CORSO DI PROGETTAZIONE DI SISTEMI DIGITALI

Implementazione di hardware su FPGA Altera dedicato al controllo di servomotori in una struttura esapode

Progetto a cura di:
Alessandro Paghi
Lorenzo De Marinis

Indice generale

Capitolo 1: Premessa

Il presente lavoro riguarda la progettazione e l'implementazione di hardware dedicato al controllo di servomotori presenti su una struttura esapode.

L'obiettivo del progetto risiede nell'ottimizzazione del sistema elettronico di bordo precedentemente presente sul robot esapode Augusto.
A tale scopo viene descritto dell'hardware dedicato al controllo dei 18 servomotori montati sulla struttura, implementato su FPGA Altera ed interfacciato con un'unità di calcolo Microchip PIC18.

La struttura meccanica non subisce mutamenti, in quanto progettata in modo da poter rendere il dispositivo in grado di svolgere una vasta gamma di movimenti.

Capitolo 2: Componentistica utilizzata

Si descrivono nel seguito i componenti utilizzati per il raggiungimento dell'obiettivo:

- Microchip PIC18LF4550;
- TerasIC DE10-Lite Board;
- Micro servo Hitec HS-53;
- Esapode Augusto.

2.1 Microchip PIC18LF4550

Il cuore del progetto è il controllore Microchip modello PIC18LF4550.

Il processore, che presenta una frequenza operativa massima pari a 48 MHz (12 MIPS), viene fornito di una memoria di programmazione flash pari a 32KB, una memoria RAM di 2048B e una memoria EEPROM di 256B.

Presenta 35 porte I/O, 13 convertitori A/D a 10 bit e supporta protocolli PWM, SPP, SPI, I2C ed EAUSART.

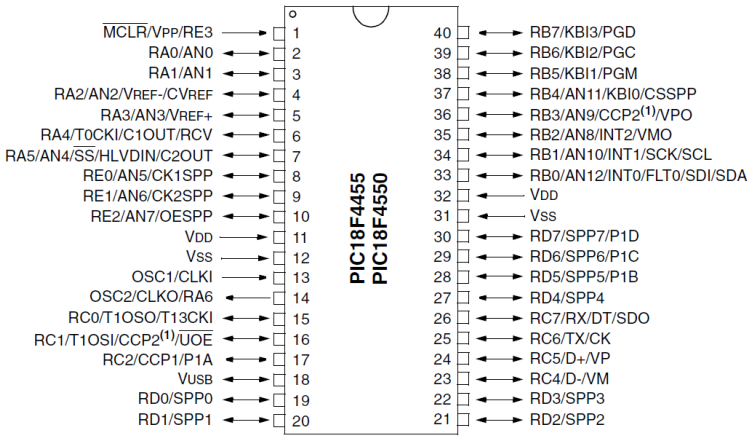

Figura: Socket PIC18F4550 40 pin, foro passante

La famiglia PIC18 fornisce alte performance ad un prezzo economico e garantisce la durata nel tempo sia del dispositivo che del programma memorizzato al suo interno. PIC18LF4550 è dotato anche di un'ampia gamma di features che riducono significativamente il consumo di potenza durante lo svolgimento delle operazioni.

Inoltre utilizza lo standard di comunicazione USB 2.0 sia in modalità low-speed che in modalità full-speed e dispone di diverse opzioni di settaggio dell'oscillatore, fornendo all'utente un ampio range di scelte durante lo sviluppo delle proprie applicazioni.

In particolar modo, il modello PIC18LF4550 differisce dal modello PIC18F4550 per la tensione di alimentazione applicabile.
Pagando in termini di frequenza di elaborazione, è possibile ridurre la tensione di alimentazione dai classici 5V nominali.
Per una V_{CC} pari a 5V si dispone di una frequenza operativa pari a 48 MHz mentre per una V_{CC} pari a 3.3V si dispone di una frequenza operativa pari a 16 MHz.

L'utilizzo di una logica 0-3.3 V è necessario per il colloquio con TerasIC DE10-Lite Board che, in assenza di traslatori di livello, non è abilitata a funzionare con logica 0-5 V.

2.2 TerasIC DE10-Lite Board

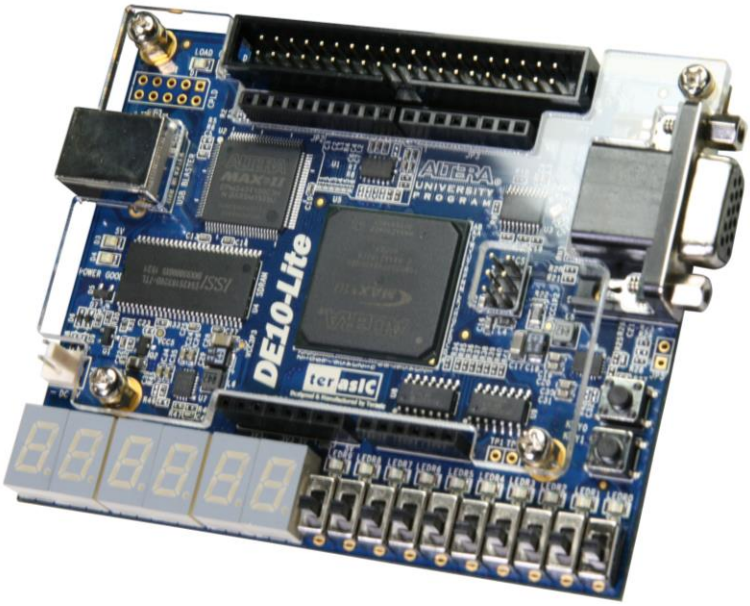

Figura: TerasIC DE10-Lite Board

La DE10-Lite presenta una piattaforma hardware estremamente robusta costruita attorno ad Altera MAX 10 FPGA.

FPGA MAX 10 è equipaggiato in modo da fornire il miglior rapporto qualità prezzo nello sviluppo di data path application e indutry-leading programmable solutions al fine di ottenere estrema flessibilità di progetto.

Con FPGA MAX 10 è possibile ottenere applicazioni a basso costo e consumo con elevate performance.

La DE10-Lite è la soluzione perfetta per progettare applicazioni di valutazione e per il prototyping di tutte le potenzialità del FPGA Altera MAX 10.

Il dispositivo FPGA contiene:

- MAX 10 10M50DAF484C7G Device;
- Due ADC integrati, ogni ADC supporta 1 input analogico dedicato ed 8 dual function pins;
- 50K elementi logici programmabili:
- 1,638 Kbits M9K memory;
- 5,888 Kbits user flash memory;

- 144 18 × 18 moltiplicatori;
- 4 PLLs.

Programmazione e configurazione:

- On-Board USB Blaster (Normal type B USB connector).

Memory Device:

- 64MB SDRAM, x16 bits data bus.

Connettori:

- 2x20 GPIO Header;
- Arduino Uno R3 Connector, including six ADC channels.

Display:

- 4-bit resistor-network DAC for VGA.

Interruttori, bottoni e LED:

- 10 LEDs;
- 10 Slide Switches;
- 2 Push Buttons with Debounced;
- Six 7-Segments.

Power:

- 5V DC input da USB o connettore di potenza esterno.

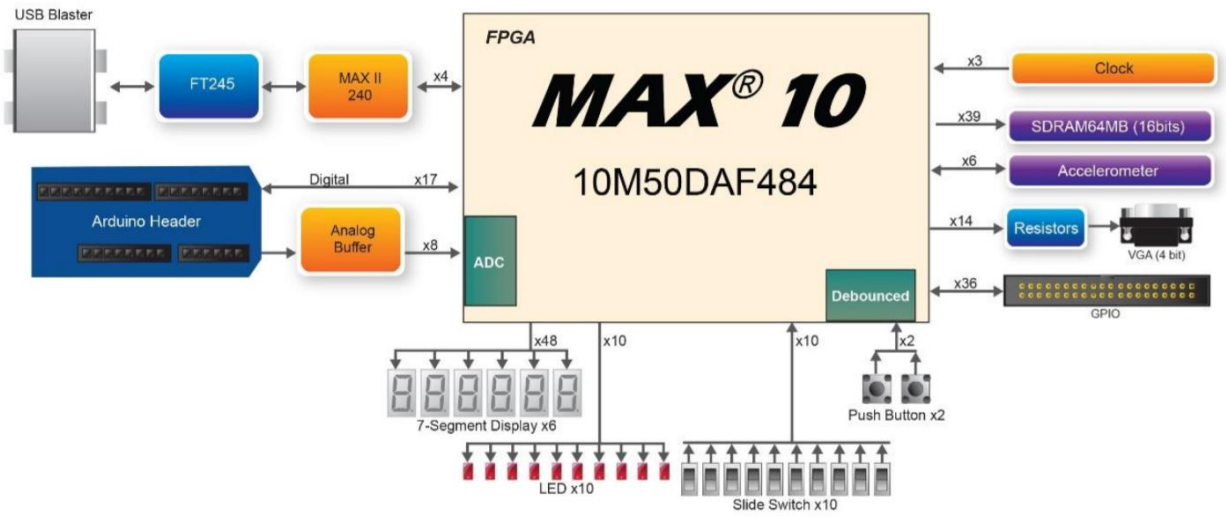

Figura: Diagramma a blocchi della board

2.3 Micro servo Hitec HS-53

Il servomotore si presenta come un piccolo contenitore di materiale plastico da cui fuoriesce un perno in grado di ruotare di un angolo compreso fra 0° e 180°, la cui posizione rimane stabile.

Per ottenere la rotazione del perno è utilizzato un motore in DC e un meccanismo di demoltiplica che consente di aumentare la coppia in fase di rotazione.

La rotazione del motore è effettuata tramite un circuito di controllo interno in grado di rilevare l'angolo di rotazione raggiunto dal perno tramite un potenziometro resistivo e di bloccare, quindi, il motore sul punto desiderato.

Il modello utilizzato viene alimentato con batteria a 6 V.

Figura: Modello CAD del servo motore Hitec HS-53

Caratteristiche tecniche:

Alimentazione	4.8 – 6.0 V
Coppia a 6 V	1.5 Kg/cm
Velocità a 6 V	0.13 sec/60 deg
Ingranaggi	Nylon
Dimensioni	22.8 x 11.6 x 22.6 mm
Peso	8 gr

2.4 Esapode Augusto

La struttura meccanica si presenta come un oggetto estremamente robusto, esteticamente accattivante e di facile montaggio.

Figura: Modello CAD dell'assieme, vista frontale

Figura: Modello CAD dell'assieme, vista laterale dx

Figura: Modello CAD dell’assieme, vista superiore

Figura: Modello CAD dell’assieme, vista in prospettiva

Capitolo 3: Specifiche di risoluzione

Il compito dell'hardware dedicato progettato è la generazione simultanea dei 18 segnali di controllo che i servomotori utilizzano per posizionarsi correttamente.

Ogni servo motore viene pilotato con il proprio segnale di controllo PWM.

La specifica di risoluzione che si intende ottenere è inferiore ad 1°.

Il periodo del segnale di controllo è fissato a $T_{PWM} = 20$ ms; di questo periodo, il tempo che il segnale trascorre al livello alto è compreso tra 500 us, in riferimento alla posizione corrispondete agli 0°, e 2300 us, in riferimento alla posizione corrispondente ai 180°.

In realtà ci possono essere discostamenti da questi valori nominali: dalle misure effettuate sono stati ricavati gli esatti DC (Duty Cicle) per cui ogni motore si posiziona a 0° e ad 180°, i quali risultano essere diversi dai valori precedentemente considerati.
L'utilizzo di tale approssimazione definisce comunque un buon modello per la schematizzazione di un generico servo motore e faremo dunque riferimento ai valori sopra riportati.

Consideriamo legame di linearità tra tempo che il segnale trascorre a livello alto e angolo raggiunto dal servo motore.

Il range di mantenimento a livello alto è dato da 2300 – 500 = 1800 us.

$$t_{1^\circ}^{high\_time} = \frac{1800}{180}\ us = 10\ us$$

Utilizzando un periodo di clock pari a:

$$T_{clk} = \frac{t_{1^\circ}^{high\_time}}{10} = 1\ us$$

è possibile ottenere una risoluzione di 0.1°.

Capitolo 4: Partizione dei processi di controllo

Il microcontrollore Microchip PIC18LF4550 ha il compito di risolvere le equazioni relative alla cinematica ed allo studio del moto, ed elaborare gli angoli che i giunti devono raggiungere.
L'hardware implementato su FPGA Altera risulta essere invece un Motor Driver: riceve dall'unità di calcolo le informazioni riguardanti gli angoli dei giunti e fornisce ai motori i riferimenti desiderati mantenendoli fino a nuovo comando.
I due componenti comunicano attraverso una linea seriale condivisa.

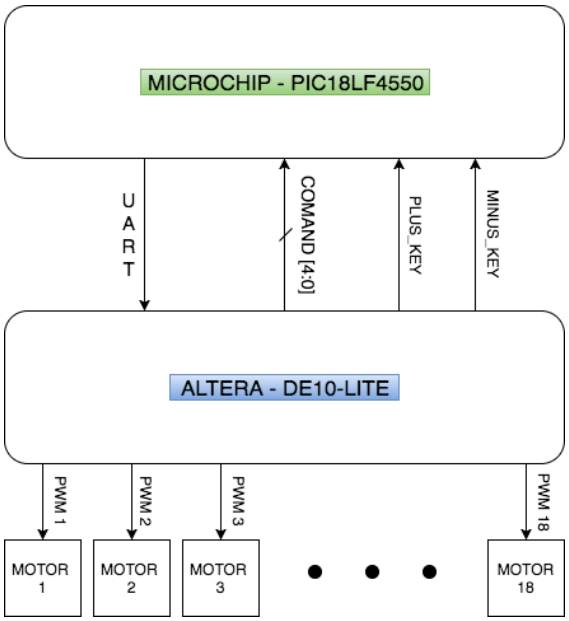

Figura: Schema a blocchi dell'architettura proposta

Una volta che i dati relativi agli angoli che i motori devo raggiungere nosono stati elaborati dal micro controllore, essi vengono suddivisi in pacchetti da 8 bit.

La trasmissione seriale comincia con l'inoltro di un byte di start in cui tutti gli 8 bit sono al valore 1; in questa maniera l'hardware dedicato implementato su FPGA Altera comprende che siamo in presenza dell'inizio di una ricezione; successivamente si trasmettono i 18 byte relativi alle posizioni dei 18 motori, ordinati per ID motore crescente.

Per ogni posizionamento del robot esapode sono necessari 19 pacchetti.

Capitolo 5: Architettura implementata

Si descrivono nel seguito i blocchi progettati e le tecniche utilizzate per il raggiungimento dell'obiettivo.

5.1 Schema a blocchi generale e regole utilizzate

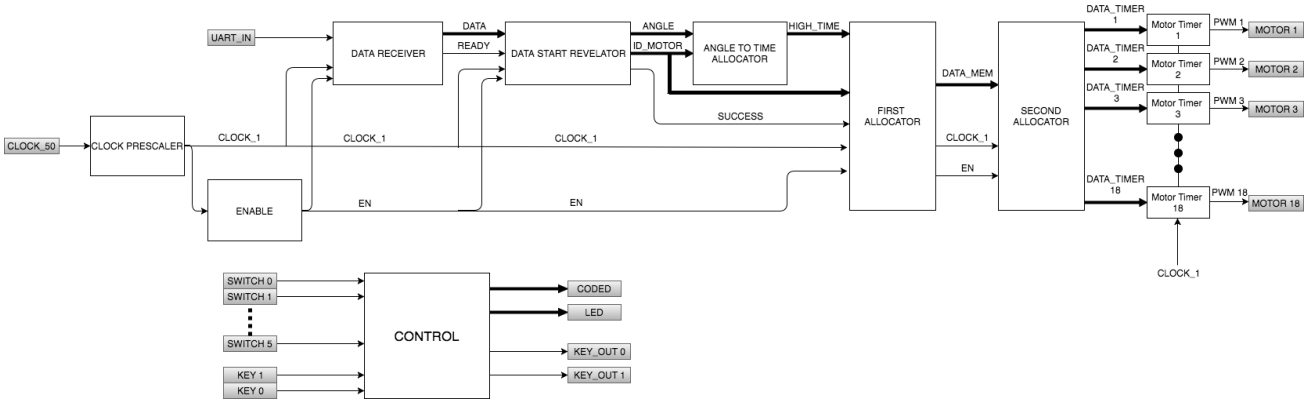

Figura: Schema a blocchi generale dell'architettura

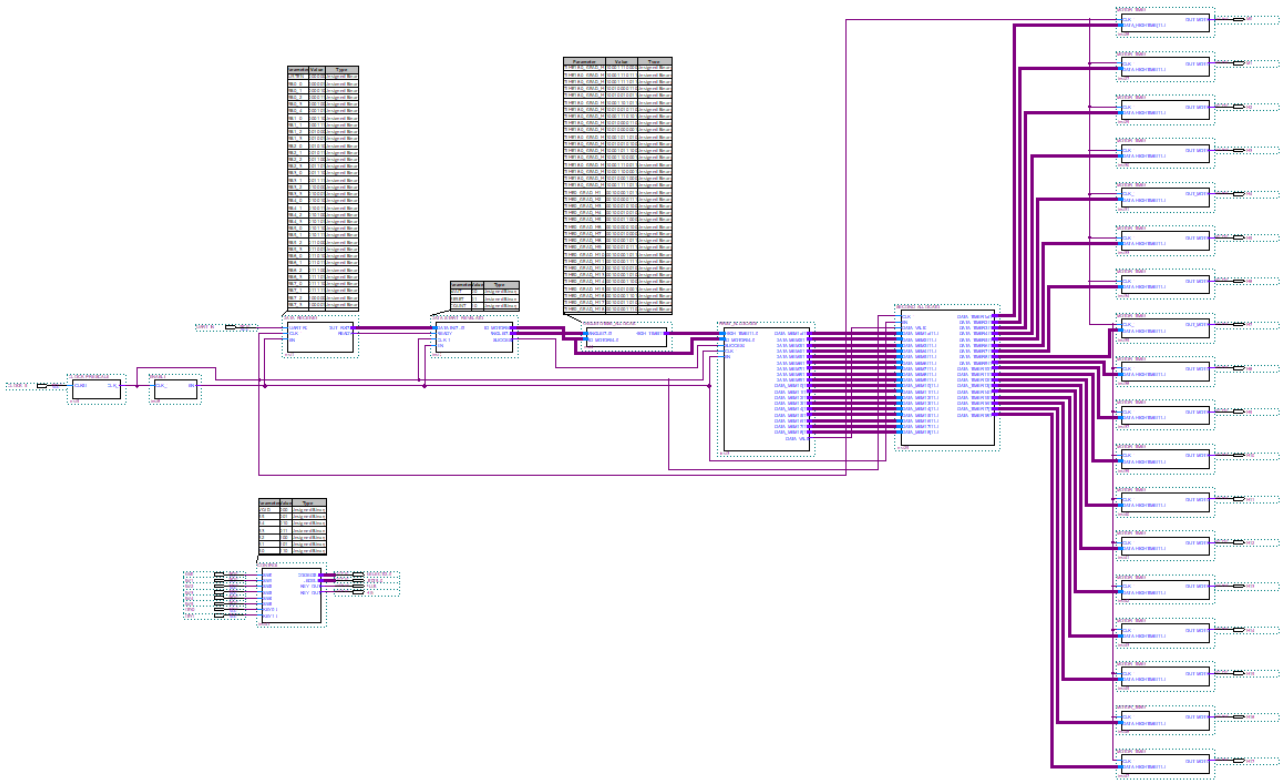

Figura: Schema a blocchi dell'architettura implementata su Quartus Prime 17

Lo schema a blocchi comprende l'utilizzo dei seguenti componenti:

- Blocco Clock Prescaler;
- Blocco Enable;
- Blocco Data Receiver;
- Blocco Data Start Revelator;
- Blocco Angle To Time Allocator;
- Blocco First Allocator;
- Blocco Second Allocator;
- 18 x Blocco Motor Timer;
- Blocco Control;

L'intero progetto è realizzato nel pieno rispetto delle regole del progetto sincrono statico: tutti i blocchi costruiti sono clockati dallo stesso fronte dello stesso segnale di clock.

Dal capitolo *Specifiche di risoluzione* è risultato che il periodo di clock necessario per poter generare segnali di controllo PWM in maniera corretta è di $T_{clk} = 1$ us, che corrisponde ad una frequenza di clock pari a $f_{clk} = 1$ MHz.

L'unità di elaborazione Microchip PIC18LF4550 colloquia con FPGA Altera attraverso una linea di comunicazione seriale UART impostata per funzionare con un Baud Rate

$$B_{RATE\ nominale} = 9600 \text{ Hz}.$$

Il Baud Rate, ovvero la frequenza di simbolo, corrisponde, nel caso di comunicazione binaria, anche alla frequenza di bit.

In realtà, il micro controllore che si occupa dalla trasmissione, utilizza una frequenza di trasmissione leggermente diversa a causa di meccanismi di approssimazione interni alla sua struttura.

Il Baud Rate utilizzato risulta essere $B_{RATE} = 9615$ Hz, che discosta dal valore nominale di progetto di 5 Hz.

$$Error = \frac{B_{RATE} - B_{RATE\ nominale}}{B_{RATE\ nominale}} = 0.16\ \%$$

Il clock presente in maniera nativa sulla TerasIC DE10-Lite Board ha una frequenza di

$$f_{CLK50MHz} = 50 \text{ MHz}.$$

È necessario un primo blocco che converta la frequenza di clock $f_{CLK50MHz}$ in f_{clk} che prende il nome di Blocco Clock Prescaler.

Nel rispetto delle regole del progetto sincrono statico è necessario che tutto il sistema progettato lavori con lo stesso fronte dello stesso segnale di clock.

I blocchi presenti in catena di ricezione operano con una frequenza di clock pari a $f_{RX} = 4 * B_{RATE} = 38460$ Hz (scelta che sarà commentata in seguito).

È necessaria la presenza di un'unità di controllo che gestisca l'attivazione delle macchine sequenziali che lavorano a frequenza $f_{RX} < f_{CLK}$.

L'attivatore si occupa di abilitare il funzionamento delle macchine sequenziali sincrone operanti a frequenza f_{RX}, una volta ogni 25 periodi di f_{CLK}.

5.2 Blocco Clock Prescaler

Blocco Clock Prescaler si occupa di dividere la frequenza del clock presente in maniera nativa su TerasIC DE10-Lite Board, pari a $f_{CLK50MHz} = 50$ MHz, in $f_{clk} = 1$ MHz.

DESCRIZIONE

```
//CLOCK_PRESCALER

//CLOCK_PRESCALER divide Fclk = 50 MHz in Fclk = 1 MHz.
//CLK_50    : Clock 50 Mhz
//CLK_1     : Clock 1 MHz

module CLOCK_PRESCALER ( CLK50, CLK_1 );
input CLK50;
output CLK_1;
reg CLK_1;
reg [5:0] ACTUAL_COUNT1;
always @ ( posedge CLK50 )
begin
    if ( ACTUAL_COUNT1 < 25)
    begin
            CLK_1 <= 1'b1;
            ACTUAL_COUNT1 <= ACTUAL_COUNT1 + 1'b1;
    end
    else if ( ACTUAL_COUNT1 < 50)
        begin
                CLK_1 <= 1'b0;
                ACTUAL_COUNT1 <= ACTUAL_COUNT1 + 1'b1;
        end
        else
        begin
                CLK_1 <= 1'b1;
                ACTUAL_COUNT1 <= 1'b0;
        end
end
endmodule
```

5.3 Blocco Enable

Blocco Enable si occupa di generare il segnale di attivazione delle macchine sequenziali sincrone operanti a frequenza $f_{RX} = 38460$ Hz, clockate con il segnale di riferimento f_{CLK}.

DESCRIZIONE

```
//ENABLE

//ENABLE gestisce l'attivazione delle macchine sequenziali sincrone a 38460 Hz.

//CLK_1 : Clock 1 MHz
//EN    : Enable di macchina -> Un'attivazione ogni 25 colpi di clock ( attivazione a
38460 Hz ~ 9615 * 4 Hz )

module ENABLE ( CLK_1, EN );
input CLK_1;
output EN;
reg EN;
reg [4:0] ACTUAL_COUNT2;
always @ ( posedge CLK_1 )
begin
    if ( ACTUAL_COUNT2 < 25 )
    begin
        EN <= 1'b0;
        ACTUAL_COUNT2 <= ACTUAL_COUNT2 + 1'b1;
    end
    else
    begin
            EN <= 1'b1;
            ACTUAL_COUNT2 <= 1'b0;
    end
end
endmodule
```

5.4 Blocco Data Receiver

Blocco Data Receiver si occupa di riconoscere l'inizio di una trasmissione da parte dell'unità di calcolo Microchip PIC18LF4550 e rivelare, bit a bit, il byte trasmesso.

Si descrive nel seguito il protocollo di comunicazione UART presente sul micro controllore Microchip PIC18LF4550.

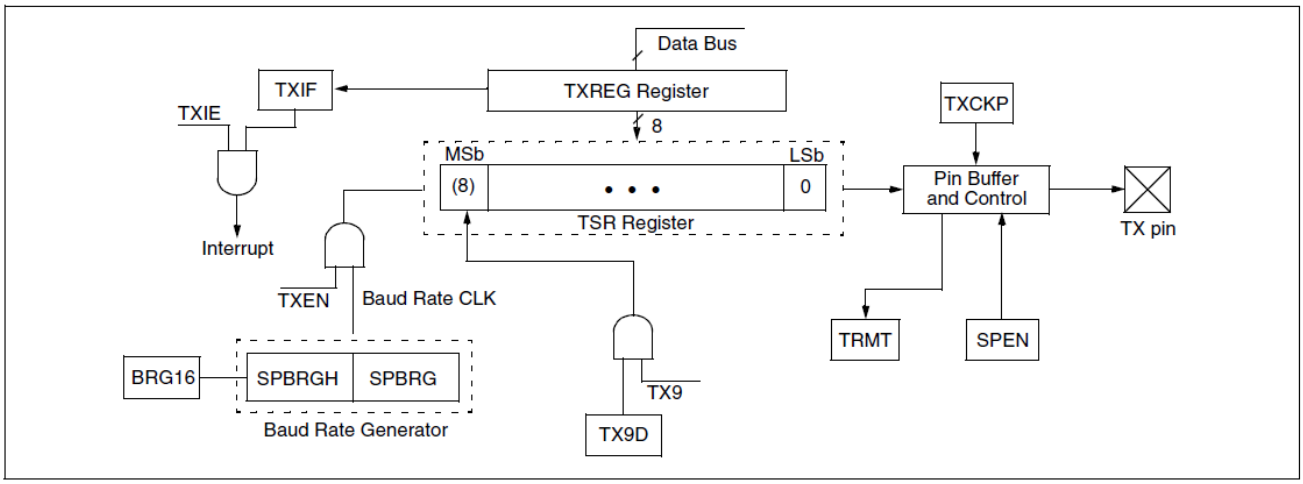

Figura: Schema a blocchi del trasmettitore UART su PIC18LF4550

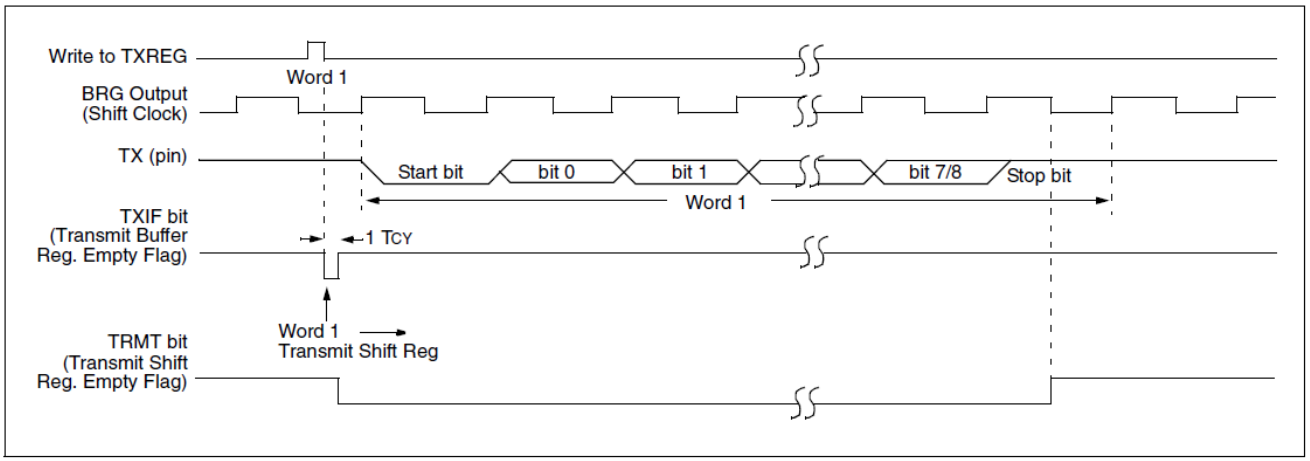

Figura: Diagramma di temporizzazione di trasmissione seriale UART su PIC18LF4550

Pin TX assume il valore 1 in assenza di trasmissione.
L'inizio di invio di informazione è segnalato dal posizionamento del pin TX al valore 0 (Start bit) per la durata di un periodo a frequenza B_{RATE}.
Successivamente avviene la trasmissione degli 8 bit contenuti nel byte presente nel registro di trasmissione TSR Register.
Infine pin TX assume il valore 1 (Stop bit) per indicare la fine della comunicazione.

Si descrivono nel seguito le regole utilizzate per il campionamento del segnale in ricezione.

La tipologia di protocollo utilizzato per la trasmissione di informazione (UART) prevede una comunicazione di tipo asincrono.

È necessario che il sistema in ricezione sia in grado di ricevere ed interpretare correttamente i bit trasmessi; per far si che ciò avvenga è necessario che esso lavori ad una frequenza maggiore della frequenza di trasmissione B_{RATE}.

Da un'analisi accurata risulta necessario che $f_{RX} = 4 * B_{RATE} = 38460$ Hz.

Operando un sovra campionamento il sistema non è ancora immune ad errori in ricezione. Il caso peggiore è quello in cui il fronte di salita del clock f_{RX} coincida col fronte di discesa di pin TX nel momento di trasmissione dello start bit.
Questa situazione può evolvere in due epiloghi: il riconoscimento istantaneo o il riconoscimento al periodo successivo del clock f_{RX}, dello start bit.
Nel caso in cui lo start bit venga campionato immediatamente, 4 cicli di clock f_{RX} successivi alla rivelazione, il sistema si troverebbe nuovamente a campionare in corrispondenza del fronte di transizione del Bit0 causando una nuova situazione di incertezza che potrebbe portare all'errato campionamento di Bit0.
Identica situazione si presenterebbe per tutti gli ulteriori bit trasmessi.

La soluzione è operare il campionamento di Bit0 dopo 5 cicli del clock f_{RX} anziché 4: qualora il riconoscimento dello start bit sia istantaneo, il campionamento di Bit0 non avverrebbe in corrispondenza dell'instante di commutazione, ma bensì $T = \frac{1}{f_{RX}}$ dopo.
Da questo momento in poi il campionamento degli ulteriori bit ricevuti avviene ogni 4 cicli del clock f_{RX}.

Si descrive nel seguito la realizzazione della macchina a stati implementata allo scopo di ricevere e rivelare il byte trasmesso.

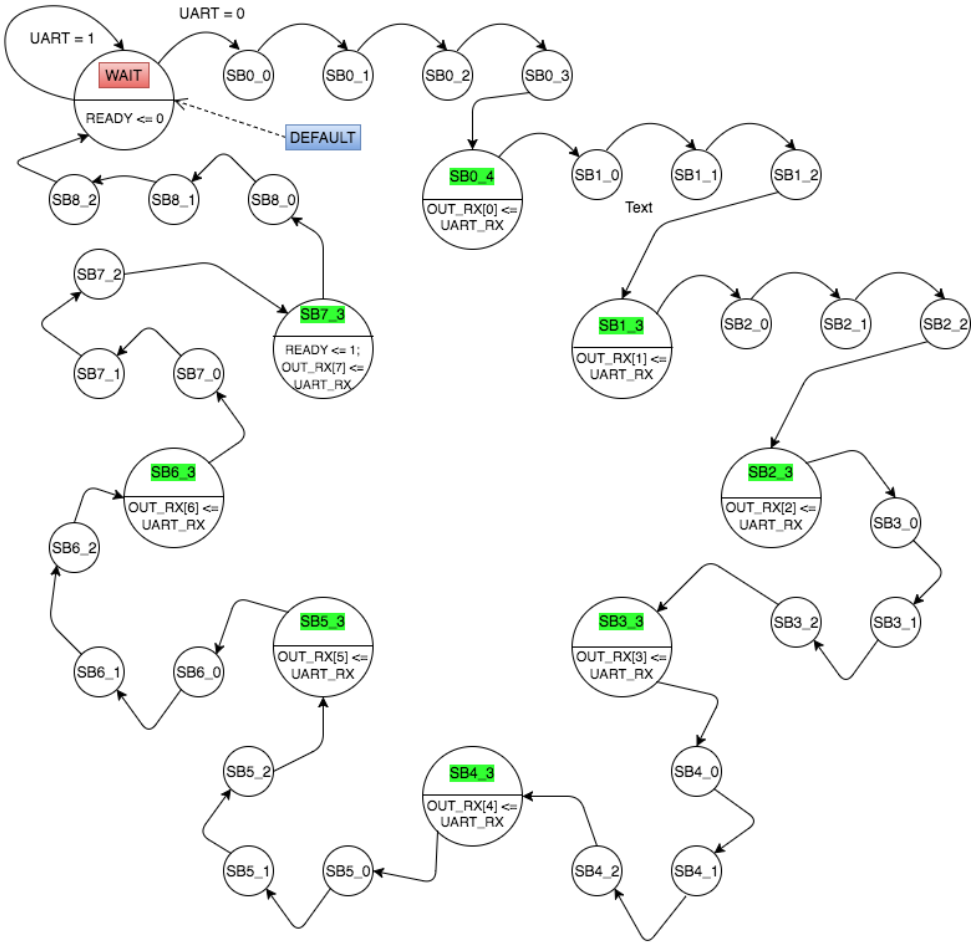

Figura: Diagramma a palle della macchina sequenziale Data Receiver

Elenco degli stati:

- WAIT;
- SBO_0;
- SB0_1;
- SB0_2;
- SB0_3;
- SB0_4 // Rivelazione Bit0;
- SB1_0;
- SB1_1;
- SB1_2;
- SB1_3 // Rivelazione Bit1;

- SB2_0;
- SB2_1;
- SB2_2;
- SB2_3 // Rivelazione Bit2;
- SB3_0;
- SB3_1;
- SB3_2;
- SB3_3 // Rivelazione Bit3;
- SB4_0;
- SB4_1;
- SB4_2;
- SB4_3 // Rivelazione Bit4;
- SB5_0;
- SB5_1;
- SB5_2;
- SB5_3 // Rivelazione Bit5;
- SB6_0;
- SB6_1;
- SB6_2;
- SB6_3 // Rivelazione Bit6;
- SB7_0;
- SB7_1;
- SB7_2;
- SB7_3 // Rivelazione Bit7 e validazione di OUT_RX;
- SB8_0;
- SB8_1;
- SB8_2.

Il sistema si trova inizialmente nello stato iniziale WAIT.
Nell'istante in cui viene riconosciuto l'inizio di una trasmissione abbiamo la commutazione dallo stato WAIT allo stato SB0_0.

I bit rivelati vanno a comporre serialmente un registro di uscita OUT_RX validato tramite il segnale READY nello stato SB7_3.

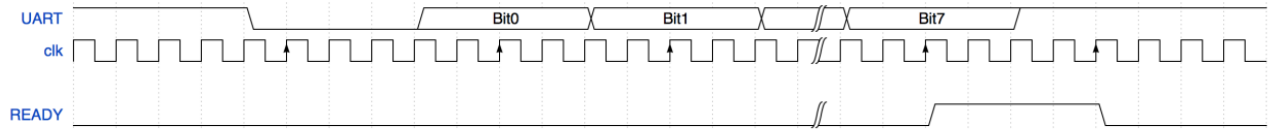

Figura: Diagramma di temporizzazione di ricezione serial UART su FPGA Altera

Nelle figura precedente sono indicati gli istanti di campionamento di f_{CLK} coerentemente con quanto descritto nella sezione riguardante le regole di campionamento

DESCRIZIONE

```
//DATA RECEIVER

//DATA RECEIVER riceve dati in forma seriale e li converte in un'uscita parallela ad 8
bit.

//UART_TX   : input seriale proveniente dal uC PIC18F4550
//CLK_1     : Clock 1 MHz
//EN        : Abilitazione della macchina
//OUT_RX    : Uscita parallela ad 8 bit
//READY     : READY = H valida OUT_RX

module DATA_RECEIVER ( UART_RX, CLK_1, EN, OUT_RX, READY );
input UART_RX, CLK_1, EN;
output [7:0] OUT_RX;
output READY;
//Codifica degli stati: codifica a numero minimo di bit
parameter LISTEN = 6'd0,
            SB0_0 = 6'd1,
            SB0_1 = 6'd2,
            SB0_2 = 6'd3,
            SB0_3 = 6'd4,
            SB0_4 = 6'd5,
            SB1_0 = 6'd6,
            SB1_1 = 6'd7,
            SB1_2 = 6'd8,
            SB1_3 = 6'd9,
            SB2_0 = 6'd10,
            SB2_1 = 6'd11,
            SB2_2 = 6'd12,
```

```
            SB2_3 = 6'd13,
            SB3_0 = 6'd14,
            SB3_1 = 6'd15,
            SB3_2 = 6'd16,
            SB3_3 = 6'd17,
            SB4_0 = 6'd18,
            SB4_1 = 6'd19,
            SB4_2 = 6'd20,
            SB4_3 = 6'd21,
            SB5_0 = 6'd22,
            SB5_1 = 6'd23,
            SB5_2 = 6'd24,
            SB5_3 = 6'd25,
            SB6_0 = 6'd26,
            SB6_1 = 6'd27,
            SB6_2 = 6'd28,
            SB6_3 = 6'd29,
            SB7_0 = 6'd30,
            SB7_1 = 6'd31,
            SB7_2 = 6'd32,
            SB7_3 = 6'd33,
            SB8_0 = 6'd34,
            SB8_1 = 6'd35,
            SB8_2 = 6'd36,
            SB8_3 = 6'd37;
reg [5:0] S_NEXT, S_REG;
reg [7:0] OUT_RX;
reg READY;
//Memoria di Stato
always @ ( posedge CLK_1 )
begin
    if ( EN ) S_REG <= S_NEXT;
    else S_REG <= S_REG;
end
//Transizione di Stato
always @ ( S_REG, UART_RX )
begin
    case ( S_REG )
        LISTEN:
                if ( UART_RX ) S_NEXT = LISTEN;
                else S_NEXT = SB0_0;
        SB0_0: S_NEXT = SB0_1;
        SB0_1: S_NEXT = SB0_2;
        SB0_2: S_NEXT = SB0_3;
        SB0_3: S_NEXT = SB0_4;
        SB0_4: S_NEXT = SB1_0;
        SB1_0: S_NEXT = SB1_1;
```

```
        SB1_1: S_NEXT = SB1_2;
        SB1_2: S_NEXT = SB1_3;
        SB1_3: S_NEXT = SB2_0;
        SB2_0: S_NEXT = SB2_1;
        SB2_1: S_NEXT = SB2_2;
        SB2_2: S_NEXT = SB2_3;
        SB2_3: S_NEXT = SB3_0;
        SB3_0: S_NEXT = SB3_1;
        SB3_1: S_NEXT = SB3_2;
        SB3_2: S_NEXT = SB3_3;
        SB3_3: S_NEXT = SB4_0;
        SB4_0: S_NEXT = SB4_1;
        SB4_1: S_NEXT = SB4_2;
        SB4_2: S_NEXT = SB4_3;
        SB4_3: S_NEXT = SB5_0;
        SB5_0: S_NEXT = SB5_1;
        SB5_1: S_NEXT = SB5_2;
        SB5_2: S_NEXT = SB5_3;
        SB5_3: S_NEXT = SB6_0;
        SB6_0: S_NEXT = SB6_1;
        SB6_1: S_NEXT = SB6_2;
        SB6_2: S_NEXT = SB6_3;
        SB6_3: S_NEXT = SB7_0;
        SB7_0: S_NEXT = SB7_1;
        SB7_1: S_NEXT = SB7_2;
        SB7_2: S_NEXT = SB7_3;
        SB7_3: S_NEXT = SB8_0;
        SB8_0: S_NEXT = SB8_1;
        SB8_1: S_NEXT = SB8_2;
        SB8_2: S_NEXT = SB8_3;
        SB8_3: S_NEXT = LISTEN;
        default: S_NEXT = LISTEN;
    endcase
end
//Transizione di Uscita
always @ ( posedge CLK_1 )
begin
    if ( EN )
    begin
        OUT_RX <= OUT_RX;
        case ( S_REG )
            SB0_4: OUT_RX[0] <= UART_RX;
            SB1_3: OUT_RX[1] <= UART_RX;
            SB2_3: OUT_RX[2] <= UART_RX;
            SB3_3: OUT_RX[3] <= UART_RX;
            SB4_3: OUT_RX[4] <= UART_RX;
            SB5_3: OUT_RX[5] <= UART_RX;
```

```
            SB6_3: OUT_RX[6] <= UART_RX;
            SB7_3:
            begin
                OUT_RX[7] <= UART_RX;
                READY <= 1'b1;
            end
            default: READY <= 1'b0;
        endcase
    end
    else OUT_RX <= OUT_RX;
end
endmodule
```

5.5 Blocco Data Start Revelator

Blocco Data Start Revelator si occupa di associare ogni byte rivelato dal blocco Data Receiver, che corrisponde al posizionamento in gradi del servo motore, col corrispettivo servo motore.

Si descrive nel seguito il protocollo di assegnazione angolo/servo motore.

L'unità di calcolo Microchip PIC18LF4550 trasmette inizialmente un byte noto di inizio sequenza in cui tutti i bit sono settati al valore 1. In tal modo FPGA Altera è in grado di riconoscere l'inizio di ricezione di una sequenza di angoli da assegnare ai servo motori. Successivamente l'unità di calcolo invia i byte relativi angli angoli che i servo motori devo raggiungere, ordinati per ID motore crescente.
FPGA Altera non fa altro che associare ad ogni servo motore, il byte ricevuto in maniera sequenziale, dal motore con ID = 0 al motore con ID = 17.

Si descrive nel seguito la realizzazione della macchina a stati implementata allo scopo di rivelare l'inizio della sequenza ed associare ogni angolo ricevuto al proprio servo motore.

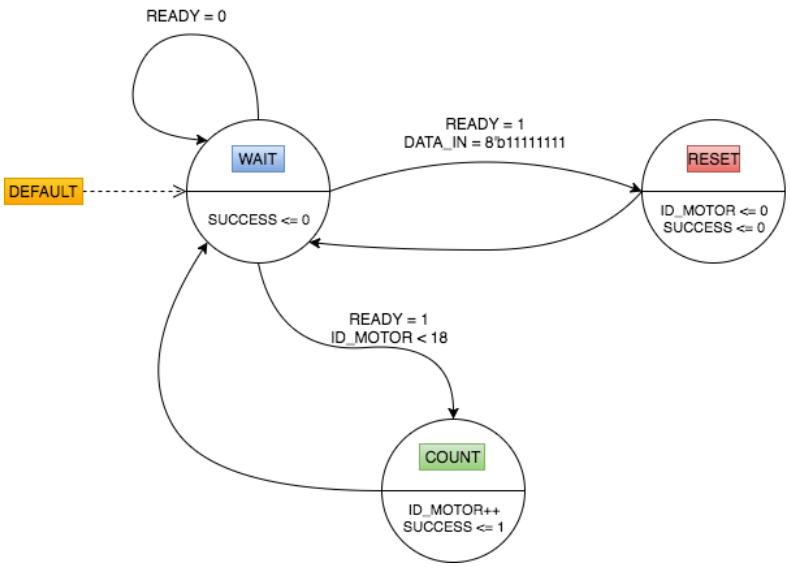

Figura: Diagramma a palle della macchina sequenziale Data Start Revelator

Elenco degli stati:

- WAIT;
- RESET;
- COUNT.

Il sistema si trova inizialmente nello stato iniziale WAIT.
Nell'istante in cui viene riconosciuta la rivelazione di un byte abbiamo la commutazione dallo stato.
Il successivo stato è determinato dal valore di OUT_RX del blocco Data Receiver: nel caso in cui OUT_RX = 11111111, il sistema passa nello stato RESET inizializzando ID_MOTOR per la ricezione dell'angolo di riferimento al servo motore 0; nel caso in cui OUT_RX != 11111111, il sistema passa nello stato COUNT associando l'angolo ricevuto col proprio servo motore e validando l'uscita del blocco attraverso il segnale SUCCESS.

DESCRIZIONE

```
//DATA START REVELATOR

//DATA START REVELATOR riconosce la codifica di inizio trasmissione (8'b11111111) e
successivamente identifica l'angolo al motore associato.

//DATA_IN   : input ad 8 bit, corrisponde all'output di DATA RECEIVER
//READY     : READY = H valida l'input
//CLK_1     : Clock 1 MHz
//EN        : Abilitazione della macchina
//ID_MOTOR  : ID del motore a cui è associato l'angolo
//ANGLE     : Angolo del motore
//SUCCESS   : SUCCESS = H valida ID_MOTOR e ANGLE

module DATA_START_REVELATOR ( DATA_IN, READY, CLK_1, EN, ID_MOTOR, ANGLE, SUCCESS );
input[7:0] DATA_IN;
input READY, CLK_1, EN;
output[4:0] ID_MOTOR;
output[7:0] ANGLE;
output SUCCESS;
//Codifica degli stati: codifica a numero minimo di bit
parameter WAIT = 2'b00,
            RESET = 2'b01,
            COUNT = 2'b10;
reg[1:0] S_NEXT, S_REG;
```

```
reg[4:0] ID_MOTOR, ID_MOTOR_NEXT;
reg[7:0] ANGLE;
reg SUCCESS, SUCCESS_NEXT;
//Memoria di Stato
always @ ( posedge CLK_1 )
begin
    if ( EN ) S_REG <= S_NEXT;
    else S_REG <= S_REG;
end
//Transizione di Stato
always @ ( S_REG, READY, DATA_IN )
begin
    case ( S_REG )
        WAIT:
                if ( READY )
                    if ( DATA_IN == 8'b11111111 ) S_NEXT = RESET;
                    else S_NEXT = COUNT;
                else S_NEXT = WAIT;
        RESET: S_NEXT = WAIT;
        COUNT: S_NEXT = WAIT;
        default: S_NEXT = WAIT;
    endcase
end
//RC Calcolo ID_MOTOR
always @ ( S_REG or ID_MOTOR)
begin
    case ( S_REG )
        WAIT:
        begin
                ID_MOTOR_NEXT = ID_MOTOR;
                SUCCESS_NEXT = 1'b0;
        end
        RESET:
        begin
                ID_MOTOR_NEXT = 5'b00000;
                SUCCESS_NEXT = 1'b0;
        end
        COUNT:
        begin
                if ( ID_MOTOR < 18 )
                begin
                    ID_MOTOR_NEXT = ID_MOTOR + 5'b00001;
                    SUCCESS_NEXT = 1'b1;
                end
                else if ( ID_MOTOR == 18)
                begin
                    ID_MOTOR_NEXT = ID_MOTOR + 5'b00001;
```

```
                    SUCCESS_NEXT = 1'b0;
                end
                else
                begin
                    ID_MOTOR_NEXT = ID_MOTOR;
                    SUCCESS_NEXT = 1'b0;
                end
        end
        default:
        begin
                ID_MOTOR_NEXT = ID_MOTOR;
                SUCCESS_NEXT = 1'b0;
        end
    endcase
end
//Memoria di Uscita
always @ ( posedge CLK_1 )
begin
    if ( EN )
    begin
        ID_MOTOR <= ID_MOTOR_NEXT;
        ANGLE <= DATA_IN;
        SUCCESS <= SUCCESS_NEXT;
    end
    else
    begin
        ID_MOTOR <= ID_MOTOR;
        ANGLE <= ANGLE;
        SUCCESS <= SUCCESS;
    end
end
endmodule
```

5.6 Blocco Angle To Time Allocator

Blocco Angle To Time Allocator traduce l'angolo rivelato dal blocco Data Start Revelator in un equivalente impulso di tensione necessario al fine di posizionare il servo motore associato.
Data l'unicità di ogni servo motore, l'operazione di conversione utilizza una serie di parametri che definiscono le caratteristiche di ogni motore.

Consideriamo legame di linearità tra tempo che l'impulso generato trascorre a livello alto e angolo raggiunto dal servo motore; l'operazione da compiere al fine di effettuare la traduzione è la seguente:

$$(1)\ T_{high} = \frac{Time_{180} - Time_0}{180} * Angle + Time_0$$

Dove:

- $Time_0$: durata dell'impulso per il posizionamento a 0°;
- $Time_{180}$: durata dell'impulso per il posizionamento a 180°;
- Angle: angolo espresso in gradi (°);
- T_{high}: durata dell'impulso generato per il posizionamento ad Angle.

La traduzione prevede l'utilizzo di operazioni di somma, moltiplicazione e divisione, con conseguente inserimento, a livello di sintesi, di blocchi sommatore, moltiplicatore e divisore.

Di seguito si descrive un metodo al fine di aggirare la necessità della presenza di un blocco divisore.
L'operazione di divisione che si compie è fatta dividendo per una costante.

Al posto di eseguire (1) si elaborano in sequenza queste tre operazioni:

1. $Time_{APP} = T_{high}*180 = (Time_{180} - Time_0)*Angle + Time_0*180$;
2. $Appoggio_1 = Time_{APP} * (0.00555\ldots)$;
3. $T_{high} = Appoggio_1[25:14]$.

Dove $Time_{APP}$ è un registro di appoggio di dimensione 20 bit ed $Appoggio_1$ è un registro di appoggio di dimensione 27 bit.

L'operazione 2 consente di approssimare la divisione per 180 con la moltiplicazione per 1/180.

Il troncamento di 1/180 prevede l'inserimento di un errore.
$[1/180]_2 = 0.000\ 000\ 010\ 110\ 110\ 000\ 01...$
$[1/180]_2 \sim\ 0.000\ 000\ 010\ 110\ 11$

Utilizzando 7 cifre significative, dopo la prima diversa da zero, si commette un errore relativo minore di 2^{-12} nel rappresentare 1/180 dato che i successivi 5 bit da considerare sarebbero stati nulli.

Infine si prelevano dal risultato ottenuto i 12 bit di interesse.

DESCRIZIONE

```
//ANGLETOTIME_ALLOCATOR

//ANGLETOTIME_ALLOCATOR  è una rete combinatoria che converte ANGLE, ampiezza dell'angolo
associato al servomotore ID_MOTOR, in HIGH_TIME, ampiezza in us del livello logico
//H del segnale PWM associato al servomotore ID_MOTOR.

//ANGLE     : Angolo del motore ID_MOTOR
//ID_MOTOR  : Numero del motore a cui è associato ANGLE
//HIGH_TIME : Tempo in us di durata dell'impulso

module ANGLETOTIME_ALLOCATOR ( ANGLE, ID_MOTOR, HIGH_TIME );
parameter[11:0] TIME180_GRAD_M1 = 12'd2272;
parameter[11:0] TIME180_GRAD_M2 = 12'd2287;
parameter[11:0] TIME180_GRAD_M3 = 12'd2294;
parameter[11:0] TIME180_GRAD_M4 = 12'd2317;
parameter[11:0] TIME180_GRAD_M5 = 12'd2342;
parameter[11:0] TIME180_GRAD_M6 = 12'd2262;
parameter[11:0] TIME180_GRAD_M7 = 12'd2348;
parameter[11:0] TIME180_GRAD_M8 = 12'd2283;
parameter[11:0] TIME180_GRAD_M9 = 12'd2316;
parameter[11:0] TIME180_GRAD_M10 = 12'd2306;
parameter[11:0] TIME180_GRAD_M11 = 12'd2229;
```

```
parameter[11:0] TIME180_GRAD_M12 = 12'd2344;
parameter[11:0] TIME180_GRAD_M13 = 12'd2233;
parameter[11:0] TIME180_GRAD_M14 = 12'd2242;
parameter[11:0] TIME180_GRAD_M15 = 12'd2278;
parameter[11:0] TIME180_GRAD_M16 = 12'd2243;
parameter[11:0] TIME180_GRAD_M17 = 12'd2321;
parameter[11:0] TIME180_GRAD_M18 = 12'd2294;
parameter[11:0] TIME0_GRAD_M1 = 12'd534;
parameter[11:0] TIME0_GRAD_M2 = 12'd527;
parameter[11:0] TIME0_GRAD_M3 = 12'd552;
parameter[11:0] TIME0_GRAD_M4 = 12'd548;
parameter[11:0] TIME0_GRAD_M5 = 12'd560;
parameter[11:0] TIME0_GRAD_M6 = 12'd521;
parameter[11:0] TIME0_GRAD_M7 = 12'd544;
parameter[11:0] TIME0_GRAD_M8 = 12'd535;
parameter[11:0] TIME0_GRAD_M9 = 12'd559;
parameter[11:0] TIME0_GRAD_M10 = 12'd534;
parameter[11:0] TIME0_GRAD_M11 = 12'd542;
parameter[11:0] TIME0_GRAD_M12 = 12'd580;
parameter[11:0] TIME0_GRAD_M13 = 12'd532;
parameter[11:0] TIME0_GRAD_M14 = 12'd537;
parameter[11:0] TIME0_GRAD_M15 = 12'd546;
parameter[11:0] TIME0_GRAD_M16 = 12'd538;
parameter[11:0] TIME0_GRAD_M17 = 12'd565;
parameter[11:0] TIME0_GRAD_M18 = 12'd540;
input[7:0] ANGLE;
input[4:0] ID_MOTOR;
output[11:0] HIGH_TIME;
reg[11:0] TIME180_GRAD, TIME0_GRAD;
wire[19:0] HIGH_TIME_180;
wire[26:0] APPOGGIO1;
assign HIGH_TIME_180 = ANGLE * ( TIME180_GRAD - TIME0_GRAD ) + 8'd180 * TIME0_GRAD;
assign APPOGGIO1 = HIGH_TIME_180 * 7'b1011011;
assign HIGH_TIME = APPOGGIO1[25:14];
always @ ( ID_MOTOR )
begin
    case ( ID_MOTOR )
        1:
        begin
            TIME180_GRAD = TIME180_GRAD_M1;
            TIME0_GRAD = TIME0_GRAD_M1;
        end
        2:
        begin
            TIME180_GRAD = TIME180_GRAD_M2;
            TIME0_GRAD = TIME0_GRAD_M2;
        end
```

```
3:
begin
    TIME180_GRAD = TIME180_GRAD_M3;
    TIME0_GRAD = TIME0_GRAD_M3;
end
4:
begin
    TIME180_GRAD = TIME180_GRAD_M4;
    TIME0_GRAD = TIME0_GRAD_M4;
end
5:
begin
    TIME180_GRAD = TIME180_GRAD_M5;
    TIME0_GRAD = TIME0_GRAD_M5;
end
6:
begin
    TIME180_GRAD = TIME180_GRAD_M6;
    TIME0_GRAD = TIME0_GRAD_M6;
end
7:
begin
    TIME180_GRAD = TIME180_GRAD_M7;
    TIME0_GRAD = TIME0_GRAD_M7;
end
8:
begin
    TIME180_GRAD = TIME180_GRAD_M8;
    TIME0_GRAD = TIME0_GRAD_M8;
end
9:
begin
    TIME180_GRAD = TIME180_GRAD_M9;
    TIME0_GRAD = TIME0_GRAD_M9;
end
10:
begin
    TIME180_GRAD = TIME180_GRAD_M10;
    TIME0_GRAD = TIME0_GRAD_M10;
end
11:
begin
    TIME180_GRAD = TIME180_GRAD_M11;
    TIME0_GRAD = TIME0_GRAD_M11;
end
12:
begin
```

```
            TIME180_GRAD = TIME180_GRAD_M12;
            TIME0_GRAD = TIME0_GRAD_M12;
        end
        13:
        begin
            TIME180_GRAD = TIME180_GRAD_M13;
            TIME0_GRAD = TIME0_GRAD_M13;
        end
        14:
        begin
            TIME180_GRAD = TIME180_GRAD_M14;
            TIME0_GRAD = TIME0_GRAD_M14;
        end
        15:
        begin
            TIME180_GRAD = TIME180_GRAD_M15;
            TIME0_GRAD = TIME0_GRAD_M15;
        end
        16:
        begin
            TIME180_GRAD = TIME180_GRAD_M16;
            TIME0_GRAD = TIME0_GRAD_M16;
        end
        17:
        begin
            TIME180_GRAD = TIME180_GRAD_M17;
            TIME0_GRAD = TIME0_GRAD_M17;
        end
        18:
        begin
            TIME180_GRAD = TIME180_GRAD_M18;
            TIME0_GRAD = TIME0_GRAD_M18;
        end
        default:
        begin
            TIME180_GRAD = 12'b0;
            TIME0_GRAD = 12'b0;
        end
    endcase;
end
endmodule
```

5.7 Blocco First Allocator

Blocco First Allocator memorizza sequenzialmente le traduzioni effettuate dal blocco Angle To Time Allocator .

Viene riempito un buffer di memorizzazione composto da 18 locazioni da 12 bit.
L'abilitazione di un flag DATA_VALID evidenzia al blocco successivo il completamento del buffer.

DESCRIZIONE

```
//FIRST_ALLOCATOR

//FIRST_ALLOCATOR riempie un buffer in maniera sequenziale con i parametri ricevuti da
ANGLETOIME_ALLOCATOR.
//Una volta completata l'allocazione si attiva DATA_VALID.

//HIGH_TIME : Tempo in us di durata dell'impulso
//ID_MOTOR  : Numero del motore a cui è associato HIGH_TIME
//SUCCESS   : Flag che valida HIGH_TIME
//CLK_1     : Clock 1 MHz
//EN        : Abilitazione della macchina
//DATA_MEMxx: Contiene il dato relativo al motore "xx"
//DATA_VALID: Flag che valida il completamento del buffer

module FIRST_ALLOCATOR ( HIGH_TIME, ID_MOTOR, SUCCESS, CLK_1, EN,
                                DATA_MEM1a, DATA_MEM2,  DATA_MEM3,  DATA_MEM4,  DATA_MEM5,
DATA_MEM6,  DATA_MEM7,  DATA_MEM8,  DATA_MEM9,
                                DATA_MEM10,    DATA_MEM11,    DATA_MEM12,    DATA_MEM13,
DATA_MEM14, DATA_MEM15, DATA_MEM16, DATA_MEM17, DATA_MEM18,
                                DATA_VALID
                                );
input[11:0] HIGH_TIME;
input[4:0] ID_MOTOR;
input SUCCESS, CLK_1, EN;
output [11:0]  DATA_MEM1a, DATA_MEM2,  DATA_MEM3,  DATA_MEM4,  DATA_MEM5,  DATA_MEM6,
DATA_MEM7,    DATA_MEM8,    DATA_MEM9,  DATA_MEM10,  DATA_MEM11,  DATA_MEM12,  DATA_MEM13,
DATA_MEM14, DATA_MEM15, DATA_MEM16, DATA_MEM17, DATA_MEM18;
output DATA_VALID;
reg DATA_VALID;
reg  [11:0]   DATA_MEM1a,  DATA_MEM2,   DATA_MEM3,   DATA_MEM4,   DATA_MEM5,   DATA_MEM6,
DATA_MEM7,    DATA_MEM8,    DATA_MEM9,  DATA_MEM10,  DATA_MEM11,  DATA_MEM12,  DATA_MEM13,
DATA_MEM14, DATA_MEM15, DATA_MEM16, DATA_MEM17, DATA_MEM18;
```

```
always @ ( posedge CLK_1 )
begin
    if ( EN )
    begin
        DATA_VALID <= 1'b0;
        DATA_MEM1a <= DATA_MEM1a;
        DATA_MEM2 <= DATA_MEM2;
        DATA_MEM3 <= DATA_MEM3;
        DATA_MEM4 <= DATA_MEM4;
        DATA_MEM5 <= DATA_MEM5;
        DATA_MEM6 <= DATA_MEM6;
        DATA_MEM7 <= DATA_MEM7;
        DATA_MEM8 <= DATA_MEM8;
        DATA_MEM9 <= DATA_MEM9;
        DATA_MEM10 <= DATA_MEM10;
        DATA_MEM11 <= DATA_MEM11;
        DATA_MEM12 <= DATA_MEM12;
        DATA_MEM13 <= DATA_MEM13;
        DATA_MEM14 <= DATA_MEM14;
        DATA_MEM15 <= DATA_MEM15;
        DATA_MEM16 <= DATA_MEM16;
        DATA_MEM17 <= DATA_MEM17;
        DATA_MEM18 <= DATA_MEM18;
        if ( SUCCESS )
            case ( ID_MOTOR )
                1 : DATA_MEM1a <= HIGH_TIME;
                2 : DATA_MEM2 <= HIGH_TIME;
                3 : DATA_MEM3 <= HIGH_TIME;
                4 : DATA_MEM4 <= HIGH_TIME;
                5 : DATA_MEM5 <= HIGH_TIME;
                6 : DATA_MEM6 <= HIGH_TIME;
                7 : DATA_MEM7 <= HIGH_TIME;
                8 : DATA_MEM8 <= HIGH_TIME;
                9 : DATA_MEM9  <= HIGH_TIME;
                10 : DATA_MEM10  <= HIGH_TIME;
                11 : DATA_MEM11  <= HIGH_TIME;
                12 : DATA_MEM12  <= HIGH_TIME;
                13 : DATA_MEM13  <= HIGH_TIME;
                14 : DATA_MEM14  <= HIGH_TIME;
                15 : DATA_MEM15  <= HIGH_TIME;
                16 : DATA_MEM16  <= HIGH_TIME;
                17 : DATA_MEM17  <= HIGH_TIME;
                18 :
                begin
                    DATA_MEM18 <= HIGH_TIME;
                    DATA_VALID <= 1'b1;
                end
```

```
            endcase
    end
    else
    begin
        DATA_VALID <= DATA_VALID;
        DATA_MEM1a <= DATA_MEM1a;
        DATA_MEM2 <= DATA_MEM2;
        DATA_MEM3 <= DATA_MEM3;
        DATA_MEM4 <= DATA_MEM4;
        DATA_MEM5 <= DATA_MEM5;
        DATA_MEM6 <= DATA_MEM6;
        DATA_MEM7 <= DATA_MEM7;
        DATA_MEM8 <= DATA_MEM8;
        DATA_MEM9 <= DATA_MEM9;
        DATA_MEM10 <= DATA_MEM10;
        DATA_MEM11 <= DATA_MEM11;
        DATA_MEM12 <= DATA_MEM12;
        DATA_MEM13 <= DATA_MEM13;
        DATA_MEM14 <= DATA_MEM14;
        DATA_MEM15 <= DATA_MEM15;
        DATA_MEM16 <= DATA_MEM16;
        DATA_MEM17 <= DATA_MEM17;
        DATA_MEM18 <= DATA_MEM18;
    end
end
endmodule
```

5.8 Blocco Second Allocator

Blocco Second Allocator si occupa di memorizzare parallelamente i dati provenienti dal blocco First Allocator una volta completato il riempimento del primo buffer.

Data l'equivalenza architetturale, anche il secondo buffer di memoria è composto da 18 locazioni a 12 bit.

DESCRIZIONE

```
//SECOND_ALLOCATOR

//SECOND_ALLOCATOR contiene il buffer contenente i parametri di conteggio dei Timer.

//CLK_1         : Clock 1 MHz
//EN            : Abilitazione della macchina
//DATA_VALID    : Flag che indica il completamento del buffer presente in FIRST_ALLOCATOR
//DATA_MEMxx    : Input contenente il dato relativo al motore "xx"
//DATA_TIMERxx  : Output contenente il dato relativo al motore "xx"

module SECOND_ALLOCATOR (  CLK_1, EN, DATA_VALID,
                                    DATA_MEM1a,   DATA_MEM2,      DATA_MEM3,      DATA_MEM4,
DATA_MEM5,  DATA_MEM6,  DATA_MEM7,  DATA_MEM8,  DATA_MEM9,
                                    DATA_MEM10,    DATA_MEM11,    DATA_MEM12,    DATA_MEM13,
DATA_MEM14, DATA_MEM15, DATA_MEM16, DATA_MEM17, DATA_MEM18,
                                    DATA_TIMER1a, DATA_TIMER2,  DATA_TIMER3,  DATA_TIMER4,
DATA_TIMER5,  DATA_TIMER6,  DATA_TIMER7,  DATA_TIMER8,
                                  DATA_TIMER9,  DATA_TIMER10, DATA_TIMER11, DATA_TIMER12,
DATA_TIMER13, DATA_TIMER14, DATA_TIMER15, DATA_TIMER16,
                                    DATA_TIMER17, DATA_TIMER18
                                    );
input DATA_VALID, CLK_1, EN;
input[11:0]  DATA_MEM1a,  DATA_MEM2,    DATA_MEM3,    DATA_MEM4,    DATA_MEM5,    DATA_MEM6,
DATA_MEM7,    DATA_MEM8,    DATA_MEM9,  DATA_MEM10,  DATA_MEM11,  DATA_MEM12,  DATA_MEM13,
DATA_MEM14, DATA_MEM15, DATA_MEM16, DATA_MEM17, DATA_MEM18;
output  [11:0]    DATA_TIMER1a,  DATA_TIMER2,    DATA_TIMER3,    DATA_TIMER4,    DATA_TIMER5,
DATA_TIMER6,    DATA_TIMER7,    DATA_TIMER8,    DATA_TIMER9,  DATA_TIMER10,  DATA_TIMER11,
DATA_TIMER12,  DATA_TIMER13,  DATA_TIMER14,  DATA_TIMER15,  DATA_TIMER16,  DATA_TIMER17,
DATA_TIMER18;
reg  [11:0]    DATA_TIMER1a,  DATA_TIMER2,    DATA_TIMER3,    DATA_TIMER4,    DATA_TIMER5,
DATA_TIMER6,    DATA_TIMER7,    DATA_TIMER8,    DATA_TIMER9,  DATA_TIMER10,  DATA_TIMER11,
DATA_TIMER12,  DATA_TIMER13,  DATA_TIMER14,  DATA_TIMER15,  DATA_TIMER16,  DATA_TIMER17,
DATA_TIMER18;
```

```
always @ ( posedge CLK_1 )
begin
    if ( EN )
    begin
        if ( DATA_VALID )
        begin
            DATA_TIMER1a <= DATA_MEM1a;
            DATA_TIMER2 <= DATA_MEM2;
            DATA_TIMER3 <= DATA_MEM3;
            DATA_TIMER4 <= DATA_MEM4;
            DATA_TIMER5 <= DATA_MEM5;
            DATA_TIMER6 <= DATA_MEM6;
            DATA_TIMER7 <= DATA_MEM7;
            DATA_TIMER8 <= DATA_MEM8;
            DATA_TIMER9 <= DATA_MEM9;
            DATA_TIMER10 <= DATA_MEM10;
            DATA_TIMER11 <= DATA_MEM11;
            DATA_TIMER12 <= DATA_MEM12;
            DATA_TIMER13 <= DATA_MEM13;
            DATA_TIMER14 <= DATA_MEM14;
            DATA_TIMER15 <= DATA_MEM15;
            DATA_TIMER16 <= DATA_MEM16;
            DATA_TIMER17 <= DATA_MEM17;
            DATA_TIMER18 <= DATA_MEM18;
        end
        else
        begin
            DATA_TIMER1a <= DATA_TIMER1a;
            DATA_TIMER2 <= DATA_TIMER2;
            DATA_TIMER3 <= DATA_TIMER3;
            DATA_TIMER4 <= DATA_TIMER4;
            DATA_TIMER5 <= DATA_TIMER5;
            DATA_TIMER6 <= DATA_TIMER6;
            DATA_TIMER7 <= DATA_TIMER7;
            DATA_TIMER8 <= DATA_TIMER8;
            DATA_TIMER9 <= DATA_TIMER9;
            DATA_TIMER10 <= DATA_TIMER10;
            DATA_TIMER11 <= DATA_TIMER11;
            DATA_TIMER12 <= DATA_TIMER12;
            DATA_TIMER13 <= DATA_TIMER13;
            DATA_TIMER14 <= DATA_TIMER14;
            DATA_TIMER15 <= DATA_TIMER15;
            DATA_TIMER16 <= DATA_TIMER16;
            DATA_TIMER17 <= DATA_TIMER17;
            DATA_TIMER18 <= DATA_TIMER18;
        end
    end
```

```
    else
    begin
        DATA_TIMER1a <= DATA_TIMER1a;
        DATA_TIMER2 <= DATA_TIMER2;
        DATA_TIMER3 <= DATA_TIMER3;
        DATA_TIMER4 <= DATA_TIMER4;
        DATA_TIMER5 <= DATA_TIMER5;
        DATA_TIMER6 <= DATA_TIMER6;
        DATA_TIMER7 <= DATA_TIMER7;
        DATA_TIMER8 <= DATA_TIMER8;
        DATA_TIMER9 <= DATA_TIMER9;
        DATA_TIMER10 <= DATA_TIMER10;
        DATA_TIMER11 <= DATA_TIMER11;
        DATA_TIMER12 <= DATA_TIMER12;
        DATA_TIMER13 <= DATA_TIMER13;
        DATA_TIMER14 <= DATA_TIMER14;
        DATA_TIMER15 <= DATA_TIMER15;
        DATA_TIMER16 <= DATA_TIMER16;
        DATA_TIMER17 <= DATA_TIMER17;
        DATA_TIMER18 <= DATA_TIMER18;
    end
end
endmodule
```

5.9 Blocco Motor Timer

Blocco Motor Timer genera il segnale di controllo PWM di un servo motore.

DESCRIZIONE

```
//MOTOR_TIMER

//MOTOR_TIMER preleva la durata dell'impulso da generare e costruisce un segnale
periodico di perido T = 20ms.

//CLK_1         : Clock 1 MHz
//DATA_HIGHTIME : Tempo in us di durata dell'impulso
//OUT_MOTOR     : Output PWM

module MOTOR_TIMER( CLK_1, DATA_HIGHTIME, OUT_MOTOR );
input CLK_1;
input[11:0] DATA_HIGHTIME;
output OUT_MOTOR;
reg[11:0] REG_HIGHTIME;
reg OUT_MOTOR;
reg[14:0] INDEX_COUNT;
always @ ( posedge CLK_1 )
begin
    if ( INDEX_COUNT < REG_HIGHTIME )
    begin
        INDEX_COUNT <= INDEX_COUNT + 1'b1;
        OUT_MOTOR <= 1'b1;
    end
    else
        if ( INDEX_COUNT < 15'd20000 )
        begin
            INDEX_COUNT <= INDEX_COUNT + 1'b1;
            OUT_MOTOR <= 1'b0;
        end
        else
        begin
            OUT_MOTOR <= 1'b0;
            INDEX_COUNT <= 15'd1;
            REG_HIGHTIME <= DATA_HIGHTIME;
        end
end
endmodule
```

5.10 Blocco Control

Blocco Control è una rete combinatoria di controllo che gestisce la comunicazione utente – unità di calcolo Microchip PIC18LF4550.

Al fine di pilotare gli algoritmi di controllo implementati sul micro controllore, si fornisce all'utente la possibilità di utilizzare 6 switch e 2 button presenti sulla board TerasIC DE10-Lite.

La selezione di ogni switch comporta l'abilitazione della funzione desiderata:

- No switch <==> Posizionamento di default;
- SW0 <==> Traslazione lungo l'asse X;
- SW1 <==> Traslazione lungo l'asse Y;
- SW2 <==> Traslazione lungo l'asse Z;
- SW3 <==> Rotazione attorno l'asse X;
- SW4 <==> Rotazione attorno l'asse Y;
- SW5 <==> Rotazione attorno l'asse Z;

La pressione di un button, piuttosto che l'altro, determina l'incremento o il decremento della traslazione/rotazione selezionata.

DESCRIZIONE

```
//CONTROL

//CONTROL preleva la durata dell'impulso da generare e costruisce un segnale periodico di
perido T = 20ms.

//SWx       : Switch x
//KEY0_L    : Pulsante 0 attivo basso
//KEY1_L    : Pulsante 1 attivo basso
//CODED     : Codifica il tipo di controllo selezionato
//LED       : Attiva un determinato LED
//KEY_OUT0  : !KEY0_L
//KEY_OUT1  : !KEY1_L
```

```
module CONTROL ( SW0, SW1, SW2, SW3, SW4, SW5, KEY0_L, KEY1_L, CODED, LED, KEY_OUT0,
KEY_OUT1 );
input SW0, SW1, SW2, SW3, SW4, SW5, KEY0_L, KEY1_L;
output KEY_OUT0, KEY_OUT1;
output [2:0] CODED;
output [5:0] LED;
reg KEY_OUT0, KEY_OUT1;
reg [2:0] CODED;
reg [5:0] LED;
parameter
            VOID = 3'd0,
            S5 = 3'd1,
            S4 = 3'd2,
            S3 = 3'd3,
            S2 = 3'd4,
            S1 = 3'd5,
            S0 = 3'd6;
// Blocco Always di codifica degli Switch
always @ (*)
begin
    if (SW5)
    begin
        CODED = S5;
        LED = 6'b100000;
    end
    else if (SW4)
    begin
        CODED = S4;
        LED = 6'b010000;
    end
    else if (SW3)
    begin
        CODED = S3;
        LED = 6'b001000;
    end
    else if (SW2)
    begin
        CODED = S2;
        LED = 6'b000100;
    end
    else if (SW1)
    begin
        CODED = S1;
        LED = 6'b000010;
    end
    else if (SW0)
    begin
```

```
        CODED = S0;
        LED = 6'b000001;
    end
    else
    begin
    CODED   = VOID;
    LED = 6'b000000;
    end
end
// Blocco Always di controllo dei pulsanti
always @ (*)
begin
    if ( !KEY0_L & !KEY1_L)
    begin
        KEY_OUT0 = 1'b0;
        KEY_OUT1 = 1'b0;
    end
    else if ( !KEY0_L & KEY1_L )
    begin
        KEY_OUT0 = 1'b1;
        KEY_OUT1 = 1'b0;
    end
    else if ( KEY0_L & !KEY1_L )
    begin
        KEY_OUT0 = 1'b0;
        KEY_OUT1 = 1'b1;
    end
    else
    begin
        KEY_OUT0 = 1'b0;
        KEY_OUT1 = 1'b0;
    end
end
endmodule
```

Capitolo 6: Implementazione Hardware

Si riportano nel seguito i risultati post Analysis & Synthesis e Fitting.

6.1 Analysis & Synthesis

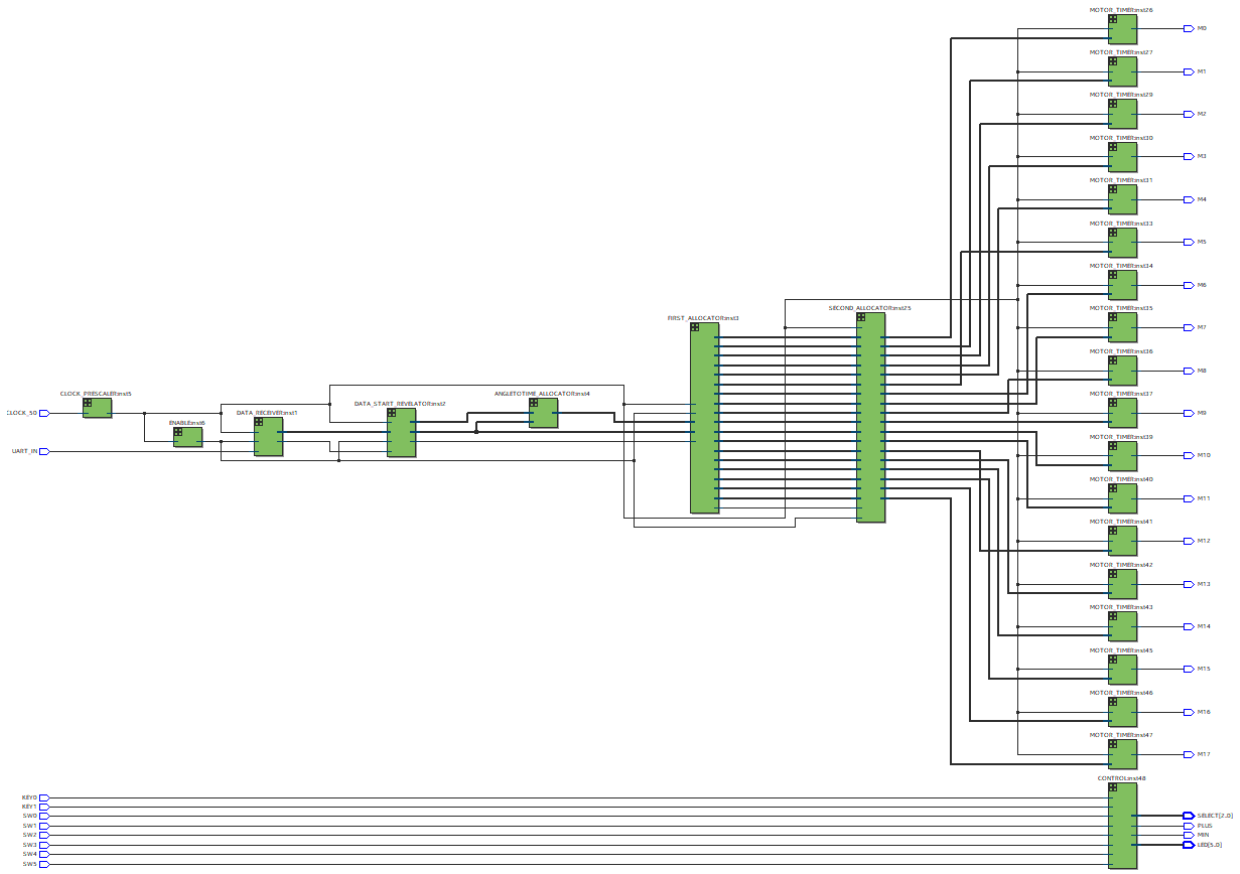

Figura: Visualizzazione RTL post sintesi

Analysis & Synthesis Summary	
<<Filter>>	
Analysis & Synthesis Status	Successful - Thu Dec 28 14:56:34 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	AUGUSTO_FPGA_PROJECT
Top-level Entity Name	AUGUSTO_FPGA_PROJECT
Family	MAX 10
Total logic elements	1,137
Total combinational functions	453
Dedicated logic registers	758
Total registers	758
Total pins	39
Total virtual pins	0
Total memory bits	0
Embedded Multiplier 9-bit elements	6
Total PLLs	0
UFM blocks	0
ADC blocks	0

Figura: Risultati Analysis & Synthesis

6.2 Fitter

Fitter Summary	
<<Filter>>	
Fitter Status	Successful - Thu Dec 28 14:57:02 2017
Quartus Prime Version	17.0.0 Build 595 04/25/2017 SJ Lite Edition
Revision Name	AUGUSTO_FPGA_PROJECT
Top-level Entity Name	AUGUSTO_FPGA_PROJECT
Family	MAX 10
Device	10M50DAF484C7G
Timing Models	Final
Total logic elements	908 / 49,760 (2 %)
Total combinational functions	453 / 49,760 (< 1 %)
Dedicated logic registers	750 / 49,760 (2 %)
Total registers	750
Total pins	39 / 360 (11 %)
Total virtual pins	0
Total memory bits	0 / 1,677,312 (0 %)
Embedded Multiplier 9-bit elements	6 / 288 (2 %)
Total PLLs	0 / 4 (0 %)
UFM blocks	0 / 1 (0 %)
ADC blocks	0 / 2 (0 %)

Figura: Risultati Fitter

Capitolo 7: Simulazioni

Si descrivono nel seguito le simulazioni effettuate a livello funzionale e timing.

7.1 Simulazione funzionale

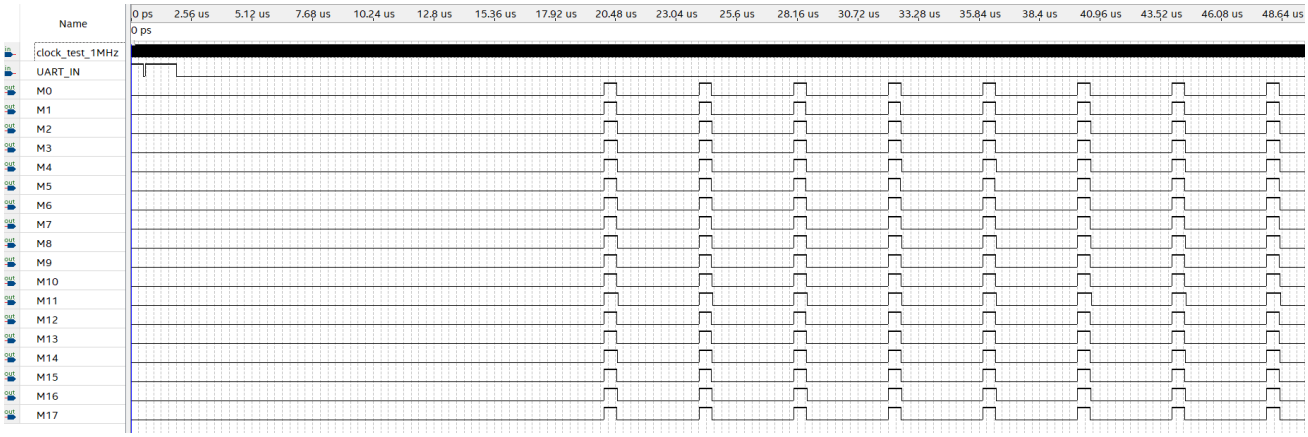

Figura: Risultati della simulazione funzionale

7.2 Simulazione timing

Si riportano nel seguito i risultati ottenuti dalle simulazioni timing effettuate sull'architettura implementata.

La simulazione fa riferimento al caso in cui:

- Vcc = 1.2V;
- Temp = 85°C;

Slow 1200mV 85C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name	Note
1	42.27 MHz	42.27 MHz	CLOCK_PRESCALER:inst5|CLK_1	
2	381.39 MHz	250.0 MHz	CLOCK_50	limit due to minimum period restriction (max I/O toggle rate)

Figura: Risultati sulla frequenza massima di funzionamento

Slow 1200mV 85C Model Setup Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_50	17.378	0.000
2	CLOCK_PRESCALER:inst5|CLK_1	976.341	0.000

Figura: Risultati dello slack sui tempi di setup [ns]

Slow 1200mV 85C Model Hold Summary

<<Filter>>

	Clock	Slack	End Point TNS
1	CLOCK_PRESCALER:inst5|CLK_1	0.340	0.000
2	CLOCK_50	0.426	0.000

Figura: Risultati dello slack sui tempi di hold [us]

Capitolo 8: Conclusioni

Le simulazioni funzionali e timing confermano il raggiungimento dell'obiettivo.

Il risultato finale è un'architettura dedicata estremamente performante che risolve i problemi legati all'accuratezza con cui veniva definita la posizione dell'esapode presenti nel sistema elettronico di bordo costituito da due unità di calcolo Microchip PIC18LF4550.

Il passo successivo consiste nella rimozione dell'unità di calcolo Microchip PIC18LF4550 con la sintesi di un Soft Core CPU su FPGA Altera MAX 10.

Si rinuncia così alla necessità di un'interfaccia di comunicazione UART interna all'applicazione e si introduce la possibilità di risolvere le complesse equazioni matematiche, legate alla cinematica inversa, che determinano la posizione dell'esapode, attraverso l'implementazione di hardware dedicato.

Bibliografia

[1] Alessandro Paghi, "Progettazione e realizzazione della struttura meccanica e del sistema di controllo di un robot esapode", University of Siena, July 2016.

[2] Microchip, "PIC18F2455/2550/4455/4550 Data Sheet", 2006.

[3] Microchip, "MPLAB X IDE User's Guide", 2015.

[4] Microchip, "MPLAB XC8 C Compiler User's Guide", 2015.

[5] TerasIC, "DE10-Lite User Manual", 2017.