Tesina di
SISTEMI EMBEDDED

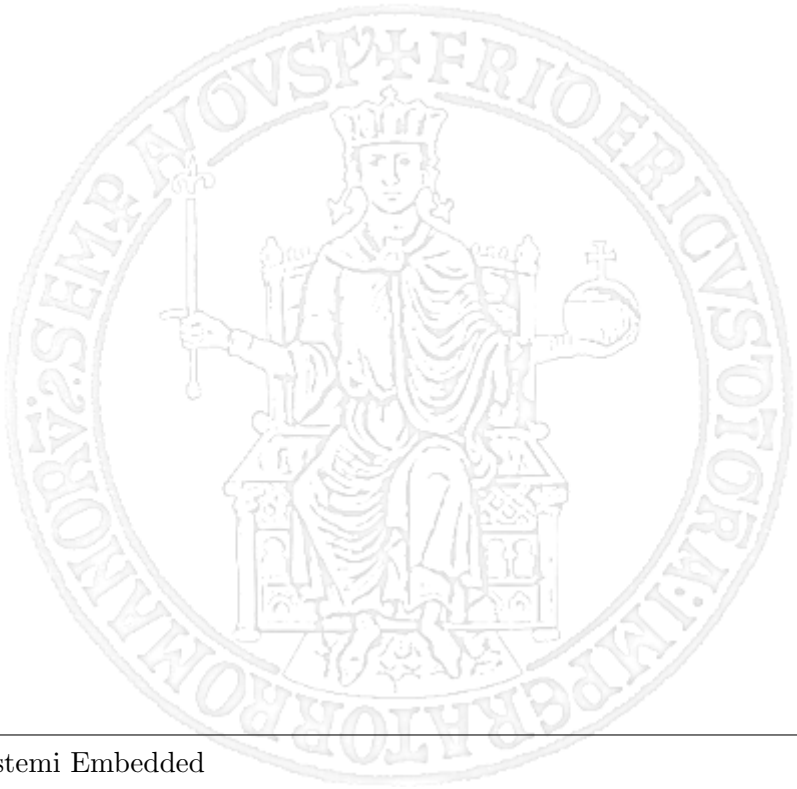
Elaborato 1: GPIO - Zybo

Prof. Antonino Mazzeo

Andrea Aletto - Matr. M63/582
Daniele Passaretti - Matr. M63/554
Maurizio Sepe - Matr. M63/494
Luigi Venuso - Matr. M63/537

A.A. 2015/2016

Indice



1 Elaborato 1 : GPIO

Obiettivo di questo primo elaborato è la realizzazione di un *driver* per il controllo di *led*, *switch* e *button* da parte del processore presente sulla *Zynq*. Al fine di realizzare la comunicazione tra processore e *device* si deve sintetizzare su *fpga* una rete di interconnessione: *AXI4Lite*.

1.1 SISTEMA DI INTERCONNESSIONE: AXI4Lite

1.1.1 IP-Core Custom

Il *software* utilizzando per definire l'architettura è *Vivado 2015.4*. Creato un nuovo progetto, il primo passo da compiere è definire l'*IP-Core* per l'*AXI4Lite* al cui interno vi deve essere il componente per la comunicazione con i *GPIO*. Nell'immagine successiva viene mostrata la creazione dell'interfaccia *AXI4Lite* che andremo a modificare per integrare la comunicazione con i device richiesti nella traccia:

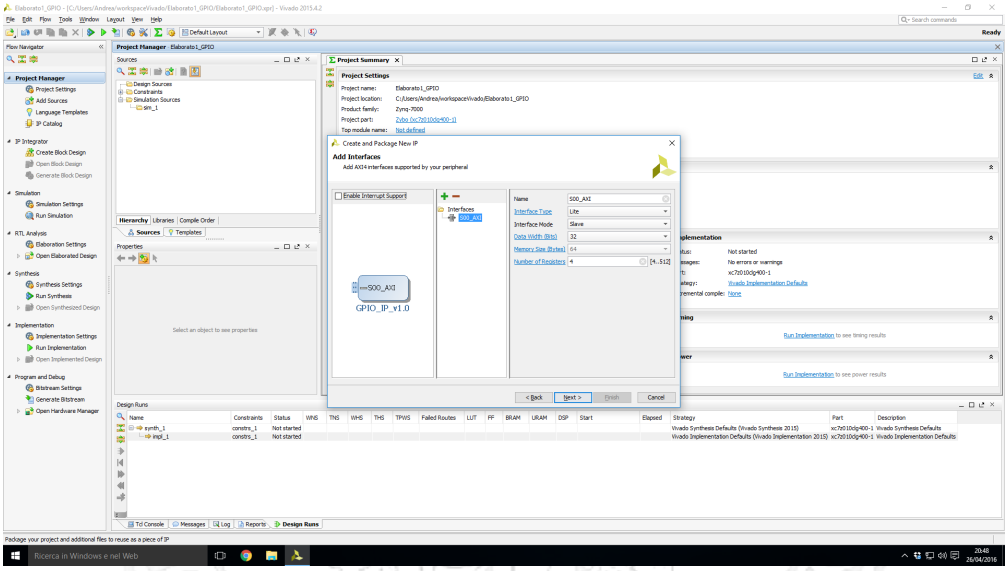


Figura 1.1:

Generato il VHDL del bus *AXI4Lite* andiamo a definire l'entità *gpio_pad.vhd* che rappresenta il componente per la gestione di un singolo segnale in logica *three-state*:

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5  entity gpio_pad is
6      Port ( en : in STD_LOGIC;
7             rw_n : in STD_LOGIC;
8             write : in STD_LOGIC;
9             read : out STD_LOGIC;
10            pad : inout STD_LOGIC);
11 end gpio_pad;
12
13 architecture Dataflow of gpio_pad is
14
15 begin
16     with en select read<=pad when '1', '0' when others;
17     with rw_n select pad <= write and en when '0', 'Z' when others;
18
19 end Dataflow;
```

Codice 1.1: gpio_pad

Si osserva che quando il segnale `en` è alto e `rw_n` è diverso da '0', allora il segnale `pad` viene mappato su `read` e la porta funziona in lettura; mentre, quando `rw_n` è pari a '0' il segnale di `write` è messo in *AND* con `en` e viene mappato su `pad`, forzando la porta in scrittura.
Dovendo gestire complessivamente 12 segnali (led, switch e button), andiamo ad implementare un componente generico (`GPIO_array`), per gestire i device in lettura o scrittura. Mostriamo quindi il componente:

```
1
2  library IEEE;
3  use IEEE.STD_LOGIC_1164.ALL;
4
5
6  entity gpio_array is
7      generic(N : natural := 12);
8      Port ( en : in STD_LOGIC_VECTOR (N-1 downto 0);
9             rw_n : in STD_LOGIC_VECTOR (N-1 downto 0);
10            write : in STD_LOGIC_VECTOR (N-1 downto 0);
11            read : out STD_LOGIC_VECTOR (N-1 downto 0);
12            pad : inout STD_LOGIC_VECTOR (N-1 downto 0));
13 end gpio_array;
14
15 architecture Structural of gpio_array is
16
17     COMPONENT gpio_pad is
18         Port ( en : in STD_LOGIC;
19                rw_n : in STD_LOGIC;
```

```
20                write : in STD_LOGIC;
21                read : out STD_LOGIC;
22                pad : inout STD_LOGIC);
23    END COMPONENT;
24
25 begin
26
27    array_of_gpio: for i in 0 to N-1 generate
28        gpio_pad_inst: gpio_pad port map(
29            en => en(i),
30            rw_n => rw_n(i),
31            write => write(i),
32            read => read(i),
33            pad => pad(i)
34        );
35    end generate;
36
37 end Structural;
```

Codice 1.2: gpio_pad

Andiamo ora ad integrare l'array di porte three-state all'interno dell' IP-Core *AXI4Lite*, apportando modifiche al bus inizialmente generato:

- istanziando il componente all'interno di AXI4Lite e mappando con i rispettivi registri
- sostituendo il registro `slv_reg3` con un segnale asincrono in lettura *read_ext* e mappando con reg_data_out all'interno del process che controlla il GPIO
- aggiungendo il generic per la dimensione del `gpio_array` fino alla top esterna dell' AXI4Lite e portando i segnali di `pad` fino all'esterno della top dell'IP-Core.

```
352    slv_reg_rden <= axi_arready and S_AXI_ARVALID and (not axi_rvalid) ;
353
354    process (read_ext, slv_reg0, slv_reg1, slv_reg2, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
355    variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
356    begin
357        -- Address decoding for reading registers
358        loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
359        case loc_addr is
360          when b"00" =>
361            reg_data_out <= slv_reg0;
362          when b"01" =>
363            reg_data_out <= slv_reg1;
364          when b"10" =>
365            reg_data_out <= slv_reg2;
366          when b"11" =>
367            reg_data_out  <= (others => '0');
368            reg_data_out(N-1 downto 0) <= read_ext;
369          when others =>
370            reg_data_out  <= (others => '0');
371        end case;
372    end process;
373
```

Figura 1.2:

```
395   gpio_array_inst: gpio_array generic map(N=>N)
396   port map(
397      en => slv_reg0(N-1 downto 0),
398      rw_n => slv_reg1(N-1 downto 0),
399      write => slv_reg2(N-1 downto 0),
400      read => read_ext,
401      pad => pad
402   );
403
404   -- User logic ends
```

Figura 1.3:

Ora andiamo a fare il package del nostro IP-Core ottenendo il risultato in figura:

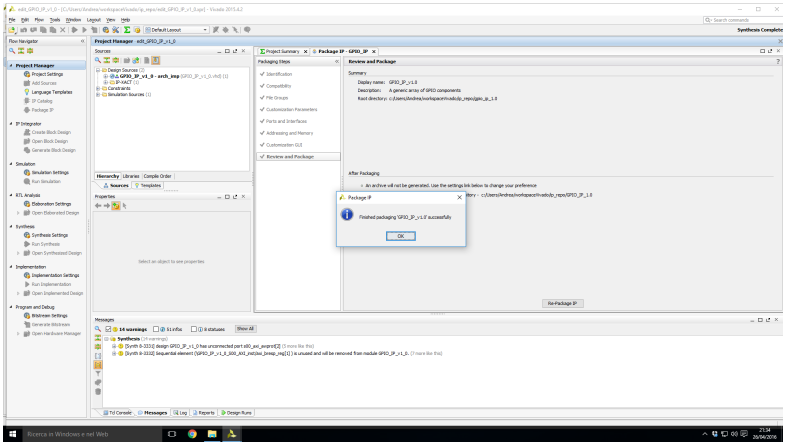


Figura 1.4:

1.1.2 Integrazione Design Board

Ottenuto il nostro IP-Core, dobbiamo integrarlo all'interno del sistema, quindi torniamo sul progetto principale e andiamo a prendere dall' IP-Repository, l'IP-Core appena generato e wrappato. Inoltre istanziamo l'IP del processore della Zybo nel Design come mostrato in figura 1.5

Come si osserva, anche dalla figura 1.5 dobbiamo interconnettere il processore ZYNQ7 con AXI4Lite; per fare ciò *Vivado* ci permette di fare in maniera automatica l'interconnessione, aggiungendo AXI Interconnect e il Processor System Reset. Infine per concludere il Design dobbiamo portare fuori il segnale `pad`, quindi fare il *make esternal*, ottenendo il risultato mostrato in figura 1.6.

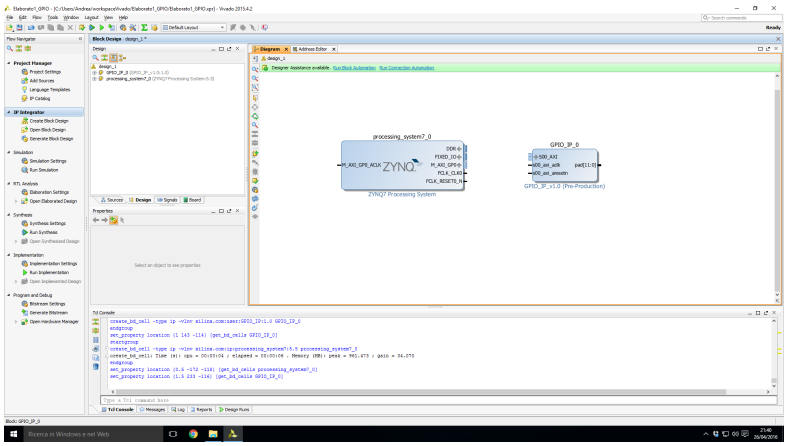


Figura 1.5:

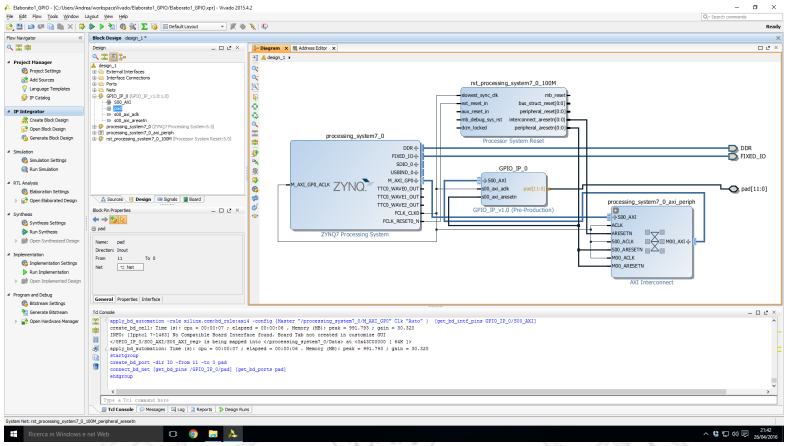


Figura 1.6:

Concluso il Design andiamo a wrappare e a definire il file dei costraint in cui settiamo i pin dei device(Led, Switch, Button) rispetto al *pad* come mostrato in figura 1.7.

Infine prima di passare allo sviluppo software del Driver generiamo il bitstream per programmare l'FPGA ed esportiamo l'Hardware ottenendo il risultato in figura 1.8.

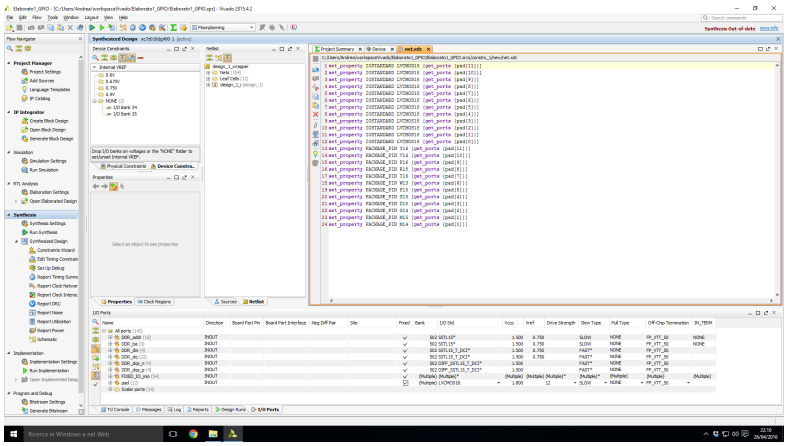


Figura 1.7:

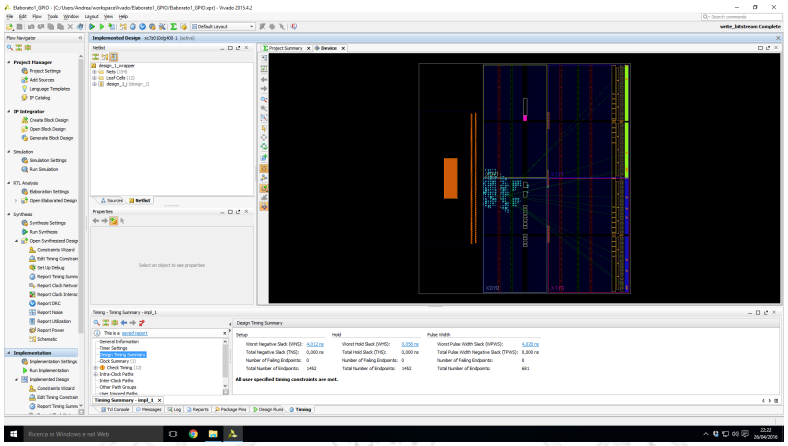


Figura 1.8:

1.2 Driver: C

Ora dobbiamo scrivere la libreria contenente le funzione per controllare e leggere Led, Switch e Button che useremo poi per scrivere il driver richiesto.
All'interno del progetto Vivado in cui abbiamo generato il bitstream apriamo l'SDK nel quale troviamo un progetto contenente le configurazioni della platform. Dovendo scrivere un Driver Custom andiamo a creare un nuovo *Application Project* in C, definendone poi la piattaforma e il processore target.

1.2.1 Libreria HAL driver

Andiamo a definire la libreria che ci permette di abilitare i device, leggere lo stato e controllare Led, Switch e Button. Tutti e tre necessiteranno di funzioni per l'abilitazione,

ma mentre per i led avrò funzioni di write per accendere o spegnerli, nel caso di switch e button ho funzioni di lettura per leggere lo stato di questi e capire quando sono premuti. Possiamo osservare le funzioni implementate nel file header :

```
/*
 * zybo_bsp.h
 *
 *  Created on: 27/apr/2016
 *      Author: DanieleMacBook
 */

#ifndef ZYBO_BSP_H_
#define ZYBO_BSP_H_

#include <stdint.h>
#include "GPIO_IP.h"
#include "xparameters.h"


#define GPIO_BASEADDRESS XPAR_GPIO_IP_0_S00_AXI_BASEADDR

#define GPIO_ENABLE GPIO_IP_S00_AXI_SLV_REG0_OFFSET
#define GPIO_RWN GPIO_IP_S00_AXI_SLV_REG1_OFFSET
#define GPIO_WRITE GPIO_IP_S00_AXI_SLV_REG2_OFFSET
#define GPIO_READ GPIO_IP_S00_AXI_SLV_REG3_OFFSET  //SLV_REG3 non e' altro che READ_DATA_OUT


#define GPIO_RWN_CONFIG 0x00000FF0 //Maschera per abilitare switch e bottoni

#define HIGH 1
#define LOW 0
#define ENABLE 1
#define DISABLE 0

typedef uint8_t LED_t;
#define LED_0    (LED_t) 0
#define LED_1    (LED_t) 1
#define LED_2    (LED_t) 2
#define LED_3    (LED_t) 3
#define LED_0_MASK  (uint32_t)(0x1 << LED_0)
#define LED_1_MASK  (uint32_t)(0x1 << LED_1)
#define LED_2_MASK  (uint32_t)(0x1 << LED_2)
#define LED_3_MASK  (uint32_t)(0x1 << LED_3)

typedef uint8_t SWITCH_t;
#define SWITCH_0    (SWITCH_t) 4
#define SWITCH_1    (SWITCH_t) 5
#define SWITCH_2    (SWITCH_t) 6
```

```
45 #define SWITCH_3    (SWITCH_t) 7
46 #define SWITCH_0_MASK  (uint32_t)(0x1 << SWITCH_0)
47 #define SWITCH_1_MASK  (uint32_t)(0x1 << SWITCH_1)
48 #define SWITCH_2_MASK  (uint32_t)(0x1 << SWITCH_2)
49 #define SWITCH_3_MASK  (uint32_t)(0x1 << SWITCH_3)
50
51 typedef uint8_t BUTTON_t;
52 #define BUTTON_0    (BUTTON_t) 8
53 #define BUTTON_1    (BUTTON_t) 9
54 #define BUTTON_2    (BUTTON_t) 10
55 #define BUTTON_3    (BUTTON_t) 11
56 #define BUTTON_0_MASK  (uint32_t)(0x1 << BUTTON_0)
57 #define BUTTON_1_MASK  (uint32_t)(0x1 << BUTTON_1)
58 #define BUTTON_2_MASK  (uint32_t)(0x1 << BUTTON_2)
59 #define BUTTON_3_MASK  (uint32_t)(0x1 << BUTTON_3)
60
61 void GPIO_init(uint32_t* baseAddress);
62 void GPIO_enableLed(LED_t led, uint8_t value);
63 void GPIO_enableLedMask(uint32_t mask, uint8_t value);
64 void GPIO_writeLed(LED_t led, uint8_t value);
65 void GPIO_writeLedMask(uint32_t mask, uint8_t value);
66 void GPIO_ToggleLed(LED_t led, uint8_t value);
67 void GPIO_ToggleLedMask(uint32_t mask, uint8_t value);
68
69
70 void GPIO_enableSwitch(SWITCH_t swtch, uint8_t value);
71 void GPIO_enableSwitchMask(uint32_t mask, uint8_t value);
72 uint32_t GPIO_readSwitch(SWITCH_t swtch);
73 uint32_t GPIO_readSwitchMask(uint32_t mask);
74
75 void GPIO_enableButton(BUTTON_t button, uint8_t value);
76 void GPIO_enableButtonMask(uint32_t mask, uint8_t value);
77 uint32_t GPIO_readButton(BUTTON_t button);
78 uint32_t GPIO_readButtonMask(uint32_t mask);
79
80
81 #endif /* ZYBO_BSP_H_ */
```

Codice 1.3: zybo_bsp.h

Si osserva nel file header, che abbiamo prima definito gli indirizzi dei registri che abbiamo precedentemente interconnesso in hardware nel bus AXI4Lite, dove mappavamo `GPIO_ENABLE`, `GPIO_RWN`, `GPIO_WRITE` e `GPIO_READ` rispettivamente con i registri REG_0, REG_1, REG_2, e REG_3. Le define dei registri sono dei semplici offset rispetto al registro base che troviamo all'interno del file che definisce la platform; questi offset, essendo registri da 32 bit ognuno, sono 0(registri base), 4, 8 e 12; infatti tali offset siccome saranno gli indici del vettore che punta la base address, saranno divise per 4, così che GPIO_pointer possa puntare a tutti e 4 i registri usando i rispettivi OFFSET/4 come indice. Naturalmente insieme alle *define* degli offset, dobbiamo andare a prendere

il GPIO_BASEADDRESS che definisce il BaseAddress del GPIO_AXI di cui troviamo il suo indirizzo reale nella libreria che genera Vivado `xparameters.h`. Bisogna però fare attenzione, poichè bisogna verificare se in tale libreria l'indirizzo del BaseAddress è corretto. Nel nostro caso a causa di un bug di Vivado lo abbiamo corretto, come mostrato in figura:

```
/******************************************************************/

/* Definitions for driver GPIO_IP */
#define XPAR_GPIO_IP_NUM_INSTANCES 1

/* Definitions for peripheral GPIO_IP_0 */
#define XPAR_GPIO_IP_0_DEVICE_ID 0
#define XPAR_GPIO_IP_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_GPIO_IP_0_S00_AXI_HIGHADDR 0x43C000FF


/******************************************************************/
```

Figura 1.9: xparameters.h

Ora definiti gli indirizzi e le define per lavorare sui registri che le funzioni della nostra libreria dovrà leggere e settare, andiamo a definire gli offset all'interno di ogni singolo registro, per evitare di dover ricalcolare ogni volta che utilizziamo le funzioni gli spiazzamenti all'interno dei singoli registri di Led, Button e Switch. Si osserva che useremo sempre variabili a 32 bit, poichè i registri sono a 32 bit e dobbiamo quindi indirizzarli come tali. Se pure avessimo voluto utilizzare variabili a 16 bit, non sapendo se i registri vengono scritti o letti in little endian o big endian potremmo avere problemi di inversione dei bit. Ogni singolo registro è così organizzato:

bit 0	bit 1	bit 2	bi3	bit 4	bit 5	bit 6	bit 7	bit 8	bit 9	bit 10	bit 11	bit 12-31
L0	L1	L2	L3	S0	S1	S2	S3	B0	B1	B2	B3	inutilizzati

Figura 1.10: Registro da 32 bit: L(Led), S(Switch), B(Button)

In base alla figura dell'organizzazione interna dei singoli registri abbiamo definito le define dei dispositivi e le relative MASK.

All'interno della libreria per i led abbiamo definito le funzioni di `enableLed`, ed `enableLedMask`, la prima ci permette di passare in input il singolo led che siamo interessati ad abilitare o a disabilitare e poi il parametro ENABLE o DISABLE. La funzione provvederà a settare il valore desiderato all'interno del registro nella posizione del led specificato. Viene creata la Mask relativa al Led da noi messo in input e chiamta la `enableLedMask`. La funzione `enableLedMask`, se riceverà in input un enable andrà a forzare un 1 nel registro di ENABLE del GPIO, nella posizione relativa al Led che si vuole abilitare; se invece riceverà un DISABLE in input andrà a forzare nel registro ENABLE del GPIO, uno 0 nella posizione relativa al Led che si vuole disabilitare. Per forzare valori all'interno di registri senza dover modificare l'intero registro, usiamo delle

maschere, chiamando la funzione `GPIO_enforceMask`.
NOTA:

- **1:** Per forzare un 1 basta mettere il registro in OR con una maschera avente tutti 0 e 1 solo nei bit da forzare ad 1
- **0:** Per forzare uno 0 basta mettere il registro in AND con una maschera che ha tutti 1 e 0 solo nei bit da forzare a 0. Per ottenere questa maschera in C risulta più semplice shiftare gli 1 dove si vogliono gli zero poichè nel resto della maschera si hanno di default tutti 0 e negare l'intera mascherà così da ottenere gli 0 dove volevamo e gli 1 nel resto della maschera.

Spiegate le funzioni dell'enable dei LED per gli enable di SWITCH e BUTTON abbiamo le stesse identiche funzioni, in cui cambia solo il range di bit all'interno del registro GPIO_ENABLE che andiamo a settare. Per lavorare sui led dobbiamo scrivere delle funzioni che ci permettono di fare la *write* sul relativo registro, quindi andiamo a definire la `writeLed` e la `writeLedMask` che sono simili alle rispettive di abilitazione. La differenza è il registro su cui andiamo a forzare 1(HIGH) o 0(LOW) che in questo caso non è più il registro *GPIO_ENABLE* ma è il registro *GPIO_WRITE*.

Siccome oltre alla write ci è stato chiesto di poter fare il Toggle sui led, abbiamo deciso di implementare una serie di funzioni che ci portano ad invertire il bit nel registro WRITE andando a mettere in input il LED su cui vogliamo che avvenga il Toggle e inserendo HIGH per invertire il bit e LOW per lasciare il registro invariato; si osserva che per fare ciò è bastato mettere HIGH(1) e LOW(0) in XOR con i bit del registro WRITE su cui vogliamo che avvenga il toggle bit. Infatti la XOR non fa nient'altro che invertire il valore 0→1 o 1→0 se ha un 1 in input e non variare nulla se ha uno 0 in input. Le funzioni utilizzate per il Toggle bit sono: `ToggleLed`, `ToggleLedMask` e la funzione interna `TogglenforceMask`.

Infine le ultime funzioni che ci mancano sono quelle di lettura, per leggere lo stato di Switch e Button, in questo caso abbiamo sempre una `readButton` o `readSwitch` e una `readButtonMask` o `reatSwitchMask` che prendono in ingresso il Led o lo Switch di cui vogliamo leggere lo stato. Viene generata così la maschera da mettere in AND con il valore che leggiamo dal registro GPIO_READ e naturalmente se il risultato della AND è 1 abbiamo che il Button è premuto o lo Switch è alzato, altrimenti è 0.

Non ci siamo soffermati a descrivere la funzione `GPIO_init` ma naturalmente quando avviamo il Driver dobbiamo assegnare l'indirizzo del Base Address al puntatore e configurare i registri degli switch e dei button in lettura quindi in tale funzione andiamo a mettere nel registro GPIO_RWN degli 1 in corrispondenza dei bit che vogliamo leggere(Button e Switch) e dei 0 in quelli che andremo a scrivere Led.

Ora mostriamo l'implementazione della libreria appena descritta:

```
1 /*
2  * zybo_bsp.c
3  *
4  *  Created on: 27/apr/2016
5  *      Author: DanieleMacBook
```

```
6   */
7
8
9   #include "zybo_bsp.h"
10
11  uint32_t* GPIO_pointer;
12
13  void GPIO_init(uint32_t* baseAddress){
14    GPIO_pointer=baseAddress;
15    GPIO_pointer[GPIO_ENABLE/4] = 0x0;
16    GPIO_pointer[GPIO_RWN/4] = GPIO_RWN_CONFIG;
17  }
18
19  void GPIO_enforceMask(uint32_t reg, uint32_t mask, uint32_t
        mask_value){
20      GPIO_pointer[reg/4] |= (mask & mask_value);
21      GPIO_pointer[reg/4] &= (~mask | mask_value);
22  }
23  void GPIO_TogglenforceMask(uint32_t reg, uint32_t mask, uint32_t
        mask_value){
24      GPIO_pointer[reg/4] ^= (mask & mask_value);
25
26  }
27  void GPIO_enableLed(LED_t led, uint8_t value){
28    GPIO_enableLedMask(0x1<<led, value);
29  }
30
31  void GPIO_enableLedMask(uint32_t mask, uint8_t value){
32    if((0x0000000F & mask) == mask){
33      value == ENABLE ? GPIO_enforceMask(GPIO_ENABLE, mask, 0x0000000F)
            : GPIO_enforceMask(GPIO_ENABLE, mask, 0x0);
34    }
35  }
36
37  void GPIO_writeLed(LED_t led, uint8_t value){
38    GPIO_writeLedMask(0x1<<led, value);
39  }
40
41  void GPIO_writeLedMask(uint32_t mask, uint8_t value){
42    if((0x0000000F & mask) == mask){
43      value == HIGH ? GPIO_enforceMask(GPIO_WRITE, mask, 0x0000000F) :
          GPIO_enforceMask(GPIO_WRITE, mask, 0x0);
44      }
45
46  }
47  void GPIO_ToggleLed(LED_t led, uint8_t value){
48      GPIO_ToggleLedMask(0x1<<led, value);
49
50  }
```

```
51 void GPIO_ToggleLedMask(uint32_t mask, uint8_t value){
52   if((0x0000000F & mask) == mask){
53       value == HIGH ? GPIO_TogglenforceMask(GPIO_WRITE, mask, 0
             x0000000F) : GPIO_TogglenforceMask(GPIO_WRITE, mask, 0x0);
54       }
55 }
56 void GPIO_enableSwitch(SWITCH_t swtch, uint8_t value){
57   GPIO_enableSwitchMask(0x1<<swtch, value);
58 }
59
60 void GPIO_enableSwitchMask(uint32_t mask, uint8_t value){
61   if((0x000000F0 & mask) == mask){
62     value == ENABLE ? GPIO_enforceMask(GPIO_ENABLE, mask, 0x000000F0)
           : GPIO_enforceMask(GPIO_ENABLE, mask, 0x0);
63   }
64 }
65
66 uint32_t GPIO_readSwitch(SWITCH_t swtch){
67   return GPIO_readSwitchMask(0x1<<swtch) == (0x1<<swtch);
68 }
69
70 uint32_t GPIO_readSwitchMask(uint32_t mask){
71   if((0x000000F0 & mask) == mask){
72       return GPIO_pointer[GPIO_READ/4] & mask;
73       }
74   return -1;
75 }
76
77 void GPIO_enableButton(BUTTON_t button, uint8_t value){
78   GPIO_enableButtonMask(0x1<<button, value);
79 }
80
81 void GPIO_enableButtonMask(uint32_t mask, uint8_t value){
82   if((0x00000F00 & mask) == mask){
83       value == ENABLE ? GPIO_enforceMask(GPIO_ENABLE, mask, 0
             x00000F00) : GPIO_enforceMask(GPIO_ENABLE, mask, 0x0);
84     }
85 }
86
87 uint32_t GPIO_readButton(BUTTON_t button){
88   return GPIO_readButtonMask(0x1<<button) == (0x1<<button);
89 }
90
91 uint32_t GPIO_readButtonMask(uint32_t mask){
92   if((0x00000F00 & mask) == mask){
93       return GPIO_pointer[GPIO_READ/4] & mask;
94       }
95   return -1;
96 }
```

Codice 1.4: zybo_bsp.c

1.2.2 Driver Custom

Ora scritta la libreria andiamo a scrivere lo specifico Driver richiesto:

1.2.2.1 Traccia Driver

Si scriva un Driver che in fase si **setup** se è premuto un button si accenda il led omologo al button premuto, mentre in fase di **loop** se alzato lo Switch avvenga il toggle del led omologo. Se lo Switch è abbassato lascia il led invariato.

1.2.2.2 Soluzione

Per scrivere tale driver andiamo a scrivere il main in cui inseriamo la funzione setup. Tale funzione dovrà chiamare la GPIO_init, abilitare tutti i dispositivi e poi controllare se c'è un button premuto, in caso affermativo accendere il rispettivo led. All'interno di un while invece andiamo ad implementare la funzione loop in cui chiamiamo la `GPIO_ToggleLed` per ogni switch e la funzione `GPIO_readSwitch` che passa in input 0 se lo switch è basso e 1 se è alto, determinando il toggle del Led. Il codice ottenuto è il seguente:

```
1  /*
2   * main.c
3   *
4   *  Created on: 27/apr/2016
5   *      Author: DanieleMacBook
6   */
7  /*Elaborato 1: In fase di setup si accende un led in corrispondenza
       del bottone omologo se premuto. Nella fase di loop
8   * se lo switch e' on il corrispettivo led e' in toggle, mentre se lo
       switch e' off il led rimane nell'ultimo stato.*/
9
10 #include <stdio.h>
11 #include <unistd.h>
12 #include "platform.h"
13 #include "zybo_bsp.h"
14
15 void print(char *str);
16
17
18 void setup();
19 void loop();
20
21 int main()
22 {
23   setup();
```

```
24    while(1)
25      loop();
26
27    cleanup_platform();
28      return 0;
29  }
30
31  void setup(){
32    init_platform();
33
34    GPIO_init(GPIO_BASEADDRESS);
35
36    //Abilitazione Led - Switch - Button
37    GPIO_enableLedMask(LED_0_MASK | LED_1_MASK | LED_2_MASK |
          LED_3_MASK , ENABLE);
38    GPIO_enableSwitchMask(SWITCH_0_MASK | SWITCH_1_MASK | SWITCH_2_MASK
          | SWITCH_3_MASK , ENABLE);
39    GPIO_enableButtonMask(BUTTON_0_MASK | BUTTON_1_MASK |BUTTON_2_MASK
          | BUTTON_3_MASK, ENABLE);
40
41    //Accensione Led se bottone omologo e' on
42    if(GPIO_readButton(BUTTON_0)) GPIO_writeLed(LED_0, HIGH);
43    if(GPIO_readButton(BUTTON_1)) GPIO_writeLed(LED_1, HIGH);
44    if(GPIO_readButton(BUTTON_2)) GPIO_writeLed(LED_2, HIGH);
45    if(GPIO_readButton(BUTTON_3)) GPIO_writeLed(LED_3, HIGH);
46  }
47
48  void loop(){
49
50    //Toggle del Led in funzione dello switch omologo
51    GPIO_ToggleLed(LED_0, GPIO_readSwitch(SWITCH_0));
52    GPIO_ToggleLed(LED_1, GPIO_readSwitch(SWITCH_1));
53    GPIO_ToggleLed(LED_2, GPIO_readSwitch(SWITCH_2));
54    GPIO_ToggleLed(LED_3, GPIO_readSwitch(SWITCH_3));
55
56    sleep(1);
57  }
```

Codice 1.5: main.c

Ora, finito di scrivere il Driver, andiamo a caricare il bitstream sulla board attraverso l' SDK con program FPGA ottenendo:

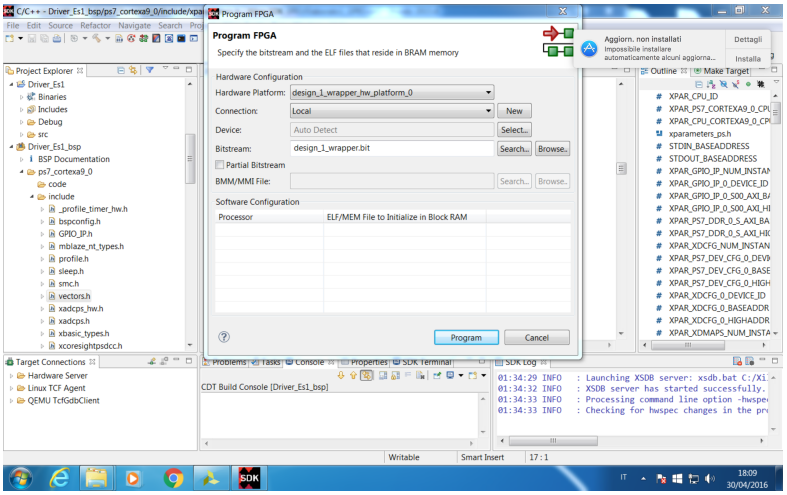


Figura 1.11:

Programmata la FPGA noteremo che sulla board si accende il led verde affianco al led di accensione indice del fatto che è pronta e possiamo caricare e avviare il driver scritto sul processore. Se si vuole avviare il programma in debugger, bisogna configure la board e la connessione sulla COM relativa alla board.

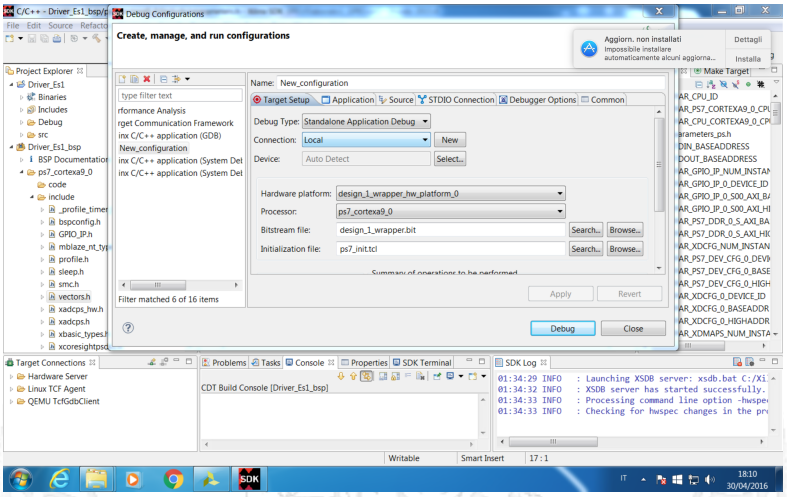


Figura 1.12: