



UNIVERSITÀ  
DEGLI STUDI  
DI BRESCIA

## Università degli Studi di Brescia

DIPARTIMENTO DI INGEGNERIA DELL'INFORMAZIONE

Corso di Laurea Magistrale in Ingegneria Elettronica

RELAZIONE

## MIPS-Like Processor Moltiplicatore di numeri interi senza segno a 16 bit

Progetto di Sistemi Elettronici per l'Internet of Things

Autori:  
**Brescia Luca**  
Matricola **706005**

**Loda Michele**  
Matricola **85967**

**Pezzottini Simone**  
Matricola **89521**

# Indice

<b>Specifiche di progetto</b>	<b>2</b>
<b>Architettura del sistema</b>	<b>3</b>
FPGA Cyclone III . . . . .	3
Raspberry Pi Model 2B . . . . .	4
<b>Struttura del moltiplicatore 16x16 bit</b>	<b>5</b>
Come funziona l'algoritmo . . . . .	5
Descrizione dell'algoritmo di moltiplicazione . . . . .	6
Motivazioni della dimensione dei dati . . . . .	8
<b>Progettazione e implementazione in VHDL</b>	<b>9</b>
<b>    Descrizione del linguaggio VHDL</b>	<b>9</b>
Codice implementato . . . . .	9
Approccio di progettazione . . . . .	13
<b>    Simulazione e verifica</b>	<b>14</b>
<b>    Introduzione alla Simulazione con Altera ModelSim per l'Architecture "basic_mult"</b>	<b>14</b>
Dettagli dello script di simulazione . . . . .	16
<b>    Sintesi e implementazione</b>	<b>18</b>
Modulo basic_mult . . . . .	18
Fase di implementazione . . . . .	18
Fase di sintesi . . . . .	21
Modulo SPI . . . . .	23
Fase di implementazione . . . . .	23
Fase di sintesi . . . . .	25
<b>    Programmazione di RaspberryPi utilizzando MATLAB Simulink</b>	<b>26</b>
Ruolo di MATLAB Simulink nel progetto . . . . .	26
Configurazione e comunicazione con RaspberryPi . . . . .	27
Funzionamento della comunicazione SPI . . . . .	27
<b>    Visualizzazione del dato in rete</b>	<b>29</b>
<b>Risultati sperimentali</b>	<b>31</b>
Studio comportamentale . . . . .	31
Analisi temporale . . . . .	32
Analisi del sistema con SignalTap Logic Analyzer . . . . .	33
<b>Conclusioni</b>	<b>35</b>

# Specifiche di progetto

È richiesto di implementare in FPGA un moltiplicatore di numeri interi a 16bit e di implementare un'interconnessione via SPI con Raspberry pi. La comunicazione SPI deve permettere l'invio di moltiplicando e moltiplicatore da Raspberry verso l'FPGA e l'invio del prodotto ottenuto da tali operandi da FPGA verso RaspberryPi. RaspberryPi dovrà successivamente il prodotto su seriale ethernet con protocollo UDP verso dispositivi connessi in rete.

# Architettura del sistema

## FPGA Cyclone III

L'FPGA Cyclone III EP3C16F484C6, mostrata in figura 1, è un dispositivo versatile e potente appartenente alla famiglia di FPGA prodotta da Intel, caratterizzato da specifiche e funzionalità avanzate che lo rendono adatto a diverse applicazioni.

- **Dimensioni:** L'FPGA è disponibile nella confezione 484-pin FineLine BGA, garantendo compattezza e adattabilità a sistemi embedded con limitazioni di spazio.
- **Capacità logica:** L'FPGA offre 16,608 elementi logici (LEs), consentendo la realizzazione di circuiti digitali complessi.
- **Memoria:** Dispone di memoria integrata di tipo M9K, con una capacità di archiviazione di 608 kilobits.
- **Alta velocità di clock:** Supporta velocità di clock elevate, permettendo un'elaborazione rapida dei dati e delle operazioni.
- **I/O flessibili:** Dispone di una vasta varietà di pin I/O configurabili, adattabili alle esigenze specifiche del sistema.
- **Consumo energetico ridotto:** Grazie alla tecnologia di fabbricazione a bassa potenza, l'FPGA offre un consumo energetico ottimizzato, rendendolo adatto a dispositivi a batteria e a sistemi a basso consumo.

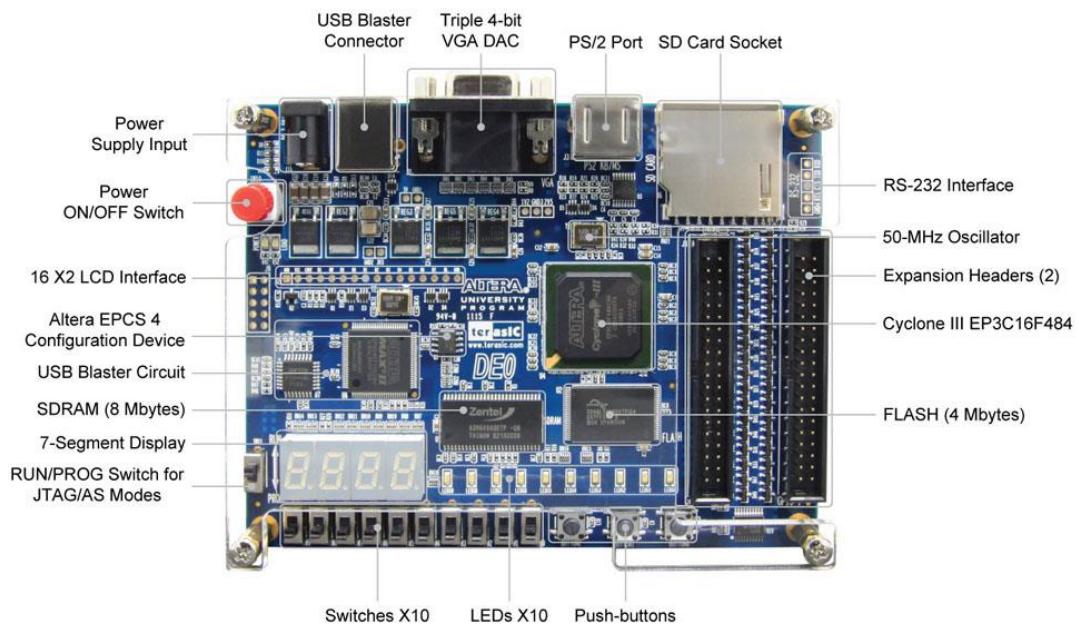


Figura 1: FPGA Cyclone III

## Raspberry Pi Model 2B

Raspberry Pi Model 2B è una delle piattaforme embedded più conosciute e utilizzate al mondo. Questa scheda è una scelta ideale per una vasta gamma di progetti, in particolare all'Internet delle cose (IoT).

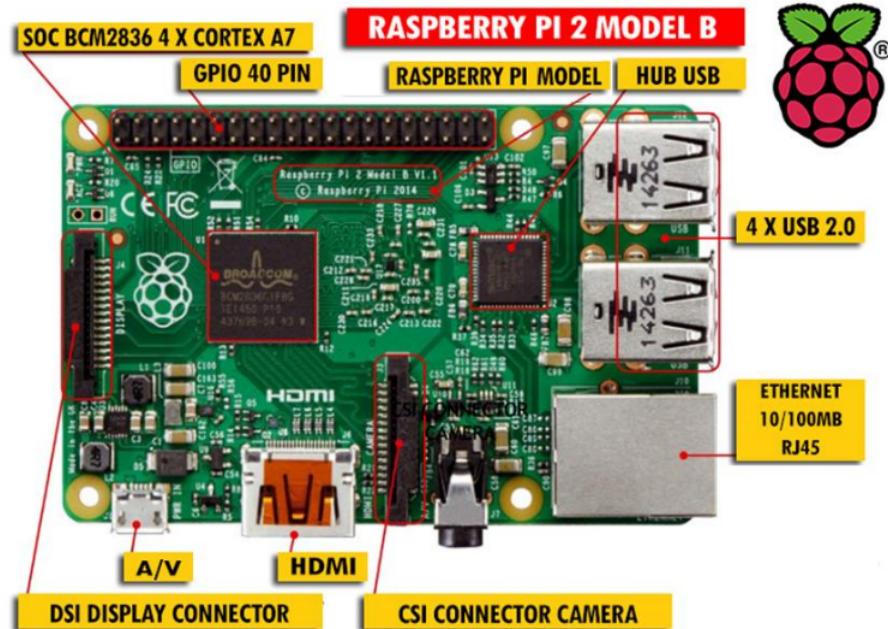


Figura 2: Raspberry Pi Model 2B

Di seguito le sue specifiche:

1. **Processore:** quad-core ARM Cortex-A7 con clock a 900 MHz;
2. **Memoria RAM:** 1 GB di memoria RAM;
3. **Porte USB:** 4 porte USB 2.0;
4. **Ethernet:** integrata;
5. **HDMI:** supporta l'uscita video Full HD (1080p), consentendo la connessione a monitor e televisori;
6. **GPIO (General-Purpose Input/Output):** 40 pin GPIO, che consentono di connettere sensori, attuatori e altri dispositivi per vari progetti;
7. **Alimentazione:** può essere alimentata tramite un adattatore microUSB da 5V;
8. **Sistema Operativo:** Linux (come Raspbian), Windows 10 IoT Core e altri.

# Struttura del moltiplicatore 16x16 bit

## Come funziona l'algoritmo

L'algoritmo di moltiplicazione 16x16 bit si compone di vari passaggi per ottenere il risultato desiderato.

Considerando due operandi  $A$  e  $B$  di 16 bit ciascuno, essi verranno scomposti in due parti più piccole, da 8 bit ciascuno come segue:

MSB	14	13	12	11	10	9	8	7	6	5	4	3	2	1	LSB
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
A[15:8]								A[7:0]							

Figura 3: Suddivisione dell'operando A

Successivamente, i passi di moltiplicazione sono suddivisi in operazioni più gestibili. Le operazioni coinvolgono la moltiplicazione tra le parti alte di  $A$  e  $B$ , la moltiplicazione tra le parti basse di  $A$  e  $B$ , e varie somme di prodotti intermedi.

Per comprendere meglio i passaggi, scomponiamo i due operandi come segue:

$$A[15 : 0] = A[15 : 8] \times 2^8 + A[7 : 0] \times 2^0 \quad B[15 : 0] = B[15 : 8] \times 2^8 + B[7 : 0] \times 2^0 \quad (1)$$

A questo punto se consideriamo di effettuare la moltiplicazione tra gli operandi, si ottiene:

$$\begin{aligned} A[15 : 0] \times B[15 : 0] &= (A[15 : 8] \times 2^8 + A[7 : 0] \times 2^0) \times (B[15 : 8] \times 2^8 + B[7 : 0] \times 2^0) \\ &= A[15 : 8] \times B[15 : 8] \times 2^{16} + (A[15 : 8] \times B[7 : 0] + A[7 : 0] \times B[15 : 8]) \times 2^8 + A[7 : 0] \times B[7 : 0] \end{aligned} \quad (2)$$

Dall'equazione 2 si ottiene l'algoritmo implementato, dove vengono effettuate più moltiplicazioni, in cascata tra le parti scomposte degli operandi, e sommate sequenzialmente. In figura 4 viene mostrato uno schema a blocchi della soluzione proposta.

L'intero processo sfrutta sia la proprietà distributiva che le proprietà delle potenze di 2, come l'espansione di 256 come  $2^8$ . Questo consente di semplificare le operazioni complesse e ridurre il carico computazionale.

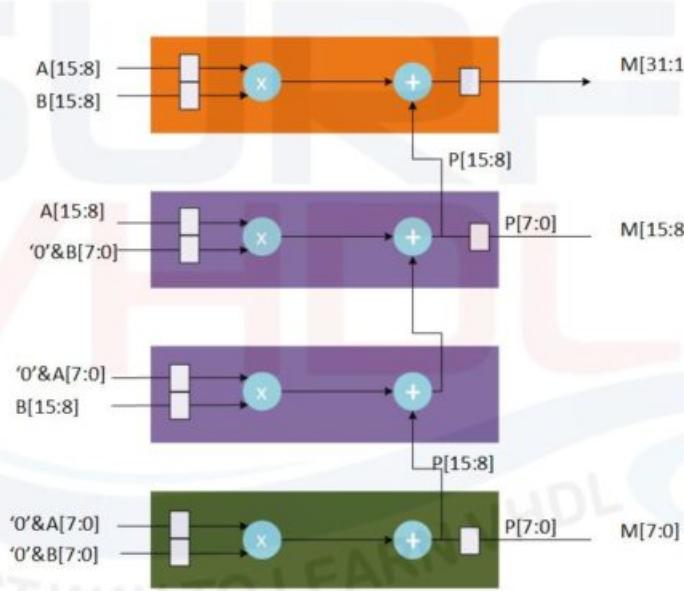


Figura 4: Schema a blocchi della moltiplicazione.

fonte: [surf-vhdl.com](http://surf-vhdl.com)

Una moltiplicazione binaria generalmente è svolta eseguendo una serie di somme e prodotti parziali. Dal punto di vista della logica combinatoria si mette in AND logica il moltiplicando con i diversi bit del moltiplicatore, se il moltiplicatore è composto da n bit si hanno n prodotti parziali potenziali ognuno dei quali richiede una somma per costruire il risultato. Avere in posizioni diverse i bit del moltiplicatore da considerare implica che il moltiplicando deve essere opportunamente "traslato": ogni posizione ha peso doppio, quindi si moltiplica per due e questo comporta lo spostamento del prodotto parziale di una posizione a sinistra per ogni bit in più. Questo approccio nel caso in cui si stia lavorando con dei numeri binari di grande dimensione comporta uno sforzo computazionale elevato.

Esempio:  $10100_2 \times 11100_2$

			1	0	1	0	0	$\times$
			1	1	1	0	0	=
				0	0	0	0	+
				0	0	0	0	+
				1	0	1	0	+
				1	0	1	0	+
				1	0	1	0	+
				1	0	1	0	=
1	0	0	0	1	1	0	0	

Tabella 1: Esempio di moltiplicazione binaria

## Descrizione dell'algoritmo di moltiplicazione

L'entity VHDL `mult_sgn_break_16x16` implementa un moltiplicatore a 16 bit che prende in input due operandi, *i\_ma* e *i\_mb*, e produce in output il risultato della moltiplicazione, *o\_m*, rappresentato su 32 bit.

Il moltiplicatore utilizza una struttura scomposta per calcolare il prodotto tra i due operandi. Il processo principale, denominato *p\_mult*, è sensibile ai segnali di clock (*i\_clk*) e di reset (*i\_rstb*). Durante la fase di reset, tutti i segnali intermedi vengono inizializzati a zero per garantire un corretto avvio del modulo.

Il calcolo della moltiplicazione avviene all'interno del processo *p\_mult*. I segnali intermedi (*r\_ma\_hi*, *r\_ma\_lo*, *r\_mb\_hi*, *r\_mb\_lo*, *r\_m\_hi*, *r\_m\_md*, *r\_m\_lo*, *r\_m*) vengono utilizzati per memorizzare i valori intermedi durante il calcolo (vedi Listato 2).

```

1 -----  

2 -- Project : Moltiplicatore a 16bit          --  

3 -- Author  : Brescia Luca                    --  

4 --           Loda Michele                   --  

5 --           Pezzottini Simone                --  

6 -- Date   : AY2022/2023                      --  

7 -- Company : UniBS                         --  

8 -- File   : mul16.vhd                       --  

9 -----  

10  

11 library ieee;  

12 use ieee.std_logic_1164.all;  

13 use ieee.numeric_std.all;  

14  

15 entity mult_sgn_break_16x16 is  

16   generic (  

17     NUM_CYCLES : integer := 16 --! Numero di cicli per effettuare la moltiplicazione  

18   );  

19   port (  

20     i_clk    : in  std_logic;  

21     i_rstb   : in  std_logic;  

22     i_en     : in  std_logic;  

23     i_ma     : in  std_logic_vector(15 downto 0);  

24     i_mb     : in  std_logic_vector(15 downto 0);  

25     o_m      : out std_logic_vector(31 downto 0);  

26     o_rdy    : out std_logic           --! output ready  

27   );  

28 end mult_sgn_break_16x16;  

29  

30 architecture rtl of mult_sgn_break_16x16 is  

31  

32   -- Segnali intermedi per il calcolo della moltiplicazione  

33   signal r_ma_hi  : signed(7 downto 0);  -- Parte alta del primo operando (A[15:8])  

34   signal r_ma_lo  : signed(8 downto 0);  -- Parte bassa del primo operando con bit di segno (A[7:0])  

35   signal r_mb_hi  : signed(7 downto 0);  -- Parte alta del secondo operando (B[15:8])  

36   signal r_mb_lo  : signed(8 downto 0);  -- Parte bassa del secondo operando con bit di segno (B[7:0])  

37   signal r_m_hi   : signed(15 downto 0); -- Moltiplicazione tra parti alte (A[15:8] * B [15:8])  

38   signal r_m_md   : signed(16 downto 0); -- Moltiplicazione tra parti alte e basse sommate (A [15:8] * B[7:0] + A[7:0] * B[15:8])  

39   signal r_m_lo   : signed(17 downto 0); -- Moltiplicazione tra parti basse (A[7:0] * B[7:0]) con bit di segno esteso  

40   signal r_m      : signed(31 downto 0); -- Risultato della moltiplicazione finale  

41   signal counter  : integer;           -- contatore cicli di clock  

42  

43 begin  

44   o_m <= std_logic_vector(r_m);  -- Assegnamento del risultato alla porta di output o_m  

45  

46   -- Calcolo della moltiplicazione a 16 bit  

47   r_m_hi <= r_ma_hi * r_mb_hi;  

48   r_m_md <= r_ma_hi * r_mb_lo + r_mb_hi * r_ma_lo;  

49   r_m_lo <= r_ma_lo * r_mb_lo;  

50  

51   p_mult : process(i_clk, i_rstb)  

52   begin  

53     if (i_rstb = '1') then  

54       -- Reset dei segnali intermedi  

55       r_ma_hi <= (others => '0');  

56       r_ma_lo <= (others => '0');  

57       r_mb_hi <= (others => '0');  

58       r_mb_lo <= (others => '0');  

59       r_m    <= (others => '0');  

60       counter <= 0;  

61       o_rdy  <= '0';  

62     elsif (rising_edge(i_clk)) then  

63       if i_en = '1' then  

64         -- Assegnazione dei valori ai segnali intermedi durante il ciclo di clock  

65         r_ma_hi <= signed(i_ma(15 downto 8));        -- Assegnazione della parte alta del primo operando  

66         r_ma_lo <= signed('0' & i_ma(7 downto 0)); -- Assegnazione della parte bassa del primo operando con estensione del bit di segno  

67         r_mb_hi <= signed(i_mb(15 downto 8));        -- Assegnazione della parte alta del secondo operando

```

```

68      r_mb_lo <= signed('0' & i_mb(7 downto 0)); -- Assegnazione della parte bassa del secondo
69      operandi con estensione del bit di segno
70      r_m      <= r_m_hi & "00000000000000000000" + resize(r_m_md & "00000000", 32) + resize(r_m_lo,
71      32); -- Calcolo del risultato finale della moltiplicazione
72      --! Aumento il numero di cicli effettuati per la moltiplicazione, se li supero segnalo
73      output ready
74      counter <= counter + 1;
75      if counter >= NUM_CYCLES then
76          o_rdy <= '1';
77      end if;
78      else
79          counter <= 0;
80          o_rdy <= '0';
81      end if;
82      end if;
83      end process p_mult;
84  end rtl;

```

Listing 1: **mul16.vhd** - architettura del moltiplicatore a 16 bit

Il risultato finale della moltiplicazione viene assegnato alla porta di output *o\_m* come una stringa binaria convertita in un vettore di segnali di tipo *std\_logic\_vector*.

Il modulo VHDL **mul16** è stato progettato per eseguire moltiplicazioni a 16 bit in modo efficiente e preciso, fornendo un'implementazione hardware ottimizzata per l'FPGA Cyclone III EP3C16F484C6.

## Motivazioni della dimensione dei dati

Nel progetto del moltiplicatore a 16 bit la scelta della dimensione dei dati influenza sull'ottimizzazione delle prestazioni e nell'occupazione delle risorse. La dimensione dei dati si riferisce alla quantità di bit utilizzati per rappresentare i valori di input e output del moltiplicatore.

Una dimensione di 16 bit permette di gestire una vasta gamma di numeri interi senza introdurre una complessità eccessiva. I dati a 16 bit possono rappresentare valori compresi tra  $-2^{15}$  e  $2^{15} - 1$ . Nel contesto di un moltiplicatore, è spesso necessario eseguire operazioni su numeri relativamente piccoli, ma con una precisione sufficiente per evitare overflow o underflow.

Dimensioni maggiori dei dati potrebbero richiedere risorse hardware aggiuntive, aumentando il consumo energetico e la latenza. Infine, le operazioni aritmetiche e logiche su dati a 16 bit sono ben supportate dalle librerie standard del linguaggio VHDL, semplificando il processo di sviluppo e di debugging.

# Progettazione e implementazione in VHDL

## Descrizione del linguaggio VHDL

Il VHDL (*VHSIC Hardware Description Language*) è un linguaggio di descrizione hardware utilizzato per descrivere circuiti digitali complessi.

Il linguaggio permette di modellare il comportamento e la struttura dei circuiti, consentendo di sviluppare soluzioni hardware in modo efficiente.

Le principali caratteristiche del VHDL includono:

- **Descrizione Comportamentale:** Il VHDL consente di definire il comportamento di un circuito attraverso processi. Questi processi contengono istruzioni sequenziali che modellano l'evoluzione del circuito nel tempo.
- **Descrizione Strutturale:** È possibile definire la struttura di un circuito attraverso la connessione di componenti predefiniti. Questo approccio permette di creare circuiti complessi combinando blocchi più semplici.
- **Tipi di Dati:** VHDL offre una vasta gamma di tipi di dati, tra cui booleani, interi, vettori e record. Questi tipi consentono di modellare sia segnali digitali che dati analogici.
- **Sintassi Gerarchica:** È possibile definire circuiti a più livelli di gerarchia, suddividendo la progettazione in moduli più piccoli e riutilizzabili.
- **Simulazione:** Uno dei vantaggi chiave del VHDL è la possibilità di eseguire simulazioni per verificare il comportamento del circuito prima della fase di implementazione hardware.

Il processo di sviluppo in VHDL inizia con la definizione dei moduli e dei componenti necessari. Questi moduli vengono collegati tra loro per creare il circuito completo. Successivamente, vengono scritti processi per descrivere il comportamento dei singoli moduli.

Una volta completata la descrizione in VHDL, è possibile eseguire simulazioni per testare il funzionamento del circuito in diverse condizioni. Questo approccio aiuta a individuare errori e problemi prima della fase di implementazione hardware.

In conclusione, il VHDL è uno strumento potente per la progettazione di circuiti digitali. La sua combinazione di descrizione comportamentale e strutturale, insieme alla capacità di simulazione, lo rende uno strumento essenziale nell'industriaia dell'elettronica digitale.

## Codice implementato

L'architettura `basic_mult` rappresenta il top level dell'architettura implementata. Essa costituisce un sistema in cui diverse entità lavorano sinergicamente per realizzare l'obiettivo complessivo del progetto. Gestisce

l'interconnessione tra entità, tra cui i componenti personalizzati e i moduli predefiniti forniti durante il corso.

Il modulo **spi**, che abilita l'interfacciamento tra dispositivi attraverso una comunicazione seriale sincrona. Questo modulo viene utilizzato per stabilire una comunicazione con dispositivi esterni. Nel nostro caso, con la scheda di sviluppo *RaspberryPi*.

Parimenti, il modulo **blink heartbeat** è un'entità, fornita durante il corso, che non richiede modifiche significative all'integrazione. Esso fornisce un indicatore visivo dello stato operativo del sistema tramite il lampeggio di un LED.

L'integrazione delle varie entità all'interno dell'architettura **basic\_mult** richiede la sincronizzazione dei segnali di controllo, flussi di dati e temporizzazioni. Per questo motivo è stata creata una macchina a stati che regola il funzionamento del sistema. Le varie connessioni tra le entity garantiscono la propagazione del segnale una volta che viene abilitato dalla macchina a stati.

Essa si suddivide in due signal:

1. **current state**: stato corrente della macchina a stati;
2. **next state**: stato successivo della macchina a stati, valorizzato al verificarsi di una condizione ben precisa.

Inoltre, sono stati creati vari processi che, parallelamente, gestiscono il funzionamento del sistema completo.

Di seguito, viene riportato per intero il codice dell'entity **basic\_mult**, che costituisce il top-level per la nostra architettura.

```

1 -----
2 -- Project : Moltiplicatore a 16bit      --
3 -- Author  : Brescia Luca                --
4 --          Loda Michele                 --
5 --          Pezzottini Simone            --
6 -- Date   : AY2022/2023                  --
7 -- Company: UniBS                      --
8 -- File   : basic_mult.vhd              --
9 -----
10
11 library IEEE;
12
13 use IEEE.STD_LOGIC_1164.all;
14 use IEEE.NUMERIC_STD.all;
15
16 entity basic_mult is
17   port
18   (
19     CLOCK_50 : in std_logic;           --! Clock di sistema a 50 MHz
20     KEY      : in std_logic_vector(2 downto 0); --! Segnali di controllo push buttons
21     LEDG     : out std_logic_vector(9 downto 0); --! Segnali di output per i LED
22     GPIO_1   : inout std_logic_vector(31 downto 0) --! Segnali di input/output GPIO
23   );
24 end basic_mult;
25
26 architecture top_arch of basic_mult is
27   --! Definizioni costanti
28   constant DATA_W      : integer := 32;  --! Dimensione del dato
29   constant Nbit        : integer := 5;    --! Numero di bit per contenere il dato
30   constant all_zeros   : std_logic_vector(DATA_W - 1 downto 0) := (others => '0');  --!
31   --! Dati in ingresso SPI
32   --! Stati per la macchina a stati
33   type my_states is (STATE_WAIT_NEW_DATA, STATE_START_MULTIPLY, STATE_MULT_READY,
34                       STATE_DATA_SENT); --! Stati possibili per la macchina a stati
35   signal current_state : my_states;  --! Stato corrente della macchina a stati
36   signal next_state: my_states;   --! Prossimo stato della macchina a stati
37   --! Definizioni signals
38   signal pb0_synchronizer : std_logic_vector(2 downto 0);           --! Sincronizzatore
39   --! del pushbutton per segnale di reset
40   signal SYS_SPI_SCK      : std_logic;                                := '0'; --! Pin di clock SPI
41   signal SYS_SPI_MOSI     : std_logic;                                := '0'; --! Pin di output dati
42   --! SPI
43   signal SYS_SPI_MISO     : std_logic;                                := '0'; --! Pin di input dati

```

```

    SPI
40 signal spi_data_in      : std_logic_vector(DATA_W - 1 downto 0);      --! Dati in ingresso
    SPI
41 signal spi_data_out     : std_logic_vector(DATA_W - 1 downto 0);      --! Dati in uscita SPI
42 signal mult_data_a      : std_logic_vector(DATA_W/2 - 1 downto 0);      --! Copia del dato in
    ingresso
43 signal mult_data_b      : std_logic_vector(DATA_W/2 - 1 downto 0);      --! Copia del dato in
    ingresso
44 signal result           : std_logic_vector(DATA_W - 1 downto 0);      --! Risultato della
    moltiplicazione
45 signal reset             : std_logic;                                := '0';      --! Flag di reset
46 signal enable_clk        : std_logic;                                := '0';      --! Flag di
    abilitazione clock per l'entity moltiplicatore
47 signal newdata           : std_logic;                                := '0';      --! Flag di presenza
    nuovo dato da inviare
48 signal multready         : std_logic;                                := '0';      --! Flag di
    segnalazione dato moltiplicato corretto
49 signal datasent          : std_logic;                                := '0';      --! Flag di invio
    corretto del dato
50
51 begin
52     --! Assegnazione dei segnali SPI ai pin fisici GPIO
53     SYS_SPI_SCK <= GPIO_1(7);    --! Segnale di clock spi
54     SYS_SPI_MOSI <= GPIO_1(5);   --! Segnale di input per FPGA (Slave)
55     GPIO_1(3) <= SYS_SPI_MISO;  --! Segnale di output per FPGA (Slave)
56
57     --! Gestione del lampeggio del LED
58     blink_hb : entity work.blink_heartbeat port map(
59         CLK => CLOCK_50,
60         LED => LEDG(0)
61     );
62
63     --! Istanza del modulo SPI
64     spi_inst : entity work.spi
65     generic
66     map (
67         DATA_W => 32,
68         Nbit    => 5
69     )
70     port
71     map (
72         CLK      => CLOCK_50,
73         reset    => reset,
74         DATA_IN  => spi_data_out,
75         DATA_OUT => spi_data_in,
76         RD       => datasent,
77         WR       => newdata,
78         SCK      => SYS_SPI_SCK,
79         MOSI     => SYS_SPI_MOSI,
80         MISO     => SYS_SPI_MISO
81     );
82
83     --! Istanza del moltiplicatore 16x16
84     mult_inst : entity work.mult_sgn_break_16x16
85     port
86     map (
87         i_clk    => CLOCK_50,
88         i_rstb   => reset,
89         i_en     => enable_clk,
90         i_ma     => mult_data_a,
91         i_mb     => mult_data_b,
92         o_m     => result,
93         o_rdy   => multready
94     );
95
96     --! Processo principale per la gestione della macchina a stati per la realizzazione del
97     --! moltiplicatore senza segno spi
98     --! Attende un dato in ingresso su spi formattato come segue: (MOLTIPLICATORE << 16 |
99     --! MOLTIPLICANDO) dove moltiplicando e moltiplicatore sono due interi senza segno a 16 bit
100    main_process : process (current_state, newdata, multready, datasent)
101    begin
102        --! Gestione della macchina a stati
103        case current_state is
104            --! Attesa del nuovo dato
105            when STATE_WAIT_NEW_DATA =>

```

```

104     enable_clk <= '0';          --! Reset clk dell'istanza moltiplicatore
105     LEDG(9 downto 2) <= (others => '0');  --! Reset dei led presenti su FPGA
106     next_state <= STATE_WAIT_NEW_DATA;
107     --! Controllo presenza flag di nuovo dato e che il dato sia diverso da 0
108     if newdata = '1' and (spi_data_in /= all_zeros) then
109     --! Divisione del dato di ingresso a 32bit come dato_a e dato_b
110     mult_data_a <= spi_data_in(15 downto 0);  --! Moltiplicando parte bassa (0 - 15) del
111     dato in ingresso
112     mult_data_b <= spi_data_in(31 downto 16); --! Moltiplicatore parte alta (16- 31) del
113     dato in ingresso
114     next_state <= STATE_START_MULTIPLY;        --! Avanzamento di stato della macchina a
115     stati
116     end if;
117
118     --! Avvio moltiplicazione e attesa completamento
119     when STATE_START_MULTIPLY =>
120     enable_clk <= '1';  --! Abilitazione del segnale di clock per la entity
121     mult_sgn_break_16x16
122     next_state <= STATE_START_MULTIPLY;
123     --! Attesa flag moltiplicazione terminata
124     if multready = '1' then
125     spi_data_out <= result;           --! Copia del risultato sul signal che inviera' su spi
126     il valore
127     next_state <= STATE_MULT_READY;   --! Avanzamento di stato della macchina a stati
128     end if;
129
130     --! Moltiplicazione completata e attesa di fine invio del dato su spi
131     when STATE_MULT_READY =>
132     enable_clk <= '0'; --! Arresto del segnale di clock per la entity mult_sgn_break_16x16
133     next_state <= STATE_MULT_READY;
134     --! Attesa flag dato inviato
135     if datasent = '1' then
136     next_state <= STATE_DATA_SENT;    --! Avanzamento di stato della macchina a stati
137     end if;
138
139     --! Termine delle operazioni
140     when STATE_DATA_SENT =>
141     LEDG(7) <= '1'; --! Segnalazione tramite accensione del led7 delle operazioni concluse
142     next_state <= STATE_DATA_SENT;--! Loopback del nuovo stato
143     --! Se il flag di dato inviato e' attivo
144     if datasent = '1' then
145     LEDG(8) <= '1'; --! Segnalazione tramite accensione del led8 del corretto invio del dato
146     end if;
147     when others =>
148     LEDG(9) <= '1'; --! Segnalazione tramite accensione del led9 della metastabilita'
149   end case;
150   end process;
151
152   --! Gestione dell'avanzamento di stato della macchina a stati
153   state_memory : process (CLOCK_50, reset) --! Stati possibili per la macchina a stati
154   begin
155     --! Gestione del segnale di reset
156     if reset = '1' then
157       current_state <= STATE_WAIT_NEW_DATA;    --! Reset macchina a stati
158     elsif (rising_edge(CLOCK_50)) then
159       current_state <= next_state;            --! Avanzamento di stato
160     end if;
161   end process;
162
163   --! Gestione del segnale di reset
164   reset_handle : process (CLOCK_50, KEY)
165   begin
166     --! Trigger delle operazioni sul fronte positivo del clock FPGA
167     if (rising_edge(CLOCK_50)) then
168       --! Sincronizzazione del pushbutton0 e generazione del segnale di reset
169       pb0_synchronizer(2 downto 1) <= pb0_synchronizer(1 downto 0);
170       pb0_synchronizer(0)           <= KEY(0);
171
172       --! Fronte positivo indica che il pushbutton0 e' stato rilasciato
173       if pb0_synchronizer(2 downto 1) = "01" then
174         LEDG(1) <= '1';  --! Segnalazione tramite accensione del led1 l'attivazione del segnale
175         di reset
176         reset <= '1';  --! Attivazione segnale di reset
177         --! Se e' stato attivato al clock precedente il segnale di reset
178         elsif (reset = '1') then

```

```

173     LEDG(1) <= '0';    --! Segnalazione tramite spegnimento del led1 la disattivazione del
174     segnale di reset
175     reset    <= '0';    --! Disattivazione segnale di reset
176     end if;
177   end if;
178   end process;
179 end top_arch;

```

Listing 2: Top level implementato per la nostra architettura

## Approccio di progettazione

La scelta di implementare una macchina a stati per gestire il processo di moltiplicazione all'interno del progetto è motivata dalla necessità di coordinare e controllare le diverse fasi coinvolte nel funzionamento del progetto. La moltiplicazione in sé è un'operazione complessa che richiede diverse fasi di calcolo, sincronizzazione e gestione dei segnali di controllo. In questo progetto viene richiesto il corretto funzionamento della comunicazione SPI, da cui bisogna ottenere i due operandi e inviare successivamente il risultato della loro moltiplicazione.

La macchina a stati offre un'organizzazione strutturata per gestire queste diverse fasi in modo sequenziale e controllato. Ciascuno stato rappresenta una fase specifica del processo di funzionamento, consentendo una suddivisione chiara e modulare delle operazioni coinvolte. Questo approccio non solo semplifica la progettazione e l'implementazione, ma contribuisce anche a una maggiore comprensione del flusso di lavoro.

In particolare, con riferimento al listato 2:

- **STATE\_WAIT\_NEW\_DATA** è lo stato responsabile della lettura dei dati in ingresso. Il sistema resta in questo stato finché non viene segnalata la presenza di un nuovo dato di ingresso *newdata* con l'aggiunta del controllo che sia diverso da zero. Questo stato è anche lo stato indotto dal segnale di reset.
- **STATE\_START\_MULTIPLY** è lo stato indotto dopo che è stato correttamente ricevuto un dato da SPI. Avvia il processo di moltiplicazione attivando il clock corrispondente all'entity *mult\_inst* e attende il segnale *o\_rdy* corrispondente alla presenza del dato moltiplicato correttamente.
- **STATE\_MULT\_READY** è lo stato indotto dopo che la moltiplicazione è terminata. Esso avvia il processo di invio del dato tramite l'entity *spi\_inst* e attende finché non è segnalato il corretto invio del dato.
- **STATE\_DATA\_SENT** è lo stato indotto quando il dato è stato correttamente inviato. Esso segna la fine di tutte le operazioni. Il sistema resta in questo stato di *IDLE* finché un nuovo segnale di reset non è ricevuto.

Ogni transizione tra stati è governata da un insieme di condizioni ben definite, che aiuta a garantire che le operazioni avvengano nel momento giusto e con il timing corretto. Ciò è particolarmente importante in un sistema sincrono come quello in questione, dove il corretto sequenziamento delle operazioni è essenziale per ottenere risultati corretti.

# Simulazione e verifica

## Introduzione alla Simulazione con Altera ModelSim per l'Architecture "basic\_mult"

La simulazione è un passo cruciale nel processo di progettazione hardware, poiché consente di verificare il funzionamento e l'interazione delle diverse componenti del sistema. Nel contesto del progetto basato sull'architecture "basic\_mult" l'obiettivo è quello di realizzare un moltiplicatore a 16 bit. Per verificare l'effettivo funzionamento di tale componente, è stato utilizzato l'ambiente di simulazione Altera ModelSim.

Di seguito viene riportato il codice utilizzato per la creazione del testbench:

```
1  -----  
2  -- Project : Moltiplicatore a 16bit      --  
3  -- Author  : Brescia Luca                 --  
4  --          Loda Michele                  --  
5  --          Pezzottini Simone             --  
6  -- Date   : AY2022/2023                   --  
7  -- Company : UniBS                      --  
8  -- File   : tb_spi.vhd                   --  
9  -----  
10 library ieee;  
11 use ieee.std_logic_1164.all;  
12 use ieee.numeric_std.all;  
13  
14 entity tb_spi is  
15 end tb_spi;  
16  
17 architecture sim of tb_spi is  
18   COMPONENT testbench  
19     port  
20   (  
21     CLOCK_50 : in std_logic;           --! Clock di sistema a 50 MHz  
22     SW       : in std_logic_vector(9 downto 0);    --! Dati in ingresso da interruttore  
23     KEY      : in std_logic_vector(2 downto 0);    --! Segnali di controllo push buttons  
24     LEDG     : out std_logic_vector(9 downto 0);   --! Segnali di output per i LED  
25     GPIO_1   : inout std_logic_vector(31 downto 0) --! Segnali di input/output GPIO  
26   );  
27   END COMPONENT;  
28  
29   -- COSTANTI  
30   constant clk_period: integer := 20;      -- ns  
31   constant sck_period: integer := 200;     -- ns  
32   constant data1: integer := 131143;    -- max 65535  
33   constant data2: integer := 18;        -- max 65535  
34   constant DATA_W: integer := 32;        -- dimensione del dato  
35   constant Nbit: integer := 5;          -- numero di bit per contenere il dato  
36  
37  
38   --inout  
39   signal GPIO_1    : std_logic_vector(DATA_W-1 downto 0);  
40  
41   --Inputs  
42   signal CLOCK_50  : std_logic:= '0';  
43   signal KEY       : std_logic_vector(2 downto 0):= (others => '0');  
44   signal SW        : std_logic_vector(9 downto 0):= (others => '0'); -- Array switch  
45   signal LEDG      : std_logic_vector(9 downto 0);  
46
```

```

47  signal reset:      std_logic := '0';
48  signal reset_sim:   std_logic := '1';
49  signal SCK:        std_logic := '0';
50  signal stop_SCK:   boolean   := true;
51  signal MOSI:       std_logic := '0';
52  signal MISO:       std_logic := '0';
53  signal wr_local:  std_logic := '0';
54  signal data_in:    std_logic_vector(DATA_W-1 downto 0);
55  signal data_out:   std_logic_vector(DATA_W-1 downto 0);
56
57
58
59 begin
60   SCK  <= GPIO_1(7);  --! Segnale di clock spi
61   MOSI <= GPIO_1(5);  --! Segnale di input per FPGA (Slave)
62   MISO <= GPIO_1(3);
63   dut: entity work.basic_mult
64     port map (
65       CLOCK_50 => CLOCK_50,
66       SW => SW,
67       KEY => KEY,
68       LEDG => LEDG,
69       GPIO_1 => GPIO_1
70     );
71
72   sim_spi: entity work.send_bits
73     generic map (
74       DATA_W => DATA_W,
75       Nbit => Nbit
76     )
77     port map (
78       clk => GPIO_1(7),
79       reset => reset_sim,
80       bit_out => GPIO_1(5),
81       data_out => data_out,
82       bit_in => GPIO_1(3),
83       data_in => data_in,
84       ready => wr_local
85     );
86
87   process
88   begin
89     reset <= '1';
90     reset_sim <= '1';
91     wait for 20 ns;
92     reset <= '0';
93     reset_sim <= '0';
94     wait for 20 ns;
95
96     -- invio mcand = 3000 (0x0BB8)
97     data_out <= std_logic_vector(to_unsigned(data1, DATA_W));
98
99     stop_SCK <= false;
100    reset_sim <= '0';
101    wait until wr_local = '1';
102
103    stop_SCK <= true;
104    reset_sim <= '1';
105    wait for 200 ns;
106
107    data_out <= std_logic_vector(to_unsigned(0, DATA_W));
108
109    stop_SCK <= false;
110    reset_sim <= '0';
111    wait until wr_local = '1';
112
113    stop_SCK <= true;
114    reset_sim <= '1';
115    wait for 200 ns;
116
117    stop_SCK <= false;
118    reset_sim <= '0';
119    wait until wr_local = '1';
120
121

```

```

122 stop_SCK <= true;
123 reset_sim <= '1';
124 wait for 200 ns;
125
126 -- visualizzo il risultato
127 report "Risultato: " & integer'image(to_integer(unsigned(data_in)));
128
129 wait;
130 end process;
131
132 process
133 constant mWait: integer := clk_period/2;
134 constant period: time := mWait * 1 ns;
135 begin
136 CLOCK_50 <= '0';
137 loop
138   wait for period;
139   CLOCK_50 <= not CLOCK_50;
140 end loop;
141 end process;
142
143 process
144 constant mWait: integer := sck_period/2;
145 constant period: time := mWait * 1 ns;
146 begin
147 GPIO_1(7) <= '0';
148 loop
149   while not stop_SCK loop
150     wait for period;
151     GPIO_1(7) <= not GPIO_1(7);
152   end loop;
153   GPIO_1(7) <= '0';
154   wait for period;
155 end loop;
156 end process;
157 end sim;
158
159

```

Listing 3: Test bench dell’architettura *basic\_mult*

## Dettagli dello Script di Simulazione

Lo script di simulazione è stato progettato per simulare il funzionamento della moltiplicazione a 16 bit, utilizzando una serie di componenti simulate che rappresentano il comportamento dei segnali e delle interconnessioni tipiche del sistema reale. Le istanze delle entity `spi_inst` e `mult_sgn_break_16x16` rappresentano rispettivamente le parti di comunicazione SPI e moltiplicazione. Inoltre, è presente un blocco per la gestione del segnale di clock e della linea di clock SCK.

### Configurazione degli input

Per simulare la moltiplicazione a 16 bit, vengono forniti input appropriati ai componenti. I valori da moltiplicare vengono forniti alle entity tramite le porte di input `i_ma` e `i_mb`. Gli input vengono preparati e inviati al componente di moltiplicazione `mult_sgn_break_16x16`.

### Generazione del segnale di clock

La simulazione richiede la generazione di un segnale di clock coerente per sincronizzare le operazioni dei componenti. Uno dei processi presenti nello script si occupa di generare il segnale di clock `clk`, che alterna i suoi stati secondo un periodo prestabilito.

### Simulazione della comunicazione SPI

Per simulare la comunicazione SPI è stata utilizzata l’entity `spi_inst`. Viene generata una sequenza di bit da inviare, rappresentata dal segnale `shift_reg`, che viene successivamente trasmesso attraverso le linee MOSI (Master Out Slave In) e MISO (Master In Slave Out). Le operazioni di trasmissione e ricezione dei bit vengono simulate utilizzando il segnale SCK come clock per sincronizzare le transizioni.

## **Simulazione della moltiplicazione**

La moltiplicazione a 16 bit è simulata utilizzando l'entity `mult_sgn_break_16x16`. Viene fornito un input appropriato rappresentato dalle variabili `local_din1` e `local_din2`, alle porte `i_ma` e `i_mb`. Il risultato della moltiplicazione viene salvato nel segnale `local_dout`.

## **Gestione dei segnali di controllo**

I segnali di controllo, come `wr_local` e `rd_local`, vengono utilizzati per sincronizzare l'invio e la ricezione dei dati tra le diverse componenti simulate. Questi segnali sono utilizzati per controllare i cicli di lettura e scrittura dei dati.

## **Visualizzazione dei risultati**

Durante la simulazione, i risultati delle operazioni vengono monitorati e visualizzati attraverso l'uso dell'istruzione `report`. In particolare, il risultato della moltiplicazione viene convertito in un valore intero e viene visualizzato sulla console di simulazione.

## **Considerazioni sull'uso di Altera Modelsim**

L'utilizzo di Altera ModelSim come ambiente di simulazione ha permesso di verificare il corretto funzionamento dell'architettura `basic_mult` in un ambiente controllato e riproducibile. Lo script di simulazione `tb_spi.vhd` ha consentito di testare diverse operazioni, come la comunicazione SPI e la moltiplicazione a 16 bit, fornendo un'anteprima delle prestazioni e del comportamento dell'architettura prima della sua effettiva implementazione su dispositivi hardware. Va notato che il codice di simulazione è stato creato ad hoc per questo progetto e il suo funzionamento è stato verificato in ambiente ModelSim.

# Sintesi e implementazione

## Modulo basic\_mult

### Fase di implementazione

L'architettura del moltiplicatore implementata e riportata di seguito, riceve in input il clock del sistema mediante *signal CLOCK\_50*, il segnale di reset mediante *KEY* e segnali di comunicazione mediante i *GPIO\_1(5)* e *GPIO\_1(3)*. In uscita pilota i *LEDG* e il *GPIO\_1* corrispondente al *MISO* dell'FPGA.

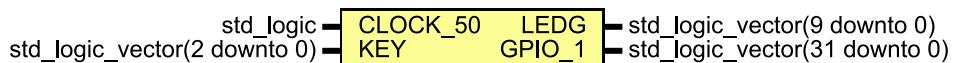


Figura 5: Architettura moltiplicatore implementato

Se i *GPIO\_1* sono posizionati nelle uscite nella figura 5 è solamente per un errore di interpretazione del plugin *TerosHDL* poiche i signal *GPIO\_1* sono definiti *inout*, come riportato in tabella 2.

Di seguito, sono riportati i dettagli dei *signal* utilizzati. I dati riportati sono stati estrapolati dall'output dell'estensione per Visual Studio Code: *TerosHDL*.

Nome porta	Direzione	Tipo	Descrizione
CLOCK_50	in	std_logic	Clock di sistema a 50 MHz
KEY	in	std_logic_vector(2 downto 0)	Segnali di controllo push buttons
LEDG	out	std_logic_vector(9 downto 0)	Segnali di output per i LED
GPIO_1	inout	std_logic_vector(31 downto 0)	Segnali di input/output GPIO

Tabella 2: Segnali del top level dell'architettura

Nome	Tipo	Descrizione
current_state	my_states	Stato corrente della macchina a stati
next_state	my_states	Prossimo stato della macchina a stati
pb0_synchronizer	std_logic_vector(2 downto 0)	Sincronizzatore del pushbutton per segnale di reset
SYS_SPLSCK	std_logic	Pin di clock SPI
SYS_SPI_MOSI	std_logic	Pin di output dati SPI
SYS_SPI_MISO	std_logic	Pin di input dati SPI
spi_data_in	std_logic_vector(DATA_W - 1 downto 0)	Dati in ingresso SPI
spi_data_out	std_logic_vector(DATA_W - 1 downto 0)	Dati in uscita SPI
mult_data_a	std_logic_vector(DATA_W/2 - 1 downto 0)	Copia del dato in ingresso
mult_data_b	std_logic_vector(DATA_W/2 - 1 downto 0)	Copia del dato in ingresso
result	std_logic_vector(DATA_W - 1 downto 0)	Risultato della moltiplicazione
reset	std_logic	Flag di reset
enable_clk	std_logic	Flag di abilitazione clock per l'entity moltiplicatore
newdata	std_logic	Flag di presenza nuovo dato da inviare
multready	std_logic	Flag di segnalazione dato moltiplicato corretto
datasent	std_logic	

Tabella 3: Segnali interni all'architettura *top\_arch* di basic\_mult

Nome	Tipo	Valore	Descrizione
DATA_W	integer	32	Dimensione del dato
Nbit	integer	5	Numero di bit per contenere il dato
all_zeros	std_logic_vector(DATA_W - 1 downto 0)	0	Dati in ingresso SPI

Tabella 4: Costanti utilizzate nell'architettura *top\_arch* di basic\_mult

Nome	Tipo
my_states	STATE_WAIT_NEW_DATA, STATE_START_MULTIPLY, STATE_MULT_READY, STATE_DATA_SENT

Tabella 5: Possibili valori per la macchina a stati nell'architettura *top\_arch* di basic\_mult

La macchina a stati che regola il funzionamento del sistema è la seguente:

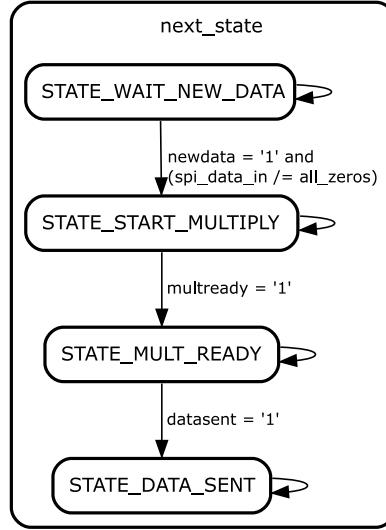


Figura 6: Finite state machine ottenuta con l'estensione *TerosHDL*

Si è verificata la correttezza della macchina a stati esposta in figura 6 ottenendo la visualizzazione della macchina a stati prodotta da *Quartus*:

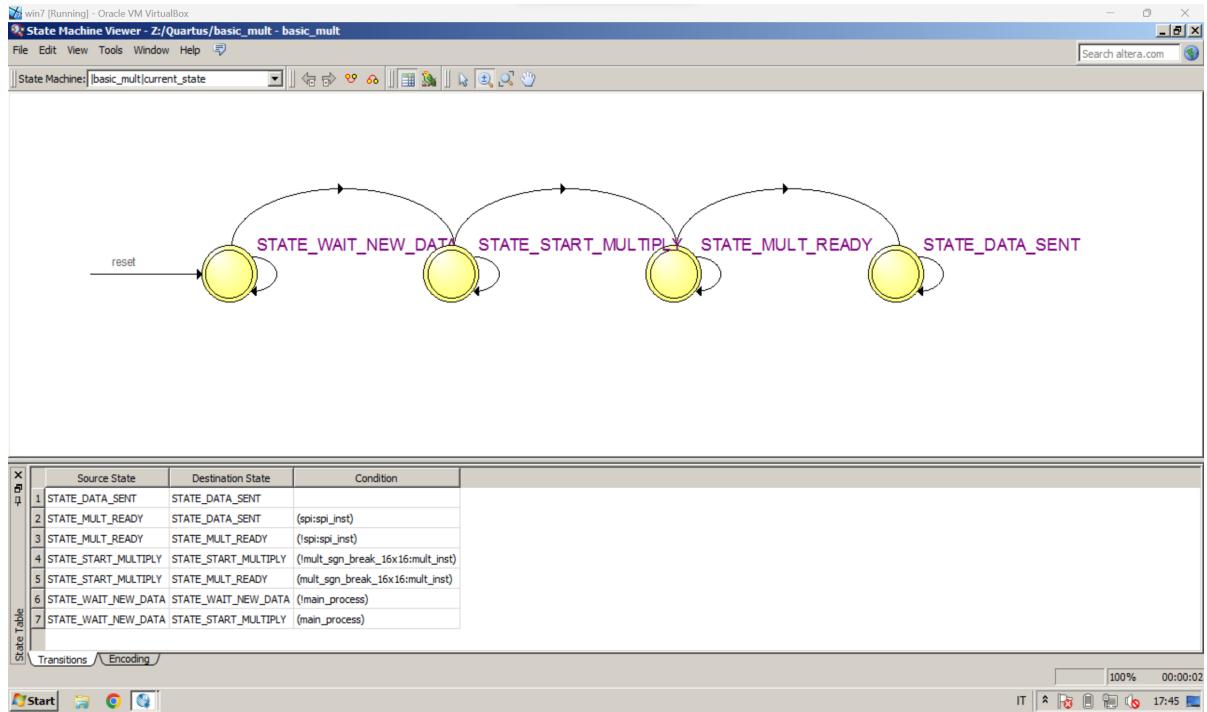


Figura 7: Finite state machine ottenuta con *Quartus*

Si può notare dalle figure 6 e 7 come le macchine a stati ottenute siano equivalenti.

In particolare, osservando la figura 7:

- L'ingresso asincrono della macchina a stati è il comando di *reset*;
- Gli stati sono quelli sopraelencati nella tabella 5.

## Fase di sintesi

La vista RTL del moltiplicatore ottenuta da *Quartus* con la compilazione del codice VHDL è la seguente:

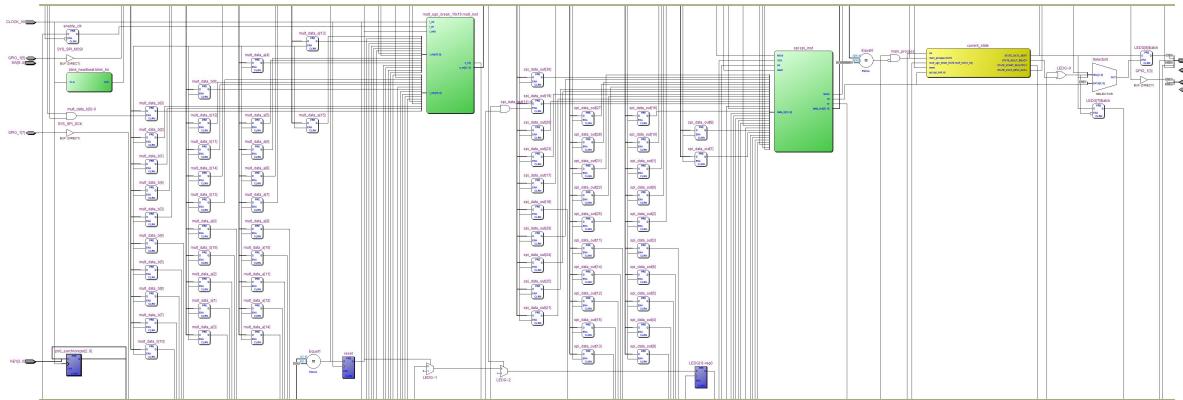


Figura 8: Vista RTL integrale ottenuta con *Quartus*

Di seguito uno zoom dell'immagine su alcune parti di interesse:

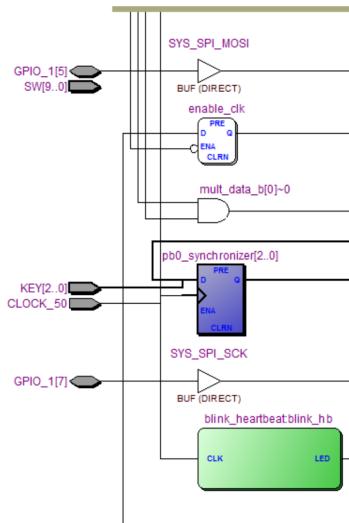


Figura 9: Prima parte della vista RTL ottenuta con *Quartus*

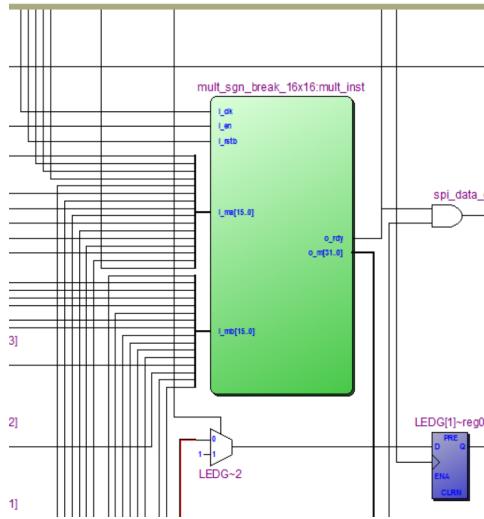


Figura 10: Seconda parte della vista RTL ottenuta con *Quartus*

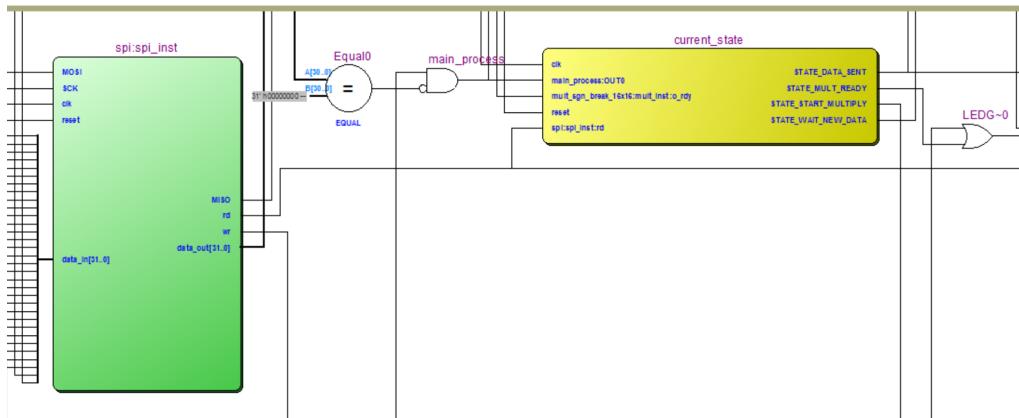


Figura 11: Terza parte della vista RTL ottenuta con *Quartus*

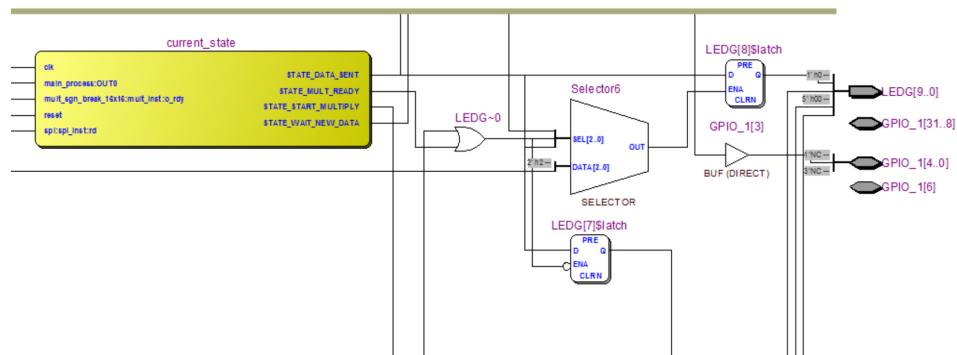


Figura 12: Quarta parte della vista RTL ottenuta con *Quartus*

Salendo di gerarchia, il sistema *basic\_mult* si presenta come *top\_level* dell'architettura. La seguente figura mostra la vista del sistema dall'esterno:



Figura 13: Vista del top level

In particolare, è importante notare che solo *GPIO\_1[3]* è riportato come uscita del sistema poiché, come definito dal file *DE0\_PinAssignment.qpf* fornito durante il corso, corrisponde al *MISO* chè è il pin di output per il dispositivo *SLAVE* nella comunicazione SPI.

## Modulo SPI

### Fase di implementazione

Di seguito viene riportata la vista dell'architettura spi implementata, ottenuta mediante *TerosHDL*.

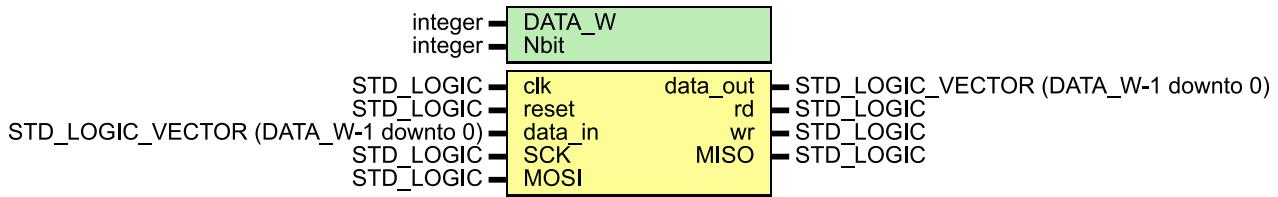


Figura 14: Architettura modulo spi utilizzato

I generic values sono riportati in tabella 6.

Nome	Tipo	Valore	Descrizione
DATA_W	integer	16	Lunghezza dei dati acquisiti/inviai in bit
Nbit	integer	4	$\log_2(\text{DATA\_W})$

Tabella 6: Generic values per l'architettura SPI implementata

In tabella 7 sono riportati gli input e output dell'architettura.

Nome porta	Direzione	Tipo	Descrizione
clk	in	std_logic	Clock
reset	in	std_logic	Reset SPI
data_in	in	std_logic_vector (DATA_W-1 downto 0)	Dati da inviare al master (dimensione DATA_W-1).
data_out	out	std_logic_vector (DATA_W-1 downto 0)	Dati letti dal master posti in uscita (dimensione DATA_W-1).
rd	out	std_logic	Settato a 1 quando l'operazione di invio in MISO è conclusa.
wr	out	std_logic	Settato a 1 quando l'operazione di lettura da MOSI è conclusa.
SCK	in	std_logic	Slave clock
MOSI	in	std_logic	Master Output Slave Input
MISO	out	std_logic	Master Input Slave Output

Tabella 7: Descrizione degli input/output dell'architettura SPI implementata

In tabella 8 sono riportati i segnali implementati nell'architettura.

Segnali	Tipo	Descrizione
spi_value	std_logic_vector(DATA_W-1 downto 0)	Registro bit da inviare in MISO
spi_readvalue	std_logic_vector(DATA_W-1 downto 0)	Registro bit letti da MOSI
sck_synchronizer	std_logic_vector(2 downto 0)	Registro per la sincronizzazione con il clock; in particolare è di lunghezza 3 bit dove i bit 2 e 1 permettono l'identificazione della transizione alto-basso o basso-alto del clock SCK. Questo registro viene shiftato a sinistra per ogni colpo di clock e in posizione 0 acquisisce SCK.
rdcnt	unsigned(Nbit-1 downto 0)	Contatore per la fase di lettura di lunghezza Nbit
wrcnt	unsigned(Nbit-1 downto 0)	Contatore per la fase di scrittura di lunghezza Nbit
feed_me	std_logic	Se settato a 1, l'operazione di invio in MISO è conclusa e se ne può iniziare un'altra
read_me	std_logic	Se settato a 1, l'operazione di acquisizione da MOSI è conclusa e se ne può iniziare un'altra

Tabella 8: Segnali dell'architettura SPI implementata

Di seguito la descrizione dell'algoritmo implementato che segue la descrizione del mode 1 della comunicazione SPI:

- Al fronte di salita del clock si shifta a sinistra di una posizione `sck_synchronizer` e si pone in `1sb SCK`.
- Se `reset` è a 1 (attivo) si azzerano i registri `spi_value`, `spi_readvalue`, `MISO`, `rdcnt`, `wrcnt`, `rd`, `wr`, `read_me`, `feed_me`, `data_out`.
- Se `reset` non è 1:
  - Se `sck_synchronizer(2 downto 1)` è "01" significa che c'è stata una transizione basso-alto, quindi si effettuano le operazioni di trasmissione dei bit:
    - \* Aggiornamento di `spi_value` con uno shift a sinistra: `spi_value <= spi_value(DATA_W-2 downto 0) & '0'`.
    - \* Immissione sulla linea `MISO` del MSB di `spi_value`: `MISO <= spi_value(DATA_W-1)`.
    - \* Incremento del contatore dei bit inviati: `wrcnt <= wrcnt + 1`.
    - \* Nel caso in cui il conteggio dei bit inviati sia completato (`wrcnt = all_ones`), si effettua l'operazione di azzeramento del contatore: `wrcnt <= (others => '0')` e si settano a 1 `feed_me` e `rd`: `feed_me <= '1'`, `rd <= '1'`.
  - Se `sck_synchronizer(2 downto 1)` è "10" significa che c'è stata una transizione alto-basso, quindi si effettuano le operazioni di acquisizione:
    - \* Aggiornamento di `spi_readvalue` con uno shift a sinistra: `spi_readvalue(DATA_W-1 downto 1) <= spi_readvalue(DATA_W-2 downto 0)`.
    - \* Lettura del bit sulla linea `MOSI` e collocamento di esso nel LSB di `spi_readvalue`: `spi_readvalue(0) <= MOSI`.
    - \* Incremento del contatore di bit letti: `rdcnt <= rdcnt + 1`.
    - \* Nel caso in cui il conteggio dei bit letti sia completato (`rdcnt = all_ones`), si effettua l'operazione di azzeramento del contatore: `rdcnt <= (others => '0')` e si settano a 1 `read_me` e `wr`: `read_me <= '1'`, `wr <= '1'`.

- Se `feed_me` è 1 si aggiorna `spi_value` con il nuovo dato da inviare: `spi_value <= data_in` e si mettono a zero i bit relativi a `rd` e `feed_me`: `rd <= '0'`, `feed_me <= '0'`.
- Se `read_me` è 1 si porta in uscita il dato letto: `data_out <= spi_readvalue` e si mettono a zero i bit relativi a `read_me` e `wr`: `read_me <= '0'`, `wr <= '0'`.

## Fase di sintesi

Di seguito viene mostrata la vista RTL dell'entity `spi_inst` mostrata nel listato 3.

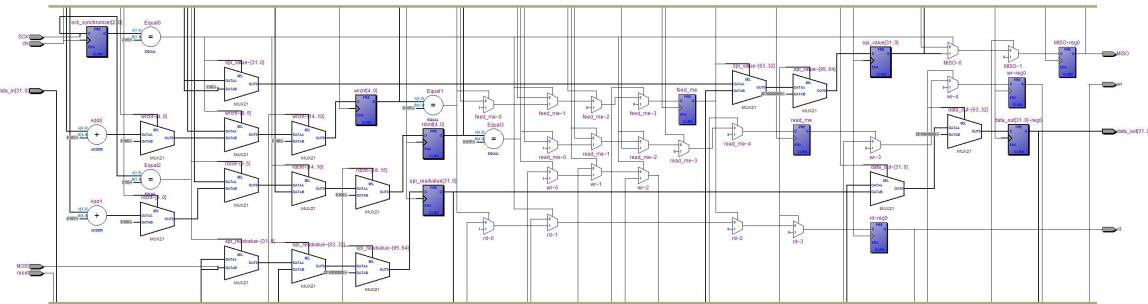


Figura 15: Vista RTL integrale ottenuta con *Quartus* dell'entity SPI

# Programmazione di RaspberryPi utilizzando MATLAB Simulink

## Ruolo di MATLAB Simulink nel progetto

L'integrazione di dispositivi hardware, come RaspberryPi, in progetti di ingegneria è un passo cruciale per realizzare soluzioni complesse e interconnesse. Una delle metodologie di programmazione che si è dimostrata efficace e versatile è l'uso dello strumento "Hardware for RaspberryPi" di MATLAB Simulink.

"Hardware for RaspberryPi" è un potente strumento che permette di creare applicazioni personalizzate per RaspberryPi in modo grafico. Sfruttando l'approccio a blocchi di Simulink è possibile progettare e sviluppare in modo rapido algoritmi, controlli e interfacce utente che possono essere implementati direttamente su RaspberryPi. Questo strumento offre una vasta libreria di blocchi funzionali come il controllo di GPIO, comunicazione con sensori, attuatori e dispositivi esterni.

Un aspetto particolarmente vantaggioso dell'utilizzo di "Hardware for RaspberryPi" è la sua capacità di generare automaticamente il codice C++ ottimizzato per RaspberryPi. Ciò significa che dopo aver creato il modello in Simulink è possibile generare il codice e caricarlo direttamente su RaspberryPi, semplificando il processo di deploy e testing.

È importante notare che "Hardware for RaspberryPi" non è compatibile con le versioni di MATLAB 2022a e successive. È stato testato con successo su MATLAB 2017b, ma non è stato valutato per versioni intermedie tra 2017b e 2022a. Nonostante non sembri esserci alcun problema di compatibilità con la versione di MATLAB, poiché lo strumento si installa correttamente, esso non compila il blocco simulink creato.

In questo progetto si è utilizzato lo strumento "Hardware for RaspberryPi" per gestire la comunicazione SPI come master, impiegando il blocco "SPI Master Transfer". Questa scelta si è rivelata essenziale per coordinare in modo efficiente la comunicazione tra FPGA e RaspberryPi, permettendo lo scambio di dati e il controllo dei segnali nel contesto del progetto in corso.

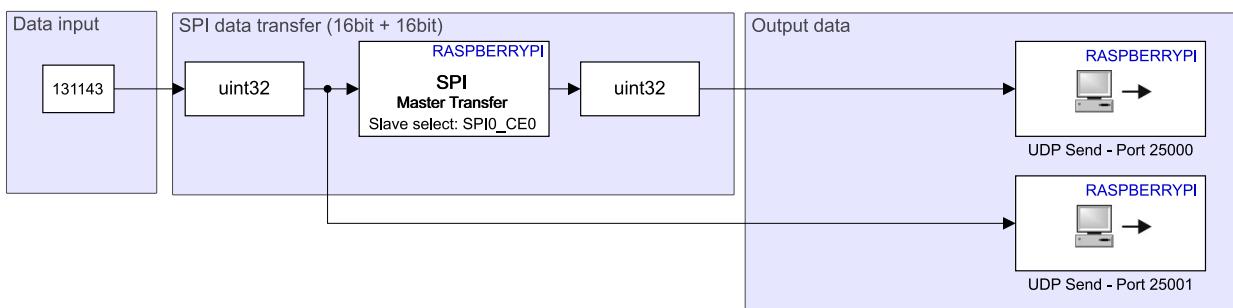


Figura 16: Schema a blocchi simulink implementato

Lo schema mostrato in figura 16 mostra il funzionamento della comunicazione SPI su RaspberryPi. Esso invia periodicamente il numero 131143 che è l'unione di due numeri a 16 bit,  $(0x02 << 16) | 0x47$  (corrispondenti ai numeri 2 e 71 in base 10), che saranno il moltiplicando e il moltiplicatore della moltiplicazione che dovrà eseguire l'FPGA. Il risultato poi verrà ritornato da FPGA e sarà un numero a 32 bit, di valore atteso 142 ossia 0x8E.

I valori che vengono letti in ingresso a RaspberryPi verranno poi inviati su seriale ethernet utilizzando il protocollo UDP.

## Configurazione e comunicazione con RaspberryPi

Per la creazione di un canale di comunicazione tra RaspberryPi e la FPGA Cyclone III si è utilizzato il protocollo di comunicazione SPI.

Di seguito vengono esposti nella tabella 9 i collegamenti da effettuare sulle due board, visualizzabili in figura 17:

FPGA	Raspberry
GPIO1_D1	GPIO8 (CE0)
GPIO1_D3	GPIO9 (MISO)
GPIO1_D5	GPIO10 (MOSI)
GPIO1_D7	GPIO11 (SCK)

Tabella 9: Collegamenti pin to pin delle due board

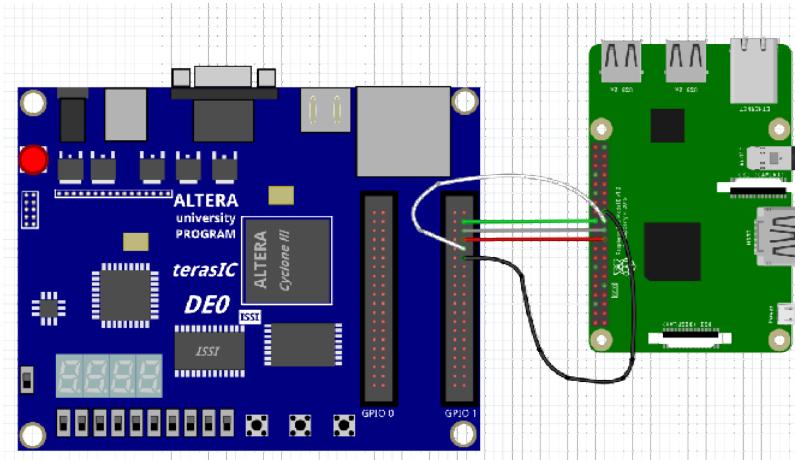


Figura 17: Schema di collegamento visivo tra le due schede

## Funzionamento della Comunicazione SPI

La comunicazione SPI (*Serial Peripheral Interface*) è un protocollo di comunicazione seriale ampiamente utilizzato nell'ambito dell'elettronica embedded. È utilizzato per collegare dispositivi digitali tra loro, consentendo loro di scambiare dati in modo sincrono e affidabile.

Il funzionamento della comunicazione SPI si basa su una connessione di tipo master-slave tra dispositivi. In questa configurazione, un dispositivo agisce da "master" e controlla il flusso dei dati, mentre uno o più dispositivi agiscono da "slave" e rispondono alle richieste del master. Il master è responsabile di generare il segnale di clock (SCK) che sincronizza la trasmissione e ricezione dei dati.

I segnali chiave utilizzati nella comunicazione SPI sono:

- **SCK (Serial Clock):** Questo segnale viene generato dal master e utilizzato per sincronizzare la trasmissione e ricezione dei dati tra i dispositivi. I dati vengono campionati sul fronte di salita o discesa del segnale di clock, a seconda della configurazione;
- **MOSI (Master Output Slave Input):** Questo è il segnale di uscita del master e di ingresso dello slave. Il master utilizza MOSI per inviare dati agli slave. Quando i dati vengono trasmessi, vengono spostati bit per bit lungo il MOSI sincronizzati dal segnale di clock;

- **MISO (Master Input Slave Output):** Questo è il segnale di ingresso del master e di uscita dello slave. Gli slave utilizzano MISO per inviare dati al master. Anche in questo caso, i dati vengono spostati bit per bit lungo il MISO sincronizzati dal segnale di clock;
- **SS/CS (Slave Select/Chip Select):** Questo segnale è utilizzato per selezionare uno specifico slave con cui il master vuole comunicare. Il master può avere più linee SS/CS per comunicare con diversi slave. Nel nostro caso indicato come CE0, ovvero *chip enable 0*.

Il funzionamento di una trasmissione SPI inizia quando il master seleziona uno specifico slave mediante il segnale SS/CS. Successivamente, il master inizia a inviare i dati lungo il MOSI, sincronizzati dal segnale di clock SCK. Gli slave campionano i dati in arrivo sul fronte di salita o discesa del segnale di clock e li trasmettono al master tramite il segnale MISO.

La comunicazione SPI può funzionare sia in modalità full-duplex, in cui il master e lo slave possono trasmettere contemporaneamente, che in modalità half-duplex, in cui la trasmissione avviene in entrambe le direzioni, ma non contemporaneamente.

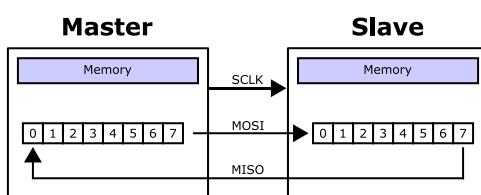


Figura 18: Esempio di comunicazione spi

Il master deve anche configurare polarità (CPOL) e fase (CPHA) del clock SCK. A seconda dei valori di CPOL e CPHA si distinguono 4 modalità di comunicazione SPI.

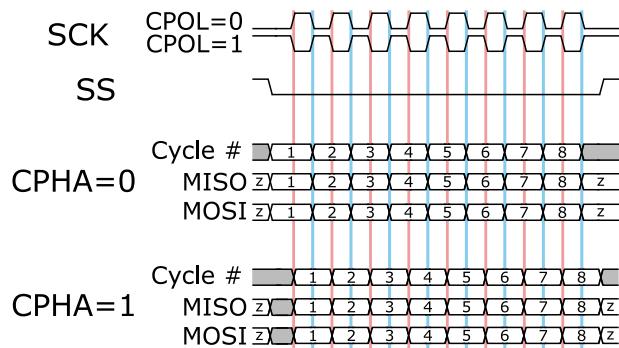


Figura 19: Diagramma temporale delle modalità di comunicazione SPI

- CPOL=0: Il valore base del clock SCK è '0' (lo stato active è 1 e lo stato idle è 0).
  - CPHA=0: acquisizione sul fronte di salita del clock, invio sul fronte di discesa.
  - CPHA=1: acquisizione sul fronte di discesa del clock, invio sul fronte di salita.
- CPOL=1: Il valore base del clock SCK è '1' (lo stato active è 0 e lo stato idle è 1).
  - CPHA=0: acquisizione sul fronte di discesa del clock, invio sul fronte di salita.
  - CPHA=1: acquisizione sul fronte di salita del clock, invio sul fronte di discesa.

La modalità implementata è: CPOL=0, CPHA=1.

# Visualizzazione del dato in rete

L'implementazione di una comunicazione affidabile e veloce è diventata cruciale in molte applicazioni moderne, spaziando dall'Internet delle Cose (IoT) al controllo industriale. Per questo contesto viene utilizzato il protocollo User Datagram Protocol (UDP) su Ethernet. Questo protocollo permette di trasmettere dati in modo efficiente attraverso reti locali senza l'onere di una connessione stabilita.

Per abbracciare questa metodologia è stato creato uno script personalizzato in linguaggio Python attingendo al suo potenziale nel mondo dell'automazione e delle reti. Questo script è stato progettato per consentire la visualizzazione fluida dei dati attraverso la comunicazione UDP su Ethernet.

Abbiamo sfruttato la flessibilità e la semplicità di Python per gestire la comunicazione UDP. È stata utilizzata la libreria `socket` di Python per creare un canale UDP e stabilendo una via di comunicazione tra il mittente e il destinatario. Successivamente, si è implementato il formato di trasmissione dei dati e il meccanismo di ricezione per garantire che i dati vengano correttamente incapsulati e decodificati all'altro capo.

Una caratteristica chiave dello script è la sua adattabilità. Il codice è stato scritto in modo che sia possibile specificare l'indirizzo IP e la porta del destinatario consentendo la configurazione flessibile della comunicazione tra i dispositivi all'interno della rete. Questa caratteristica permette di integrare lo script in una varietà di contesti e scenari: dalla visualizzazione dei dati a scopi di monitoraggio all'integrazione con applicazioni di controllo.

Va notato che l'utilizzo di questo protocollo è particolarmente adatto per reti locali, ma data la mancanza di conferme di ricezione potrebbe non essere adatto per scenari in cui l'affidabilità della comunicazione è critica. Inoltre, è importante sottolineare che lo script è stato creato e testato con successo su versioni di Python anteriori alla 3.10.

Di seguito viene riportato il codice python utilizzato per la lettura dei dati inviati da RaspberryPi:

```
1 import socket
2
3 def main():
4     # Creazione di un socket UDP
5     udp_socket = socket.socket(socket.AF_INET, socket.SOCK_DGRAM)
6
7     # Selezione di IP e numero di porta da ascoltare
8     ip_address = '0.0.0.0' # Ascolta su tutte le interfacce disponibili
9     port = 25000
10
11    # Associazione del socket all'indirizzo IP e alla porta
12    udp_socket.bind((ip_address, port))
13
14    # Feedback di avvio
15    print(f" Ascolto UDP avviato su {ip_address}:{port}")
16
17    # Main loop
18    while True:
19        # Ricezione di dati e IP del mittente
20        data, address = udp_socket.recvfrom(1024)
21
22        # Verifica su dati non vuoti prima di stamparli
23        if data and data != b'\x00' * len(data):
24            print(f" Ricevuti {len(data)} byte da {address[0]}:{address[1]}")
25            print(f" Dati ricevuti (RAW): {data.hex()}" ) # Stampa dei dati in formato esadecimale (hex)
```

```
26  
27 if __name__ == '__main__':  
28     main()  
29
```

Listing 4: Script `udp_listener.py` per l'ascolto della porta UDP utilizzata dal RaspberryPi

Di seguito viene mostrato l'output prodotto dallo script Python in ascolto al socket UDP.

Figura 20: `udp_listener.py` - Output dello script python

# Risultati sperimentali

## Studio comportamentale

La simulazione è stata effettuata utilizzando il software *Altera Modelsim* e, in particolare, il file `tb_spi.vhd` esposto nel listato 3. La figura 21 presenta la simulazione completa di tutti gli stati del sistema.

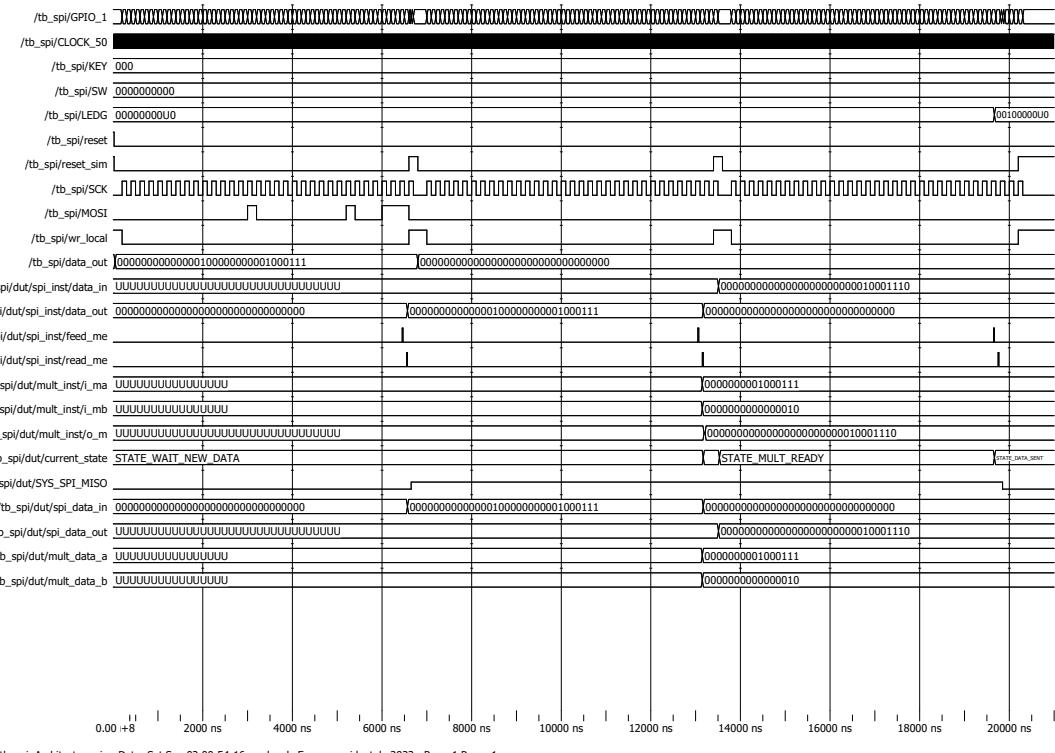


Figura 21: Simulazione completa ottenuta con Modelsim

In dettaglio, vengono esposti tutti i **signal** coinvolti nel processo di moltiplicazione e invio del dato. Infatti, si è simulato l'invio e la ricezione del dato come se fossero stati ottenuti mediante una comunicazione SPI.

Sempre in riferimento al listato 3, il periodo di clock dell'FPGA, denominato `clk_period`, e il periodo di clock di comunicazione SPI, denominato `sck_period`, sono due valori costanti che definiscono i tempi operativi di lavoro. Ovviamente, il periodo di comunicazione deve necessariamente essere maggiore del periodo di lavoro: per la simulazione si è scelto un periodo di clock di 20ns mentre il periodo di comunicazione è stato impostato a 200ns.

Con questi valori operativi, la simulazione mostrata in figura 21 riporta un tempo di circa 2ms da quando il dato viene inviato su SPI prima che ritorni il dato moltiplicato al Raspberry Pi.

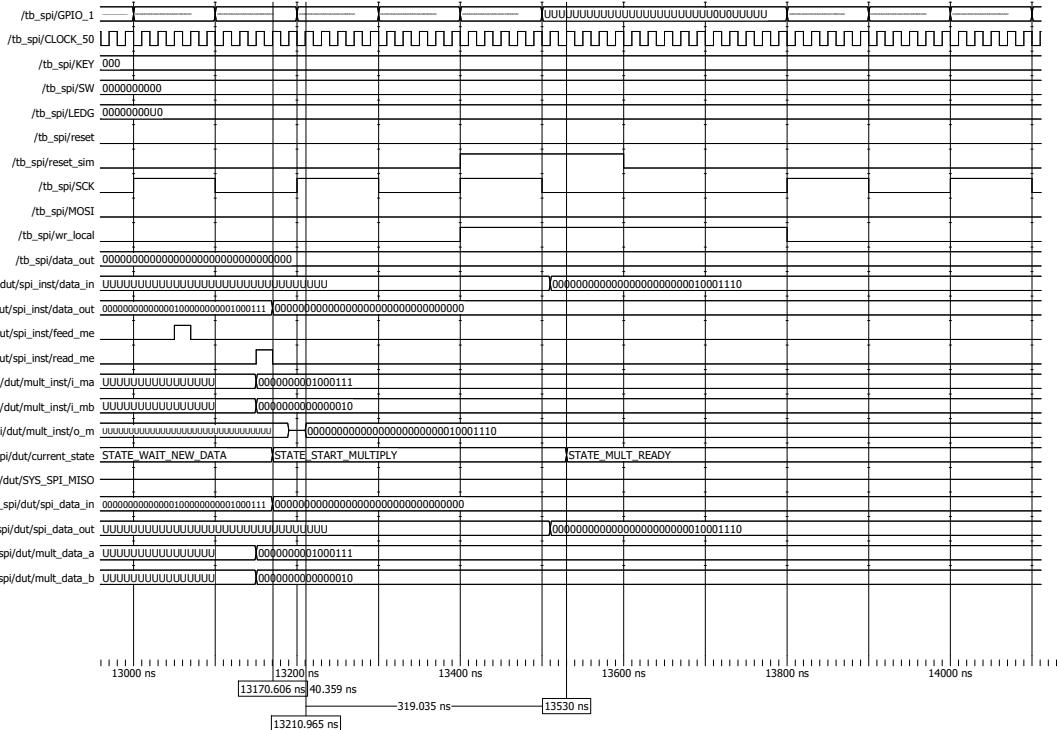
È importante notare che lo stato `STATE_WAIT_NEW_DATA` permane nonostante il dato sia già presente. Questo è dovuto al controllo che viene effettuato all'interno della macchina a stati. Con riferimento al listato 2, in particolare alla riga 108, il controllo per il passaggio di stato vuole che sia presente il flag `newdata` (collegato al signal `feed_me` dell'entity SPI) e che il signal `spi_data_in` sia diverso da 0. Tuttavia, poiché il signal `feed_me` si aggiorna sul fronte di discesa del clock di comunicazione `SCK` mentre il segnale `data_out` si aggiorna sul fronte di salita successivo del clock di comunicazione. Non avendo il processo `main_process` nella sua *sensitivity list* il signal `spi_data_in`, la macchina a stati si aggiorna al signal `feed_me` successivo. Questa può essere considerata un'imprecisione dell'architettura ma non ne compromette il funzionamento.

Alla conferma di ricezione del dato, esso viene scomposto nei due operandi e viene effettuato il passaggio di stato della macchina a stati nello stato **STATE\_START\_MULTIPLY**. Nella figura 21 queste operazioni sono visibili nei signal **mult\_data\_a** e **mult\_data\_b** che passano da una condizione **UNDEFINED** ad un valore determinato, in particolare ai numeri 2 e 71 presentati in notazione binaria.

Successivamente, al termine delle operazioni di moltiplicazione, lo stato passa al valore `STATE_MULT_READY` indice della presenza del dato correttamente moltiplicato e pronto per essere inviato su seriale SPI. È possibile visualizzare il risultato della moltiplicazione in prossimità dello stato corrispondente osservando il signal `o_m` oppure il signal `dut.spi_data.out`.

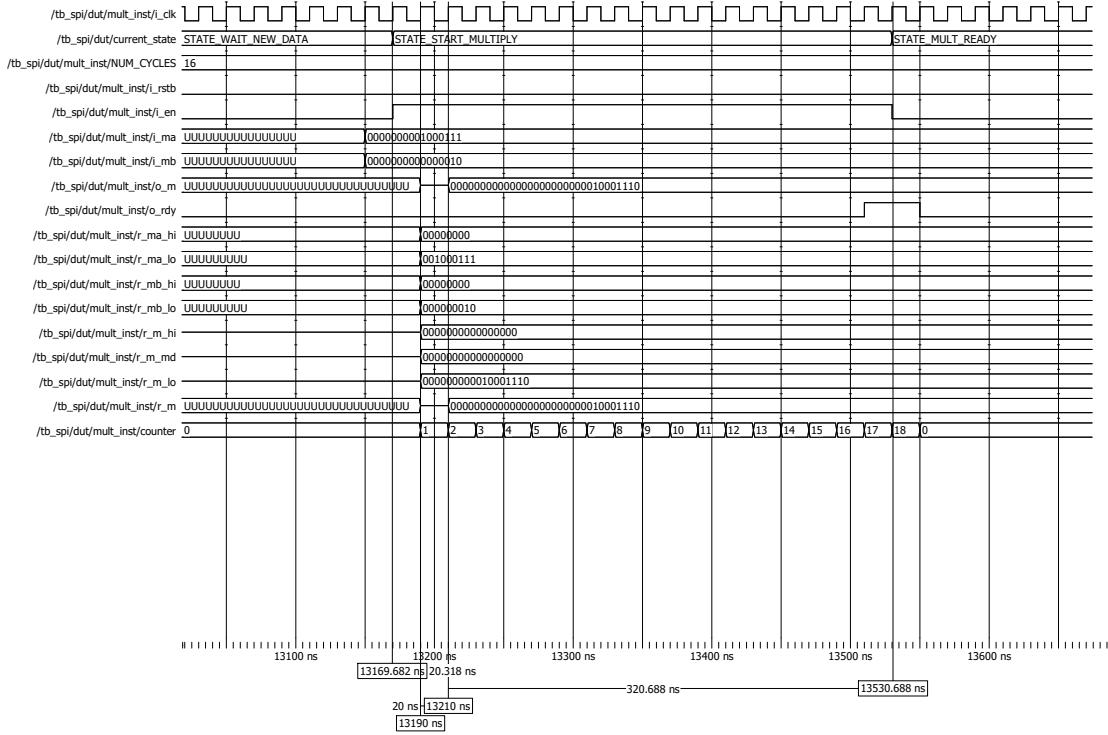
Infine, al termine dell'invio del dato il sistema passa allo stato **STATE\_DATA\_SENT** terminando così il suo ciclo operativo, in attesa di un nuovo segnale di reset per ripartire. In questo caso, non è stato possibile visualizzare in simulazione il corretto movimento del signal **MISO** per problemi di architettura e direzione dei port. Tuttavia, il risultato ottenuto è in linea con i risultati ottenuti per la ricezione del dato in ingresso quindi è possibile sostenere che il tempo impiegato dal sistema al passaggio di stato tra lo stato **STATE\_MULT\_READY** e lo stato **STATE\_DATA\_SENT** sia un tempo congruo per l'invio di un dato di 32bit su seriale SPI con periodo di clock SCK di 200ns.

## Analisi temporale



Entity:tb\_spi Architecture:sim Date: Sat Sep 02 00:58:04 ora legale Europa occidentale 2023 Row: 1 Page: 1

Figura 22: Focus della simulazione ottenuta con Modelsim sullo stato di avvio della moltiplicazione



Entity:tb\_spi Architecture:sim Date: Sat Sep 02 01:02:26 ora legale Europa occidentale 2023 Row: 1 Page: 1

Figura 23: Simulazione dell'entity mult\_16x16\_sign\_break ottenuta con Modelsim

## Analisi del funzionamento del sistema con SignalTap Logic Analyzer

In figura 24 è mostrato l'output dei segnali prodotti dall'FPGA utilizzando l'entity basic\_mult.

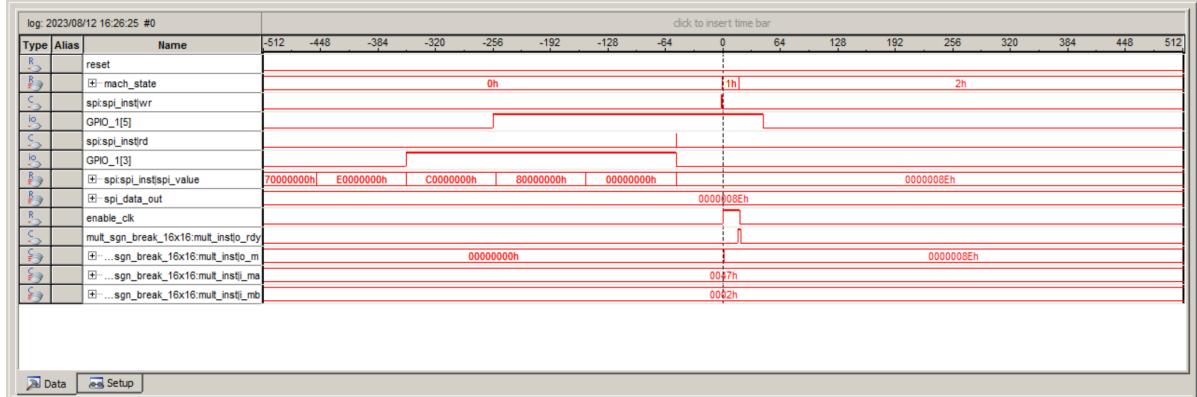


Figura 24: Simulazione dell'entity basic\_mult ottenuta con SignalTap Logic Analyzer

In particolare, il trigger per la visualizzazione è stato impostato sul fronte di salita del segnale enable\_clk.

È importante evidenziare anche il cambiamento degli stati del segnale `mach_state` e che il risultato ottenuto rispecchi perfettamente la simulazione ottenuta nella sezione *Studio comportamentale*.

In figura 25 viene riportato il dettaglio della fase di moltiplicazione proponendo i segnali prodotti dall'entity `mul_sgn_break_16x16`.

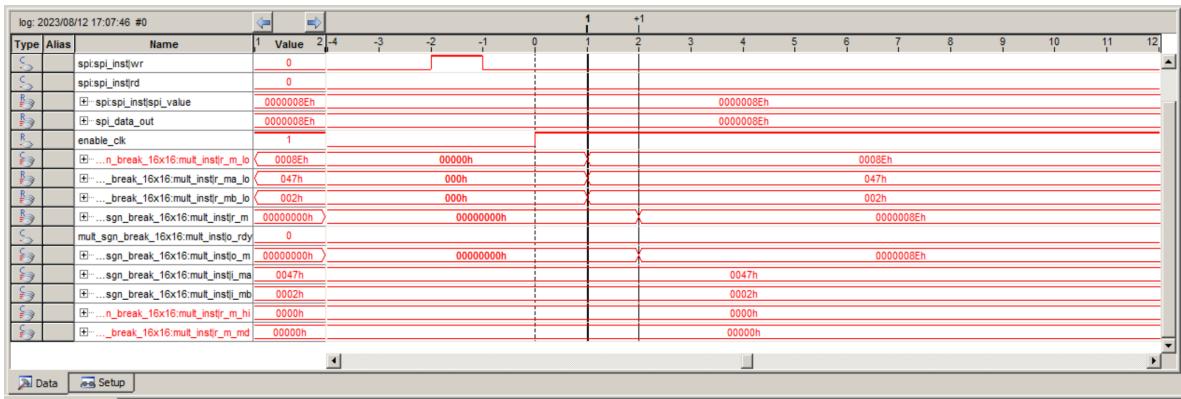


Figura 25: Simulazione dell’entity `mul_sgn_break_16x16` ottenuta con SignalTap Logic Analyzer

Anche in questo caso i risultati ottenuti sono in accordo con i risultati proposti dalla simulazione.

# Conclusioni

Visti i risultati ottenuti, il sistema rispetta le specifiche di progetto e riesce a produrre il risultato voluto. Sicuramente è un sistema imperfetto che necessita di qualche accorgimento per essere ottimizzato nel tempismo di produzione del dato. Ciò potrebbe essere ottenuto utilizzando meglio i l'analisi dei segnali in gioco, in particolare il `feed_me` dell'entity SPI come analizzato in precedenza.

È sicuramente possibile migliorare l'aspetto implementativo del sistema. Il listato 3 presenta alcuni problemi di astrazione che si riscontrano anche in figura 8. La divisione tra top level e livelli inferiori non è netta come dovrebbe essere da teoria.

Inoltre, lo script python prodotto per la visualizzazione del dato presenta alcune imperfezioni di lettura, ma non essendo l'obiettivo di questo progetto non è stato perfezionato.

# Elenco delle figure

1	FPGA Cyclone III . . . . .	3
2	Raspberry Pi Model 2B . . . . .	4
3	Suddivisione dell'operando A . . . . .	5
4	Schema a blocchi della moltiplicazione. . . . .	6
5	Architettura moltiplicatore implementato . . . . .	18
6	Finite state machine ottenuta con l'estensione <i>TerosHDL</i> . . . . .	20
7	Finite state machine ottenuta con <i>Quartus</i> . . . . .	20
8	Vista RTL integrale ottenuta con <i>Quartus</i> . . . . .	21
9	Prima parte della vista RTL ottenuta con <i>Quartus</i> . . . . .	21
10	Seconda parte della vista RTL ottenuta con <i>Quartus</i> . . . . .	22
11	Terza parte della vista RTL ottenuta con <i>Quartus</i> . . . . .	22
12	Quarta parte della vista RTL ottenuta con <i>Quartus</i> . . . . .	22
13	Vista del top level . . . . .	23
14	Architettura modulo spi utilizzato . . . . .	23
15	Vista RTL integrale ottenuta con <i>Quartus</i> dell'entity SPI . . . . .	25
16	Schema a blocchi simulink implementato . . . . .	26
17	Schema di collegamento visivo tra le due schede . . . . .	27
18	Esempio di comunicazione spi . . . . .	28
19	Diagramma temporale delle modalità di comunicazione SPI . . . . .	28
20	<b>udp_listener.py</b> - Output dello script python . . . . .	30
21	Simulazione completa ottenuta con Modelsim . . . . .	31
22	Focus della simulazione ottenuta con Modelsim sullo stato di avvio della moltiplicazione . . . . .	32
23	Simulazione dell'entity mult_16x16_sign_break ottenuta con Modelsim . . . . .	33
24	Simulazione dell'entity basic_mult ottenuta con SignalTap Logic Analyzer . . . . .	33
25	Simulazione dell'entity mul_sgn_break_16x16 ottenuta con SignalTap Logic Analyzer . . . . .	34

# Elenco delle tabelle

1	Esempio di moltiplicazione binaria . . . . .	6
2	Segnali del top level dell'architettura . . . . .	18
3	Segnali interni all'architettura <i>top_arch</i> di basic_mult . . . . .	19
4	Costanti utilizzate nell'architettura <i>top_arch</i> di basic_mult . . . . .	19
5	Possibili valori per la macchina a stati nell'architettura <i>top_arch</i> di basic_mult . . . . .	19
6	Gerneric values per l'architettura SPI implementata . . . . .	23
7	Descrizione degli input/output dell'architettura SPI implementata . . . . .	23
8	Segnali dell'architettura SPI implementata . . . . .	24
9	Collegamenti pin to pin delle due board . . . . .	27

# Listings

1	<b>mul16.vhd</b> - architettura del moltiplicatore a 16 bit . . . . .	7
2	Top level implementato per la nostra architettura . . . . .	10
3	Test bench dell'architettura <i>basic_mult</i> . . . . .	14
4	Script <code>udp_listener.py</code> per l'ascolto della porta UDP utilizzata dal RaspberryPi . . . . .	29