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

Progetto di Sistemi Elettronici per l'Internet of Things

Autori:
Luca Brescia
Matricola 706005

Loda Michele
Matricola 85967

Simone Pezzottini
Matricola 89521

Anno Accademico 2022–2023

Indice

Introduzione

0.1 Descrizione generale del progetto

0.1.1 Obiettivi della relazione

Capitolo 1

Architettura del sistema

1.1 Descrizione dell'FPGA Cyclone III

1.1.1 Specifiche della FPGA

L'FPGA Cyclone III EP3C16F484C6, 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.

Esso si contraddistingue per la sua affidabilità, flessibilità e prestazioni, ed è ampiamente utilizzato in una varietà di applicazioni, tra cui elaborazione di segnali, telecomunicazioni, automazione industriale e sistemi embedded.

Capitolo 2

Struttura del moltiplicatore 16x16 bit

2.1 Descrizione dell'algoritmo di moltiplicazione

Il modulo VHDL **mul16** 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 3.1).

```
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
               operandi (A[15:8])
34         signal r_ma_lo  : signed(8 downto 0);   -- Parte bassa del primo
               operandi con bit di segno (A[7:0])
35         signal r_mb_hi  : signed(7 downto 0);   -- Parte alta del secondo
               operandi (B[15:8])
36         signal r_mb_lo  : signed(8 downto 0);   -- Parte bassa del secondo
               operandi 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 operandi
66                 r_ma_lo <= signed('0' & i_ma(7 downto 0)); -- Assegnazione
                       della parte bassa del primo operandi con estensione del
                       bit di segno
67                 r_mb_hi <= signed(i_mb(15 downto 8));      -- Assegnazione
                       della parte alta del secondo operandi
68                 r_mb_lo <= signed('0' & i_mb(7 downto 0)); -- Assegnazione
                       della parte bassa del secondo operandi con estensione del
                        bit di segno
69                 r_m     <= r_m_hi & "0000000000000000" + resize(r_m_md & "
                       00000000", 32) + resize(r_m_lo, 32);  -- Calcolo del
                       risultato finale della moltiplicazione
70                 --! Aumento il numero di cicli effettuati per la
```

```
                    moltiplicazione, se li supero segnalo output ready
                counter <= counter + 1;
                if counter >= NUM_CYCLES then
                        o_rdy <= '1';
                end if;
                else
                counter <= 0;
                o_rdy <= '0';
                end if;
        end if;
        end process p_mult;
end rtl;
```

Listing 2.1: Processo principale del moltiplicatore

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.

2.1.1 Motivazioni della dimensione dei dati

Nel progetto del moltiplicatore a 16 bit, la scelta della dimensione dei dati riveste un ruolo cruciale nell'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.

La decisione di utilizzare dati a 16 bit è stata guidata da diverse considerazioni. Innanzitutto, 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 $2^{15} - 1$, coprendo un intervallo adeguato per molte applicazioni.

Inoltre, l'adozione di dati a 16 bit si allinea con le specifiche e le esigenze dell'applicazione. Ad esempio, nel contesto di un moltiplicatore, è spesso necessario eseguire operazioni su numeri relativamente piccoli, ma con una precisione sufficiente per evitare overflow o underflow.

Dal punto di vista delle prestazioni, l'utilizzo di dati a 16 bit permette di mantenere un equilibrio tra la precisione dei calcoli e l'efficienza dell'hardware. Dimensioni maggiori dei dati potrebbero richiedere risorse hardware aggiuntive, aumentando il consumo energetico e la latenza. D'altra parte, dimensioni più piccole potrebbero limitare la precisione dei risultati.

Infine, l'adozione di dati a 16 bit semplifica la progettazione e l'implementazione del moltiplicatore. 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.

In conclusione, la scelta di utilizzare dati a 16 bit nel moltiplicatore è stata guidata da considerazioni di rappresentatività, precisione, efficienza e praticità. Questa dimensione è adeguata per l'applicazione specifica, consentendo un equilibrio tra prestazioni e complessità.

Capitolo 3

Progettazione e implementazione in VHDL

3.1 Descrizione del linguaggio VHDL

Il VHDL (*VHSIC Hardware Description Language*) è un linguaggio di descrizione hardware ampiamente utilizzato per progettare e descrivere circuiti digitali complessi. Esso è diventato uno standard nell'industria dell'elettronica digitale.

Il funzionamento del VHDL si basa sulla descrizione dei circuiti digitali attraverso un insieme di dichiarazioni e processi. 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'industria dell'elettronica digitale.

3.2 Codice implementato

L'architettura `basic_mult` rappresenta l'aspetto centrale di un progetto di moltiplicatore a 16 bit. Essa costituisce un sistema in cui diverse entità lavorano sinergicamente per realizzare l'obiettivo complessivo del progetto.

L'architettura `basic_mult` gestisce l'interconnessione tra entità eterogenee, tra cui i componenti personalizzati e i moduli predefiniti fornitici durante il corso. Questa integrazione è vitale per la corretta comunicazione e cooperazione tra le componenti del sistema.

Un esempio cruciale di integrazione è il modulo `spi`, che abilita l'interfacciamento tra dispositivi attraverso una comunicazione seriale sincrona. Questo modulo viene utilizzato all'interno dell'architettura `basic_mult` per stabilire una comunicazione standardizzata con dispositivi esterni.

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 entità all'interno dell'architettura `basic_mult` richiede la sincronizzazione dei segnali di controllo, flussi di dati e temporizzazioni. Questi aspetti sono gestiti attraverso processi definiti all'interno dell'architettura, garantendo un funzionamento coordinato e armonioso del sistema.

La progettazione di questa architettura richiede la conoscenza approfondita delle specifiche delle entità coinvolte e delle modalità di interazione tra di esse. L'interconnessione tra `spi`, `blink heartbeat` e `basic_mult` riflette la capacità di progettare e implementare un sistema complesso, combinando abilmente le varie entità in un'architettura funzionale.

```
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     SW       : in std_logic_vector(9 downto 0);       --! Dati in ingresso da
       interruttore
21     KEY      : in std_logic_vector(2 downto 0);       --! Segnali di controllo
        push buttons
22     LEDG     : out std_logic_vector(9 downto 0);      --! Segnali di output
       per i LED
23     GPIO_1   : inout std_logic_vector(31 downto 0)    --! Segnali di input/
       output GPIO
24   );
25 end basic_mult;
26
27 architecture top_arch of basic_mult is
28   --! Definizioni costanti
29   constant DATA_W         : integer := 32;   --! Dimensione del dato
30   constant Nbit           : integer := 5;    --! Numero di bit per contenere
       il dato
```

```
31
32    --! Stati per la macchina a stati
33    constant STATE_WAIT_NEW_DATA  : std_logic_vector(2 downto 0) := "000"; --!
          Stato di attesa del nuovo dato
34    constant STATE_START_MULTIPLY : std_logic_vector(2 downto 0) := "001"; --!
          Stato di avvio della moltiplicazione e attesa del risultato
35    constant STATE_MULT_READY     : std_logic_vector(2 downto 0) := "010"; --!
          Stato di risultato pronto e attesa invio su spi
36    constant STATE_DATA_SENT      : std_logic_vector(2 downto 0) := "100"; --!
          Stato di fine delle operazioni e attesa segnale di reset
37
38    --! Definizioni signals
39    signal pb0_synchronizer : std_logic_vector(2 downto 0);                --!
         Sincronizzatore 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 SPI
42    signal SYS_SPI_MISO     : std_logic                          := '0';   --!
         Pin di input dati SPI
43    signal spi_data_in      : std_logic_vector(DATA_W - 1 downto 0);       --!
         Dati in ingresso SPI
44    signal spi_data_out     : std_logic_vector(DATA_W - 1 downto 0);       --!
         Dati in uscita SPI
45    signal mult_data_a      : std_logic_vector(DATA_W/2 - 1 downto 0);     --!
         Copia del dato in ingresso
46    signal mult_data_b      : std_logic_vector(DATA_W/2 - 1 downto 0);     --!
         Copia del dato in ingresso
47    signal result           : std_logic_vector(DATA_W - 1 downto 0);       --!
         Risultato della moltiplicazione
48    signal reset            : std_logic                          := '0';   --!
         Flag di reset
49    signal enable_clk       : std_logic                          := '0';   --!
         Flag di abilitazione clock per l'entity moltiplicatore
50    signal newdata          : std_logic                          := '0';   --!
         Flag di presenza nuovo dato da inviare
51    signal multready        : std_logic                          := '0';   --!
         Flag di segnalazione dato moltiplicato corretto
52    signal datasent         : std_logic                          := '0';   --!
         Flag di invio corretto del dato
53    signal mach_state       : std_logic_vector(2 downto 0) := (others => '0');
          --! Macchina a stati
54
55  begin
56    --! Assegnazione dei segnali SPI ai pin fisici GPIO
57    SYS_SPI_SCK  <= GPIO_1(7);  --! Segnale di clock spi
58    SYS_SPI_MOSI <= GPIO_1(5);  --! Segnale di input per FPGA (Slave)
59    GPIO_1(3) <= SYS_SPI_MISO;  --! Segnale di output per FPGA (Slave)
60
61    --! Gestione del lampeggio del LED
62    blink_hb : entity work.blink_heartbeat port map(
63      CLK => CLOCK_50,
64      LED => LEDG(0)
65      );
66
67    --! Istanza del modulo SPI
68    spi_inst : entity work.spi
69      generic
70      map (
```

```
71      DATA_W => 32,
72      Nbit   => 5
73      )
74      port
75      map (
76      CLK      => CLOCK_50,
77      reset    => reset,
78      DATA_IN  => spi_data_out,
79      DATA_OUT => spi_data_in,
80      RD       => datasent,
81      WR       => newdata,
82      SCK      => SYS_SPI_SCK,
83      MOSI     => SYS_SPI_MOSI,
84      MISO     => SYS_SPI_MISO
85      );
86
87    --! stanza del moltiplicatore 16x16
88    mult_inst : entity work.mult_sgn_break_16x16
89      port
90      map (
91      i_clk  => CLOCK_50,
92      i_rstb => reset,
93      i_en   => enable_clk,
94      i_ma   => mult_data_a,
95      i_mb   => mult_data_b,
96      o_m    => result,
97      o_rdy  => multready
98      );
99
100   --! Processo principale per la gestione della macchina a stati per la
        realizzazione del moltiplicatore senza segno spi
101   --! Attende un dato in ingresso su spi formattato come segue: (
        MOLTIPLICATORE << 16 | MOLTILPICANDO) dove moltiplicando e
        moltiplicatore sono due interi senza sefno a 16 bit
102   --!
103   main_process : process (CLOCK_50, reset)
104   begin
105     --! Trigger delle operazioni sul fronte positivo del clock FPGA
106     if rising_edge(CLOCK_50) then
107
108       --! Gestione del segnale di reset
109       if reset = '1' then
110         mach_state <= STATE_WAIT_NEW_DATA;     --! Reset macchina a stati
111         enable_clk <= '0';                     --! Reset clk dell'istanza
              moltiplicatore
112         LEDG(8 downto 2) <= (others => '0');  --! Reset dei led presenti su
              FPGA
113       end if;
114
115       --! Gestione della macchina a stati
116       case mach_state is
117
118         --! Attesa del nuovo dato
119         when STATE_WAIT_NEW_DATA =>
120           --! Controllo presenza flag di nuovo dato e che il dato sia
                diverso da 0
121           if newdata = '1' and to_integer(unsigned(spi_data_in)) /= 0  then
122             --! Divisione del dato di ingresso a 32bit come dato_a e dato_b
123             mult_data_a <= spi_data_in(15 downto 0);  --! Moltiplicando
```

```
                parte bassa (0 - 15) del dato in ingresso
124             mult_data_b <= spi_data_in(31 downto 16); --! Moltiplicatore
                    parte alta (16- 31) del dato in ingresso
125             mach_state <= STATE_START_MULTIPLY;       --! Avanzamento di
                    stato della macchina a stati
126           end if;
127
128         --! Avvio moltiplicazione e attesa completamento
129         when STATE_START_MULTIPLY =>
130           enable_clk <= '1';  --! Abilitazione del segnale di clock per la
                entity mult_sgn_break_16x16
131
132           --! Attesa flag moltiplicazione terminata
133           if multready = '1' then
134             spi_data_out <= result;           --! Copia del risultato sul
                    signal che inviera' su spi il valore
135             mach_state <= STATE_MULT_READY;   --! Avanzamento di stato della
                    macchina a stati
136           end if;
137
138         --! Moltiplicazione completata e attesa di fine invio del dato su
              spi
139         when STATE_MULT_READY =>
140           enable_clk <= '0'; --! Arresto del segnale di clock per la entity
                mult_sgn_break_16x16
141
142           --! Attesa flag dato inviato
143           if datasent = '1' then
144             mach_state <= STATE_DATA_SENT;    --! Avanzamento di stato della
                    macchina a stati
145           end if;
146
147         --! Termine delle operazioni
148         when others =>
149         LEDG(7) <= '1'; --! Segnalazione tramite accensione del led7 delle
              operazioni concluse
150
151         --! Se il flag di dato inviato e' attivo
152         if datasent = '1' then
153           LEDG(8) <= '1'; --! Segnalazione tramite accensione del led8 del
                corretto invio del dato
154         end if;
155       end case;
156
157     end if;
158   end process;
159
160   --! Gestione del segnale di reset
161   reset_handle : process (CLOCK_50, KEY)
162   begin
163     --! Trigger delle operazioni sul fronte positivo del clock FPGA
164     if (rising_edge(CLOCK_50)) then
165       --! Sincronizzazione del pushbutton0 e generazione del segnale di
            reset
166       pb0_synchronizer(2 downto 1) <= pb0_synchronizer(1 downto 0);
167       pb0_synchronizer(0)          <= KEY(0);
168
169       --! Fronte positivo indica che il pushbutton0 e' stato rilasciato
170       if pb0_synchronizer(2 downto 1) = "01" then
```

```
171         LEDG(1) <= '1';    --! Segnalazione tramite accensione del led1 l'
                attivazione del segnale di reset
172         reset   <= '1';    --! Attivazione segnale di reset
173       --! Se e' stato attivato al clock precedente il segnale di reset
174       elsif (reset = '1') then
175         LEDG(1) <= '0';    --! Segnalazione tramite spegnimento del led1 la
                disattivazione del segnale di reset
176         reset   <= '0';    --! Disattivazione segnale di reset
177       end if;
178     end if;
179   end process;
180
181 end top_arch;
```

Listing 3.1: Processo principale del moltiplicatore

3.2.1 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 nella moltiplicazione di numeri interi a 16 bit. La moltiplicazione in sé è un'operazione complessa che richiede diverse fasi di calcolo, sincronizzazione e gestione dei segnali di controllo.

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 moltiplicazione, 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 da parte dei progettisti e degli sviluppatori.

Ogni stato nella macchina a stati corrisponde a un'operazione o a un insieme di operazioni ben definite. Ad esempio, uno stato potrebbe essere responsabile della lettura dei dati in ingresso, un altro potrebbe eseguire le operazioni di moltiplicazione effettive, mentre uno successivo potrebbe gestire la trasmissione dei risultati e la segnalazione della fine dell'operazione. Questa suddivisione in stati semplifica la verifica, il debug e la manutenzione del codice, consentendo di individuare e risolvere eventuali problemi in modo più efficiente.

Inoltre, l'uso della macchina a stati contribuisce a migliorare la gestione del flusso di controllo e dei segnali di timing all'interno del circuito. 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 accurati.

In sintesi, l'uso della macchina a stati rappresenta un approccio organizzato e strutturato per gestire il processo di moltiplicazione di numeri interi a 16 bit. Fornisce una suddivisione modulare delle operazioni, semplifica la gestione del flusso di controllo e dei segnali di timing, e migliora la comprensione e la manutenibilità del codice complessivo.

Capitolo 4

Simulazione e verifica

4.1 Tecniche di simulazione utilizzate

4.1.1 Testbench per la verifica del moltiplicatore

Capitolo 5

Sintesi e implementazione sulla FPGA

5.1 Fasi di sintesi e implementazione

5.1.1 Impostazioni di sintesi e implementazione

Capitolo 6

Programmazione del Raspberry Pi tramite MATLAB Simulink

6.1 Ruolo di MATLAB Simulink nel progetto

L'integrazione di dispositivi hardware, come il Raspberry Pi, 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 del tool "Hardware for RaspberryPi" di MATLAB Simulink.

"Hardware for RaspberryPi" è un potente strumento che permette agli sviluppatori di creare applicazioni personalizzate per il Raspberry Pi in modo intuitivo e 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 sul Raspberry Pi.

Questo tool offre una vasta libreria di blocchi funzionali che coprono una varietà di funzioni, dal controllo di periferiche hardware come i GPIO, alla comunicazione con sensori, attuatori e dispositivi esterni. Ciò consente agli sviluppatori di concentrarsi sulla logica dell'applicazione senza dover affrontare dettagli di basso livello della programmazione del Raspberry Pi.

Un aspetto particolarmente vantaggioso dell'utilizzo di "Hardware for RaspberryPi" è la sua capacità di generare automaticamente il codice C++ ottimizzato per il Raspberry Pi. Ciò significa che dopo aver creato il modello in Simulink, è possibile generare il codice e caricarlo direttamente sul Raspberry Pi, 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 il tool si installa correttamente, il tool non compila il blocco simulink creato.

In questo progetto, si è utilizzato il tool "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 l'FPGA e il Raspberry Pi, permettendo lo scambio di dati e il controllo dei segnali nel contesto del progetto in corso.

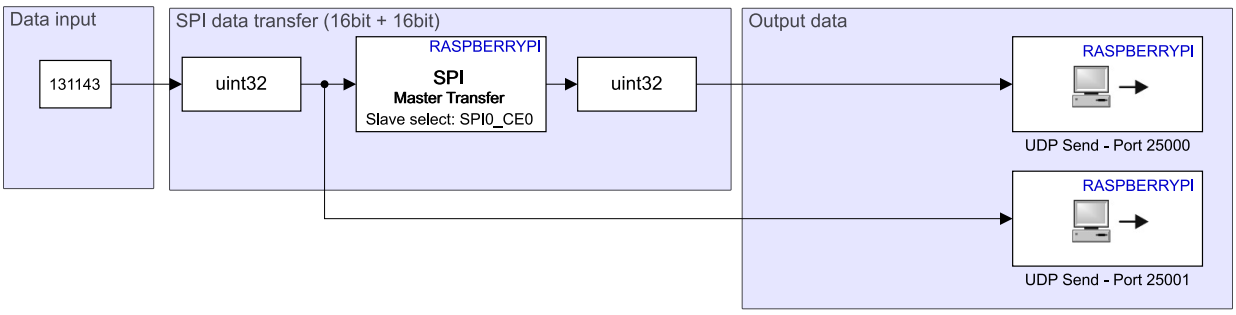

Figura 6.1: Schema a blocchi simulink implementato

6.2 Configurazione e comunicazione con il Raspberry Pi

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 i collegamenti da effettuare sulle due board:

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

Tabella 6.1: Collegamenti pin to pin delle due board

6.2.1 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.

Infine, la comunicazione SPI offre un meccanismo efficiente e affidabile per lo scambio di dati tra dispositivi digitali. La sua semplicità e la sua capacità di supportare sia applicazioni a corta distanza che a lunga distanza, ne fanno una scelta ideale per molte applicazioni nell'ambito dell'elettronica e dell'informatica embedded.

Capitolo 7

Risultati sperimentali

7.1 Test eseguiti per valutare le prestazioni del moltiplicatore

7.1.1 Risultati ottenuti

Capitolo 8

Conclusioni

8.1 Principali conclusioni del lavoro svolto

8.1.1 Sviluppi futuri e miglioramenti