



UNIVERSITÀ DEGLI STUDI DI NAPOLI FEDERICO II

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

Elaborato del corso di Architettura dei Sistemi Digitali

Anno Accademico 2022/23

Studenti:
Martedì Gaetano M63/1226
Salzillo Biagio M63/1227

Indice

1	Multiplexer	14
1.1	Traccia	14
1.2	Soluzione	15
1.3	Codice	19
1.3.1	Multiplexer 4:1	19
1.3.2	Multiplexer 16:1	19
1.3.3	Demultiplexer 1:4	20
1.3.4	Rete di interconnessione	21
1.3.5	Rete di interconnessione on-board	22
1.4	Simulazione	24
1.4.1	Multiplexer 4:1	24
1.4.2	Multiplexer 16:1	25
1.4.3	Demultiplexer 1:4	27
1.4.4	Rete di interconnessione	28
1.4.5	Rete di interconnessione on-board	29
1.5	Sintesi	33
2	Encoder BCD	34
2.1	Traccia	34
2.2	Soluzione	35
2.3	Codice	39
2.3.1	Encoder BCD	39
2.3.2	Arbitro	39
2.3.3	Priority encoder BCD	40
2.3.4	Gestore dei catodi	41
2.3.5	Encoder BCD on-board	42
2.4	Simulazione	44
2.4.1	Encoder BCD	44
2.4.2	Arbitro	45
2.4.3	Priority encoder BCD	47
2.4.4	Cathodes manager	49
2.4.5	Encoder BCD on-board	50
2.5	Sintesi	54

3 Riconoscitore di sequenze	55
3.1 Traccia	55
3.2 Soluzione	56
3.3 Codice	60
3.3.1 Riconoscitore	60
3.3.2 Button debouncer	63
3.3.3 Riconoscitore di sequenze on-board	65
3.4 Simulazione	67
3.4.1 Riconoscitore di sequenze	67
3.5 Sintesi	72
4 Shift register	73
4.1 Traccia	73
4.2 Soluzione	74
4.3 Codice	77
4.3.1 Flip-Flop D	77
4.3.2 Approccio strutturale	77
4.3.3 Approccio comportamentale	79
4.4 Simulazione	82
4.4.1 Approccio strutturale	82
4.4.2 Approccio comportamentale	84
5 Cronometro	87
5.1 Traccia	87
5.2 Soluzione	88
5.2.1 Unità operativa	90
5.2.2 Unità di controllo	92
5.2.3 Display a sette segmenti	93
5.3 Codice	97
5.3.1 Contatore	97
5.3.2 Cronometro	98
5.3.3 Convertitore BCD	100
5.3.4 Memoria	101
5.3.5 Unità operativa	102
5.3.6 Unità di controllo	105
5.3.7 Sistema complessivo	109
5.3.8 Cronometro on-board	111
5.4 Simulazione	115
5.4.1 Cronometro	115
5.4.2 Sistema complessivo	117
5.5 Sintesi	120

6 Sistema di testing	121
6.1 Traccia	121
6.2 Soluzione	122
6.3 Codice	126
6.3.1 Memoria ROM	126
6.3.2 Macchina combinatoria	127
6.3.3 Contatore	128
6.3.4 Unità operativa	129
6.3.5 Unità di controllo	130
6.3.6 Sistema di testing on-board	132
6.4 Simulazione	134
6.5 Sintesi	136
7 Comunicazione con handshaking	137
7.1 Traccia	137
7.2 Soluzione	138
7.2.1 Base dei tempi	140
7.2.2 Nodo A	140
7.2.3 Nodo B	142
7.3 Codice	147
7.3.1 Base dei tempi	147
7.3.2 Nodo A	148
7.3.3 Nodo B	154
7.3.4 Sistema complessivo	160
7.4 Simulazione	164
7.4.1 Base dei tempi	164
7.4.2 Nodo A	165
7.4.3 Nodo B	171
7.4.4 Sistema complessivo	177
8 Processore Mic-1	181
8.1 Traccia	181
8.2 La struttura del processore	182
8.2.1 Il datapath	182
8.2.2 L'unità di controllo	186
8.2.3 Protocollo con la memoria	189
8.3 Simulazione	191
8.4 ILOAD	195
8.5 IF_ICMPEQ	198
8.6 Shift Left Logic	205

9 Interfaccia seriale	208
9.1 Traccia	208
9.2 Soluzione	209
9.2.1 Esercizio 9.1	211
9.2.2 Esercizio 9.2	212
9.3 Codice	217
9.3.1 Esercizio 9.1	217
9.3.2 Esercizio 9.2	220
9.4 Simulazione	231
9.4.1 Testbench esercizio 9.1	231
9.4.2 Testbench esercizio 9.2	232
10 Switch multistadio	236
10.1 Traccia	236
10.2 Soluzione	237
10.3 Codice	241
10.3.1 Switch	241
10.3.2 Unità operativa	241
10.3.3 Arbitro	243
10.3.4 Unità di controllo	244
10.3.5 Switch multistadio	245
10.4 Simulazione	247
10.4.1 Switch	247
10.4.2 Unità operativa	248
10.4.3 Switch multistadio	250
11 Moltiplicatore di Booth	252
11.1 Traccia	252
11.2 Soluzione	253
11.2.1 Adder/Subtractor	257
11.3 Codice	259
11.3.1 Moltiplicatore di Booth	259
11.3.2 Adder/Subtractor	263
11.4 Simulazione	266
11.4.1 Adder/Subtractor	266
11.4.2 Moltiplicatore di Booth: unità di controllo	269
11.4.3 Moltiplicatore di Booth	272
11.5 Timing analysis	275
11.6 Sintesi	278
12 Esercizio libero	280
12.1 Traccia	280
12.2 Soluzione	282
12.2.1 Nodo A	285

12.2.2	Nodo B	286
12.2.3	Nodo C	286
12.3	Codice	290
12.3.1	Comparatore di un bit	290
12.3.2	Comparatore di word	290
12.3.3	Encoder	291
12.3.4	Nodo A: unità operativa	292
12.3.5	Nodo A: unità di controllo	293
12.3.6	Nodo A	295
12.3.7	Nodo B: unità operativa	297
12.3.8	Nodo B: unità di controllo	298
12.3.9	Nodo B	300
12.3.10	Nodo C: unità operativa	301
12.3.11	Nodo C: unità di controllo	304
12.3.12	Nodo C	306
12.3.13	Sistema complessivo	307
12.3.14	Sistema complessivo on-board	308
12.4	Simulazione	311
12.4.1	Testbench comparatore di word	311
12.4.2	Testbench sistema complessivo	313
12.5	Sintesi	317

Elenco delle figure

1.1	Schematico multiplexer 4:1	15
1.2	Schematico multiplexer 16:1	16
1.3	Schematico demultiplexer 1:4	16
1.4	Schematico rete di interconnessione a 16 ingressi e 4 uscite . .	17
1.5	Schematico rete di interconnessione on-board	18
1.6	Waveform multiplexer 4:1	25
1.7	Waveform multiplexer 16:1	26
1.8	Waveform demultiplexer 1:4	28
1.9	Waveform rete di interconnessione	29
1.10	Waveform rete di interconnessione on-board	32
1.11	Schema di mapping della rete di interconnessione su board . .	33
2.1	Schematico priority encoder BCD	35
2.2	Schematico encoder BCD	35
2.3	Schematico rete di priorità	36
2.4	Schematico encoder BCD on board	37
2.5	Anodi e catodi del display a 7 segmenti	37
2.6	Waveform encoder BCD	45
2.7	Waveform arbitro	47
2.8	Waveform priority encoder BCD	48
2.9	Waveform cathodes manager	51
2.10	Waveform encoder BCD on-board	53
2.11	Schema di mapping dell'encoder BCD su board	54
3.1	Schematico riconoscitore di sequenze	56
3.2	Automa riconoscitore in modalità $m = 0$	57
3.3	Automa riconoscitore in modalità $m = 1$	57
3.4	Button debouncer	59
3.5	Automa button debouncer	59
3.6	Waveform riconoscitore $m = 0$	71
3.7	Waveform riconoscitore $m = 1$	71
3.8	Schema di mapping del riconoscitore di sequenze su board . .	72
4.1	Schematico flip-flop D	74

4.2	Shift register	75
4.3	Shift register top module per N = 4	76
4.4	Waveform shift register strutturale	84
4.5	Waveform shift register comportamentale	86
5.1	Schematico rappresentativo del contatore modulo N	89
5.2	Schematico rappresentativo del cronometro	89
5.3	Schematico rappresentativo del convertitore BCD	90
5.4	Schematico rappresentativo della memoria	91
5.5	Schematico unità operativa del sistema	92
5.6	Schematico automa dell'unità di controllo del sistema	94
5.7	Display a sette segmenti	95
5.8	Display manager	96
5.9	Waveform cronometro	117
5.10	Waveform sistema complessivo	119
5.11	Schema di mapping del sistema cronometro	120
6.1	Schematico rappresentativo del sistema di testing	122
6.2	Schematico rappresentativo di una ROM	122
6.3	Schematico rappresentativo della macchina combinatoria . . .	123
6.4	Unità operativa: sistema di testing	124
6.5	Unità di controllo: sistema di testing	125
6.6	Waveform sistema di testing	135
6.7	Schema di mapping del sistema di testing	136
7.1	Procedura di handshake	138
7.2	Sistema di comunicazione tramite handshake: schematico . .	139
7.3	Base dei tempi: schematico	140
7.4	Unità operativa e unità di controllo	141
7.5	Nodo A: unità operativa	142
7.6	Nodo A: unità di controllo	143
7.7	Nodo B: unità operativa	144
7.8	Nodo B: unità di controllo	145
7.9	Base dei tempi: waveform	165
7.10	Nodo A: waveform dell'unità di controllo	168
7.11	Nodo A: waveform	171
7.12	Nodo B: waveform dell'unità di controllo	174
7.13	Nodo B: waveform	177
7.14	Sistema complessivo: waveform	180
8.1	Architettura del processore Mic-1	182
8.2	Datapath del processore Mic-1	183
8.3	Adattamento dell'indirizzamento di MAR da word a byte .	186
8.4	Formato delle control word del Mic-1	187
8.5	Unità di controllo del processore Mic-1	188

8.6	Protocollo con la memoia del processore Mic-1	190
8.7	Codice IJVM per il testing del progetto del Mic-1	191
8.8	File ajvm.mal del progetto del Mic-1	192
8.9	Contenuto della RAM del Mic-1	193
8.10	Simulazione del programma di test IJVM	194
8.11	Microprocedura di ILOAD	195
8.12	Test ILOAD: codice IJVM	196
8.13	Test ILOAD: contenuto della RAM	197
8.14	Test ILOAD: simulazione del processore Mic-1	197
8.15	Microprocedura di IF_ICMPEQ	198
8.16	IF_ICMPEQ: contenuto del Control Store	200
8.17	IF_ICMPEQ: contenuto del Control Store, ramo False	201
8.18	IF_ICMPEQ: contenuto del Control Store, ramo True	201
8.19	Test IF_ICMPEQ: codice IJVM	202
8.20	Test IF_ICMPEQ: contenuto della memoria	203
8.21	Test IF_ICMPEQ: waveform prodotte dalla simulazione	204
8.22	SLL: codice MAL	205
8.23	SLL: modifica del Control Store	206
8.24	SLL: contenuto della memoria	207
8.25	SLL: risultati della simulazione	207
9.1	Campionamento della linea dato	210
9.2	Schematico esercizio 9.1	212
9.3	Schematico esercizio 9.2	213
9.4	Schematico unità operativa del nodo A	214
9.5	Unità di controllo del nodo A	214
9.6	Schematico unità operativa del nodo B	215
9.7	Unità di controllo del nodo B	216
9.8	Esercizio 9.1 - Risultati della simulazione	232
9.9	Esercizio 9.2 - Risultati della simulazione	235
10.1	Switch a singolo stadio	237
10.2	Algoritmo di perfect shuffling	238
10.3	Switch	238
10.4	Messaggio di ingresso	239
10.5	Switch multistadio: unità operativa	240
10.6	Switch multistadio: unità di controllo	240
10.7	Waveform switch	248
10.8	Waveform unità operativa	249
10.9	Waveform switch multistadio	251
11.1	Schematico del moltiplicatore di Booth ad 8 bit	255
11.2	Algoritmo del moltiplicatore di Booth ad 8 bit	256
11.3	Unità di controllo del moltiplicatore di Booth	257

11.4 Schematico dell'adder/subtractor	257
11.5 Adder/Subtractor - Waveform	268
11.6 Unità di controllo - Waveform	272
11.7 Moltiplicatore di Booth - Waveform	274
11.8 Adder/Subtractor - Unconstrained timing report	275
11.9 Adder/Subtractor - Constrained timing analysis	276
11.10Adder/Subtractor - Constrained timing report	277
11.11Moltiplicatore di Booth - Constrained timing report	277
11.12Moltiplicatore di Booth on board	278
11.13Mapping dell'I/O	279
 12.1 Comparatore ad un solo bit	282
12.2 Comparatore di stringhe da N bit	283
12.3 Sistema complessivo	284
12.4 Nodo A: unità operativa	285
12.5 Nodo A: unità di controllo	285
12.6 Nodo B: unità operativa	286
12.7 Nodo B: unità di controllo	287
12.8 Nodo C: unità operativa	288
12.9 Nodo C: unità di controllo	289
12.10Comparatore di word - Risultati della simulazione	313
12.11Sistema complessivo - Risultati della simulazione	316
12.12Schema di mapping del sistema complessivo su board	317

Elenco dei codici sorgente

1.1	Multiplexer 4:1	19
1.2	Multiplexer 16:1	20
1.3	Demultiplexer 1:4	21
1.4	Rete di interconnessione 16:4	22
1.5	Rete di interconnessione on-board	23
1.6	Testbench multiplexer 4:1	25
1.7	Testbench multiplexer 16:1	26
1.8	Testbench demultiplexer 1:4	27
1.9	Testbench rete di interconnessione	29
1.10	Testbench rete di interconnessione on-board	32
2.1	Encoder BCD	39
2.2	Arbitro	40
2.3	Priority encoder BCD	41
2.4	Cathodes manager	42
2.5	Encoder BCD on board	43
2.6	Testbench encoder BCD	45
2.7	Testbench arbitro	47
2.8	Testbench priority encoder BCD	48
2.9	Testbench cathodes manager	50
2.10	Testbench encoder BCD on-board	53
3.1	Riconoscitore di sequenze	63
3.2	Button debouncer	65
3.3	Riconoscitore di sequenze on-board	66
3.4	Testbench riconoscitore di sequenze	70
4.1	Flip-flop D	77
4.2	Shift register: approccio strutturale	79
4.3	Shift register: approccio comportamentale	81
4.4	Testbench shift register con approccio strutturale	83
4.5	Testbench shift register con approccio comportamentale	86
5.1	Contatore modulo 60	98
5.2	Cronometro	100
5.3	Convertitore BCD	101
5.4	Memoria	102
5.5	Unità operativa del sistema cronometro	105

5.6	Unità di controllo del sistema cronometro	109
5.7	Sistema complessivo	111
5.8	Cronometro on-board	114
5.9	Testbench cronometro	117
5.10	Testbench sistema complessivo	119
6.1	Memoria ROM	127
6.2	Macchina combinatoria	127
6.3	Contatore	129
6.4	Unita operativa: sistema di testing	130
6.5	Unita di controllo: sistema di testing	132
6.6	Sistema di testing on-board	133
6.7	Testbench sistema di testing	135
7.1	Base dei tempi	148
7.2	Nodo A	150
7.3	Nodo A: unità operativa	151
7.4	Nodo A: unità di controllo	154
7.5	Nodo B	155
7.6	Nodo B: unità operativa	157
7.7	Nodo B: unità di controllo	160
7.8	Sistema di comunicazione tramite handshake	162
7.9	Sistema di comunicazione tramite handshake con base dei tempi	163
7.10	Base dei tempi: testbench	165
7.11	Nodo A: testbench dell'unità di controllo	167
7.12	Nodo A: testbench	170
7.13	Nodo B: testbench dell'unità di controllo	173
7.14	Nodo B: testbench	176
7.15	Sistema complessivo: testbench	180
9.1	Esercizio 9.1 - Nodo A	217
9.2	Esercizio 9.1 - Nodo B	218
9.3	Esercizio 9.1 - Sistema complessivo	220
9.4	Esercizio 9.2 - Unità operativa nodo A	221
9.5	Esercizio 9.2 - Unità di controllo nodo A	223
9.6	Esercizio 9.2 - Nodo A	224
9.7	Esercizio 9.2 - Unità operativa nodo B	226
9.8	Esercizio 9.2 - Unità di controllo nodo B	228
9.9	Esercizio 9.2 - Nodo B	229
9.10	Esercizio 9.2 - Sistema complessivo	230
9.11	Esercizio 9.1 - Testbench del sistema complessivo	232
9.12	Esercizio 9.2 - Testbench del sistema complessivo	234
10.1	Switch	241
10.2	Unità operativa: omega network	243
10.3	Arbitro	243
10.4	Unità di controllo	245
10.5	Switch multistadio	246

10.6	Test bench switch	247
10.7	Test bench unità operativa	249
10.8	Test bench switch multistadio	251
11.1	Unità operativa del moltiplicatore di Booth	261
11.2	Unità di controllo del moltiplicatore di Booth	263
11.3	Adder/Subtractor	264
11.4	Ripple Carry Adder	265
11.5	Full Adder	265
11.6	Adder/Subtractor - Testbench	268
11.7	Unità di controllo - Testbench	271
11.8	Moltiplicatore di Booth - Testbench	274
12.1	Comparatore di un bit	290
12.2	Comparatore di word	291
12.3	Encoder	292
12.4	Nodo A: unità operativa	293
12.5	Nodo A: unità di controllo	295
12.6	Nodo A	297
12.7	Nodo B: unità operativa	298
12.8	Nodo B: unità di controllo	300
12.9	Nodo B	301
12.10	Nodo C: unità operativa	304
12.11	Nodo C: unità di controllo	306
12.12	Nodo C	307
12.13	Sistema complessivo	308
12.14	Sistema complessivo on-board	310
12.15	Testbench comparatore di word	312
12.16	Testbench sistema complessivo	316

Elenco delle tabelle

1.1	Multiplexer 4:1	15
1.2	Demultiplexer 1:4	15
2.1	Encoder BCD	36
8.1	Operazioni ALU	185
8.2	Traduzione da IJVM a codice macchina	192
8.3	Test ILOAD: traduzione da IJVM a codice macchina	196
8.4	Test IF_ICMPEQ: traduzione da IJVM a codice macchina	202
8.5	Test SLL: traduzione da IJVM a codice macchina	206
11.1	Codifica di Booth	254

Capitolo 1

Multiplexer

1.1 Traccia

Esercizio 1.1 Progettare, implementare in VHDL e testare mediante simulazione un multiplexer indirizzabile 16:1, utilizzando un approccio di progettazione per composizione a partire da multiplexer 4:1.

Esercizio 1.2 Utilizzando il componente sviluppato al punto precedente, progettare, implementare in VHDL e testare mediante simulazione una rete di interconnessione a 16 sorgenti e 4 destinazioni.

Esercizio 1.3 Sintetizzare ed implementare su board il progetto della rete di interconnessione sviluppato al punto 1.2, utilizzando gli switch per fornire gli input di selezione e i led per visualizzare i 4 bit di uscita. Per quanto riguarda i 16 bit dato in input, essi possono essere precaricati nel sistema oppure immessi anch'essi mediante switch, sviluppando in questo secondo caso un'apposita rete di controllo per l'acquisizione.

1.2 Soluzione

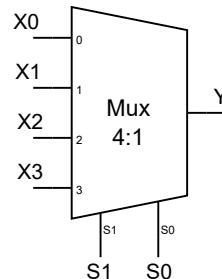


Figura 1.1: Schematico multiplexer 4:1

Il blocco base del progetto è il multiplexer 4:1 (figura 1.1). Il suo funzionamento è descritto nella tabella di verità "sintetica" 1.1.

Tabella 1.1: Multiplexer 4:1

S_1	S_0	Y
0	0	X_0
0	1	X_1
1	0	X_2
1	1	X_3

Per risolvere l'esercizio 1.1 si è progettato un multiplexer 16:1, combinando cinque multiplexer 4:1 secondo lo schema ad albero in figura 1.2.

Per realizzare la rete di interconnessione a 16 ingressi e 4 uscite (esercizio 1.2) si è utilizzato il multiplexer 16:1 progettato al punto precedente, collegandogli in cascata un demultiplexer 1:4 (figura 1.3), il cui funzionamento è descritto dalla tabella di verità 1.2 .

Tabella 1.2: Demultiplexer 1:4

S_1	S_0	Y_3	Y_2	Y_1	Y_0
0	0	0	0	0	X
0	1	0	0	X	0
1	0	0	X	0	0
1	1	X	0	0	0

La rete di interconnessione presenta:

- 16 ingressi dato;
- 4 linee di selezione per la sorgente;

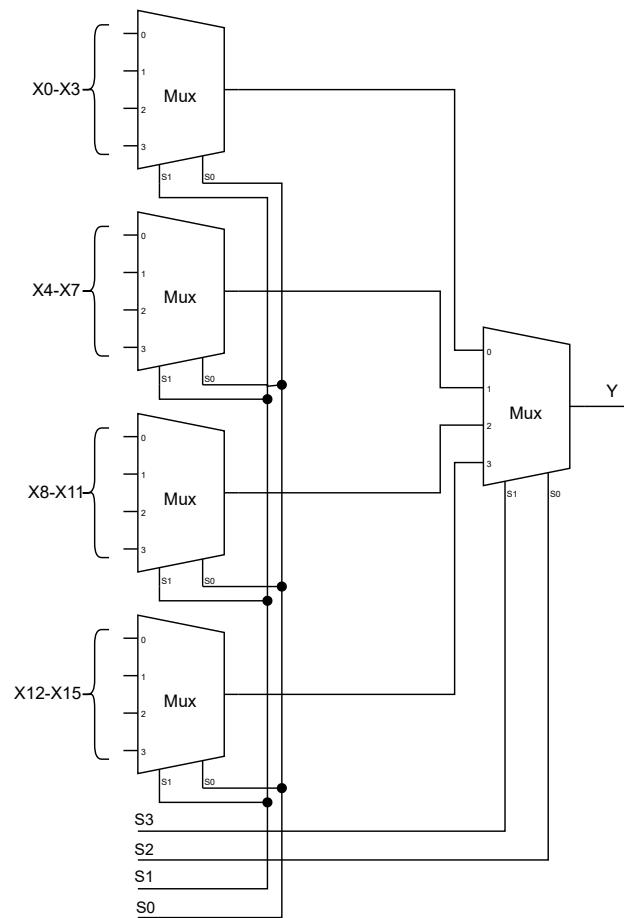


Figura 1.2: Schematico multiplexer 16:1

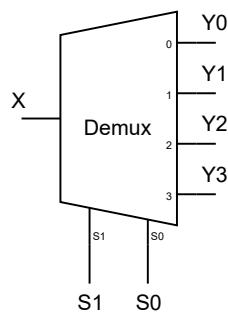


Figura 1.3: Schematico demultiplexer 1:4

- 2 linee di selezione per la destinazione;

per un totale di 20 linee d'ingresso e 4 di uscita. Lo schema della rete è rappresentato in figura 1.4 .

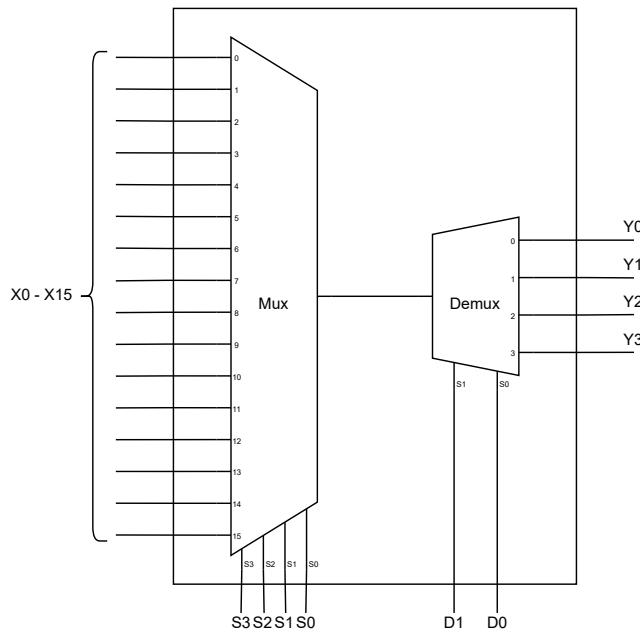


Figura 1.4: Schematico rete di interconnessione a 16 ingressi e 4 uscite

L'esercizio 1.3 prevede di caricare il progetto della rete di interconnessione sulla board Nexys-A7 della Digilent. Per fare ciò, è necessario progettare un sistema per dare in input i 16 ingressi «dato» della rete, dato che gli switch della board non sono in numero sufficiente per poter controllare 20 linee di input. Per risolvere tale problema è stata progettata una ROM 16x16 indirizzata tramite i valori di 4 switch, le cui uscite sono state collegate in ingresso alla rete di interconnessione. Lo schematico del progetto caricato sulla board è rappresentato in figura 1.5 .

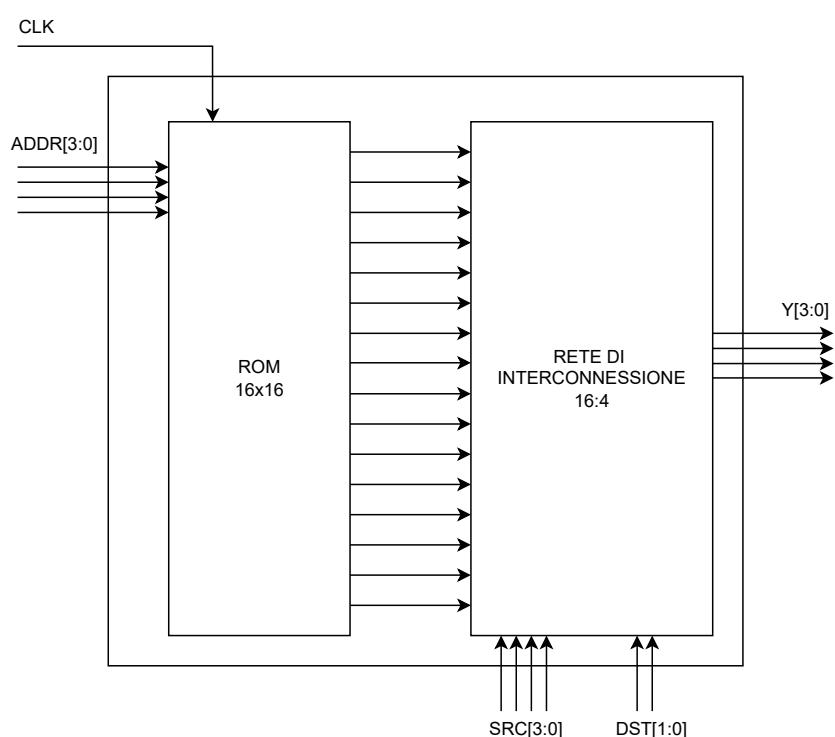


Figura 1.5: Schematico rete di interconnessione on-board

1.3 Codice

1.3.1 Multiplexer 4:1

Il multiplexer 4:1, blocco base del nostro progetto, è stato descritto secondo un'architettura *dataflow* come si evince dal listato 1.1 .

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux4to1 is
5     Port ( x : in STD_LOGIC_VECTOR (3 downto 0);
6             s : in STD_LOGIC_VECTOR (1 downto 0);
7             y : out STD_LOGIC);
8 end mux4to1;
9
10 architecture dataflow of mux4to1 is
11
12 begin
13     with s select
14         y <= x(0) when "00",
15                 x(1) when "01",
16                 x(2) when "10",
17                 x(3) when others;
18 end dataflow;
```

Listing 1.1: Multiplexer 4:1

1.3.2 Multiplexer 16:1

Il multiplexer 16:1 è stato realizzato con un'approccio *strutturale*, combinando tra loro cinque mutliplexer 4:1 in un struttura ad albero. In particolare: quattro multiplexer sul primo livello che raccolgono le 16 linee di ingresso ed effettuano un primo instradamento usando i due bit di selezione meno significativi, ed un ultimo multiplexer sul secondo livello che raccoglie le uscite dei primi quattro e ne instrada in output solo una, in base al valore dei due bit di selezione più significativi. Per generare i quattro multiplexer 4:1 del primo livello è stato utilizzato il costrutto di `for...generate`.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux16to1 is
5     Port (
```

```

6      x : in std_logic_vector(15 downto 0);
7      s : in std_logic_vector(3 downto 0);
8      y : out std_logic );
9 end mux16to1;
10
11 architecture structural of mux16to1 is
12     component mux4to1 port(
13         x : in std_logic_vector(3 downto 0);
14         s : in std_logic_vector(1 downto 0);
15         y : out std_logic );
16     end component;
17     -- for all : mux4to1 use entity work.mux4to1(dataflow);
18
19     signal w : std_logic_vector (3 downto 0);
20 begin
21     m0_3 : for i in 0 to 3 generate
22         m : mux4to1 port map(
23             x => x(i*4+3 downto i*4),
24             s => s(1 downto 0),
25             y => w(i) );
26     end generate;
27     m4 : mux4to1 port map(
28         x => w,
29         s => s(3 downto 2),
30         y => y );
31
32 end structural;

```

Listing 1.2: Multiplexer 16:1

1.3.3 Demultiplexer 1:4

Analogamente al multiplexer 4:1, il demultiplexer 1:4 è stato descritto secondo un approccio *dataflow* (listato 1.3).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity demux1to4 is
5     Port ( x : in STD_LOGIC;
6             s : in STD_LOGIC_VECTOR (1 downto 0);
7             y : out STD_LOGIC_VECTOR (3 downto 0));
8 end demux1to4;

```

```

9
10 architecture dataflow of demux1to4 is
11
12 begin
13     y(0) <= x when s = "00" else '0';
14     y(1) <= x when s = "01" else '0';
15     y(2) <= x when s = "10" else '0';
16     y(3) <= x when s = "11" else '0';
17
18 end dataflow;

```

Listing 1.3: Demultiplexer 1:4

1.3.4 Rete di interconnessione

La rete di interconnessione 16:4 è stata progettata secondo un approccio *strutturale*, istanziando un multiplexer 16:1 ed un demultiplexer 1:4, e collegando l'uscita del primo con l'ingresso del secondo.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity rete_inter is
5     Port ( x : in STD_LOGIC_VECTOR (15 downto 0);
6             s : in STD_LOGIC_VECTOR (3 downto 0);
7             d : in STD_LOGIC_VECTOR (1 downto 0);
8             y : out STD_LOGIC_VECTOR (3 downto 0) );
9 end rete_inter;
10
11 architecture structural of rete_inter_16to4 is
12     component mux16to1 port(
13         x : in std_logic_vector(15 downto 0);
14         s : in std_logic_vector(3 downto 0);
15         y : out std_logic );
16     end component;
17     component demux1to4 port(
18         x : in std_logic;
19         s : in std_logic_vector(1 downto 0);
20         y : out std_logic_vector(3 downto 0) );
21     end component;
22     signal w : std_logic;
23 begin
24     mux : mux16to1 port map(

```

```

25      x => x,
26      s => s,
27      y => w );
28  demux: demux1to4 port map(
29      x => w,
30      s => d,
31      y => y );
32
33 end structural;

```

Listing 1.4: Rete di interconnessione 16:4

1.3.5 Rete di interconnessione on-board

Come descritto nel paragrafo 1.2, per poter implementare la rete di interconnessione 16:4 sulla board è stata utilizzata una memoria ROM 16x16 per fornire gli ingressi «dato» alla rete, per far fronte alla carenza di switch sulla scheda. Il codice VHDL della memoria ROM non viene riportato in questo capitolo, perché tale argomento sarà trattato in maniera dettagliata nei capitoli successivi. Il progetto on-board è stato realizzato secondo un approccio *strutturale* istanziando una memoria ROM e la rete di interconnessione, e collegando le uscite della prima con gli ingressi della seconda. I collegamenti verso l'esterno del componente sono:

- 4 linee di input per indirizzare la ROM;
- 4 linee di input per selezionare la sorgente della rete;
- 2 linee di input per selezionare la destinazione della rete;
- 4 linee di output che trasportano le uscite della rete 16:4.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity rete_on_board is
5     Port ( addr : in STD_LOGIC_VECTOR (3 downto 0);
6             src : in STD_LOGIC_VECTOR (3 downto 0);
7             dst : in STD_LOGIC_VECTOR (1 downto 0);
8             clk : in STD_LOGIC;
9             y : out STD_LOGIC_VECTOR (3 downto 0));
10 end rete_on_board;
11
12 architecture Behavioral of rete_on_board is

```

```

13     component rete_inter port(
14         x : in std_logic_vector(15 downto 0);
15         s : in std_logic_vector(3 downto 0);
16         d : in std_logic_vector(1 downto 0);
17         y : out std_logic_vector(3 downto 0) );
18     end component;
19
20     component rom port(
21         addr : in std_logic_vector(3 downto 0);
22         clk : in std_logic;
23         y : out std_logic_vector(15 downto 0) );
24     end component;
25
26     signal w : std_logic_vector(15 downto 0);
27 begin
28     net : rete_inter port map(
29         x => w,
30         s => src,
31         d => dst,
32         y => y );
33     mem : rom port map(
34         addr => addr,
35         clk => clk,
36         y => w );
37 end Behavioral;

```

Listing 1.5: Rete di interconnessione on-board

1.4 Simulazione

Per testare ogni componente del progetto è stata effettuata una simulazione comportamentale per ognuno di esso. Per effettuare la simulazione occorre definire per ogni componente da testare un *testbench*, ossia un'unità in VHDL che istanzi il componente (*DUT, Design Under Test*), vi applichi gli ingressi, e confronti l'output con un *oracolo*, utilizzando il comando **assert...report...severity**. In alcuni casi, può risultare sufficiente un controllo manuale dei risultati di test attraverso l'ispezione dei diagrammi temporali prodotti dalla simulazione.

Di seguito si riportano i codici VHDL per l'implementazione dei *testbench* per ogni componente del progetto e i diagrammi temporali risultanti dalle simulazioni.

1.4.1 Multiplexer 4:1

Per testare il multiplexer 4:1 si è scelto di tenere fissi gli input «dato» al valore "0101" e si è fatto variare gli input di selezione su tutte le 2^2 possibili configurazioni.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity mux4to1_tb is
5   -- Port ( );
6 end mux4to1_tb;
7
8 architecture Behavioral of mux4to1_tb is
9   component mux4to1 port(
10     x : in std_logic_vector(3 downto 0);
11     s : in std_logic_vector(1 downto 0);
12     y : out std_logic);
13   end component;
14   signal input : std_logic_vector(3 downto 0) := "0101";
15   signal sel : std_logic_vector(1 downto 0);
16   signal output : std_logic;
17 begin
18   dut : mux4to1 port map(
19     x => input,
20     s => sel,
21     y => output );
22   process
23   begin
24     wait for 5ns;
```

```

25      sel <= "00";
26      wait for 5ns;
27      assert output = input(0) report "errore" severity
28          => failure;
28      sel <= "01";
29      wait for 5ns;
30      assert output = input(1) report "errore" severity
31          => failure;
31      sel <= "10";
32      wait for 5ns;
33      assert output = input(2) report "errore" severity
34          => failure;
34      sel <= "11";
35      wait for 5ns;
36      assert output = input(3) report "errore" severity
37          => failure;
37
38  end process;
39
40 end Behavioral;

```

Listing 1.6: Testbench multiplexer 4:1

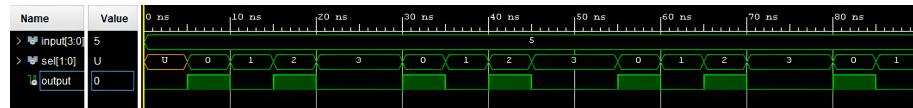


Figura 1.6: Waveform multiplexer 4:1

1.4.2 Multiplexer 16:1

Anche per testare il multiplexer 16:1 si sono tenute fisse le linee di ingresso al valore esadecimale D555, che corrisponde alla string di bit 11010101010101010101. Sfruttando il costrutto `for...loop`, si sono fati variare i bit di selezione (`s[3:0]`) su tutte le 2^4 possibili combinazioni.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.all;
4
5 entity mux16to1_tb is
6     -- Port ( );

```

```

7  end mux16to1_tb;
8
9  architecture Behavioral of mux16to1_tb is
10 component mux16to1 port(
11     x : in std_logic_vector(15 downto 0);
12     s : in std_logic_vector(3 downto 0);
13     y : out std_logic );
14 end component;
15
16 signal input : std_logic_vector(15 downto 0) := x"d555";
17 signal sel : std_logic_vector(3 downto 0);
18 signal output : std_logic;
19 begin
20     dut : mux16to1 port map(
21         x => input,
22         s => sel,
23         y => output );
24
25 process
26 begin
27     wait for 5ns;
28     for i in 0 to 15 loop
29         sel <= std_logic_vector(to_unsigned(i,sel'length));
30         wait for 5ns;
31         assert output = input(i) report "errore" severity
32             failure;
33     end loop;
34 end process;
35 end Behavioral;

```

Listing 1.7: Testbench multiplexer 16:1

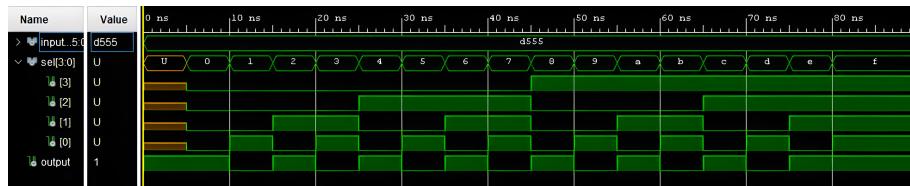


Figura 1.7: Waveform multiplexer 16:1

1.4.3 Demultiplexer 1:4

Il testing del demultiplexer 1:4 è molto semplice. Si è tenuto alto il singolo bit «dato» in ingresso, e si è fatto variare l'ingresso di selezione su tutte le 2^2 possibili combinazioni.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.all;
4
5 entity demux1to4_tb is
6 -- Port ( );
7 end demux1to4_tb;
8
9 architecture Behavioral of demux1to4_tb is
10 component demux1to4 port(
11     x : in std_logic;
12     s : in std_logic_vector(1 downto 0);
13     y : out std_logic_vector(3 downto 0) );
14 end component;
15 signal input : std_logic := '1';
16 signal sel : std_logic_vector(1 downto 0);
17 signal output : std_logic_vector(3 downto 0);
18 begin
19     dut : demux1to4 port map(
20         x => input,
21         s => sel,
22         y => output );
23
24     process
25     begin
26         wait for 5ns;
27         for i in 0 to 3 loop
28             sel <= std_logic_vector(to_unsigned(i,sel'length));
29             wait for 5ns;
30             assert output(i) = input report "errore" severity
31             → failure;
32         end loop;
33     end process;
34 end Behavioral;
```

Listing 1.8: Testbench demultiplexer 1:4

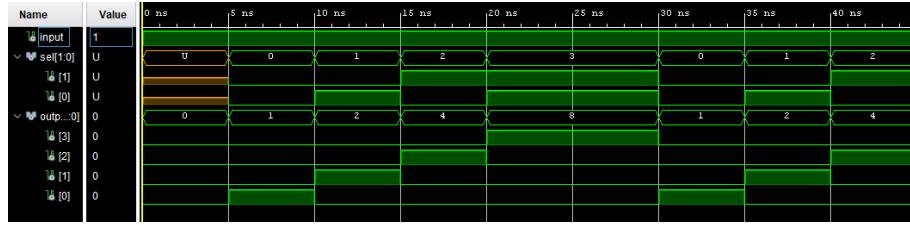


Figura 1.8: Waveform demultiplexer 1:4

1.4.4 Rete di interconnessione

Anche in questo caso la procedura di testing è analoga a quella dei multiplexer e demultiplexer precedenti: le linee «dato» sono fisse, mentre variano le linee per selezionare la sorgente e la destinazione. Il dato in input è fisso al valore esadecimale A5A5 (in bit 1010010110100101), mentre le linee «sorgente» e le linee «destinazione» variano complessivamente su $2^{4+2} = 2^6$ possibili configurazioni diverse. Per far variare le linee di selezione si è utilizzato due **for...loop** innestati: quello esterno per instradare la sorgente (*i*), mentre quello interno per instradare la destinazione (*j*). Infine, il costrutto **assert** controlla che il valore sulla linea d'uscita *j* coincida col valore sulla linea d'ingresso *i*.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.all;
4
5 entity rete_inter_tb is
6 -- Port ( );
7 end rete_inter_tb;
8
9 architecture Behavioral of rete_inter_tb is
10 component rete_inter port(
11     x : in std_logic_vector(15 downto 0);
12     s : in std_logic_vector(3 downto 0);
13     d : in std_logic_vector(1 downto 0);
14     y : out std_logic_vector(3 downto 0) );
15 end component;
16 signal input : std_logic_vector(15 downto 0) := x"a5a5";
17 signal src : std_logic_vector(3 downto 0);
18 signal dest : std_logic_vector(1 downto 0);
19 signal output : std_logic_vector(3 downto 0);
20 begin
21     dut : rete_inter port map(
22         x => input,

```

```

23         s => src,
24         d => dest,
25         y => output );
26 process
27     constant n_src : positive := 2**src'length;
28     constant n_dest : positive := 2**dest'length;
29 begin
30     wait for 5ns;
31     for i in 0 to n_src-1 loop
32         src <= std_logic_vector(to_unsigned(i,src'length));
33
34     for j in 0 to n_dest-1 loop
35         dest <=
36             std_logic_vector(to_unsigned(j,dest'length));
37         wait for 5ns;
38         assert output(j) = input(i) report "errore"
39             severity failure;
40     end loop;
41 end process;
42
43 end Behavioral;

```

Listing 1.9: Testbench rete di interconnessione

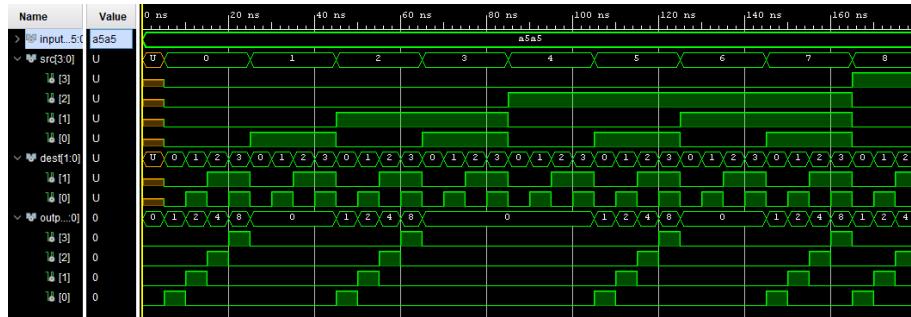


Figura 1.9: Waveform rete di interconnessione

1.4.5 Rete di interconnessione on-board

La differenza tra il progetto della rete "on-board" e il progetto della rete al paragrafo precedente è semplicemente l'aggiunta di una memoria ROM per fornire l'input «dato» alla rete. Anche in questo caso, il progetto della

memoria ROM è stato testato con un apposito *testbench*, ma tale test non è qui riportato per evitare di appesantire questo capitolo. Il progetto della memoria ROM sarà poi approfondito nei capitoli successivi. Avendo già testato la memoria ROM e la rete di interconnessione singolarmente, il focus del test "on-board" è verificare la corretta integrazione dei due componenti. Per questo motivo, e anche per il gran numero di possibili combinazioni di input fornibili alla macchina (4 linee di indirizzo per la ROM, 4 linee di selezione della sorgente e 2 linee di selezione della destinazione, per un totale di $2^{4+4+2} = 2^{10}$ possibili combinazioni), si è testato il sistema contro poche combinazioni di input rispetto al totale, ma cercando le combinazioni più significative. Rispetto ai test precedenti, in questo caso viene fatto variare anche l'input «dato» della rete, perché tale input rappresenta il collegamento tra le uscite della ROM e gli ingressi della rete di interconnessione. Tale input «dato» viene fatto variare selezionando la locazione di memoria della ROM da leggere, attraverso la variazione dei valori sulle linee di indirizzo della ROM (`addr[3:0]`).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity rete_on_board_tb is
5   -- Port ( );
6 end rete_on_board_tb;
7
8 architecture Behavioral of rete_on_board_tb is
9   component rete_on_board port(
10     addr : in std_logic_vector(3 downto 0);
11     src : in std_logic_vector(3 downto 0);
12     dst : in std_logic_vector(1 downto 0);
13     clk : in std_logic;
14     y : out std_logic_vector(3 downto 0) );
15   end component;
16
17   signal address : std_logic_vector(3 downto 0) := "0000";
18   signal source : std_logic_vector(3 downto 0) := "0000";
19   signal destination : std_logic_vector(1 downto 0) := "00";
20   signal clock : std_logic := '0';
21   signal output : std_logic_vector(3 downto 0);
22 begin
23   --Design Under Test
24   dut : rete_on_board port map(
25     addr => address,
26     src => source,
```

```

27         dst => destination,
28         clk => clock,
29         y => output );
30
31     gen_clk : process
32 begin
33     wait for 5ns;
34     clock <= '1';
35     wait for 5ns;
36     clock <= '0';
37 end process;
38
39 process
40 begin
41     wait for 50ns;
42
43     address <= "0000";
44     source <= "0000";
45     destination <= "11";
46     wait for 50ns;
47     assert output = "1000" report "error_1" severity
48         => failure;
49
50     destination <= "01";
51     wait for 50ns;
52     assert output = "0010" report "error_2" severity
53         => failure;
54
55     source <= "1111";
56     wait for 50ns;
57     assert output = "0000" report "error_3" severity
58         => failure;
59
60     address <= "1111";
61     wait for 50ns;
62     assert output = "0010" report "error_4" severity
63         => failure;
64
65     source <= "0010";
66     destination <= "11";
67     wait for 50ns;
68     assert output = "1000" report "error_5" severity
69         => failure;
70 end process;

```

66

67 **end Behavioral;**

Listing 1.10: Testbench rete di interconnessione on-board

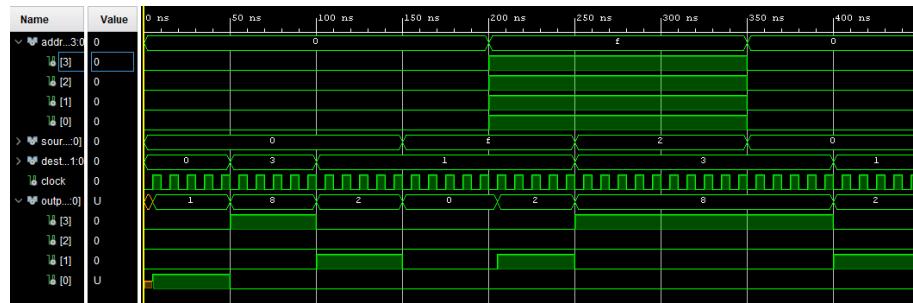


Figura 1.10: Waveform rete di interconnessione on-board

1.5 Sintesi

Il progetto della rete di interconnessione 16:4 "on-board" è stato caricato sulla scheda Nexys-A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente, che ricordiamo essere:

- 4 linee di indirizzamento per la ROM;
- 4 linee di selezione della sorgente;
- 2 linee di selezione della destinazione;
- 4 linee di output della rete;

con gli elementi di I/O presenti sulla scheda. In particolare, tutte le linee di input sono state mappate sugli switch della scheda, mentre le quattro linee di output sono state mappate sui LED, secondo la configurazione in figura 1.11. Inoltre, nel progetto della rete di interconnessione "on-board" è presente un ulteriore ingresso utilizzato per tempificare la ROM, ovvero il *clock*. Tale ingresso è stato collegato al generatore di clock presente sulla scheda, che genera un segnale alla frequenza di 100MHz.

Il mapping tra le linee verso l'esterno del progetto sulla FPGA e i componenti fisici della board, è stato implementato modificando il file dei *constraints* "Nexys-A7-100T-Master.xdc".

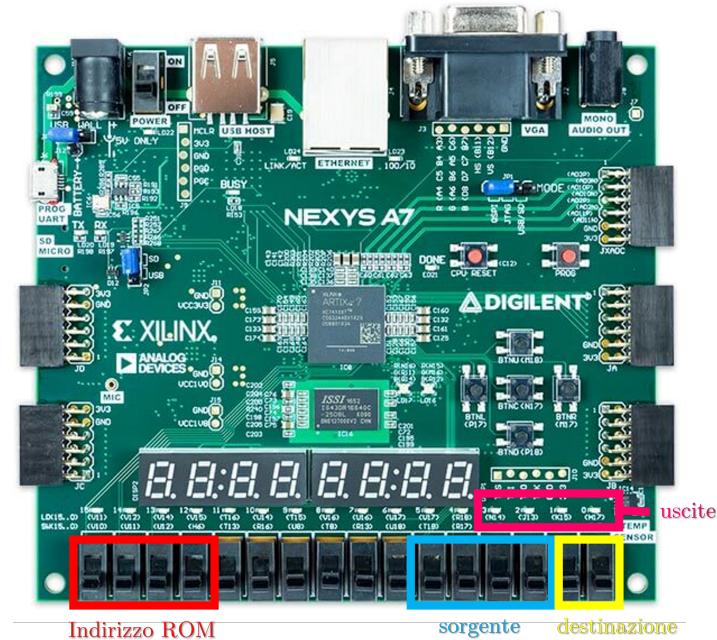


Figura 1.11: Schema di mapping della rete di interconnessione su board

Capitolo 2

Encoder BCD

2.1 Traccia

Esercizio 2.1 Progettare, implementare in VHDL e testare mediante simulazione una rete che, data in ingresso una stringa binaria X di 10 bit X9 X8 X7 X6 X5 X4 X3 X2 X1 X0 che corrisponde alla rappresentazione decodificata di una cifra decimale (cioè, una rappresentazione in cui ogni stringa contiene un solo bit alto), fornisce in uscita la rappresentazione Y della cifra mediante codifica Binary-Coded Decimal (BCD).

Input: 0000000001 Output: 0000 (cifra 0)

Input: 0000000010 Output: 0001 (cifra 1)

Input: 0000000100 Output: 0010 (cifra 2)

....

Esercizio 2.2 Sintetizzare ed implementare su board il progetto dell'encoder BCD utilizzando gli switch per fornire la stringa X in ingresso, e i led per visualizzare Y. Nel caso in cui si utilizzi una board dotata di soli 8 switch, è possibile sviluppare il progetto considerando X di soli 8 bit (la macchina sarà allora in grado di fornire in uscita la rappresentazione BCD delle cifre decimali da 0 a 7).

Esercizio 2.3 Utilizzare un display a 7 segmenti per visualizzare la cifra decimale codificata da Y (pilotare opportunamente i catodi del display per visualizzare la cifra).

2.2 Soluzione

Un codificatore (o encoder) è un circuito digitale combinatorio dotato di 2^n segnali di ingresso e di n segnali di uscita. Se viene attivata una delle linee di ingresso, in uscita viene prodotto il codice corrispondente. In particolare, realizzeremo un priority encoder BCD (figura 2.1), ovvero un codificatore BCD, con 10 ingressi e 4 uscite, a cui viene anteposta una rete di priorità, che garantisce che l'input fornito all'encoder presenti una codifica con un singolo bit alto, ovvero la codifica relativa al numero decimale più alto.

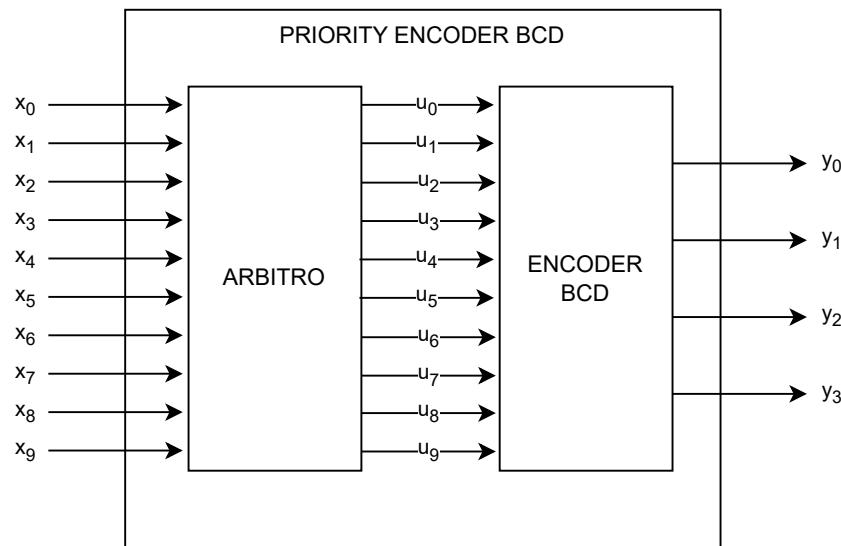


Figura 2.1: Schematico priority encoder BCD

Il blocco base del progetto dunque è l'encoder BCD (figura 2.2), il cui funzionamento è descritto nella tabella di verità 2.1, ovvero fornisce in uscita in base agli ingessi un numero decimale corrispondente.

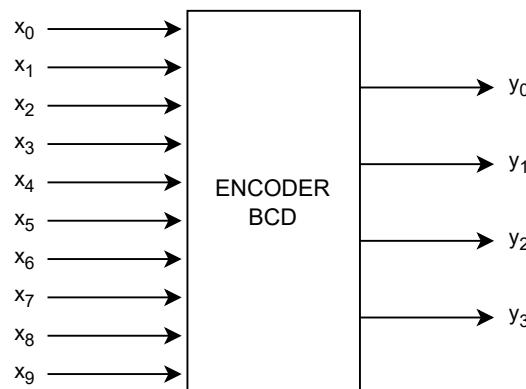


Figura 2.2: Schematico encoder BCD

Tabella 2.1: Encoder BCD

x_9	x_8	x_7	x_6	x_5	x_4	x_3	x_2	x_1	x_0	y_3	y_2	y_1	y_0
0	0	0	0	0	0	0	0	0	1	0	0	0	0
0	0	0	0	0	0	0	0	1	0	0	0	0	1
0	0	0	0	0	0	0	1	0	0	0	0	1	0
0	0	0	0	0	0	1	0	0	0	0	0	1	1
0	0	0	0	0	1	0	0	0	0	0	1	0	0
0	0	0	0	1	0	0	0	0	0	0	1	0	1
0	0	0	1	0	0	0	0	0	0	0	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	1	1
0	1	0	0	0	0	0	0	0	0	1	0	0	0
1	0	0	0	0	0	0	0	0	0	1	0	0	1

All'encoder BCD viene anteposto una rete di priorità (o arbitro, figura 2.3) a 10 ingressi e 10 uscite e tale macchina combinatoria fa sì che solo una linea di uscita, in base agli ingressi, può essere alta (ovvero la linea che rappresenta la cifra decimale più alta). Ad esempio:

X	Y
1111111111	1000000000
0010100001	0010000000
0000000011	0000000010
0101010101	0100000000

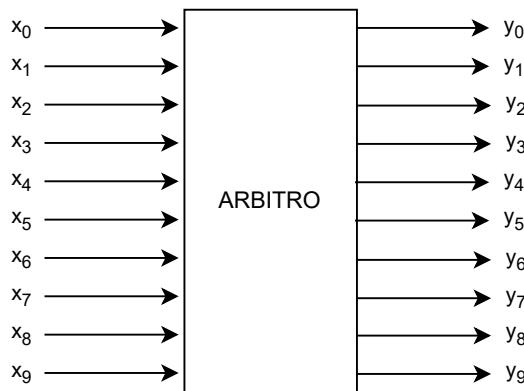


Figura 2.3: Schematico rete di priorità

L'esercizio 2.2 prevede di caricare il progetto dell' encoder BCD sulla board Nexys-A7 della Digilent. Le 10 linee di input verranno collegate agli switch della board invece le uscite sono state collegate in ingresso a 4 led presenti sempre sulla board.

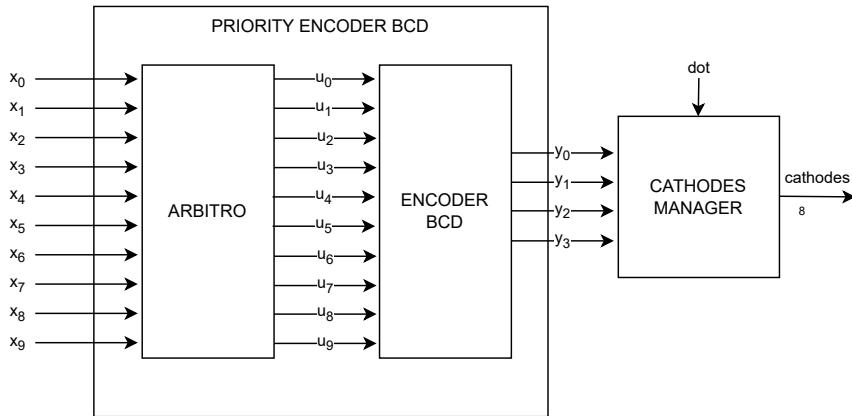


Figura 2.4: Schematico encoder BCD on board

Infine l'esercizio 2.3, richiede di utilizzare per visualizzare la cifra decimale codificata dall'uscita non più i led ma il visore presente sulla board di sviluppo. La scheda Nexys A7 dispone di due display a sette segmenti con anodo comune, configurati come un unico display a otto cifre. Ogni cifra è composta da sette segmenti LED che possono essere illuminati individualmente, permettendo di visualizzare diverse cifre e schemi. Gli anodi dei segmenti sono collegati insieme in un nodo comune, ma i catodi rimangono separati (vedi figura 2.5). Questo crea un display multiplex, dove i catodi sono comuni a tutte le cifre ma solo un anodo alla volta può essere attivato.

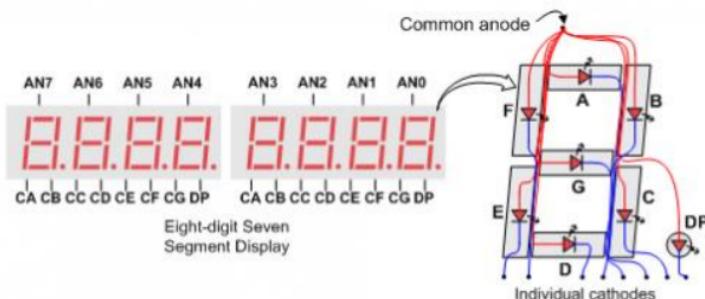


Figura 2.5: Anodi e catodi del display a 7 segmenti

Per illuminare un segmento, l'anodo deve essere acceso mentre il catodo è spento. È importante notare che i segnali dell'anodo sono invertiti a causa dell'uso di transistor per fornire la corrente necessaria. Pertanto, quando un segmento è attivo, sia il segnale dell'anodo sia quello del catodo sono bassi. In questo esercizio, per la corretta visualizzazione dell'uscita dell'encoder, utilizzeremo un'unica cifra del display, evitando così i problemi di refreshing dei catodi, che consente la visualizzazione di più cifre sul display. La corretta

visualizzazione della cifra sul display è gestita dal solo *cathodes manager*, figura 2.4, dove l'uscita di quest'ultimo rappresenta appunto qual è la configurazione di accensione dei 7 led. Nel presente esercizio, non avremo bisogno di un gestore degli anodi poiché a noi interessa visualizzare l'uscita su un'unica cifra, quindi l'anodo relativo alla cifra è fissato.

2.3 Codice

2.3.1 Encoder BCD

L'encoder BCD, blocco base del nostro progetto, è stato descritto secondo un'architettura *dataflow* come si evince dal listato 2.1 .

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoderBCD is
5     Port ( x : in STD_LOGIC_VECTOR (9 downto 0);
6             y : out STD_LOGIC_VECTOR (3 downto 0));
7 end encoderBCD;
8
9 architecture dataflow of encoderBCD is
10
11 begin
12     with x select
13         y <= "0000" when "0000000001",
14                     "0001" when "0000000010",
15                     "0010" when "0000000100",
16                     "0011" when "0000001000",
17                     "0100" when "0000010000",
18                     "0101" when "0000100000",
19                     "0110" when "0001000000",
20                     "0111" when "0010000000",
21                     "1000" when "0100000000",
22                     "1001" when "1000000000",
23                     "----" when others;
24
25 end dataflow;
```

Listing 2.1: Encoder BCD

2.3.2 Arbitro

Analogamente all' encoder BCD, l'arbitro è stato descritto secondo un approccio *dataflow* (listato 2.2).

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity arbiter is
```

```

5      Port ( x : in STD_LOGIC_VECTOR (9 downto 0);
6          y : out STD_LOGIC_VECTOR (9 downto 0));
7  end arbiter;
8
9  architecture dataflow of arbiter is
10
11 begin
12     y <= "1000000000" when x(9) = '1' else
13         "0100000000" when x(8) = '1' else
14         "0010000000" when x(7) = '1' else
15         "0001000000" when x(6) = '1' else
16         "0000100000" when x(5) = '1' else
17         "0000010000" when x(4) = '1' else
18         "0000001000" when x(3) = '1' else
19         "0000000100" when x(2) = '1' else
20         "0000000010" when x(1) = '1' else
21         "0000000001" when x(0) = '1' else
22         (others => '-');
23
24 end dataflow;

```

Listing 2.2: Arbitro

2.3.3 Priority encoder BCD

Il priority encoder BCD è stato realizzato con un’approccio *strutturale*, collegando, con un opportuno segnale interno w , l’uscita dell’arbitro con l’ingresso dell’encoder BCD (figura 2.1). Il relativo codice VHDL è presente al listato 2.3.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity priority_encoder is
5     Port ( x : in STD_LOGIC_VECTOR (9 downto 0);
6             y : out STD_LOGIC_VECTOR (3 downto 0));
7 end priority_encoder;
8
9 architecture structural of priority_encoder is
10    component arbiter port(
11        x : in std_logic_vector(9 downto 0);
12        y : out std_logic_vector(9 downto 0) );
13    end component;

```

```

14     component encoderBCD port(
15         x : in std_logic_vector(9 downto 0);
16         y : out std_logic_vector(3 downto 0) );
17     end component;
18     signal w : std_logic_vector(9 downto 0);
19 begin
20     arb : arbiter port map(
21         x => x,
22         y => w );
23     enc : encoderBCD port map(
24         x => w,
25         y => y);
26
27 end structural;

```

Listing 2.3: Priority encoder BCD

2.3.4 Gestore dei catodi

Per l'esercizio 2.3, fissiamo l'anodo fissando così anche la cifra del display su cui visualizzare l'ingresso. Dunque abbiamo bisogno solo del gestore dei catodi per visualizzare correttamente la cifra decimale su quella cifra del display selezionata, in particolare tale componente riceve in ingresso *value* (la stringa totale che vogliamo visualizzare) e *dot* (configurazione di accensione del punto sul display), invece in uscita restituisce il segnale *cathodes*, ovvero i 7 catodi più la configurazione del punto . Tale componente è definito al listato 2.4.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.std_logic_unsigned.all;
4
5 entity cathodes_manager is
6     Port ( value : in STD_LOGIC_VECTOR (3 downto 0);
7             dot : in STD_LOGIC;
8             cathodes : out STD_LOGIC_VECTOR (7 downto 0));
9 end cathodes_manager;
10
11 architecture dataflow of cathodes_manager is
12     -----
13     -- cathodes(0) => CA
14     -- cathodes(1) => CB
15     -- ...

```

```

16      -- cathodes(7) => DP
17      -----
18      type digit_type is array (0 to 15) of std_logic_vector(6
19          <-- downto 0);
20      constant digit : digit_type := (
21          "1000000", -- 0
22          "1111001", -- 1
23          "0100100", -- 2
24          "0110000", -- 3
25          "0011001", -- 4
26          "0010010", -- 5
27          "0000010", -- 6
28          "1111000", -- 7
29          "0000000", -- 8
30          "0010000", -- 9
31          "0001000", -- A
32          "0000011", -- b
33          "1000110", -- C
34          "0100001", -- d
35          "0000110", -- E
36          "0001110"); -- F
37      begin
38          cathodes <= (not dot) & digit(conv_integer(value));
39      end dataflow;

```

Listing 2.4: Cathodes manager

2.3.5 Encoder BCD on-board

L'encoder BCD sulla board è stato realizzato con un'approccio *strutturale*, presente al listato 2.5, di tutti i componenti visti precedentemente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoderBCD_on_board is
5     Port ( switch : in STD_LOGIC_VECTOR (9 downto 0);
6             led : out STD_LOGIC_VECTOR (3 downto 0);
7             anodes : out STD_LOGIC_VECTOR (7 downto 0);
8             cathodes : out STD_LOGIC_VECTOR (7 downto 0));
9 end encoderBCD_on_board;
10

```

```

11  architecture structural of encoderBCD_on_board is
12      component priority_encoder port(
13          x : in std_logic_vector(9 downto 0);
14          y : out std_logic_vector(3 downto 0) );
15      end component;
16
17      component cathodes_manager port(
18          value : in std_logic_vector(3 downto 0);
19          dot : in std_logic;
20          cathodes : out std_logic_vector(7 downto 0) );
21      end component;
22
23      signal w : std_logic_vector(3 downto 0);
24  begin
25      encoder : priority_encoder port map(
26          x => switch,
27          y => w );
28
29      cm : cathodes_manager port map(
30          value => w,
31          dot => '0',
32          cathodes => cathodes);
33
34      anodes <= x"fe";
35      led <= w;
36
37  end structural;

```

Listing 2.5: Encoder BCD on board

2.4 Simulazione

Di seguito si riportano i codici VHDL per l'implementazione dei *testbench* per ogni componente del progetto e i diagrammi temporali risultanti dalle simulazioni.

2.4.1 Encoder BCD

Per testare l'encoder BCD si è scelto di far variare l'input attraverso un ciclo for, in particolare ad ogni iterazione di ciclo diventa alto il bit di posizione i , cambiando così per ogni iterazione la codifica di uscita. Inoltre, si è testato anche il comportamento dell'encoder quando in ingresso viene fornita una string di bit con nessuno o più bit alti.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoderBCD_tb is
5 -- Port ( );
6 end encoderBCD_tb;
7
8 architecture Behavioral of encoderBCD_tb is
9     component encoderBCD port(
10         x : in std_logic_vector (9 downto 0);
11         y : out std_logic_vector (3 downto 0) );
12     end component;
13
14     signal input : std_logic_vector (9 downto 0) :=
15         "0000000000";
16     signal output : std_logic_vector (3 downto 0);
17
18     type oracle_type is array (0 to 9) of std_logic_vector(3
19         downto 0);
20     constant oracle : oracle_type := (
21         "0000", "0001", "0010", "0011", "0100", "0101" , "0110",
22         "0111", "1000", "1001");
23
24 begin
25     dut : encoderBCD port map(
26         x => input,
27         y => output );
28
29     process
30     begin
```

```

28      input <= "0000000001";
29      wait for 50ns;
30      assert output = oracle(0) report "error" severity
31          failure;
32
33      for i in 1 to 9 loop
34          input <= input(8 downto 0) & '0';
35          wait for 50ns;
36          assert output = oracle(i) report "error" severity
37              failure;
38      end loop;
39
40      input <= "0000000000";
41      wait for 50ns;
42      assert output = "----" report "error" severity failure;
43
44      input <= "1111111111";
45      wait for 50ns;
46      assert output = "----" report "error" severity failure;
47
48  end Behavioral;

```

Listing 2.6: Testbench encoder BCD



Figura 2.6: Waveform encoder BCD

2.4.2 Arbitro

Anche in questo caso per testare l' arbitro si è scelto di far variare l'input attraverso un ciclo for.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity arbiter_tb is
5 -- Port ( );
6 end arbiter_tb;
7
8 architecture Behavioral of arbiter_tb is
9 component arbiter port(
10     x : in std_logic_vector(9 downto 0);
11     y : out std_logic_vector(9 downto 0) );
12 end component;
13 signal input : std_logic_vector(9 downto 0) :=
14     "0000000000";
15 signal output : std_logic_vector(9 downto 0);
16
17 type oracle_type is array(0 to 9) of std_logic_vector(9
18     downto 0);
19 constant oracle : oracle_type := (
20     "0000000001",
21     "0000000010",
22     "0000000100",
23     "0000001000",
24     "0000010000",
25     "0000100000",
26     "0001000000",
27     "0010000000",
28     "0100000000",
29     "1000000000");
30 begin
31     --Design Under Test
32     dut : arbiter port map(
33         x => input,
34         y => output );
35
36     process
37     begin
38         input <= "0000000000";
39         wait for 50ns;
40         assert output = "-----" report "error" severity
41             failure;
42
43         for i in 0 to 9 loop

```

```

41      input(i) <= '1';
42      wait for 50ns;
43      assert output = oracle(i) report "error" severity
44          => failure;
45      end loop;
46  end process;
47 end Behavioral;

```

Listing 2.7: Testbench arbitro

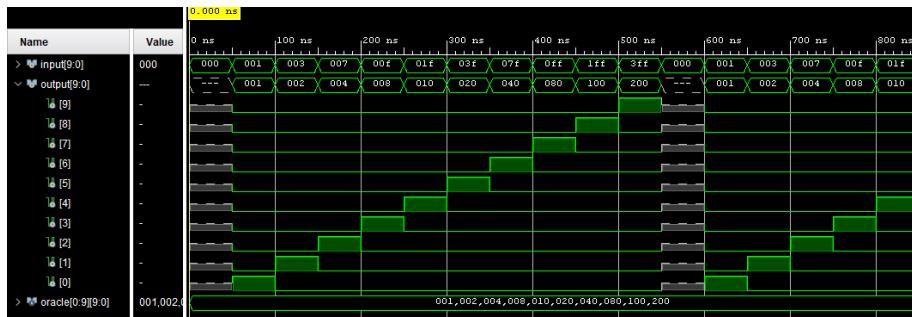


Figura 2.7: Waveform arbitro

2.4.3 Priority encoder BCD

Come nei casi precedenti, anche per il priority encoder BCD l'input varia ad ogni iterazione di un ciclo for, in particolare ad ogni iterazione di ciclo diventa alto il bit di posizione i .

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity priority_encoder_tb is
5 -- Port ();
6 end priority_encoder_tb;
7
8 architecture Behavioral of priority_encoder_tb is
9     component priority_encoder port(
10         x : in std_logic_vector(9 downto 0);
11         y : out std_logic_vector(3 downto 0) );
12     end component;
13     signal input : std_logic_vector(9 downto 0) :=
14         "0000000000";

```

```

14     signal output : std_logic_vector(3 downto 0);
15
16     type oracle_type is array(0 to 9) of std_logic_vector(3
17     <-- downto 0);
18     constant oracle : oracle_type := (
19         x"0", x"1", x"2", x"3", x"4", x"5", x"6", x"7", x"8",
20         <-- x"9" );
21 begin
22     --Design Under Test
23     dut : priority_encoder port map(
24         x => input,
25         y => output );
26
27     process
28     begin
29         input <= "0000000000";
30         wait for 50ns;
31         assert output = "----" report "error" severity failure;
32
33         for i in 0 to 9 loop
34             input(i) <= '1';
35             wait for 50ns;
36             assert output = oracle(i) report "error" severity
37             <-- failure;
38         end loop;
39     end process;
40
41 end Behavioral;

```

Listing 2.8: Testbench priority encoder BCD

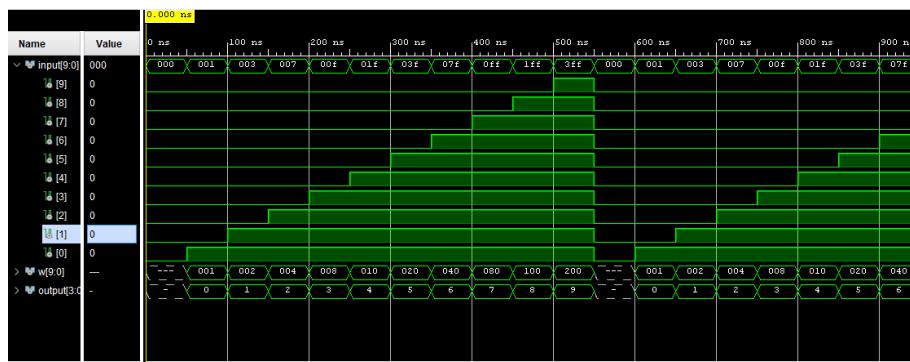


Figura 2.8: Waveform priority encoder BCD

2.4.4 Cathodes manager

Per testare il gestore dei catodi facciamo variare l'input in modo tale che ad ogni iterazione di ciclo tale input assume un valore binario che va da 0 a 15. Ad ognuno di questi ingressi verificheremo se il componente restituisca in uscita la codifica giusta per la rappresentazione di tale input sul display.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity cathodes_manager_tb is
6 -- Port ( );
7 end cathodes_manager_tb;
8
9 architecture Behavioral of cathodes_manager_tb is
10 component cathodes_manager port(
11     value : in std_logic_vector(3 downto 0);
12     dot : in std_logic;
13     cathodes : out std_logic_vector(7 downto 0) );
14 end component;
15 signal input_value : std_logic_vector(3 downto 0) :=
16     "0000";
17 signal input_dot : std_logic := '0';
18 signal output : std_logic_vector(7 downto 0);
19
20 type digit_type is array (0 to 15) of std_logic_vector(6
21     downto 0);
22 constant oracle : digit_type := (
23     "1000000", -- 0
24     "1111001", -- 1
25     "0100100", -- 2
26     "0110000", -- 3
27     "0011001", -- 4
28     "0010010", -- 5
29     "0000010", -- 6
30     "1111000", -- 7
31     "0000000", -- 8
32     "0010000", -- 9
33     "0001000", -- A
34     "0000011", -- b
35     "1000110", -- C
36     "0100001", -- d
37     "0000110", -- E
```

```

36      "0001110"); -- F
37  begin
38      -- Design Under Test
39      dut : cathodes_manager port map(
40          value => input_value,
41          dot => input_dot,
42          cathodes => output );
43
44  process
45  begin
46      input_dot <= '0';
47      for i in 0 to 15 loop
48          input_value <=
49              std_logic_vector(to_unsigned(i,input_value'length));
50          wait for 30ns;
51          assert output = '1' & oracle(i) report "error"
52              severity failure;
53      end loop;
54
55      input_dot <= '1';
56      for i in 0 to 15 loop
57          input_value <=
58              std_logic_vector(to_unsigned(i,input_value'length));
59          wait for 30ns;
60          assert output = '0' & oracle(i) report "error"
61              severity failure;
62      end loop;
63  end process;
64
65 end Behavioral;

```

Listing 2.9: Testbench cathodes manager

2.4.5 Encoder BCD on-board

La differenza tra l'encoder BCD "on-board" e il priority encoder BCD del paragrafo 2.4.3 è semplicemente l'aggiunta del cathodes manager. Avendo già testato il priority encoder BCD e il cathodes manager singolarmente, il focus del test "on-board" è verificare la corretta integrazione dei due componenti. Per questo motivo, faremo variare gli input in modo tale che ognuno di essi restituisca una codifica diversa delle 10 cifre decimali.

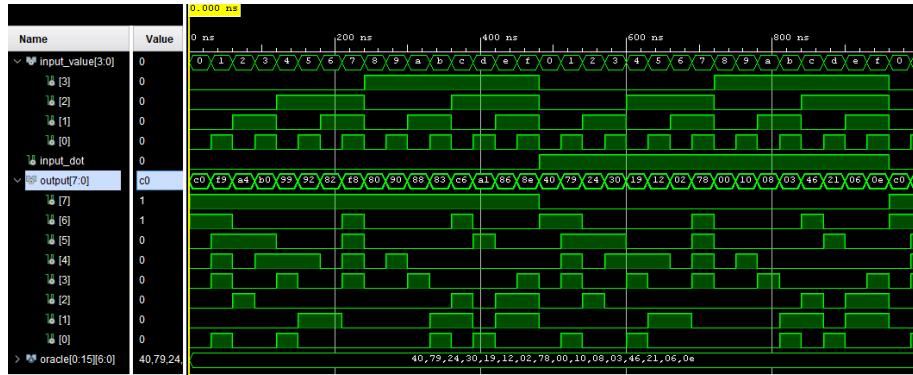


Figura 2.9: Waveform cathodes manager

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoderBCD_on_board_tb is
5   -- Port ( );
6 end encoderBCD_on_board_tb;
7
8 architecture Behavioral of encoderBCD_on_board_tb is
9   component encoderBCD_on_board port(
10     switch : in STD_LOGIC_VECTOR (9 downto 0);
11     led : out STD_LOGIC_VECTOR (3 downto 0);
12     anodes : out STD_LOGIC_VECTOR (7 downto 0);
13     cathodes : out STD_LOGIC_VECTOR (7 downto 0) );
14 end component;
15
16   signal input : STD_LOGIC_VECTOR (9 downto 0);
17   signal output_led : STD_LOGIC_VECTOR (3 downto 0);
18   signal output_anodes : STD_LOGIC_VECTOR (7 downto 0);
19   signal output_cathodes : STD_LOGIC_VECTOR (7 downto 0);
20
21   type oracle_led_type is array(0 to 9) of std_logic_vector(3
22   ↵  downto 0);
23   constant oracle_led : oracle_led_type := (
24     x"0", x"1", x"2", x"3", x"4", x"5", x"6", x"7", x"8",
25     ↵  x"9" );
26
27   type digit_type is array (0 to 15) of std_logic_vector(6
28   ↵  downto 0);
29   constant oracle_cathodes : digit_type := (
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
121
122
123
124
125
126
127
128
129
130
131
132
133
134
135
136
137
138
139
140
141
142
143
144
145
146
147
148
149
150
151
152
153
154
155
156
157
158
159
159
160
161
162
163
164
165
166
167
168
169
169
170
171
172
173
174
175
176
177
178
179
179
180
181
182
183
184
185
186
187
188
189
189
190
191
192
193
194
195
196
197
198
199
200
201
202
203
204
205
206
207
208
209
210
211
212
213
214
215
216
217
218
219
219
220
221
222
223
224
225
226
227
228
229
229
230
231
232
233
234
235
236
237
238
239
239
240
241
242
243
244
245
246
247
248
249
249
250
251
252
253
254
255
256
257
258
259
259
260
261
262
263
264
265
266
267
268
269
269
270
271
272
273
274
275
276
277
278
279
279
280
281
282
283
284
285
286
287
288
289
289
290
291
292
293
294
295
296
297
298
299
299
300
301
302
303
304
305
306
307
308
309
309
310
311
312
313
314
315
316
317
318
319
319
320
321
322
323
324
325
326
327
328
329
329
330
331
332
333
334
335
336
337
338
339
339
340
341
342
343
344
345
346
347
348
349
349
350
351
352
353
354
355
356
357
358
359
359
360
361
362
363
364
365
366
367
368
369
369
370
371
372
373
374
375
376
377
378
378
379
380
381
382
383
384
385
386
387
388
388
389
389
390
391
392
393
394
395
396
397
398
399
399
400
401
402
403
404
405
406
407
408
409
409
410
411
412
413
414
415
416
417
418
419
419
420
421
422
423
424
425
426
427
428
429
429
430
431
432
433
434
435
436
437
438
439
439
440
441
442
443
444
445
446
447
448
449
449
450
451
452
453
454
455
456
457
458
459
459
460
461
462
463
464
465
466
467
468
469
469
470
471
472
473
474
475
476
477
478
478
479
480
481
482
483
484
485
486
487
488
488
489
489
490
491
492
493
494
495
496
497
497
498
499
499
500
501
502
503
504
505
506
507
508
509
509
510
511
512
513
514
515
516
517
518
519
519
520
521
522
523
524
525
526
527
528
529
529
530
531
532
533
534
535
536
537
538
539
539
540
541
542
543
544
545
546
547
548
549
549
550
551
552
553
554
555
556
557
558
559
559
560
561
562
563
564
565
566
567
568
569
569
570
571
572
573
574
575
576
577
578
578
579
579
580
581
582
583
584
585
586
587
588
588
589
589
590
591
592
593
594
595
596
597
597
598
599
599
600
601
602
603
604
605
606
607
608
609
609
610
611
612
613
614
615
616
617
618
618
619
619
620
621
622
623
624
625
626
627
628
628
629
629
630
631
632
633
634
635
636
637
638
638
639
639
640
641
642
643
644
645
646
647
648
648
649
649
650
651
652
653
654
655
656
657
658
659
659
660
661
662
663
664
665
666
667
668
669
669
670
671
672
673
674
675
676
677
678
678
679
679
680
681
682
683
684
685
686
687
688
688
689
689
690
691
692
693
694
695
696
697
697
698
698
699
699
700
701
702
703
704
705
706
707
708
709
709
710
711
712
713
714
715
716
717
718
718
719
719
720
721
722
723
724
725
726
727
728
728
729
729
730
731
732
733
734
735
736
737
738
738
739
739
740
741
742
743
744
745
746
747
748
748
749
749
750
751
752
753
754
755
756
757
758
759
759
760
761
762
763
764
765
766
767
768
768
769
769
770
771
772
773
774
775
776
777
778
778
779
779
780
781
782
783
784
785
786
787
788
788
789
789
790
791
792
793
794
795
796
796
797
797
798
798
799
799
800
801
802
803
804
805
806
807
808
809
809
810
810
811
812
813
814
815
816
817
817
818
818
819
819
820
821
822
823
824
825
826
827
827
828
828
829
829
830
831
832
833
834
835
836
837
838
838
839
839
840
841
842
843
844
845
846
847
847
848
848
849
849
850
851
852
853
854
855
856
857
858
858
859
859
860
861
862
863
864
865
866
867
868
868
869
869
870
871
872
873
874
875
876
877
877
878
878
879
879
880
881
882
883
884
885
886
887
887
888
888
889
889
890
891
892
893
894
895
896
897
897
898
898
899
899
900
901
902
903
904
905
906
907
908
908
909
909
910
911
912
913
914
915
916
917
917
918
918
919
919
920
921
922
923
924
925
926
927
927
928
928
929
929
930
931
932
933
934
935
936
937
937
938
938
939
939
940
941
942
943
944
945
946
947
947
948
948
949
949
950
951
952
953
954
955
956
957
958
958
959
959
960
961
962
963
964
965
966
967
967
968
968
969
969
970
971
972
973
974
975
976
977
977
978
978
979
979
980
981
982
983
984
985
986
987
987
988
988
989
989
990
991
992
993
994
995
996
997
998
999
999
1000
1001
1002
1003
1004
1005
1006
1007
1008
1009
1009
1010
1010
1011
1012
1013
1014
1015
1016
1017
1017
1018
1018
1019
1019
1020
1021
1022
1023
1024
1025
1026
1027
1027
1028
1028
1029
1029
1030
1031
1032
1033
1034
1035
1036
1037
1037
1038
1038
1039
1039
1040
1041
1042
1043
1044
1045
1046
1047
1047
1048
1048
1049
1049
1050
1051
1052
1053
1054
1055
1056
1057
1058
1058
1059
1059
1060
1061
1062
1063
1064
1065
1066
1067
1067
1068
1068
1069
1069
1070
1071
1072
1073
1074
1075
1076
1077
1077
1078
1078
1079
1079
1080
1081
1082
1083
1084
1085
1086
1087
1087
1088
1088
1089
1089
1090
1091
1092
1093
1094
1095
1096
1097
1097
1098
1098
1099
1099
1100
1101
1102
1103
1104
1105
1106
1107
1107
1108
1108
1109
1109
1110
1111
1112
1113
1114
1115
1116
1116
1117
1117
1118
1118
1119
1119
1120
1121
1122
1123
1124
1125
1126
1126
1127
1127
1128
1128
1129
1129
1130
1131
1132
1133
1134
1135
1136
1137
1137
1138
1138
1139
1139
1140
1141
1142
1143
1144
1145
1146
1147
1147
1148
1148
1149
1149
1150
1151
1152
1153
1154
1155
1156
1157
1157
1158
1158
1159
1159
1160
1161
1162
1163
1164
1165
1166
1166
1167
1167
1168
1168
1169
1169
1170
1171
1172
1173
1174
1175
1176
1176
1177
1177
1178
1178
1179
1179
1180
1181
1182
1183
1184
1185
1186
1187
1187
1188
1188
1189
1189
1190
1191
1192
1193
1194
1195
1196
1196
1197
1197
1198
1198
1199
1199
1200
1201
1202
1203
1204
1205
1206
1206
1207
1207
1208
1208
1209
1209
1210
1211
1212
1213
1214
1215
1216
1216
1217
1217
1218
1218
1219
1219
1220
1221
1222
1223
1224
1225
1226
1226
1227
1227
1228
1228
1229
1229
1230
1231
1232
1233
1234
1235
1236
1237
1237
1238
1238
1239
1239
1240
1241
1242
1243
1244
1245
1246
1247
1247
1248
1248
1249
1249
1250
1251
1252
1253
1254
1255
1256
1257
1257
1258
1258
1259
1259
1260
1261
1262
1263
1264
1265
1266
1267
1267
1268
1268
1269
1269
1270
1271
1272
1273
1274
1275
1276
1276
1277
1277
1278
1278
1279
1279
1280
1281
1282
1283
1284
1285
1286
1287
1287
1288
1288
1289
1289
1290
1291
1292
1293
1294
1295
1296
1296
1297
1297
1298
1298
1299
1299
1300
1301
1302
1303
1304
1305
1306
1306
1307
1307
1308
1308
1309
1309
1310
1311
1312
1313
1314
1315
1316
1316
1317
1317
1318
1318
1319
1319
1320
1321
1322
1323
1324
1325
1326
1326
1327
1327
1328
1328
1329
1329
1330
1331
1332
1333
1334
1335
1336
1337
1337
1338
1338
1339
1339
1340
1341
1342
1343
1344
1345
1346
1347
1347
1348
1348
1349
1349
1350
1351
1352
1353
1354
1355
1356
1357
1357
1358
1358
1359
1359
1360
1361
1362
1363
1364
1365
1366
1366
1367
1367
1368
1368
1369
1369
1370
1371
1372
1373
1374
1375
1376
1376
1377
1377
1378
1378
1379
1379
1380
1381
1382
1383
1384
1385
1386
1387
1387
1388
1388
1389
1389
1390
1391
1392
1393
1394
1395
1396
1396
1397
1397
1398
1398
1399
1399
1400
1401
1402
1403
1404
1405
1406
1406
1407
1407
1408
1408
1409
1409
1410
1411
1412
1413
1414
1415
1416
1416
1417
1417
1418
1418
1419
1419
1420
1421
1422
1423
1424
1425
1426
1426
1427
1427
1428
1428
1429
1429
1430
1431
1432
1433
1434
1435
1436
1436
1437
1437
1438
1438
1439
1439
1440
1441
1442
1443
1444
1445
1446
1446
1447
1447
1448
1448
1449
1449
1450
1451
1452
1453
1454
1455
1456
1456
1457
1457
1458
1458
1459
1459
1460
1461
1462
1463
1464
1465
1466
1466
1467
1467
1468
1468
1469
1469
1470
1471
1472
1473
1474
1475
1476
1476
1477
1477
1478
1478
1479
1479
1480
1481
1482
1483
1484
1485
1486
1486
1487
1487
1488
1488
1489
1489
1490
1491
1492
1493
1494
1495
1496
1496
1497
1497
1498
1498
1499
1499
1500
1501
1502
1503
1504
1505
1506
1506
1507
1507
1508
1508
1509
1509
1510
1511
1512
1513
1514
1515
1516
1516
1517
1517
1518
1518
1519
1519
1520
1521
1522
1523
1524
1525
1526
1526
1527
1527
1528
1528
1529
1529
1530
1531
1532
1533
1534
1535
1536
1536
1537
1537
1538
1538
1539
1539
1540
1541
1542
1543
1544
1545
1546
1546
1547
1547
1548
1548
1549
1549
1550
1551
1552
1553
1554
1555
1556
1556
1557
1557
1558
1558
1559
1559
1560
1561
1562
1563
1564
1565
1566
1566
1567
1567
1568
1568
1569
1569
1570
1571
1572
1573
1574
1575
1576
1576
1577
1577
1578
1578
1579
1579
1580
1581
1582
1583
1584
1585
1586
1586
1587
1587
1588
1588
1589
1589
1590
1591
1592
1593
1594
1595
1596
1596
1597
1597
1598
1598
1599
1599
1600
1601
1602
1603
1604
1605
1606
1606
1607
1607
1608
1608
1609
1609
1610
1611
1612
1613
1614
1615
1616
1616
1617
1617
1618
1618
1619
1619
1620
1621
1622
1623
1624
1625
1626
1626
1627
1627
1628
1628
1629
1629
1630
1631
1632
1633
1634
1635
1636
1636
1637
1637
1638
1638
1639
1639
1640
1641
1642
1643
1644
1645
1646
1646
1647
1647
1648
1648
1649
1649
1650
1651
1652
1653
1654
1655
1656
1656
1657
1657
1658
1658
1659
1659
1660
1661
1662
1663
1664
1665
1666
1666
1667
1667
1668
1668
1669
1669
1670
1671
1672
1673
1674
1675
1676
1676
1677
1677
1678
1678
1679
1679
1680
1681
1682
1683
1684
1685
1686
1686
1687
1687
1688
1688
1689
1689
1690
1691
1692
1693
1694
1695
1696
1696
1697
1697
1698
1698
1699
1699
1700
1701
1702
1703
1704
1705
1706
1706
1707
1707
1708
1708
1709
1709
1710
1711
1712
1713
1714
1715
1716
1716
1717
1717
1718
1718
1719
1719
1720
1721
1722
1723
1724
1725
1726
1726
1727
1727
1728
1728
1729
1729
1730
1731
1732
1733
1734
1735
1736
1736
1737
1737
1738
1738
1739
1739
1740
1741
1742
1743
1744
1745
1746
1746
1747
1747
1748
1748
1749
1749
1750
1751
1752
1753
1754
1755
1756
1756
1757
1757
1758
1758
1759
1759
1760
1761
1762
1763
1764
1765
1766
1766
1767
1767
1768
1768
1769
1769
1770
1771
1772
1773
1774
1775
1776
1776
1777
1777
1778
1778
1779
1779
1780
1781
1782
1783
1784
1785
1786
1786
1787
1787
1788
1788
1789
1789
1790
1791
1792
1793
1794
1795
1796
1796
1797
1797
1798
1798
1799
1799
1800
1801
1802
1803
1804
1805
1806
1806
1807
1807
1808
1808
1809
1809
1810
1811
1812
1813
1814
1815
1816
1816
1817
1817
1818
1818
1819
1819
1820
1821
1822
1823
1824
1825
1826
1826
1827
1827
1828
1828
1829
1829
1830
1831
1832
1833
1834
1835
1836
1836
1837
1837
1838
1838
1839
1839
1840
1841
1842
1843
1844
1845
1846
1846
1847
1847
1848
1848
1849
1849
1850
1851
1852
1853
1854
1855
1856
1856
1857
1857
1858
1858
1859
1859
1860
1861
1862
1863
1864
1865
1866
1866
1867
1867
1868
1868
1869
1869
1870
1871
1872
1873
1874
1875
1876
1876
1877
1877
1878
1878
1879
1879
1880
1881
1882
1883
1884
1885
1886
1886
1887
1887
1888
1888

```

```

27      "1000000", -- 0
28      "1111001", -- 1
29      "0100100", -- 2
30      "0110000", -- 3
31      "0011001", -- 4
32      "0010010", -- 5
33      "0000010", -- 6
34      "1111000", -- 7
35      "0000000", -- 8
36      "0010000", -- 9
37      "0001000", -- A
38      "0000011", -- b
39      "1000110", -- C
40      "0100001", -- d
41      "0000110", -- E
42      "0001110");-- F
43 begin
44   -- Design Under Test
45   dut : encoderBCD_on_board port map(
46     switch => input,
47     led => output_led,
48     anodes => output_anodes,
49     cathodes => output_cathodes );
50
51 process
52 begin
53   input <= "0000000000";
54   wait for 50ns;
55   assert output_led = "----" report "error led" severity
56   -- failure;
57   assert output_anodes = x"fe" report "error anode"
58   -- severity failure;
59   --assert output_cathodes = "----" report "error led"
60   -- severity failure;
61
62   for i in 0 to 9 loop
63     input(i) <= '1';
64     wait for 50ns;
65     assert output_led = oracle_led(i) report "error
66     -- led" severity failure;
67     assert output_anodes = x"fe" report "error anode"
68     -- severity failure;
69     assert output_cathodes = '1' & oracle_cathodes(i)
70     -- report "error cathodes" severity failure;

```

```

65         end loop;
66     end process;
67
68 end Behavioral;

```

Listing 2.10: Testbench encoder BCD on-board



Figura 2.10: Waveform encoder BCD on-board

2.5 Sintesi

Il progetto dell'encoder BCD "on-board" è stato caricato sulla scheda Nexys A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente con gli elementi di I/O presenti sulla scheda. In particolare, tutte le linee di input sono state mappate sugli switch della scheda, mentre le quattro linee di output sono state mappate sui LED, secondo la configurazione in figura 2.11. Inoltre, per la corretta visualizzazione della cifra sul display, sono state mappate le otto linee di output del gestore dei catodi sui catodi del display, presente sulla board. In maniera analoga sono state mappate le linee sugli anodi, con l'unica differenza che qui la configurazione è fissa, ovvero un'unica cifra sul display resterà accesa (in particolare $AN = '11111110'$).

Il mapping tra le linee verso l'esterno del progetto sulla FPGA e i componenti fisici della board, è stato implementato modificando il file dei *constraints* "Nexys-A7-100T-Master.xdc".

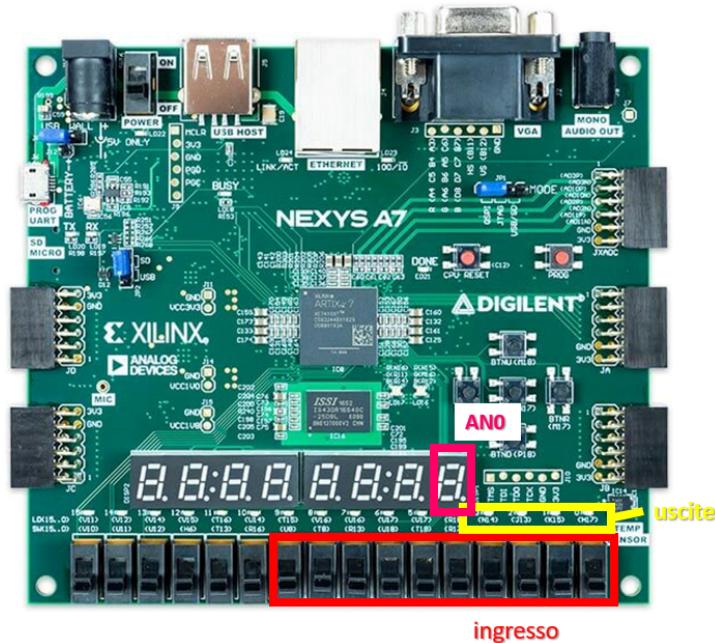


Figura 2.11: Schema di mapping dell'encoder BCD su board

Capitolo 3

Riconoscitore di sequenze

3.1 Traccia

Esercizio 3.1 Progettare, implementare in VHDL e testare mediante simulazione una macchina in grado di riconoscere la sequenza 1001. La macchina prende in ingresso un segnale binario i che rappresenta il dato, un segnale CLK di temporizzazione e un segnale M di modo, che ne disciplina il funzionamento, e fornisce un'uscita Y alta quando la sequenza viene riconosciuta. In particolare,

- se $M=0$, la macchina valuta i bit seriali in ingresso a gruppi di 4,
- se $M=1$, la macchina valuta i bit seriali in ingresso uno alla volta, tornando allo stato iniziale ogni volta che la sequenza viene correttamente riconosciuta.

Esercizio 3.2 Sintetizzare e implementare su board la rete sviluppata al punto precedente, utilizzando uno switch S1 per codificare l'input i e uno switch S2 per codificare il modo M , in combinazione con due bottoni B1 e B2 utilizzati rispettivamente per acquisire l'input da S1 e S2 in sincronismo con il segnale di temporizzazione A, che deve essere ottenuto a partire dal clock della board. Infine, l'uscita Y può essere codificata utilizzando un led.

3.2 Soluzione

Per il terzo esercizio si vuole realizzare un riconoscitore della sequenza **1001**. Ciò lo si realizza attraverso una macchina sequenziale, ovvero un sistema in cui l'uscita in un certo istante dipende non solo dal valore degli ingressi nello stesso istante, ma anche dagli ingressi precedenti. Le informazioni sull'attività passata del sistema ne costituiscono lo *stato*, che informalmente può essere definito come la memoria interna di un sistema. Il comportamento di una macchina sequenziale può essere rappresentato in maniera non ambigua tramite un *automa a stati finiti* (ASF), un modello matematico di "sistema" caratterizzato da un insieme finito di ingressi, uscite e stati. Le scelte per quanto riguarda la modellazione della macchina sequenziale possono essere due: progettare una macchina di Mealy, in cui l'uscita è una funzione tanto dello stato corrente quanto dell'ingresso, o una macchina di Moore, in cui invece l'uscita è una funzione solo dello stato corrente e non c'è dipendenza dell'uscita rispetto all'ingresso corrente.

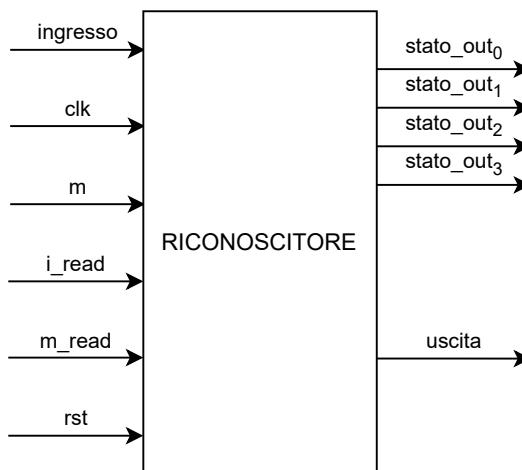


Figura 3.1: Schematico riconoscitore di sequenze

Il blocco base del progetto è dunque il riconoscitore (figura 3.1). Tale componente prende in ingresso un segnale di clock di temporizzazione, un segnale ingresso, un segnale *m* di modo, un segnale di reset della macchina e i due corrispettivi segnali di abilitazione a leggere rispettivamente l'ingresso ed *m*, ovvero *i_read* e *m_read*. In uscita il componente fornisce un segnale che impostato ad uno 1 quando viene riconosciuta correttamente la sequenza 1001. Abbiamo inoltre previsto un'uscita addizionale, un vettore di 4 elementi che indica la codifica dello stato corrente, che consente di verificare lo stato in cui si trova la macchina istante per istante. Il comportamento del riconoscitore di sequenze è stato modellato tramite un automa a stati finiti. In particolare, dato che il riconoscitore ha due modalità di funzionamento modelleremo il comportamento con due automi distinti. La modalità è indicata dalla

linea di ingresso m e al variare del modo si passerà da un automa all'altro, passando allo stato base di uno dei due automi, S_0 nel caso di $m=0$ e S_7 nel caso di $m=1$. Nel nostro progetto, abbiamo optato per l'implementazione di una macchina di Mealy. Questa scelta ci consente di visualizzare l'uscita, oltre all'ingresso, in ogni transizione di stato dell'automa. Questa decisione semplifica notevolmente il numero di stati, che è già considerevole con Mealy. Se avessimo scelto una macchina di Moore, avremmo dovuto aggiungere due nuovi stati in cui l'uscita del riconoscitore sarebbe stata alta, uno per $m=0$ e uno per $m=1$.

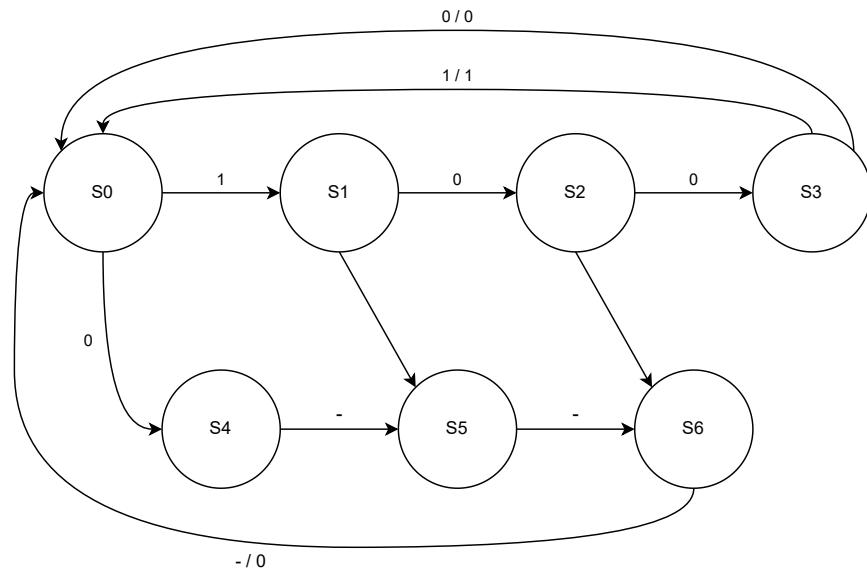


Figura 3.2: Automa riconoscitore in modalità $m = 0$

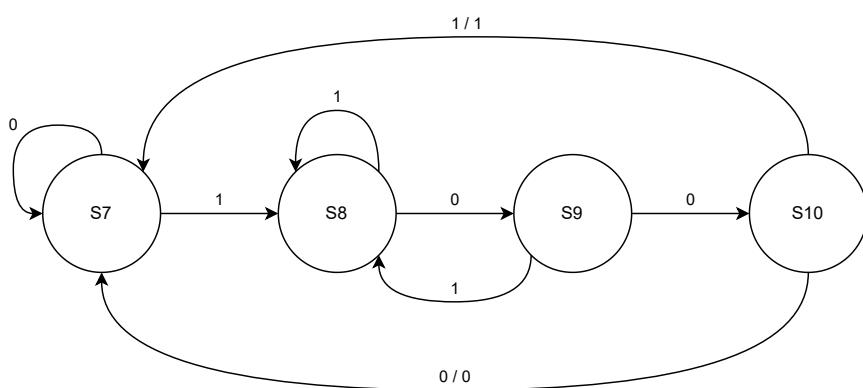


Figura 3.3: Automa riconoscitore in modalità $m = 1$

Quando m è uguale a zero, figura 3.2, il sistema deve riconoscere la sequenza a gruppi di 4 bit. Infatti ciò comporta un aumento degli stati,

dividendoli in due gruppi, un gruppo che rappresenta la sequenza corretta e l'altro la sequenza non corretta. In entrambi i casi, sia se viene riconosciuta la sequenza corretta che non, il comportamento della macchina evolve su quattro stati, dove solo la giusta sequenza porterà un'uscita alta ($S_0 \rightarrow S_1 \rightarrow S_2 \rightarrow S_3 \rightarrow S_0$, rispettivamente con ingressi "1-0-0-1"). Gli stati aggiuntivi (S_4, S_5, S_6) permettono alla macchina di entrare in un stato di errore, o meglio di sequenza non corretta, non appena uno dei 4 bit da riconoscere non è esatto. Quindi una volta entrato in uno di questi stati la macchina sicuramente, a prescindere dai futuri ingressi (che coprono i 4 bit della relativa sequenza da riconoscere) restituirà in uscita zero, ovvero sequenza non riconosciuta.

Invece quando m è uguale a 1, l'automa è più semplice in quanto il riconoscimento dei bit avviene uno alla volta, senza il vincolo di dover valutare entro 4 bit la presenza della sequenza target 1001, figura 3.3.

Altro elemento fondamentale è la presenza del segnale di *clock*, fornito in ingresso al sistema. Essendo il riconoscitore una macchina sequenziale, richiede un segnale di temporizzazione adeguato che faccia sì che la macchina passi da uno stato all'altro sul fronte di salita del segnale. Di conseguenza, il comportamento dell'intero sistema è sincronizzato con il *clock*, garantendo un funzionamento sincrono.

L'esercizio 3.2 prevede di caricare il progetto del riconoscitore di sequenze sulla board Nexys-A7 della Digilent. Per poterlo fare in maniera corretta è stato considerato il problema del "*button bouncing*" legato ai pulsanti utilizzati per acquisire l'input in sincronismo con il segnale di temporizzazione. Il problema principale riguarda l'oscillazione generata durante la pressione del pulsante, la quale deve essere attentamente gestita per prevenire comportamenti anomali all'interno del sistema. In questo esercizio è fondamentale risolvere tale problema, poiché tali pulsanti permettono di acquisire, a seguito della loro pressione, gli input, permettendo così di far evolvere la macchina verso nuovi stati. Introducendo quindi il componente **button_debouncer** ciò viene risolto (figura 3.4). Tale componente riceve come ingressi il *clock* (temporizzazione), il segnale del button e il segnale di reset, ed invece come uscita il segnale *btn_cleared*, ovvero un segnale "ripulito" che indica l'avvenuta pressione del button. In questo progetto, il button debouncer ci servirà per entrambi i bottoni, sia *i_read* per l'abilitazione a leggere l'input, che *m_read* per l'abilitazione a leggere il segnale *m* di modo.

Il button debouncer è un semplice dispositivo il cui comportamento può essere modellato tramite un automa a stati finiti, come mostrato in figura 3.5. Il dispositivo rimane in uno stato di "NOT_PRESSED" finché il pulsante non viene premuto. Una volta premuto, passa allo stato "PRESSED", e la macchina rimane in questo stato per un certo numero di colpi di *clock*, "max_count". Appena il conteggio dei colpi di *clock* arriva a *max_count*, si alza il segnale di output e la macchina torna allo stato NOT_PRESSED, qui il contatore viene azzerato e il segnale di output viene abbassato. Quindi, il segnale di output *btn_cleared* durerà per un solo colpo di *clock*, eliminando

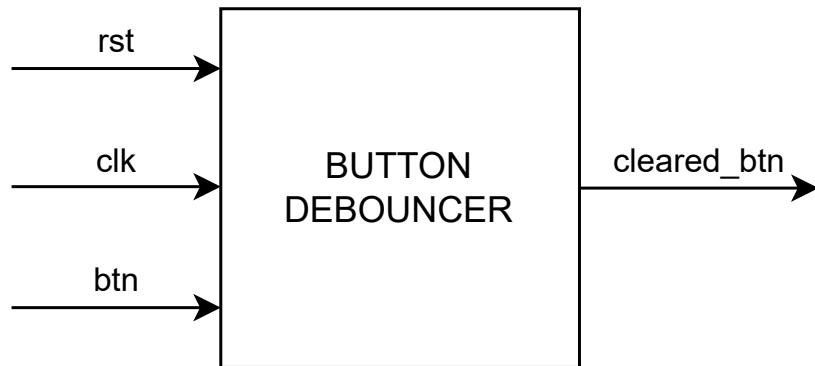


Figura 3.4: Button debouncer

tutte le oscillazioni che disturbavano il comportamento della macchina, in altri termini il button debouncer funge da una sorta di filtro che elimina le dinamiche oscillatorie alle alte frequenze.

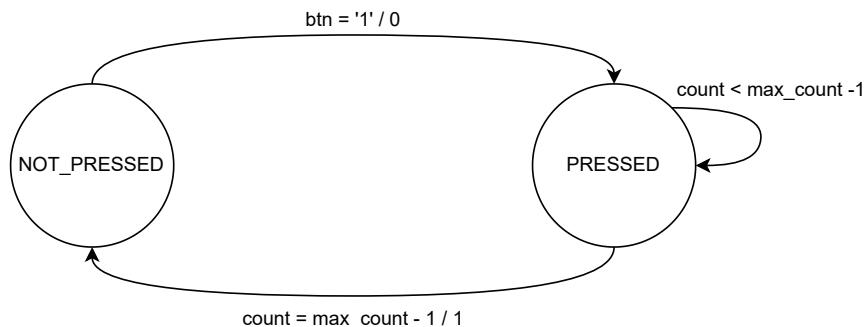


Figura 3.5: Automa button debouncer

Quindi per il progetto finale avremo che i due debouncer istanziati per i due bottoni verranno anteposti al riconoscitore, in modo tale che quest'ultimo abbia in ingresso i segnali di abilitazione puliti dal rumore che potrebbe causare malfunzionamenti o anomalie.

3.3 Codice

3.3.1 Riconoscitore

Il riconoscitore, blocco base del nostro progetto, è stato descritto secondo un'architettura *behavioral* come si evince dal listato 3.1. Per questo esercizio abbiamo deciso di modellare l'automa con un *unico processo* sensibile al solo clock, che descrive come deve evolvere la macchina solo in corrispondenza del fronte di salita. Oltre al processo, nell'architecture del riconoscitore abbiamo inserito anche una assegnazione selezionata riguardante il segnale sc. Infatti, essa ci permette di assegnare un determinato valore (in esadecimale) al vettore stato_out di 4 bit, a seconda del valore assunto dal segnale sc. In questo modo, possiamo controllare il valore dello stato corrente ad ogni colpo di clock, per poi visualizzarlo su 4 LED della scheda.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity riconoscitore is
5     Port ( ingresso : in STD_LOGIC;
6             clk : in STD_LOGIC;
7             m : in STD_LOGIC;
8             i_read : in STD_LOGIC;
9             m_read : in STD_LOGIC;
10            rst : in STD_LOGIC;
11            uscita : out STD_LOGIC;
12            stato_out : out STD_LOGIC_VECTOR(3 downto 0));
13 end riconoscitore;
14
15 architecture Behavioral of riconoscitore is
16 type stato is (s0,s1,s2,s3,s4,s5,s6,s7,s8,s9,s10);
17 signal sc : stato := s0; --stato corrente
18
19 begin
20     automa: process(clk)
21     begin
22         if(clk'event and clk = '1') then
23             if(rst = '1') then
24                 sc <= s0;
25                 uscita <= '0';
26             elsif(rst = '0') then
27                 if(m_read = '1') then
```

```

28      if(m = '1' and (sc = s0 or sc = s1 or sc =
29          ↪  s2 or sc = s3 or sc = s4 or sc = s5 or
30          ↪  sc = s6)) then
31          sc <= s7;
32          uscita <= '0';
33      elsif (m = '0' and (sc = s7 or sc = s8 or
34          ↪  sc = s9 or sc = s10)) then
35          sc <= s0;
36          uscita <= '0';
37      end if;
38      elsif(i_read ='1') then
39          case sc is
40              when s0 =>
41                  if(ingresso = '0')then
42                      sc <= s4;
43                      uscita <= '0';
44                  elsif(ingresso = '1') then
45                      sc <= s1;
46                      uscita <= '0';
47                  end if;
48              when s1 =>
49                  if(ingresso = '0')then
50                      sc <= s2;
51                      uscita <= '0';
52                  elsif(ingresso = '1') then
53                      sc <= s5;
54                      uscita <= '0';
55                  end if;
56              when s2 =>
57                  if(ingresso = '0')then
58                      sc <= s3;
59                      uscita <= '0';
60                  elsif(ingresso = '1') then
61                      sc <= s6;
62                      uscita <= '0';
63                  end if;
64              when s3 =>
65                  if(ingresso = '0')then
66                      sc <= s0;
67                      uscita <= '0';
68                  elsif(ingresso = '1') then
                      sc <= s0;
                      uscita <= '1';
                  end if;

```

```

69      when s4 =>
70          sc <= s5;
71          uscita <= '0';
72      when s5 =>
73          sc <= s6;
74          uscita <= '0';
75      when s6 =>
76          sc <= s0;
77          uscita <= '0';
78      when s7 =>
79          if(ingresso = '0')then
80              sc <= s7;
81              uscita <= '0';
82          elsif(ingresso = '1') then
83              sc <= s8;
84              uscita <= '0';
85          end if;
86      when s8 =>
87          if(ingresso = '0')then
88              sc <= s9;
89              uscita <= '0';
90          elsif(ingresso = '1') then
91              sc <= s8;
92              uscita <= '0';
93          end if;
94      when s9 =>
95          if(ingresso = '0')then
96              sc <= s10;
97              uscita <= '0';
98          elsif(ingresso = '1') then
99              sc <= s8;
100             uscita <= '0';
101         end if;
102     when s10 =>
103         if(ingresso = '0')then
104             sc <= s7;
105             uscita <= '0';
106         elsif(ingresso = '1') then
107             sc <= s7;
108             uscita <= '1';
109         end if;
110     end case;
111 end if;
112 end if;

```

```

113         end if;
114     end process;
115
116     with sc select
117         stat0_out <= x"0" when s0,
118                     x"1" when s1,
119                     x"2" when s2,
120                     x"3" when s3,
121                     x"4" when s4,
122                     x"5" when s5,
123                     x"6" when s6,
124                     x"7" when s7,
125                     x"8" when s8,
126                     x"9" when s9,
127                     x"a" when s10,
128                     x"f" when others;
129 end Behavioral;

```

Listing 3.1: Riconoscitore di sequenze

3.3.2 Button debouncer

Il button debouncer è stato descritto anch'esso secondo un'architettura *behavioral*. Nell'entity vengono specificati due generics, permettendo di parametrizzarne la descrizione, in particolare:

- CLK_PERIOD è il periodo del clock, di tipo intero, inizializzato a 10,
- btn_noise_time è l'intervallo di tempo Δt del bottone, sempre di tipo intero, inizializzato a 6500000.

In questo progetto assumiamo Δt , 65000000 ns, molto superiore a quello strettamente necessario per l'estinzione del transitorio allo scopo di evitare che una pressione prolungata del bottone venisse erroneamente letta come una sequenza di più ingressi e dunque causasse un'evoluzione indesiderata del sistema. Come anche nel caso precedente, il funzionamento del componente è descritto da un automa a stati finiti modellato da un unico processo sensibile al clock. La sua evoluzione è descritta al listato 3.2.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity button_debouncer is

```

```

6   generic (
7     clk_period : integer := 10; -- periodo del clock
8     btn_noise_time : integer := 6500000 -- stima della
9       → durata del transitorio del button
10    );
11   Port ( rst : in STD_LOGIC;
12         clk : in STD_LOGIC;
13         btn : in STD_LOGIC;
14         cleared_btn : out STD_LOGIC);
15 end button_debouncer;
16
17 architecture Behavioral of button_debouncer is
18   type state_type is (NOT_PRESSED, PRESSED);
19   signal btn_state : state_type := NOT_PRESSED;
20   constant max_count : integer := btn_noise_time/clk_period;
21   → -- 6500000/10 = 650000 [colpi di clock]
22 begin
23   deb: process (clk)
24     variable count : integer := 1;
25   begin
26     if rising_edge(clk) then
27       if( rst = '1') then
28         btn_state <= NOT_PRESSED;
29         cleared_btn <= '0';
30         count := 0;
31       else
32         case btn_state is
33           when NOT_PRESSED =>
34             cleared_btn <= '0';
35             if( btn = '1' ) then
36               btn_state <= PRESSED;
37             else
38               btn_state <= NOT_PRESSED;
39             end if;
40           when PRESSED =>
41             if(count = max_count) then
42               count := 1;
43               cleared_btn <= '1';
44               btn_state <= NOT_PRESSED;
45             else
46               count := count + 1;
47               btn_state <= PRESSED;
48             end if;
49           when others =>

```

```

48                     btn_state <= NOT_PRESSED;
49             end case;
50         end if;
51     end if;
52 end process;
53
54 end Behavioral;
```

Listing 3.2: Button debouncer

3.3.3 Riconoscitore di sequenze on-board

Il riconoscitore di sequenze on-board è stato realizzato con un'approccio *strutturale*, presente al listato 3.3, di tutti i componenti visti precedentemente. Nella parte dichiarativa dell'architecture vengono quindi definiti i tre componenti, che verranno usati per il nostro sistema, e due segnali interni cleared_i e cleared_m di tipo STD_LOGIC.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity riconoscitore_on_board is
5     Port ( switch_i : in STD_LOGIC;
6             clk : in STD_LOGIC;
7             switch_m : in STD_LOGIC;
8             butt_i : in STD_LOGIC;
9             butt_m : in STD_LOGIC;
10            butt_rst : in STD_LOGIC;
11            led_uscita : out STD_LOGIC;
12            led_stato : out STD_LOGIC_VECTOR(3 downto 0));
13 end riconoscitore_on_board;
14
15 architecture Behavioral of riconoscitore_on_board is
16
17 signal cleared_i : std_logic;
18 signal cleared_m : std_logic;
19
20 begin
21     butt1_i: entity work.button_debouncer generic map (
22             clk_period => 10 , btn_noise_time => 650000000
23         ) port map ( rst => butt_rst, clk => clk, btn => butt_i,
24             ↳ cleared_btn => cleared_i);
```

```

25      butt2_m: entity work.button_debouncer generic map (
26          clk_period => 10 , btn_noise_time => 650000000
27      ) port map ( rst => butt_RST, clk => CLK, btn => BUTT_M,
28                  ↵ cleared_BTN => Cleared_M);
29
30      automa1: entity work.riconoscitore port map(
31          ingresso => Switch_I, CLK => CLK, M => Switch_M,
32          I_read => Cleared_I, M_read => Cleared_M, RST =>
33          ↵ BUTT_RST,
34          Uscita => LED_Uscita, Stato_Out => LED_Stato);
35
36  end Behavioral;

```

Listing 3.3: Riconoscitore di sequenze on-board

3.4 Simulazione

3.4.1 Riconoscitore di sequenze

A questo punto del progetto, abbia simulato il nostro componente con un apposito test bench (listato 3.4). Innanzitutto, abbiamo creato un process `clock_tb` che simula un andamento periodico del clock, come un'onda quadra di periodo 80 ns, dopodiché abbiamo creato un altro process, chiamato `inputs_tb`, in cui simuliamo il comportamento del riconoscitore, fornendo una sequenza di ingressi `i`, preceduti chiaramente dal segnale di abilitazione `i_read`. In questa fase, abbiamo testato il comportamento della macchina quando la sequenza viene riconosciuta correttamente in entrambi le modalità.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity riconoscitore_tb is
5 end riconoscitore_tb;
6
7 architecture Behavioral of riconoscitore_tb is
8     component riconoscitore is port(
9         ingresso : in STD_LOGIC;
10        clk : in STD_LOGIC;
11        m : in STD_LOGIC;
12        i_read : in STD_LOGIC;
13        m_read : in STD_LOGIC;
14        rst : in STD_LOGIC;
15        uscita : out STD_LOGIC;
16        stato_out : out STD_LOGIC_VECTOR(3 downto 0)
17    );
18 end component;
19
20     signal ingresso : std_logic := '0';
21     signal clk : std_logic := '0';
22     signal m : std_logic := '0';
23     signal i_read : std_logic := '0';
24     signal m_read : std_logic := '0';
25     signal rst : std_logic := '0';
26     signal uscita : std_logic := '0';
27     signal stato_out : std_logic_vector(3 downto 0);
28
29 begin
30
31     r: riconoscitore port map (
```

```

32         ingresso => ingresso,
33         clk => clk,
34         m => m,
35         i_read => i_read,
36         m_read => m_read,
37         rst => rst,
38         uscita => uscita,
39         stato_out => stato_out
40     );
41
42     clock_tb: process begin
43         wait for 40 ns;
44         clk <= '1';
45         wait for 40 ns;
46         clk <='0';
47     end process;
48
49     inputs_tb: process begin
50         --m = 0
51         --         wait for 40 ns;
52         --         i_read <= '0';
53         --         m_read <= '1';
54         --         m <= '0';
55         --         wait for 20 ns;
56         --         m_read <= '0';
57         --         ingresso <= '1';
58         --         wait for 20 ns;
59         --         i_read <='1';
60         --         wait for 80 ns;
61         --         i_read <= '0';
62         --         wait for 20 ns;
63         --         ingresso <= '0';
64         --         wait for 20 ns;
65         --         i_read <= '1';
66         --         wait for 80 ns;
67         --         i_read <= '0';
68         --         wait for 20 ns;
69         --         ingresso <= '0';
70         --         wait for 20 ns;
71         --         i_read <= '1';
72         --         wait for 80 ns;
73         --         i_read <= '0';
74         --         wait for 20 ns;
75         --         ingresso <= '1';

```

```

76    --      wait for 20 ns;
77    --      i_read <= '1';
78    --      wait for 80 ns;
79    --      i_read <= '0';
80    --      wait for 20 ns;
81    --      i_read <= '1';
82    --      ingresso <= '0';
83    --      wait for 80 ns;
84    --      i_read <= '0';
85    --      wait for 20 ns;
86    --      rst <= '1';
87    --      wait for 80 ns;
88    --      rst <= '0';
89    --      wait for 20 ns;
90    --      i_read <= '1';
91    --      ingresso <= '1';
92    --      wait for 80 ns;
93    --      i_read <= '0';

94

95

96      --m = 1
97      wait for 40 ns;
98          i_read <= '0';
99          m_read <= '1';
100         m <= '1';
101         wait for 80 ns;
102         m_read <= '0';
103         ingresso <= '1';
104         wait for 20 ns;
105         i_read <='1';
106         wait for 80 ns;
107         i_read <= '0';
108     --      wait for 20 ns;
109     --      m <= '0';
110     --      m_read <= '1';
111     --      wait for 80 ns;
112     --      m_read <= '0';
113     --      wait for 20 ns;
114     --      ingresso <= '0';
115     --      wait for 20 ns;
116     --      i_read <= '1';
117     --      wait for 80 ns;
118     --      i_read <= '0';
119     --      wait for 20 ns;

```

```

120      ingresso <= '1';
121      i_read <= '1';
122      wait for 80 ns;
123      i_read <= '0';
124      ingresso <= '0';
125      wait for 20 ns;
126      i_read <= '1';
127      wait for 80 ns;
128      i_read <= '0';
129      wait for 20 ns;
130      i_read <= '1';
131      wait for 80 ns;
132      i_read <= '0';
133      wait for 20 ns;
134      ingresso <= '1';
135      i_read <= '1';
136      wait for 80 ns;
137      i_read <= '0';
138      wait for 20 ns;
139      ingresso <= '1';
140      i_read <= '1';
141      wait for 80 ns;
142      i_read <= '0';
143      wait for 20 ns;
144      rst <= '1';
145      wait for 80 ns;
146      rst <= '0';
147      wait for 20 ns;
148      ingresso <= '1';
149      i_read <= '1';
150      wait for 80 ns;
151      i_read <= '0';

152      wait;
153  end process;
154
155
156 end Behavioral;

```

Listing 3.4: Testbench riconoscitore di sequenze



Figura 3.6: Waveform riconoscitore $m = 0$



Figura 3.7: Waveform riconoscitore $m = 1$

3.5 Sintesi

Il progetto del riconoscitore "on-board" è stato caricato sulla scheda Nexys A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente con gli elementi di I/O presenti sulla scheda. In particolare, le due linee di input, l'ingresso e m, sono state mappate su due switch della scheda, le quattro linee di output che rappresentano lo stato corrente e l'uscita sono state mappate sui LED. Inoltre, i segnali di abilitazione di lettura degli ingressi sugli switch e il segnale di reset sono stati mappato sui pulsanti presenti sulla board, secondo la configurazione in figura 3.8.

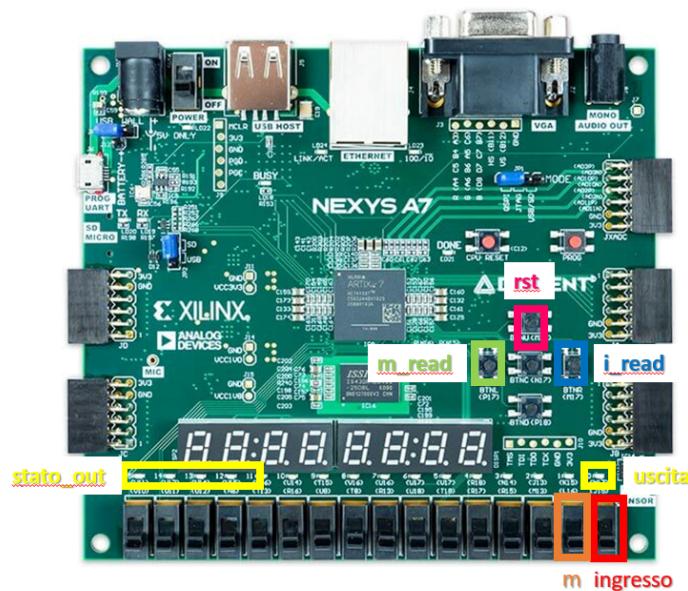


Figura 3.8: Schema di mapping del riconoscitore di sequenze su board

Capitolo 4

Shift register

4.1 Traccia

Esercizio 4.1 Progettare, implementare in VHDL e testare mediante simulazione un registro a scorrimento di N bit in grado di shiftare a destra o a sinistra di un numero Y variabile di posizioni a seconda di una opportuna selezione. Il componente deve essere realizzato utilizzando sia un a) approccio comportamentale sia un b) approccio strutturale.

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

4.2 Soluzione

Per il terzo esercizio si vuole realizzare un componente fondamentale, lo shift register. Gli shift register si dimostrano efficaci in operazioni di moltiplicazione (mediante shift verso destra) e di divisione (attraverso shift verso sinistra). In risposta alle specifiche della traccia, abbiamo adottato sia un approccio comportamentale che strutturale. Inoltre abbiamo optato per un approccio circolare nella realizzazione di questo progetto, consentendo simultaneamente il caricamento e la lettura parallela dei dati memorizzati. Riguardo all'approccio comportamentale abbiamo deciso di implementare le equazioni che regolano il componente attraverso un semplice codice contenuto all'interno di un process. Il comportamento della macchina è controllato da più ingressi, i quali riescono ad individuare l'ampiezza e la direzione dello shift. Inoltre, il componente è stato realizzato in maniera sincrona e che quindi risulta abilitato in corrispondenza dei fronti di salita del clock.

Invece per quanto riguarda la realizzazione del componente secondo un approccio strutturale, è stato realizzato connettendo più componenti elementari fra di loro. In particolare, abbiamo iniziato con l'implementazione del singolo flip-flop D, per poi procedere collegando tutti i flip-flop utilizzati attraverso una rete di interconnessione costituita da multiplexer. In particolare i multiplexer 2:1 permettono di selezionare l'ingresso parallelo e i multiplexer 4:1 di effettuare effettivamente lo shift dei bit dato. Dunque per la realizzazione di uno shift register di N bit secondo un approccio strutturale con al più due shift bidirezionali, avremo bisogno di N flip-flop D, N multiplexer 2:1 e N multiplexer 4:1 (collegati come mostrato in figura 4.3).

Un flip-flop D è un elemento di memoria digitale che conserva uno stato binario (0 o 1). Tale componente in questo esercizio è stato realizzato in maniera tale da essere sensibile al fronte di salita del segnale di *clock* e può essere azzerato mediante un segnale di *reset*. Inoltre è stato aggiunto un ulteriore segnale di abilitazione, che chiameremo segnale di *enable*. In figura 4.1 è possibile osservare lo schematico del componente.

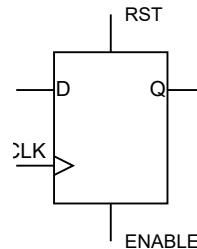


Figura 4.1: Schematico flip-flop D

Lo shift register (figura 4.2) è dotato di quattro segnali di controllo in ingresso: *clock*, *reset*, *enable* e *load*. Lo shift register, in esame, legge

in parallelo un vettore di ingresso composto da N bit, rappresentato da *in_parallel*. Inoltre, il sistema accetta un vettore *shifts* di due bit come input, il quale indica le diverse modalità di shift del registro a scorrimento. Le opzioni sono:

- "00": i dati scorrono di una posizione da destra verso sinistra,
- "01": i dati scorrono di due posizioni da destra verso sinistra,
- "10": i dati scorrono di una posizione da sinistra verso destra,
- "11": i dati scorrono di due posizioni da sinistra verso destra.

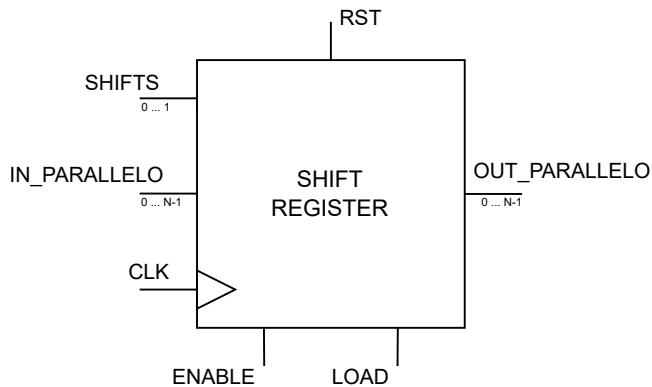


Figura 4.2: Shift register

L'uscita del sistema è rappresentata dal vettore *out_parallel* di N elementi, che riflette lo stato del registro e dei suoi N bit dopo l'esecuzione della traslazione. Una caratteristica distintiva di questo registro a scorrimento è la sua natura circolare. Quando si effettuano gli shift nelle posizioni intermedie, i bit vengono spostati a destra o a sinistra di una o due posizioni, a seconda del valore di *shifts*. Tuttavia, per i bit agli estremi del registro, viene implementata una soluzione circolare, ovvero utilizzando l'operatore di modulo, i dati vengono traslati dalle ultime posizioni alle prime durante uno shift verso destra e viceversa durante uno shift verso sinistra.

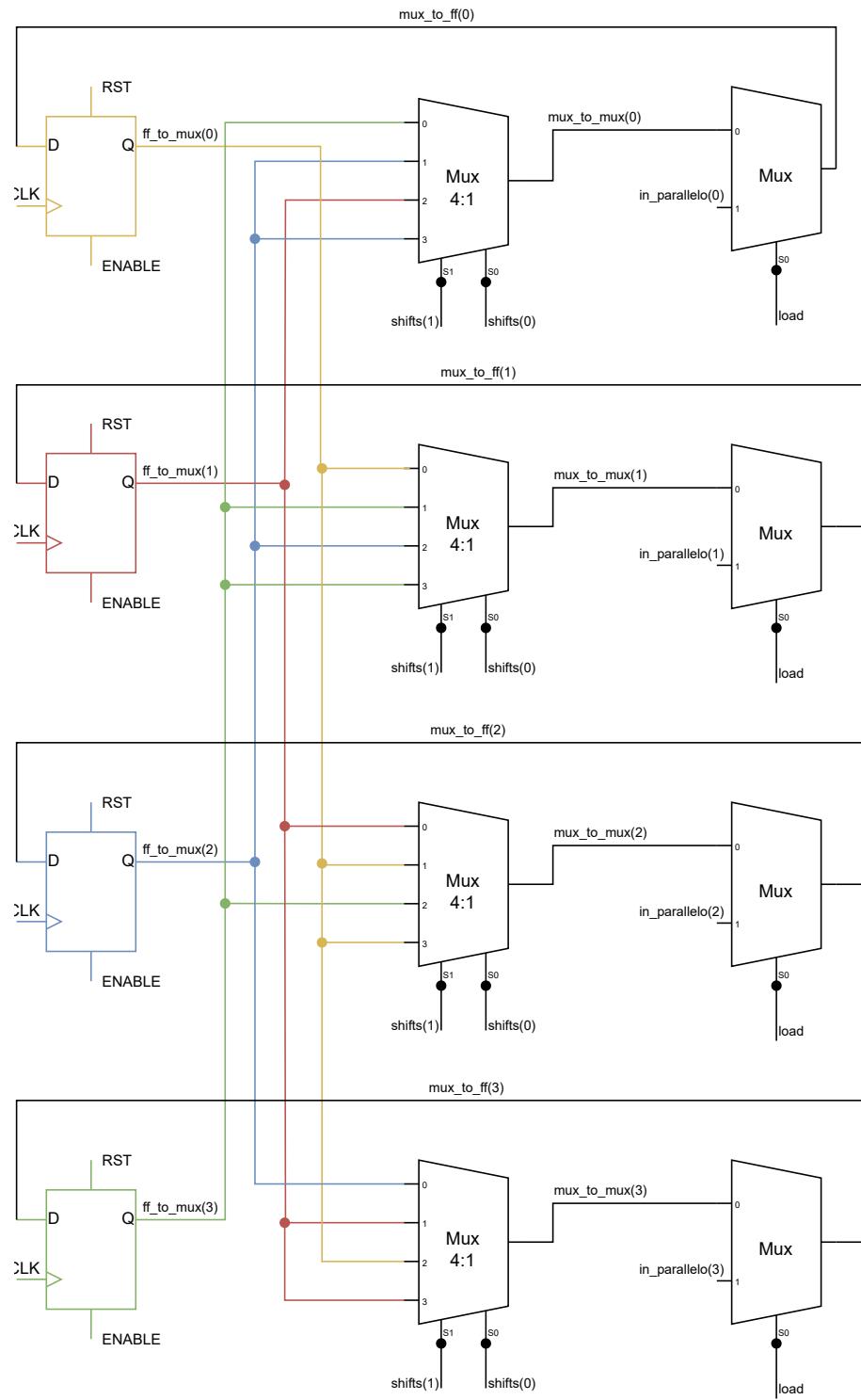


Figura 4.3: Shift register top module per $N = 4$

4.3 Codice

4.3.1 Flip-Flop D

Come detto in precedenza, il punto iniziale per la costruzione strutturale di uno shift register consiste nella realizzazione della cella elementare di memoria, ovvero il flip-flop D. Nel nostro progetto, il componente è stato progettato secondo una architettura behavioral e configurato in modo tale che fosse sensibile al fronte di salita del segnale di clock. Questo componente non solo è influenzato dal segnale di abilitazione (*enable*), ma anche dal *clock* e dal segnale di *reset*. La sua descrizione VHDL è presentata al listato 4.1.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity flip_flopD is
5     Port ( d : in STD_LOGIC;
6             clk : in STD_LOGIC;
7             rst : in STD_LOGIC;
8             enable : in STD_LOGIC;
9             q : out STD_LOGIC);
10 end flip_flopD;
11
12 architecture Behavioral of flip_flopD is
13 begin
14     process(clk)
15     begin
16         if( clk'event and clk = '1' ) then
17             if( rst = '1' ) then
18                 q <= '0';
19             elsif(enable = '1') then
20                 q <= d;
21             end if;
22         end if;
23     end process;
24
25 end Behavioral;
```

Listing 4.1: Flip-flop D

4.3.2 Approccio strutturale

Lo shift register descritto secondo un approccio strutturale è realizzato mediante un generic e sette porte. Il generic, identificato come N, è stato

inizializzato con un valore di default pari a 8, indicando il numero di bit conservati nel registro. All'interno dell'architecture, abbiamo definito le interfacce per i tre componenti essenziali coinvolti nella costruzione dello shift register. Inoltre, sono stati dichiarati tre vettori di N bit: *mux_to_ff*, *ff_to_mux*, e *mux_to_mux*. Questi rappresentano, rispettivamente, i segnali interni tra i multiplexer 2:1 e i flip-flop D, quelli tra i flip-flop D e i multiplexer 4:1, e infine, quelli tra i multiplexer 4:1 e i flip-flop D. Attraverso l'uso del costrutto FOR...GENERATE, sono stati istanziati N flip-flop D, N multiplexer 2:1 e N multiplexer 4:1 (vedi figura 4.3).

I multiplexer 2:1 utilizzano il segnale *load* come segnale di selezione. Se questo segnale è impostato a '0', la selezione avviene sull'uscita del multiplexer 4:1 (*mux_to_mux*), che esegue lo shift dei dati. Il segnale *mux_to_ff* viene quindi instradato all'ingresso del flip-flop i-esimo. Al contrario, se *load* è impostato a '1', il multiplexer prende in ingresso *in_parallel*, inserito dall'utente, e lo instrada al flip-flop corrispondente attraverso *mux_to_ff*.

Invece, i multiplexer 4:1 permettono l'effettivo shift dei bit. Tali multiplexer utilizzano come segnale di selezione il vettore di ingresso *shifts*, che indicano l'ampiezza e la direzione della traslazione (come mostrato precedentemente) ed in base a questo segnale si assegna al flip-flop i-esimo, attraverso il vettore *mux_to_mux*, il nuovo dato.

Infine, i flip-flop D hanno il compito di memorizzare il bit di informazione salvato nella i-esima posizione del registro a scorrimento. Ricevono in ingresso l'uscita del multiplexer 2:1 (*mux_to_ff*) e producono l'uscita *ff_to_mux*, indirizzata al multiplexer 4:1 per effettuare lo shift.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity shift_register_strutturale is
5     generic (
6         N : integer := 8);
7     Port ( in_parallel : in STD_LOGIC_VECTOR (N-1 downto 0);
8             clk : in STD_LOGIC;
9             rst : in STD_LOGIC;
10            load : in STD_LOGIC;
11            enable : in STD_LOGIC;
12            shifts : in STD_LOGIC_VECTOR (1 downto 0);
13            out_parallel : out STD_LOGIC_VECTOR (N-1 downto
14                                         0));
14 end shift_register_strutturale;
15
16 architecture Structural of shift_register_strutturale is
17     signal mux_to_ff : std_logic_vector(N-1 downto 0);

```

```

18     signal ff_to_mux : std_logic_vector(N-1 downto 0);
19     signal mux_to_mux : std_logic_vector(N-1 downto 0);
20
21
22 begin
23
24     out_parallel <= ff_to_mux;
25
26     SR: for i in 0 to N-1 generate
27         FF : entity work.flip_flopD port map(
28             d=> mux_to_ff(i), clk => clk, rst => rst, enable =>
29             enable,
30             q => ff_to_mux(i)
31         );
32
33         MUX2_1 : entity work.mux2to1 port map(
34             x(0) => mux_to_mux(i), x(1) => in_parallel(i), s
35             => load,
36             y => mux_to_ff(i));
37
38         MUX4_1 : entity work.mux4to1 port map(
39             x(0) => ff_to_mux((i-1) mod N), x(1) =>
40             ff_to_mux((i-2) mod N), x(2) => ff_to_mux((i+1)
41             mod N), x(3) => ff_to_mux((i+2) mod N),
42             s => shifts, y => mux_to_mux(i)
43         );
44
45     end generate;
46 end Structural;

```

Listing 4.2: Shift register: approccio strutturale

4.3.3 Approccio comportamentale

A differenza dell'approccio strutturale, lo shift register con approccio comportamentale, è stato progettato in modo tale da poter effettuare uno shift fino a 15 posizioni in entrambe le direzioni. Tale significativa ottimizzazione delle prestazioni deriva dalla semplificazione del circuito. Mentre, a livello strutturale, uno shift di 2 posizioni avrebbe richiesto l'utilizzo di N multiplexer 4:1, estendere tale principio a uno shift di 16 posizioni avrebbe comportato la progettazione di N multiplexer 32:1, rendendo il circuito del progetto notevolmente più complesso. Tuttavia, tale maggiore efficienza comporta il rischio di rendere la macchina non più sintetizzabile su FPGA, poiché il suo

comportamento è stato definito attraverso un processo altamente algoritmico. Lo shift register a livello comportamentale mantiene somiglianze con quella strutturale, ma introduce delle piccole variazioni nei segnali di ingresso. In particolare, *shifts* è ora un vettore di 4 bit, e si aggiunge un nuovo segnale di ingresso, *is_right*, che, quando impostato a '0', indica uno shift verso sinistra, altrimenti verso destra. L'architettura del registro a scorrimento, definito a livello comportamentale, incorpora un processo sensibile al fronte di salita del segnale di clock. Se *load*='1', i bit vengono impostati in base al valore del segnale *in_parallel*.

La gestione dello shift avviene in base al valore di *is_right*. Nel caso di uno shift verso sinistra, i bit da 0 a N-1-shifts vengono traslati di shifts posizioni, dove *shifts* è interpretato come un numero intero anche se presentato come un vettore di quattro bit. Tuttavia, i restanti bit del registro, ovvero da N-shifts a N-1, a causa della circolarità, vengono traslati nelle prime posizioni, precedentemente liberate dallo shift precedente. Un ragionamento analogo si applica allo shift verso destra. Di seguito al listato 4.3 viene presentato il codice dello shift register analizzato in questo paragrafo.

```

1 library IEEE;
2 use ieee.std_logic_1164.all;
3 use ieee.std_logic_unsigned.all;
4
5 entity shift_register_comportamentale is
6     generic( N : integer := 8);
7     Port ( in_parallel : in STD_LOGIC_VECTOR (N-1 downto 0);
8             clk : in STD_LOGIC;
9             rst : in STD_LOGIC;
10            load : in STD_LOGIC;
11            enable : in STD_LOGIC;
12            shifts : in STD_LOGIC_VECTOR (3 downto 0);
13            is_right : in STD_LOGIC;
14            out_parallel : out STD_LOGIC_VECTOR (N-1 downto
15                                         ← 0));
15 end shift_register_comportamentale;
16
17 architecture Behavioral of shift_register_comportamentale is
18 signal x : std_logic_vector (N-1 downto 0);
19 begin
20     process(clk)
21     begin
22         if(clk'event and clk = '1') then
23             if(rst = '1') then
24                 x <= (others => '0');

```

```

25      elsif(enable = '1') then
26          if(load = '1') then
27              x <= in_parallel;
28          elsif(is_right = '0') then
29              x(N - 1 downto conv_integer(shifts)) <= x(N
30                  <- 1 - conv_integer(shifts) downto 0);
31              x(conv_integer(shifts) - 1 downto 0) <=
32                  <- x(N-1 downto N - conv_integer(shifts));
33              x(N - 1 downto N - conv_integer(shifts)) <=
34                  <- x(conv_integer(shifts) - 1 downto 0);
35          end if;
36      end if;
37  end process;
38
39  out_parallel <= x;
40
41 end Behavioral;

```

Listing 4.3: Shift register: approccio comportamentale

4.4 Simulazione

Dopo aver descritto il componente, abbiamo sviluppato un test bench per testare sia la versione comportamentale che quella strutturale in un ambiente simulato. Questo ci ha permesso di verificare il corretto funzionamento del componente in condizioni controllate. Il nostro approccio ha facilitato la valutazione delle prestazioni e l'identificazione di eventuali problemi, contribuendo così a garantire la robustezza del componente prima dell'integrazione nel sistema più ampio.

4.4.1 Approccio strutturale

In questo paragrafo presentiamo il test bench per testare lo shift register utilizzando un approccio strutturale (listato 4.4).

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity shift_register_strutturale_tb is
5 end shift_register_strutturale_tb;
6
7 architecture Behavioral of shift_register_strutturale_tb is
8 component shift_register_strutturale is
9     generic(
10         N : integer := 8
11     );
12     port(
13         in_parallel : in STD_LOGIC_VECTOR (N-1 downto 0);
14         clk : in STD_LOGIC;
15         rst : in STD_LOGIC;
16         load : in STD_LOGIC;
17         enable : in STD_LOGIC;
18         shifts : in STD_LOGIC_VECTOR (1 downto 0);
19         out_parallel : out STD_LOGIC_VECTOR (N-1 downto 0)
20     );
21 end component;
22
23 signal in_parallel : STD_LOGIC_VECTOR (7 downto 0);
24 signal clk : STD_LOGIC;
25 signal rst : STD_LOGIC;
26 signal load : STD_LOGIC;
27 signal enable : STD_LOGIC;
28 signal shifts : STD_LOGIC_VECTOR (1 downto 0);
29 signal out_parallel : STD_LOGIC_VECTOR (7 downto 0);
```

```

30
31 begin
32     uut : shift_register_strutturale generic map(
33         N => 8
34     ) port map (
35         in_parallello => in_parallello, clk => clk, rst => rst,
36         ↵ load => load, enable => enable, shifts => shifts,
37         ↵ out_parallello => out_parallello
38     );
39
40     clock_tb : process begin
41         clk <= '0';
42         wait for 10 ns;
43         clk <= '1';
44         wait for 10 ns;
45     end process;
46
47     inputs_tb : process begin
48         rst <= '0';
49         enable <= '1';
50         load <= '1';
51         in_parallello <= x"05";
52         wait for 15 ns;
53         load <= '0';
54         shifts <= "00";
55         wait for 50 ns;
56         shifts <= "01";
57         wait for 50 ns;
58         shifts <= "11";
59         wait for 50 ns;
60         shifts <= "10";
61         wait for 50 ns;
62         rst <= '1';
63         wait;
64     end process;

```

Listing 4.4: Testbench shift register con approccio strutturale

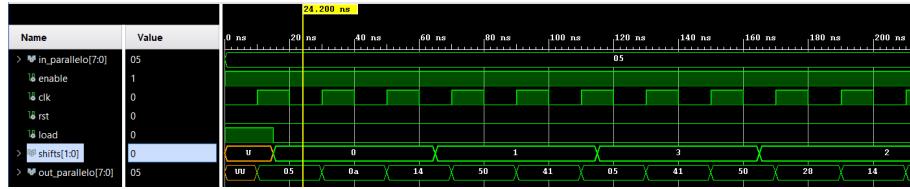


Figura 4.4: Waveform shift register strutturale

4.4.2 Approccio comportamentale

In questo paragrafo presentiamo il test bench per testare lo shift register utilizzando un approccio comportamentale (listato 4.5).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity shift_register_comportamentale_tb is
5 end shift_register_comportamentale_tb;
6
7 architecture Behavioral of shift_register_comportamentale_tb is
8     component shift_register_comportamentale is
9         generic(
10             N : integer := 16
11         );
12         port(
13             in_parallel : in STD_LOGIC_VECTOR (N-1 downto 0);
14             clk : in STD_LOGIC;
15             rst : in STD_LOGIC;
16             load : in STD_LOGIC;
17             enable : in STD_LOGIC;
18             shifts : in STD_LOGIC_VECTOR (3 downto 0);
19             is_right : in STD_LOGIC;
20             out_parallel : out STD_LOGIC_VECTOR (N-1 downto 0)
21         );
22     end component;
23
24     signal in_parallel : STD_LOGIC_VECTOR (15 downto 0);
25     signal clk : STD_LOGIC;
26     signal rst : STD_LOGIC;
27     signal load : STD_LOGIC;
28     signal enable : STD_LOGIC;
29     signal shifts : STD_LOGIC_VECTOR (3 downto 0);
30     signal is_right : STD_LOGIC;
31     signal out_parallel : STD_LOGIC_VECTOR (15 downto 0);

```

```

32
33 begin
34
35     uut : shift_register_comportamentale generic map(
36         N => 16
37     ) port map(
38         in_parallello => in_parallello, clk => clk, rst => rst,
39         ↵ load => load, enable => enable, shifts => shifts,
40         ↵ is_right => is_right, out_parallello =>
41         ↵ out_parallello
42     );
43
44
45     clock_tb : process begin
46         clk <= '0';
47         wait for 10 ns;
48         clk <= '1';
49         wait for 10 ns;
50     end process;
51
52
53     inputs_tb : process begin
54         rst <= '0';
55         enable <= '1';
56         shifts <= x"1";
57         is_right <= '0';
58         load <= '1';
59         in_parallello <= x"0001";
60         wait for 15 ns;
61         load <= '0';
62         wait for 20 ns;
63         shifts <= x"1";
64         wait for 20 ns;
65         shifts <= x"2";
66         wait for 20 ns;
67         is_right <= '1';
68         wait for 20 ns;
69         shifts <= x"2";
70         wait for 20 ns;
71         shifts <= x"3";
72         wait for 20 ns;
73         shifts <= x"3";
74         wait for 50 ns;
75         rst <= '1';
76         wait;
77     end process;

```

73

74 **end Behavioral;**

Listing 4.5: Testbench shift register con approccio comportamentale

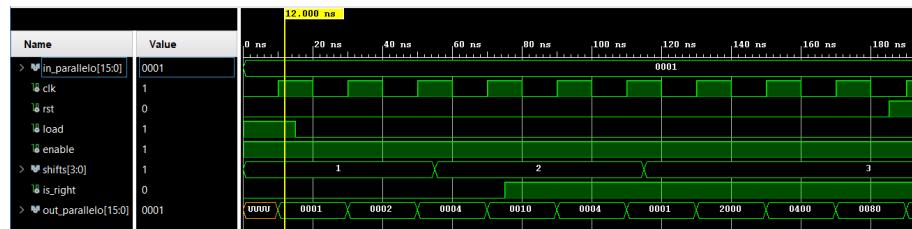


Figura 4.5: Waveform shift register comportamentale

Capitolo 5

Cronometro

5.1 Traccia

Esercizio 5.1 Progettare, implementare in VHDL e testare mediante simulazione un cronometro, in grado di scandire secondi, minuti e ore a partire da una base dei tempi prefissata (es. si consideri il clock a disposizione sulla board). Il progetto deve prevedere la possibilità di inizializzare il cronometro con un valore iniziale, sempre espresso in termini di ore, minuti e secondi, mediante un opportuno ingresso di set, e deve prevedere un ingresso di reset per azzerare il tempo. Il componente deve essere realizzato utilizzando un approccio strutturale, collegando opportunamente dei contatori secondo uno schema a scelta.

Esercizio 5.2 Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando i display a 7 segmenti per la visualizzazione dell'orario (o una combinazione di display e led nel caso in cui i display a disposizione siano in numero inferiore a quello necessario), gli switch per l'immissione dell'orario iniziale e due bottoni, uno per il set dell'orario e uno per il reset. Si utilizzi una codifica a scelta dello studente per la visualizzazione dell'orario sui display (esadecimale o decimale).

Esercizio 5.3 Estendere il componente sviluppato ai punti precedenti in modo che sia in grado di acquisire e memorizzare internamente fino ad un numero N di intertempi in corrispondenza di un ingresso di stop. Opzionalmente, il componente può prevedere una modalità di visualizzazione in cui, alla pressione di un bottone, vengano visualizzati sui display gli intertempi memorizzati (uno per ogni pressione).

5.2 Soluzione

Per il quinto esercizio si vuole realizzare un cronometro con diverse modalità di funzionamento, ovvero:

1. funzione di conteggio, che permetterà lo scorrere dei secondi, minuti e ore, come un classico cronometro;
2. funzione di orologio, può essere combinata con la funzione di conteggio in modo da far scorrere il tempo a partire da un certo orario;
3. funzione di visualizzazione degli intertempi memorizzati, permette all'utente di visualizzare i vari intertempi salvati.

La progettazione dell'esercizio completo è stata divisa in due parti principali: l'unità di controllo (UC) e l'unità operativa (UO). L'UC definisce il comportamento complessivo del sistema attraverso un'automa a stati finiti. D'altra parte, l'UO comprende tutti i componenti architetturali necessari per implementare concretamente le operazioni richieste, come memorie, multiplexer, contatori ecc. Questi componenti ricevono comandi specifici, noti come segnali di controllo, dall'UC.

L'unità di controllo è stata progettata con un approccio comportamentale, così come si farebbe per ogni macchina sequenziale modellata con un automa a stati finiti, i cui ingressi sono forniti dagli utenti. Invece le uscite, sono i segnali di controllo che saranno forniti come input all'unità operativa. Per la UO, invece abbiamo adottato un approccio strutturale, dove al suo interno sono stati definiti tutti i componenti architetturali essenziali per garantire il corretto funzionamento del sistema. Spiegheremo meglio l' UO e l' UC nei seguenti paragrafi, solo dopo aver spiegato nel dettaglio prima tutti i componenti che le compongono.

Per svolgere il primo punto di questo esercizio abbiamo bisogno di un componente fondamentale, ovvero il contatore modulo N (figura 5.1). Tale componente è sensibile al fronte di salita del clock e ad ogni colpo incrementa una variabile interna che viene poi restituita in output. Appena tale variabile raggiunge il valore $N-1$, si azzera, permettendo l'inizio di un nuovo ciclo di conteggio e restituisce in uscita, su di un segnale apposito, un "impulso" che sta ad indicare che ha contato N valori.

Questo meccanismo è utile poiché permette, come nel nostro progetto, di costruire un sistema di contatori in cascata, dove il precedente abilita il successivo a contare non appena esso ha terminato il suo conteggio. In particolare avremo tre contatori posti in cascata dove il primo calcola i secondi da 0 a 59, il secondo conta i minuti sempre da 0 a 59 e il terzo conta le ore da 0 a 23, ovvero il nostro cronometro (figura 5.2). Ciascun contatore è stato opportunamente realizzato con un approccio comportamentale. La base dei tempi di riferimento è quello della board, ovvero di 100 MHz, dunque

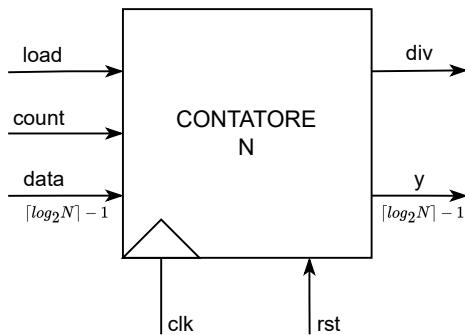


Figura 5.1: Schematico rappresentativo del contatore modulo N

occorre un divisore di frequenza con cui si possa passare a 1 Hz per avere un opportuno conteggio dei secondi ($1/T$ con $T = 1$ s). Tale base dei tempi è posta in ingresso al contatore dei secondi in modo tale che conti i secondi ogni 1 Hz (ovvero ogni secondo) e raggiunto 59 abilita un segnale che attiva l'incremento del contatore dei minuti che a sua volta raggiungerà anche esso un conteggio di 59 ed attiverà l'incremento del contatore delle ore.

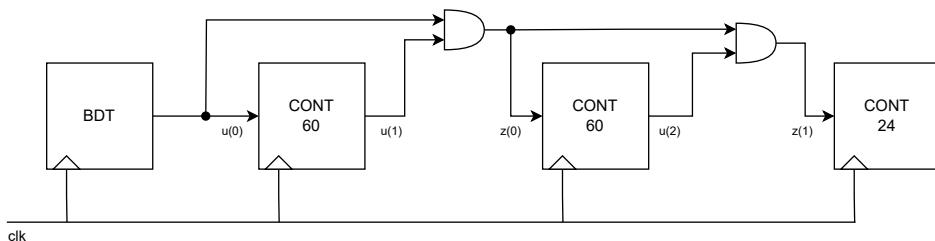


Figura 5.2: Schematico rappresentativo del cronometro

L'esercizio 5.2 prevede di sintetizzare su board il componente precedentemente descritto utilizzando un display per la visualizzazione decimale dell'orario, gli switch per l'immissione dell'orario iniziale e due button uno per il set dell'orario iniziale e uno per il reset della macchina. Per la funzionalità di set e reset del cronometro basta mappare opportunamente i due segnali con due button debouncer, utili per risolvere il problema delle oscillazioni dovute alla pressione dei button presenti sulla scheda. Per poter visualizzare correttamente l'orario sul display, abbiamo bisogno di un convertitore BCD (figura 5.3) che ha funzione di trasformare il valore di secondi, minuti e ore (che sono in binario) dapprima in decimale e successivamente trasformare le due cifre (unità e decine) separatamente in binario. Quindi in ingresso avremo un vettore di 6 bit, che rappresenta il numero da convertire, mentre in uscita un vettore di 8 bit, i cui primi 4 identificano le unità e i secondi 4 bit le decine. In questo modo riusciamo a visualizzare correttamente le cifre decimali sul display. Per la costruzione di tale componente abbiamo

sfruttato il fatto che per ottenere la cifra delle unità di un intero basta fare l'operazione di modulo per 10 e che per ottenere la cifra delle decine basta dividere il numero intero per 10 ed eseguire di nuovo l'operazione di modulo.

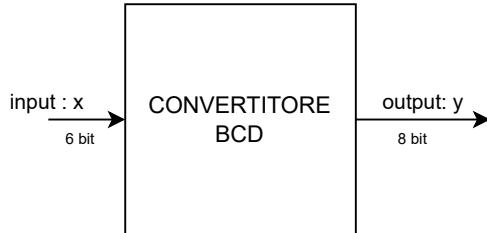


Figura 5.3: Schematico rappresentativo del convertitore BCD

Per l'ultimo punto del progetto, ovvero quello che riguarda la memorizzazione degli intertempi, abbiamo progettato una memoria (figura 5.4) avente 8 locazioni da 24 bit dedicate al salvataggio degli intertempi. Per accedere a questa memoria, è necessario un indirizzo ottenuto come output da un multiplexer 2:1. Il multiplexer presenta due ingressi, rappresentanti rispettivamente il prossimo indirizzo in cui salvare l'intertempo i-esimo e la prossima locazione da cui leggere l'intertempo memorizzato. In modo intuitivo, il primo ingresso sarà selezionato quando si desidera scrivere nella memoria, mentre il secondo sarà selezionato quando si intende leggere da essa. Per tale memoria indicheremo con *width* la grandezza di una singola cella di memoria (nel nostro caso 24 bit) e *nbits_addr* numero di bit di indirizzamento. Oltre al segnale di ingresso di reset (*rst*) e al segnale di sincronizzazione (*clk*), la macchina ha in ingresso un segnale di abilitazione *en* e un segnale *r_w* per attivare la lettura o la scrittura in memoria. Se il segnale *r_w* è '0' il componente è abilitato a scrivere se è '1' il componente è abilitato a leggere (in entrambi i casi ovviamente il segnale *en* deve essere alto, altrimenti non si è abilitati né a leggere né a scrivere). Per accedere ad una locazione di memoria abbiamo in ingresso un segnale *addr*, di *nbits_addr* bit, invece se volessimo accedere in modalità scrittura, oltre all'indirizzo, abbiamo un ingresso *data_in*, di *width* bit, che indica il valore da salvare in memoria. Infine, se la memoria è in modalità lettura, abbiamo un'uscita output di *width* elementi, che indica il valore letto. Le funzionalità di salvataggio e visualizzazione degli intertempi sono ottenute dalla pressione di due pulsanti.

5.2.1 Unità operativa

L'Unità Operativa (UO) del nostro esercizio presenta un'interfaccia notevolmente complessa, con la gestione di numerosi ingressi di controllo. Di seguito presenteremo i componenti di cui esso è composto:

- un cronometro

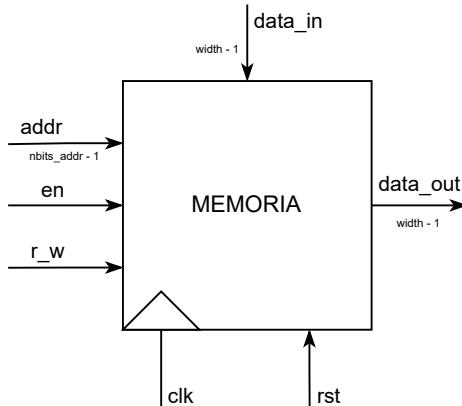


Figura 5.4: Schematico rappresentativo della memoria

- un convertitore BCD per i secondi,
- un convertitore BCD per i minuti,
- un convertitore BCD per le ore,
- una memoria con 8 locazioni da 24 bit,
- due contatori modulo 8, uno che punta al prossimo intertempo in memoria da visualizzare e altro che punta alla prossima locazione sui cui memorizzare un intertempo,
- due multiplexer 2:1, uno per la selezione dell'uscita dei due contatori, da cui dipende l'accesso in memoria, e l'altro per la visualizzazione degli intertempi o la visualizzazione dello stato di avanzamento del cronometro.

La complessità della struttura è attribuibile alle diverse modalità operative del nostro sistema (diciamo che una è in modalità visualizza cronometro e l'altra è visualizza intertempi salvati). In particolare, oltre ai segnali di reset e di clock e ai segnali di ingresso forniti dall'utente (*sec_in*, *min_in*, *ore_in*), abbiamo una serie di ingressi di controllo i quali sono elencati di seguito:

- set, permette di iniziare il conteggio del cronometro a partire da degli ingressi forniti dall'utente,
- en, permette di abilitare la lettura o la scrittura in memoria,
- r_w, se $r_w = 1$ e $en = 1$, allora si è abilitati a leggere in memoria, al contrario se $r_w = 0$ e $en = 1$ si è abilitati a scrivere in memoria,

- `incr_v`, permette di incrementare il contatore dedito alla visualizzazione degli intertempi
- `incr_s`, permette di incrementare il contatore dedito alla memorizzazione degli intertempi

Invece in uscita abbiamo un unico segnale da 24 bit, `uscita_visore`, ovvero l'orario da visualizzare sul display. In figura 5.5 è rappresentato lo schematico complessivo dell' unità operativa del nostro sistema.

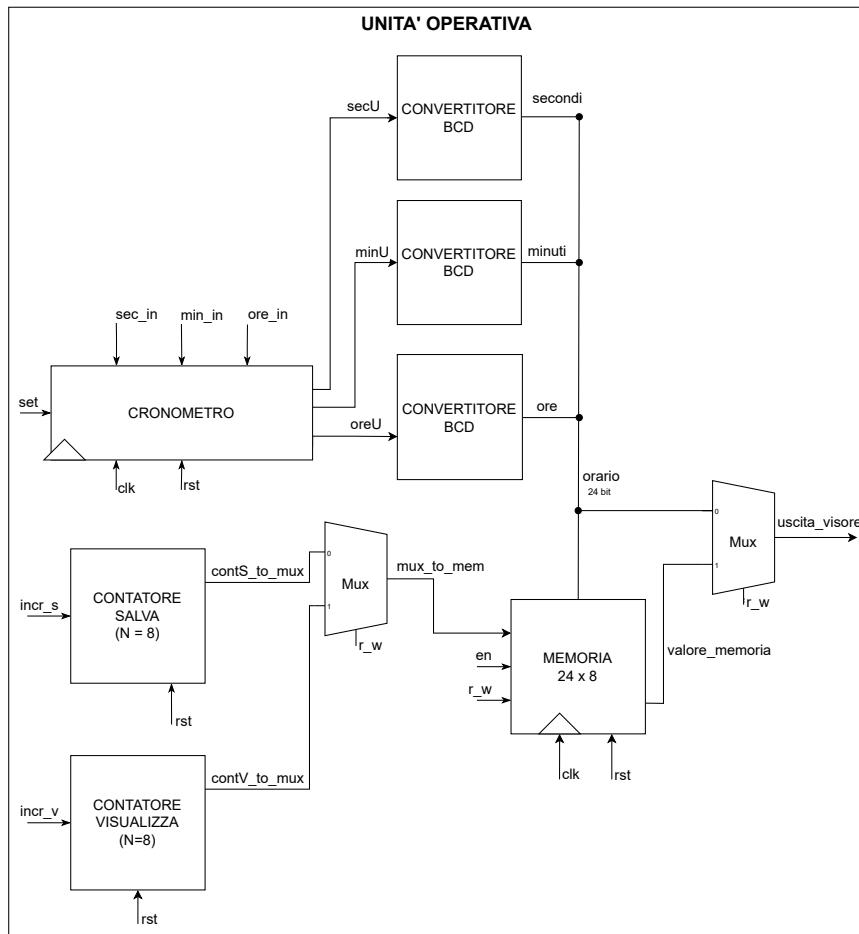


Figura 5.5: Schematico unità operativa del sistema

5.2.2 Unità di controllo

L'unità di controllo del nostro progetto, oltre ai segnali di reset e di clock, ha tutta una serie di segnali i quali sono:

- segnali di ingresso forniti dall'utente

- il segnale *mode*, permette di controllare la modalità di funzionamento, attraverso un switch, del nostro cronometro; ovvero se *mode* = '1' funzionerà in modalità visualizzazione degli intertempi salvati in memoria, altrimenti in modalità visualizzazione dello stato di avanzamento del cronometro, con possibilità di salvataggio degli intertempi,
 - il segnale *salva_b*, permette di memorizzare un intertempo ad un certo istante,
 - il segnale *visualizza_b*, permette di incrementare il contatore che indirizza la memoria, ovvero permette di visualizzare sul display l'intertempo successivo,
 - il segnale *set_b*, permette di pre-caricare un determinato orario sul cronometro,
- segnali di controllo (o di uscita), sono identici ai precedenti definiti per la UO, con l'unica variazione che adesso sono segnali di uscita anziché di ingresso.

I possibili stati della UC (Unità di Controllo) sono: *RESET*, *CRONOMETRO*, *SET*, *SALVA*, *VISUALIZZA*, *INC_S* e *INC_V*. Come evidenziato nella figura 5.6, lo stato centrale è *CRONOMETRO*, in cui tutti i segnali di controllo sono impostati a '0'. Successivamente, in base agli input attivati dall'utente, la macchina transita in uno degli stati possibili, ognuno dei quali descrive in modo intuitivo diverse modalità operative del nostro cronometro. È importante notare che la fase di memorizzazione di un intertempo è divisa in due stati elementari: *SALVA*, in cui il segnale di abilitazione alla scrittura nella memoria è impostato a '1', e *INC_S*, in cui viene aggiornato di una posizione l'indirizzo per il prossimo salvataggio. Dopo aver memorizzato otto intertempi, il successivo salvataggio non viene bloccato ma bensì il prossimo indirizzo punterà alla prima locazione della memoria, sovrascrivendo così le locazioni di memoria precedentemente occupate. La particolarità di questo automa è che se la modalità selezionata dall'utente è zero, allora la macchina evolverà tra gli stati *RESET*, *CRONOMETRO*, *SET*, *SALVA* e *INC_S* invece se è impostata ad uno la macchina evolverà tra gli stati *RESET*, *VISUALIZZA* e *INC_V*.

5.2.3 Display a sette segmenti

L'esercizio richiede esplicitamente, nel secondo punto, l'utilizzo del display a sette segmenti presente sulla board di sviluppo, dunque vale la pena illustrarne velocemente il funzionamento. La board Nexys-A7-100T in dotazione, possiede un display a sette segmenti composto da otto cifre. Ogni cifra può essere vista come un singolo display a sette segmenti. Ogni segmento del display possiede un anodo ed un catodo, che pilotati opportunamente consentono

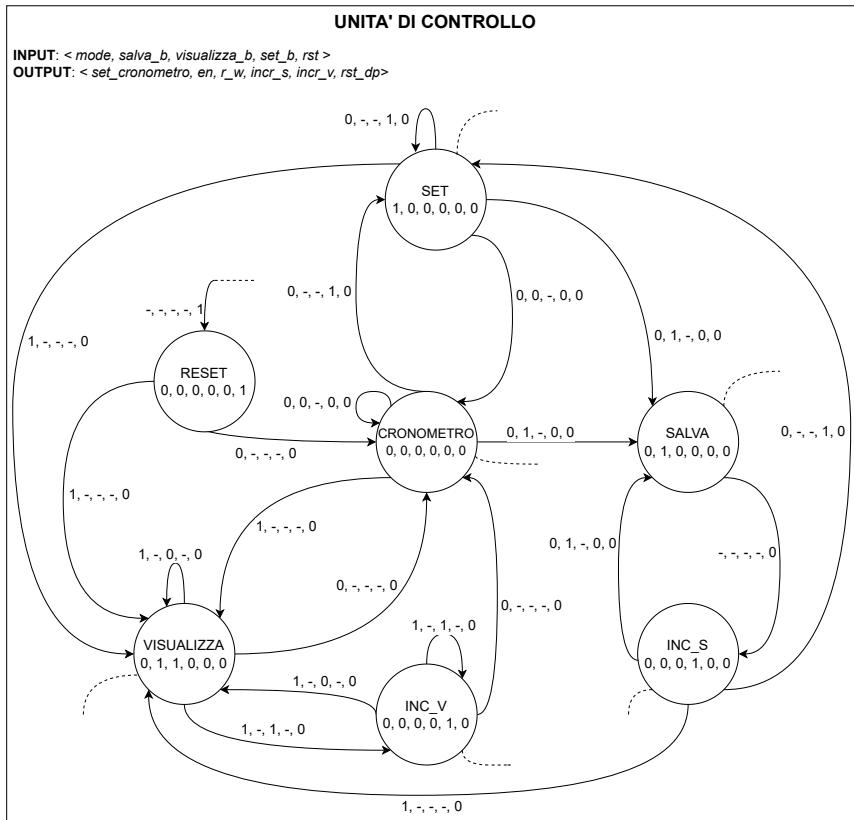


Figura 5.6: Schematico automa dell'unità di controllo del sistema

l'accensione dello stesso. In particolare, abbassando la tensione sul catodo, ed alzando quella sull'anodo si ha l'accensione del segmento. Tuttavia, nella board di sviluppo in dotazione, il controllo dell'anodo è invertito, dunque occorre fornire un valore logico basso al pin dell'anodo per poter accendere il segmento ad esso relativo. Ogni cifra del display ha otto catodi differenti (CA, CB, ..., DP), ovvero uno per ogni segmento più uno per il punto a destra della cifra, ma un solo anodo in comune a tutti i segmenti. Invece, guardando al display complessivo, ogni cifra ha il proprio anodo, mentre tutte condividono gli stessi otto catodi. Tale configurazione è visibile in figura 5.7.

Tale struttura del display impedisce l'accensione simultanea di tutte le cifre, obbligandone l'accensione alternata di una sola di esse alla volta. Affinché l'utilizzatore del sistema abbia l'illusione che tutte le cifre si accendano contemporaneamente è necessario accenderle tutte in successione, in un intervallo di tempo talmente breve che l'occhio umano non riesca a rendersi conto di ciò. L'intervallo di tempo in cui avviene il refresh di tutte le otto cifre del display è detto *refresh period*, ed è solitamente scelto tra 1 ms e 16 ms; meno di 1 ms si avrebbero problemi con i limiti fisici dei circuiti del display, mentre oltre i 16 ms l'occhio umano inizierebbe a notare uno sfarfallio sulle cifre. In

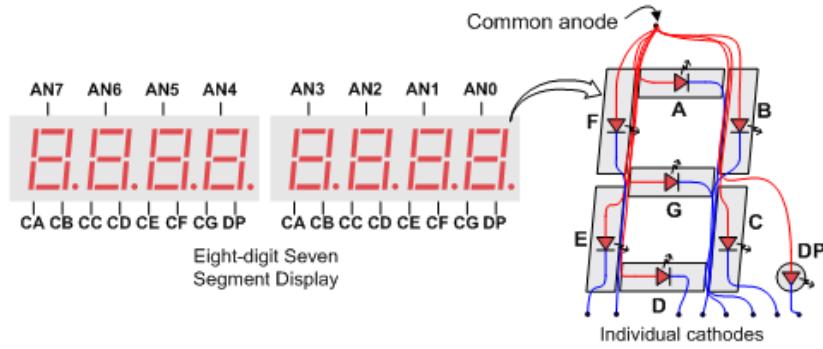


Figura 5.7: Display a sette segmenti

questo elaborato, il refresh period viene fissato proprio a 16 ms. In pratica, ogni cifra non viene tenuta accesa costantemente, ma viene aggiornata ad una frequenza pari al refresh period. Grazie al fenomeno della persistenza dei fosfori e ai limiti dell'occhio umano, tale meccanismo non viene notato dall'utente. Dividendo il refresh period per otto si ottiene l'intervallo di tempo per il quale viene alimentata ogni singola cifra, ovvero il *digit period*.

$$\text{refresh_period} = 16 \text{ ms} \implies \text{digit_period} = 2 \text{ ms}$$

Occorre dunque un segnale di temporizzazione con periodo 2 ms (ovvero con una frequenza di 500 Hz) per scandire il refresh di ogni singola cifra.

Chiarito il funzionamento del display presente sulla board, occorre progettare un componente che sia in grado di ricevere in ingresso la codifica binaria del valore da visualizzare, e tradurla in un'opportuna configurazione di anodi e catodi che tenga aggiornato il display nel tempo, con un refresh period complessivo di 16 ms. Tale componente sarà chiamato, da qui in poi, *display manager*, ed è raffigurato in figura 5.8.

Supponendo di voler rappresentare su ogni cifra del display il valore di una cifra esadecimale (0, 1, ..., F), occorrono 4 bit per codificare il valore di una singola cifra. Moltiplicando tale quantità per il numero totale di cifre del display, si rendono necessari 32 bit per rappresentare il valore da visualizzare sull'intero schermo. Oltre alla codifica delle cifre da rappresentare, è opportuno fornire anche un segnale per ogni *dot* che è possibile accendere sul display, dunque si hanno altri 8 segnali in ingresso. In ingresso al componente, sono previsti altri otto segnali (DIGITS ENABLE) che indicano quali cifre illuminare, e quali no. Per poter passare dal clock della board, che solitamente ha una frequenza molto elevata (es. 100 MHz), ad un segnale con una frequenza relativamente bassa come 500 Hz, è utilizzato un componente detto CLOCK FILTER, che altro non è che un contatore che restituisce in uscita il

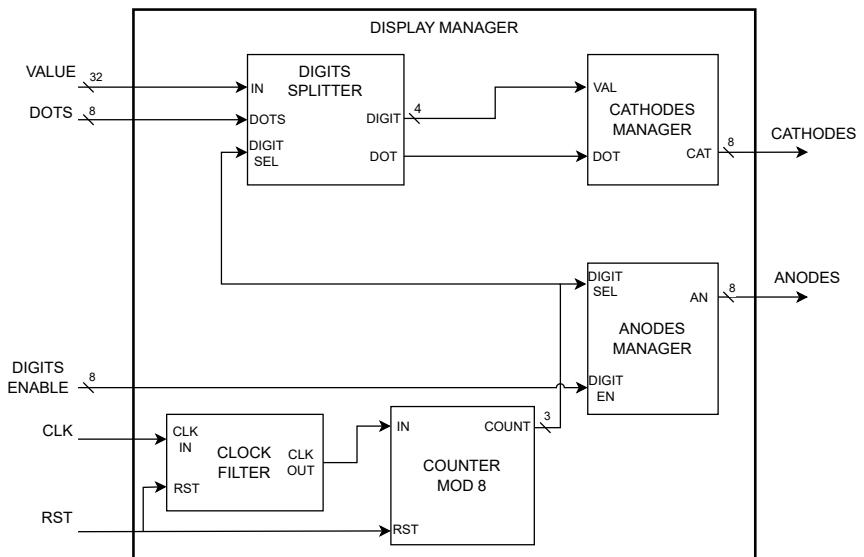


Figura 5.8: Display manager

segnale alla frequenza desiderata. Il segnale di clock "filtrato" viene dato in ingresso ad un contatore modulo 8, che ogni 2 ms scandisce:

- quale cifra illuminare, attraverso l'ANODES MANAGER,
 - quale porzione da 4 bit, dei 32 in ingresso, e quale segnale DOTS, utilizzare per pilotare i catodi attraverso il CATHODES MANAGER.

5.3 Codice

5.3.1 Contatore

Come detto in precedenza, il punto iniziale per la costruzione strutturale di un cronometro consiste nella realizzazione di due contatori modulo 60 e uno modulo 24. Per semplicità, sarà proposto al listato 5.1 solamente il codice del contatore modulo 60. Il componente è stato progettato secondo una architettura behavioral e configurato in modo tale che fosse sensibile al fronte di salita del segnale di clock.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.std_logic_unsigned.all;
4
5 entity contatore_mod60 is
6     Port ( clk : in STD_LOGIC;
7             rst : in STD_LOGIC;
8             load : in STD_LOGIC;
9             count : in STD_LOGIC;
10            data : in STD_LOGIC_VECTOR(5 downto 0);
11            div : out STD_LOGIC;
12            y : out STD_LOGIC_VECTOR(5 downto 0));
13 end contatore_mod60;
14
15 architecture Behavioral of contatore_mod60 is
16 signal TY: std_logic_vector(5 downto 0);
17
18 begin
19
20     Y <= TY;
21
22     process(clk,rst) begin
23
24         if(clk'event and clk='1') then
25             if(rst = '1') then
26                 TY <= (others => '0');
27             elsif(load = '1') then
28                 TY <= data;
29             elsif(count = '1') then
30                 if(TY = "111011") then
31                     TY <= (others => '0');
32                 else
33                     TY <= TY + "000001";
34             end if;
35         end if;
36     end process;
37 end Behavioral;
```

```

34         end if;
35     end if;
36 end if;
37 end process;
38
39 process(TY) begin
40     if(TY = "111011") then
41         div <= '1';
42     else
43         div <= '0';
44     end if;
45 end process;
46
47
48 end Behavioral;

```

Listing 5.1: Contatore modulo 60

5.3.2 Cronometro

Il cronometro è stato progettato attraverso un approccio strutturale, ovvero costituito da più contatori in cascata i quali scandiscono i secondi, minuti e ore. Al listato 5.2 è presente il codice del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4 use IEEE.math_real.all;
5 use ieee.std_logic_unsigned.all;
6
7 entity cronometro is
8     generic ( clk_B_factor : positive);
9     port(
10         clk, rst, set : in std_logic;
11         sec_in : in std_logic_vector(5 downto 0);
12         min_in : in std_logic_vector(5 downto 0);
13         ore_in : in std_logic_vector(4 downto 0);
14
15         sec_out : out std_logic_vector(5 downto 0);
16         min_out : out std_logic_vector(5 downto 0);
17         ore_out : out std_logic_vector(4 downto 0)
18     );
19 end cronometro;

```

```

20
21  architecture Structural of cronometro is
22
23      component base_dei_tempi is
24          generic ( clk_B_factor : positive);
25          Port ( clk : in STD_LOGIC;
26                  rst : in STD_LOGIC;
27                  clk_B : out STD_LOGIC);
28      end component;
29
30      component contatore_mod60 is
31          port(clk : in STD_LOGIC;
32                  rst : in STD_LOGIC;
33                  load : in STD_LOGIC;
34                  count : in STD_LOGIC;
35                  data : in STD_LOGIC_VECTOR(5 downto 0);
36                  div : out STD_LOGIC;
37                  y : out STD_LOGIC_VECTOR(5 downto 0));
38      end component;
39
40      component contatore_mod24 is
41          port(clk : in STD_LOGIC;
42                  rst : in STD_LOGIC;
43                  load : in STD_LOGIC;
44                  count : in STD_LOGIC;
45                  data : in STD_LOGIC_VECTOR(4 downto 0);
46                  div : out STD_LOGIC;
47                  y : out STD_LOGIC_VECTOR(4 downto 0));
48      end component;
49
50      signal u : std_logic_vector(2 downto 0);
51      signal z : std_logic_vector(1 downto 0);
52
53  begin
54
55      z(0) <= u(0) and u(1);
56      z(1) <= u(0) and u(1) and u(2);
57
58      bdt: base_dei_tempi generic map(clk_B_factor =>
59                                     clk_B_factor)
60                                     port map(
61                                     clk => clk, rst => rst, clk_b => u(0)
62 );

```

```

63     sec: contatore_mod60 port map(
64         clk => clk, rst => rst, load => set, count => u(0),
65         ↵  data => sec_in, div => u(1), y => sec_out
66     );
67
68     min: contatore_mod60 port map(
69         clk => clk, rst => rst, load => set, count => z(0),
70         ↵  data => min_in, div => u(2), y => min_out
71     );
72
73     ore: contatore_mod24 port map(
74         clk => clk, rst => rst, load => set, count => z(1),
75         ↵  data => ore_in, y => ore_out
76     );
77
78 end Structural;

```

Listing 5.2: Cronometro

5.3.3 Convertitore BCD

Il convertitore BCD, definito a livello comportamentale, dichiara due variabili, *unita* e *decine*, all'interno del process, le quali rispettivamente conterranno, grazie all'operatore di modulo, le unità e le decine del segnale d'ingresso *x*, composto da 6 bit. L'uscita *y* è rappresentata dai primi 4 bit come l'unità del valore di *x*, invece i restanti 4 bit rappresentano le decine del valore di *x*.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.std_logic_unsigned.all;
4 use IEEE.NUMERIC_STD.ALL;
5
6 entity convertitore_BCD is
7     Port ( x : in STD_LOGIC_VECTOR(5 downto 0);
8             y : out STD_LOGIC_VECTOR(7 downto 0));
9 end convertitore_BCD;
10
11 architecture Behavioral of convertitore_BCD is
12
13 begin
14     process(x)
15         variable unita : integer;
16         variable decine : integer;

```

```

17      begin
18          unita := conv_integer(x) mod 10;
19          decine := (conv_integer(x)/10) mod 10;
20
21          y(3 downto 0) <=
22              → std_logic_vector(to_unsigned(unita,4));
23          y(7 downto 4) <=
24              → std_logic_vector(to_unsigned(decine,4));
25      end process;
26
27  end Behavioral;

```

Listing 5.3: Convertitore BCD

5.3.4 Memoria

La memoria è stata realizzata adottando un approccio comportamentale. I generic *nbits_addr* e *width*, rappresentano rispettivamente il numero di bit di indirizzamento e la dimensione di una word di memoria. L'operatività della memoria è sincronizzata con il segnale di clock (*clk*), evolvendo sul fronte di salita, ed è abilitata da un segnale *en*, il quale se è alto può leggere o scrivere dalla memoria a seconda di se il segnale *r_w* è alto o basso (*r_w* = 1 legge, *r_w* = 0 scrive). Di seguito è riportata l'implementazione.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity memory is
6     generic(
7         nbits_addr : positive := 3;      -- numero di bit di
8                                     → indirizzamento
8         width : positive := 32 );       -- dimensione di una word
9                                     → di memoria
9     Port ( addr : in STD_LOGIC_VECTOR (nbits_addr - 1 downto
10           → 0);
11         en : in STD_LOGIC;
12         clk : in STD_LOGIC;
13         rst : in STD_LOGIC;
14         data_in : in STD_LOGIC_VECTOR (width - 1 downto 0);
15         r_w : in STD_LOGIC;           -- r_w = 1 => read ; r_w
16                                     → = 0 => write

```

```

15      data_out : out STD_LOGIC_VECTOR (width - 1 downto
16          < 0));
17  end memory;
18
18 architecture Behavioral of memory is
19     constant depth : positive := 2**nbits_addr;
20     subtype word is std_logic_vector (width - 1 downto 0);
21
22     type mem_type is array(0 to depth - 1) of word;
23     signal mem : mem_type;
24
25 begin
26     process(clk)
27     begin
28         if clk = '1' and clk'event then
29             if rst = '1' then
30                 for i in 0 to depth-1 loop
31                     mem(i) <= (others => '0');
32                 end loop;
33             else
34                 if en = '1' then
35                     if r_w = '1' then
36                         data_out <=
37                             mem(to_integer(unsigned(addr)));
38                     else
39                         mem(to_integer(unsigned(addr))) <=
40                             data_in;
41                     end if;
42                 end if;
43             end if;
44     end process;
45 end Behavioral;

```

Listing 5.4: Memoria

5.3.5 Unità operativa

L’interfaccia dell’unità operativa del nostro progetto è governata da molteplici ingressi di controllo. All’interno dell’architecture, viene creato il segnale *orario* concatenando le uscite dei tre convertitori BCD (*secondi, minuti, ore*). Tale segnale riflette il valore corrente del cronometro, offrendo la possibilità di

conservarlo come intertempo in memoria o di visualizzarlo sul display a 7 segmenti.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_operativa is
5     generic ( clk_B_factor : positive);
6     Port (
7         rst : in STD_LOGIC;
8         clk : in STD_LOGIC;
9         set : in STD_LOGIC;
10        sec_in : in STD_LOGIC_VECTOR(5 downto 0);
11        min_in : in STD_LOGIC_VECTOR(5 downto 0);
12        ore_in : in STD_LOGIC_VECTOR(4 downto 0);
13        en : in STD_LOGIC;
14        r_w : in STD_LOGIC;
15        incr_v : in STD_LOGIC;
16        incr_s : in STD_LOGIC;
17        uscita_visore : out STD_LOGIC_VECTOR(23 downto
18                                     ← 0));--da dare in ingresso al display
18 end unita_operativa;
19
20 architecture Structural of unita_operativa is
21
22 signal secU : std_logic_vector(5 downto 0);
23 signal minU : std_logic_vector(5 downto 0);
24 signal oreU : std_logic_vector(5 downto 0);
25
26 signal secondi : std_logic_vector(7 downto 0); -- secondi,
27   ← primi 4 bit unita secondi 4 bit decine
27 signal minuti : std_logic_vector(7 downto 0); -- minuti, primi
28   ← 4 bit unita secondi 4 bit decine
28 signal ore : std_logic_vector(7 downto 0); -- ore, primi 4 bit
29   ← unita secondi 4 bit decine
30
30 signal mux_to_mem : std_logic_vector(2 downto 0); --possono
31   ← essere memorizzati massimo 8 intertempi
31 signal orario : std_logic_vector(23 downto 0);
32
33 signal contS_to_mux : std_logic_vector(2 downto 0); --con tre
34   ← bit possono essere memorizzati massimo 8 intertempi
34 signal contV_to_mux : std_logic_vector(2 downto 0);
```

```

35
36 signal valore_memoria : std_logic_vector(23 downto 0);--4 bit
   → unita secondi, 4 bit decine secondi, 4 bit unita minuti ...
37
38 begin
39     oreU(5)<= '0';
40     orario <= ore & minuti & secondi;
41
42     cronometro: entity work.cronometro generic map(clk_B_factor
   → => clk_B_factor)
43         port map(
44             clk => clk, rst => rst,
45             set => set, sec_in => sec_in, min_in => min_in,
   → ore_in => ore_in,
46             sec_out => secU, min_out => minU, ore_out =>
   → oreU(4 downto 0));
47     convertitore_s: entity work.convertitore_BCD port map(
48         x => secU, y => secondi);
49
50     convertitore_m: entity work.convertitore_BCD port map(
51         x => minU, y => minuti);
52
53     convertitore_o: entity work.convertitore_BCD port map(
54         x => oreU, y => ore);
55
56
57     memoria: entity work.memory generic map(nbites_addr => 3,
   → width => 24)
58         port map(
59             addr => mux_to_mem, en => en, clk => clk, rst =>
   → rst, data_in => orario, r_w => r_w, data_out =>
   → valore_memoria
60         );
61     contatore_salva: entity work.counter generic map(bits => 3,
   → divider => 8)
62         port map(x => incr_s, rst => rst, y =>
   → contS_to_mux);
63
64     contatore_visualizza: entity work.counter generic map(bits
   → => 3, divider => 8)
65         port map(x => incr_v, rst => rst, y =>
   → contV_to_mux);
66
67     mux_adress: entity work.mux2to1 generic map(N => 3)

```

```

68      port map(
69          x1 => contS_to_mux, x2 => contV_to_mux, s =>
70          r_w, y => mux_to_mem);
71
72      mux_display: entity work.mux2to1 generic map(N => 24)
73          port map(
74              x1 => orario, x2 => valore_memoria, s => r_w, y
75                  => uscita_visore);
76
77
78  end Structural;

```

Listing 5.5: Unità operativa del sistema cronometro

5.3.6 Unità di controllo

L’unità di controllo è stata modellata utilizzando un approccio comportamentale, seguendo la stessa logica degli altri automi a stati finiti. La peculiarità da notare è che mentre l’unità operativa (UO) opera sul fronte di salita del clock, l’unità di controllo (UC) funziona sul fronte di discesa. Questa scelta è stata fatta intenzionalmente per garantire che le due parti non evolvano simultaneamente sullo stesso fronte del clock, evitando potenziali inconsistenze e malfunzionamenti. Il corpo dell’architecture è composto da due processi: uno sensibile al clock, che descrive l’evoluzione dello stato corrente (*sc*); l’altro, sensibile al segnale *sc* e agli ingressi dell’unità di controllo, descrive le variazioni nei segnali di controllo durante la transizione da uno stato all’altro.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_controllo is
5     Port ( mode : in STD_LOGIC; --switch
6             salva_b : in STD_LOGIC; --salva -> 0 , visualizza ->
7                 1, button
8             visualizza_b : in STD_LOGIC; --button per la
9                 visualizzazione
10            set_b : in STD_LOGIC;
11            rst : in STD_LOGIC; --button
12            clk : in STD_LOGIC;
13            set_cronometro : out STD_LOGIC;
14            en : out STD_LOGIC;
15            r_w : out STD_LOGIC;
16            incr_s : out STD_LOGIC;
17            incr_v : out STD_LOGIC;

```

```

16         rst_dp : out STD_LOGIC);
17 end unita_controllo;
18
19 architecture Behavioral of unita_controllo is
20 type stato is (reset,cronometro,set,salva,visualizza,inc_s,
21    ↳ inc_v);
22 signal sc : stato := reset; --stato corrente
23 signal next_state : stato;
24 begin
25     process(clk)
26     begin
27         if(falling_edge(clk)) then
28             if(rst = '1') then
29                 sc <= reset;
30             else sc <= next_state;
31             end if;
32         end if;
33     end process;
34
35     process(sc, mode, set_b, salva_b, visualizza_b, rst)
36     begin
37         case sc is
38             when reset =>
39                 set_cronometro <= '0';
40                 en <= '0';
41                 r_w <= '0';
42                 incr_s <= '0';
43                 incr_v <= '0';
44                 rst_dp <= '1';
45
46                 if(mode = '1') then
47                     next_state <=
48                         ↳ visualizza;
49                 else next_state <=
50                         ↳ cronometro;
51             end if;
52
53             when cronometro =>
54                 set_cronometro <= '0';
55                 en <= '0';
56                 r_w <= '0';

```

```

57
58          if(mode = '1') then
59              ↪ next_state <=
60                  ↪ visualizza;
61          elsif(set_b = '1') then
62              ↪ next_state <= set;
63                  ↪ elsif (salva_b = '1')
64                      ↪ then next_state
65                          ↪ <= salva;
66                      ↪ else next_state <=
67                          ↪ sc;
68          end if;
69
70
71
72      when set =>
73          set_cronometro <= '1';
74          en <= '0';
75          r_w <= '0';
76          incr_s <= '0';
77          incr_v <= '0';
78          rst_dp <= '0';
79
80          if(mode = '1') then
81              ↪ next_state <=
82                  ↪ visualizza;
83          elsif(set_b = '1') then
84              ↪ next_state <= sc;
85                  ↪ elsif(salva_b = '1')
86                      ↪ then next_state <=
87                          ↪ salva;
88                      ↪ else next_state <=
89                          ↪ cronometro;
90          end if;
91
92
93
94      when salva =>
95          set_cronometro <= '0';
96          en <= '1';
97          r_w <= '0';
98          incr_s <= '0';
99          incr_v <= '0';
100         rst_dp <= '0';
101
102         next_state <= inc_s;
103
104
105      when inc_s =>

```

```

89           set_cronometro <= '0';
90           en <= '0';
91           r_w <= '0';
92           incr_s <= '1';
93           incr_v <= '0';
94           rst_dp <= '0';
95
96           if(mode = '1') then
97               ↳ next_state <=
98               ↳ visualizza;
99           elsif(set_b = '1') then
100              ↳ next_state <= set;
101              elsif(salva_b = '1')
102                  ↳ then next_state <=
103                  ↳ salva;
104              else next_state <=
105                  ↳ cronometro;
106          end if;
107
108      when visualizza =>
109          set_cronometro <= '0';
110          en <= '1';
111          r_w <= '1';
112          incr_s <= '0';
113          incr_v <= '0';
114          rst_dp <= '0';
115
116          if(mode = '0') then
117              ↳ next_state <=
118              ↳ cronometro;
119          elsif(visualizza_b =
120              ↳ '1') then
121              ↳ next_state <=
122              ↳ inc_v;
123          else next_state <=
124              ↳ sc;
125      end if;
126
127      when inc_v =>
128          set_cronometro <= '0';
129          en <= '0';
130          r_w <= '0';
131          incr_s <= '0';
132          incr_v <= '1';

```

```

121                         rst_dp <= '0';
122
123                     if(mode = '0') then
124                         next_state <=
125                         cronometro;
126                     elsif(visualizza_b = '1')
127                         then next_state <=
128                         sc;
129                     else next_state <=
130                         visualizza;
131                     end if;
132
133                 end case;
134             end process;
135
136         end Behavioral;

```

Listing 5.6: Unità di controllo del sistema cronometro

5.3.7 Sistema complessivo

Il codice del sistema complessivo (listato 5.7) è stato definito attraverso un approccio strutturale, collegando opportunamente l'unità di controllo e l'unità operativa. Si può notare che in questo modo il sistema cronometro ha un'interfaccia più semplice e comprensibile, ovvero in ingresso avremo solo il segnale di sincronizzazione clk e gli input utente ed invece in uscita solo il segnale da dare in ingresso al display. Da notare che attraverso il generic *clk_B_factor* riusciamo a controllare il ciclo di conteggio della base dei tempi del cronometro.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cronometro_sys is
5     Generic ( clk_B_factor : positive);
6     Port(
7         clk : in STD_LOGIC;
8         rst : in STD_LOGIC;
9         salva_b : in STD_LOGIC;
10        visualizza_b : in STD_LOGIC;
11        mode : in STD_LOGIC;
12        set_b : in STD_LOGIC;
13        sec_in : in STD_LOGIC_VECTOR(5 downto 0);

```

```

14      min_in : in STD_LOGIC_VECTOR(5 downto 0);
15      ore_in : in STD_LOGIC_VECTOR(4 downto 0);
16      ingresso_visore : out STD_LOGIC_VECTOR(31 downto
17          ↳ 0));
17  end cronometro_sys;
18
19  architecture Structural of cronometro_sys is
20
21  component unita_controllo is
22      Port ( mode : in STD_LOGIC; --switch
23              salva_b : in STD_LOGIC; --salva -> 0 , visualizza ->
24                  ↳ 1, button
25              visualizza_b : in STD_LOGIC; --button per la
26                  ↳ visualizzazione
27              set_b : in STD_LOGIC;
28              rst : in STD_LOGIC; --button reset
29              clk : in STD_LOGIC;
30              set_cronometro : out STD_LOGIC;
31              en : out STD_LOGIC;
32              r_w : out STD_LOGIC;
33              incr_s : out STD_LOGIC;
34              incr_v : out STD_LOGIC;
35              rst_dp : out STD_LOGIC);
36  end component;
37
38  component unita_operativa is
39      generic ( clk_B_factor : positive);
40      Port (
41          rst : in STD_LOGIC;
42          clk : in STD_LOGIC;
43          set : in STD_LOGIC;
44          sec_in : in STD_LOGIC_VECTOR(5 downto 0);
45          min_in : in STD_LOGIC_VECTOR(5 downto 0);
46          ore_in : in STD_LOGIC_VECTOR(4 downto 0);
47          en : in STD_LOGIC;
48          r_w : in STD_LOGIC;
49          incr_v : in STD_LOGIC;
50          incr_s : in STD_LOGIC;
51          uscita_visore : out STD_LOGIC_VECTOR(23 downto
52              ↳ 0));--da dare in ingresso al display
52  end component;
53
54  signal set_cronometro : STD_LOGIC;
55  signal en : STD_LOGIC;

```

```

54  signal r_w : STD_LOGIC;
55  signal incr_s : STD_LOGIC;
56  signal incr_v : STD_LOGIC;
57  signal rst_dp : STD_LOGIC;
58
59 begin
60
61 ingresso_visore(31 downto 24) <= "00000000";
62
63 UC: unita_controllo port map(
64     mode => mode, salva_b => salva_b, visualizza_b =>
65         visualizza_b, set_b => set_b,
66     rst => rst, clk => clk,
67     set_cronometro => set_cronometro, en => en, r_w => r_w,
68         incr_s => incr_s, incr_v => incr_v, rst_dp =>
69             rst_dp
70 );
71
72 UO: unita_operativa generic map ( clk_B_factor =>
73     clk_B_factor)
74     port map(
75         rst => rst_dp, clk => clk,
76         set => set_cronometro, sec_in => sec_in, min_in =>
77             min_in, ore_in => ore_in,
78         en => en, r_w => mode, incr_v => incr_v, incr_s =>
79             incr_s, uscita_visore => ingresso_visore(23 downto
80                 0)
81 );
82
83
84 end Structural;

```

Listing 5.7: Sistema complessivo

5.3.8 Cronometro on-board

Il cronometro on-board è stato definito in maniera strutturale (in riferimento al listato 5.8), il quale è costituito dal sistema cronometro, dal display a 7 segmenti e da tre button debouncer, ovvero tutti i componenti necessari per poter sintetizzare ed implementare su board il componente sviluppato in questo capitolo. Da notare inoltre di come il valore di *clk_B_factor* è stato impostato al valore di 100000000, ovvero lo stesso valore del clock presente sulla board. In questo modo manipoliamo il ciclo di conteggio della base dei tempi del cronometro in modo tale da scandire i secondi in maniera corretta.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cronometro_on_board is
5     Port ( clk : in STD_LOGIC;
6             rst_b : in STD_LOGIC;
7             salva_b : in STD_LOGIC;
8             visualizza_b : in STD_LOGIC;
9             mode_sw : in STD_LOGIC;
10            set_b : in STD_LOGIC;
11            min_in : in STD_LOGIC_VECTOR(5 downto 0);
12            ore_in : in STD_LOGIC_VECTOR(4 downto 0);
13            anodes : out STD_LOGIC_VECTOR(7 downto 0);
14            cathodes : out STD_LOGIC_VECTOR(7 downto 0));
15 end cronometro_on_board;
16
17 architecture Structural of cronometro_on_board is
18
19 component cronometro_sys is
20     Generic ( clk_B_factor : positive);
21     Port(
22         clk : in STD_LOGIC;
23         rst : in STD_LOGIC;
24         salva_b : in STD_LOGIC;
25         visualizza_b : in STD_LOGIC;
26         mode : in STD_LOGIC;
27         set_b : in STD_LOGIC;
28         sec_in : in STD_LOGIC_VECTOR(5 downto 0);
29         min_in : in STD_LOGIC_VECTOR(5 downto 0);
30         ore_in : in STD_LOGIC_VECTOR(4 downto 0);
31         ingresso_visore : out STD_LOGIC_VECTOR(31 downto
32             ← 0));
32 end component;
33
34 component display_7seg_manager is
35     Port ( value : in STD_LOGIC_VECTOR (31 downto 0);
36             dots : in STD_LOGIC_VECTOR (7 downto 0);
37             digits_enable : in STD_LOGIC_VECTOR (7 downto 0);
38             clk : in STD_LOGIC;
39             rst : in STD_LOGIC;
40             cathodes : out STD_LOGIC_VECTOR (7 downto 0);
41             anodes : out STD_LOGIC_VECTOR (7 downto 0));
42 end component;

```

```

43
44 component button_debouncer is
45     generic (
46         clk_period : integer := 10; -- periodo del clock
47         btn_noise_time : integer := 650000000 -- stima della
        → durata del transitorio del button
48     );
49     Port ( rst : in STD_LOGIC;
50             clk : in STD_LOGIC;
51             btn : in STD_LOGIC;
52             cleared_btn : out STD_LOGIC);
53 end component;
54
55 signal set_cronometro : STD_LOGIC;
56 signal en : STD_LOGIC;
57 signal r_w : STD_LOGIC;
58 signal incr_s : STD_LOGIC;
59 signal incr_v : STD_LOGIC;
60 signal rst_dp : STD_LOGIC;
61
62 signal ingresso_visore : STD_LOGIC_VECTOR(31 downto 0);
63
64 signal cleared_salva : STD_LOGIC;
65 signal cleared_visualizza : STD_LOGIC;
66 signal cleared_set : STD_LOGIC;
67
68
69 begin
70
71     sys: cronometro_sys generic map( clk_B_factor => 100000000)
72         port map(
73             clk => clk, rst => rst_b, salva_b => cleared_salva,
74             → visualizza_b => cleared_visualizza,
75             mode => mode_sw, set_b => cleared_set,
76             sec_in => "000000", min_in => min_in, ore_in => ore_in,
77             → ingresso_visore => ingresso_visore
78         );
79
80
81     display: display_7seg_manager port map (
82         value => ingresso_visore, dots => "00010100",
83         → digits_enable => "00111111",
84         clk => clk, rst => rst_b, cathodes => cathodes,
85         → anodes => anodes);

```

```

82
83     button_salva : button_debouncer generic map(
84         CLK_period=>10, btn_noise_time=>325000000)
85     port map(
86         rst=> rst_b, clk=>clk, btn=>salva_b,
87         ↵ cleared_btn=>cleared_salva);
88
89     button_visualizza : button_debouncer generic map(
90         CLK_period=>10, btn_noise_time=>325000000)
91     port map(
92         rst=> rst_b, clk=>clk, btn=>visualizza_b,
93         ↵ cleared_btn=>cleared_visualizza);
94
95     button_set : button_debouncer generic map(
96         CLK_period=>10, btn_noise_time=>325000000)
97     port map(
98         rst=> rst_b, clk=>clk, btn=>set_b,
99         ↵ cleared_btn=>cleared_set);
100
101 end Structural;

```

Listing 5.8: Cronometro on-board

5.4 Simulazione

A questo punto, abbiamo simulato il nostro progetto. Dapprima abbiamo testato il componente base, ovvero il cronometro, con un apposito test bench (listato 5.9) dopodiché abbiamo testato il sistema complessivo 5.10.

5.4.1 Cronometro

Per testare il cronometro, come prima cosa abbiamo creato un process `clock_tb` che simula un andamento periodico del clock, come un'onda quadra di periodo 20 ps (il quale sarà poi mappato, successivamente, con il clock presente sulla board di frequenza 100MHz), dopodiché abbiamo creato un altro process in cui simuliamo il comportamento del nostro sistema. Innanzitutto, viene alzato il segnale di `rst`, che simula la pressione del button di reset del sistema. Di seguito poniamo il segnale di `set` pari ad '1', simulando in questo caso la pressione del pulsante di `set_b`, ovvero in questo modo il cronometro inizierà a contare a partire dal valore precaricato sui segnali di `sec_in`, `min_in` e `ore_in`. A questo punto, data la natura automatizzata del sistema, l'osservazione dell'uscite del cronometro avviene attraverso i segnali di uscita `sec_out`, `min_out` e `ore_out`.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity cronometro_tb is
5 end cronometro_tb;
6
7 architecture Behavioral of cronometro_tb is
8     component cronometro is
9         port(clk, rst, set : in std_logic;
10             sec_in : in std_logic_vector(5 downto 0);
11             min_in : in std_logic_vector(5 downto 0);
12             ore_in : in std_logic_vector(4 downto 0);
13
14             sec_out : out std_logic_vector(5 downto 0);
15             min_out : out std_logic_vector(5 downto 0);
16             ore_out : out std_logic_vector(4 downto 0)
17         );
18     end component;
19
20     signal clk, rst, set : std_logic;
21     signal sec_in : std_logic_vector(5 downto 0);
22     signal min_in : std_logic_vector(5 downto 0);
23     signal ore_in : std_logic_vector(4 downto 0);
```

```

24  signal sec_out : std_logic_vector(5 downto 0);
25  signal min_out : std_logic_vector(5 downto 0);
26  signal ore_out : std_logic_vector(4 downto 0);
27
28 begin
29     uut: cronometro port map (
30         clk => clk,
31         rst => rst,
32         set => set,
33         sec_in => sec_in,
34         min_in => min_in,
35         ore_in => ore_in,
36         sec_out => sec_out,
37         min_out => min_out,
38         ore_out => ore_out
39     );
40
41     clock_tb: process begin
42         clk <= '1';
43         wait for 10 ps;
44         clk <='0';
45         wait for 10 ps;
46     end process;
47
48     prc: process
49     begin
50         rst <= '1';
51         set <= '0';
52         wait for 10 ps;
53         rst <= '0';
54         wait for 100 ns;
55         sec_in <= "111000";
56         min_in <= "111011";
57         ore_in <= "01110";
58         wait for 15 ps;
59         set <= '1';
60         wait for 20 ps;
61         set <= '0';
62         wait for 200 ns;
63         rst <= '1';
64         wait for 20 ps;
65         rst <= '0';
66         wait;
67     end process;

```

```

68
69 end Behavioral;

```

Listing 5.9: Testbench cronometro

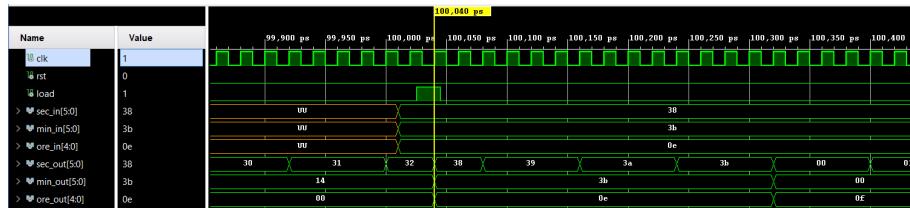


Figura 5.9: Waveform cronometro

5.4.2 Sistema complessivo

Anche in questo caso, abbiamo creato due process, uno per il clock e uno per il componente da testare. In particolare, abbiamo testato il sistema complessivo nella sua interezza andando ad alzare opportunamente i segnali di *set_b*, *salva_b* e *visualizza_b* e verificando di conseguenza l’evoluzione della macchina. In figura 5.10 sono mostrati i relativi risultati di testing del componente.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity cronometro_sys_tb is
6 end cronometro_sys_tb;
7
8 architecture Behavioral of cronometro_sys_tb is
9 signal clk : STD_logic;
10 signal rst : STD_logic;
11 signal salva_b : STD_LOGIC;
12 signal visualizza_b : STD_LOGIC;
13 signal mode : STD_LOGIC;
14 signal set_b : STD_LOGIC;
15 signal sec_in : STD_LOGIC_VECTOR(5 downto 0);
16 signal min_in : STD_LOGIC_VECTOR(5 downto 0);
17 signal ore_in : STD_LOGIC_VECTOR(4 downto 0);
18 signal ingresso_visore : STD_LOGIC_VECTOR(31 downto 0);
19
20 begin

```

```

21     uut: entity work.cronometro_sys generic map( clk_B_factor
22         => 2)
23         port map(
24             clk => clk, rst => rst, salva_b => salva_b,
25             visualizza_b => visualizza_b,
26             mode => mode, set_b => set_b,
27             sec_in => sec_in, min_in => min_in, ore_in => ore_in,
28             ingresso_visore => ingresso_visore
29         );
30
31
32
33
34
35
36
37
38
39
40
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61

```

```

62         visualizza_b <= '1';
63         wait for 20ps;
64         visualizza_b <= '0';
65
66         wait;
67
68     end process;
69
70 end Behavioral;

```

Listing 5.10: Testbench sistema complessivo

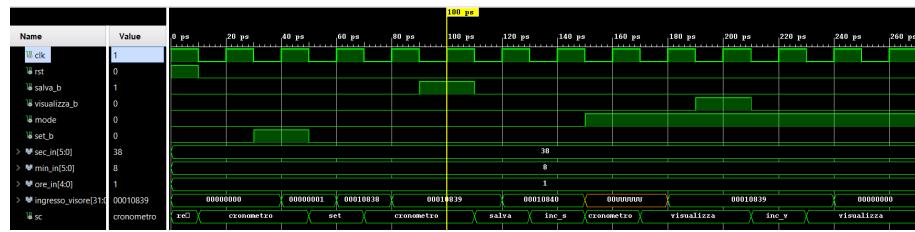


Figura 5.10: Waveform sistema complessivo

5.5 Sintesi

Il progetto del cronometro "on-board" è stato caricato sulla scheda Nexys-A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente con gli elementi di I/O presenti sulla scheda. In particolare, i quattro segnali di salva_b, visualizza_b, set_b e rst_b sono stati mappati con quattro button presenti sulla scheda, il segnale di mode (che indica la modalità di funzionamento del nostro sistema) e i segnali di input (min_in e ore_in, per mancanza di switch il segnale sec_in è sempre inizializzato a tutti zeri) sono stati mappati con gli switch presenti sulla board, mentre le linee di output sono stati mappati con gli anodi e i catodi del display a 7 segmenti.

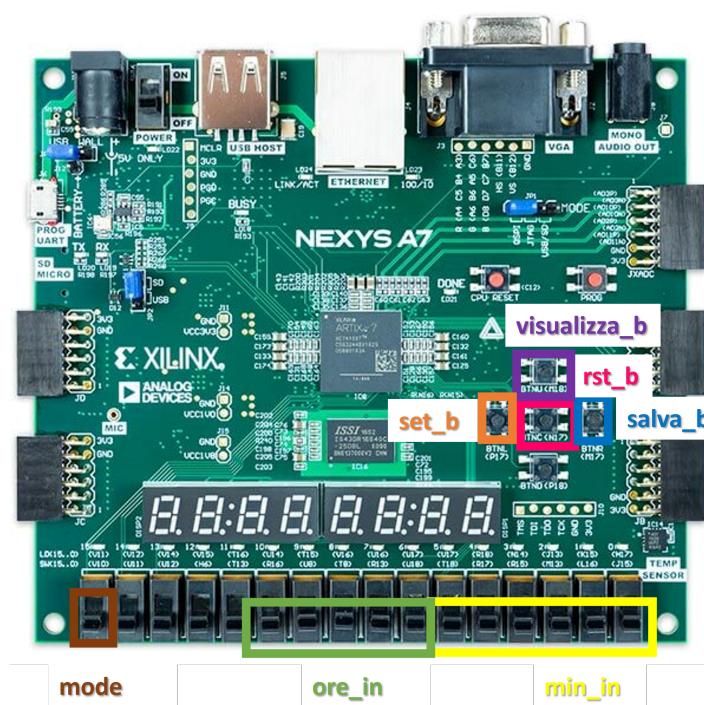


Figura 5.11: Schema di mapping del sistema cronometro

Capitolo 6

Sistema di testing

6.1 Traccia

Esercizio 6.1 Progettare, implementare in VHDL e verificare mediante simulazione un sistema in grado di testare in maniera automatica una macchina combinatoria M avente 4 ingressi e 3 uscite binarie sottoponendole N ingressi diversi (si considerino una macchina M e un numero di input N a scelta dello studente). Gli N valori di input per il test devono essere letti da una ROM, in cui essi sono precaricati, in corrispondenza di un segnale read. Le N uscite fornite della macchina in corrispondenza di ciascuno degli input devono essere memorizzate in una memoria interna, che deve poter essere svuotata in qualsiasi momento in presenza di un segnale di reset.

Esercizio 6.2 Sintetizzare ed implementare su board il componente sviluppato al punto precedente, utilizzando due buttoni per i segnali di read e reset rispettivamente e i led per la visualizzazione delle uscite della macchina istante per istante.

6.2 Soluzione

Per la progettazione del sistema di testing di una macchina combinatoria (figura 6.1), è stato adottato un approccio strutturale per la sua realizzazione. In particolare, sono stati messi insieme più componenti già studiati nei capitoli precedenti quali sono: ROM, contatore, memoria e multiplexer.

Il Sistema di testing prende in ingresso il segnale di temporizzazione *clk*, un segnale di *read* e un segnale di reset (*rst*). Invece, in uscita viene fornito il segnale per l'accensione dei led opportuni.

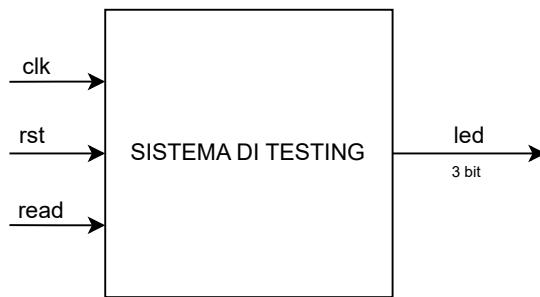


Figura 6.1: Schematico rappresentativo del sistema di testing

All'interno della ROM sono stati precaricati gli input da restituire alla nostra macchina combinatoria, la quale in uscita mostrerà la sua elaborazione. Per indirizzare la ROM attraverso il segnale *addr*, utilizziamo l'uscita di un contatore (come visto al capitolo precedente). La ROM in uscita, *data_out* fornirà il contenuto della cella letta solo se il segnale di controllo, *read*, è alto. Lo schematico di tale componente è presentato in figura 6.2.

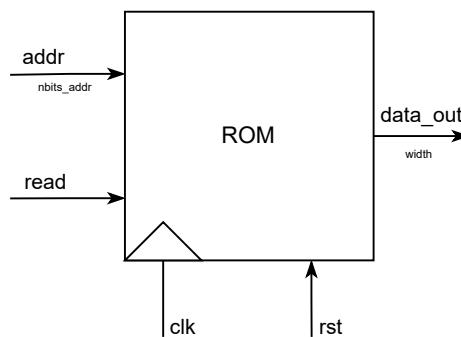


Figura 6.2: Schematico rappresentativo di una ROM

La macchina combinatoria dell'esercizio riceve in ingresso un segnale di 4 bit e restituisce in uscita un segnale di 3 bit. L'uscita è calcolata a partire dall'ingresso, semplicemente eliminando il primo bit e negati i restanti bit. Ad esempio, se il vettore di ingresso è "0001" allora l'uscita risulterà essere

"110". In figura 6.3 è mostrato lo schematico della macchina combinatoria presentata.



Figura 6.3: Schematico rappresentativo della macchina combinatoria

La progettazione dell'esercizio completo è stata divisa in due parti principali: l'Unità di Controllo (UC) e l'Unità Operativa (UO). L'unità di controllo fornisce i segnali di controllo che guidano l'unità operativa, i quali sono l'abilitazione della lettura dalla ROM, l'abilitazione del contatore per l'indirizzamento, il reset e l'abilitazione della scrittura alla memoria. I segnali di read e reset vengono forniti dall'esterno attraverso l'uso di appositi pulsanti. In particolare, premendo il pulsante di read, il contenuto della ROM, cella dopo cella, viene fornito alla macchina combinatoria, la cui uscita viene poi memorizzata nella memoria. Al termine del processo di testing, tutte le sequenze elaborate saranno scritte in memoria.

L'unità operativa è stata progettata adottando un approccio strutturale (figura 6.4), mentre l'unità di controllo è stata descritta attraverso un automa a stati finiti (figura 6.5). L'unità operativa è dunque composta da:

- un contatore che riceve in ingresso un segnale di abilitazione all'incremento (*inc*), la cui uscita serve ad indirizzare sia la ROM che la memoria (*cont_to_rom*)),
- una ROM che riceve in ingresso un segnale di *clk*, un segnale di *rst_dp*, un segnale di abilitazione alla lettura, *read*, e un segnale relativo all'indirizzo della locazione da leggere, che altro non è il valore di uscita del contatore (*cont_to_rom*), mentre l'uscita, il valore della cella letta, è fornita in ingresso alla macchina combinatoria,
- una macchina combinatoria con un segnale d'ingresso di 4 bit (*rom_to_mc*) e un segnale di uscita di 3 bit (*mc_to_mem*),
- una memoria che riceve in ingresso il segnale di *clk*, il segnale di *rst_dp*, un segnale di *en*, il quale se alto abilita la scrittura, del vettore di ingresso *mc_to_mem*, ad un locazione in memoria specificata dall'uscita del contatore,

- un multiplexer, grazie al quale in corrispondenza del segnale di *rst_dp* i led o sono spenti o sono accesi in base alla configurazione di bit di cui è composta l'uscita della macchina combinatoria.

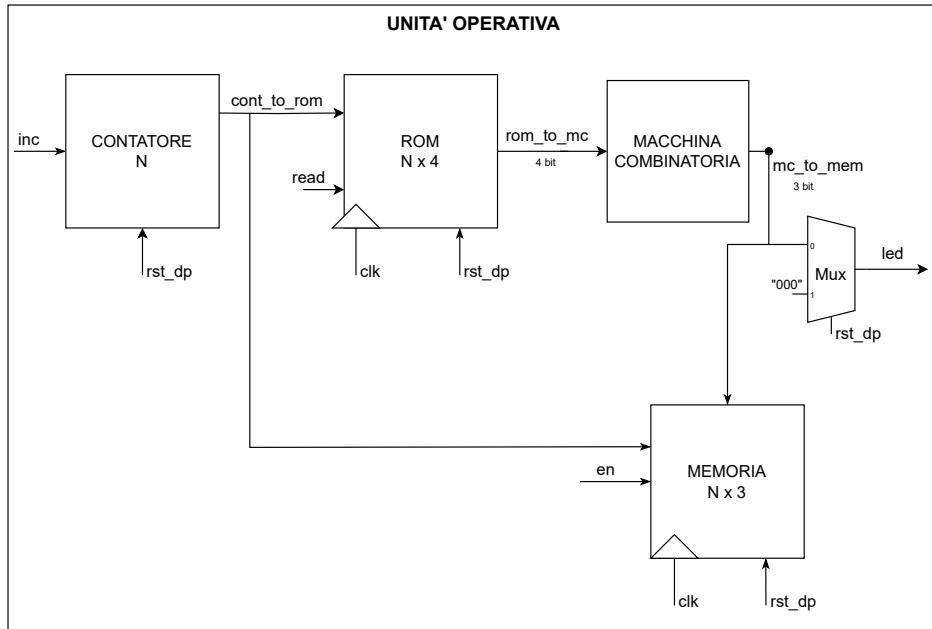


Figura 6.4: Unità operativa: sistema di testing

L'unità di controllo gestisce i segnali di comando della parte operativa. L'avvio avviene dallo stato di RESET e, con la pressione del pulsante di lettura, si passa allo stato READ_ROM, abilitando la ROM. Al successivo fronte di clock, si raggiunge uno stato di memorizzazione (MEM), leggendo dalla ROM e abilitando la scrittura nella MEM. Al successivo fronte di clock, si passa allo stato INC, dove il valore del contatore viene incrementato. Per poi passare infine o in uno stato di IDLE, il quale attende che il pulsante di read venga premuto nuovamente, oppure nello stato di READ_ROM qualora il pulsante fosse premuto mentre si è nello stato di INC.

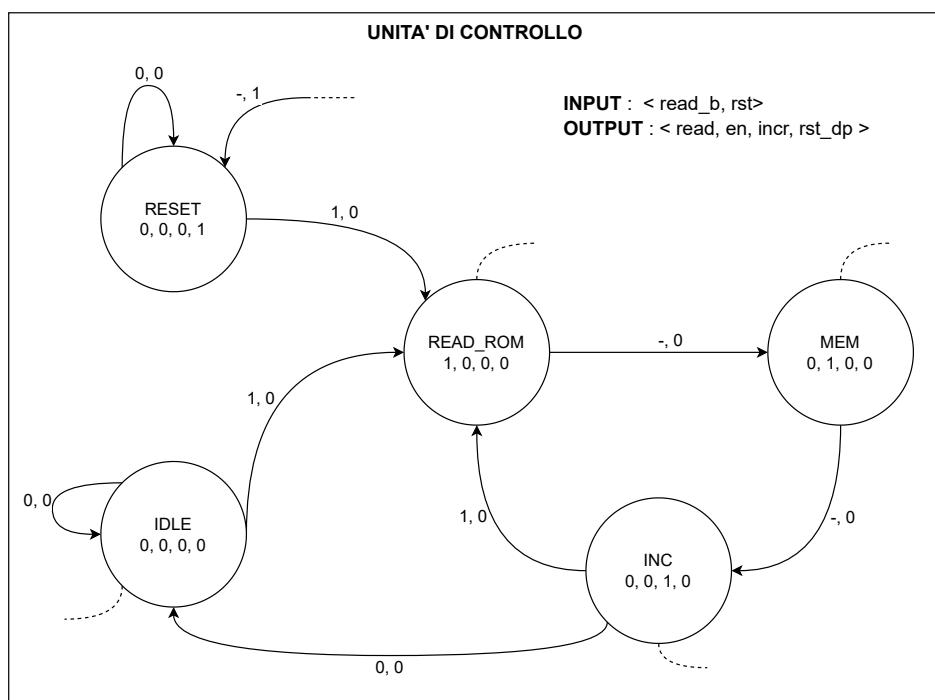


Figura 6.5: Unità di controllo: sistema di testing

6.3 Codice

6.3.1 Memoria ROM

La memoria ROM (Read-Only-Memory) è un tipo di memoria non volatile, in cui i dati scritti (o memorizzati) non possono essere modificati elettronicamente dopo l'inizializzazione. In questo esercizio, la ROM è modellata secondo un approccio comportamentale. In particolare, il processo che la descrive è sensibile al fronte di salita del clock. Quando il segnale di reset è alto, l'uscita della ROM (*data_out*) è impostata su un valore predefinito, in particolare il valore contenuto nella prima cella. Se il segnale di *read* è alto, viene restituito in uscita il valore contenuto all'indirizzo *addr*.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 use IEEE.NUMERIC_STD.ALL;
5
6 entity rom is
7     Generic (
8         nbit_addr : positive := 3;
9         width : positive := 16 );
10    Port ( addr : in STD_LOGIC_VECTOR (nbit_addr-1 downto 0);
11            clk : in STD_LOGIC;
12            read : in STD_LOGIC;
13            rst : in STD_LOGIC;
14            data_out : out STD_LOGIC_VECTOR (width-1 downto 0));
15 end rom;
16
17 architecture behavioral of rom is
18     constant depth : positive := 2**nbit_addr;
19     subtype word is std_logic_vector(width-1 downto 0);
20     type rom_type is array(0 to depth-1) of word;
21
22     signal rom : rom_type := (
23         x"1",
24         x"2",
25         x"3",
26         x"4",
27         x"5",
28         x"6",
29         x"7",
30         x"8" );
```

31

```

32 begin
33     process(clk)
34     begin
35         if clk = '1' and clk'event then
36             if rst = '1' then
37                 data_out <= rom(0);
38             else
39                 if read = '1' then
40                     data_out <=
41                         → rom(to_integer(unsigned(addr)));
42                 end if;
43             end if;
44         end process;
45
46 end behavioral;

```

Listing 6.1: Memoria ROM

6.3.2 Macchina combinatoria

La macchina combinatoria è stata realizzata a livello dataflow, eseguendo come operazione base una semplice not. La sua descrizione è mostrata al listato 6.2.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity macchina_combinatoria is
5     Port ( x : in STD_LOGIC_VECTOR(3 downto 0);
6             y : out STD_LOGIC_VECTOR(2 downto 0));
7 end macchina_combinatoria;
8
9 architecture Dataflow of macchina_combinatoria is
10
11 begin
12     y(0) <= not x(0);
13     y(1) <= not x(1);
14     y(2) <= not x(2);
15
16 end Dataflow;

```

Listing 6.2: Macchina combinatoria

6.3.3 Contatore

Per la realizzazione del contatore di indirizzamento nelle memorie, è stato fatto uso del componente counter (generico) precedentemente realizzato, di cui viene riportato al listato 6.3 il suo codice.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.std_logic_unsigned.all;
4
5 use IEEE.NUMERIC_STD.ALL;
6
7 entity counter is
8     generic (
9         bits : positive := 3;
10        delay : time := 0ns );
11    Port ( x : in STD_LOGIC;
12            rst : in STD_LOGIC;
13            y : buffer STD_LOGIC_VECTOR (bits-1 downto 0);
14            div : out STD_LOGIC);
15 end counter;
16
17 architecture behavioral of counter is
18
19 constant divider : natural := 2**bits - 1;
20
21 begin
22     process(x)
23     begin
24         if rst = '1' then
25             y <= (others => '0');
26         elsif x'event and x = '1' then
27             y <= y + "1" after delay;
28         end if;
29     end process;
30
31     process(y)
32     begin
33         if y = std_logic_vector(to_unsigned(divider,y'length))
34             then
35             div <= '1';
36         else
37             div <= '0';
38         end if;
```

```

38     end process;
39
40 end behavioral;
```

Listing 6.3: Contatore

6.3.4 Unità operativa

L’unità operativa riceve i segnali di controllo dall’unità di controllo e ha come uscita la combinazione di led da accendere. Viene descritta dal listato 6.4 secondo un approccio strutturale, composto da tutti i componenti elementari presentati precedentemente in questo capitolo.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity unita_operativa is
5      Port ( read : in STD_LOGIC;
6              en : in STD_LOGIC;
7              inc : in STD_LOGIC;
8              rst_dp : in STD_LOGIC;
9              clk : in STD_LOGIC;
10             led : out STD_LOGIC_VECTOR(2 downto 0));
11 end unita_operativa;
12
13 architecture Structural of unita_operativa is
14
15
16 signal cont_to_rom : std_logic_vector(2 downto 0);
17 signal rom_to_mc : std_logic_vector(3 downto 0);
18 signal mc_to_mem : std_logic_vector(2 downto 0);
19
20
21 begin
22
23     cont : entity work.counter generic map(
24         bits => 3)
25         port map(
26             x => inc, rst=> rst_dp, y => cont_to_rom
27         );
28
29     rom : entity work.rom generic map(nbit_addr => 3,
30                                     width => 4 )
```

```

31      port map(
32        addr => cont_to_rom, clk => clk, read => read, rst =>
33          ↵  rst_dp, data_out => rom_to_mc
34      );
35
36      m : entity work.macchina_combinatoria port map(
37        x => rom_to_mc, y => mc_to_mem
38      );
39
40      memoria : entity work.memory generic map( nbits_addr => 3,
41        ↵  width => 3 )
42      port map (
43        addr => cont_to_rom, en => read, clk => clk, rst =>
44          ↵  rst_dp,
45        data_in =>mc_to_mem, r_w =>'0'
46      );
47
48
49
50 end Structural;

```

Listing 6.4: Unita operativa: sistema di testing

6.3.5 Unità di controllo

L’unità di controllo presenta come ingressi i segnali provenienti dall’esterno, ovvero i segnali di read, rst e clk, e ha come uscite i segnali di controllo (ingressi per l’unità operativa). Il comportamento viene definito, come detto in precedenza, attraverso un’automma a stati finiti descritto, in forma di codice VHDL, al listato 6.5.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_controllo is
5   Port ( read_b : in STD_LOGIC;
6         rst : in STD_LOGIC;
7         clk : in std_logic;
8         read : out STD_LOGIC;

```

```

9      en : out STD_LOGIC;
10     incr : out STD_LOGIC;
11     rst_dp : out STD_LOGIC);
12 end unita_controllo;
13
14 architecture Behavioral of unita_controllo is
15
16 type stato is (reset, read_rom, idle, mem, inc);
17 signal sc : stato := reset; --stato corrente
18 signal next_state : stato;
19
20 begin
21     process(clk)
22     begin
23         if(falling_edge(clk)) then
24             if(rst = '1') then
25                 sc <= reset;
26             else sc <= next_state;
27             end if;
28         end if;
29     end process;
30
31     process(sc, read_b, rst)
32     begin
33         case sc is
34             when reset =>
35                 read <= '0';
36                 en <= '0';
37                 incr <= '0';
38                 rst_dp <= '1';
39
40             if(read_b = '1') then
41                 next_state <= read_rom;
42             else next_state <= sc;
43             end if;
44
45             when idle =>
46                 read <= '0';
47                 en <= '0';
48                 incr <= '0';
49                 rst_dp <= '0';
50
51             if(read_b = '1') then
52                 next_state <= read_rom;

```

```

53                     else next_state <= sc;
54                 end if;
55
56
57             when read_rom =>
58                 read <= '1';
59                 en <= '0';
60                 incr <= '0';
61                 rst_dp <= '0';
62
63                     next_state <= mem;
64
65
66             when mem =>
67                 read <= '0';
68                 en <= '1';
69                 incr <= '0';
70                 rst_dp <= '0';
71
72                     next_state <= inc;
73
74             when inc =>
75                 read <= '0';
76                 en <= '0';
77                 incr <= '1';
78                 rst_dp <= '0';
79
80                 if(read_b = '1') then
81                     next_state <= read_rom;
82                 else next_state <= idle;
83                 end if;
84
85             end case;
86         end process;
87
88
89     end Behavioral;

```

Listing 6.5: Unita di controllo: sistema di testing

6.3.6 Sistema di testing on-board

Il sistema di testing on-board viene realizzato adottando un approccio strutturale, istanziando come componenti l'unità di controllo, l'unità operativa

e i button debouncer (definiti in precedenza) e collegandoli in modo appropriato per la costruzione dell'architettura complessiva. Ciò avviene mediante l'utilizzo di appositi segnali interni, come viene mostrato al listato 6.6.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity sistema_di_testing_on_board is
5     Port ( clk : in STD_LOGIC;
6             rst_b : in STD_LOGIC;
7             read_b : in STD_LOGIC;
8             led : out STD_LOGIC_VECTOR(2 downto 0));
9 end sistema_di_testing_on_board;
10
11 architecture Structural of sistema_di_testing_on_board is
12
13 signal cleared_read : STD_LOGIC;
14
15 begin
16
17     sistema_testing : entity work.sistema_di_testing port map(
18         clk => clk, rst => rst_b, read => cleared_read, led
19         => led
20     );
21
22     button_read : entity work.button_debouncer generic map (
23         clk_period => 10, btn_noise_time => 650000000)
24         port map ( rst => rst_b, clk => clk, btn => read_b,
25         cleared_btn => cleared_read);
26
27 end Structural;
```

Listing 6.6: Sistema di testing on-board

6.4 Simulazione

A questo punto del progetto, abbia simulato il nostro componente con un apposito test bench (listato 6.7). Innanzitutto, abbiamo creato un process clock_tb che simula un andamento periodico del clock, come un'onda quadra di periodo 20 ns, dopodiché abbiamo creato un altro process in cui simuliamo il comportamento del nostro sistema. Innanzitutto, viene alzato il segnale di reset, che simula la pressione del button di reset del sistema. Di seguito poniamo il segnale di read pari ad '1', simulando in questo caso la pressione del pulsante di read. A questo punto, data la natura automatizzata del sistema, l'osservazione dell'uscite della macchina combinatoria istante per istante avviene attraverso il segnale denominato led.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity sistema_di_testing_tb is
5 end sistema_di_testing_tb;
6
7 architecture Behavioral of sistema_di_testing_tb is
8
9 signal clk : STD_LOGIC;
10 signal rst : STD_LOGIC;
11 signal read : STD_LOGIC;
12 signal led : STD_LOGIC_VECTOR(2 downto 0);
13
14 begin
15
16     uut : entity work.sistema_di_testing port map(
17         clk => clk, rst => rst, read => read, led => led
18     );
19
20     clock_tb: process begin
21         clk <= '1';
22         wait for 10 ns;
23         clk <='0';
24         wait for 10 ns;
25     end process;
26
27     prc: process
28     begin
29         rst <= '1';
30         wait for 10 ns;
31         rst <= '0';
```

```

32          wait for 20 ns;
33          read<='1';
34          wait for 500 ns;
35          read<='0';
36          wait;
37      end process;
38
39
40 end Behavioral;

```

Listing 6.7: Testbench sistema di testing

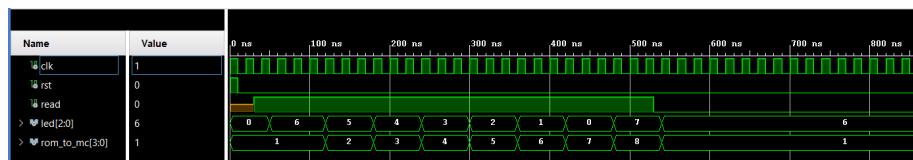


Figura 6.6: Waveform sistema di testing

6.5 Sintesi

Il progetto del sistema di testing "on-board" è stato caricato sulla scheda Nexys-A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente con gli elementi di I/O presenti sulla scheda. In particolare, i due segnali di `read_b` e `rst_b` sono stati mappati con due button presenti sulla scheda, mentre le tre linee di output che rappresentano l'uscita della macchina combinatoria sono state mappate sui LED.

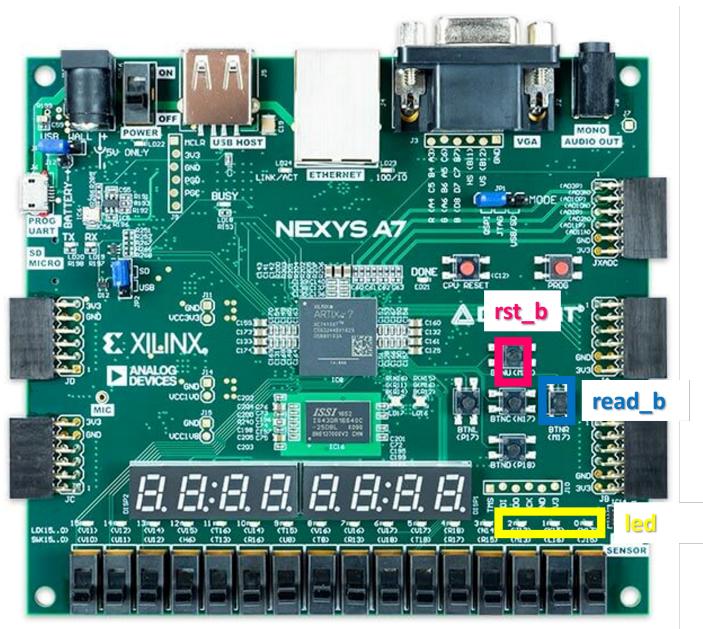


Figura 6.7: Schema di mapping del sistema di testing

Capitolo 7

Comunicazione con handshaking

7.1 Traccia

Esercizio 7.1 Progettare, implementare in VHDL e testare mediante simulazione un sistema composto da 2 nodi, A e B, che comunicano mediante un protocollo di handshaking. Il nodo A e il nodo B possiedono entrambi una memoria interna in cui sono memorizzate N stringhe di M bit, denominate $X(i)$ e $Y(i)$ rispettivamente ($i = 0, \dots, N - 1$). Il nodo A trasmette a B ciascuna stringa $X(i)$ utilizzando un protocollo di handshaking; B, ricevuta la stringa $X(i)$, calcola $S(i) = X(i) + Y(i)$ e immagazzina la somma in opportune locazioni della propria memoria interna. Per il progetto è possibile considerare una implementazione di tipo comportamentale per effettuare la somma, mentre è necessario prevedere esplicitamente un componente contatore sia nel sistema A sia nel sistema B per scandire la trasmissione/ricezione delle stringhe e per terminare la comunicazione.

7.2 Soluzione

L'esercizio presentato al paragrafo 7.1 richiede la progettazione di un sistema composto da due nodi distinti A e B che comunicano tra di loro. In particolare, il nodo A invia delle stringhe di bit al nodo B, il quale andrà poi ad effettuare una somma tra i dati ricevuti e altri memorizzati al suo interno. Per la soluzione, si è scelto di progettare il nodo A ed il nodo B come due nodi completamente separati, senza un riferimento temporale comune. Per ottenere due riferimenti temporali distinti, sia in fase che in frequenza. si è utilizzata una *base dei tempi*, la quale a partire da un segnale temporale in ingresso (CLK) ne genera altri due in uscita:

- CLK_A: coincidente con CLK;
- CLK_B: con un periodo multiplo di quello di CLK, e sfasato rispetto ad esso.

In particolare, il CLK_B avrà un periodo 4 volte quello di CLK e sarà sfasato rispetto ad esso di 4 ns.

Affinché i due nodi A e B possano comunicare tra di loro senza un riferimento temporale comune c'è bisogno di instaurare un protocollo di comunicazione tra di essi, più precisamente un protocollo di comunicazione *asincrono*. La soluzione implementa un protocollo asincrono semplice, che prevede un meccanismo di *handshaking* ogni volta che avviene lo scambio di una word di dati (lunga 32 bit nel progetto). Il porto di comunicazione tra il nodo A e il nodo B è composto da tre elementi:

- la linea DATA composta da 32 fili che trasportano il dato, scritta dal nodo A e letta dal nodo B;
- il segnale di READY utilizzato dal nodo A per indicare a B che il dato è pronto per essere letto;
- il segnale ACK controllato da B, utilizzato per comunicare ad A che il dato sulla linea è stato letto.

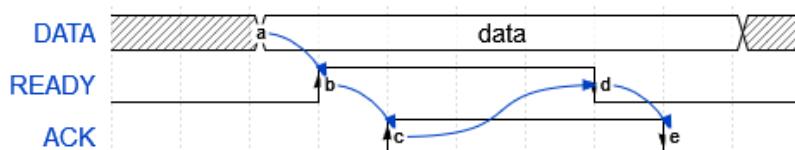


Figura 7.1: Procedura di handshake

In figura 7.1 è riportata la sequenza di handshaking completa.

- (a) il dato viene scritto sulla linea DATA dal nodo A;

- (b) il nodo A alza il segnale di **READY** per indicare a B che il dato è pronto per essere letto;
- (c) il nodo B legge il dato dalla linea, dopodiché alza il segnale di **ACK** per notificare l'avvenuta lettura;
- (d) il nodo A, visto che il segnale **ACK** si è alzato, procede ad abbassare il segnale di **READY**;
- (e) il nodo B, visto che il segnale di **READY** si è abbassato provvede ad abbassare il segnale **ACK**, concludendo la procedura di handshaking.

Osservando il progetto da un punto di vista ad alto livello (figura 7.2) è possibile individuare tre componenti principali:

- la base dei tempi
- il nodo A
- il nodo B.

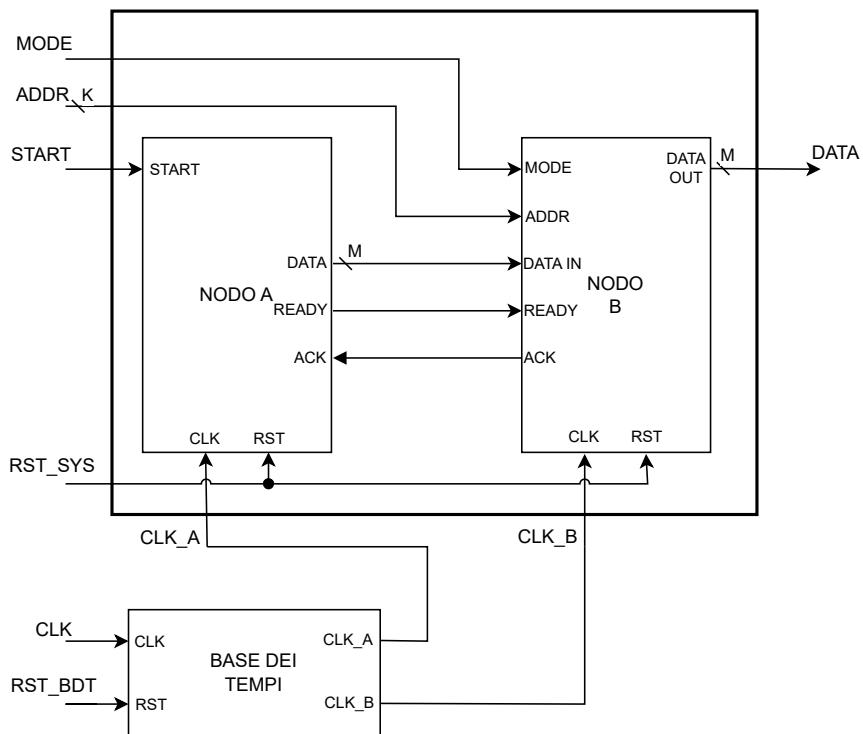


Figura 7.2: Sistema di comunicazione tramite handshake: schematico

7.2.1 Base dei tempi

Si parte con l'analizzare la base dei tempi, che è il componente più semplice tra quelli che compongono il progetto. Si ricordi che la base dei tempi deve, dato un riferimento temporale in ingresso CLK, generare due riferimenti temporali in uscita, di cui:

- uno, CLK_A, identico a CLK;
- l'altro, CLK_B, con periodo 4 volte quello di CLK e una differenza in fase di 4 ns.

Per implementare il primo punto basta semplicemente portare il segnale d'ingresso CLK in uscita così com'è. Per il secondo punto invece, è necessario filtrare il segnale di clock in ingresso, facendolo passare attraverso un contatore, per ridurne la frequenza, e attraverso un elemento di ritardo per aggiungere un ritardo di fase. Per generare un segnale di clock con un periodo N volte

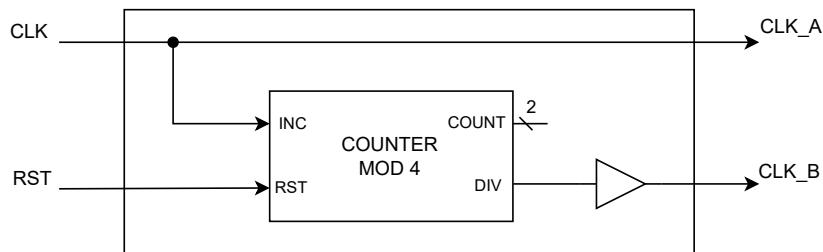


Figura 7.3: Base dei tempi: schematico

superiore a quello di un segnale fissato, si utilizza un contatore modulo N, e in particolare la sua uscita div. L'uscita div di un contatore trasporta un segnale che è alto quando il contatore raggiunge il valore massimo di conteggio, e basso altrimenti. Dato che nella soluzione implementata, il segnale CLK_B deve avere un periodo 4 volte quello di CLK, si è utilizzato un contatore modulo 4.

7.2.2 Nodo A

Proseguendo l'analisi dei componenti dal più semplice al più complesso, è il turno del nodo A. Per progettare tale componente si è adottata la metodologia di progetto che prevede la scomposizione dell'elemento in due sub-unità funzionali:

- l'unità operativa.
- l'unità di controllo;

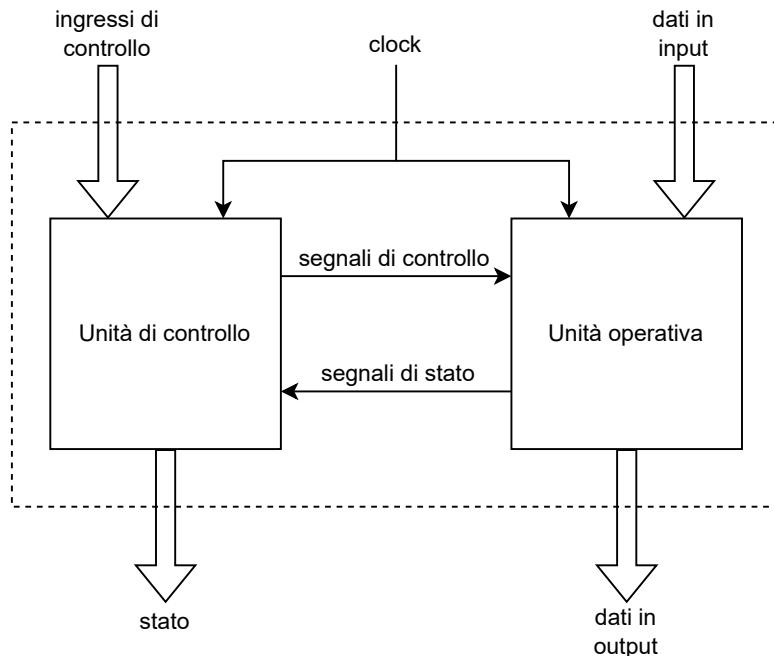


Figura 7.4: Unità operativa e unità di controllo

L'*unità operativa* ha il compito di elaborare i dati in ingresso e produrre dei dati in uscita, operando sotto il controllo di opportuni *segnali di controllo* e generando dei segnali che codificano lo stato attuale dell'unità stessa verso l'unità di controllo.

L'*unità di controllo*, invece, ha la funzione di generare i segnali di controllo per l'unità operativa, a partire da *ingressi di controllo* esterni e i *segnali di stato* provenienti dall'unità operativa. Inoltre, può inoltrare verso l'esterno dei segnali che codificano lo stato attuale della macchina.

Unità operativa

Dunque il nodo A è stato progettato nell'ottica della suddetta metodologia, la quale richiede sempre di progettare prima l'unità operativa, e solo in seguito la relativa unità di controllo. Si ricordi che tale nodo ha il compito di inviare al nodo B una sequenza di N stringhe, ognuna lunga M bit. Quindi, si è dotata l'unità operativa di una memoria ROM di dimensioni $N \times M$, per memorizzare i dati da inviare, e di un contatore modulo N per indirizzare la memoria. Vedesi l'unità operativa in figura 7.5. I dati in uscita dalla ROM alimentano la linea DATA verso il nodo B, mentre il segnale di READY sarà governato dall'unità di controllo. Affinché il circuito funzioni correttamente è necessario orchestrare correttamente l'evoluzione dei segnali di controllo, che sono:

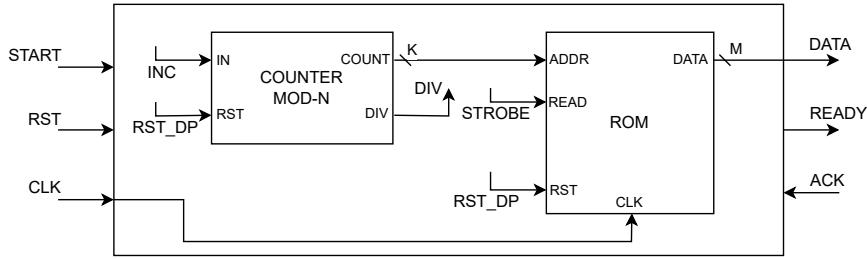


Figura 7.5: Nodo A: unità operativa

1. INC: il segnale di conteggio del contatore;
2. STROBE: il segnale di strobe della memoria ROM;
3. RST_DP: il segnale per resettare i componenti dell'unità operativa.

Inoltre, l'unità operativa genera un segnale di stato che sarà utilizzato come input dall'unità di controllo: il segnale DIV in uscita al contatore.

Unità di controllo

Per progettare l'unità di controllo del nodo A occorre modellare il comportamento della macchina, ad esempio con un automa a stati finiti. Il nodo A deve, a seguito dell'arrivo di un impulso di START, procedere all'invio in sequenza di tutte le N stringhe contenute all'interno della memoria ROM, coordinandosi con il nodo B attraverso i segnali READY e ACK di handshaking, e infine ritornare in uno stato di riposo in cui attendere il prossimo impulso di START. Tale comportamento è stato modellato con l'automa di Moore in figura 7.6.

Gli ingressi dell'automa comprendono tre segnali di controllo esterni (START, ACK, RST) e un segnale di stato proveniente dall'unità operativa (DIV), mentre i quattro segnali in uscita si compongono di tre per il controllo dell'unità operativa (INC, STROBE, RST_DP) e uno di stato (READY) diretto verso il nodo B. L'automa in figura 7.6 può essere facilmente implementato con una macchina sequenziale ad-hoc.

7.2.3 Nodo B

L'ultimo componente da analizzare dell'architettura della soluzione qui presentata è il nodo B. Anche tale componente è stato realizzato secondo la metodologia atta alla progettazione di sistemi complessi, che prevede la scomposizione dell'elemento in un'unità operativa ed un'unità di controllo. Prima di vedere come sono fatte le due sub-unità, si ricordi il funzionamento del nodo B. Tale nodo deve:

INPUT: <START, DIV, ACK, RST>
OUTPUT: <INC, STROBE, READY, RST_DP>

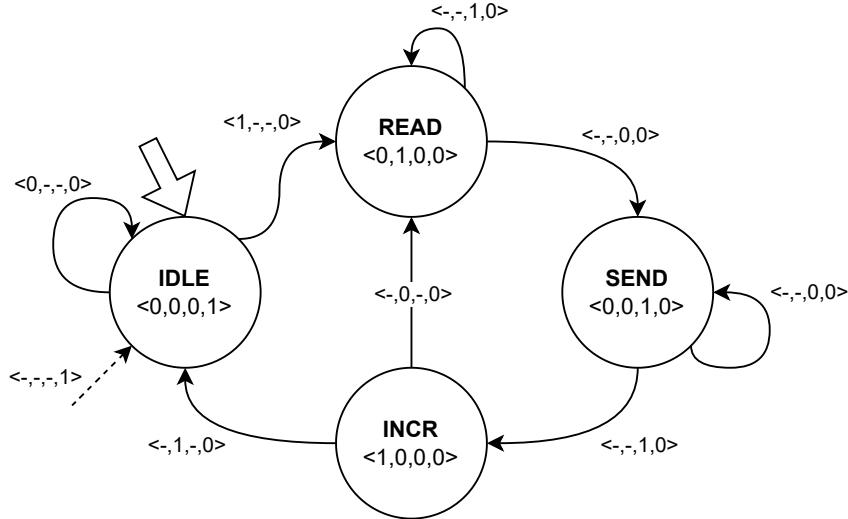


Figura 7.6: Nodo A: unità di controllo

1. restare in attesa che arrivi un nuovo dato sulla linea DATA aspettando che si alzi il segnale di READY;
2. nel momento in cui si alza tale segnale, leggere la stringa di bit dalla linea dati e chiudere la procedura di handshake governando opportunamente il segnale ACK;
3. sommare il dato $X(i)$ appena letto con il dato $Y(i)$ contenuto in una memoria interna al nodo, e salvare il risultato $S(i) = X(i) + Y(i)$ in un'altra memoria del nodo B;
4. ripartire dal punto 1.

Per poter controllare il corretto funzionamento della macchina, il nodo B è stato dotato di una seconda modalità di funzionamento, che consente di leggere il contenuto della memoria che salva i risultati $S(i)$ della somma e li porta verso l'esterno del nodo.

Unità operativa

Si comincia sempre con l'unità operativa, che dev'essere dotata di tutti i componenti necessari affinché il nodo B elabori i dati correttamente. Per quanto detto al paragrafo precedente, è già possibile individuare alcuni di questi componenti:

- una memoria ROM che contenga le stringhe di bit $Y(i)$ da sommare a quelle lette dalla linea DATA;

- una memoria che funzioni sia in lettura che in scrittura per leggere e salvare i risultati $S(i)$ dell'operazione di somma;
- almeno un contatore attivo ad indirizzare le due memorie precedenti;
- un *adder* che ha il compito di eseguire l'operazione di somma;
- un registro tampone che permetta di separare la lettura del dato $X(i)$ dalla linea DATA, dall'operazione di somma $S(i) = X(i) + Y(i)$.

In questa soluzione si è scelto di utilizzare due memorie distinte per memorizzare i valori $Y(i)$ e $S(i)$, ma se ne potrebbe utilizzare anche una sola, sovrascrivendo ogni volta i valori precedenti. Inoltre, dato che la memoria ROM e l'altra memoria, che chiameremo **MEM** per semplicità, sono sempre indirizzate dallo stesso valore, perché se si sta leggendo il dato $Y(i)$ significa che si sta memorizzando il risultato $S(i)$ (stesso indice), allora è necessario un solo contatore per indirizzarle entrambe. Si veda l'unità operativa appena descritta in figura 7.7. Rispetto a quanto appena detto, c'è un'altra cosa

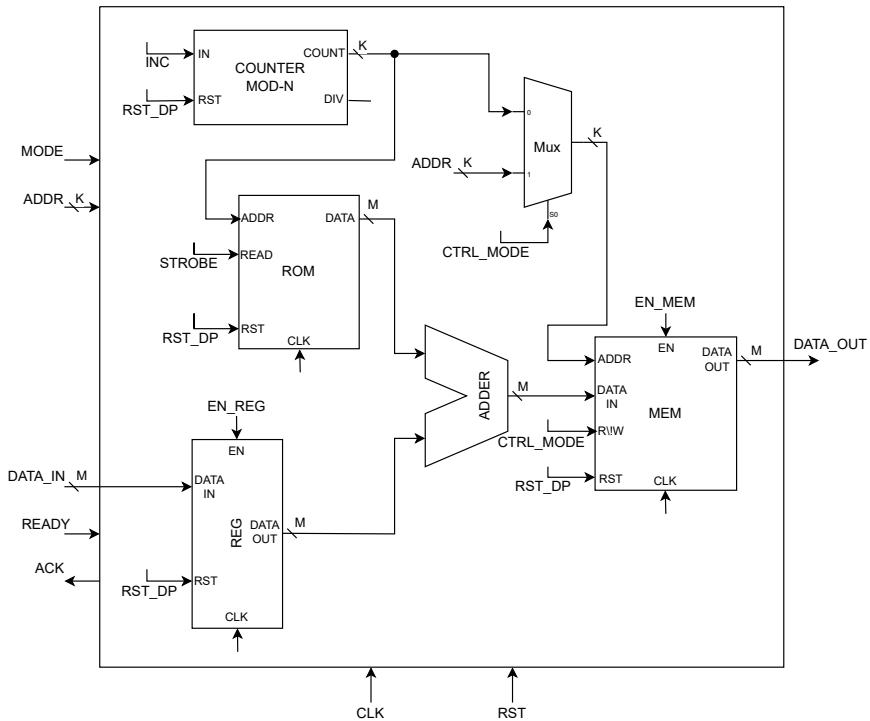


Figura 7.7: Nodo B: unità operativa

da aggiungere. Come si vede in figura 7.7, la linea indirizzo della memoria **MEM** è l'uscita di un multiplexer controllato dal segnale **CTRL_MODE**, e che ha come ingressi l'uscita **COUNT** del contatore e il segnale esterno di **ADDR**. Inoltre, il segnale **CTRL_MODE** lo ritroviamo anche sul pin di input di **MEM**, che

regola la modalità di funzionamento della memoria, in lettura o in scrittura. Tali soluzioni architetturali sono state aggiunte per permettere al nodo B di funzionare nelle due modalità descritte nel paragrafo precedente:

Modalità 0 (CTRL_MODE = 0) il nodo B calcola i valori $S(i)$ e li scrive nella memoria MEM all'indirizzo indicato dall'uscita del contatore;

Modalità 1 (CTRL_MODE = 1) il nodo B legge i valori memorizzati in MEM all'indirizzo impostato dal segnale esterno di ADDR, e li porta sulla linea di uscita DATA_OUT.

Unità di controllo

Il funzionamento del nodo B è stato già descritto in precedenza; non resta che tradurlo in un automa (figura 7.8). L'unità di controllo riceve come input

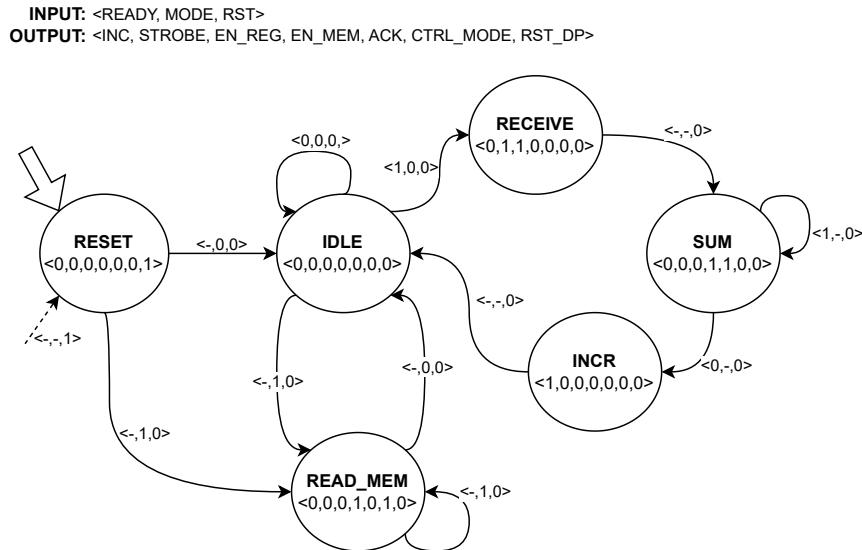


Figura 7.8: Nodo B: unità di controllo

i segnali:

- **READY**: dal nodo A per indicare che il dato in ingresso è pronto per essere letto;
- **MODE**: dall'esterno per selezionare la modalità di funzionamento del nodo B;
- **RST**: per riportare il nodo alla configurazione nota iniziale.

Per governare l'unità operativa, l'unità di controllo produce ben sei segnali, più uno verso il nodo A per effettuare la procedura di handshake.

- INC: per incrementare il valore del contatore;
- STROBE: segnale di strobe di lettura per la memoria ROM;
- EN_REG: per abilitare il registro atto a memorizzare il dato ricevuto dal nodo A;
- EN_MEM: per abilitare la memoria MEM;
- ACK: per indicare al nodo A di aver letto il dato da esso inviato;
- CTRL_MODE: per impostare l'unità operativa in modalità 0 o 1;
- RST_DP: per resettare l'unità operativa.

Anche in questo caso, l'automa in figura 7.8 può essere facilmente implementato con una macchina sequenziale.

7.3 Codice

In seguito è riportato il codice VHDL utilizzato per implementare la soluzione descritta al paragrafo 7.2.

7.3.1 Base dei tempi

Per realizzare la base dei tempi si è utilizzato un approccio strutturale istanziando un contatore e collegandolo opportunamente ai pin di ingresso e uscita. Il fattore moltiplicativo del periodo del clock B rispetto al periodo del clock A è stato definito con un generic, così come il ritardo di fase del primo rispetto al secondo. Anche il contatore richiede due generic per essere istanziato:

`bits` il numero di bit da usare per il conteggio;

`delay` il ritardo con il quale generare il segnale `div` in uscita, rispetto al segnale `x` in ingresso.

Il secondo valore coincide esattamente con `clk_B_delay`, mentre il primo può essere ricavato da `clk_B_factor` secondo la relazione:

$$bits = \lceil \log_2(clk_B_factor) \rceil$$

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.math_real.all;
4
5 entity base_dei_tempi is
6     generic(
7         clk_B_factor : positive := 4;
8         clk_B_delay : time := 4ns );
9     Port ( clk : in STD_LOGIC;
10            rst : in STD_LOGIC;
11            clk_A : out STD_LOGIC;
12            clk_B : out STD_LOGIC);
13 end base_dei_tempi;
14
15 architecture structural of base_dei_tempi is
16     component counter is
17         generic (
18             bits : positive :=
19             positive(ceil(log2(real(clk_B_factor))));
```

```

19      delay : time := clk_B_delay );      -- delay per
20      -- sfasare clk_A e clk_B
21  Port ( x : in STD_LOGIC;
22          rst : in STD_LOGIC;
23          y : buffer STD_LOGIC_VECTOR (bits-1 downto 0);
24          div : out STD_LOGIC);
25
26 begin
27     clk_A <= clk;
28
29     counter0 : counter
30         port map(
31             x => clk,
32             rst => rst,
33             div => clk_B );
34
35 end structural;

```

Listing 7.1: Base dei tempi

7.3.2 Nodo A

Il nodo A è stato realizzato secondo un approccio strutturale, prima progettando separatamente parte operativa e parte di controllo, e poi collegandole insieme (vedi codice 7.2). Anche il nodo A è stato realizzato definendo due generic che ritroveremo in tutto il progetto:

- **M**: la lunghezza in bit di ogni stringa dato $X(i)$ o $Y(i)$;
- **K**: il numero di bit da utilizzare per indirizzare le memorie, ovvero, indirettamente, il numero massimo di stringhe $X(i)$ o $Y(i)$.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA is
5     Generic(
6         M : positive := 32;
7         K : positive := 3 );
8     Port ( start : in STD_LOGIC;
9             ack : in STD_LOGIC;
10            rst : in STD_LOGIC;

```

```

11      clk : in STD_LOGIC;
12      ready : out STD_LOGIC;
13      data_out : out STD_LOGIC_VECTOR (31 downto 0));
14  end nodoA;
15
16 architecture structural of nodoA is
17     component nodoA_CU is
18         Port ( start : in STD_LOGIC;
19                 ack : in STD_LOGIC;
20                 div : in STD_LOGIC;
21                 rst : in STD_LOGIC;
22                 clk : in STD_LOGIC;
23                 ready : out STD_LOGIC;
24                 inc : out STD_LOGIC;
25                 strobe : out STD_LOGIC;
26                 rst_dp : out STD_LOGIC);
27     end component;
28
29     component nodoA_datapath is
30         Generic(
31             M : positive;
32             K : positive );
33         Port ( inc : in STD_LOGIC;
34                 strobe : in STD_LOGIC;
35                 rst_dp : in STD_LOGIC;
36                 clk : in STD_LOGIC;
37                 data_out : out STD_LOGIC_VECTOR (31 downto 0);
38                 div : out STD_LOGIC );
39     end component;
40
41     -- segnali di stato
42     signal div : std_logic;
43
44     -- segnali di controllo
45     signal increment : std_logic;
46     signal strobe : std_logic;
47     signal reset_dp : std_logic;
48
49 begin
50     dp : nodoA_datapath
51     generic map(
52         M => M,
53         K => K )
54     port map(

```

```

55      inc => increment,
56      strobe => strobe,
57      rst_dp => reset_dp,
58      clk => clk,
59      data_out => data_out,
60      div => div );
61
62  cu : nodoA_CU port map(
63      start => start,
64      ack => ack,
65      ready => ready,
66      clk => clk,
67      rst => rst,
68      div => div,
69      inc => increment,
70      strobe => strobe,
71      rst_dp => reset_dp );
72
73 end structural;

```

Listing 7.2: Nodo A

Unità operativa

L’unità operativa del nodo A è stata realizzata naturalmente con un approccio strutturale, collegando tra di loro un contatore modulo 2^K con una memoria ROM $2^K \times M$.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA_datapath is
5     Generic(
6         M : positive := 32;
7         K : positive := 3 );
8     Port ( inc : in STD_LOGIC;
9             strobe : in STD_LOGIC;
10            rst_dp : in STD_LOGIC;
11            clk : in std_logic;
12            data_out : out STD_LOGIC_VECTOR (M-1 downto 0);
13            div : out STD_LOGIC );           -- segnale di stato
14                                     -- del contatore
14 end nodoA_datapath;

```

```

15
16  architecture structural of nodoA_datapath is
17      component counter is
18          generic (
19              bits : positive := K);
20              Port ( x : in STD_LOGIC;
21                      rst : in STD_LOGIC;
22                      y : buffer STD_LOGIC_VECTOR (bits-1 downto 0);
23                      div : out STD_LOGIC);
24      end component;
25
26      component rom is
27          generic(
28              nbit_addr : positive := K;           -- numero di bit
29              width : positive := M );           -- dimensione di una
30              width word di memoria
31              Port ( addr : in STD_LOGIC_VECTOR (nbit_addr - 1 downto
32                  0);
33                      clk : in STD_LOGIC;
34                      read : in STD_LOGIC;
35                      rst : in STD_LOGIC;
36                      data_out : out STD_LOGIC_VECTOR (width - 1
37                          downto 0));
38      end component;
39
40      signal address : std_logic_vector (K-1 downto 0);
41
42 begin
43     counter0 : counter port map(
44         x => inc,
45         rst => rst_dp,
46         y => address,
47         div => div );
48
49     rom0 : rom port map(
50         addr => address,
51         clk => clk,
52         rst => rst_dp,
53         read => strobe,
54         data_out => data_out );
55
56 end structural;

```

Listing 7.3: Nodo A: unità operativa

Unità di controllo

L'unità di controllo è stata descritta secondo un approccio *behavioral*, tipico per la realizzazione di automi a stati finiti, con una struttura a due process.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA_CU is
5     Port ( start : in STD_LOGIC;
6             ack : in STD_LOGIC;
7             div : in STD_LOGIC;
8             rst : in STD_LOGIC;
9             clk : in STD_LOGIC;
10            ready : out STD_LOGIC;
11            inc : out STD_LOGIC;
12            strobe : out STD_LOGIC;
13            rst_dp : out STD_LOGIC);
14 end nodoA_CU;
15
16 architecture Behavioral of nodoA_CU is
17     type state_type is (
18         IDLE,
19         READ,
20         SEND,
21         INCR );
22     signal state : state_type := IDLE;
23     signal next_state : state_type;
24
25 begin
26
27     process(clk)
28 begin
29         if clk'event and clk = '0' then
30             if rst = '1' then
31                 state <= IDLE;
32             else
33                 state <= next_state;
34             end if;
35         end if;
36     end process;
37
```

```

38     process(state,start,ack,div)
39 begin
40     case state is
41         when IDLE =>
42             inc <= '0';
43             strobe <= '0';
44             ready <= '0';
45             rst_dp <= '1';
46             if start = '1' then
47                 next_state <= READ;
48             else
49                 next_state <= state;
50             end if;
51         when READ =>
52             inc <= '0';
53             strobe <= '1';
54             ready <= '0';
55             rst_dp <= '0';
56             if ack = '0' then
57                 next_state <= SEND;
58             else
59                 next_state <= state;
60             end if;
61         when SEND =>
62             inc <= '0';
63             strobe <= '0';
64             ready <= '1';
65             rst_dp <= '0';
66             if ack = '1' then
67                 next_state <= INCR;
68             else
69                 next_state <= state;
70             end if;
71         when INCR =>
72             inc <= '1';
73             strobe <= '0';
74             ready <= '0';
75             rst_dp <= '0';
76             if div = '1' then
77                 next_state <= IDLE;
78             else
79                 next_state <= READ;
80             end if;
81     end case;

```

```

82      end process;
83
84  end Behavioral;
```

Listing 7.4: Nodo A: unità di controllo

7.3.3 Nodo B

Anche in questo caso il nodo è realizzato secondo un paradigma strutturale, componendo tra di loro la parte operativa e la parte di controllo.

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity nodoB is
5      generic(
6          M : positive := 32;
7          K : positive := 3 );
8      Port ( clk : in STD_LOGIC;
9              rst : in STD_LOGIC;
10             mode : in STD_LOGIC;
11             ready : in STD_LOGIC;
12             data_in : in STD_LOGIC_VECTOR (M-1 downto 0);
13             addr : in STD_LOGIC_VECTOR (K-1 downto 0);
14             ack : out STD_LOGIC;
15             data_out : out std_logic_vector(M-1 downto 0) );
16 end nodoB;
17
18 architecture structural of nodoB is
19     signal increment : std_logic;
20     signal strobe : std_logic;
21     signal enable_reg : std_logic;
22     signal enable_mem : std_logic;
23     signal reset_dp : std_logic;
24     signal control_mode : std_logic;
25
26 begin
27     dp : entity work.nodoB_datapath(structural)
28         generic map (
29             M => M,
30             K => K )
31         port map(
32             addr => addr,
```

```

33      data_in => data_in,
34      rst_dp => reset_dp,
35      clk => clk,
36      inc => increment,
37      en_reg => enable_reg,
38      strobe => strobe,
39      ctrl_mode => control_mode,
40      en_mem => enable_mem,
41      data_out => data_out );
42
43  cu : entity work.nodoB_CU(Behavioral)
44    port map(
45      ready => ready,
46      mode => mode,
47      clk => clk,
48      rst => rst,
49      ack => ack,
50      en_mem => enable_mem,
51      en_reg => enable_reg,
52      strobe => strobe,
53      inc => increment,
54      rst_dp => reset_dp,
55      ctrl_mode => control_mode );
56
57 end structural;

```

Listing 7.5: Nodo B

Unità operativa

L’unità operativa è realizzata secondo un approccio strutturale, collegando tra di loro tutti i componenti presentati nel paragrafo 7.2.3, secondo lo schema in figura 7.7.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoB_datapath is
5   generic(
6     M : positive := 32;
7     K : positive := 3 );
8   Port ( addr : in STD_LOGIC_VECTOR (K-1 downto 0);
9         data_in : in STD_LOGIC_VECTOR (M-1 downto 0);

```

```

10      rst_dp : in STD_LOGIC;
11      clk : in STD_LOGIC;
12      inc : in STD_LOGIC;
13      en_reg : in STD_LOGIC;
14      ctrl_mode : in STD_LOGIC;
15      en_mem : in STD_LOGIC;
16      strobe : in STD_LOGIC;
17      data_out : out std_logic_vector(M-1 downto 0) );
18 end nodoB_datapath;
19
20 architecture structural of nodoB_datapath is
21
22     signal op1 : std_logic_vector (M-1 downto 0);
23     signal op2 : std_logic_vector (M-1 downto 0);
24     signal count : std_logic_vector (K-1 downto 0);
25     signal mem_addr : std_logic_vector (K-1 downto 0);
26     signal sum : std_logic_vector (M-1 downto 0);
27
28 begin
29     reg0 : entity work.register_pp(behavioral)
30         generic map(
31             dim => M )
32         port map(
33             x => data_in,
34             clk => clk,
35             rst => rst_dp,
36             enable => en_reg,
37             y => op2 );
38
39     counter0 : entity work.counter(behavioral)
40         generic map(
41             bits => K)
42         Port map(
43             x => inc,
44             rst => rst_dp,
45             y => count );
46
47     mux0 : entity work.mux_2_1(dataflow)
48         port map(
49             x0 => count,
50             x1 => addr,
51             sel => ctrl_mode,
52             y => mem_addr );
53

```

```

54     mem0 : entity work.memory(behavioral)
55         generic map(
56             nbits_addr => K,      -- numero di bit di
57             -- indirizzamento
58             width => M )       -- dimensione di una word di
59             -- memoria
60             Port map(
61                 addr => mem_addr,
62                 en => en_mem,
63                 clk => clk,
64                 rst => rst_dp,
65                 data_in => sum,
66                 r_w => ctrl_mode,           -- r_w= 1 => read ; r_w =
67                 -- 0 => write
68                 data_out => data_out );
69
70
71
72
73     adder0 : entity work.adder(behavioral)
74         port map(
75             a => op1,
76             b => op2,
77             sum => sum );
78
79
80
81 end structural;

```

Listing 7.6: Nodo B: unità operativa

Unità di controllo

L’unità di controllo è implementata secondo un approccio comportamentale con due process.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoB CU is

```

```

5      Port ( ready : in STD_LOGIC;
6          mode : in STD_LOGIC;
7          clk : in STD_LOGIC;
8          rst : in STD_LOGIC;
9          ack : out STD_LOGIC;
10         en_mem : out STD_LOGIC;
11         en_reg : out STD_LOGIC;
12         strobe : out STD_LOGIC;
13         rst_dp : out STD_LOGIC;
14         inc : out STD_LOGIC;
15         ctrl_mode : out std_logic);
16 end nodoB_CU;
17
18 architecture Behavioral of nodoB_CU is
19 type state_type is (
20     RESET,
21     IDLE,
22     READ_MEM,
23     RECEIVE,
24     SUM,
25     INCR );
26 signal state : state_type := RESET;
27 signal next_state : state_type;
28
29 subtype ctrl_word_type is std_logic_vector(6 downto 0);
30 signal ctrl_word : ctrl_word_type;
31
32 begin
33
34     inc <= ctrl_word(6);
35     strobe <= ctrl_word(5);
36     en_reg <= ctrl_word(4);
37     en_mem <= ctrl_word(3);
38     ack <= ctrl_word(2);
39     ctrl_mode <= ctrl_word(1);
40     rst_dp <= ctrl_word(0);
41
42 process(state,ready,mode)
43 begin
44     case state is
45         when RESET =>
46             ctrl_word <= "0000001";
47             if mode = '0' then
48                 next_state <= IDLE;

```

```

49         else
50             next_state <= READ_MEM;
51         end if;
52
53     when IDLE =>
54         ctrl_word <= "0000000";
55         if mode = '1' then
56             next_state <= READ_MEM;
57         elsif ready = '1' then
58             next_state <= RECEIVE;
59         else
60             next_state <= IDLE;
61         end if;
62
63     when RECEIVE =>
64         ctrl_word <= "0110000";
65         next_state <= SUM;
66
67     when SUM =>
68         ctrl_word <= "0001100";
69         if ready = '0' then
70             next_state <= INCR;
71         else
72             next_state <= SUM;
73         end if;
74
75     when INCR =>
76         ctrl_word <= "1000000";
77         next_state <= IDLE;
78
79     when READ_MEM =>
80         ctrl_word <= "0001010";
81         if mode = '0' then
82             next_state <= IDLE;
83         else
84             next_state <= READ_MEM;
85         end if;
86     end case;
87 end process;
88
89 process(clk)
90 begin
91     if clk'event and clk = '0' then
92         if rst = '1' then

```

```

93         state <= RESET;
94     else
95         state <= next_state;
96     end if;
97 end if;
98 end process;
99
100
101 end Behavioral;

```

Listing 7.7: Nodo B: unità di controllo

7.3.4 Sistema complessivo

Infine, il sistema complessivo si realizza collegando prima il nodo A con il nodo B (listato 7.8), e poi aggiungendovi la base dei tempi per fornire i due riferimenti temporali (listato 7.9).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity handshake_system is
5     generic(
6         M : positive := 32;
7         K : positive := 3 );
8     Port ( start : in STD_LOGIC;
9             clk_A : in STD_LOGIC;
10            clk_B : in STD_LOGIC;
11            rst : in STD_LOGIC;
12            addr : in STD_LOGIC_VECTOR (K-1 downto 0);
13            mode : in STD_LOGIC;
14            data_out : out std_logic_vector(M-1 downto 0) );
15 end handshake_system;
16
17 architecture structural of handshake_system is
18     component nodoA is
19         Generic (
20             M : positive;
21             K : positive );
22         Port ( start : in STD_LOGIC;
23                 ack : in STD_LOGIC;
24                 rst : in STD_LOGIC;
25                 clk : in STD_LOGIC;

```

```

26             ready : out STD_LOGIC;
27             data_out : out STD_LOGIC_VECTOR (M-1 downto 0));
28 end component;
29
30 component nodoB is
31     generic(
32         M : positive;
33         K : positive );
34     Port ( clk : in STD_LOGIC;
35            rst : in STD_LOGIC;
36            mode : in STD_LOGIC;
37            ready : in STD_LOGIC;
38            data_in : in STD_LOGIC_VECTOR (M-1 downto 0);
39            addr : in STD_LOGIC_VECTOR (K-1 downto 0);
40            ack : out STD_LOGIC;
41            data_out : out std_logic_vector(M-1 downto 0) );
42 end component;
43
44 signal ready : std_logic;
45 signal data : std_logic_vector (31 downto 0);
46 signal ack : std_logic;
47
48 begin
49
50     A : nodoA
51     generic map(
52         M => M,
53         K => K )
54     port map(
55         start => start,
56         ack => ack,
57         rst => rst,
58         clk => clk_A,
59         ready => ready,
60         data_out => data );
61
62     B : nodoB
63     generic map(
64         M => M,
65         K => K )
66     port map(
67         clk => clk_B,
68         rst => rst,
69         mode => mode,

```

```

70      ready => ready,
71      data_in => data,
72      addr => addr,
73      ack => ack,
74      data_out => data_out);
75
76 end structural;

```

Listing 7.8: Sistema di comunicazione tramite handshake

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity handshake_sys_bdt is
5   Generic(
6     M : positive := 32;
7     K : positive := 3;
8     clk_B_factor : positive := 4;
9     clk_B_delay : time := 4ns);
10  Port ( start : in STD_LOGIC;
11        addr : in STD_LOGIC_VECTOR (K-1 downto 0);
12        mode : in STD_LOGIC;
13        clk : in STD_LOGIC;
14        rst_sys : in STD_LOGIC;
15        rst_bdt : in STD_LOGIC;
16        data_out : out STD_LOGIC_VECTOR(M-1 downto 0) );
17 end handshake_sys_bdt;
18
19 architecture structural of handshake_sys_bdt is
20   signal clock_A : std_logic;
21   signal clock_B : std_logic;
22
23 begin
24
25   hs_sys : entity work.handshake_system
26   generic map(
27     M => M,
28     K => K )
29   port map(
30     start => start,
31     clk_A => clock_A,
32     clk_B => clock_B,
33     rst => rst_sys,

```

```
34         addr => addr,
35         mode => mode,
36         data_out => data_out );
37
38     bdt : entity work.base_dei_tempi
39     generic map(
40         clk_B_factor => clk_B_factor,
41         clk_B_delay => clk_B_delay)
42     port map(
43         clk => clk,
44         rst => rst_bdt,
45         clk_A => clock_A,
46         clk_B => clock_B);
47
48 end structural;
```

Listing 7.9: Sistema di comunicazione tramite handshake con base dei tempi

7.4 Simulazione

Per ogni componente analizzato in precedenza si riporta il codice del testbench utilizzato per testarlo, e le waveform risultanti dalla simulazione. Le unità operative del nodo A e del nodo B non sono state testate in autonomia, ma direttamente collegate alle rispettive unità di controllo.

7.4.1 Base dei tempi

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity base_dei_tempi_tb is
5 -- Port ( );
6 end base_dei_tempi_tb;
7
8 architecture Behavioral of base_dei_tempi_tb is
9     signal clock : std_logic;
10    signal reset : std_logic;
11    signal clock_A : std_logic;
12    signal clock_B : std_logic;
13
14 begin
15     dut : entity work.base_dei_tempi(structural)
16         port map(
17             clk => clock,
18             rst => reset,
19             clk_A => clock_A,
20             clk_B => clock_B );
21
22     clk_gen : process
23     begin
24         clock <= '0';
25         wait for 5ns;
26         clock <= '1';
27         wait for 5ns;
28     end process;
29
30     process
31     begin
32         reset <= '1';
33         wait for 25ns;
34         reset <= '0';
35     end process;
```

```

35      wait for 200ns;
36  end process;
37
38 end Behavioral;
```

Listing 7.10: Base dei tempi: testbench

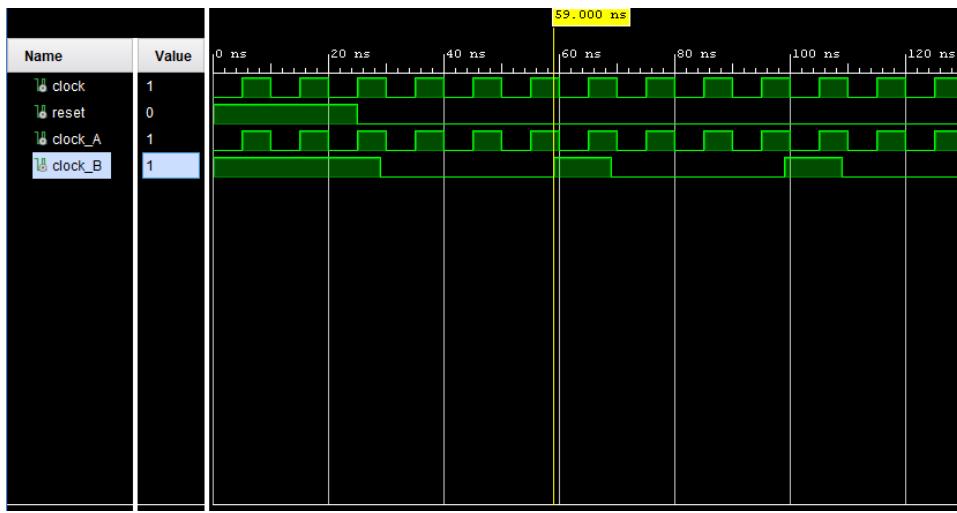


Figura 7.9: Base dei tempi: waveform

7.4.2 Nodo A

Unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA CU_tb is
5   -- Port ( );
6 end nodoA CU_tb;
7
8 architecture Behavioral of nodoA CU_tb is
9   component nodoA CU is
10     Port ( start : in STD_LOGIC;
11            ack : in STD_LOGIC;
12            div : in STD_LOGIC;
13            rst : in STD_LOGIC;
14            clk : in STD_LOGIC;
15            ready : out STD_LOGIC);
```

```

16          inc : out STD_LOGIC;
17          strobe : out STD_LOGIC;
18          rst_dp : out STD_LOGIC);
19 end component;
20
21 signal start : std_logic;
22 signal ack : std_logic;
23 signal div : std_logic;
24 signal rst : std_logic;
25 signal clk : std_logic;
26
27 signal ready : std_logic;
28 signal inc : std_logic;
29 signal strobe : std_logic;
30 signal rst_dp : std_logic;
31
32 shared variable count : natural := 0;
33
34 begin
35     dut : nodoA_CU port map(
36         start => start,
37         ack => ack,
38         div => div,
39         rst => rst,
40         clk => clk,
41         ready => ready,
42         inc => inc,
43         strobe => strobe,
44         rst_dp => rst_dp );
45
46     clk_gen : process
47 begin
48     clk <= '0';
49     wait for 5ns;
50     clk <= '1';
51     wait for 5ns;
52 end process;
53
54 process
55
56 begin
57     count := 0;
58     start <= '0';
59     ack <= '0';

```

```

60         rst <= '0';
61         div <= '1';
62         wait for 15ns;
63
64         rst <= '1';
65         wait for 15ns;
66         assert (inc = '0' and strobe = '0' and ready = '0' and
67             →  rst_dp = '1') report "Errore IDLE" severity
68             →  failure;
69
70         rst <= '0';
71         start <= '1';
72         wait for 15ns;
73         assert (inc = '0' and strobe = '1' and ready = '0' and
74             →  rst_dp = '0') report "Errore READ" severity
75             →  failure;
76         start <= '0';
77
78         while count < 7 loop
79             ack <= '0';
80             wait until ready = '1';
81             ack <= '1';
82             count := count + 1;
83             wait for 15ns;
84             assert (inc = '1' and strobe = '0' and ready = '0'
85                 →  and rst_dp = '0') report "Errore INCR" severity
86                 →  failure;
87             div <= '0';
88             end loop;
89
90         end process;
91
92     end Behavioral;

```

Listing 7.11: Nodo A: testbench dell'unità di controllo

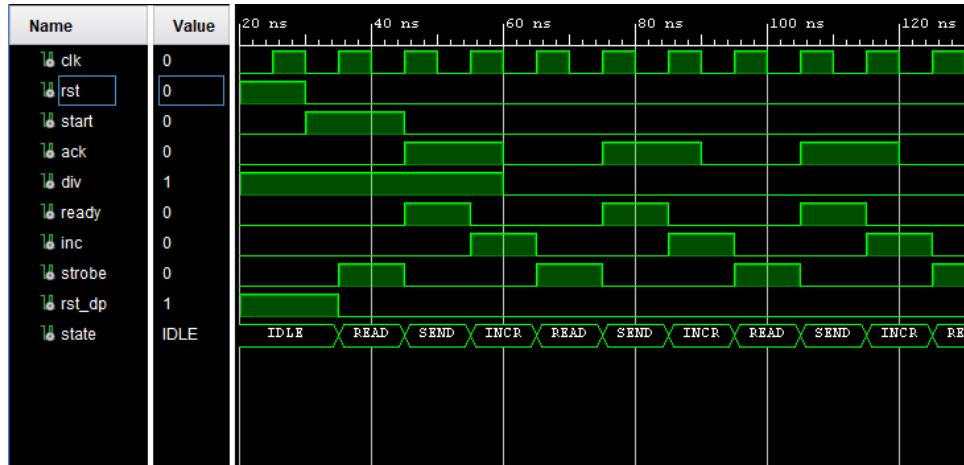


Figura 7.10: Nodo A: waveform dell'unità di controllo

Nodo complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA_tb is
5   -- Port ( );
6 end nodoA_tb;
7
8 architecture Behavioral of nodoA_tb is
9   constant width : positive := 32;
10  constant nbit_addr : positive := 3;
11  constant depth : positive := 2**nbit_addr;
12
13 component nodoA is
14   Generic(
15     M : positive := width;
16     K : positive := nbit_addr );
17   Port ( start : in STD_LOGIC;
18         ack : in STD_LOGIC;
19         rst : in STD_LOGIC;
20         clk : in STD_LOGIC;
21         ready : out STD_LOGIC;
22         data_out : out STD_LOGIC_VECTOR (width-1 downto
23           0));

```

```

23 end component;
24
25 signal start : std_logic;
26 signal ack : std_logic := '0';
27 signal rst : std_logic;
28 signal clk : std_logic;
29
30 signal ready : std_logic;
31 signal data_out : std_logic_vector (width-1 downto 0);
32
33 shared variable count : integer := 0;
34
35 subtype word is std_logic_vector (width-1 downto 0);
36 type rom_type is array(0 to depth-1) of word;
37 constant oracle : rom_type := (
38     x"00010002",
39     x"00030004",
40     x"00050006",
41     x"00070008",
42     x"0009000A",
43     x"000B000C",
44     x"000D000E",
45     x"000F0010"
46 );
47
48 begin
49     dut : nodoA port map(
50         start => start,
51         ack => ack,
52         rst => rst,
53         clk => clk,
54         ready => ready,
55         data_out => data_out );
56
57     clk_gen : process
58 begin
59         clk <= '0';
60         wait for 5ns;
61         clk <= '1';
62         wait for 5ns;
63     end process;
64
65     process
66 begin

```

```

67         rst <= '1';
68         start <= '0';
69         wait for 15ns;
70
71         rst <= '0';
72         wait for 15ns;
73
74         start <= '1';
75         wait for 15ns;
76         start <= '0';
77
78
79         wait for 255ns;
80         start <= '1';
81         wait for 15ns;
82         start <= '0';
83
84         wait for 285ns;
85     end process;
86
87
88     process
89     begin
90         wait until ready'event and ready = '1';
91         assert data_out <= oracle(count) report "error"
92             severity failure;
93         count := (count + 1)mod 8;
94         wait for 5ns;
95         ack <= '1';
96
97         wait until ready'event and ready = '0';
98         wait for 5ns;
99         ack <= '0';
100
101    end process;
102
103 end Behavioral;

```

Listing 7.12: Nodo A: testbench

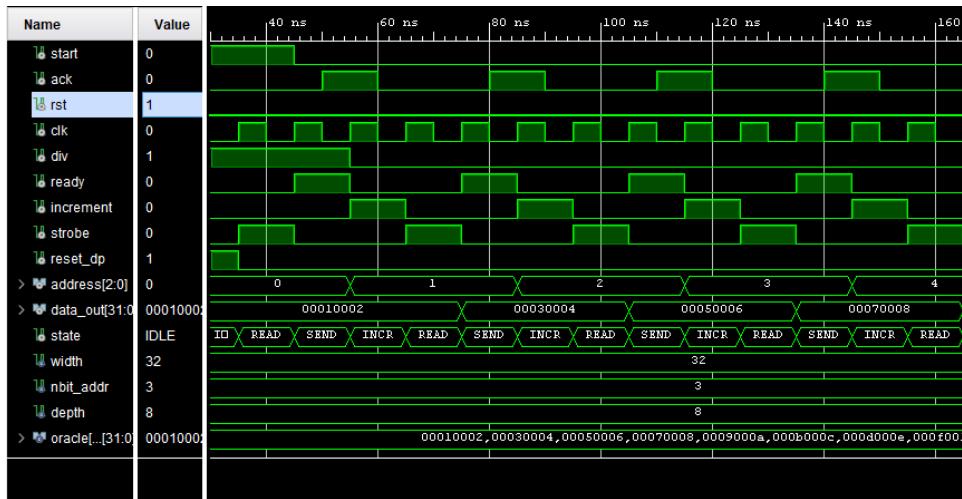


Figura 7.11: Nodo A: waveform

7.4.3 Nodo B

Unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoB CU_tb is
5   -- Port ( );
6 end nodoB CU_tb;
7
8 architecture Behavioral of nodoB CU_tb is
9   constant clk_period : time := 10ns;
10
11   signal ready : std_logic;
12   signal mode : std_logic;
13   signal clk : std_logic;
14   signal rst : std_logic;
15   signal ack : std_logic;
16   signal en_mem : std_logic;
17   signal en_reg : std_logic;
18   signal strobe : std_logic;
19   signal rst_dp : std_logic;
20   signal inc : std_logic;
21   signal ctrl_mode : std_logic;
22
23   signal ctrl_word : std_logic_vector(6 downto 0);

```

```

24
25 begin
26   dut : entity work.nodoB_CU(Behavioral)
27   port map(
28     ready => ready,
29     mode => mode,
30     clk => clk,
31     rst => rst,
32     ack => ack,
33     en_mem => en_mem,
34     en_reg => en_reg,
35     strobe => strobe,
36     rst_dp => rst_dp,
37     inc => inc,
38     ctrl_mode => ctrl_mode );
39
40   ctrl_word <= inc & strobe & en_reg & en_mem & ack &
41   ↪ ctrl_mode & rst_dp;
42
43   clk_gen : process
44 begin
45   clk <= '0';
46   wait for clk_period/2;
47   clk <= '1';
48   wait for clk_period/2;
49 end process;
50
51   test : process
52 begin
53   ready <= '0';
54   mode <= '0';
55   rst <= '1';
56   wait for 22ns;
57   assert ctrl_word = "0000001" report "Errore RESET"
58   ↪ severity failure;
59
60   rst <= '0';
61   wait for 12ns;
62   assert ctrl_word = "0000000" report "Errore IDLE"
63   ↪ severity failure;
64
65   ready <= '1';
66   wait for clk_period;

```

```

64      assert ctrl_word = "0110000" report "Errore RECEIVE"
65          -- severity failure;
66
67      wait until ctrl_word(2) = '1'; -- ack = '1'
68      assert ctrl_word = "0001100" report "Errore SUM"
69          -- severity failure;
70      wait for 22ns;
71
72      ready <= '0';
73      wait for clk_period;
74      assert ctrl_word = "1000000" report "Errore INCR"
75          -- severity failure;
76
77      mode <= '1';
78      wait for 2*clk_period;
79      assert ctrl_word = "0001010" report "Errore READ_MEM"
80          -- severity failure;
81      wait for 22ns;
82
83      rst <= '1';
84      wait for clk_period;
85      assert ctrl_word = "0000001" report "Errore RESET"
86          -- severity failure;
87
88      wait for 30ns;
89  end process;
90
91
92 end Behavioral;

```

Listing 7.13: Nodo B: testbench dell'unità di controllo

Nodo complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.std_logic_unsigned.all;
4 use IEEE.NUMERIC_STD.ALL;
5
6 entity nodoB_tb is
7     -- Port ( );
8 end nodoB_tb;
9

```



Figura 7.12: Nodo B: waveform dell'unità di controllo

```

10  architecture Behavioral of nodoB_tb is
11      constant clk_period : time := 10ns;
12      constant M : positive := 32;
13      constant K : positive := 3;
14
15      signal clk : std_logic;
16      signal rst : std_logic;
17      signal mode : std_logic;
18      signal ready : std_logic;
19      signal data_in : std_logic_vector (M-1 downto 0);
20      signal addr : std_logic_vector (K-1 downto 0);
21      signal ack : std_logic;
22      signal data_out : std_logic_vector(M-1 downto 0);
23
24      constant depth : positive := 2**K;
25      subtype word is std_logic_vector (M-1 downto 0);
26      type rom_type is array(0 to depth-1) of word;
27      constant rom : rom_type := (
28          x"00010002",
29          x"00030004",
30          x"00050006",
31          x"00070008",
32          x"0009000A",
33          x"000B000C",
34          x"000D000E",
35          x"000F0010"
36      );

```

```

37
38 begin
39     dut : entity work.nodoB
40         port map(
41             clk => clk,
42             rst => rst,
43             mode => mode,
44             ready => ready,
45             data_in => data_in,
46             addr => addr,
47             ack => ack,
48             data_out => data_out);
49
50     clk_gen : process
51 begin
52     clk <= '0';
53     wait for clk_period/2;
54     clk <= '1';
55     wait for clk_period/2;
56 end process;
57
58 process
59 begin
60     rst <= '1';
61     mode <= '0';
62     ready <= '0';
63     data_in <= (others => '0');
64     addr <= (others => '0');
65     wait for 2*clk_period;
66
67     rst <= '0';
68     wait for clk_period;
69
70     for i in 0 to 7 loop
71         ready <= '1';
72         wait until ack = '1';
73         wait for 2*clk_period;
74         ready <= '0';
75         wait until ack = '0';
76     end loop;
77
78     mode <= '1';
79     wait for 2*clk_period;
80     for i in 0 to 7 loop

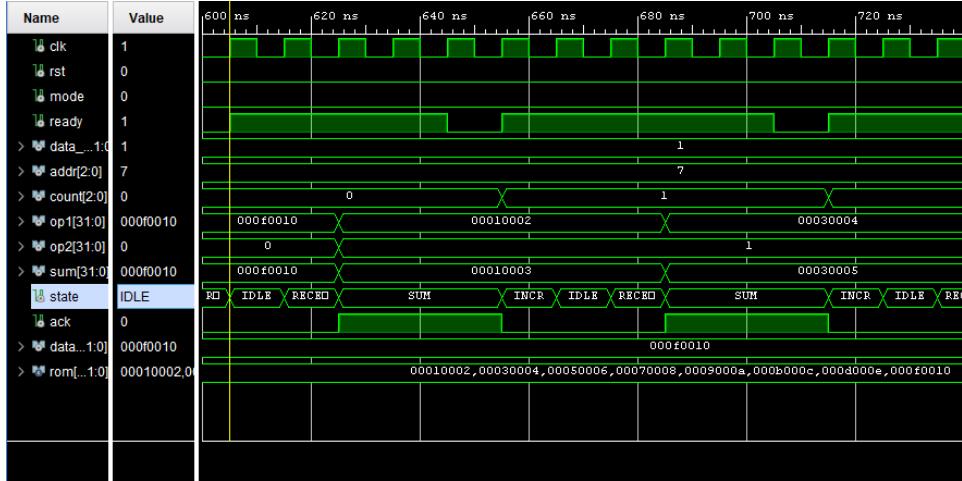
```

```

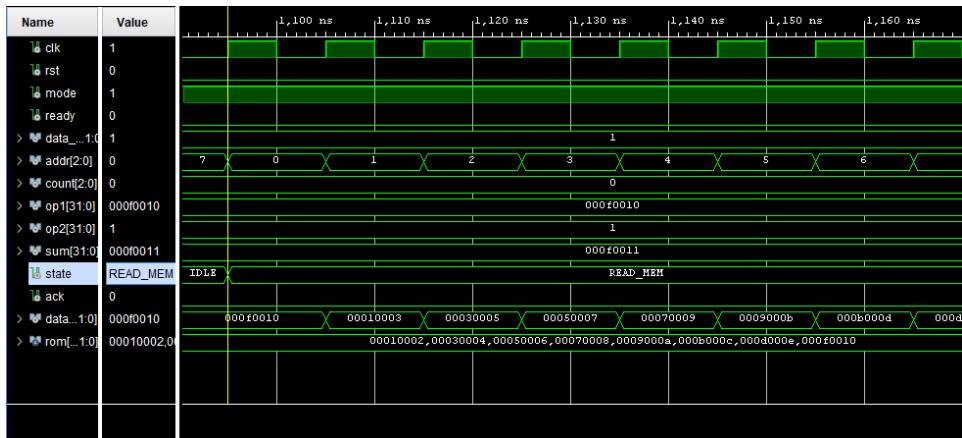
81         addr <=
82             → std_logic_vector(to_unsigned(i,addr'length));
83         wait on data_out;
84         assert data_out = rom(i) report "Errore" severity
85             → failure;
86     end loop;
87
88     mode <= '0';
89     data_in <= x"00000001";
90     wait for clk_period;
91
92     for i in 0 to 7 loop
93         ready <= '1';
94         wait until ack = '1';
95         wait for 2*clk_period;
96         ready <= '0';
97         wait until ack = '0';
98     end loop;
99
100    mode <= '1';
101    wait for 2*clk_period;
102    for i in 0 to 7 loop
103        addr <=
104            → std_logic_vector(to_unsigned(i,addr'length));
105        wait on data_out;
106        assert data_out = rom(i) + "1"  report "Errore"
107            → severity failure;
108    end loop;
109
110    rst <= '1';
111    wait for 2*clk_period;
112    rst <= '0';
113    wait for clk_period;
114    for i in 0 to 7 loop
115        addr <=
116            → std_logic_vector(to_unsigned(i,addr'length));
117    end loop;
118 end process;
119
120 end Behavioral;

```

Listing 7.14: Nodo B: testbench



(a) Modalità 0



(b) Modalità 1

Figura 7.13: Nodo B: waveform

7.4.4 Sistema complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use ieee.numeric_std.all;
4 use ieee.std_logic_unsigned.all;
5
6 entity handshake_sys_bdt_tb is
7
8 end handshake_sys_bdt_tb;

```

```

9
10 architecture behavioral of handshake_sys_bdt_tb is
11     constant M : positive := 32;
12     constant K : positive := 3;
13     constant clk_B_factor : positive := 2;
14     constant clk_B_delay : time := 4ns;
15     constant clkA_period : time := 10ns;
16     constant clkB_period : time := clk_B_factor * clkA_period;
17
18     signal start : std_logic;
19     signal addr : std_logic_vector(K-1 downto 0);
20     signal mode : STD_LOGIC;
21     signal clk : STD_LOGIC := '1';
22     signal rst_sys : STD_LOGIC;
23     signal rst_bdt : STD_LOGIC;
24     signal data_out : STD_LOGIC_VECTOR(M-1 downto 0);
25
26     constant depth : positive := 2**K;
27     subtype word is std_logic_vector (M - 1 downto 0);
28     type rom_type is array(0 to depth - 1) of word;
29     constant rom : rom_type := (
30         x"00010002",
31         x"00030004",
32         x"00050006",
33         x"00070008",
34         x"0009000A",
35         x"000B000C",
36         x"000D000E",
37         x"000F0010"
38     );
39
40 begin
41
42     dut: entity work.handshake_sys_bdt
43     Generic map (
44         M => M,
45         K => K,
46         clk_B_factor => clk_B_factor,
47         clk_B_delay => clk_B_delay)
48     Port map (
49         start => start,
50         addr => addr,
51         mode => mode,
52         clk => clk,

```

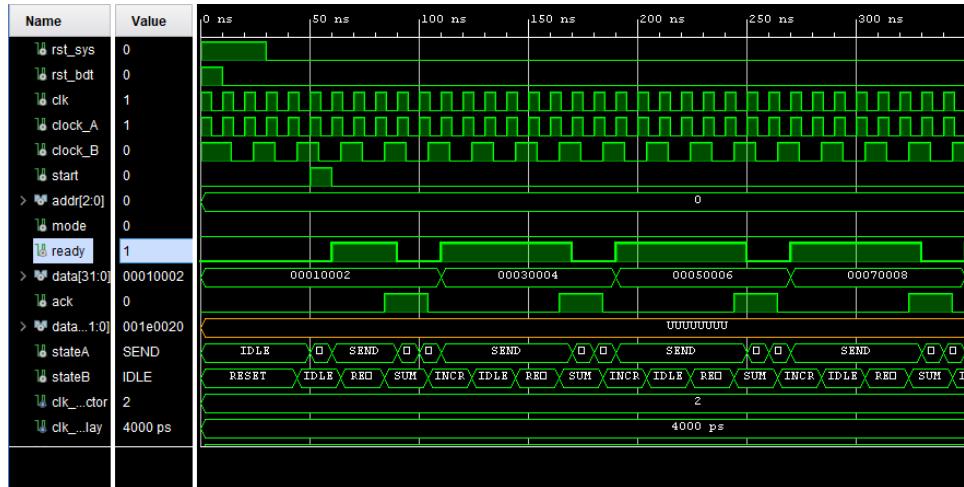
```

53         rst_sys => rst_sys,
54         rst_bdt => rst_bdt,
55         data_out => data_out );
56
57     clk_gen : process
58     begin
59         wait for clkA_period/2;
60         clk <= not(clk);
61     end process;
62
63     test: process
64     begin
65         start <= '0';
66         mode <= '0';
67         addr <= (others => '0');
68         rst_sys <= '1';
69         rst_bdt <= '1';
70         wait for clkA_period;
71
72         rst_bdt <= '0';
73         wait for clkB_period;
74
75         rst_sys <= '0';
76         wait for clkB_period;
77
78         start <= '1';
79         wait for clkA_period;
80         start <= '0';
81
82         wait for 630ns;
83         mode <= '1';
84         for i in 0 to 7 loop
85             addr <=
86                 std_logic_vector(to_unsigned(i,addr'length));
87             wait until data_out'active;
88             assert data_out = rom(i) + rom(i) report "Errore"
89                 severity failure;
90             end loop;
91
92         mode <= '0';
93         wait for clkB_period;
94     end process;

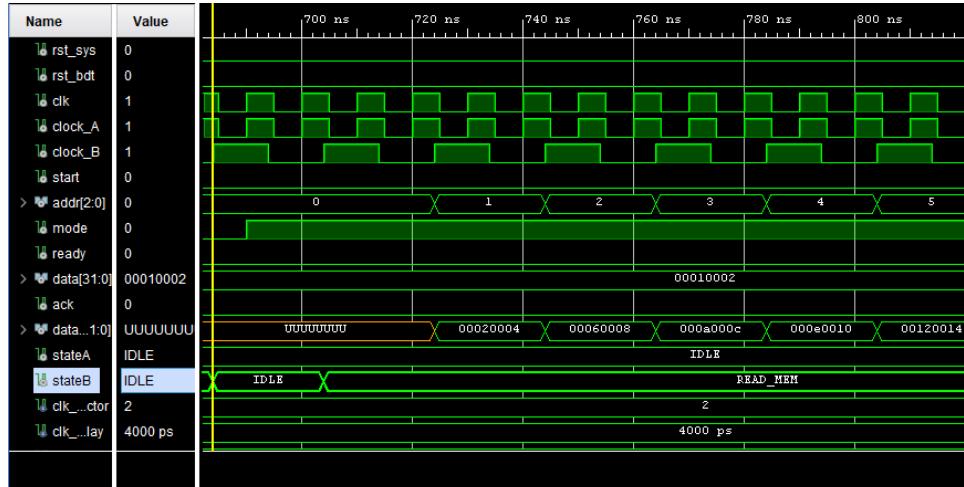
```

```
94 end behavioral;
```

Listing 7.15: Sistema complessivo: testbench



(a) Modalità 0



(b) Modalità 1

Figura 7.14: Sistema complessivo: waveform

Capitolo 8

Processore Mic-1

8.1 Traccia

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

Esercizio 8.1 si proceda all'analisi dell'architettura mediante simulazione e si approfondisca lo studio del suo funzionamento per due istruzioni a scelta;

Esercizio 8.2 si modifichi un codice operativo a scelta, documentando tutte le modifiche effettuate;

Esercizio 8.3 (opzionale) si descriva il funzionamento del processore in merito alle istruzioni di input/output;

Esercizio 8.4 (solo ove possibile) si sintetizzi il processore su FPGA.

8.2 La struttura del processore

Il Mic-1 è un processore la cui architettura è stata ideata da Tanenbaum ed è pensato per l'esecuzione di bytecode Java. L'ISA (Instruction Set Architecture) di questo processore prevede istruzioni a lunghezza variabile. Il Mic-1 possiede un'architettura a stack, dunque il suo modello di programmazione non prevede il controllo diretto dell'utente sui registri interni del processore, come nei modelli a registri generali, ma consente all'utente di programmare la macchina sfruttando una particolare area di memoria chiamata *stack* su cui memorizzare i dati necessari per l'elaborazione. Tale modello di programmazione consente l'utilizzo di istruzioni macchina con operandi impliciti, dato che i dati da elaborare saranno sempre quelli presenti sulla cima dello stack.

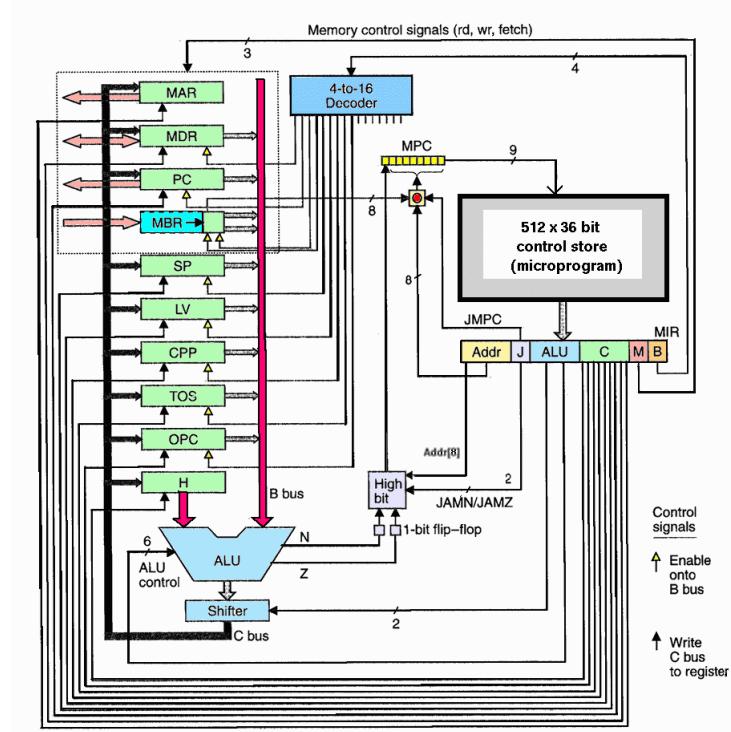


Figura 8.1: Architettura del processore Mic-1

Anche se l'utente non ha accesso ai registri interni del processore, non vuol dire che non ve ne siano, come si vede dalla figura 8.1. Tali registri non sono visibili a livello ISA, ma lo sono a livello micro-architetturale.

8.2.1 Il datapath

Il datapath del Mic-1 è composto da una serie di componenti:

- undici registri a 32 bit;

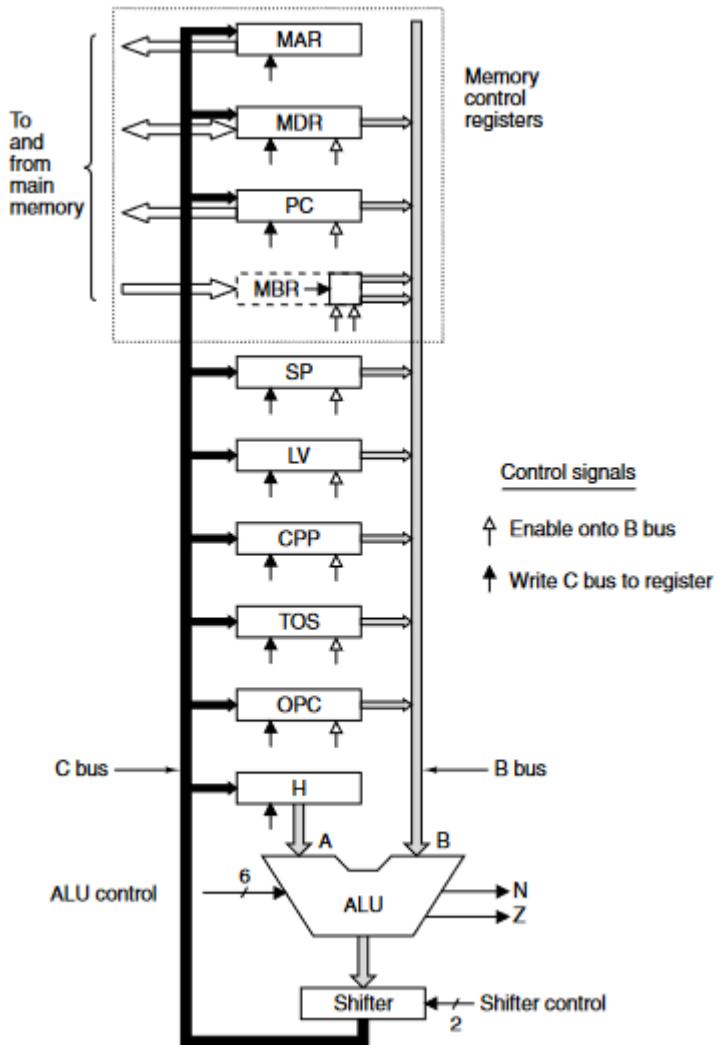


Figura 8.2: Datapath del processore Mic-1

- una ALU;
- tre bus a 32 bit.

Registri

Come registri abbiamo:

1. **MAR (Memory Address Register)**, che punta alla word in memoria da cui leggere/scrivere il dato in MDR.
2. **MDR (Memory Data Register)**, che contiene la word letta dalla memoria, o da scrivere in memoria.

3. **PC (Program Counter)**, che punta ad un byte in memoria contenente la prossima istruzione da eseguire.
4. **MBR (Memory Buffer Register)**, che contiene il byte letto dalla memoria all'indirizzo del Program Counter.
5. **SP (Stack Pointer)**, punta alla cima dello stack, ovvero l'ultima locazione occupata dello stack.
6. **LV (Local Variable)**, contiene l'indirizzo sullo stack della prima variabile locale dello stack frame attuale. Serve per effettuare l'indirizzamento relativo dei dati in memoria, che è l'unico tipo di indirizzamento supportato dal Mic-1.
7. **CPP (Constant Pool Pointer)**, punta all'area di memoria in cui sono memorizzate le costanti del processo attualmente in esecuzione.
8. **TOS (Top Of Stack)**, contiene il valore della word in cima allo stack, ovvero della locazione di memoria puntata da SP. Tale proprietà è garantita solo all'inizio e alla fine di ogni micro-procedura.
9. **OPC (Old Program Counter)**, è un registro libero, senza una funzione specifica. Deve il suo nome all'utilizzo che ne viene fatto nelle istruzioni di salto per memorizzare il valore del vecchio Program Counter.
10. **H (Holding)**, serve per memorizzare l'operando A della ALU.
11. **Shifter**, è un registro tampone che memorizza l'output della ALU, e permette di effettuare uno shift logico verso sinistra di un byte, oppure uno shift aritmetico verso destra di un bit.

ALU

La ALU del processore presenta un'interfaccia composta da:

- due ingressi per gli operandi a 32 bit;
- un'uscita per il risultato a 32 bit;
- sei segnali in ingresso di controllo;
- due segnali di stato in uscita.

I due segnali di stato sono N e Z, ed indicano rispettivamente il risultato dell'ultima operazione eseguita abbia restituito un risultato negativo, o pari a zero. I sei segnali di controllo consentono un massimo di $2^6 = 64$ operazioni differenti. Non tutte le combinazioni sono utilizzate, come si vede nella tabella 8.1.

Tabella 8.1: Operazioni ALU

F_0	F_1	EN_A	EN_B	INV_A	INC	Function
0	1	1	0	0	0	A
0	1	0	1	0	0	B
0	1	1	0	1	0	\bar{A}
1	0	1	1	0	0	\bar{B}
1	1	1	1	0	0	$A + B$
1	1	1	1	0	1	$A + B + 1$
1	1	1	0	0	1	$A + 1$
1	1	0	1	0	1	$B + 1$
1	1	1	1	1	1	$B - A$
1	1	0	1	1	0	$B - 1$
1	1	1	0	1	1	$-A$
0	0	1	1	0	0	$A \text{ and } B$
0	1	1	1	0	0	$A \text{ or } B$
0	1	0	0	0	0	0
1	1	0	0	0	1	1
1	1	0	0	1	0	-1

Per poter eseguire operazioni a due operandi occorre pre-caricare il registro H con il valore richiesto, facendo passare tale dato inalterato attraverso la ALU (vedesi riga 1 e 2 della tabella 8.1).

Bus

Il datapath del Mic-1 prevede tre bus a 32 bit per la movimentazione dei dati:

- **Bus A:** bus di sola lettura che collega il registro H all'ingresso A della ALU; non richiede segnali di controllo.
- **Bus B:** bus di sola lettura che collega i restanti registri all'ingresso B della ALU. La scrittura sul bus B avviene in mutua-esclusione e sono richiesti 9 segnali di controllo per regolare l'accesso al bus (2 solo per il registro MBR).
- **Bus C:** bus di sola scrittura che permette di scrivere il valore memorizzato in *Shifter* in uno o più registri contemporaneamente; sono necessari 9 segnali di controllo.

Il registro MBR contiene un byte letto dalla memoria. Per poter scrivere tale byte sul bus B occorre estendere tale valore su 32 bit. Esistono due modi diversi per estendere tale valore:

- *senza segno:* si prepongono 3 byte nulli al byte contenuto in MBR;

- *con segno*: si replica l'MSb (Most Significant bit) di MBR 24 volte.

Per indicare quale estensione effettuare, nel momento in cui si voglia scrivere tale valore sul bus B, sono necessari due diversi segnali di controllo per MBR.

Interfacciamento con la memoria

Il Mic-1 si interfaccia con la memoria principale attraverso due porti distinti in grado di operare in parallelo:

- il porto costituito dai registri MAR/MDR che serve per la movimentazione dei dati. È un porto di lettura/scrittura, che lavora con un parallelismo a 32 bit.
- il porto costituito dai registri PC/MBR che serve per la *fetch* delle istruzioni. È un porto di sola lettura che lavora con un parallelismo a 8 bit.

Dall’interfaccia si evince che l’area dati e l’area istruzioni sono separate all’interno della memoria. Tanenbaum ha progettato tale architettura sotto l’assunzione di lavorare con una *cache perfetta*, ovvero che non si verifichino mai dei *cache-miss*.

Il porto MAR/MDR lavora indirizzando delle word in memoria, dunque gli indirizzi presenti in MAR sono gli indirizzi logici delle word. Tuttavia la memoria principale lavora con un indirizzamento a byte, dunque c’è bisogno di un meccanismo di adattamento. Tale meccanismo shifta verso sinistra l’indirizzo contenuto in MAR di due bit, rimuovendo i due bit più significativi e moltiplicando il valore di MAR per un fattore 4 (vedesi figura 8.3).

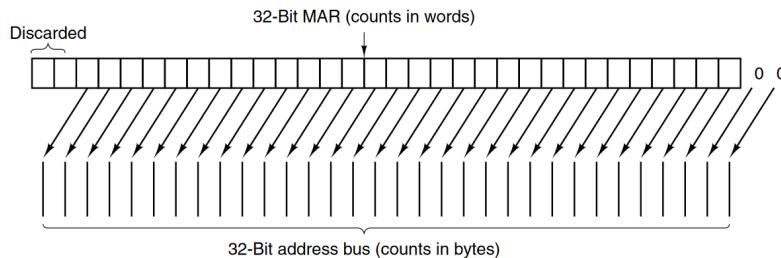


Figura 8.3: Adattamento dell’indirizzamento di MAR da word a byte

8.2.2 L’unità di controllo

Il formato delle control word

Prima di analizzare la struttura dell’unità di controllo del Mic-1, individuiamo i segnali che occorrono per pilotare il datapath. Abbiamo:

- 6 segnali per selezionare l'operazione desiderata dalla ALU, più 2 segnali per selezionare le due modalità di shift del registro Shifter (SLL8 o SRA1);
- 9 segnali per controllare il bus C in scrittura;
- 9 segnali per controllare il bus B in lettura, che diventano 4 andando a codificare le 9 configurazioni possibili su soli 4 bit;
- 3 segnali per gestire le operazioni di memoria, ovvero lettura/scrittura dei dati, e fetch delle istruzioni.

Per codificare i segnali di controllo del bus B, sfruttiamo la proprietà che la scrittura sul bus B avviene in mutua esclusione, dunque dei nove bit di controllo uno solo può essere alto per ogni periodo del clock. Occorre, quindi, un totale di 24 segnali per controllare il datapath del processore, dunque le control word dell'unità di controllo devono essere lunghe almeno 24 bit. Osserviamo il formato di una control word del processore Mic-1 in figura 8.4.

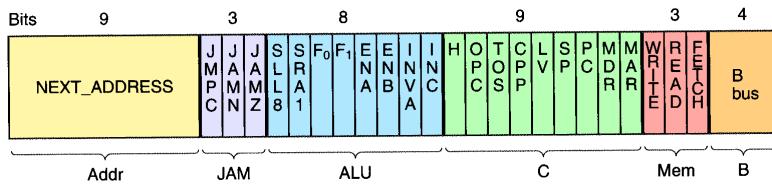


Figura 8.4: Formato delle control word del Mic-1

Tale stringa di bit contiene tutti i 24 bit elencati in precedenza, più altri 12 che illustreremo successivamente, per un totale di 36 bit.

L'unità di controllo

Il processore Mic-1 presenta un'unità di controllo implementata in logica microprogrammata, ovvero le control word sono memorizzate in una micro-ROM e sono organizzate in unità logiche dette *microprocedure*, ognuna delle quali implementa un'istruzione macchina IJVM. In questo modello le control word sono anche chiamate *microistruzioni*. Osserviamo la struttura dell'unità di controllo in figura 8.5.

L'unità di controllo del Mic-1 è composta da:

- una micro-ROM 512x36 bit, che memorizza tutte le microprocedure necessarie;
- il registro **MPC** (Micro Program Counter) che contiene l'indirizzo della microistruzione successiva all'interno della micro-ROM;
- il registro **MIR** (Micro Instruction Register) che contiene la control word della microistruzione attuale;

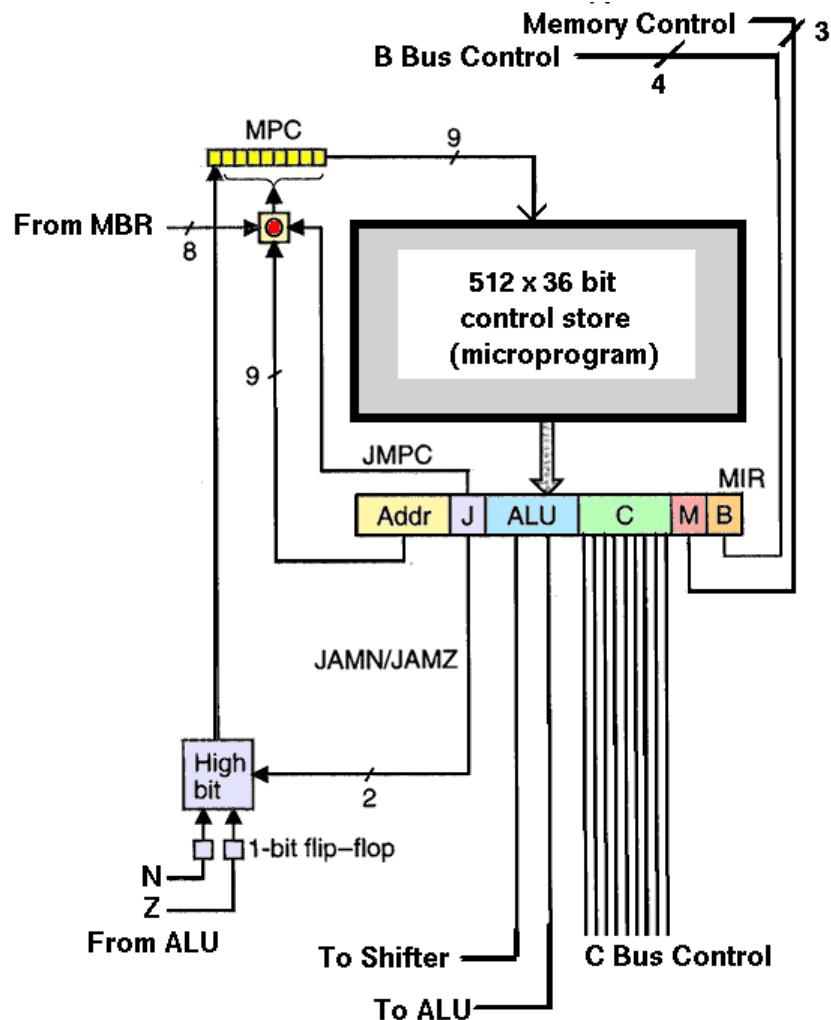


Figura 8.5: Unità di controllo del processore Mic-1

- una logica combinatoria per la gestione dei salti delle microprocedure;
- un decoder 4:16 per decodificare i segnali di controllo del bus B.

Per poter ridurre la dimensione della micro-ROM a soli 512 microistruzioni, è importante riutilizzare parti in comune a più microprocedure, piuttosto che replicarle ogni volta. Per semplificare tale meccanismo, le microistruzioni non sono memorizzate in ordine lineare in memoria, ma ogni microistruzione chiama la sua successiva. È per questo motivo che ogni control word contiene l'indirizzo della microistruzione prossima, che è composto da 9 bit dato che la micro-ROM contiene 512 locazioni.

Per quanto riguarda il formato delle control word, non ci resta che spiegare il funzionamento dei 3 bit di salto: JMPC, JAMN e JAMZ. Iniziamo con

gli ultimi due. JAMN e JAMZ servono per implementare la logica di salto nelle microprocedure in relazione al risultato dell'operazione precedente. In particolare:

- JAMN indica di saltare se il bit N della ALU è asserito;
- JAMZ indica di saltare se il bit Z della ALU è asserito.

Ma come viene implementato il salto? Questo meccanismo consente unicamente di saltare di 256 locazioni all'interno della micro-ROM, controllando l'MSb del registro MPC. Precisamente, il bit 8 del registro MPC è determinato da una logica del tipo:

$$\text{MPC}[8] = (\text{JAMN and N}) \text{ or } (\text{JAMZ and Z}) \text{ or } \text{ADDR}[8]$$

Il bit di salto JMPC, invece, serve ad implementare una logica di salto incondizionata; in particolare serve a caricare in MPC[7:0] un byte a scelta. Tale funzionalità è utile, ad esempio, all'inizio di ogni microprocedura, quando occorre caricare in MPC l'indirizzo della prima microistruzione da eseguire. Per eseguire tale operazione nel Mic-1 esiste una microprocedura speciale chiamate *main*, che esegue la fetch del codice operativo della nuova istruzione macchina e lo carica in MPC.¹ Ciò avviene perché l'ISA del processore Mic-1 è progettato in modo tale che ***il codice operativo di ogni istruzione è codificato con l'indirizzo della prima microistruzione della microprocedura che implementa tale istruzione***. Dunque JMPC consente di caricare negli 8 bit meno significativi di MPC il contenuto di MBR secondo una logica del tipo:

$$\text{MPC}[7:0] = (\text{MBR}[7:0] \text{ and JMPC}) \text{ or } \text{ADDR}[7:0]$$

dove per "*MBR[7:0] and JMPC*" si intende otto operazioni di *and* tra ogni bit di MBR e il bit JMPC.

8.2.3 Protocollo con la memoria

All'interno del processore, la comunicazione con la memoria è controllata attraverso tre segnali di controllo:

- READ
- WRITE
- FETCH

Tali segnali controllano i porti interni al processore MAR/MDR e PC/MBR, ma il protocollo tra memoria e processore è realizzato mediante un solo segnale di controllo che è quello di **Write Enable (WE)**. In particolare,

¹Non è precisamente così, ma chiariremo tale dettaglio in seguito.

- $WE = 1$: il contenuto di MDR è scritto in memoria all'indirizzo contenuto in MAR;
- $WE = 0$: in MDR è scritta la word di memoria all'indirizzo $MAR \ll 2$, e in MBR è scritto il byte di memoria contenuto all'indirizzo in PC.

Tuttavia, come si vede in figura 8.6, affinché il protocollo con la memoria sia completo occorre attendere due cicli di clock. Ciò significa che, per le operazioni di lettura dalla memoria, è necessario attendere due colpi di clock prima di poter utilizzare il valore richiesto.

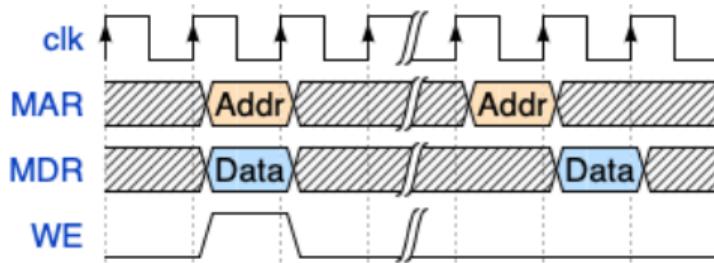
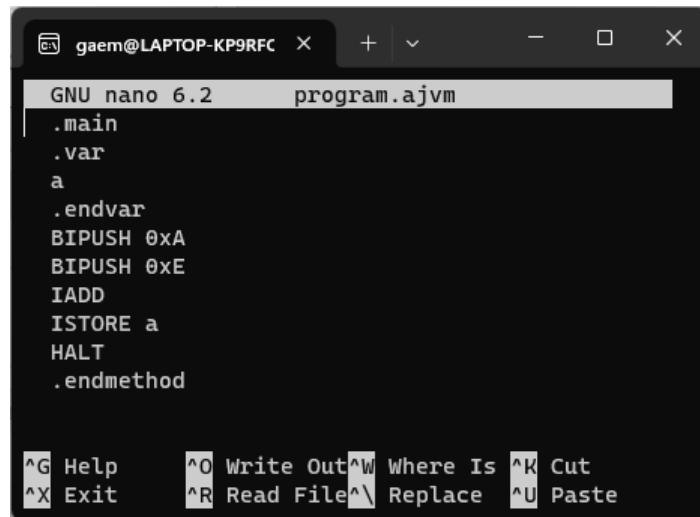


Figura 8.6: Protocollo con la memoia del processore Mic-1

Tale ritardo influenza il funzionamento dell'intero processore, in particolare l'operazione di fetch che richiede continue letture in memoria. Per risolvere tale problema si adotta la strategia di anticipare l'operazione di fetch, pre-caricando in MBR il valore successivo. Per tale motivo, la microprocedura di *main* descritta al paragrafo precedente in realtà non effettua la fetch della prossima istruzione, che è già stata effettuata in precedenza, ma effettua l'operazione di fetch successiva.

8.3 Simulazione

Testiamo il funzionamento del progetto VHDL del processore eseguendo un semplice programma IJVM in simulazione, ed analizzando le waveform ottenute. Il programma da simulare è mostrato in figura 8.7.



The screenshot shows a terminal window titled "gaem@LAPTOP-KP9RFC ~" running "GNU nano 6.2". The file being edited is "program.ajvm". The code in the editor is:

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

At the bottom of the terminal window, there is a menu bar with the following options: ^G Help, ^O Write Out, ^W Where Is, ^K Cut, ^X Exit, ^R Read File, ^\ Replace, and ^U Paste.

Figura 8.7: Codice IJVM per il testing del progetto del Mic-1

Il programma riserva sullo stack dello spazio per la variabile locale **a**, dopodiché esegue le seguenti operazioni:

1. push sullo stack del byte 0xA;
2. push sullo stack del byte 0xE;
3. somma dei due valori in cima allo stack, che prevede:
 - (a) pop dei due operandi
 - (b) esecuzione della somma
 - (c) push del risultato;
4. memorizzazione dell'elemento in cima allo stack nell'area di memoria assegnata alla variabile **a**.

Per eseguire tale codice IJVM c'è bisogno di un assemblatore che traduca il linguaggio Assembly in linguaggio macchina. Proviamo a noi a predire il risultato dell'assemblatore eseguendo una traduzione manuale del programma. Ogni istruzione macchina sappiamo avere un formato del tipo:

<cod.operativo> <operando>

Sappiamo poi che il codice operativo di ogni istruzione coincide con l'indirizzo della prima microistruzione della microprocedura relativa a tale istruzione. Per ricavare tale indirizzo osserviamo il contenuto del file di progetto `ajvm.mal`. Tale file contiene tutto il contenuto della micro-ROM del processore, ovvero tutte le microprocedture supportate dal processore Mic-1. Le microistruzioni al suo interno sono scritte in linguaggio *MAL* (*Micro Assembly Language*), ovvero un linguaggio simbolico di più alto livello in cui ogni riga rappresenta una microistruzione. In questo file sono riportati anche gli indirizzi degli entry-point delle microprocedture. Vedesi un estratto del file `ajvm.mal` in figura 8.8.

```

GNU nano 6.2      ajvm.mal
# MIC-1 Microprogram
# Copyright (C) 2019 Alberto Moriconi

    goto mic1_entry

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

nop = 0x00:
    goto main

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

isub = 0x5C:

```

Figura 8.8: File `ajvm.mal` del progetto del Mic-1

Siamo ora in grado di abbozzare una traduzione in linguaggio macchina del codice IJVM in figura 8.7.

Tabella 8.2: Traduzione da IJVM a codice macchina

Codice IJVM	Codice macchina
BIPUSH 0xA	0x10 0x0A
BIPUSH 0xE	0x10 0x0E
IADD	0x65
ISTORE a	0x36 <offset>

L'indirizzo della locazione di memoria contenente il valore della variabile **a** è espresso come offset rispetto all'indirizzo base dello stack frame, contenuto nel registro LV.

Il codice macchina del programma in esame sarà poi caricato in memoria, ad una certa locazione, per poter essere eseguito. Andando infatti a visionare il contenuto della RAM nei file VHDL del progetto si possono osservare i byte attesi (vedesi figura 8.9, ricordando che la RAM è organizzata in word da 32 bit in formato little-endian).

```

-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000000000000",
1 => "0000111000010000000010000000", ←
2 => "10100111000000010011011001100101", ←
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);

```

Figura 8.9: Contenuto della RAM del Mic-1

Guardando il contenuto della memoria, si può notare che prima delle istruzioni del nostro programma è presente una word, precisamente la word zero, che sembrerebbe essere inutilizzata. In realtà, tale locazione di memoria contiene i dati utili ad una microprocedura la cui esecuzione precede quella del programma di test. Tale microprocedura è la **mic1_entry**, una sottoparte della microprocedura **invokevirtual**, il cui scopo è quello di inizializzare alcuni registri di base del processore come SP, LV e TOS. Questa procedura di inizializzazione è composta da 20 microistruzioni, dunque richiede 20 colpi di clock per essere completata. La microprocedura **mic1_entry** si trova nella micro-ROM all'indirizzo 0xBC. Dopo l'esecuzione di tale microprocedura, viene eseguita quella di **main**, codice operativo 0x06, che provvede ad effettuare la fetch della prima istruzione macchina del programma. Si può a questo punto passare alla fase di simulazione del programma in figura 8.7, per andare ad osservare l'evoluzione dei registri interni del processore durante il flusso di esecuzione. Per permettere alla macchina di evolvere si è simulato un segnale di clock con periodo 10ns. L'esecuzione del programma IJVM comincia al tempo $t_0 = 215\text{ns}$, perché:

- i primi 15ns sono utilizzati per resettare la macchina;
- 200ns ($20 \times 10\text{ns}$) sono richiesti per eseguire la microprocedura **mic1_entry**.

Per i risultati della simulazione si osservi la figura 8.10.

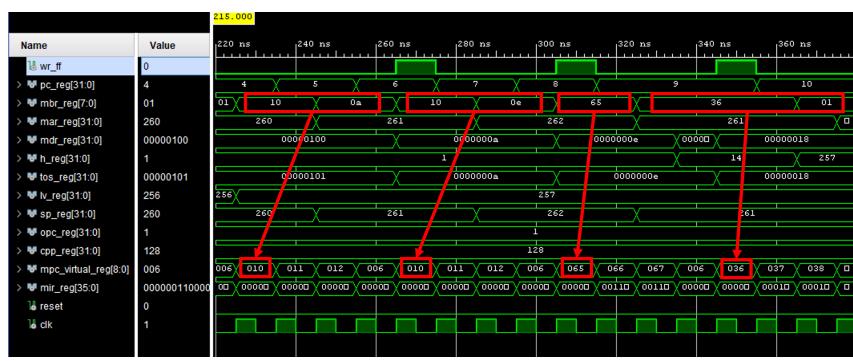
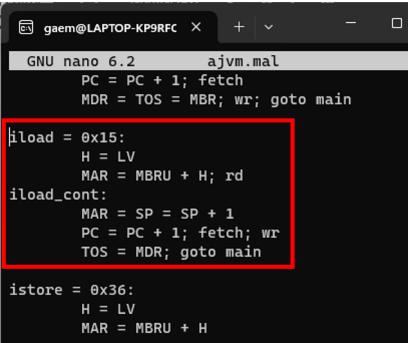


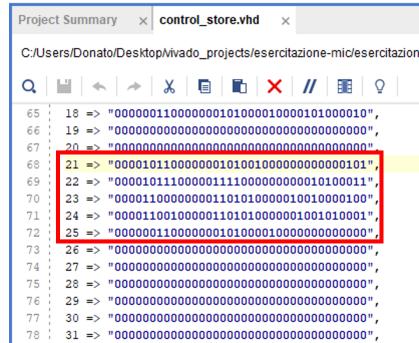
Figura 8.10: Simulazione del programma di test IJVM

8.4 ILOAD

La prima istruzione macchina analizzata è l'istruzione di **ILOAD**, la quale esegue il push del valore di una variabile locale sullo stack. La microprocedura relativa a tale istruzione è riportata in figura 8.11, sia in formato MAL (8.11a) sia in termini di bit all'interno della micro-ROM (8.11b). Essa è posta all'indirizzo 0x15 (21 in decimale) della micro-ROM, ed è composta da cinque microistruzioni.



(a) Codice MAL



(b) Control words nella micro-ROM

Figura 8.11: Micropredicura di ILOAD

Analizziamo il codice una microistruzione alla volta:

1. **H = LV** : viene caricato nel registro H l'indirizzo base che punta all'area di memoria riservata alle variabili locali.
2. **MAR = MBRU + H ; rd** : si somma l'indirizzo base presente in H con l'offset relativo alla variabile da caricare sullo stack. Tale offset è stato caricato nel registro MBR dalla microprocedura di **main** e viene usato in modalità *unsigned* in quanto gli offset possono essere solo positivi. Infine, il risultato della somma è scritto nel registro MAR e viene dato il segnale di **read** per leggere il valore contenuto in tale locazione di memoria.
3. **MAR = SP = SP + 1** : si incrementa il valore dello stack pointer, puntando ad una nuova locazione vuota dello stack, e si scrive tale valore anche nel registro MAR.
4. **PC = PC + 1 ; fetch ; wr** : si incrementa il Program Counter, e si esegue in anticipo la fetch della prossima istruzione. Inoltre si dà il segnale di **write** per scrivere il contenuto del registro MDR (cioè il valore della variabile locale, caricato in MDR dalla read della microistruzione 2) nella locazione puntata da MAR, ovvero la cima dello stack.

5. **TOS = MDR ; goto main** : si scrive in TOS il valore del registro MDR, che coincide con il valore appena caricato sulla cima dello stack, per garantire l'invariante per cui in TOS è sempre presente il valore in cima allo stack. Infine si predispone il ritorno alla microprocedura di **main** per eseguire la prossima istruzione.

Per analizzare l'istruzione di **ILOAD** in esecuzione è stato scritto un semplice programma IJVM che: dichiara una variabile **a**, la inizializza con il valore costante **0x2**, e infine esegue la **ILOAD** per caricare il valore di **a** sullo stack.

A questo punto non ci resta che simulare il progetto VHDL del Mic-1 ed

```
src/main/ajvm/program.ajvm *
.main
.var
a
.endvar
BIPUSH 0x2
ISTORE a
ILOAD a
HALT
.endmethod
```

^G Help ^O Write Out ^W Where Is ^K Cut
^X Exit ^R Read File ^\ Replace ^U Paste

Figura 8.12: Test ILOAD: codice IJVM

Tabella 8.3: Test ILOAD: traduzione da IJVM a codice macchina

Codice IJVM	Codice macchina
BIPUSH 0x2	0x10 0x02
ISTORE a	0x36 <offset_a>
ILOAD a	0x15 <offset_a>

osservare l'evoluzione dei segnali e dei registri durante l'esecuzione della istruzione di **ILOAD** (figura 8.14).

```

-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "0000000100000000000000010000000",
1 => "000000010011011000000100010000", ←
2 => "000000010100111000000300010101", ←
3 => "00000000000000000000000000000000",
others => (others => '0')
--END_WORDS_ENTRY
);

```

Figura 8.13: Test ILOAD: contenuto della RAM

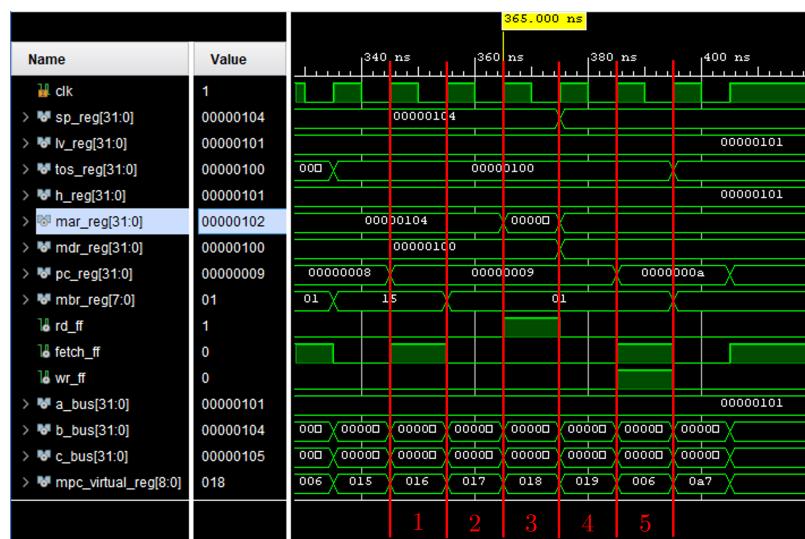


Figura 8.14: Test ILOAD: simulazione del processore Mic-1

8.5 IF_ICMPEQ

La seconda istruzione macchina analizzata è la **IF_ICMPEQ**. Tale istruzione è un codice di salto, e ciò che fa è eseguire le seguenti operazioni:

1. esegue il pop delle due word in cima allo stack;
2. le confronta, eseguendo una differenza;
3. se sono uguali salta all'indirizzo di memoria desiderato;
4. se sono diverse l'esecuzione prosegue linearmente.

La sintassi per utilizzare l'istruzione è la seguente:

```
IF_ICMPEQ <label>
```

dove la **<label>** verrà sostituita poi dall'assemblatore con l'offset da sommare al PC per poter eseguire il salto all'indirizzo voluto. Il processore Mic-1, infatti, implementa solo logiche di salto relativo, e non di salto assoluto. L'offset per eseguire il salto è rappresentato su 2 byte.

Studiamo ora la microprecedura che implementa l'istruzione **IF_ICMPEQ**, analizzando una microistruzione alla volta, ognuna espressa in formato MAL. Il codice è riportato in figura 8.15.

The figure consists of two side-by-side screenshots of a terminal window titled "GNU nano 6.2 src/main/mal/ajvm.mal".

(a) IF_ICMPEQ: codice MAL

```

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

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

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

^G Help      ^O Write Out^W Where Is ^K Cut
^X Exit      ^R Read File^L Replace ^U Paste

```

(b) goto_cont: codice MAL

```

ijvm_goto = 0xA7:
    OPC = PC - 1
    MDR = MBR + H; wr; goto main

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

iflt = 0x9D:
    MAR = SP = SP - 1; rd
    OPC = TOS

^G Help      ^O Write Out^W Where Is ^K Cut
^X Exit      ^R Read File^L Replace ^U Paste

```

Figura 8.15: Microprecedura di **IF_ICMPEQ**

IF_ICMPEQ:

1. **MAR = SP = SP - 1; rd**: decrementa lo stack pointer per puntare alla seconda word sullo stack (**word_2**), ne carica l'indirizzo nel registro MAR e alza il segnale di lettura.

2. $\text{MAR} = \text{SP} = \text{SP} - 1$: decrementa di nuovo lo stack pointer per puntare alla terza word dello stack (`word_3`), che diventerà la nuova cima dello stack a seguito della microprocedura. Copia il valore di SP in MAR.
3. $H = \text{MDR}$; `rd`: scrive la `word_2` letta al passo 1 nel registro H. Alza il segnale di `read` per leggere la `word_3`.
4. $\text{OPC} = \text{TOS}$: scrive il valore di TOS che conteneva la word in cima allo stack, ovvero la `word_1`, nel registro di appoggio OPC.
5. $\text{TOS} = \text{MDR}$: scrive il valore della `word_3`, ovvero la nuova cima dello stack a seguito delle due pop, in TOS per ripristinare l'invariante del registro.
6. $Z = \text{OPC} - H$; `if(Z) goto T; else goto F`: questa microistruzione esegue il confronto tra la `word_1`, contenuta in OPC, e la `word_2`, contenuta in H, facendo la differenza tra i due valori. Se il risultato dell'operazione è zero il flusso di controllo salta alla label T, altrimenti salta alla label F. Tale meccanismo di salto è implementato sfruttando il bit JAMZ della Control Word del processore. Il registro Z in cui viene salvato i risultato della differenza non esiste nel datapath del processore. È un registro fittizio (insieme al registro N) usato nelle operazioni di salto per indicare che il risultato dell'operazione eseguita della ALU non dev'essere salvato in nessun registro.
7. `[T]rue: word_1 = word_2` dunque occorre eseguire il salto sommando l'offset, memorizzato dopo il cod. operativo in memoria, al valore del PC.
 - (a) $\text{OPC} = \text{PC} - 1$; `fetch; goto goto_cont`: in OPC si ripristina il valore del vecchio Program Counter, che era stato incrementato dalla microprocedura di `main`. Si alza il segnale di `fetch` per leggere il byte più significativo dell'offset e si passa alla microprocedura di `goto_cont` che eseguirà il salto aggiornando opportunamente il valore di PC.
8. `[F]alse: word_1 ≠ word_2` dunque l'esecuzione del programma prosegue linearmente.
 - (a) $\text{PC} = \text{PC} + 1$: si incrementa il valore del PC che punterà in questo momento al secondo byte dell'offset di salto.
 - (b) $\text{PC} = \text{PC} + 1$; `fetch`: si incrementa ancora PC per farlo puntare al cod. operativo della prossima istruzione e si alza il segnale di `fetch`.
 - (c) `goto main`: la microprocedura attuale termina e si salta a quella di `main`.

goto_cont:

1. PC = PC + 1; fetch: si incrementa il PC che punterà al byte meno significativo dell'offset e si alza il segnale di fetch.
2. H = MBR << 8: si shifta il contenuto di MBR di un byte verso sinistra e si scrive il risultato nel registro H. In MBR è contenuto il byte più significativo dell'offset.
3. H = MBRU or H: si esegue l'operazione di or tra il contenuto del registro H e il contenuto di MBR, ovvero il byte meno significativo dell'offset, letto senza estensione di segno. Alla fine, nel registro H è scritto l'offset completo a 16 bit con cui eseguire il salto.
4. PC = OPC + H; fetch: si aggiorna il Program Counter scrivendoci la somma tra il valore dell'Old Program Counter e l'offset contenuto in H. Si alza il segnale di fetch per leggere il cod. operativo della prossima istruzione.
5. goto main: la microprocedura attuale termina e si salta a quella di main.

Dopo aver analizzato il codice MAL della microprocedura di `if_icmpEQ`, andiamo a dare un'occhiata al contenuto del Control Store del processore Mic-1. Come si vede in figura 8.15a, l'entry-point della microprocedura è posto all'indirizzo 0xA1, ovvero 166 in decimale. Il contenuto del Control Store è visibile in figura 8.16.

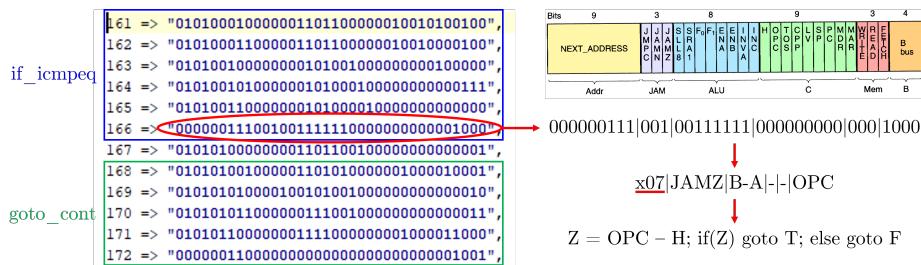


Figura 8.16: IF_ICMP_EQ: contenuto del Control Store

Nel dettaglio della figura 8.16, è riportata la codifica della microistruzione che esegue il confronto tra la `word_1`, contenuta in `OPC`, e la `word_2`, contenuta in `H`. Si può notare che il campo `next_address` contiene il valore `x07`, e che il bit di salto `JAMZ` è asserito. Ricordando che il bit `JAMZ` consente di alzare il bit più significativo del campo `next_address` nel caso in cui il risultato di un'operazione della `ALU` dia come risultato zero, possiamo concludere che:

- il ramo `[F]alse` è posto nel control store a partire dall'indirizzo `x07`;

- il ramo [T]rue è posto nel control store a partire dall'indirizzo x107, ovvero 255 + 7 in decimale.

```
7 => "0000010000000001101010000001000000001",
8 => "000001001000001101010000001000010001",
9 => "000000110000000000000000000000000000001001",
10 => "000000000000000000000000000000000000000000000000000",
11 => "000000000000000000000000000000000000000000000000000",
12 => "000000000000000000000000000000000000000000000000000"
```

Figura 8.17: IF_ICMPEQ: contenuto del Control Store, ramo False

In figura 8.17 sono evidenziate le tre control word appartenenti al ramo [F]alse.

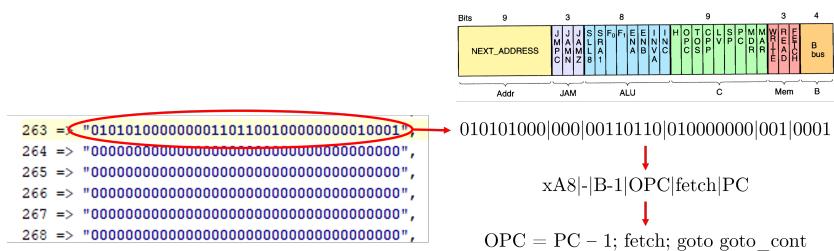


Figura 8.18: IF_ICMPEQ: contenuto del Control Store, ramo True

In figura 8.18 è evidenziata la singola control word appartenente al ramo **[T]rue**, dalla quale si osserva che il campo **next_address** contiene il valore **xA8**, ovvero 168 in decimale; cioè tale campo punta alla prima microistruzione della microprocedura **goto_cont**, com'è possibile osservare in figura 8.16.

Per analizzare l'istruzione di `IF_ICMPEQ` in esecuzione è stato scritto un programma di test in linguaggio IJVM, riportato in figura 8.19.

Il codice succitato esegue le seguenti operazioni:

1. definisce due costanti **x** e **y** pari rispettivamente a 128 e 127, memorizzate nell'area di memoria detta *constant pool* e puntata dal registro CPP;
 2. carica le due costanti **x** e **y** sullo stack tramite l'istruzione LDC_W;
 3. esegue l'istruzione di IF_ICMPEQ che esegue il pop dei due valori appena caricati e ne effettua un confronto; al primo confronto i due valori risulteranno diversi, dunque il programma prosegue l'esecuzione linearmente;
 4. il programma riesegue il push della costante **x** e ne crea una copia identica sullo stack tramite l'istruzione macchina DUP;
 5. attraverso l'istruzione GOTO il flusso di controllo ritorna all'istruzione IF_ICMPEQ;

```

src/main/ajvm/program.ajvm
.constant
x    128
y    127
.endconstant

.main
LDC_W x
LDC_W y
ciclo:
IF_ICMPEQ fine
LDC_W x
DUP
GOTO ciclo
fine:
HALT
.endmethod

```

^G Help ^O Write Out ^W Where Is ^K Cut
^X Exit ^R Read File ^\ Replace ^U Paste

Figura 8.19: Test IF_ICMPEQ: codice IJVM

6. la seconda volta, il confronto effettuato da IF_ICMPEQ tra le due parole in cima allo stack darà esito positivo, saltando direttamente alla fine del programma.

Tabella 8.4: Test IF_ICMPEQ: traduzione da IJVM a codice macchina

Codice IJVM	Codice macchina
LDC_W x	0x20 <off_x>
LDC_W y	0x20 <off_y>
IF_ICMPEQ fine	0xA1 <off_FINE>
LDC_W x	0x20 <off_x>
DUP	0x57
GOTO ciclo	0xA7 <off_ciclo>
HALT	0xA7

In tabella 8.4 è riportata la traduzione del programma IJVM in linguaggio macchina in formato esadecimale. Si può notare come tutte le label presenti nel listato siano state sostituite dagli offset necessari ad accedere a tali risorse con una logica di indirizzamento relativo. In particolare,

- <off_x> e <off_y> sono gli offset delle due costanti x e y rispetto al valore del CPP;
- <off_FINE> e <off_ciclo> sono gli offset delle due istruzioni etichettate con le label fine e ciclo rispetto al valore del Program Counter.

```

-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN_WORDS_ENTRY
128 => "00000000000000000000000000000000",
129 => "0000000000000000000000000000000010000000", ▶ 128
130 => "000000000000000000000000000000001111111" ▶ 127
0 => "0000000000000000000000000000000010000000",
1 => "0010000000000000100000000001000000", ▶
2 => "00000000101000001000000010000000", ▶
3 => "000000000000000000100000000001010", ▶
4 => "11111001111111101000110101011", ▶
5 => "0000000000000000000000000000000010100111",
others => (others => '0')
--END_WORDS_ENTRY
);

```

20	01	00	20
00	A1	02	00
01	00	20	0A
F9	FF	A7	57
00	00	00	A7

Figura 8.20: Test IF_ICMPEQ: contenuto della memoria

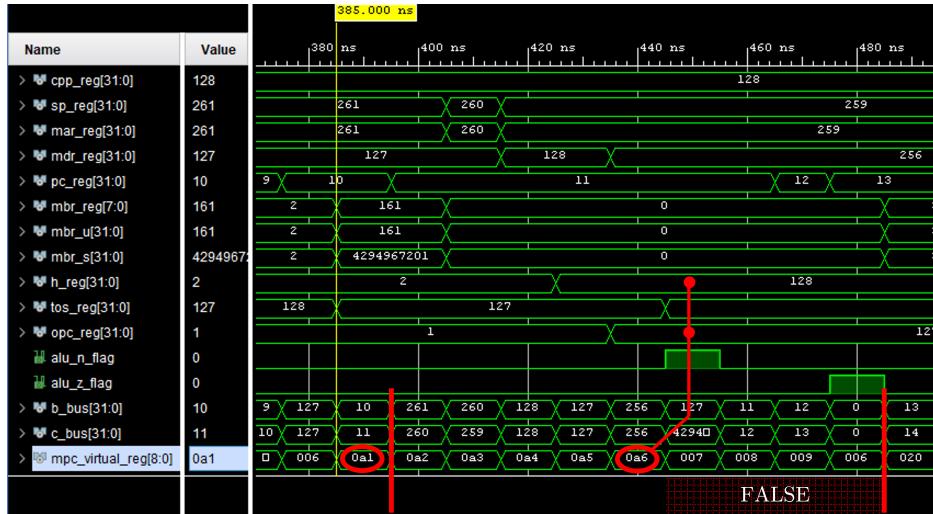
In figura 8.20, possiamo osservare come il programma IJVM sia stato tradotto all'interno della memoria RAM. In particolare, sono state evidenziate in rosso le istruzioni macchina che compongono il programma e in blu le costanti memorizzate all'interno del constant pool. Osservando i byte memorizzati in memoria (in formato little-endian) notiamo una perfetta congruenza con quanto riportato in tabella 8.4, dove ai placeholder sono sostituiti i valori effettivi degli offset riportati su 2 byte. L'unico valore particolare che notiamo in memoria è quello relativo a `<off_ciclo>`, che è stato riempito col valore `FF F9`, che altro non è che la rappresentazione in complemento a due di `-7`. Infatti, come si vede nel codice in figura 8.19, l'istruzione `GOTO ciclo` è l'unica istruzione che effettua un salto all'indietro.

A questo punto, non ci resta che eseguire il programma sul processore Mic1 simulato in VHDL, ed osservare l'evoluzione della macchina sulle waveform prodotte dal test.

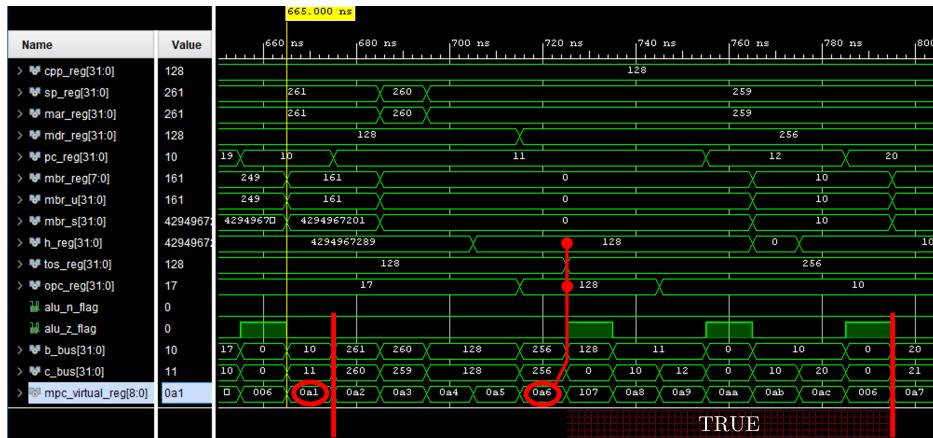
In figura 8.21 è riportata l'evoluzione dei segnali e dei registri interni al processore sia nel caso in cui il confronto dia esito positivo, sia nel caso che dia esito negativo. Per leggere opportunamente i grafici si può osservare l'evoluzione del registro MPC che ci permette di monitorare l'esecuzione delle varie microistruzioni del processore, ricordandoci però che il registro MPC punta sempre alla microistruzione successiva, e non quella corrente. Nei grafici sono stati messi in evidenza:

- l'inizio e la fine della microprocedura di `IF_ICMPEQ`;
- l'indirizzo della prima microistruzione `0xA1`;
- l'indirizzo della microistruzione che effettua il confronto `0xA6`, con i valori dei registri confrontati ovvero `H` e `OPC`;
- le microistruzioni relative al ramo False e al ramo True.

Nella figura 8.21b è chiaramente visibile come al momento del confronto il bit `Z` della ALU sia alto, ad indicare che i due operandi sono uguali. La micro-



(a) Esecuzione del ramo False



(b) Esecuzione del ramo True

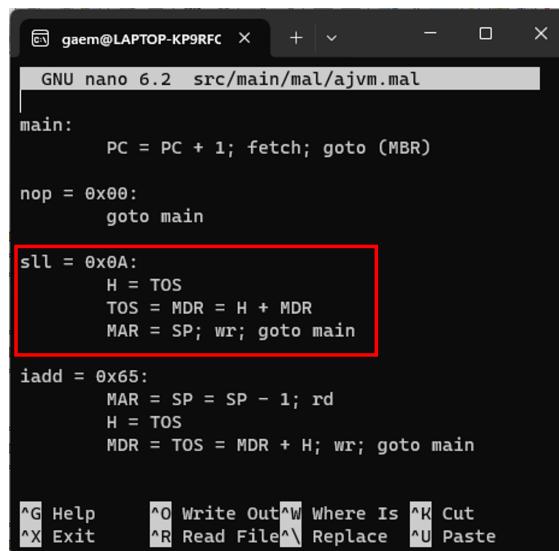
Figura 8.21: Test IF_ICMPEQ: waveform prodotte dalla simulazione

procedura di IF_ICMPEQ termina sempre con un ritorno alla microprocedura di main (0x06).

8.6 Shift Left Logic

Come soluzione per l'esercizio 8.2 si è deciso di estendere il set di istruzioni macchina del processore Mic-1, aggiungendo un nuovo codice operativo. L'operazione aggiuntiva prende il nome di Shift Left Logic (SLL) ed esegue le seguenti operazioni:

1. fa il pop di una parola dallo stack;
2. ne calcola lo shift verso sinistra di un bit;
3. esegue il push del risultato.



The screenshot shows a terminal window titled "gaem@LAPTOP-KP9RFC ~" running "GNU nano 6.2". The file being edited is "src/main/mal/ajvm.mal". The assembly code includes the SLL instruction at address 0x0A:

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

nop = 0x00:
    goto main

sll = 0x0A:
    H = TOS
    TOS = MDR = H + MDR
    MAR = SP; wr; goto main

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

^G Help      ^O Write Out^W Where Is ^K Cut
^X Exit      ^R Read File^L Replace ^U Paste
```

The line "TOS = MDR = H + MDR" is highlighted with a red rectangle.

Figura 8.22: SLL: codice MAL

In figura 8.22 è riportato il codice MAL con cui è stata implementata l'operazione di shift logico. In particolare, per eseguire tale comando si è sfruttata la proprietà del codice binario per cui sommando un numero a sé stesso, ovvero calcolandone il doppio, si ottiene la sua versione shiftata verso sinistra di esattamente un bit, e viceversa. Analizziamo il codice MAL riga per riga:

1. $H = TOS$: si sfrutta la proprietà per cui nel registro TOS è già presente il valore della parola in cima allo stack per non eseguire un pop esplicito. Si carica tale valore nel registro H .
2. $TOS = MDR = H + TOS$: si sommano il contenuto del registro H e quello del registro TOS , ottenendo il doppio di ciò che era contenuto in TOS inizialmente. Si scrive il risultato nel registro MDR per poterlo riportare poi sullo stack, e nel registro TOS per ripristinare l'invariante del registro.

3. **MAR=SP; wr; goto main:** infine si imposta in MAR il valore dello Stack Pointer e si alza il segnale di write per scrivere il risultato sullo stack. Il flusso di controllo ritorna poi alla microprocedura di **main**.

Una volta prodotto il codice MAL necessario ad implementare l'operazione **SLL**, occorre individuare una locazione libera all'interno del Control Store in cui piazzare la nuova microprocedura. Dato che la microprocedura **SLL** è composta da tre microistruzioni, occorre individuare nella micro-ROM un buco di almeno tre parole di memoria. Come si può notare in figura 8.23, tale spazio libero è stato individuato tra la locazione 0x0A e la locazione 0x0C (da 10 a 12 in decimale).

Figura 8.23: SLL: modifica del Control Store

Ora non ci resta che testare il funzionamento del nuovo codice operativo scrivendo un semplice programma IJVM da simulare all'interno del progetto VHDL del Mic-1. Tuttavia, c'è una complicazione: affinché l'assemblatore IJVM riconosca il nuovo codice operativo SLL è necessario che questi venga riprogrammato e ricompilato. Per evitare ciò, si è scelto di scrivere manualmente il codice macchina del programma di test all'interno della memoria della macchina. Si riporta in tabella 8.5 il codice del programma di test, e in figura 8.24 il contenuto della memoria RAM.

Tabella 8.5: Test SLL: traduzione da IJVM a codice macchina

Codice IJVM	Codice macchina
BIPUSH 0x2	0x10 0x02
SLL	0x0A
ISTORE x	0x36 <offset_x>

Infine, non ci resta che eseguire il programma di test in simulazione, e controllare che la macchina evolva come atteso. I risultati della simulazione sono riportati in figura 8.25.

Dalla simulazione si osserva che non ci sono errori: la macchina evolve correttamente, ed il risultato prodotto dall'operazione è corretto come si può osservare dal contenuto del registro **TOS**, che passa da valere 2 a 4.

```

-- RAM content
signal mem : dp_ar_ram_type := (
--BEGIN WORDS_ENTRY
128 => "00000000000000000000000000000000",
0 => "00000001000000000000000010000000",
1 => "001101100000101000000100010000", 36 0A 02 10
2 => "00000000000000001010011000000001", 00 00 A7 01
others => (others => '0')
--END WORDS_ENTRY
);

```

Figura 8.24: SLL: contenuto della memoria

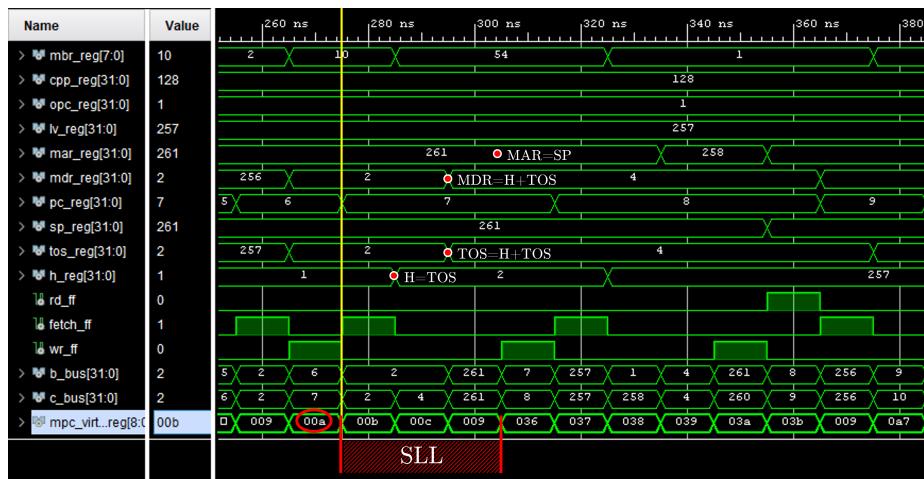


Figura 8.25: SLL: risultati della simulazione

Capitolo 9

Interfaccia seriale

9.1 Traccia

Esercizio 9.1 Sfruttando l'implementazione fornita dalla Digilent di un dispositivo UART (componente RS232RefComp.vhd), progettare e implementare in VHDL un sistema costituito da 2 nodi A e B collegati tra loro mediante una interfaccia seriale. Il sistema A acquisisce una stringa di 8 bit dall'utente (mediante gli switch della board di sviluppo) e la invia mediante la seriale al sistema B, che la manda in output sui led della board di sviluppo.

Esercizio 9.2 Implementare uno dei seguenti sistemi a scelta dello studente:

- (a) 2_UART_MEM: come variante dell'esercizio 8.1, il sistema A invia al sistema B tramite l'interfaccia seriale N stringhe di 8 bit contenute all'interno di una memoria ROM. Le stringhe ricevute vengono memorizzate in una memoria locale a B. Il progetto deve prevedere che A utilizzi un componente contatore per scandire le N stringhe da inviare.
- (b) UART_PC: il sistema realizza la comunicazione fra un nodo A rappresentato da un componente sintetizzato su un FPGA e un nodo B rappresentato da 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 A acquisisce una stringa di 8 bit che rappresenta un carattere in codifica ASCII fornita dall'utente mediante gli switch della board di sviluppo, e la invia mediante il dispositivo UART al terminale B 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.

9.2 Soluzione

La UART (Universal Asynchronous Receiver-Transmitter) è un dispositivo periferico utilizzato per implementare una comunicazione seriale asincrona tra due nodi di calcolo. Il protocollo di comunicazione seriale implementato da tale dispositivo è il protocollo RS232. La comunicazione seriale sfrutta di base solo due fili: uno per dare un riferimento di tensione comune ai due nodi (GND), e l'altro su cui sono modulati i bit del dato da trasmettere. Ovviamente, oltre tali segnali ne sono previsti altri dal protocollo RS232, tra cui alcuni dedicati all'instaurazione del processo di comunicazione tramite un meccanismo di handshaking. Questi ultimi segnali sono:

- **DTR** (Data Terminal Ready);
- **DSR** (Data Set Ready);
- **RTS** (Request To Send);
- **CTS** (CLear To Send);

La comunicazione seriale può essere di tipo *simplex*, *half-duplex*, o *full-duplex*, in base all'implementazione dei dispositivi periferici utilizzati. La trasmissione tra due nodi in modalità asincrona avviene per unità atomiche dette *data frame* il cui formato è definibile, entro certi limiti, dai due interlocutori. Il protocollo RS232 prevede che trasmettitore e ricevitore si mettano d'accordo a priori su alcune caratteristiche della trasmissione:

- **baud rate**: numero di bit trasmessi al secondo;
- numero di bit dato contenuti in un data frame;
- eventuale presenza di un bit di parità nel data frame, e il tipo di controllo di parità: pari o dispari;
- numero di bit di stop.

Ogni data frame, dunque, è così composto:

- un bit di start, pari a zero [‘0’];
- i bit dato [5...8];
- bit di parità [0...1];
- bit di stop [1...2].

La linea dato, nel momento in cui non viene utilizzata, è mantenuta ad un valore di tensione alto. Il trasmettitore si occupa di modulare nel tempo i data frame da trasmettere sulla linea dato, con una frequenza di trasmissione pari

al baud rate fissato. Il ricevitore, per poter ricevere il dato, deve campionare la linea dato, e lo fa con una frequenza multipla di quella di trasmissione (16x o 64x). È importante notare che ricevitore e trasmettitore non condividono un riferimento temporale comune, ma evolvono ognuno secondo il proprio segnale di clock. Affinché il ricevitore sia in grado di campionare correttamente la linea dato, senza avere lo stesso riferimento temporale del trasmettitore, è necessario definire un protocollo di comunicazione seriale (l'RS232 appunto). Oltre a dover campionare il segnale in ricezione con una frequenza multipla di quella di trasmissione, il ricevitore deve anche campionare i bit che viaggiano sulla linea in istanti di tempo ben precisi, ovvero al centro del segnale (figura 9.1). Questo avviene per evitare due diversi tipi di errori di campionamento:

- limitare gli effetti della deriva del segnale di clock che può portare a mancare alcuni bit;
- evitare di campionare il segnale nell'intorno dei fronti, dove un segnale reale presenta un andamento *smooth* che può causare il campionamento di un valore errato.

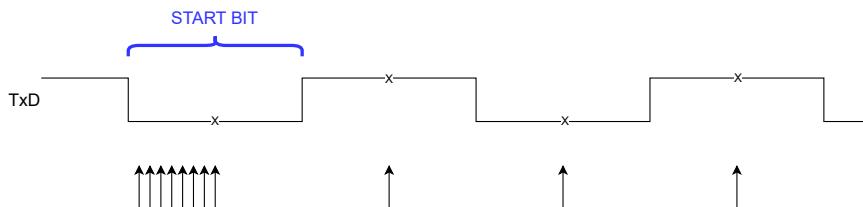


Figura 9.1: Campionamento della linea dato

Il ricevitore, per portarsi al centro dei bit da campionare, sfrutta lo *start bit* del data frame. Supponiamo che esso lavori con una frequenza pari a 16 volte quella di trasmissione; nel momento in cui il ricevitore si accorge che la linea dato è stata abbassata, a causa dello start bit appunto, inizierà a contare 8 colpi del proprio clock per portarsi a centro del bit, e da quel momento in poi campionerà il segnale ogni 16 colpi di clock.

La UART è in grado di rilevare degli errori in fase di ricezione, e notificarli all'esterno. Gli errori rilevati in ricezione sono di tre tipi:

Parity Error (PE) errore di parità nel dato, rilevato grazie al *parity bit*.

Frame Error (FE) errore nel formato del frame ricevuto, rilevato nel momento in cui il ricevitore non riesca a campionare gli *stop bit* prefissati.

Overrun Error (OE) errore di sovrascrittura del buffer in ricezione, che avviene nel momento in cui un secondo dato viene ricevuto dalla UART prima che il precedente sia letto dall'esterno.

Oltre ai tre tipi di errori rilevati in ricezione, la UART ne prevede anche uno in trasmissione:

Transmit Buffer Empty (TBE) è un errore di underrun che viene notificato nel momento in cui il buffer di trasmissione è vuoto.

Il segnale TBE viene usato dalla UART in trasmissione per notificare verso l'esterno che essa è libera per trasmettere. Analogamente in ricezione, la UART alza un segnale RDA (Read Data Available) per notificare verso l'esterno che è stato ricevuto un dato e che è pronto per essere letto sulla linea di uscita.

Gli esercizi 9.1 e 9.2 sono stati sviluppati in maniera separata e indipendente, e così saranno trattati all'interno del presente elaborato. Le soluzioni di entrambi gli esercizi fanno uso del componente *UARTcomponent*, definito nel file `RS232RefComp2.vhd`. Tale componente modella il comportamento di una UART con due porti: uno in ricezione (Rx) e uno in trasmissione (Tx). Istanziando due componenti UART, e collegando il porto TX dell'uno con il porto Rx dell'altro, è possibile implementare uno schema di comunicazione seriale. Il componente in questione non richiede segnali di handshaking, comportandosi come se tale meccanismo fosse già stato eseguito a monte. Il codice VHDL dell'UARTcomponent definisce due generic: `BAUD_RATE_G` e `BAUD_DIVIDE_G`, per impostare la frequenza di trasmissione e quella di ricezione, secondo le seguenti relazioni:

$$f_{TX} = \frac{f_{CLK}}{BAUD_RATE_G} \quad f_{RX} = \frac{f_{CLK}}{BAUD_DIVIDE_G}$$

Si è scelto, in fase di simulazione e implementazione, dei valori di `BAUD_RATE_G` e `BAUD_DIVIDE_G` tali per cui

$$f_{RX} = 16 \times f_{TX}$$

Il formato del data frame utilizzato dall'UARTcomponent durante la trasmissione è il seguente:

- un bit di start ('0')
- otto bit dato
- un bit di parità, con parità dispari
- un bit di stop ('1').

9.2.1 Esercizio 9.1

L'esercizio 9.1 è abbastanza semplice, in quanto richiede unicamente di istanziare i due componenti UART, collegarli tra di loro e pilotarli in maniera opportuna. I segnali in ingresso e in uscita dal sistema sono poi mappati

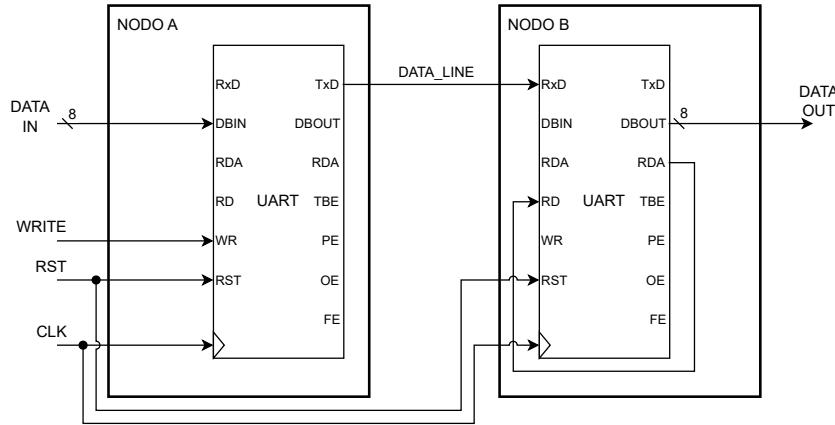


Figura 9.2: Schematico esercizio 9.1

sui componenti di I/O della board di sviluppo Xilinx. L'unica cosa un po' particolare di questa soluzione la si vede nel nodo B in figura 9.2: il segnale RD in ingresso alla UART è stato collegato con l'uscita RDA della UART stessa. Ciò è stato fatto per fare in modo che lo strobe di lettura su RD venga dato automaticamente nel momento in cui RDA si alza per notificare l'arrivo di un nuovo dato. Ciò evita l'implementazione di ulteriore logica per la gestione del segnale RD che in questo esercizio non era richiesta, ma che vedremo nel successivo.

9.2.2 Esercizio 9.2

Questo secondo esercizio è una variante del primo in cui occorre aumentare la complessità dei due nodi A e B, in modo che questi non alimentino, né siano alimentati dai segnali di I/O della board, ma attraverso delle memorie interne. Ovviamente l'impulso per avviare la comunicazione tra nodo A e nodo B proverrà sempre dall'esterno, in particolare da un button della scheda. I due nodi sono stati progettati, in questo caso, secondo l'approccio tipico per sistemi complessi, scomponendo ognuno di essi in parte operativa e parte di controllo, e realizzando le due parti in maniera separata. La parte di controllo è stata realizzata in logica cablata. Vediamo in figura 9.3 lo schema complessivo del progetto, che esternamente non è molto diverso da quello dell'esercizio precedente.

Nodo A

Il nodo A deve, in seguito al ricevimento di un segnale di avvio, inviare in maniera sequenziale N stringhe da 8 bit contenute in una memoria ROM verso il nodo B, attraverso un meccanismo di comunicazione seriale. Dunque l'*unità operativa* del nodo A avrà come componenti costituenti:

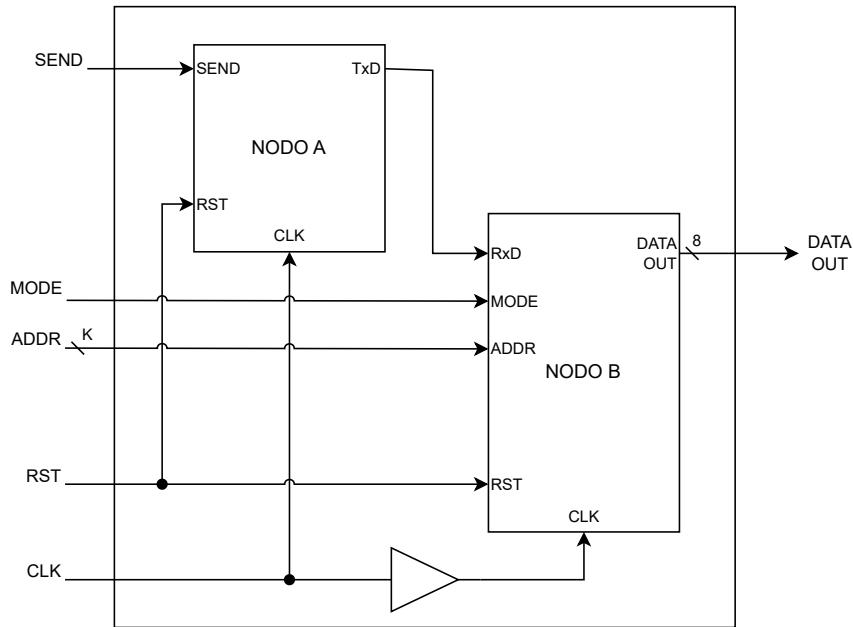


Figura 9.3: Schematico esercizio 9.2

- un contatore modulo N
- una memoria ROM di dimensioni $N \times 8$
- un dispositivo UART

collegati come in figura 9.4. Per funzionare correttamente, l'unità operativa del nodo A richiede la corretta orchestrazione dei segnali di controllo:

- INCR: per incrementare il valore di conteggio del contatore;
- READ: per dare lo strobe di lettura alla ROM;
- WRITE: per fornire alla UART lo strobe per avviare la trasmissione
- RST_OU: per resettare l'unità operativa.

Inoltre, l'unità operativa fornisce all'unità di controllo i due segnali DIV e TBE. Il primo si alza dopo aver inviato tutte le N stringhe contenute nella ROM, quando il valore di conteggio del counter torna a zero, mentre il secondo si alza per indicare che la UART è pronta a trasmettere.

L'*unità di controllo* del nodo A è realizzata mediante l'implementazione dell'automa in figura 9.5.

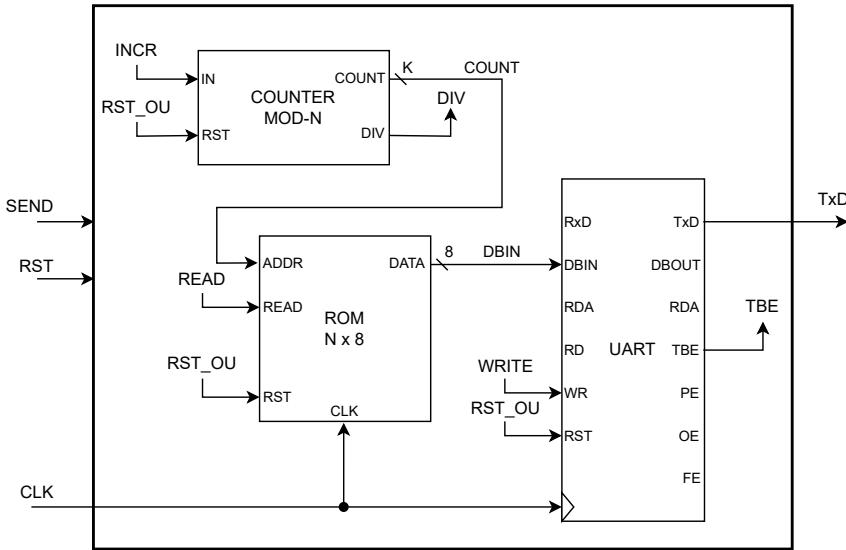


Figura 9.4: Schematico unità operativa del nodo A

INPUT: <SEND, TBE, DIV, RST>
OUTPUT: <INCR, READ, WRITE, RST_OU>

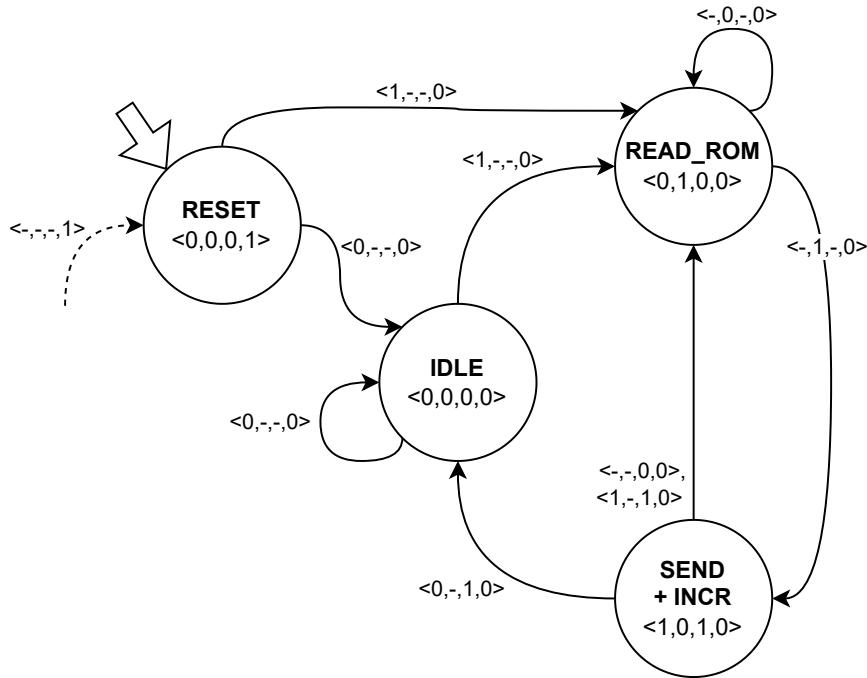


Figura 9.5: Unità di controllo del nodo A

Nodo B

Il nodo B ha il compito di ricevere, attraverso un dispositivo UART, le stringhe inviategli dal nodo A e memorizzarle in maniera sequenziale all'interno di una

propria memoria interna. A tale scopo, si è utilizzato un contatore modulo N, ed una memoria di dimensione $N \times 8$. Inoltre, si è deciso di aggiungere una seconda modalità di funzionamento in cui è possibile leggere il contenuto della memoria, settando l'indirizzo della locazione da leggere tramite dei segnali esterni mappabili sugli switch della scheda di valutazione. Il cambio di modalità avviene in base al valore di un segnale proveniente dall'esterno, anch'esso mappabile su di uno switch. L'*unità operativa* appena descritta è visibile in figura 9.6.

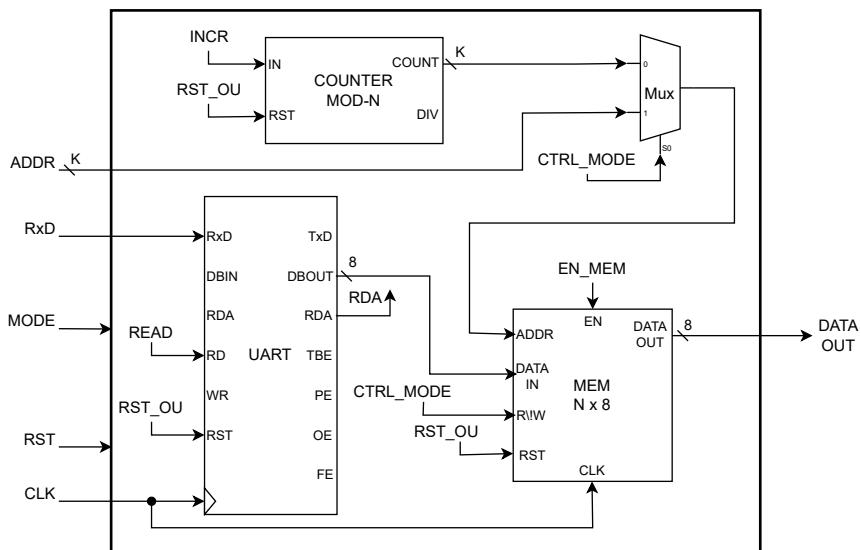


Figura 9.6: Schematico unità operativa del nodo B

L'*unità di controllo* del nodo B deve opportunamente pilotare i segnali: INCR, EN_MEM, READ, CTRL_MODE e RST_OU, affinché l'unità operativa evolva correttamente. Per fare ciò, la control unit opera secondo il modello in figura 9.7, utilizzando in ingresso i segnali di MODE, RST e RDA.

INPUT: < MODE, RDA, RST >
OUTPUT: < READ, INCR, EN_MEM, CTRL_MODE, RST_OU >

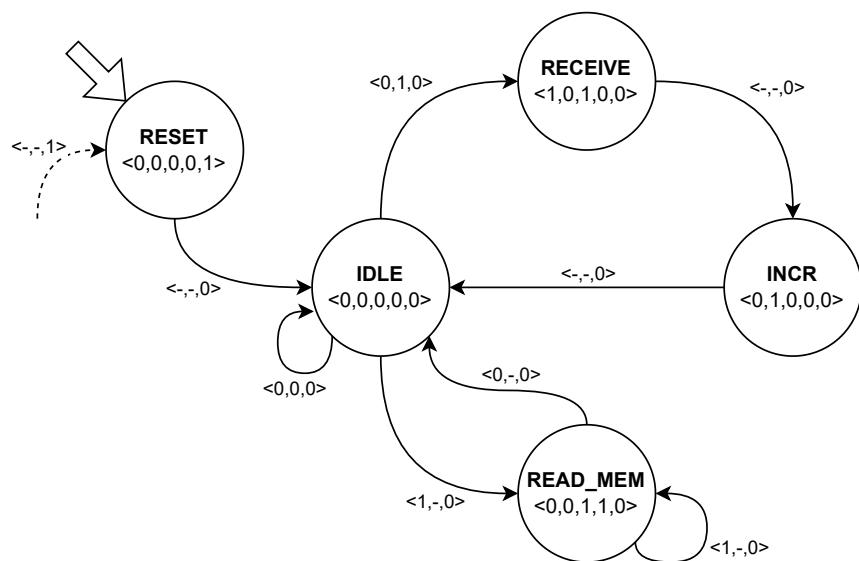


Figura 9.7: Unità di controllo del nodo B

9.3 Codice

9.3.1 Esercizio 9.1

Nodo A

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA is
5     generic(
6         BAUD_RATE_G : positive );           -- f_Tx = f_CLK /
7         --> BAUD_RATE_G
8     Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
9             write : in STD_LOGIC;
10            rst : in STD_LOGIC;
11            clk : in STD_LOGIC;
12            TxD : out STD_LOGIC);
13
14 end nodoA;
15
16 architecture structural of nodoA is
17
18 begin
19     uart_A : entity work.UARTcomponent
20     generic map(
21         BAUD_RATE_G => BAUD_RATE_G )           -- f_Tx = f_CLK /
22         --> BAUD_RATE_G
23     port map(
24         DBIN => data_in,
25         WR => write,
26         RST => rst,
27         CLK => clk,
28         TXD => TxD,
29         RXD => '1',
30         RD => '0' );
31
32 end structural;
```

Listing 9.1: Esercizio 9.1 - Nodo A

Nodo B

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoB is
5     generic(
6         BAUD_DIVIDE_G : positive );           -- f_Rx = f_CLK /
7         -- / BAUD_DIVIDE_G
8     Port ( RxD : in STD_LOGIC;
9             rst : in STD_LOGIC;
10            clk : in STD_LOGIC;
11            data_out : out STD_LOGIC_VECTOR (7 downto 0));
12 end nodoB;
13
14
15 architecture structural of nodoB is
16
17
18 begin
19     rd <= rda after 1ns;
20
21     uart_B : entity work.UARTcomponent
22     generic map(
23         BAUD_DIVIDE_G => BAUD_DIVIDE_G)           -- f_Rx = f_CLK
24         -- / BAUD_DIVIDE_G
25     port map(
26         RXD => RxD,
27         RST => rst,
28         CLK => clk,
29         RDA => rda,
30         RD => rd,
31         DBOUT => data_out,
32         DBIN => (others => '0'),
33         WR => '0' );
34 end structural;
```

Listing 9.2: Esercizio 9.1 - Nodo B

Sistema complessivo

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity sys is
5     generic(
6         BAUD_DIVIDE_G : positive := 651);          -- baud_rate =
7         -- 9600 con f_CLK = 100MHz
8     Port ( data_in : in STD_LOGIC_VECTOR (7 downto 0);
9             write : in STD_LOGIC;
10            rst : in STD_LOGIC;
11            clk : in STD_LOGIC;
12            data_out : out STD_LOGIC_VECTOR (7 downto 0));
13
14 end sys;
15
16
17 architecture structural of sys is
18
19 begin
20     nodoA : entity work.nodoA
21     generic map(
22         BAUD_RATE_G => BAUD_RATE_G)           -- f_Tx = f_CLK /
23         -- BAUD_RATE_G
24     port map(
25         data_in => data_in,
26         write => write,
27         rst => rst,
28         clk => clk,
29         TxD => data_line);
30
31     nodoB : entity work.nodoB
32     generic map(
33         BAUD_DIVIDE_G => BAUD_DIVIDE_G)      -- f_Rx = f_CLK /
34         -- BAUD_DIVIDE_G
35     port map(
36         RxD => data_line,
37         rst => rst,
38         clk => clk,
39         data_out => data_out );
```

```
38
39 end structural;
```

Listing 9.3: Esercizio 9.1 - Sistema complessivo

9.3.2 Esercizio 9.2

Per l'implementazione dei nodi A e B sono stati definiti due package VHDL di supporto nei file `nodoA_util.vhd` e `nodoB_util.vhd`. I due package sono molto semplici, ed hanno il solo scopo di definire un nuovo tipo per le *control word* che le unità di controllo generano in output verso le rispettive unità operative. Il contenuto dei package non viene riportato nell'elaborato per non appesantire ulteriormente la trattazione.

Nodo A - Unità operativa

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoA_util.all;
4
5 entity nodoA_ou is
6     generic(
7         BAUD_RATE_G : positive;           -- f_Tx = f_CLK /
8             ↳ BAUD_RATE_G
9         K : positive );
10    Port ( clk : in STD_LOGIC;
11            ctrl_word : in ctrl_word_type;
12            TxD : out STD_LOGIC;
13            div : out STD_LOGIC;
14            TBE : out std_logic);
15 end nodoA_ou;
16
17 architecture structural of nodoA_ou is
18
19     signal count : std_logic_vector(K-1 downto 0);
20     signal dbin : std_logic_vector(7 downto 0);
21
22 begin
23     counter0 : entity work.counter
24         generic map(
25             bits => K,
26             delay => 0ns )
27         port map(
```

```

27          x => ctrl_word(incr),
28          rst => ctrl_word(rst_ou),
29          y => count,
30          div => div );
31
32      rom0 : entity work.rom
33          generic map(
34              nbit_addr => K,
35              width => 8 )
36          port map(
37              addr => count,
38              clk => clk,
39              read => ctrl_word(read),
40              rst => ctrl_word(rst_ou),
41              data_out => dbin );
42
43      uart0 : entity work.UARTComponent
44          generic map(
45              BAUD_RATE_G => BAUD_RATE_G)           -- f_Tx = f_CLK
46                      -- / BAUD_RATE_G
47          port map(
48              DBIN => dbin,
49              WR => ctrl_word(write),
50              RST => ctrl_word(rst_ou),
51              CLK => clk,
52              RXD => '1',
53              RD => '0',
54              TBE => TBE,
55              TXD => TxD );
56
57 end structural;

```

Listing 9.4: Esercizio 9.2 - Unità operativa nodo A

Nodo A - Unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoA_util.all;
4
5 entity nodoA_cu is
6     Port ( send : in STD_LOGIC;
7             TBE : in STD_LOGIC;

```

```

8      div : in STD_LOGIC;
9      rst : in STD_LOGIC;
10     clk : in std_logic;
11     ctrl_word : out ctrl_word_type );
12 end nodoA_cu;
13
14 architecture Behavioral of nodoA_cu is
15
16     type state_type is (RESET, IDLE, READ_ROM, SEND_INCR);
17     signal state : state_type := RESET;
18     signal next_state : state_type;
19
20 begin
21
22     process(clk)
23     begin
24         if clk'event and clk = '0' then
25             if rst = '1' then
26                 state <= RESET;
27             else
28                 state <= next_state;
29             end if;
30         end if;
31     end process;
32
33     process(state,send,TBE,div)
34     begin
35         case state is
36             when RESET =>
37                 ctrl_word <= "0001";
38                 if send = '1' then
39                     next_state <= READ_ROM;
40                 else
41                     next_state <= IDLE;
42                 end if;
43
44             when IDLE =>
45                 ctrl_word <= "0000";
46                 if send = '1' then
47                     next_state <= READ_ROM;
48                 else
49                     next_state <= IDLE;
50                 end if;
51

```

```

52      when READ_ROM =>
53          ctrl_word <= "0100";
54          if TBE = '1' then
55              next_state <= SEND_INCR;
56          else
57              next_state <= READ_ROM;
58          end if;
59
60      when SEND_INCR =>
61          ctrl_word <= "1010";
62          if div = '0' then
63              next_state <= READ_ROM;
64          elsif send = '1' then
65              next_state <= READ_ROM;
66          else
67              next_state <= IDLE;
68          end if;
69      end case;
70  end process;
71
72 end Behavioral;

```

Listing 9.5: Esercizio 9.2 - Unità di controllo nodo A

Nodo A

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.nodoA_util.all;
4
5  entity nodoA is
6      generic(
7          BAUD_RATE_G : positive;           -- f_Tx = f_CLK /
8          K : positive );
9      Port ( send : in STD_LOGIC;
10             rst : in STD_LOGIC;
11             clk : in STD_LOGIC;
12             TxD : out STD_LOGIC);
13  end nodoA;
14
15 architecture structural of nodoA is
16

```

```

17      signal cword : ctrl_word_type;
18      signal div : std_logic;
19      signal TBE : std_logic;
20
21 begin
22     ou : entity work.nodoA_ou
23     generic map(
24       BAUD_RATE_G => BAUD_RATE_G,           -- f_Tx = f_CLK /
25       ↳ BAUD_RATE_G
26       K => K )
27     port map(
28       clk => clk,
29       ctrl_word => cword,
30       TxD => TxD,
31       div => div,
32       TBE => TBE );
33
34     cu : entity work.nodoA_cu
35     port map(
36       send => send,
37       TBE => TBE,
38       div => div,
39       rst => rst,
40       clk => clk,
41       ctrl_word => cword );
42 end structural;

```

Listing 9.6: Esercizio 9.2 - Nodo A

Nodo B - Unità operativa

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4
5 entity nodoB_ou is
6   generic(
7     BAUD_DIVIDE_G : positive;           -- f_Rx = f_CLK /
8     ↳ BAUD_DIVIDE_G
9     K : positive );
10  Port ( RxD : in STD_LOGIC;
11        clk : in STD_LOGIC;

```

```

11      addr : in STD_LOGIC_VECTOR (K-1 downto 0);
12      ctrl_word : in ctrl_word_type;
13      data_out : out STD_LOGIC_VECTOR (7 downto 0);
14      RDA : inout STD_LOGIC);
15  end nodoB_ou;
16
17 architecture structural of nodoB_ou is
18
19     signal dbout : std_logic_vector(7 downto 0);
20     signal count : std_logic_vector(K-1 downto 0);
21     signal mem_addr : std_logic_vector(K-1 downto 0);
22
23 begin
24     uart0 : entity work.UARTcomponent
25         generic map(
26             BAUD_DIVIDE_G => BAUD_DIVIDE_G )           -- f_Rx =
27             ↵ f_CLK / BAUD_DIVIDE_G
28         port map(
29             DBIN => (others => '0'),
30             WR => '0',
31             RST => ctrl_word(rst_ou),
32             CLK => clk,
33             RXD => RxD,
34             RD => ctrl_word(read),
35             DBOUT => dbout,
36             RDA => RDA );
37
38     counter0 : entity work.counter
39         generic map(
40             bits => K,
41             delay => 0ns )
42         port map(
43             x => ctrl_word(incr),
44             rst => ctrl_word(rst_ou),
45             y => count );
46
47     mux0 : entity work.mux_2_1
48         generic map(
49             width => K )
50         port map(
51             x0 => count,
52             x1 => addr,
53             sel => ctrl_word(ctrl_mode),
54             y => mem_addr );

```

```

54
55     mem0 : entity work.memory
56         generic map(
57             nbits_addr => K,
58             width => 8 )
59         port map(
60             addr => mem_addr,
61             en => ctrl_word(en_mem),
62             clk => clk,
63             rst => ctrl_word(rst_ou),
64             data_in => dbout,
65             r_w => ctrl_word(ctrl_mode),
66             data_out => data_out);
67
68 end structural;

```

Listing 9.7: Esercizio 9.2 - Unità operativa nodo B

Nodo B - Unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4
5 entity nodoB_cu is
6     Port ( mode : in STD_LOGIC;
7             RDA : in STD_LOGIC;
8             rst : in STD_LOGIC;
9             clk : in std_logic;
10            ctrl_word : out ctrl_word_type);
11 end nodoB_cu;
12
13 architecture Behavioral of nodoB_cu is
14
15     type state_type is (RESET, IDLE, RECEIVE, INCR, READ_MEM);
16     signal state : state_type := RESET;
17     signal next_state : state_type;
18
19 begin
20     process(clk)
21     begin
22         if clk'event and clk = '0' then
23             if rst = '1' then

```

```

24         state <= RESET;
25     else
26         state <= next_state;
27     end if;
28 end if;
29 end process;
30
31 process(state,mode,RDA)
32 begin
33     case state is
34         when RESET =>
35             ctrl_word <= "00001";
36             next_state <= IDLE;
37
38         when IDLE =>
39             ctrl_word <= "00000";
40             if mode = '1' then
41                 next_state <= READ_MEM;
42             elsif RDA = '1' then
43                 next_state <= RECEIVE;
44             else
45                 next_state <= IDLE;
46             end if;
47
48         when RECEIVE =>
49             ctrl_word <= "10100";
50             next_state <= INCR;
51
52         when INCR =>
53             ctrl_word <= "01000";
54             next_state <= IDLE;
55
56         when READ_MEM =>
57             ctrl_word <= "00110";
58             if mode = '0' then
59                 next_state <= IDLE;
60             else
61                 next_state <= READ_MEM;
62             end if;
63
64     end case;
65 end process;
66

```

```
67 end Behavioral;
```

Listing 9.8: Esercizio 9.2 - Unità di controllo nodo B

Nodo B

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4
5 entity nodoB is
6     generic(
7         BAUD_DIVIDE_G : positive;          -- f_Rx = f_CLK /
8             ↳ BAUD_DIVIDE_G
9         K : positive );
10    Port ( mode : in STD_LOGIC;
11            addr : in STD_LOGIC_VECTOR (K-1 downto 0);
12            RxD : in STD_LOGIC;
13            clk : in STD_LOGIC;
14            rst : in STD_LOGIC;
15            data_out : out STD_LOGIC_VECTOR (7 downto 0));
16 end nodoB;
17
18
19 architecture structural of nodoB is
20
21     signal cword : ctrl_word_type;
22     signal RDA : std_logic;
23
24 begin
25     ou : entity work.nodoB_ou
26         generic map(
27             BAUD_DIVIDE_G => BAUD_DIVIDE_G,          -- f_Rx = f_CLK
28             ↳ / BAUD_DIVIDE_G
29             K => K )
30         port map(
31             RxD => RxD,
32             clk => clk,
33             addr => addr,
34             ctrl_word => cword,
35             data_out => data_out,
36             RDA => RDA );
37
38     cu : entity work.nodoB_cu
```

```

36     port map(
37         mode => mode,
38         RDA => RDA,
39         rst => rst,
40         clk => clk,
41         ctrl_word => cword);
42
43 end structural;

```

Listing 9.9: Esercizio 9.2 - Nodo B

Sistema complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity sys is
5     generic(
6         BAUD_DIVIDE_G : positive := 651;          -- baud_rate =
7             -- 9600 con f_CLK = 100MHz
8         K : positive := 3;
9         CLK_B_DELAY : time := 0ns );
10    Port ( send : in STD_LOGIC;
11            mode : in STD_LOGIC;
12            addr : in std_logic_vector (K-1 downto 0);
13            rst : in STD_LOGIC;
14            clk : in STD_LOGIC;
15            data_out : out STD_LOGIC_VECTOR (7 downto 0));
16
17 end sys;
18
19 architecture structural of sys is
20
21     constant BAUD_RATE_G : positive := 16 *
22         BAUD_DIVIDE_G;      -- f_Tx = f_Rx/16
23     signal data_line : std_logic;
24     signal clk_B : std_logic;
25
26 begin
27     clk_B <= clk after CLK_B_DELAY;
28
29     nodoA : entity work.nodoA
30         generic map(

```

```

28      BAUD_RATE_G => BAUD_RATE_G,           -- f_Tx = f_CLK
29      --> / BAUD_RATE_G
30      K => K )
31  port map(
32      send => send,
33      rst => rst,
34      clk => clk,
35      TxD => data_line );
36
37  nodoB : entity work.nodoB
38  generic map(
39      BAUD_DIVIDE_G => BAUD_DIVIDE_G,       -- f_Rx = f_CLK
40      --> / BAUD_DIVIDE_G
41      K => K )
42  port map(
43      mode => mode,
44      addr => addr,
45      RxD => data_line,
46      clk => clk_B,
47      rst => rst,
48      data_out => data_out);
49
50 end structural;

```

Listing 9.10: Esercizio 9.2 - Sistema complessivo

9.4 Simulazione

Ogni singolo componente del progetto è stato testato attraverso un testbench ad-hoc. Tuttavia, per questioni di brevità espositiva, si riportano unicamente i risultati delle simulazioni dei sistemi complessivi dell'esercizio 9.1 e 9.2, rimandando il lettore ai progetti Vivado, qualora quest'ultimo avesse la curiosità di visionare i testbench qui non riportati.

9.4.1 Testbench esercizio 9.1

```
1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity sys_tb is
5     -- Port();
6 end sys_tb;
7
8 architecture behavioral of sys_tb is
9
10    constant BAUD_DIVIDE_G : positive := 2;
11    constant CLK_PERIOD : time := 1ns;
12
13    signal data_in : std_logic_vector(7 downto 0);
14    signal data_out : std_logic_vector(7 downto 0);
15    signal rst : std_logic;
16    signal clk : std_logic := '0';
17    signal write : std_logic;
18
19 begin
20     dut : entity work.sys
21     generic map(
22         BAUD_DIVIDE_G => BAUD_DIVIDE_G)
23     port map(
24         data_in => data_in,
25         write => write,
26         rst => rst,
27         clk => clk,
28         data_out => data_out );
29
30     clk_gen : process
31     begin
32         wait for CLK_PERIOD/2;
33         clk <= not clk;
```

```

34      end process;
35
36      test : process
37      begin
38          wait for CLK_PERIOD/4;
39          rst <= '1';
40          data_in <= x"77";
41          write <= '0';
42          wait for CLK_PERIOD;
43
44          rst <= '0';
45          wait for CLK_PERIOD;
46          write <= '1';
47          wait for CLK_PERIOD;
48          write <= '0';
49
50          wait on data_out;
51          assert data_out = x"77" report "Error" severity
52              => failure;
53
54      end process;
55
56  end behavioral;

```

Listing 9.11: Esercizio 9.1 - Testbench del sistema complessivo

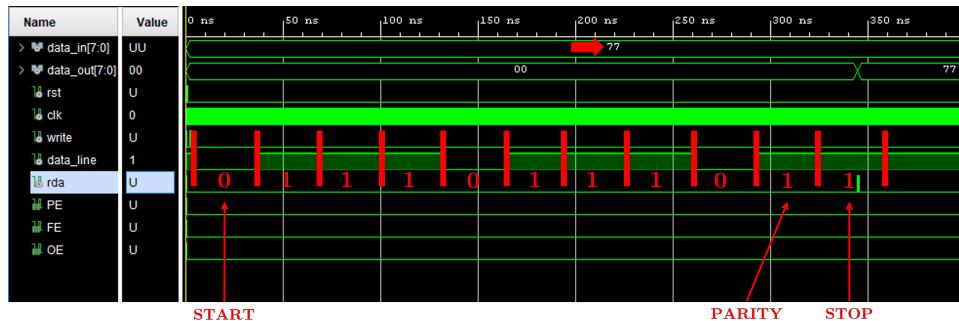


Figura 9.8: Esercizio 9.1 - Risultati della simulazione

9.4.2 Testbench esercizio 9.2

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;

```

```

3  use IEEE.numeric_std.all;
4
5  entity sys_tb is
6  --  Port ( );
7  end sys_tb;
8
9  architecture Behavioral of sys_tb is
10    constant CLK_PERIOD : time := 0.5ns;
11    constant BAUD_DIVIDE_G : positive := 2;
12
13    constant K : positive := 3;
14    constant depth : positive := 2**K;
15    subtype word is std_logic_vector(7 downto 0);
16    type rom_type is array(0 to depth-1) of word;
17    constant oracle : rom_type := (
18      x"01",
19      x"02",
20      x"04",
21      x"08",
22      x"10",
23      x"20",
24      x"40",
25      x"80" );
26
27    signal send : std_logic;
28    signal mode : std_logic;
29    signal addr : std_logic_vector(2 downto 0);
30    signal rst : std_logic;
31    signal clk : std_logic := '0';
32    signal data_out : std_logic_vector(7 downto 0);
33
34 begin
35   dut : entity work.sys
36   generic map(
37     BAUD_DIVIDE_G => BAUD_DIVIDE_G,      -- f_Rx = f_CLK /
38     -- BAUD_DIVIDE_G
39     K => K,                                -- f_Tx = f_Rx / 16
40     CLK_B_DELAY => Ons )
41   port map(
42     send => send,
43     mode => mode,
44     addr => addr,
45     rst => rst,
46     clk => clk,

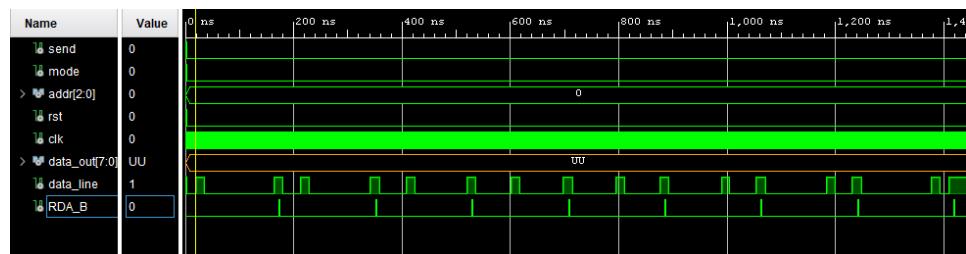
```

```

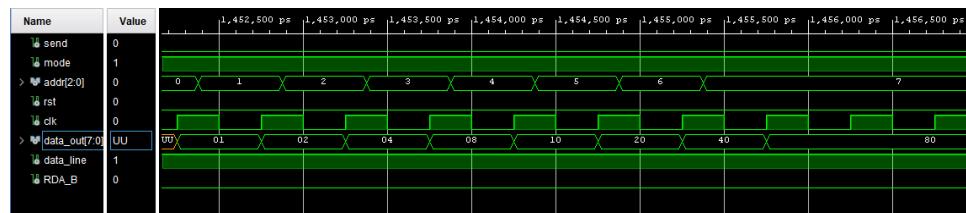
46         data_out => data_out );
47
48 clk_gen : process
49 begin
50     wait for CLK_PERIOD/2;
51     clk <= not clk;
52 end process;
53
54 test : process
55 begin
56     wait for CLK_PERIOD/4;
57
58     mode <= '0';
59     addr <= (others => '0');
60     send <= '0';
61     rst <= '1';
62     wait for CLK_PERIOD;
63
64     rst <= '0';
65     wait for CLK_PERIOD;
66
67     send <= '1';
68     wait for CLK_PERIOD;
69     send <= '0';
70
71     wait for 2900*CLK_PERIOD;      -- aspetto che finisca la
72     -- trasmissione
73     mode <= '1';
74     for i in 0 to depth-1 loop
75         addr <=
76             -- std_logic_vector(to_unsigned(i,addr'length));
77         wait until data_out'active;
78         assert data_out = oracle(i) report "Error" severity
79             failure;
80         wait for CLK_PERIOD/4;
81     end loop;
82
83     wait;
84 end process;
85
86 end Behavioral;

```

Listing 9.12: Esercizio 9.2 - Testbench del sistema complessivo



(a) Modalità 0 - Ricezione



(b) Modalità 1 - Display

Figura 9.9: Esercizio 9.2 - Risultati della simulazione

Capitolo 10

Switch multistadio

10.1 Traccia

Esercizio 7.1 Progettare ed implementare in VHDL uno switch multistadio secondo il modello omega network. Lo switch progettato deve operare come segue:

- a) Lo switch deve consentire lo scambio di messaggi di 2 bit ciascuno da un nodo sorgente a un nodo destinazione in un rete con 4 nodi, implementando uno schema a priorità fissa fra i nodi (ed. nodo 1 più prioritario, con priorità decrescenti fino al nodo 4).
- b) (Opzionale) rimuovendo l'ipotesi di lavorare secondo uno schema a priorità fra i nodi e considerando una rete di 8 nodi, lo switch deve gestire eventuali conflitti generati da collisioni secondo un meccanismo a scelta (ad es. perdendo uno dei messaggi in conflitto).
- c) (Opzionale) Si implementi un protocollo di handshaking semplice regolato da una coppia di segnali (pronto a inviare/pronto a ricevere) per l'invio di ciascun messaggio fra due nodi.

10.2 Soluzione

Per l'esercizio 10 si vuole realizzare uno switch multistadio secondo un modello omega network composto da 4 nodi. Uno switch multistadio è un dispositivo utilizzato per connettere N sorgenti a N destinazioni. L'obiettivo principale di un switch multistadio è migliorare le prestazioni in una rete di interconnessioni rispetto agli switch a singolo stadio o a connessione diretta (figura 10.1).

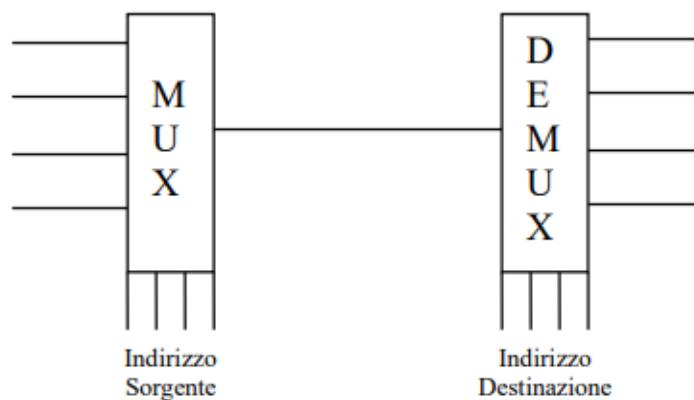


Figura 10.1: Switch a singolo stadio

In particolare, per la creazione di uno switch a connessione diretta, è necessario avere due informazioni fondamentali: l'indirizzo del nodo sorgente e quello del nodo destinazione. Il dispositivo che si vuole realizzare dovrà avere ' N ' nodi in ingresso ed ' N ' nodi in uscita (nell'ipotesi che il numero di nodi della rete sia ' N '), mediante gli indirizzi del sorgente e della destinazione si stabilirà la comunicazione tra i nodi. Per poter implementare tale architettura si deve scomporre il sistema in due sottosistemi diversi, quello di ingresso e quello di uscita. Tale suddivisione viene realizzata mediante l'utilizzo di un dispositivo multiplexer, per quanto riguarda l'ingresso ed un dispositivo demultiplexer per quanto riguarda l'uscita. L'inconveniente隐含的 del meccanismo implementato, è che automaticamente si realizza una mutua esclusione tra i nodi dovuta al fatto che, potrà essere attivo un solo collegamento per volta. Per ovviare a tale problema, si realizzano appunto degli switch che si basano sull'utilizzo di più stadi intermedi. Un esempio di architettura di interconnessione multistadio molto interessante è la **omega network**. Una omega network $N \times N$ consiste in $\log_2 N$ stadi identici che sfruttano un'interconnessione fra i nodi basata su un algoritmo, chiamato **perfect shuffling**. Tale algoritmo nasce dal gioco delle carte che consiste nel mischiare le carte dividendo il mazzo in metà uguali e interlacciadole perfettamente, in modo da ritrovarsi con la prima carta della metà sinistra, seguita dalla prima carta della metà destra e così via. Continuando a mischiare in questo modo, dopo un certo numero di mischiate ($\log_2 N$) il mazzo ritornerà alla

situazione di partenza (figura 10.2). Questo algoritmo può essere sfruttato per determinare gli accoppiamenti dei nodi nei singoli stadi in modo da ottimizzare i percorsi.



Figura 10.2: Algoritmo di perfect shuffling

Dunque per la realizzazione della tipologia di uno switch multistadio secondo il modello omega network, si utilizzano $\log_2 N$ stadi, dove in ogni stadio sono presenti $N/2$ switch a singolo stadio (con 2 ingressi e 2 uscite).

La progettazione dell'esercizio completo è stata divisa in due parti principali: l'Unità di Controllo (UC) e l'Unità Operativa (UO). Dove l'unità operativa (figura 10.5) ha la stessa topologia e funzione della rete di interconnessione della omega network 4x4, mentre l'unità di controllo (figura 10.6) riceve i segnali dall'esterno e poi invia i segnali di controllo all'unità operativa. Il progetto è stato risolto attraverso un approccio strutturale, in particolare abbiamo fatto uso dei seguenti componenti base per la realizzazione completa dell'esercizio:

- switch,
- multiplexer,
- demultiplexer,
- arbitro.

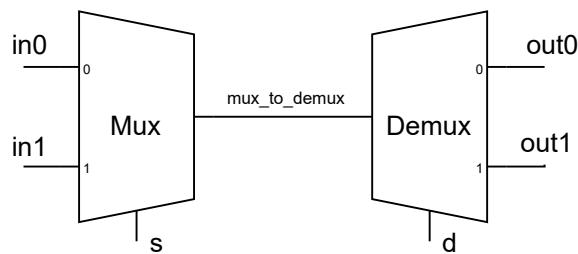


Figura 10.3: Switch

L'unità operativa è composta da quattro switch opportunamente connessi secondo lo schema del perfect shuffling. In particolare, tale componente base è realizzato anch'esso secondo un approccio strutturale come viene mostrato in figura 10.3.

L'unità operativa (10.5) è stata realizzata secondo un approccio strutturale disponendo gli switch in modo tale da realizzare uno schema omega network 4x4. L'instradamento dei messaggi viene stabilito sulla base dell'indirizzo di destinazione (d). Si parte dal bit più significativo dell'indirizzo di destinazione, che decide quale uscita prendere nello switch interessato:

- se il bit è '0' => uscita superiore
- se il bit è '1' => uscita inferiore

L'unità di controllo (figura 10.6) è composta da un multiplexer e un demultiplexer (che compongono praticamente uno switch a singolo stadio) i cui segnali di selezione sono forniti dal componente arbitro. I messaggi x_i in ingresso all'unità di controllo sono strutturati come in figura 10.4, ovvero i primi due bit più significativi rappresentano il messaggio da trasmettere mentre i restanti bit indicano rispettivamente l'indirizzo di destinazione (il quale riceverà il messaggio) e l'indirizzo sorgente (il quale invierà il messaggio). L'arbitro è utile per realizzare uno schema a priorità fissa, ovvero uno solo dei nodi (quello a valore di indirizzo più elevato) è abilitato alla trasmissione del messaggio. Questo è uno dei modi per poter gestire il problema di possibili conflitti, ovvero la situazione per cui ad esempio due nodi di ingresso vogliono parlare con lo stesso nodo di uscita.

	2 bit	2 bit	2 bit
	messaggio	destinazione	sorgente

Figura 10.4: Messaggio di ingresso

Le soluzioni possibili per gestire i conflitti sono quattro, ovvero:

- *blocco*, soluzione vista poc'anzi con l'assegnazione delle priorità,
- *perdita del messaggio*, ovvero uno dei due segnali viene perso, mentre l'altro procede verso la destinazione
- *re-instradamento*, ovvero si stabilisce un percorso alternativo a quello bloccato
- *cut-through*, ovvero consiste nell'adottare una filosofia store e forward, dove i messaggi che non possono essere inoltrati vengono bufferizzati finché c'è spazio.

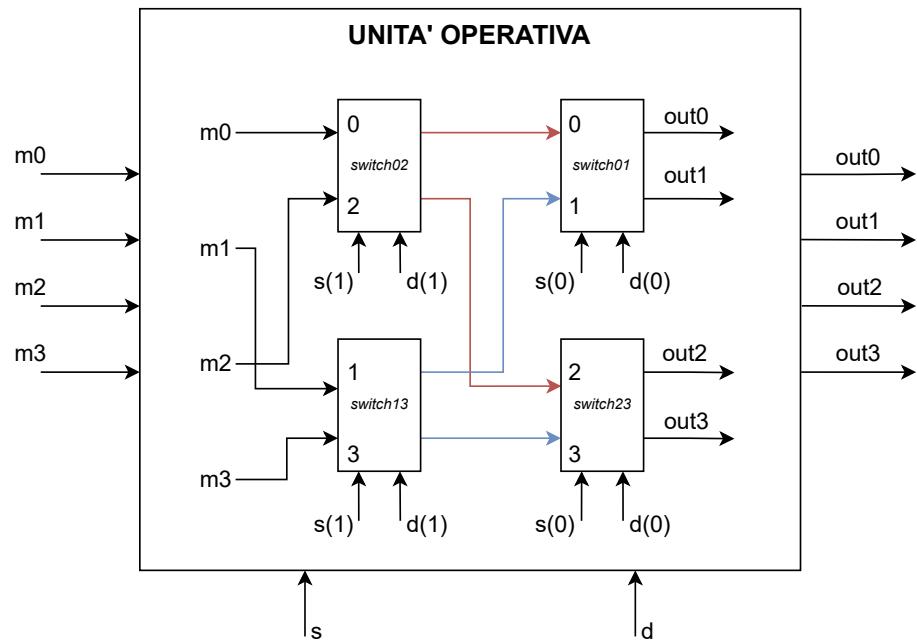


Figura 10.5: Switch multistadio: unità operativa

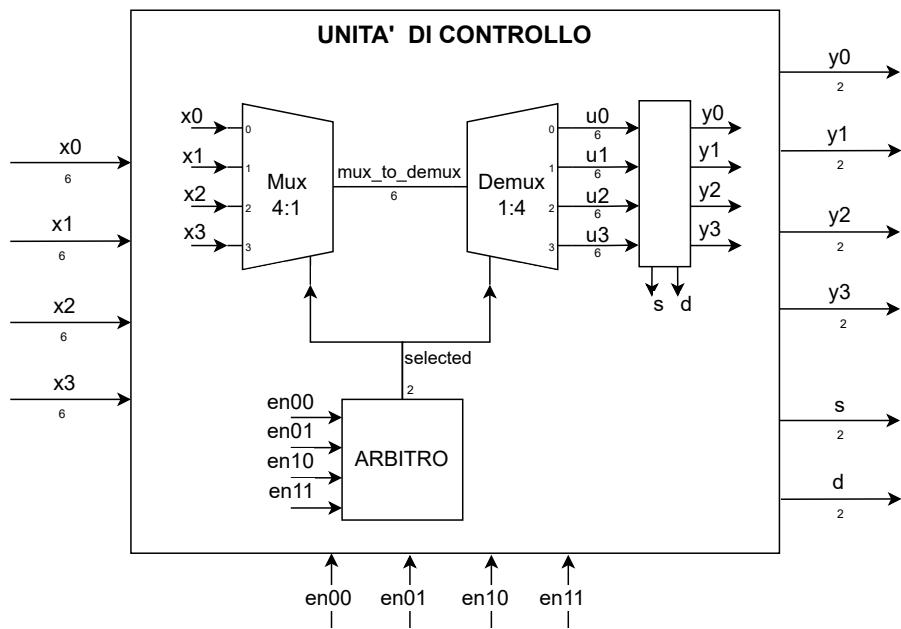


Figura 10.6: Switch multistadio: unità di controllo

10.3 Codice

10.3.1 Switch

Lo switch, componente base del progettto, è stato realizzato con un approccio strutturale, in particolare, come mostra anche il listato 10.1, è composto da un multiplexer 2:1 e un demultiplexer 1:2.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity switch is
5     Port ( in0 : in STD_LOGIC_VECTOR(1 downto 0);
6             in1 : in STD_LOGIC_VECTOR(1 downto 0);
7             s : in STD_LOGIC;
8             d : in STD_LOGIC;
9             out0 : out STD_LOGIC_VECTOR(1 downto 0);
10            out1 : out STD_LOGIC_VECTOR(1 downto 0));
11 end switch;
12
13 architecture Structural of switch is
14
15 signal mux_to_demux : STD_LOGIC_VECTOR(1 downto 0);
16
17 begin
18     mux: entity work.mux2to1 generic map( N=>2)
19         port map(x1 => in0, x2 =>in1, s => s, y => mux_to_demux);
20
21     demux: entity work.demux1to2 generic map(N=>2)
22         port map(x => mux_to_demux, s => d, y0 => out0, y1 =>
23             out1);
24 end Structural;
```

Listing 10.1: Switch

10.3.2 Unità operativa

L'unità operativa anch'essa è stata realizzata con un approccio strutturale, in particolare è composta da quattro switch disposti secondo uno schema omega network, composto da due stadi che sfruttano un'interconnessione fra i nodi (in totale 4) basata sul perfect shuffling. Ad ogni input corrisponde un solo percorso per arrivare ad un determinato output. Il problema delle

collisioni fra messaggi che viaggiano sullo stesso percorso ma che provengono da indirizzi differenti è rimandato all'unità di controllo.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity unita_operativa is
6     Port ( m0 : in STD_LOGIC_VECTOR(1 downto 0);
7             m1 : in STD_LOGIC_VECTOR(1 downto 0);
8             m2 : in STD_LOGIC_VECTOR(1 downto 0);
9             m3 : in STD_LOGIC_VECTOR(1 downto 0);
10            s : in STD_LOGIC_VECTOR(1 downto 0);
11            d : in STD_LOGIC_VECTOR(1 downto 0);
12            out0 : out STD_LOGIC_VECTOR(1 downto 0);
13            out1 : out STD_LOGIC_VECTOR(1 downto 0);
14            out2 : out STD_LOGIC_VECTOR(1 downto 0);
15            out3 : out STD_LOGIC_VECTOR(1 downto 0));
16 end unita_operativa;
17
18 architecture Structural of unita_operativa is
19
20 signal sw_to_sw0 : std_logic_vector(1 downto 0);
21 signal sw_to_sw1 : std_logic_vector(1 downto 0);
22 signal sw_to_sw2 : std_logic_vector(1 downto 0);
23 signal sw_to_sw3 : std_logic_vector(1 downto 0);
24
25
26
27 begin
28     switch02: entity work.switch port map(
29         in0 => m0, in1 => m2, s => s(1), d => d(1), out0 =>
30         sw_to_sw0, out1 => sw_to_sw2
31     );
32
33     switch01: entity work.switch port map(
34         in0 => sw_to_sw0, in1 => sw_to_sw1, s => s(0), d =>
35         d(0), out0 => out0, out1 => out1
36     );
37
38     switch13: entity work.switch port map(
39         in0 => m1, in1 => m3, s => s(1), d => d(1), out0 =>
40         sw_to_sw1, out1 => sw_to_sw3
```

```

38     );
39
40     switch23: entity work.switch port map(
41         in0 => sw_to_sw2, in1 => sw_to_sw3, s => s(0), d =>
42         d(0), out0 => out2, out1 => out3
43     );
44 end Structural;

```

Listing 10.2: Unità operativa: omega network

10.3.3 Arbitro

L'arbitro è stato descritto a livello dataflow, ricevendo in ingresso quattro segnali di selezioni e restituendo in uscita un vettore di 2 bit che codifica il porto abilitato. Nella nostra configurazione, la massima priorità è assegnata al nodo con il valore di indirizzo più elevato, ossia il nodo all'indirizzo "11" (nodo 3).

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity arbitro is
5     Port ( en00 : in STD_LOGIC;
6             en01 : in STD_LOGIC;
7             en10 : in STD_LOGIC;
8             en11 : in STD_LOGIC;
9             selected : out STD_LOGIC_VECTOR(1 downto 0));
10 end arbitro;
11
12 architecture dataflow of arbitro is
13
14 begin
15     selected <= "11" when en11 = '1' else
16         "10" when en10 = '1' else
17         "01" when en01 = '1' else
18         "00" when en00 = '1' else
19         "--";
20
21 end dataflow;

```

Listing 10.3: Arbitro

10.3.4 Unità di controllo

L'unità di controllo invece è costituita oltre che dall'arbitro, anche da uno switch diretto, il quale a seconda del segnale *selected* (che rappresenta l'indirizzo del nodo prioritario) abilita una sorgente a trasmettere il messaggio ad un nodo destinazione. Inoltre, l'unità di controllo è composta anche da un'opportuna logica che ha il compito di estrapolare dal messaggio di ingresso x_i i relativi indirizzi sorgente e destinazione ed il messaggio da trasmettere. Tali segnali estrapolati saranno poi trasmessi in ingresso all'unità operativa.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_controllo is
5     Port ( x0 : in STD_LOGIC_VECTOR(5 downto 0); -- x => // 2
6            ↵ bit sorgente/ 2 bit destinazione / 2 bit messaggio // 
7             x1 : in STD_LOGIC_VECTOR(5 downto 0);
8             x2 : in STD_LOGIC_VECTOR(5 downto 0);
9             x3 : in STD_LOGIC_VECTOR(5 downto 0);
10            en00 : in STD_LOGIC;
11            en01 : in STD_LOGIC;
12            en10 : in STD_LOGIC;
13            en11 : in STD_LOGIC;
14            y0 : out STD_LOGIC_VECTOR(1 downto 0);
15            y1 : out STD_LOGIC_VECTOR(1 downto 0);
16            y2 : out STD_LOGIC_VECTOR(1 downto 0);
17            y3 : out STD_LOGIC_VECTOR(1 downto 0);
18            s : out STD_LOGIC_VECTOR(1 downto 0);
19            d : out STD_LOGIC_VECTOR(1 downto 0));
20
21 end unita_controllo;
22
23 architecture Structural of unita_controllo is
24
25 signal selected : std_logic_vector(1 downto 0);
26 signal mux_to_demux : std_logic_vector(5 downto 0);
27 signal u0 : std_logic_vector(5 downto 0);
28 signal u1 : std_logic_vector(5 downto 0);
29 signal u2 : std_logic_vector(5 downto 0);
30 signal u3 : std_logic_vector(5 downto 0);
31
32 begin
33     mux: entity work.mux4to1 port map(
```

```

34         x0 => x0, x1 => x1, x2 => x2, x3 => x3, s => selected,
35         → y => mux_to_demux
36     );
37
38     demux: entity work.demux1to4 port map(
39         x => mux_to_demux, s => selected, y0 => u0, y1 => u1,
40         → y2 => u2, y3 => u3
41     );
42
43     arbitro: entity work.arbitro port map(
44         en00 => en00, en01 => en01, en10 => en10, en11 => en11,
45         → selected => selected
46     );
47
48     y0 <= u0(5 downto 4);
49     y1 <= u1(5 downto 4);
50     y2 <= u2(5 downto 4);
51     y3 <= u3(5 downto 4);
52
53     d <= u0 (3 downto 2) when selected ="00" else
54         u1 (3 downto 2) when selected ="01" else
55         u2 (3 downto 2) when selected ="10" else
56         u3 (3 downto 2) when selected ="11" else
57             "--";
58
59     s <= selected;
60
61 end Structural;

```

Listing 10.4: Unità di controllo

10.3.5 Switch multistadio

Infine il blocco complessivo lo switch multistadio, componente traccia del nostro esercizio, viene realizzato seguendo un approccio strutturale compонendo unità di controllo e unità operativa, come viene mostrato al listato 10.5.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity switch_multistadio is
5     Port ( x0 : in STD_LOGIC_VECTOR(5 downto 0);

```

```

6      x1 : in STD_LOGIC_VECTOR(5 downto 0);
7      x2 : in STD_LOGIC_VECTOR(5 downto 0);
8      x3 : in STD_LOGIC_VECTOR(5 downto 0);
9      s0 : in STD_LOGIC;
10     s1 : in STD_LOGIC;
11     s2 : in STD_LOGIC;
12     s3 : in STD_LOGIC;
13     out0 : out STD_LOGIC_VECTOR(1 downto 0);
14     out1 : out STD_LOGIC_VECTOR(1 downto 0);
15     out2 : out STD_LOGIC_VECTOR(1 downto 0);
16     out3 : out STD_LOGIC_VECTOR(1 downto 0));
17 end switch_multistadio;
18
19 architecture structural of switch_multistadio is
20
21 signal sc_0 : STD_LOGIC_VECTOR(1 downto 0);
22 signal sc_1 : STD_LOGIC_VECTOR(1 downto 0);
23 signal sc_2 : STD_LOGIC_VECTOR(1 downto 0);
24 signal sc_3 : STD_LOGIC_VECTOR(1 downto 0);
25
26 signal sorg : STD_LOGIC_VECTOR(1 downto 0);
27 signal dest : STD_LOGIC_VECTOR(1 downto 0);
28
29 begin
30
31 uc: entity work.unita_controllo port map(
32     x0=> x0, x1 => x1, x2 => x2, x3 => x3, en00 => s0, en01
33     => s1, en10 => s2, en11 => s3,
34     y0 => sc_0, y1 => sc_1, y2 => sc_2, y3 => sc_3, s =>
35     => sorg, d => dest);
36
37 uo: entity work.unita_operativa port map(
38     m0 => sc_0, m1 => sc_1, m2 => sc_2, m3 => sc_3, s =>
39     => sorg, d => dest,
40     out0 => out0, out1 => out1, out2 => out2, out3 =>
41     => out3);
42
43 end structural;

```

Listing 10.5: Switch multistadio

10.4 Simulazione

10.4.1 Switch

In questa sezione è presentato il test bench del componente base del nostro progetto, ovvero lo switch.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity switch_tb is
5
6 end switch_tb;
7
8 architecture Behavioral of switch_tb is
9 signal in0 : STD_LOGIC_VECTOR(1 downto 0);
10 signal in1 : STD_LOGIC_VECTOR(1 downto 0);
11 signal s : STD_LOGIC;
12 signal d : STD_LOGIC;
13 signal out0 : STD_LOGIC_VECTOR(1 downto 0);
14 signal out1 : STD_LOGIC_VECTOR(1 downto 0);
15
16 begin
17     uut: entity work.switch port map(
18         in0 => in0, in1 => in1,
19         s => s, d => d,
20         out0 => out0, out1 => out1);
21
22     prc_tb: process begin
23         in0 <= "11";
24         in1 <= "10";
25
26         s <= '1';
27         d <= '0';
28         wait for 20 ns;
29         s <= '0';
30         wait for 20 ns;
31         d <= '1';
32         wait;
33     end process;
34
35
36 end Behavioral;
```

Listing 10.6: Test bench switch

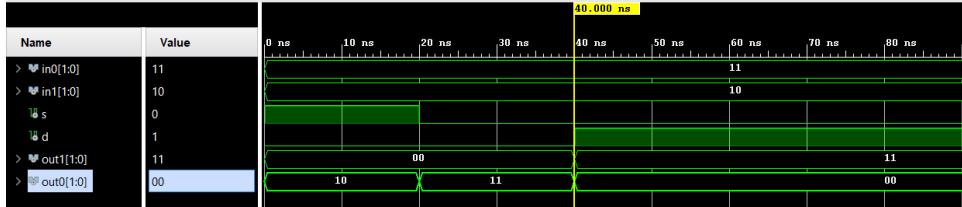


Figura 10.7: Waveform switch

10.4.2 Unità operativa

In questa sezione, invece è presentato il testing della rete di interconnessione omega network, ovvero la nostra unità operativa. Dai risultati ottenuti si evince come ad ogni input corrisponda un solo percorso per arrivare ad un determinato output. Tale percorso è selezionato grazie ai segnali s e d , che rappresentano rispettivamente indirizzo nodo sorgente e indirizzo nodo destinazione.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity unita_operativa_tb is
5 end unita_operativa_tb;
6
7 architecture Behavioral of unita_operativa_tb is
8
9 signal m0 : STD_LOGIC_VECTOR(1 downto 0);
10 signal m1 : STD_LOGIC_VECTOR(1 downto 0);
11 signal m2 : STD_LOGIC_VECTOR(1 downto 0);
12 signal m3 : STD_LOGIC_VECTOR(1 downto 0);
13 signal s : STD_LOGIC_VECTOR(1 downto 0);
14 signal d : STD_LOGIC_VECTOR(1 downto 0);
15 signal out0 : STD_LOGIC_VECTOR(1 downto 0);
16 signal out1 : STD_LOGIC_VECTOR(1 downto 0);
17 signal out2 : STD_LOGIC_VECTOR(1 downto 0);
18 signal out3 : STD_LOGIC_VECTOR(1 downto 0);
19
20 begin
21
22     uut: entity work.unita_operativa port map(
23         m0 => m0, m1 => m1, m2 => m2, m3 => m3,

```

```

24         s => s, d => d, out0 => out0, out1 => out1, out2 =>
25             → out2, out3 => out3
26     );
27
28     prc_tb: process begin
29         m0 <= "11";
30         m1 <= "10";
31         m2 <= "01";
32         m3 <= "10";
33
34         s <= "00";
35         d <= "10";
36         wait for 20 ns;
37         s <= "01";
38         d <= "00";
39
40         wait for 20 ns;
41         s <= "10";
42         d <= "11";
43
44         wait for 20 ns;
45         s <= "11";
46         d <= "01";
47
48         wait;
49
50
51 end Behavioral;

```

Listing 10.7: Test bench unità operativa

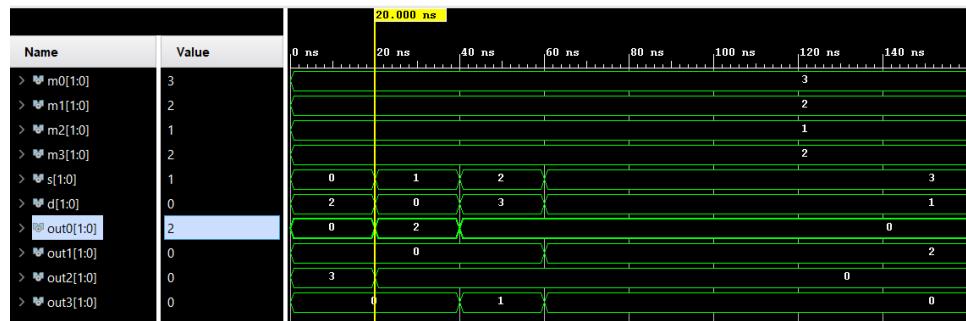


Figura 10.8: Waveform unità operativa

10.4.3 Switch multistadio

Infine viene presentato il testing del sistema complessivo. In particolare, viene mostrato anche come viene risolto il problema della concorrenza fra più nodi, consentendo, secondo uno schema a priorità, solo ad uno di essi di comunicare con la destinazione.

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity switch_multistadio_tb is
5 end switch_multistadio_tb;
6
7 architecture Behavioral of switch_multistadio_tb is
8
9 signal x0 : STD_LOGIC_VECTOR(5 downto 0);
10 signal x1 : STD_LOGIC_VECTOR(5 downto 0);
11 signal x2 : STD_LOGIC_VECTOR(5 downto 0);
12 signal x3 : STD_LOGIC_VECTOR(5 downto 0);
13 signal s0 : STD_LOGIC;
14 signal s1 : STD_LOGIC;
15 signal s2 : STD_LOGIC;
16 signal s3 : STD_LOGIC;
17 signal out0 : STD_LOGIC_VECTOR(1 downto 0);
18 signal out1 : STD_LOGIC_VECTOR(1 downto 0);
19 signal out2 : STD_LOGIC_VECTOR(1 downto 0);
20 signal out3 : STD_LOGIC_VECTOR(1 downto 0);
21
22 begin
23
24     uut: entity work.switch_multistadio port map(
25         x0 => x0, x1 => x1, x2 => x2, x3 => x3,
26         s0 => s0, s1 => s1, s2 => s2, s3 => s3,
27         out0 => out0, out1 => out1, out2 => out2, out3 =>
28             → out3);
29
30     prc_tb: process begin
31         x0 <= "11" & "10" & "00"; -- // 2 bit messaggio / 2 bit
32             → destinazione / 2 bit sorgente //
33         x1 <= "10" & "00" & "01";
34         x2 <= "01" & "11" & "10";
35         x3 <= "10" & "01" & "11";
```

```

36
37     s0 <= '1';
38     s1 <= '1';
39     s2 <= '1';
40     s3 <= '1';
41     wait for 20 ns;
42
43     s3 <= '0';
44     wait for 20 ns;
45     s2 <= '0';
46     wait for 20 ns;
47     s1 <= '0';
48     wait;
49 end process;
50
51 end Behavioral;

```

Listing 10.8: Test bench switch multistadio



Figura 10.9: Waveform switch multistadio

Capitolo 11

Moltiplicatore di Booth

11.1 Traccia

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

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

Opzionalmente, la macchina implementata può essere sintetizzata su FPGA e testata mediante l'utilizzo dei dispositivi di input/output (switch, bottoni, led, display) presenti sulla board di sviluppo in dotazione al gruppo. La modalità di utilizzo degli stessi è a completa discrezione degli studenti.

Nel presente elaborato, si è scelto di implementare il *moltiplicatore di Booth*.

11.2 Soluzione

Il moltiplicatore di Booth è una macchina aritmetica sequenziale, in grado di eseguire il prodotto tra due numeri interi relativi rappresentati in complemento a due. Il suo principio di funzionamento si basa sull'algoritmo manuale della moltiplicazione (procedimento in colonna), come l'algoritmo di Robertson, ma si differenzia da quest'ultimo in quanto sfrutta una particolare codifica dell'input per rendere più omogenei i passi dell'algoritmo, e ridurre il numero di operazioni. Dato un numero intero relativo X espresso nella sua rappresentazione in complemento a due su n bit come:

$$x_{n-1}x_{n-2}\dots x_0 \implies X = -x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_0 \cdot 2^0$$

Si costruisca una successione di n elementi sì fatti:

$$\begin{aligned} y_0 &= -x_0 \\ y_1 &= -x_1 + x_0 \\ y_2 &= -x_2 + x_1 \\ &\dots \\ y_{n-1} &= -x_{n-1} + x_{n-2} \end{aligned}$$

e si consideri il numero Y rappresentato in notazione posizionale, con base due, come $y_{n-1}y_{n-2}\dots y_0$. Attenzione che il numero $y_{n-1}y_{n-2}\dots y_0$ non è un numero binario, in quanto $y_i \in \{-1, 0, 1\}$. Il numero Y varrà quindi:

$$\begin{aligned} Y &= \sum_{i=0}^{n-1} y_i \cdot 2^i = y_{n-1} \cdot 2^{n-1} + y_{n-2} \cdot 2^{n-2} + \dots + y_1 \cdot 2 + y_0 = \\ &= -x_{n-1} \cdot 2^{n-1} + x_{n-2} \cdot 2^{n-2} + \dots + x_1 \cdot 2 + x_0 = \\ &= -x_{n-1} \cdot 2^{n-1} + \sum_{i=0}^{n-2} x_i \cdot 2^i = X \end{aligned}$$

Dunque il numero $y_{n-1}y_{n-2}\dots y_0$ non è altro che una codifica alternativa del valore X , detta appunto *codifica di Booth*. Ogni termine y_i della codifica di Booth è ottenuto a partire da una coppia di cifre binarie della rappresentazione in complemento a due del numero X da rappresentare, secondo la relazione:

$$y_i = -x_i + x_{i-1}$$

Per rendere il calcolo delle cifre y_i omogeneo, anche nel caso in cui $i = 0$, si introduce una cifra binaria fittizia $x_{-1} = 0$. In tabella 11.1 è riportata l'equivalenza tra una coppia di cifre in complemento a due e la loro codifica di Booth.

Tabella 11.1: Codifica di Booth

x_i	x_{i-1}	y_i
0	0	0
0	1	1
1	0	-1
1	1	0

Dati due interi relativi X e Y si vuole calcolare il loro prodotto P . Detto Y il moltiplicando e X il moltiplicatore, si supponga che di X sia nota la codifica di Booth, pari a $x_{n-1}x_{n-2}\dots x_0$; dunque:

$$X = \sum_{i=0}^{n-1} x_i \cdot 2^i \quad , x_i \in \{-1, 0, 1\}$$

Effettuando quindi il prodotto secondo l'algoritmo manuale, come nel caso dell'algoritmo di Robertson, si ottiene una sequenza di prodotti parziali P_i con $i = 0, \dots, n$ calcolati secondo la relazione:

$$P_{i+1} = (P_i + x_i \cdot Y) \cdot 2^{-1} \quad , x_i \in \{-1, 0, 1\}$$

dove $P_0 = 0$. Dunque,

- se $x_i = 1$, si esegue una somma e poi uno shift verso destra;
- se $x_i = -1$, si esegue una differenza e poi uno shift verso destra;
- se $x_i = 0$, si esegue solo lo shift verso destra.

Conoscendo la relazione che lega le cifre della rappresentazione in complemento a due con quelle della codifica di Booth, non è necessario calcolare le seconde in maniera esplicita, ma possiamo lavorare con le prime considerandole in coppia. Quindi, ridefinendo x_i come l' i -esima cifra della rappresentazione in complemento di X , l'algoritmo diventa:

- se $x_i x_{i-1} = 01$, si esegue una somma e poi uno shift verso destra;
- se $x_i x_{i-1} = 10$, si esegue una differenza e poi uno shift verso destra;
- se $x_i x_{i-1} = 00$ oppure $x_i x_{i-1} = 11$, si esegue solo lo shift verso destra;

ricordandosi sempre di aggiungere una cifra fittizia iniziale $x_{-1} = 0$.

Per realizzare un circuito che esegua le operazioni appena descritte occorre:

- un registro M di n bit, per memorizzare il moltiplicando;
- un registro Q di $n + 1$ bit, per memorizzare il moltiplicatore, più un bit fittizio iniziale;

- uno shift register di $2n$ bit per shiftare verso destra i prodotti parziali, e mantenere il risultato finale;
- un *adder/subtractor* per eseguire le operazioni di addizione e sottrazione;
- un contatore, per contare il numero di passi iterativi dell'algoritmo fino alla conclusione dello stesso;
- un'unità di controllo che governi opportunamente il datapath affinché l'algoritmo sia eseguito correttamente.

Osservando il fatto che ad ogni passo dell'algoritmo una cifra del moltiplicatore può essere scartata, e che il prodotto parziale i -esimo non occupa più di $n + i$ bit, è possibile risparmiare dello spazio memorizzando i prodotti parziali e il moltiplicatore all'interno dello stesso shift register. In particolare, nella parte alta dello shift register (A) saranno memorizzati i prodotti parziali, mentre nelle parti bassa (Q) è inizialmente salvato il moltiplicatore per intero. In figura 11.1 è riportato lo schematico dell'architettura del moltiplicatore di Booth con operandi a 8 bit.

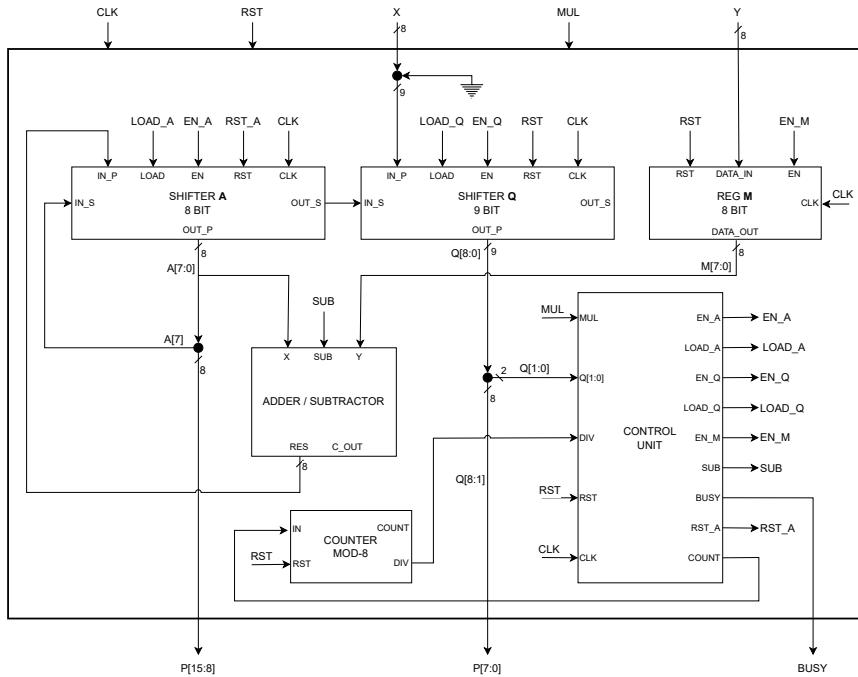


Figura 11.1: Schematico del moltiplicatore di Booth ad 8 bit

Osservando l'architettura del moltiplicatore è importante notare un paio di cose:

- l'operazione di shift verso destra, è un'operazione di shift aritmetico, ovvero con estensione del segno. Dunque durante lo shift viene replicato il bit più significativo, per preservare il segno del prodotto parziale.
- per com'è fatto l'algoritmo di Booth, il risultato dell'adder/subtractor non causerà mai una condizione di overflow/underflow.

In figura 11.2 è riportata una schematizzazione dell'algoritmo di Booth con fattori ad 8 bit.

```

BoothMultiplier: (in:INBUS; OUT:OUTBUS)
register A[7:0],M[7:0],Q[7:-1],COUNT[2:0];
bus INBUS[7:0],OUTBUS[7:0];

BEGIN:      A:=0,COUNT:=0;
INPUT:      M:=INBUS; Q[7:0]:=INBUS;Q[-1]:=0;

SCAN:       if Q[0]Q[-1] = 01
            then A[7:0]:= A[7:0] + M [7:0];
            else if Q[0]Q[-1] = 10
            then A[7:0]:= A[7:0] - M[7:0];

RSHIFT:     A[7]:= A[7], A[6:0].Q:= A.Q[7:0];
INCREMENT:  COUNT:=COUNT+1; go to SCAN;

TEST:       if COUNT<8 then go to SCAN;

OUTPUT:    OUTBUS:=A;
            OUTBUS:=Q[7:0];
END BoothMultiplier;

```

Figura 11.2: Algoritmo del moltiplicatore di Booth ad 8 bit

È l'unità di controllo del moltiplicatore a governare il flusso dell'algoritmo descritto in figura 11.2. Essa è realizzata in logica cablata, ed implementa l'automa a stati finiti rappresentato in figura 11.3.

Questa particolare implementazione del moltiplicatore di Booth, presenta dei segnali aggiuntivi di I/O di cui non si è ancora discusso:

MUL è il segnale in input utilizzato per avviare il calcolo del prodotto. Dev'essere alzato solo dopo aver settato in modo opportuno le linee X e Y.

BUSY è un segnale di output che la macchina utilizza per indicare all'esterno quando essa è già impegnata in un'operazione di calcolo, e quando no.

INPUT: < MUL, Q1, Q0, DIV, RST >
OUTPUT: < EN_A, LOAD_A, EN_Q, LOAD_Q, EN_M, SUB, COUNT, RST_A, BUSY >

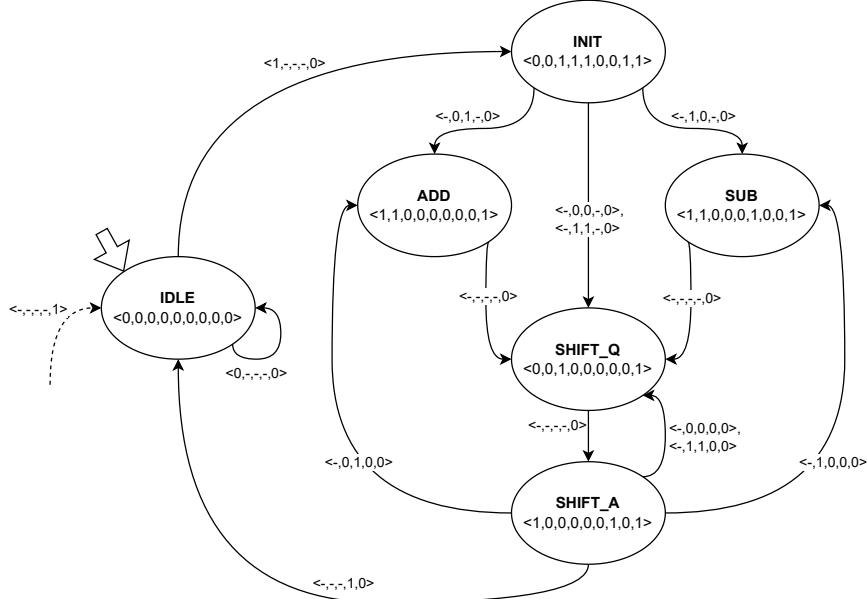


Figura 11.3: Unità di controllo del moltiplicatore di Booth

11.2.1 Adder/Subtractor

Per terminare la discussione sulla soluzione architetturale adottata per implementare il moltiplicatore di Booth, non ci resta che analizzare l'unico componente notevole non ancora analizzato in questo elaborato, ovvero l'*adder/subtractor*. Tale componente, raffigurato in figura 11.4, è composto

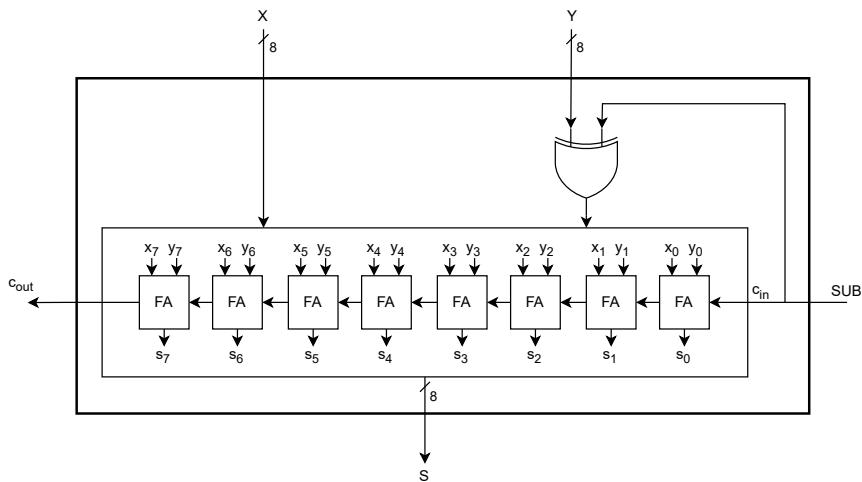


Figura 11.4: Schematico dell'adder/subtractor

da un circuito Ripple Carry Adder (RCA), preceduto da un logica combinatoria che effettua l'operazione di XOR tra ogni bit dell'ingresso Y con il segnale SUB in ingresso. Tale circuito è indicato simbolicamente da una porta XOR. Il segnale SUB è alto nel caso in cui occorra eseguire una sottrazione, e basso nel caso in cui occorra eseguire un'addizione. In particolare, nel caso in cui $SUB = 1$, i bit del numero Y saranno tutti negati a seguito della XOR, e come segnale d'ingresso c_{in} del blocco RCA sarà fornito il valore logico alto.

Il circuito Ripple Carry Adder, invece, è a sua volta composto da una catena di Full Adder collegati in serie attraverso i segnali di riporto, il quale si propaga a partire dal bit meno significativo fino a quello più significativo.

Attenzione: osservando la figura 11.1, si può notare come l'adder/subtractor sia una macchina combinatoria racchiusa tra più macchine sequenziali. Inoltre, osservando l'automa in figura 11.3, in particolare il passaggio di stato da ADD/SUB a SHIFT_Q, si evince che il progettista della macchina si aspetti che il circuito combinatorio sia in grado di ultimare il calcolo e presentare un'uscita stabile in al più un periodo di clock (un periodo e mezzo in realtà considerando che parte operativa e parte di controllo evolvono su fronti opposti). Tale assunzione non è scontata, e dev'essere verificata con un'opportuna analisi dei tempi.

11.3 Codice

11.3.1 Moltiplicatore di Booth

Il moltiplicatore di Booth è composto da un'unità operativa descritta con un approccio strutturale, ed un'unità di controllo descritta secondo un approccio comportamentale. Inoltre è stato definito un package di supporto nel file CU_util.vhd per semplificare la realizzazione dell'unità di controllo.

Unità operativa

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.math_real.all;
4 use work.CU_util.all;
5
6 entity Booth_multiplier is
7     Generic(
8         WIDTH : positive := 8); -- WIDTH has to be a power of 2
9     Port (
10         X : in std_logic_vector(WIDTH-1 downto 0);
11         Y : in std_logic_vector(WIDTH-1 downto 0);
12         multiply : in std_logic;
13         rst : in std_logic;
14         clk : in std_logic;
15         product : out std_logic_vector(2*WIDTH-1 downto 0);
16         busy : out std_logic );
17 end Booth_multiplier;
18
19 architecture Structural of Booth_multiplier is
20     signal sum : std_logic_vector(WIDTH-1 downto 0);
21     signal parallel_out_A : std_logic_vector(WIDTH-1 downto 0);
22     signal rst_A : std_logic;
23     signal serial_in_Q : std_logic;
24     signal parallel_in_Q : std_logic_vector(WIDTH downto 0);
25     signal parallel_out_Q : std_logic_vector(WIDTH downto 0);
26     signal reg_output : std_logic_vector(WIDTH-1 downto 0);
27     signal div : std_logic;
28     signal ctrl_word : ctrl_word_type;
29 begin
30     busy <= ctrl_word(busy_CU);
31     rst_A <= rst or ctrl_word(rst_A_CU);
32
33     product(2*WIDTH-1 downto WIDTH) <= parallel_out_A;
```

```

34     A : entity work.right_shifter
35     generic map(
36         WIDTH => WIDTH )
37     port map(
38         input_p => sum,
39         input_s => parallel_out_A(WIDTH-1),
40         en => ctrl_word(enable_A),
41         load => ctrl_word(load_A),
42         clk => clk,
43         rst => rst_A,
44         output_s => serial_in_Q,
45         output_p => parallel_out_A );
46
47     parallel_in_Q <= X & '0';
48     Q : entity work.right_shifter
49     generic map(
50         WIDTH => WIDTH+1 )
51     port map(
52         input_p => parallel_in_Q,
53         input_s => serial_in_Q,
54         en => ctrl_word(enable_Q),
55         load => ctrl_word(load_Q),
56         clk => clk,
57         rst => rst,
58         output_p => parallel_out_Q );
59     product(WIDTH-1 downto 0) <= parallel_out_Q(WIDTH downto
60             ↳ 1);
61
62     M : entity work.register_pp
63     generic map(
64         dim => WIDTH )
65     port map(
66         x => Y,
67         clk => clk,
68         rst => rst,
69         enable => ctrl_word(enable_M),
70         y => reg_output );
71
72     adder : entity work.adder_subtractor
73     generic map(
74         NBIT => WIDTH)
75     port map(
76         a => parallel_out_A,
77         b => reg_output,

```

```

77         sub => ctrl_word(subtract),
78         res => sum );
79
80     counter : entity work.counter
81     generic map(
82         bits => positive(ceil(log2(real(WIDTH)))) )
83     port map(
84         x => ctrl_word(count_in),
85         rst => rst,
86         div => div );
87
88     CU : entity work.control_unit
89     port map(
90         mul => multiply,
91         Q => parallel_out_Q(1 downto 0),
92         div => div,
93         rst => rst,
94         clk => clk,
95         ctrl_word => ctrl_word);
96 end Structural;

```

Listing 11.1: Unità operativa del moltiplicatore di Booth

Unità di controllo

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3  use work.CU_util.all;
4
5  entity control_unit is
6      Port ( mul : in STD_LOGIC;
7              Q : in std_logic_vector(1 downto 0);
8              div : in std_logic;
9              rst : in STD_LOGIC;
10             clk : in std_logic;
11             ctrl_word : out ctrl_word_type);
12 end control_unit;
13
14 architecture Behavioral of control_unit is
15     type state_type is (IDLE, INIT, ADD, SUB, SHIFT_Q,
16                         ↳ SHIFT_A);
17     signal state : state_type := IDLE;
18     signal next_state : state_type;

```

```

18     type cword_map_type is array(state_type) of ctrl_word_type;
19     constant cword_map : cword_map_type := (
20         IDLE => "00000000",
21         INIT => "001110011",
22         ADD => "110000001",
23         SUB => "110001001",
24         SHIFT_Q => "001000001",
25         SHIFT_A => "100000101" );
26 begin
27     seq : process(clk)
28 begin
29     if clk'event and clk = '0' then
30         if rst = '1' then
31             state <= IDLE;
32         else
33             state <= next_state;
34         end if;
35     end if;
36 end process;
37
38 comb : process(state, mul, Q, div)
39 begin
40     case state is
41         when IDLE =>
42             ctrl_word <= cword_map(IDLE);
43             if mul = '1' then
44                 next_state <= INIT;
45             else
46                 next_state <= IDLE;
47             end if;
48
49         when INIT =>
50             ctrl_word <= cword_map(INIT);
51             if Q = "01" then
52                 next_state <= ADD;
53             elsif Q = "10" then
54                 next_state <= SUB;
55             else
56                 next_state <= SHIFT_Q;
57             end if;
58
59         when ADD =>
60             ctrl_word <= cword_map(ADD);
61             next_state <= SHIFT_Q;

```

```

62
63     when SUB =>
64         ctrl_word <= cword_map(SUB);
65         next_state <= SHIFT_Q;
66
67     when SHIFT_Q =>
68         ctrl_word <= cword_map(SHIFT_Q);
69         next_state <= SHIFT_A;
70
71     when SHIFT_A =>
72         ctrl_word <= cword_map(SHIFT_A);
73         if div = '1' then
74             next_state <= IDLE;
75         elsif Q = "01" then
76             next_state <= ADD;
77         elsif Q = "10" then
78             next_state <= SUB;
79         else
80             next_state <= SHIFT_Q;
81         end if;
82     end case;
83 end process;
84
85 end Behavioral;

```

Listing 11.2: Unità di controllo del moltiplicatore di Booth

11.3.2 Adder/Subtractor

L'adder/subtractor, e il circuito RCA in esso contenuto, sono realizzati secondo un approccio strutturale, mentre il Full Adder di base è descritto con un modello dataflow.

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity adder_subtractor is
5     generic(
6         NBIT : positive := 8);
7     Port ( a : in STD_LOGIC_VECTOR (NBIT-1 downto 0);
8            b : in STD_LOGIC_VECTOR (NBIT-1 downto 0);
9            sub : in STD_LOGIC;
10           res : out STD_LOGIC_VECTOR (NBIT-1 downto 0));

```

```

11         c_out : out STD_LOGIC );
12 end adder_subtractor;
13
14 architecture structural of adder_subtractor is
15     signal op2 : std_logic_vector(NBIT-1 downto 0);
16 begin
17     rca : entity work.ripple_carry_adder
18     generic map(
19         nbit => NBIT )
20     port map(
21         a => a,
22         b => op2,
23         c_in => sub,
24         s => res,
25         c_out => c_out );
26
27     b_xor_cin : for i in 0 to NBIT-1 generate
28         op2(i) <= b(i) xor sub;
29     end generate;
30 end structural;

```

Listing 11.3: Adder/Subtractor

Ripple Carry Adder

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity ripple_carry_adder is
5     generic(
6         nbit : positive );
7     Port ( a : in STD_LOGIC_VECTOR (nbit-1 downto 0);
8             b : in STD_LOGIC_VECTOR (nbit-1 downto 0);
9             c_in : in STD_LOGIC;
10            s : out STD_LOGIC_VECTOR (nbit-1 downto 0);
11            c_out : out STD_LOGIC);
12 end ripple_carry_adder;
13
14 architecture structural of ripple_carry_adder is
15     signal carry : std_logic_vector(nbit downto 0);
16 begin
17     carry(0) <= c_in;
18     c_out <= carry(nbit);

```

```

19
20     FA_1toN : for i in 0 to nbit-1 generate
21         FA : entity work.full_adder
22             port map(
23                 a => a(i),
24                 b => b(i),
25                 c_in => carry(i),
26                 s => s(i),
27                 c_out => carry(i+1) );
28     end generate;
29
30 end structural;

```

Listing 11.4: Ripple Carry Adder

Full Adder

```

1 library IEEE;
2 use IEEE.std_logic_1164.all;
3
4 entity full_adder is
5     Port(
6         a : in std_logic;
7         b : in std_logic;
8         c_in : in std_logic;
9         s : out std_logic;
10        c_out : out std_logic );
11    end full_adder;
12
13 architecture dataflow of full_adder is
14
15 begin
16     s <= (a xor b) xor c_in;
17     c_out <= (a and b) or (c_in and (a xor b));
18 end dataflow;

```

Listing 11.5: Full Adder

11.4 Simulazione

In questo paragrafo si riportano i listati dei testbench, e le waveform risultanti, dei tre componenti principali del progetto corrente:

- l'Adder/Subtractor
- l'unità di controllo del moltiplicatore
- il moltiplicatore di Booth nella sua interezza.

I test degli altri componenti non sono qui riportati per brevità.

11.4.1 Adder/Subtractor

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.NUMERIC_STD.ALL;
4
5 entity add_sub_tb is
6 -- Port ( );
7 end add_sub_tb;
8
9 architecture Behavioral of add_sub_tb is
10 constant N : positive := 4;
11 signal a : std_logic_vector(N-1 downto 0);
12 signal b : std_logic_vector(N-1 downto 0);
13 signal sub : std_logic;
14 signal res : std_logic_vector(N-1 downto 0);
15 signal c_out : std_logic;
16 begin
17 dut : entity work.adder_subtractor
18 generic map(
19 NBIT => N)
20 port map(
21 a => a,
22 b => b,
23 sub => sub,
24 res => res,
25 c_out => c_out );
26
27 test : process
28 begin
29 a <= std_logic_vector(to_signed(4,N));
30 b <= std_logic_vector(to_signed(3,N));
```

```

31      sub <= '0';
32      wait for 10ns;
33      assert res = std_logic_vector(to_signed(7,N)) report
34          "Error sum" severity failure;
35      assert c_out = '0' report "Error carry out" severity
36          failure;
37
38      sub <= '1';
39      wait for 10ns;
40      assert res = std_logic_vector(to_signed(1,N)) report
41          "Error sum" severity failure;
42      assert c_out = '1' report "Error carry out" severity
43          failure;
44
45      a <= std_logic_vector(to_signed(-2,N));
46      b <= std_logic_vector(to_signed(2,N));
47      sub <= '0';
48      wait for 10ns;
49      assert res = std_logic_vector(to_signed(0,N)) report
50          "Error sum" severity failure;
51      assert c_out = '1' report "Error carry out" severity
52          failure;
53
54      sub <= '1';
55      wait for 10ns;
56      assert res = std_logic_vector(to_signed(-4,N)) report
57          "Error sum" severity failure;
58      assert c_out = '1' report "Error carry out" severity
59          failure;
60
61      a <= std_logic_vector(to_signed(1,N));
62      b <= std_logic_vector(to_signed(-4,N));

```

```

63      assert c_out = '0' report "Error carry out" severity
64          => failure;
65
66      a <= std_logic_vector(to_signed(-7,N));
67      b <= std_logic_vector(to_signed(-2,N));
68      sub <= '0';
69      wait for 10ns;
70      -- underflow
71      assert res = std_logic_vector(to_signed(-9,N)) report
72          => "Error sum" severity failure;
73      assert c_out = '1' report "Error carry out" severity
74          => failure;
75
76      sub <= '1';
77      wait for 10ns;
78      assert res = std_logic_vector(to_signed(-5,N)) report
79          => "Error sum" severity failure;
80      assert c_out = '0' report "Error carry out" severity
81          => failure;
82
83  end process;
84 end Behavioral;

```

Listing 11.6: Adder/Subtractor - Testbench

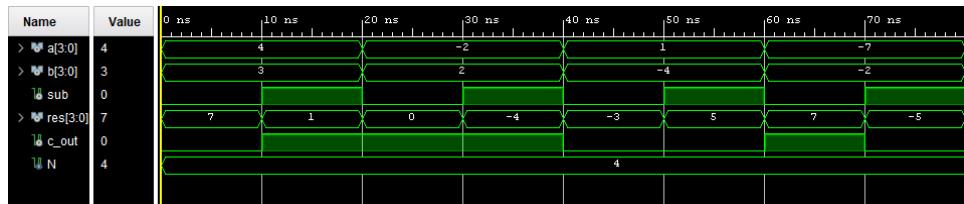


Figura 11.5: Adder/Subtractor - Waveform

In figura 11.5, si può osservare che il componente fornisce sempre il risultato corretto sulla linea `res`. Gli input forniti sono otto, ottenuti combinando le tre condizioni:

- $a > 0$ oppure $a < 0$;
- $b > 0$ oppure $b < 0$;
- $SUB = 0$ oppure $SUB = 1$;

11.4.2 Moltiplicatore di Booth: unità di controllo

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.CU_util.all;
4
5 entity control_unit_tb is
6 -- Port ( );
7 end control_unit_tb;
8
9 architecture Behavioral of control_unit_tb is
10 constant CLK_PERIOD : time := 10ns;
11 signal mul : std_logic;
12 signal Q : std_logic_vector(1 downto 0);
13 signal div : std_logic;
14 signal rst : std_logic;
15 signal clk : std_logic := '0';
16 signal cword : ctrl_word_type;
17 type state_type is (IDLE, INIT, ADD, SUB, SHIFT_Q,
18 ↵ SHIFT_A);
19 type oracle_type is array(state_type) of ctrl_word_type;
20 constant oracle : oracle_type := (
21     IDLE => "00000000",
22     INIT => "001110011",
23     ADD => "110000001",
24     SUB => "110001001",
25     SHIFT_Q => "001000001",
26     SHIFT_A => "100000101" );
27 begin
28     dut : entity work.control_unit
29     port map(
30         mul => mul,
31         Q => Q,
32         div => div,
33         rst => rst,
34         clk => clk,
35         ctrl_word => cword );
36
37     clk_gen : process
38     begin
39         wait for CLK_PERIOD/2;
40         clk <= not clk;
41     end process;
```

```

41
42      test : process
43    begin
44      wait until clk'event and clk = '1';
45
46      mul <= '0';
47      Q <= "00";
48      div <= '1';
49      rst <= '1';
50      wait for CLK_PERIOD;
51      assert cword = oracle(IDLE) report "Error IDLE"
52        ↳ severity failure;
53
54      rst <= '0';
55      wait for CLK_PERIOD;
56      assert cword = oracle(IDLE) report "Error IDLE"
57        ↳ severity failure;
58
59      mul <= '1';
60      wait for CLK_PERIOD;
61      assert cword = oracle(INIT) report "Error INIT"
62        ↳ severity failure;
63
64      mul <= '0';
65      wait for CLK_PERIOD;
66      assert cword = oracle(SHIFT_Q) report "Error SHIFT_Q"
67        ↳ severity failure;
68
69      wait for CLK_PERIOD;
70      assert cword = oracle(SHIFT_A) report "Error SHIFT_A"
71        ↳ severity failure;
72
73      div <= '0';
74      wait for CLK_PERIOD;
75      assert cword = oracle(SHIFT_Q) report "Error SHIFT_Q"
76        ↳ severity failure;
77
78      Q <= "01";
79      wait for CLK_PERIOD;
80      assert cword = oracle(SHIFT_A) report "Error SHIFT_A"
81        ↳ severity failure;
82
83      wait for CLK_PERIOD;

```

```

77      assert cword = oracle(ADD) report "Error ADD" severity
    →  failure;
78
79      wait for CLK_PERIOD;
80      assert cword = oracle SHIFT_Q report "Error SHIFT_Q"
    →  severity failure;
81
82      Q <= "10";
83      wait for CLK_PERIOD;
84      assert cword = oracle SHIFT_A report "Error SHIFT_A"
    →  severity failure;
85
86      wait for CLK_PERIOD;
87      assert cword = oracle(SUB) report "Error SUB" severity
    →  failure;
88
89      wait for CLK_PERIOD;
90      assert cword = oracle SHIFT_Q report "Error SHIFT_Q"
    →  severity failure;
91
92      Q <= "11";
93      wait for CLK_PERIOD;
94      assert cword = oracle SHIFT_A report "Error SHIFT_A"
    →  severity failure;
95
96      wait for CLK_PERIOD;
97      assert cword = oracle SHIFT_Q report "Error SHIFT_Q"
    →  severity failure;
98
99      wait for CLK_PERIOD;
100     assert cword = oracle SHIFT_A report "Error SHIFT_A"
    →  severity failure;
101
102     div <= '1';
103     wait for CLK_PERIOD;
104     assert cword = oracle(IDLE) report "Error IDLE"
    →  severity failure;
105
106     wait for 3*CLK_PERIOD;
107   end process;
108 end Behavioral;

```

Listing 11.7: Unità di controllo - Testbench

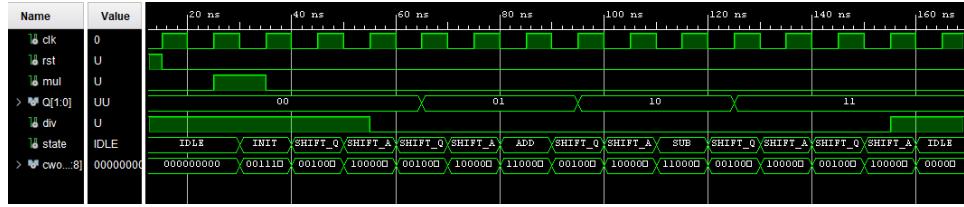


Figura 11.6: Unità di controllo - Waveform

In figura 11.6, si può notare sulla linea `state` l'avanzamento corretto tra gli stati dell'unità di controllo, in perfetto accordo con quanto rappresentato in figura 11.3.

11.4.3 Moltiplicatore di Booth

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.all;
4
5 entity booth_mult_tb is
6 -- Port ( );
7 end booth_mult_tb;
8
9 architecture behavioral of booth_mult_tb is
10 constant CLK_PERIOD : time := 10ns;
11 constant WIDTH : positive := 8;
12 constant OPERAND_1 : integer := -5;
13 constant OPERAND_2 : integer := 47;
14 constant RESULT : integer := -235;
15
16 signal X : std_logic_vector(wIDTH-1 downto 0);
17 signal Y : std_logic_vector(wIDTH-1 downto 0);
18 signal multiply : std_logic;
19 signal rst : std_logic;
20 signal clk : std_logic := '0';
21 signal P : std_logic_vector(2*wIDTH-1 downto 0);
22 signal busy : std_logic;
23 begin
24 dut : entity work.Booth_multiplier
25 generic map(
26     WIDTH => WIDTH)
27 port map(
28     X => X,
29     Y => Y,
```

```

30         multiply => multiply,
31         rst => rst,
32         clk => clk,
33         product => P,
34         busy => busy);
35
36     clk_gen : process
37 begin
38     wait for CLK_PERIOD/2;
39     clk <= not clk;
40 end process;
41
42     test : process
43 begin
44     wait for CLK_PERIOD/4;
45     rst <= '1';
46     wait for CLK_PERIOD;
47
48     rst <= '0';
49     wait for CLK_PERIOD;
50
51     assert busy = '0' report "Error: busy" severity
52         => failure;
53     -- X e Y devono restare stabili per almeno 2 colpi di
54     -- clock;
55     X <= std_logic_vector(to_signed(OPERAND_1,X'length));
56     Y <= std_logic_vector(to_signed(OPERAND_2,Y'length));
57     multiply <= '1';
58     wait for CLK_PERIOD;
59
60     assert busy = '1' report "Error: busy" severity
61         => failure;
62     multiply <= '0';
63
64     wait until falling_edge(busy);
65     assert P = std_logic_vector(to_signed(RESULT,P'length))
66         report "Error result" severity failure;
67
68     wait for CLK_PERIOD/4;
69     -- X e Y devono restare stabili per almeno 2 colpi di

```

```

70      wait for CLK_PERIOD;
71
72      assert busy = '1' report "Error: busy" severity
73          => failure;
74      multiply <= '0';
75
76      wait until falling_edge(busy);
77      assert P = std_logic_vector(to_signed(RESULT,P'length))
78          report "Error result" severity failure;
79
80      wait;
81
82  end process;
83
84 end behavioral;

```

Listing 11.8: Moltiplicatore di Booth - Testbench

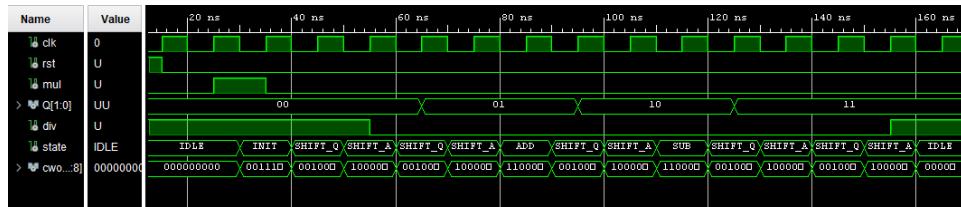


Figura 11.7: Moltiplicatore di Booth - Waveform

In figura 11.7, sono riportate le waveform dei segnali generati dal moltiplicatore a seguito dell'avvio di un'operazione di calcolo. Alla fine dell'esecuzione si può leggere sulla linea P[15:0] il risultato del calcolo, ovvero $-5 \cdot 47 = -235$.

11.5 Timing analysis

Come riportato alla fine del paragrafo 11.2, affinché il moltiplicatore di Booth funzioni correttamente è necessario che il circuito combinatorio di Adder/Subtractor evolva completamente in al più un periodo di clock. O meglio, per sintetizzare la macchina, è opportuno scegliere un segnale di clock con un periodo pari almeno all'intervallo di tempo necessario al componente più lento del circuito per presentare dei segnali d'ingresso stabili al componente sequenziale successivo. Tuttavia, il ritardo effettivo di una macchina, non dipende unicamente dal suo progetto in VHDL, ma dipende fortemente da come il sintetizzatore effettua le fasi di Place&Route dei circuiti e dalle caratteristiche fisiche dell'FPGA su cui la macchina verrà implementata.

Il tool di sviluppo della Xilinx, mette a disposizione dei progettisti un sistema di *timing analysis*, che permette di analizzare le caratteristiche temporali dei componenti prima ancora che questi vengano implementati sulla FPGA. È possibile generare un report di analisi temporale sia dopo la fase di sintesi, che dopo la fase di implementazione. Ovviamente, il secondo report sarà più attendibile, in quanto generato a valle della fase di Place&Route. Inoltre, è possibile generare dei report temporali sia per macchine puramente combinatorie, sia per macchine sequenziali. Le prime, come sappiamo, non vedono e non sono influenzate nella loro esecuzione dal segnale di clock, dunque non sono vincolate al suo periodo. Per poter generare un report temporale per le macchine combinatorie pure è necessario abilitare l'opzione di "report unconstrained_path". Tuttavia, un report così generato non è molto significativo, in quanto il tool di sintesi, dovendo generare una macchina puramente combinatoria, che dunque non è vincolata nei tempi al segnale di clock, può prendere delle scelte di sintesi con dei vincoli temporali molto laschi, senza intaccare la funzionalità della macchina. Si prenda ad esempio il componente Adder/Subtractor in isolamento. In

```
Max Delay Paths
-----
Slack:           inf
Source:          a[0]
                  (input port)
Destination:    c_out
                  (output port)
Path Group:     (none)
Path Type:      Max at Slow Process Corner
Data Path Delay: 18.903ns (logic 4.632ns (24.501%) route 14.272ns (75.499%))
Logic Levels:   6 (IBUF=1 LUT5=1 LUT6=3 OBUF=1)
```

Figura 11.8: Adder/Subtractor - Unconstrained timing report

figura 11.8 è riportato il timing report della macchina combinatoria, e si evince come il ritardo massimo sia quello del percorso che va dall'ingresso `a[0]` all'uscita `c_out`, con un tempo di ben 18,903 ns. Tuttavia, leggendo attentamente il report, si può notare che solo il 24,5% del ritardo totale

è dovuto alla logica combinatoria del circuito, mentre il restante 75,5% è dovuto alle scelte di routing del sintetizzatore. Ciò è avvenuto perché il tool di sintesi, non avendo vincoli temporali da rispettare, ha effettuato delle scelte di Place&Route temporalmente poco efficienti.

Per evitare situazioni del genere, è opportuno testare una macchina combinatoria, non in isolamento, ma all'interno di un circuito sequenziale più esteso, in cui l'esecuzione della macchina complessiva è vincolata al periodo del segnale di temporificazione in ingresso. In questo modo, il tool di sintesi sarà in grado di riconoscere i vincoli temporali del progetto, ed effettuerà delle scelte implementative tali da poterli rispettare. Non è sempre detto che il sintetizzatore riesca a rispettare i vincoli temporali, soprattutto se un sistema è particolarmente complesso e i vincoli eccessivamente stringenti, tuttavia tenterà il più possibile di limitare i ritardi della logica e dei percorsi. A valle di ciò, rieseguiamo l'analisi temporale del circuito Adder/Subtractor, questa volta non più in isolamento, ma con il componente inserito tra un banco di registri governati dallo stesso segnale di clock, come in figura 11.9. Andando dunque a generare un nuovo report temporale per l'architettura in

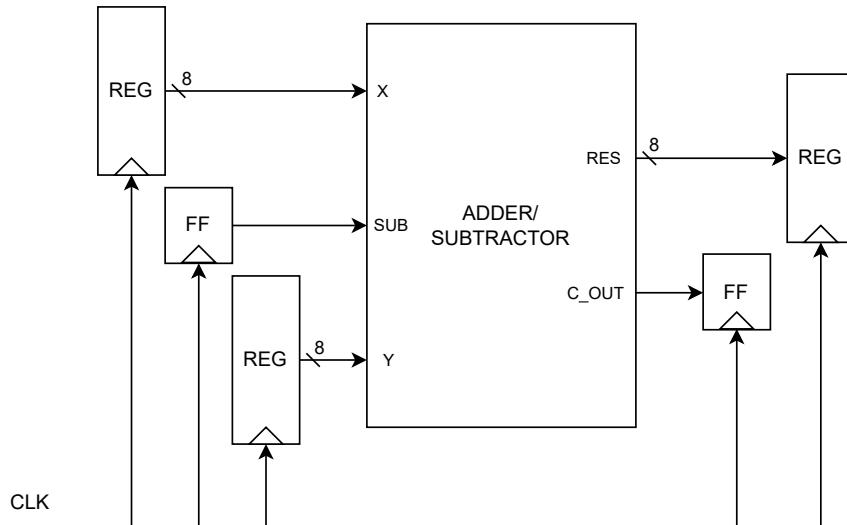


Figura 11.9: Adder/Subtractor - Constrained timing analysis

figura 11.9 a seguito della fase di implementazione, con un periodo di clock di 10 ns, si ottengono i risultati in figura 11.10. Come si può osservare, i vincoli temporali sono pienamente rispettati ($slack > 0$), ed il ritardo massimo del circuito è passato dai 18,903 ns precedenti ad appena 3,412 ns. È importante notare che, nei due casi analizzati, il progetto dell'Adder/Subtractor è il medesimo; ciò che cambia è la sintesi effettuata dal tool di sviluppo.

Non ci resta ora che analizzare se il tool di sintesi sia in grado di implementare l'intero progetto del moltiplicatore di Booth sulla FPGA a disposizione,

```

Max Delay Paths
-----
Slack (MET) : 6.454ns (required time - arrival time)
Source: b_reg_reg[1]/C
          (rising edge-triggered cell FDRE clocked by sys_clk_pin (rise@0.000ns fall@1.000ns period=10.000ns))
Destination: y_reg[5]/D
          (rising edge-triggered cell FDRE clocked by sys_clk_pin (rise@0.000ns fall@1.000ns period=10.000ns))
Path Group: sys_clk_pin
Path Type: Setup (Max at Slow Process Corner)
Requirement: 10.000ns (sys_clk_pin rise@10.000ns - sys_clk_pin rise@0.000ns)
Data Path Delay: 3.412ns (logic 0.966ns (28.312%) route 2.446ns (71.688%)
Logic Levels: 3 (LUT4=1 LUT5=1 LUT6=1)
Clock Path Skew: -0.041ns (DCD - SCD + CPR)
    Destination Clock Delay (DCD): 4.937ns = ( 14.937 - 10.000 )
    Source Clock Delay (SCD): 5.237ns
    Clock Pessimism Removal (CPR): 0.259ns
Clock Uncertainty: 0.035ns ((TSJ^2 + TIJ^2)^1/2 + DJ) / 2 + PE
    Total System Jitter (TSJ): 0.071ns
    Total Input Jitter (TIJ): 0.000ns
    Discrete Jitter (DJ): 0.000ns
    Phase Error (PE): 0.000ns

```

Figura 11.10: Adder/Subtractor - Constrained timing report

rispettando il vincolo temporale dato dal segnale di clock con frequenza fissata a 100 MHz. Come si vede in figura 11.11, i vincoli temporali del progetto

```

-----| Design Timing Summary | -----
-----| WNS(ns) TNS(ns) TNS Failing Endpoints TNS Total Endpoints WHS(ns) THS(ns) THS Failing Endpoints THS Total Endpoints |
-----| 0.890 0.000 0 161 0.138 0.000 0 161 |
-----| All user specified timing constraints are met. |
-----| Clock Summary | -----
-----| Clock Waveform(ns) Period(ns) Frequency(MHz) |
-----| sys_clk_pin {0.000 5.000} 10.000 100.000 |

```

Figura 11.11: Moltiplicatore di Booth - Constrained timing report

sono rispettati, in quanto il *Worst Negative Slack (WNS)* è un valore positivo. Ricordiamo che lo *slack* di un determinato path, è la differenza tra il ritardo massimo del percorso ed il suo ritardo effettivo.

11.6 Sintesi

Per sintetizzare il progetto del moltiplicatore di Booth sulla FPGA Artix-A7-100T in dotazione, si sono effettuate le seguenti scelte per il mapping dell'I/O:

- i 16 segnali dei due operandi X ed Y sono prelevati dai 16 switch della scheda;
- il segnale di MUL per dare il via al calcolo del prodotto è collegato ad un bottone, così come il segnale di reset RST;
- il segnale di clock CLK è prelevato dalla board, ed è generato con una frequenza di 100 MHz;
- il risultato dell'operazione è visualizzato sul display a 7 segmenti della scheda.

Per gestire l'I/O si è reso necessario inserire degli ulteriori componenti intorno al progetto, come si vede in figura 11.12. Innanzitutto è stato inserito un

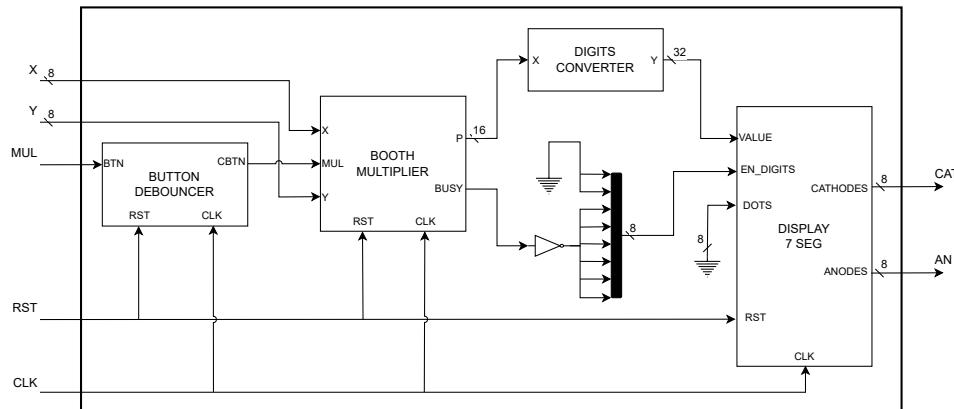


Figura 11.12: Moltiplicatore di Booth on board

button debouncer per stabilizzare il segnale di ingresso MUL, che proviene da un bottone della scheda. Dopodiché, è stato inserito un display manager per governare opportunamente gli anodi e i catodi del display a 7 segmenti, più un circuito detto "digits converter" per alimentare opportunamente il gestore del display.

Il display manager è stato leggermente modificato, in modo tale da non riprodurre tutte le cifre esadecimale (0123...DEF) sullo schermo, ma solo le cifre decimali, più il segno "-" (meno) da anteporre ai risultati negativi. Per tale motivo, non è stato utilizzato un classico encoder BCD per alimentare il display manager, ma è stato realizzato un circuito ad-hoc detto "digits

converter", che altro non è che un decoder che mappa i 16 bit del prodotto sui 32 che il display manager si aspetta in ingresso.

Inoltre, dato che il prodotto è espresso su 16 bit, significa che l'intervallo di rappresentazione in complemento a due del risultato è il seguente:

$$[-2^{15}, 2^{15} - 1] = [-32\,768, 32\,767]$$

Dunque, in decimale, il risultato sarà espresso su al più 5 cifre, più un carattere per anteporre l'eventuale segno negativo. Quindi, sul display dovranno essere abilitate solo le 6 cifre meno significative, e inoltre esse dovranno essere abilitate solo al termine dell'operazione di calcolo, quando sull'uscita P del moltiplicatore si leggerà il risultato finale dell'operazione, ovvero quando il segnale di BUSY è basso.

Infine, si riporta in figura 11.13 lo schema di mapping degli switch e dei bottoni della board di sviluppo.

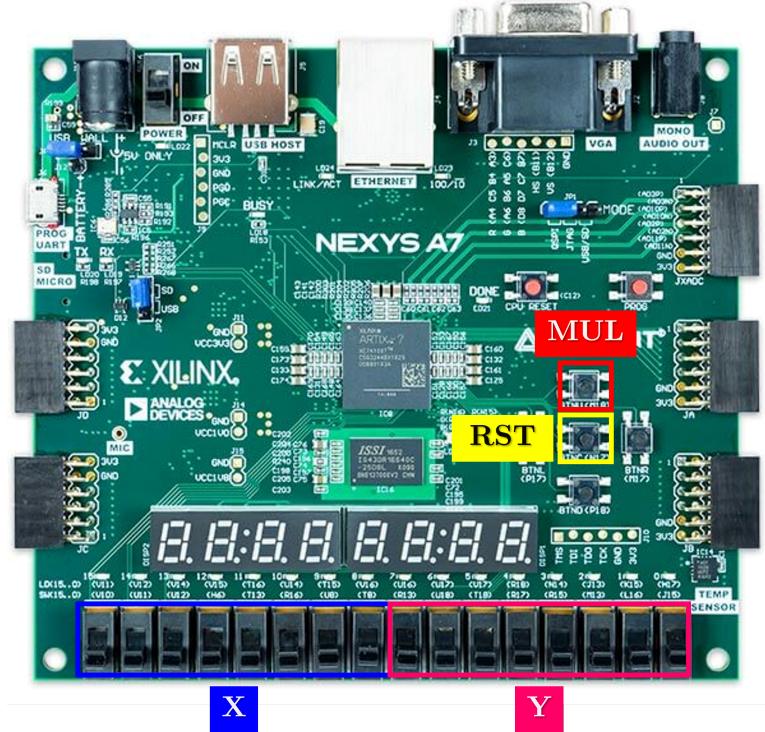


Figura 11.13: Mapping dell'I/O

Capitolo 12

Esercizio libero

12.1 Traccia

Progettare, implementare in VHDL e testare mediante simulazione (e, optionalmente, mediante sintesi su board), un sistema le cui specifiche siano definite dallo studente e rientrino in una delle seguenti tipologie:

- a) Modifica di esercizi già proposti (processore, rete di interconnessione o interfaccia seriale) mediante aggiunta/aggiornamento di funzionalità.
 - Esempio: si potrebbe pensare di modificare l'interfaccia seriale aggiungendo segnali specifici per l'handshaking fra due entità.
- b) Progetto di sistemi che assolvono a specifici compiti noti
 - Esempio: si potrebbe pensare di implementare una specifica macchina aritmetica non trattata a lezione, una funzione crittografica, una rete neurale, ecc.
- c) Progetto di sistemi che integrano opportunamente componenti visti a lezione (contatori, registri, macchine aritmetiche, ecc.). A titolo di esempio, è possibile fare riferimento ai seguenti due esercizi:
 - Progettare un sommatore di byte seriale (le cifre degli addendi devono essere fornite serialmente a coppie alla macchina) a partire da un sommatore di bit. Il sommatore deve terminare le sue operazioni appena il valore temporaneo della somma diventa maggiore di un valore M fornito in input.
 - Si consideri un nodo A che contiene una memoria ROM di N ($N \geq 4$) locazioni da 8 bit ciascuna. Progettare un sistema in grado di trasmettere mediante handshaking completo tutti i valori strettamente positivi contenuti nella memoria di A ad un nodo B. Il nodo B, ricevuti i valori da A, li trasmetterà ad un nodo C mediante una comunicazione parallela con handshaking.

Per tale esercizio, abbiamo deciso di progettare un sistema che rientri nella tipologia *c*, ovvero il progetto di un sistema che integri opportunamente i componenti visti a lezione. La traccia di riferimento è la seguente:

- Progettare, implementare, simulare e sintetizzare il sistema successivamente descritto. Un'unità A e un'unità B trasmettono N byte ciascuna ad un'unità C. La prima invia i dati in parallelo tramite handshaking, mentre la seconda invia i dati attraverso un'interfaccia seriale. Sia l'unità A che la B leggono i byte da una memoria ROM di dimensioni $N \times 8$. L'unità C confronta ogni coppia di byte omologhi ricevuta, e in base al risultato del confronto esegue diverse operazioni. Il risultato delle operazioni eseguite deve essere opportunamente salvato in una memoria.

Sia X il byte inviato da A, e sia Y il byte inviato da B, allora:

- se $X > Y$, in memoria sarà salvata la somma di X e Y;
- se $X = Y$, in memoria sarà salvato indifferentemente il valore di X o Y;
- se $X < Y$, in memoria sarà salvato il risultato della AND bit a bit di X e Y.

Infine bisogna permettere la lettura in memoria dei risultati salvati.

12.2 Soluzione

Per la risoluzione di tale esercizio abbiamo dapprima risolto il problema della comparazione presente al nodo C, progettando opportunamente un comparatore di byte (*byte comparator*). Un comparatore di stringhe da N bit può essere ottenuto in maniera strutturale disponendo in cascata N comparatori di un solo bit. Quindi abbiamo prima progettato il comparatore ad un solo bit (figura 12.1), dopodichè siamo passati alla progettazione del byte comparator. In particolare il comparatore ad un solo bit presenta in ingresso i segnali a e b che rappresentano esattamente i bit da confrontare e i segnali di uscita $a_{gt}b$, $a_{eq}b$ e $a_{lt}b$ che permettono di individuare quali dei due bit è maggiore, uguale o minore. I restanti segnali d'ingresso gt , eq e lt permettono di costruire un comparatore di stringhe su un numero qualsiasi di bit. In particolare il suo comportamento è riportato come di seguito:

- $a_{gt}b = 1$ se $a > b$ oppure $a = b$ e $gt = 1$;
- $a_{eq}b = 1$ se $a = b$ e $eq = 1$;
- $a_{lt}b = 1$ se $a < b$ oppure $a = b$ e $lt = 1$;

Sulla base del comportamento richiesto dalle specifiche del comparatore si possono derivare facilmente le equazioni booleane e progettare correttamente il componente.

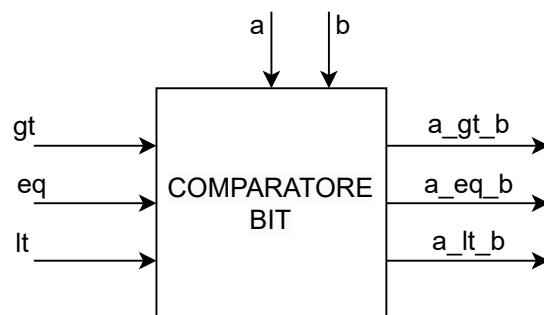


Figura 12.1: Comparatore ad un solo bit

Il comparatore di stringhe è ottenuto mettendo opportunamente in cascata i comparatori ad un bit. In figura 12.2 è presentato il suo schematico e per semplicità è stato realizzato per stringhe da 4 bit. In particolare vengono confrontati i bit di stessa posizione di a e di b attraverso N comparatori di bit e ognuno comunica al comparatore successivo il suo risultato. In questo modo, per come è stato progettato il comparatore ad un solo bit, si riesce a determinare chi tra a e b è maggiore o uguale.

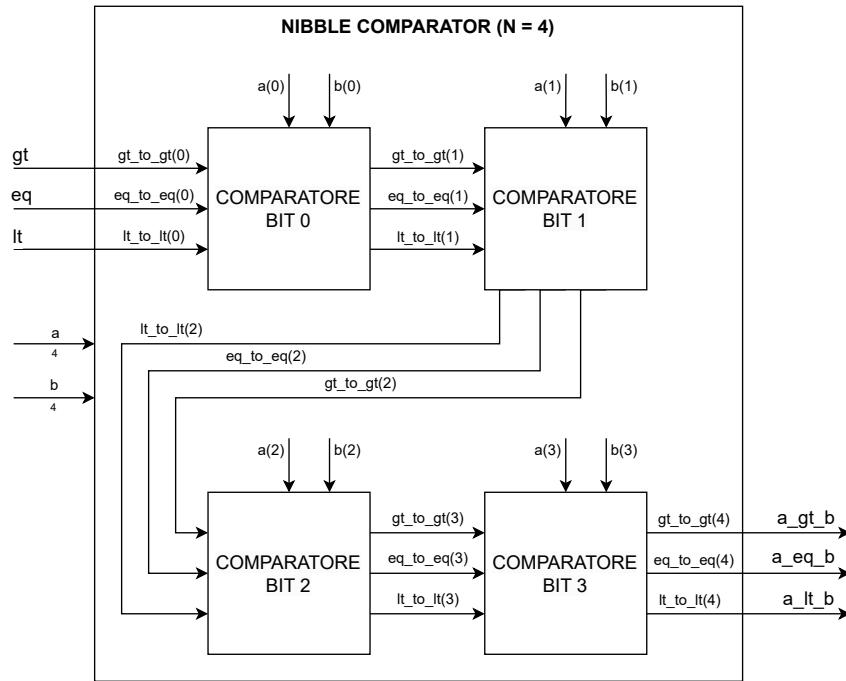


Figura 12.2: Comparatore di stringhe da N bit

A questo punto, facendo riferimento ai capitoli precedenti, abbiamo tutti i componenti per poter realizzare in maniera strutturale i tre nodi dell'esercizio e collegarli tra di loro. Ogni nodo è stato realizzato suddividendolo in due parti: unità operativa e unità di controllo. Lo schema complessivo del nostro esercizio è presentato in figura 12.3.

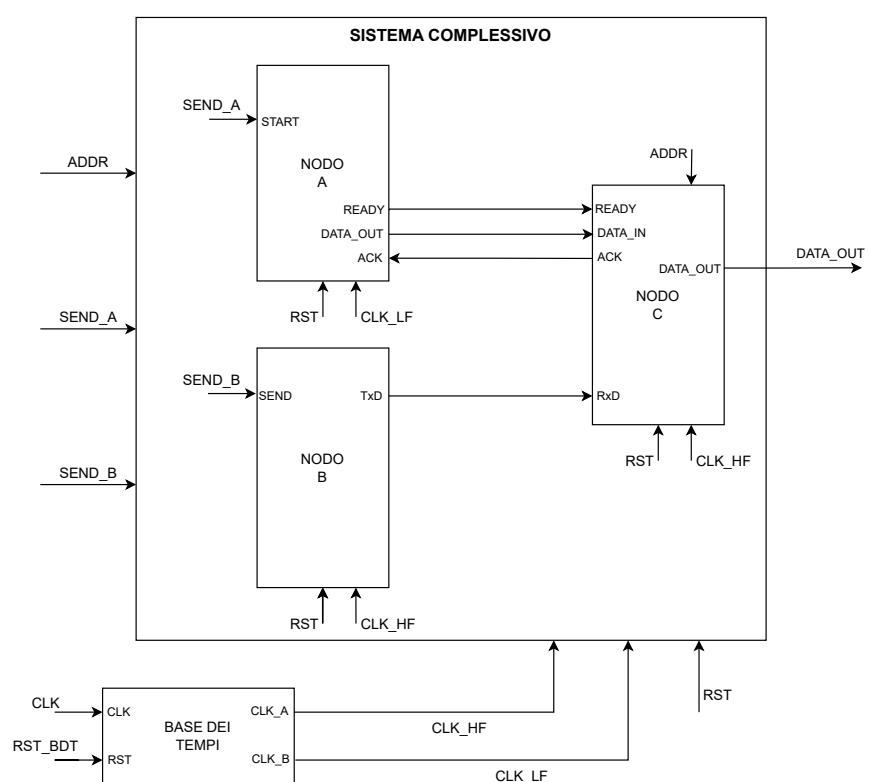


Figura 12.3: Sistema complessivo

12.2.1 Nodo A

Il nodo A ha il compito di inviare al nodo C una sequenza di N stringhe, ognuna lunga 8 bit. Quindi, si è dotata l'unità operativa di una memoria ROM di dimensioni $N \times 8$, per memorizzare i dati da inviare, e di un contatore modulo N per indirizzare la memoria. Vedesi l'unità operativa in figura 12.4. In particolare tale nodo comunica con il nodo C in parallelo mediante un protocollo di handshaking. Per questione di praticità, si rimandi la lettura dell'handshaking al capitolo 7.

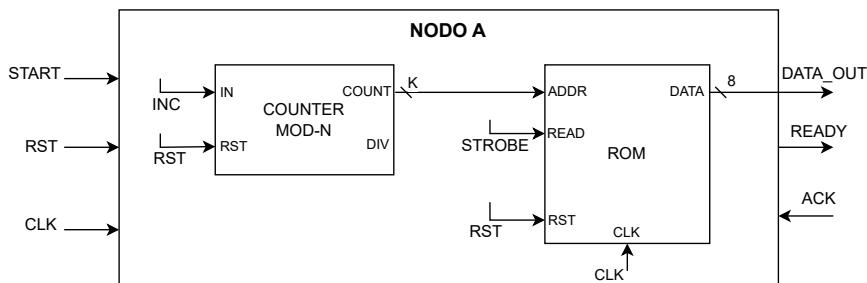


Figura 12.4: Nodo A: unità operativa

Invece per progettare l'unità di controllo del nodo A modelliamo il comportamento della macchina con un automa a stati finiti. Si mostra come in figura 12.5 sia stato realizzato tale automa.

INPUT: <START, ACK, RST>
OUTPUT: <INC, STROBE, READY>

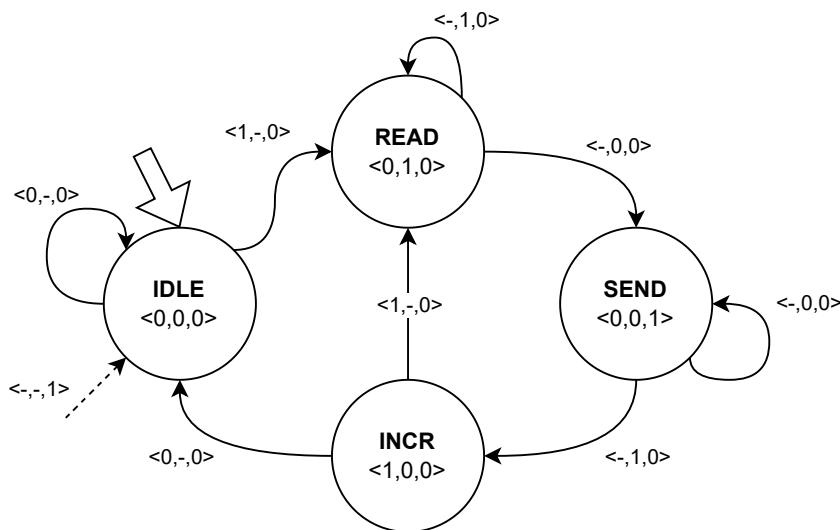


Figura 12.5: Nodo A: unità di controllo

12.2.2 Nodo B

Anche il nodo B ha il compito di inviare al nodo C una sequenza di N stringhe, ognuna lunga 8 bit, con l'unica differenza che qui il messaggio viene inviato in maniera seriale e non parallela. L'unità operativa è dotata anch'essa di una memoria ROM di dimensioni $N \times 8$, per memorizzare i dati da inviare, e di un contatore modulo N per indirizzare la memoria. Vedesi l'unità di controllo in figura 12.6. In particolare tale nodo invia messaggi al nodo C attraverso un dispositivo UART. Per questione di praticità, si rimandi la lettura del funzionamento di tale dispositivo al capitolo 9.

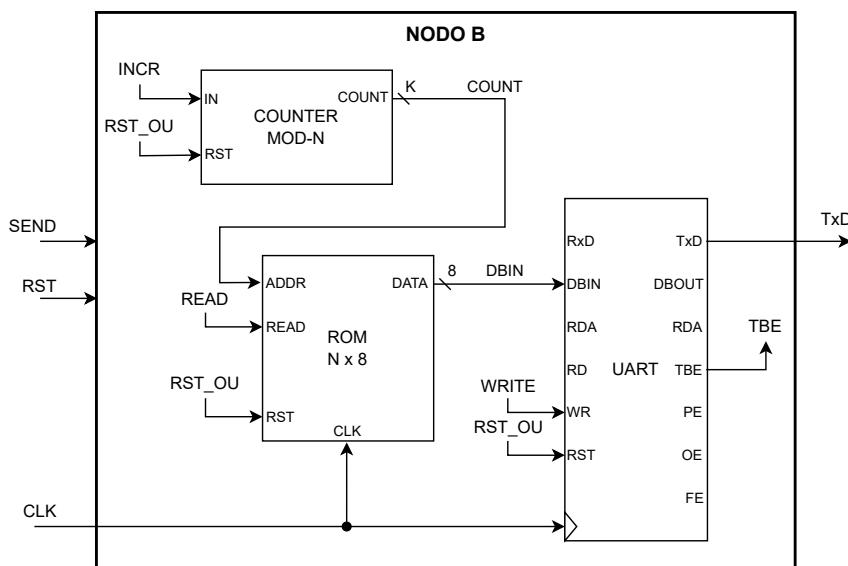


Figura 12.6: Nodo B: unità operativa

Invece per progettare l'unità di controllo del nodo B modelliamo il comportamento della macchina con un automa a stati finiti. Si mostra come in figura 12.7 sia stato realizzato tale automa.

12.2.3 Nodo C

Il nodo C ha il compito di ricevere i messaggi dal nodo A tramite comunicazione parallela e dal nodo B tramite comunicazione seriale. Le stringhe inviategli da entrambi i nodi vengono dapprima salvati in degli opportuni registri, dopodiché i messaggi contenuti nei registri vengono confrontati da un comparatore di byte, il cui risultato di confronto seleziona l'operazione da eseguire sui due messaggi. I risultati di tale operazioni verranno poi salvati opportunamente in memoria. In particolare:

INPUT: < SEND, TBE, RST >
OUTPUT: < INCR, READ, WRITE, RST_OU >

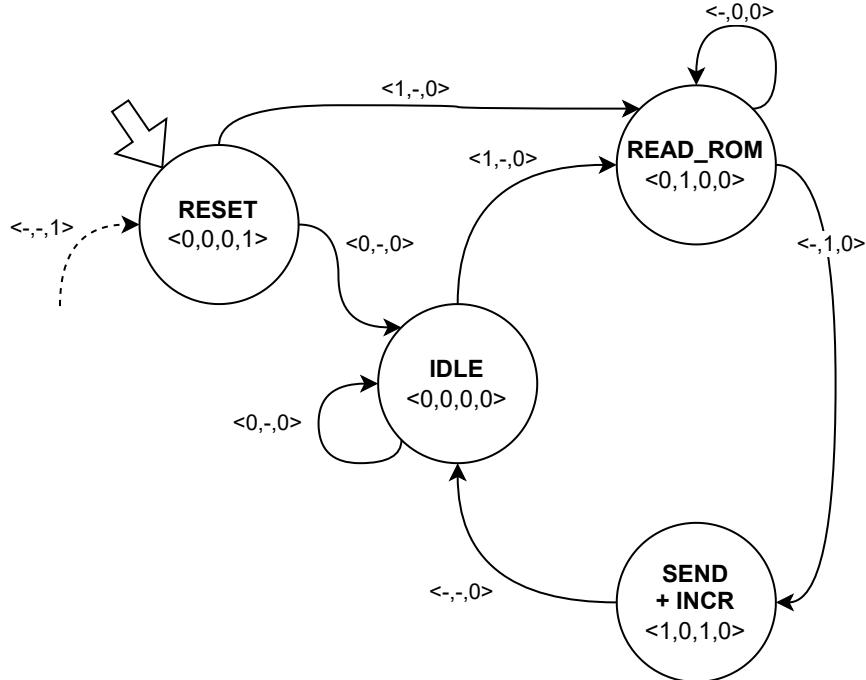


Figura 12.7: Nodo B: unità di controllo

- se il valore del messaggio inviato da A è maggiore del valore del messaggio inviato da B allora viene eseguita un'addizione dei due messaggi tramite un *ripple carry adder*;
- se il valore del messaggio inviato da A è uguale al valore del messaggio inviato da B allora non viene eseguita nessuna operazione e il risultato da salvare in memoria è indifferentemente il messaggio di A o di B;
- se il valore del messaggio inviato da A è minore del valore del messaggio inviato da B allora viene eseguita un'operazione di AND bit a bit tra i due messaggi.

Inoltre è possibile leggere il contenuto della memoria, settando l'indirizzo della locazione da leggere tramite dei segnali esterni mappabili sugli switch della scheda di valutazione. L'*unità operativa* appena descritta è visibile in figura 12.8.

L' unità di controllo del nodo C deve opportunamente pilotare i segnali: EN_REG, EN_MEM, READ, ACK, INCR e CTRL_MODE, affinché l'unità operativa evolva correttamente. Per fare ciò, l'unità di controllo opera secondo il modello in figura 12.9, utilizzando in ingresso i segnali di RDA, READY e RST.

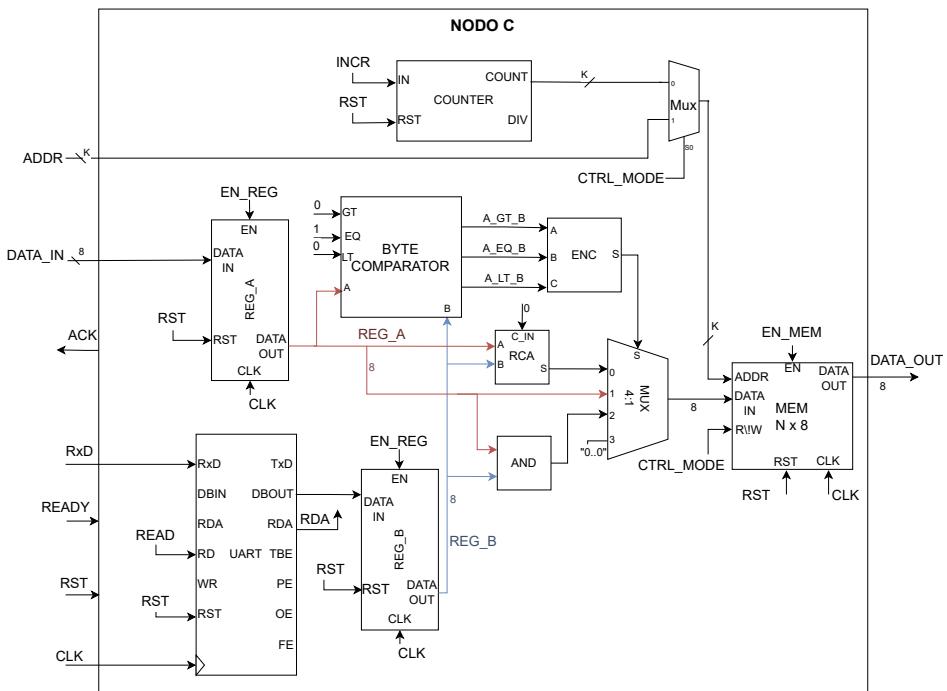


Figura 12.8: Nodo C: unità operativa

INPUT: <RDA, READY, RST>
OUTPUT: <EN_REG, EN_MEM, READ, ACK, INCR, CTRL_MODE>

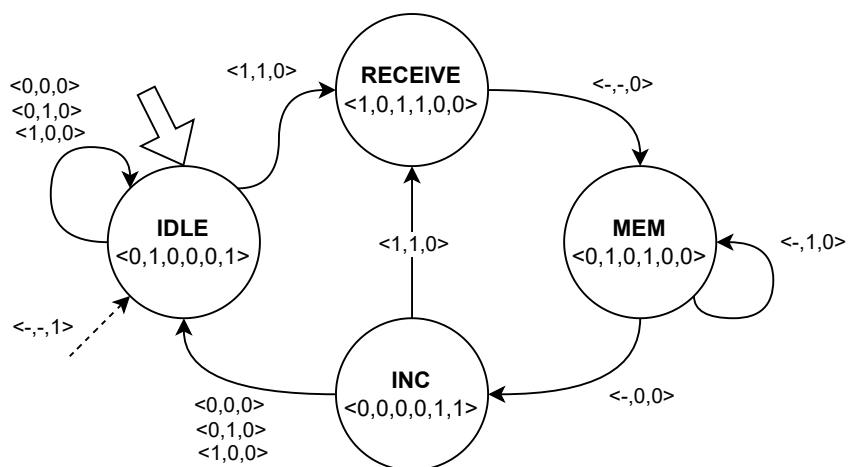


Figura 12.9: Nodo C: unità di controllo

12.3 Codice

12.3.1 Comparatore di un bit

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity bit_comparator is
5     Port ( a : in STD_LOGIC;
6             b : in STD_LOGIC;
7             gt : in STD_LOGIC;
8             eq : in STD_LOGIC;
9             lt : in STD_LOGIC;
10            a_gt_b : out STD_LOGIC;
11            a_eq_b : out STD_LOGIC;
12            a_lt_b : out STD_LOGIC);
13 end bit_comparator;
14
15 architecture dataflow of bit_comparator is
16
17 begin
18
19 a_gt_b <= (a and gt) or (not(b) and gt) or (a and not(b));
20 a_eq_b <= (a and b and eq) or (not(a) and not(b) and eq);
21 a_lt_b <= (not(a) and lt) or (b and lt) or (not(a) and b);
22
23 end dataflow;
```

Listing 12.1: Comparatore di un bit

12.3.2 Comparatore di word

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity byte_comparator is
5     Generic( N : integer := 8);
6     Port ( a : in STD_LOGIC_VECTOR(N-1 downto 0);
7             b : in STD_LOGIC_VECTOR(N-1 downto 0);
8             gt : in STD_LOGIC;
9             eq : in STD_LOGIC;
10            lt : in STD_LOGIC;
11            a_gt_b : out STD_LOGIC;
```

```

12         a_eq_b : out STD_LOGIC;
13         a_lt_b : out STD_LOGIC);
14 end byte_comparator;
15
16 architecture Structural of byte_comparator is
17
18 signal gt_to_gt : std_logic_vector(N downto 0);
19 signal eq_to_eq : std_logic_vector(N downto 0);
20 signal lt_to_lt : std_logic_vector(N downto 0);
21
22 begin
23
24 gt_to_gt(0) <= gt;
25 eq_to_eq(0) <= eq;
26 lt_to_lt(0) <= lt;
27
28 n_c: for i in 0 to N-1 generate
29     comp: entity work.bit_comparator port map (a => a(i), b
30         => b(i), gt => gt_to_gt(i), eq => eq_to_eq(i), lt
31         => lt_to_lt(i), a_gt_b => gt_to_gt(i+1), a_eq_b =>
32         eq_to_eq(i+1), a_lt_b => lt_to_lt(i+1));
33 end generate;
34
35 a_gt_b <= gt_to_gt(N);
36 a_eq_b <= eq_to_eq(N);
37 a_lt_b <= lt_to_lt(N);
38
39 end Structural;

```

Listing 12.2: Comparatore di word

12.3.3 Encoder

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity encoder is
5     Port ( a : in STD_LOGIC;
6             b : in STD_LOGIC;
7             c : in STD_LOGIC;
8             s : out STD_LOGIC_VECTOR(1 downto 0));
9 end encoder;
10

```

```

11  architecture Behavioral of encoder is
12
13  signal x : STD_LOGIC_VECTOR(2 downto 0);
14
15 begin
16  x <= a & b & c;
17
18  with x select
19    s <= "00" when "100",
20            "01" when "010",
21            "10" when "001",
22            "11" when others;
23
24
25 end Behavioral;

```

Listing 12.3: Encoder

12.3.4 Nodo A: unità operativa

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA_datapath is
5   Generic(
6     M : positive := 8;
7     K : positive := 3 );
8   Port ( inc : in STD_LOGIC;
9         strobe : in STD_LOGIC;
10        rst_dp : in STD_LOGIC;
11        clk : in std_logic;
12        data_out : out STD_LOGIC_VECTOR (M-1 downto 0);
13        div : out STD_LOGIC );           -- segnale di stato
14                                     -- del contatore
15
16 architecture structural of nodoA_datapath is
17   component counter is
18     generic (
19       bits : positive := K);
20     Port ( x : in STD_LOGIC;
21             rst : in STD_LOGIC;
22             y : buffer STD_LOGIC_VECTOR (bits-1 downto 0);

```

```

23           div : out STD_LOGIC);
24    end component;
25
26  component rom is
27    generic(
28      nbit_addr : positive := K;          -- numero di bit
29      width : positive := M );          -- dimensione di una
30      Port ( addr : in STD_LOGIC_VECTOR (nbit_addr - 1 downto
31      0);
32          clk : in STD_LOGIC;
33          read : in STD_LOGIC;
34          rst : in STD_LOGIC;
35          data_out : out STD_LOGIC_VECTOR (width - 1
36          downto 0));
37    end component;
38    FOR ALL : rom USE ENTITY work.rom (dataflowA);
39
40 begin
41   counter0 : counter port map(
42     x => inc,
43     rst => rst_dp,
44     y => address,
45     div => div );
46
47   rom0 : rom port map(
48     addr => address,
49     clk => clk,
50     rst => rst_dp,
51     read => strobe,
52     data_out => data_out );
53
54 end structural;

```

Listing 12.4: Nodo A: unità operativa

12.3.5 Nodo A: unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;

```

```

3
4  entity nodoA_CU is
5      Port ( start : in STD_LOGIC;
6              ack : in STD_LOGIC;
7              rst : in STD_LOGIC;
8              clk : in STD_LOGIC;
9              ready : out STD_LOGIC;
10             inc : out STD_LOGIC;
11             strobe : out STD_LOGIC);
12 end nodoA_CU;
13
14 architecture Behavioral of nodoA_CU is
15     type state_type is (
16         IDLE,
17         READ,
18         SEND,
19         INCR );
20     signal state : state_type := IDLE;
21     signal next_state : state_type;
22
23 begin
24
25     process(clk, rst)
26     begin
27         if rst = '1' then
28             state <= IDLE;
29         elsif clk'event and clk = '0' then
30             state <= next_state;
31         end if;
32     end process;
33
34     process(state,start,ack)
35     begin
36         case state is
37             when IDLE =>
38                 inc <= '0';
39                 strobe <= '0';
40                 ready <= '0';
41                 if start = '1' then
42                     next_state <= READ;
43                 else
44                     next_state <= state;
45                 end if;
46             when READ =>

```

```

47           inc <= '0';
48           strobe <= '1';
49           ready <= '0';
50           if ack = '0' then
51               next_state <= SEND;
52           else
53               next_state <= state;
54           end if;
55           when SEND =>
56               inc <= '0';
57               strobe <= '0';
58               ready <= '1';
59               if ack = '1' then
60                   next_state <= INCR;
61               else
62                   next_state <= state;
63               end if;
64           when INCR =>
65               inc <= '1';
66               strobe <= '0';
67               ready <= '0';
68
69               if start = '0' then
70                   next_state <= IDLE;
71               else
72                   next_state <= READ;
73               end if;
74
75           end case;
76       end process;
77
78   end Behavioral;

```

Listing 12.5: Nodo A: unità di controllo

12.3.6 Nodo A

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoA is
5     Generic(
6         M : positive := 8;

```

```

7          K : positive := 3 );
8  Port ( start : in STD_LOGIC;
9          ack : in STD_LOGIC;
10         rst : in STD_LOGIC;
11         clk : in STD_LOGIC;
12         ready : out STD_LOGIC;
13         data_out : out STD_LOGIC_VECTOR (M-1 downto 0));
14 end nodoA;
15
16 architecture structural of nodoA is
17 component nodoA_CU is
18     Port ( start : in STD_LOGIC;
19             ack : in STD_LOGIC;
20             rst : in STD_LOGIC;
21             clk : in STD_LOGIC;
22             ready : out STD_LOGIC;
23             inc : out STD_LOGIC;
24             strobe : out STD_LOGIC );
25 end component;
26
27 component nodoA_datapath is
28     Generic(
29         M : positive;
30         K : positive );
31     Port ( inc : in STD_LOGIC;
32             strobe : in STD_LOGIC;
33             rst_dp : in STD_LOGIC;
34             clk : in STD_LOGIC;
35             data_out : out STD_LOGIC_VECTOR (7 downto 0);
36             div : out STD_LOGIC );
37 end component;
38
39 -- segnali di controllo
40 signal increment : std_logic;
41 signal strobe : std_logic;
42
43 begin
44     dp : nodoA_datapath
45     generic map(
46         M => M,
47         K => K )
48     port map(
49         inc => increment,
50         strobe => strobe,

```

```

51      rst_dp => rst,
52      clk => clk,
53      data_out => data_out);
54
55      cu : nodoA_CU port map(
56          start => start,
57          ack => ack,
58          ready => ready,
59          clk => clk,
60          rst => rst,
61          inc => increment,
62          strobe => strobe );
63
64 end structural;

```

Listing 12.6: Nodo A

12.3.7 Nodo B: unità operativa

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4
5 entity nodoB_ou is
6     generic(
7         BAUD_RATE_G : positive;           -- f_Tx = f_CLK /
8             → BAUD_RATE_G
9         K : positive );
10    Port ( clk : in STD_LOGIC;
11            ctrl_word : in ctrl_word_type;
12            TxD : out STD_LOGIC;
13            div : out STD_LOGIC;
14            TBE : out std_logic);
15
16 architecture structural of nodoB_ou is
17
18     signal count : std_logic_vector(K-1 downto 0);
19     signal dbin : std_logic_vector(7 downto 0);
20
21 begin
22     counter0 : entity work.counter
23         generic map(

```

```

24         bits => K,
25         delay => 0ns )
26     port map(
27         x => ctrl_word(incr),
28         rst => ctrl_word(rst_ou),
29         y => count,
30         div => div );
31
32     rom0 : entity work.rom (dataflowB)
33         generic map(
34             nbit_addr => K,
35             width => 8 )
36         port map(
37             addr => count,
38             clk => clk,
39             read => ctrl_word(read),
40             rst => ctrl_word(rst_ou),
41             data_out => dbin );
42
43     uart0 : entity work.UARTComponent
44         generic map(
45             BAUD_RATE_G => BAUD_RATE_G)           -- f_Tx = f_CLK
46             -- / BAUD_RATE_G
47         port map(
48             DBIN => dbin,
49             WR => ctrl_word(write),
50             RST => ctrl_word(rst_ou),
51             CLK => clk,
52             RXD => '1',
53             RD => '0',
54             TBE => TBE,
55             TXD => TxD );
56 end structural;

```

Listing 12.7: Nodo B: unità operativa

12.3.8 Nodo B: unità di controllo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4

```

```

5  entity nodoB_cu is
6      Port ( send : in STD_LOGIC;
7              TBE : in STD_LOGIC;
8              rst : in STD_LOGIC;
9              clk : in std_logic;
10             ctrl_word : out ctrl_word_type );
11 end nodoB_cu;
12
13 architecture Behavioral of nodoB_cu is
14
15     type state_type is (RESET, IDLE, READ_ROM, SEND_INCR);
16     signal state : state_type := RESET;
17     signal next_state : state_type;
18
19 begin
20
21     process(clk)
22     begin
23         if clk'event and clk = '0' then
24             if rst = '1' then
25                 state <= RESET;
26             else
27                 state <= next_state;
28             end if;
29         end if;
30     end process;
31
32     process(state,send,TBE)
33     begin
34         case state is
35             when RESET =>
36                 ctrl_word <= "0001";
37                 if send = '1' then
38                     next_state <= READ_ROM;
39                 else
40                     next_state <= IDLE;
41                 end if;
42
43             when IDLE =>
44                 ctrl_word <= "0000";
45                 if send = '1' then
46                     next_state <= READ_ROM;
47                 else
48                     next_state <= IDLE;

```

```

49           end if;
50
51       when READ_ROM =>
52           ctrl_word <= "0100";
53           if TBE = '1' then
54               next_state <= SEND_INCR;
55           else
56               next_state <= READ_ROM;
57           end if;
58
59       when SEND_INCR =>
60           ctrl_word <= "1010";
61
62           next_state <= IDLE;
63       end case;
64   end process;
65
66 end Behavioral;

```

Listing 12.8: Nodo B: unità di controllo

12.3.9 Nodo B

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use work.nodoB_util.all;
4
5 entity nodoB is
6     generic(
7         BAUD_RATE_G : positive;          -- f_Tx = f_CLK /
8             → BAUD_RATE_G
9         K : positive );
10    Port ( send : in STD_LOGIC;
11            rst : in STD_LOGIC;
12            clk : in STD_LOGIC;
13            TxD : out STD_LOGIC);
14
15 architecture structural of nodoB is
16
17     signal cword : ctrl_word_type;
18     signal div : std_logic;
19     signal TBE : std_logic;

```

```

20
21 begin
22     ou : entity work.nodoB_ou
23     generic map(
24         BAUD_RATE_G => BAUD_RATE_G,           -- f_Tx = f_CLK /
25         ↳ BAUD_RATE_G
26         K => K )
27     port map(
28         clk => clk,
29         ctrl_word => cword,
30         TxD => TxD,
31         TBE => TBE );
32
33     cu : entity work.nodoB_cu
34     port map(
35         send => send,
36         TBE => TBE,
37         rst => rst,
38         clk => clk,
39         ctrl_word => cword );
40 end structural;

```

Listing 12.9: Nodo B

12.3.10 Nodo C: unità operativa

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4
5 entity nodoC_ou is
6     generic(BAUD_DIVIDE_G : positive;
7             K : positive);           -- f_Rx = f_CLK /
8             ↳ BAUD_DIVIDE_G)
9     port(
10         clk : in STD_LOGIC;
11         addr : in STD_LOGIC_VECTOR (K-1 downto 0);
12         --NODO A
13         data_in : in STD_LOGIC_VECTOR(7 downto 0);
14         --NODO B
15         RxD : in STD_LOGIC;

```

```

16      --SEGNALI DI CONTROLLO
17      read : in STD_LOGIC;
18      incr : in STD_LOGIC;
19      en_mem : in STD_LOGIC;
20      en_reg : in STD_LOGIC;
21      ctrl_mode : in STD_LOGIC;
22      rst_ou : in STD_LOGIC;
23      --USCITA
24      data_out : out STD_LOGIC_VECTOR (7 downto 0);
25      RDA : inout STD_LOGIC);
26 end nodoC_ou;
27
28 architecture Structural of nodoC_ou is
29
30     signal dbout : std_logic_vector(7 downto 0);
31     signal count : std_logic_vector(K-1 downto 0);
32     signal regB : std_logic_vector(7 downto 0);
33     signal regA : std_logic_vector(7 downto 0);
34
35     signal mux_to_mem : std_logic_vector(K-1 downto 0);
36     signal cont1_to_mux : std_logic_vector(K-1 downto 0);
37
38     signal gt_to_enc : std_logic;
39     signal eq_to_enc : std_logic;
40     signal lt_to_enc : std_logic;
41     signal sel : std_logic_vector(1 downto 0);
42
43     signal sum : std_logic_vector(7 downto 0);
44     signal and_res : std_logic_vector(7 downto 0);
45     signal data_mem : std_logic_vector(7 downto 0);
46
47
48
49 begin
50
51     and_res <= regA and regB;
52
53
54     registerB : entity work.register_pp generic map ( dim => 8)
55         port map(x => dbout, clk => clk, enable => en_reg,
56                   ↳rst => rst_ou, y => regB);
57
58     registerA : entity work.register_pp generic map ( dim => 8)

```

```

59      port map(x => data_in, clk => clk, enable => en_reg,
60                  &rst => rst_ou, y => regA);
61
62  uart0 : entity work.UARTcomponent generic map( BAUD_DIVIDE_G =>
63          BAUD_DIVIDE_G )           -- f_Rx = f_CLK / BAUD_DIVIDE_G
64          port map( DBIN => (others => '0'), WR => '0',
65          RST => rst_ou, CLK => clk,
66          RXD => RxD, RD => read,
67          DBOUT => dbout,RDA => RDA );
68
69  comp : entity work.byte_comparator generic map( N => 8)
70          port map(a => regA, b => regB, gt =>'0', eq => '1',
71                      &lt => '0',
72          a_gt_b => gt_to_enc, a_eq_b => eq_to_enc, a_lt_b =>
73          lt_to_enc );
74
75  enc : entity work.encoder
76          port map(a => gt_to_enc, b => eq_to_enc, c =>
77                      &lt_to_enc, s => sel);
78
79  mux4_1 : entity work.mux4to1 generic map( N => 8 )
80          port map( x0 => sum, x1 => regA, x2 => and_res, x3
81                      &gt;= "00000000",
82          s => sel, y => data_mem);
83
84  rca : entity work.ripple_carry_adder generic map (nbit => 8)
85          port map(a => regA, b => regB, c_in => '0', s => sum);
86
87  mem : entity work.memory generic map( nbits_addr => K, width =>
88          8 )
89          port map( addr => mux_to_mem,
90          en => en_mem, clk => clk, rst => rst_ou,
91          data_in => data_mem, r_w => ctrl_mode, data_out =>
92          &data_out);
93
94  mux2_1 : entity work.mux2to1 generic map( N => K)
95          port map( x1 => count, x2 => addr, s => ctrl_mode, y =>
96          &mux_to_mem);

```

```

94
95  counter0 : entity work.counter generic map( bits => K, delay =>
96    → 0ns )
97          port map( x => incr, rst => rst_ou, y => count );
98
99 end Structural;

```

Listing 12.10: Nodo C: unità operativa

12.3.11 Nodo C: unità di controllo

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity nodoC_cu is
5      Port ( rda : in STD_LOGIC;
6              ready : in STD_LOGIC;
7              rst : in STD_LOGIC;
8              clk : in STD_LOGIC;
9              en_reg : out STD_LOGIC;
10             en_mem : out STD_LOGIC;
11             read : out STD_LOGIC;
12             incr : out STD_LOGIC;
13             ctrl_mode : out STD_LOGIC;
14             ack : out STD_LOGIC);
15 end nodoC_cu;
16
17 architecture Behavioral of nodoC_cu is
18
19 type stato is (idle, receive, mem, inc);
20 signal sc : stato := idle; --stato corrente
21 signal next_state : stato;
22
23 begin
24     process(clk)
25     begin
26         if(falling_edge(clk)) then
27             if(rst = '1') then
28                 sc <= idle;
29                 else sc <= next_state;
30                 end if;
31             end if;
32     end process;

```

```

33
34     process(sc, rda, ready, rst)
35         begin
36             case sc is
37
38                 when idle =>
39                     en_reg <= '0';
40                     en_mem <= '1';
41                     read <= '0';
42                     ack <= '0';
43                     incr <= '0';
44                     ctrl_mode <= '1';
45
46                     if(rda = '1' and ready = '1') then
47                         next_state <= receive;
48                     else next_state <= sc;
49                     end if;
50
51                 when receive =>
52                     en_reg <= '1';
53                     en_mem <= '0';
54                     read <= '1';
55                     ack <= '1';
56                     incr <= '0';
57                     ctrl_mode <= '0';
58
59                     next_state <= mem;
60
61                 when mem =>
62                     en_reg <= '0';
63                     en_mem <= '1';
64                     read <= '0';
65                     ack <= '1';
66                     incr <= '0';
67                     ctrl_mode <= '0';
68
69                     if(ready = '1') then next_state <=
70                         sc;
71                     else next_state <= inc;
72                     end if;
73
74                 when inc =>
75                     en_reg <= '0';
76                     en_mem <= '0';

```

```

75          read <= '0';
76          ack <= '0';
77          incr <= '1';
78          ctrl_mode <= '1';
79
80          if(rda = '1' and ready = '1') then
81              next_state <= receive;
82          else next_state <= idle;
83          end if;
84      end case;
85  end process;
86 end Behavioral;

```

Listing 12.11: Nodo C: unità di controllo

12.3.12 Nodo C

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity nodoC is generic(
5     BAUD_DIVIDE_G : positive;           -- f_Rx = f_CLK /
6     K : positive );
7     port (clk : in std_logic;
8         rst : in std_logic;
9         RxD : in std_logic;
10        data_in : in std_logic_vector(7 downto 0);
11        ready : in std_logic;
12        addr : in std_logic_vector(K-1 downto 0);
13        data_out : out std_logic_vector(7 downto 0);
14        ack : out std_logic);
15
16 end nodoC;
17
18 architecture Structural of nodoC is
19
20 signal rda : std_logic;
21
22 signal en_reg : STD_LOGIC;
23 signal en_mem : STD_LOGIC;
24 signal read : STD_LOGIC;

```

```

25  signal incr : STD_LOGIC;
26  signal ctrl_mode : STD_LOGIC;
27
28 begin
29
30     cu: entity work.nodoC_cu port map(
31         rda =>rda, ready => ready, rst => rst, clk => clk,
32         en_reg => en_reg, en_mem => en_mem, read => read, incr
33         => incr,
34         ctrl_mode => ctrl_mode, ack => ack);
35
36     ou: entity work.nodoC_ou generic map ( BAUD_DIVIDE_G =>
37         BAUD_DIVIDE_G, K => K)
38         port map(clk => clk, addr => addr, data_in => data_in,
39             => RxD => RxD,
40             read => read, incr => incr, en_mem => en_mem, en_reg =>
41             => en_reg,
42             ctrl_mode => ctrl_mode, rst_ou => rst,
43             data_out => data_out, RDA => rda);
44
45
46 end Structural;

```

Listing 12.12: Nodo C

12.3.13 Sistema complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity system is generic(
5     BAUD_DIVIDE_G : positive := 651;      -- baud_rate =
6     -- 9600 con f_CLK = 100MHz
7     K : positive := 3 );
8
9     Port ( rst : in STD_LOGIC;
10            clk_A : in STD_LOGIC;
11            clk_B : in STD_LOGIC;
12            addr : in STD_LOGIC_VECTOR(2 downto 0);
13            send_A : in STD_LOGIC;
14            send_B : in STD_LOGIC;
15            data_out : out STD_LOGIC_VECTOR(7 downto 0));
16
17 end system;

```

```

16
17  architecture Structural of system is
18      constant BAUD_RATE_G : positive := 16 *
19          BAUD_DIVIDE_G;    --  $f_{Tx} = f_{Rx}/16$ 
20      signal ack : std_logic;
21      signal ready : std_logic;
22      signal data_sendA : std_logic_vector(7 downto 0);
23      signal TxD : std_logic;
24  begin
25
26      nodoA: entity work.nodoA
27          generic map(
28              K => K )
29          port map(
30              start => send_A, ack => ack, rst => rst, clk => clk_A,
31              ready => ready, data_out => data_sendA);
32
33      nodoB: entity work.nodoB
34          generic map(BAUD_RATE_G => BAUD_RATE_G, K => K)
35          port map(send => send_B, rst => rst, clk => clk_B, TxD =>
36              TxD);
37
38      nodoC: entity work.nodoC
39          generic map(BAUD_DIVIDE_G => BAUD_DIVIDE_G, K => K)
40          port map(clk => clk_B, rst => rst, RxD => TxD, data_in =>
41              data_sendA,
42              ready => ready, addr => addr, data_out => data_out, ack
43              => ack);
44
45  end Structural;

```

Listing 12.13: Sistema complessivo

12.3.14 Sistema complessivo on-board

```

1  library IEEE;
2  use IEEE.STD_LOGIC_1164.ALL;
3
4  entity system_on_board is
5      Generic(
6          CLK_PERIOD : positive := 10;
7          BTN_NOISE_TIME : positive := 125000000;
8          CLK_B_FACTOR : positive := 4;

```

```

9      CLK_B_DELAY : time := 2ns );
10     Port(external_rst_btn : in std_logic;
11           internal_rst_btn : in std_logic;
12           clk : in std_logic;
13           addr_sw : in std_logic_vector(2 downto 0);
14           send_A_btn : in std_logic;
15           send_B_btn : in std_logic;
16           led : out std_logic_vector(7 downto 0));
17   end system_on_board;
18
19 architecture structural of system_on_board is
20   signal cleared_send_A : std_logic;
21   signal cleared_send_B : std_logic;
22   signal cleared_RST : std_logic;
23   signal clk_HF : std_logic;
24   signal clk_LF : std_logic;
25
26 begin
27   sys: entity work.system
28   port map(rst => cleared_RST,
29             clk_A => clk_LF,
30             clk_B => clk_HF,
31             addr => addr_sw,
32             send_A => cleared_send_A,
33             send_B => cleared_send_B,
34             data_out => led );
35
36   btn_deb_send_A : entity work.button_debouncer
37   generic map (
38     clk_period => CLK_PERIOD * CLK_B_FACTOR,
39     btn_noise_time => BTN_NOISE_TIME)
40   port map (
41     rst => cleared_RST,
42     clk => clk_LF,
43     btn => send_A_btn,
44     cleared_btn => cleared_send_A);
45
46   btn_deb_send_B : entity work.button_debouncer
47   generic map (
48     clk_period => CLK_PERIOD,
49     btn_noise_time => BTN_NOISE_TIME)
50   port map (
51     rst => cleared_RST,
52     clk => clk_HF,

```

```

53     btn => send_B_btn,
54     cleared_btn => cleared_send_B);
55
56     btn_deb_reset : entity work.button_debouncer
57     generic map (
58         clk_period => CLK_PERIOD,
59         btn_noise_time => BTN_NOISE_TIME)
60     port map (
61         rst => external_rst_btn,
62         clk => clk_LF,
63         btn => internal_rst_btn,
64         cleared_btn => cleared_RST);
65
66     bdt: entity work.base_dei_tempi
67     generic map(
68         clk_B_factor => CLK_B_FACTOR,
69         clk_B_delay => CLK_B_DELAY)
70     port map(
71         clk => clk,
72         rst => external_rst_btn,
73         clk_A => clk_HF,
74         clk_B => clk_LF );
75
76 end structural;

```

Listing 12.14: Sistema complessivo on-board

12.4 Simulazione

Ogni singolo componente del progetto è stato testato attraverso un testbench ad-hoc. Tuttavia, per questioni di brevità espositiva, si riportano unicamente i risultati delle simulazioni del sistema complessivo e del comparatore di word.

12.4.1 Testbench comparatore di word

```
1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3
4 entity byte_comparator_tb is
5 -- Port ( );
6 end byte_comparator_tb;
7
8 architecture Behavioral of byte_comparator_tb is
9
10 signal a : STD_LOGIC_VECTOR(7 downto 0);
11 signal b : STD_LOGIC_VECTOR(7 downto 0);
12 signal gt : STD_LOGIC;
13 signal eq : STD_LOGIC;
14 signal lt : STD_LOGIC;
15 signal a_gt_b : STD_LOGIC;
16 signal a_eq_b : STD_LOGIC;
17 signal a_lt_b : STD_LOGIC;
18
19
20 begin
21
22     uut: entity work.byte_comparator generic map(N => 8)
23         port map (a => a, b => b, gt => gt, eq => eq, lt => lt,
24                    a_gt_b => a_gt_b, a_eq_b => a_eq_b, a_lt_b =>
25                    a_lt_b
26    );
27
28     prc_tb: process begin
29         a <= "11110000";
30         b <= "00011111";
31
32         lt <= '0';
33         eq <= '1';
34         gt <= '0';
35         wait for 50ns;
```

```

34
35      b <= "11110000";
36      a <= "00001111";
37
38      lt <= '0';
39      eq <= '1';
40      gt <= '0';
41      wait for 50ns;
42
43      b <= "10110000";
44      a <= "10110000";
45
46      lt <= '0';
47      eq <= '1';
48      gt <= '0';
49      wait for 50ns;
50
51      a <= "10110000";
52      b <= "01110000";
53
54      lt <= '0';
55      eq <= '1';
56      gt <= '0';
57      wait for 50ns;
58
59      b <= "10110000";
60      a <= "00110011";
61
62      lt <= '0';
63      eq <= '1';
64      gt <= '0';
65      wait for 50ns;
66
67      a <= "10010000";
68      b <= "00011111";
69
70      lt <= '0';
71      eq <= '1';
72      gt <= '0';
73      wait;
74      end process;
75
76 end Behavioral;

```

Listing 12.15: Testbench comparatore di word



Figura 12.10: Comparatore di word - Risultati della simulazione

12.4.2 Testbench sistema complessivo

```

1 library IEEE;
2 use IEEE.STD_LOGIC_1164.ALL;
3 use IEEE.numeric_std.all;
4
5 entity system_tb is
6 end system_tb;
7
8 architecture Behavioral of system_tb is
9
10 constant BAUD_DIVIDE_G : positive := 2;
11 constant CLK_PERIOD_B : time := 5ps;
12 constant CLK_PERIOD_A : time := 4*CLK_PERIOD_B;
13
14 signal rst : STD_LOGIC;
15 signal addr : STD_LOGIC_VECTOR(2 downto 0);
16 signal send_A : STD_LOGIC;
17 signal send_B : STD_LOGIC;
18 signal data_out : STD_LOGIC_VECTOR(7 downto 0);
19 signal clk_A : std_logic := '0';
20 signal clk_B : std_logic := '0';
21
22 type rom_type is array(0 to 7) of std_logic_vector(7 downto 0);
23 constant oracle : rom_type := (
24    X"01",
25    X"41",
26    X"04",
27    X"08",
28    X"11",
29    X"22",
30    X"40",

```

```

31    x"80" );
32
33  begin
34
35      uut: entity work.system generic map(BAUD_DIVIDE_G =>
36          BAUD_DIVIDE_G)      -- baud_rate = 9600 con f_CLK =
37          100MHz
38          port map(rst => rst, clk_A => clk_A, clk_B => clk_B,
39                      -- addr => addr, send_A => send_A, send_B => send_B,
40                      -- data_out=> data_out);
41
42
43      clk_A_gen : process begin
44          wait for CLK_PERIOD_A/2;
45          clk_A <= not clk_A;
46      end process;
47
48      clk_B_gen : process begin
49          wait for CLK_PERIOD_B/2;
50          clk_B <= not clk_B;
51      end process;
52
53      prc_tb: process begin
54          wait for CLK_PERIOD_B/4;
55          rst <= '1';
56
57          send_A <= '0';
58          send_B <= '0';
59          addr <= "000";
60          wait for CLK_PERIOD_A;
61          rst <= '0';
62
63          --         for i in 0 to 7 loop
64          --             wait for CLK_PERIOD_A;
65          --             addr <=
66          --                 std_logic_vector(to_unsigned(i,addr'length));
67          --                 send_B <= '1';
68          --                 wait for CLK_PERIOD_B;
69          --                 send_B <= '0';
70
71          --                 send_A <= '1';
72          --                 wait for CLK_PERIOD_A;
73          --                 send_A <= '0';
74
75          --         wait on data_out;

```

```

70      ----             assert data_out = oracle(i) report "Error"
71      --> severity failure;
72      --          end loop;
73
74
75      --          wait for 2*CLK_PERIOD_A;
76
77      --          for i in 0 to 7 loop
78      --            addr <=
79      --              std_logic_vector(to_unsigned(i,addr'length));
80      --          wait until data_out'active;
81      --          assert data_out = oracle(i) report "Error"
82      --> severity failure;
83      --          end loop;
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110

```

```

111
112
113      end process;
114 end Behavioral;
```

Listing 12.16: Testbench sistema complessivo

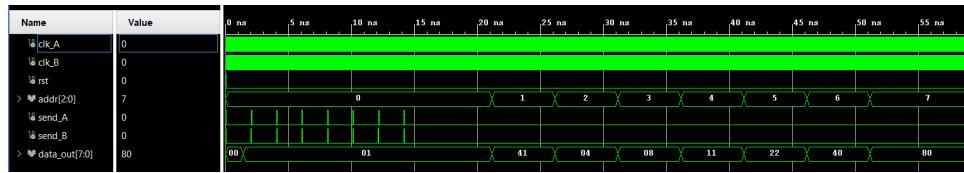


Figura 12.11: Sistema complessivo - Risultati della simulazione

12.5 Sintesi

Il sistema complessivo "on-board" è stato caricato sulla scheda Nexys-A7 della Digilent. Per fare ciò è stato necessario mappare i collegamenti verso l'esterno del componente con gli elementi di I/O presenti sulla scheda. In particolare, i quattro segnali di sendA_btn, sendB_btn, external_rst_btn e internal_rst_btn sono stati mappati con quattro button presenti sulla scheda e il segnale addr_sw è stato mappato con gli switch presenti sulla board, mentre le linee di output sono state mappate sui led della scheda.

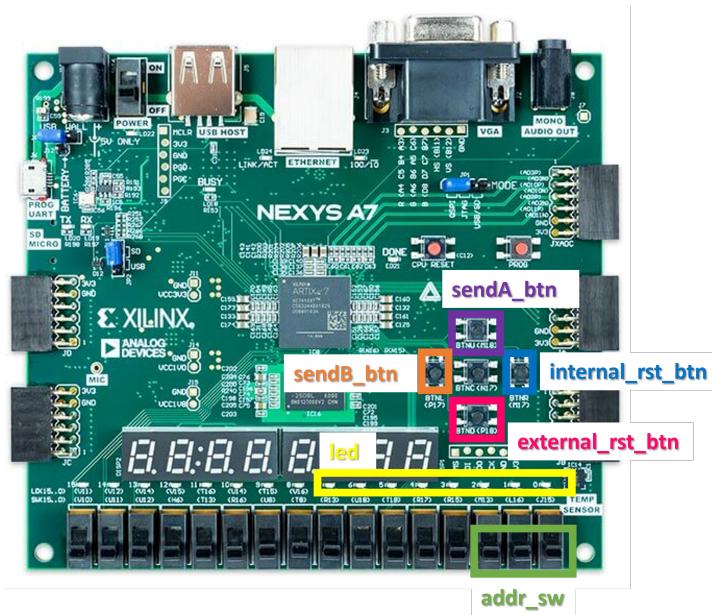


Figura 12.12: Schema di mapping del sistema complessivo su board