOSTANDARD LVCMOS33 } [get_ports { output[0] }];
set_property -dict { PACKAGE_PIN K15      IOSTANDARD LVCMOS33 } [get_ports { output[1] }];
set_property -dict { PACKAGE_PIN J13      IOSTANDARD LVCMOS33 } [get_ports { output[2] }];
set_property -dict { PACKAGE_PIN N14      IOSTANDARD LVCMOS33 } [get_ports { output[3] }];
set_property -dict { PACKAGE_PIN R18      IOSTANDARD LVCMOS33 } [get_ports { output[4] }];
set_property -dict { PACKAGE_PIN V17      IOSTANDARD LVCMOS33 } [get_ports { output[5] }];
set_property -dict { PACKAGE_PIN U17      IOSTANDARD LVCMOS33 } [get_ports { output[6] }];
set_property -dict { PACKAGE_PIN U16      IOSTANDARD LVCMOS33 } [get_ports { output[7] }];
set_property -dict { PACKAGE_PIN V16      IOSTANDARD LVCMOS33 } [get_ports { output[8] }];
set_property -dict { PACKAGE_PIN T15      IOSTANDARD LVCMOS33 } [get_ports { output[9] }];
set_property -dict { PACKAGE_PIN U14      IOSTANDARD LVCMOS33 } [get_ports { output[10] }];
set_property -dict { PACKAGE_PIN T16      IOSTANDARD LVCMOS33 } [get_ports { output[11] }];
set_property -dict { PACKAGE_PIN V15      IOSTANDARD LVCMOS33 } [get_ports { output[12] }];
set_property -dict { PACKAGE_PIN V14      IOSTANDARD LVCMOS33 } [get_ports { output[13] }];
set_property -dict { PACKAGE_PIN V12      IOSTANDARD LVCMOS33 } [get_ports { output[14] }];
set_property -dict { PACKAGE_PIN V11      IOSTANDARD LVCMOS33 } [get_ports { output[15] }];
```

[...]

```
##Buttons  
  
set_property -dict { PACKAGE_PIN C12    IOSTANDARD LVCMOS33 } [get_ports { RST }];  
set_property -dict { PACKAGE_PIN N17    IOSTANDARD LVCMOS33 } [get_ports { START }];
```

Si è quindi proceduto a sottoporre alla macchina gli stessi ingressi riportati in fase di simulazione tramite switch, ottenendo i seguenti risultati:

- $(+5) \cdot (-123) = -615 \rightarrow 00000101 * 10000101 = 1111110110011001$. Il risultato è mostrato in Figura 12.6.
 - $(-118) \cdot (-110) = +12980 \rightarrow 10001010 * 10010010 = 0011001010110100$. Il risultato è mostrato in Figura 12.7.
 - $(+11) \cdot (+21) = +231 \rightarrow 00001011 * 00010101 = 0000000011100111$. Il risultato è mostrato in Figura 12.8.

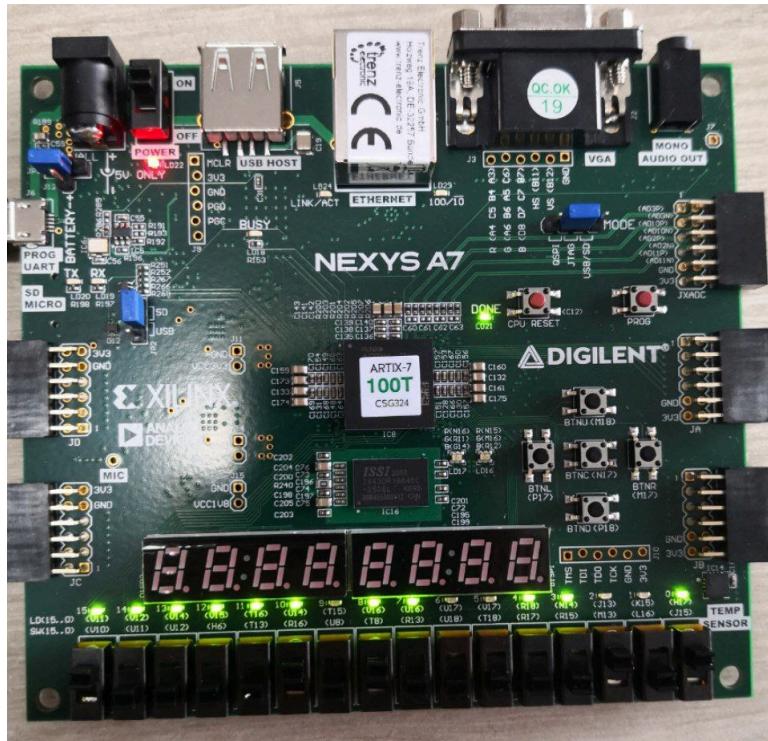


Figura 12.6: Sintesi su scheda del moltiplicatore di Booth - positivo per negativo

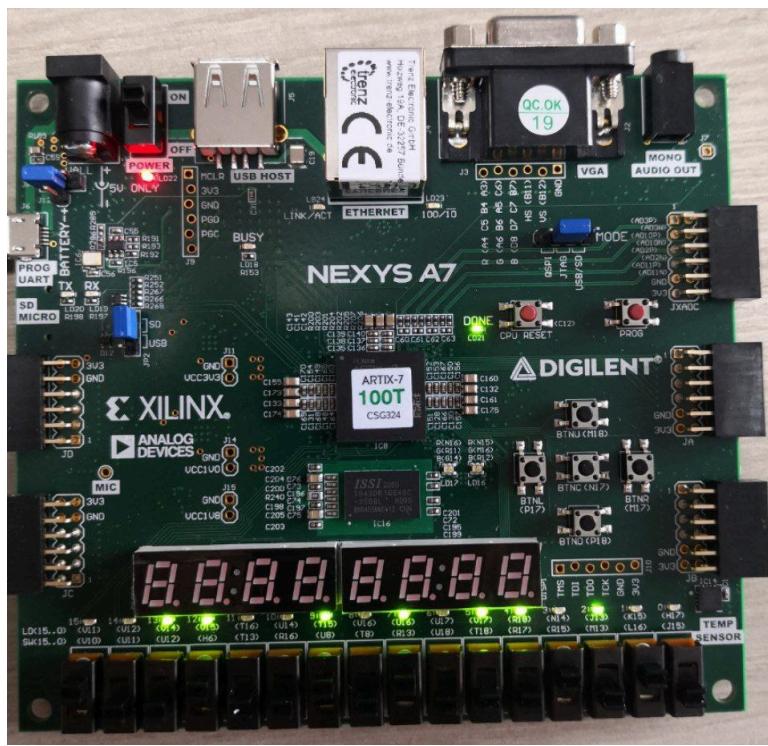


Figura 12.7: Sintesi su scheda del moltiplicatore di Booth - negativo per negativo

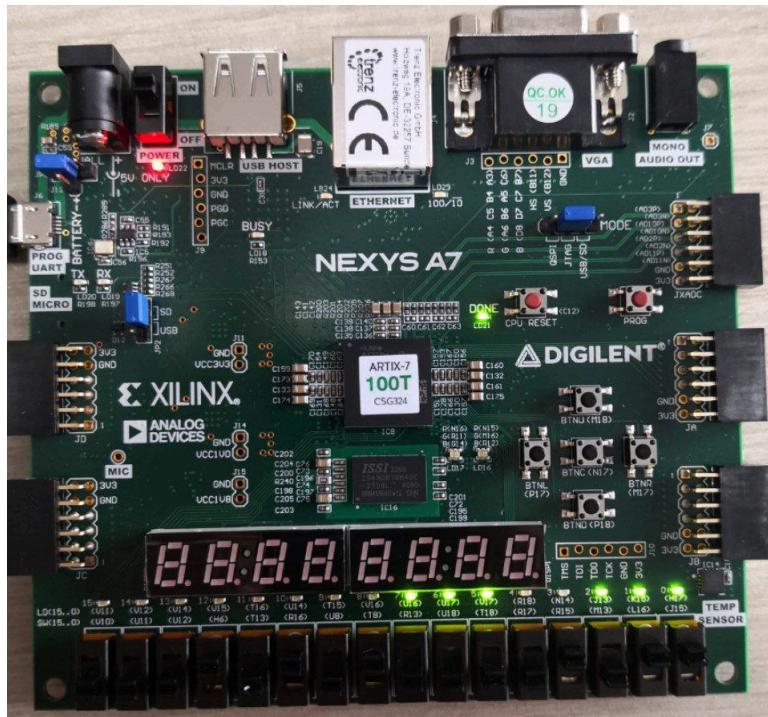


Figura 12.8: Sintesi su scheda del moltiplicatore di Booth - positivo per positivo