
0.1 Soluzione

0.1.1 Descrizione GPIO

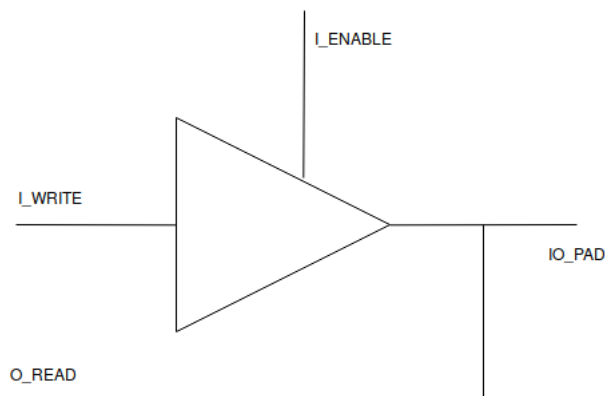


Figure 1: Schema singolo GPIO

Il GPIO (*general purpose input output*) permettere di leggere o scrivere un valore sul segnale *PAD* che è un segnale di *inout* per il componente.

Se il segnale di input *ENABLE* è asserito allora *WRITE* viene forzato sul segnale *PAD*, altrimenti il segnale *PAD* è di input per il componente. Il segnale *READ* mostra il valore corrente del segnale *PAD*. Si omette il codice del componente vhd in quanto basilare, verrà istanziato un componente `GPIO_Array` composto da un numero generico (*width*) di singoli GPIO.

0.1.2 Creazione Custom IP

Per utilizzare il componente è necessario che la sua interfaccia sia conforme al bus AXI. Vivado permettere di adattare il nostro componente `GPIO_Array` al bus AXI inglobandolo all'interno di un wrapper che ne implementa l'interfaccia. Si mostra di seguito il procedimento da seguire:

1. Dal menù di Vivado selezionare “**Tools->Create and Package New IP**”. Si aprirà la seguente finestra

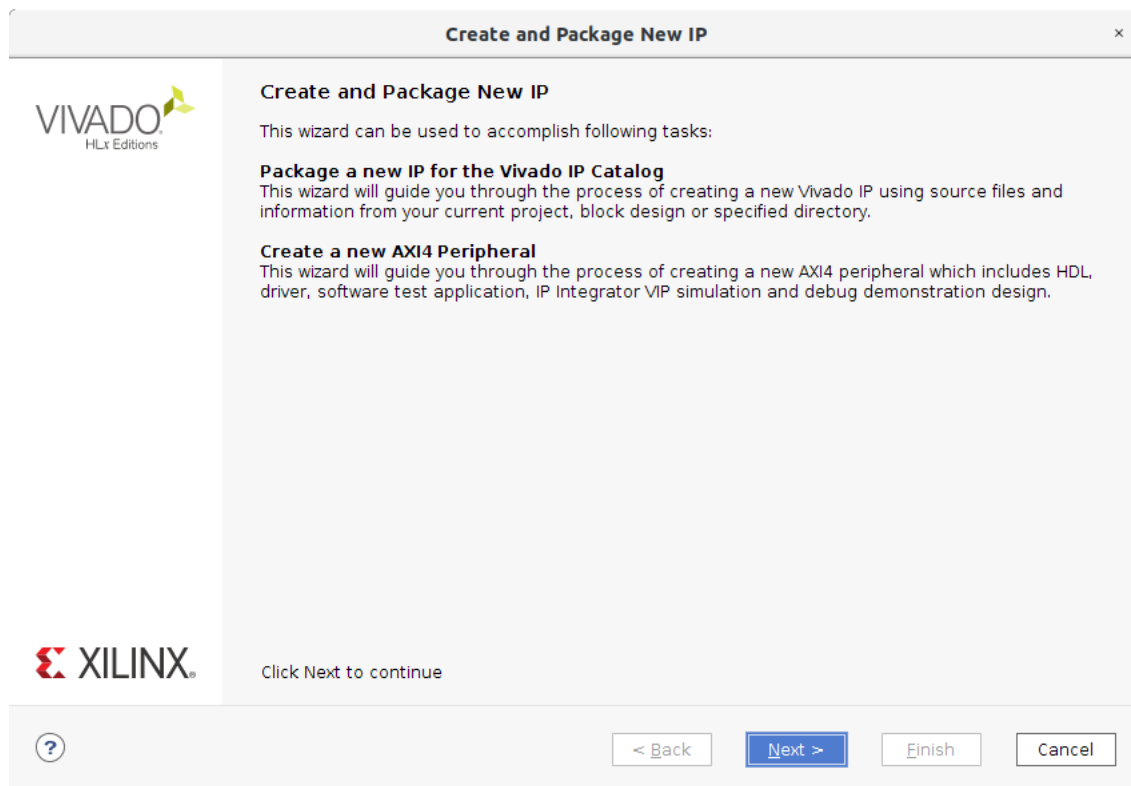


Figure 2: Finestra create and package new ip

2. Cliccare su “Next”, selezionare “**Create a New AXI4 peripheral**”, di nuovo “Next”

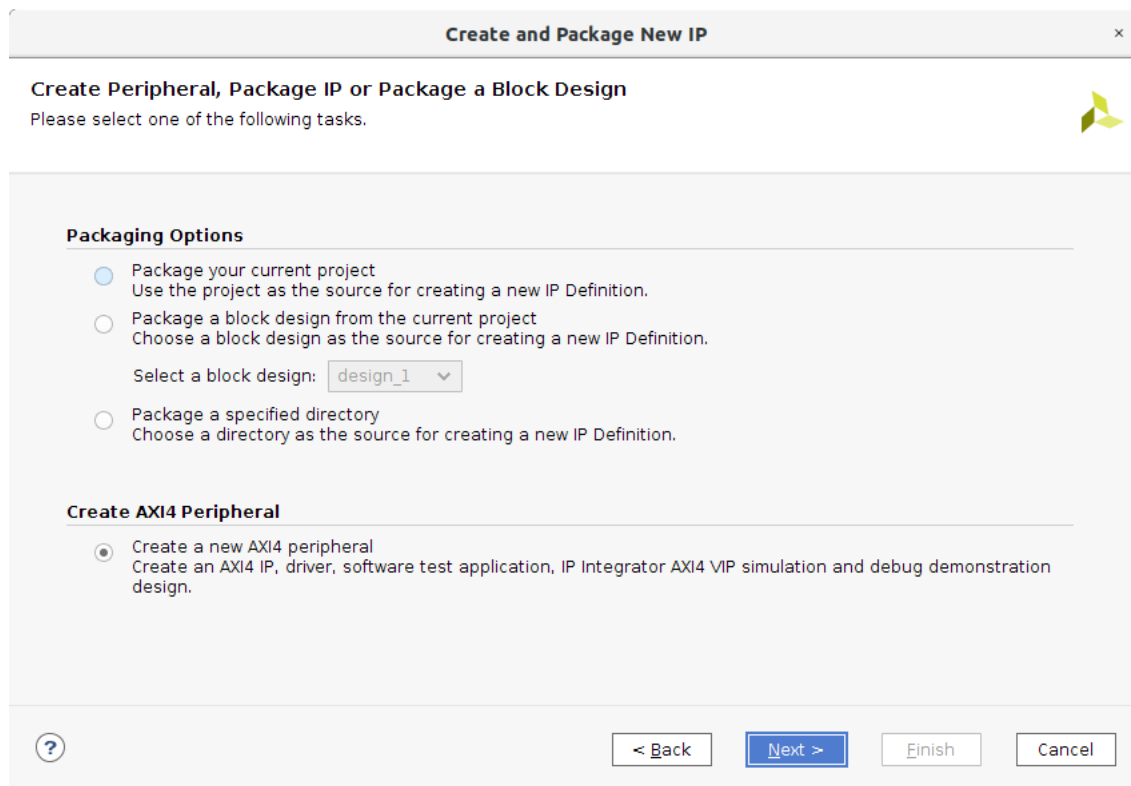


Figure 3: Finestra create new AXI Peripheral

3. Compilare i vari campi con le informazioni dell'IP. In questa finestra è importante il la directory "IP location" che permette di selezionare in quale cartella salveremo il nostro custom IP. Si consiglia di creare un'unica repository per tutti i custom ip. Dopo aver compilato tutti i campi, procedere cliccando su "Next".

The screenshot shows a software window titled "Create and Package New IP" with a close button (X) in the top right corner. Below the title bar, the tab "Peripheral Details" is selected, with a subtitle "Specify name, version and description for the new peripheral" and a small yellow logo to the right. The main area contains several input fields: "Name:" with the value "myip", "Version:" with "1.0", "Display name:" with "myip_v1.0", "Description:" with "My new AXI IP", and "IP location:" with "/home/saverio/ip_repo". Each text field has a small 'X' icon on its right side. The "IP location" field also has a file explorer icon (three dots) to its right. Below these fields is an unchecked checkbox labeled "Overwrite existing". At the bottom of the window, there is a help icon (question mark in a circle) on the left and four buttons: "< Back", "Next >" (highlighted in blue), "Finish", and "Cancel".

Figure 4:

4. Nella finestra successiva è possibile configurare il tipo di interfaccia del nostro componente. Nel nostro caso l'interfaccia da utilizzare è *AXI LITE* e il device deve essere uno Slave in quanto non gestisce le transizioni del bus AXI. Il parametro *number of register* indica il numero di registri (**SLAVE_REG**) che utilizzeremo per interfacciare il nostro componente alla CPU. Nel nostro caso i registri necessari sono tre: uno per pilotare il segnale di WRITE, uno per il segnale di ENABLE e uno per leggere il segnale READ. Si noti che il numero di registri minimo è 4 quindi uno rimarrà inutilizzato, il segnale PAD è un segnale di inout per il componente, non necessita di comunicare con il bus quindi non avrà un registro dedicato. Clicchiamo su "Next" una volta settati i parametri desiderati.

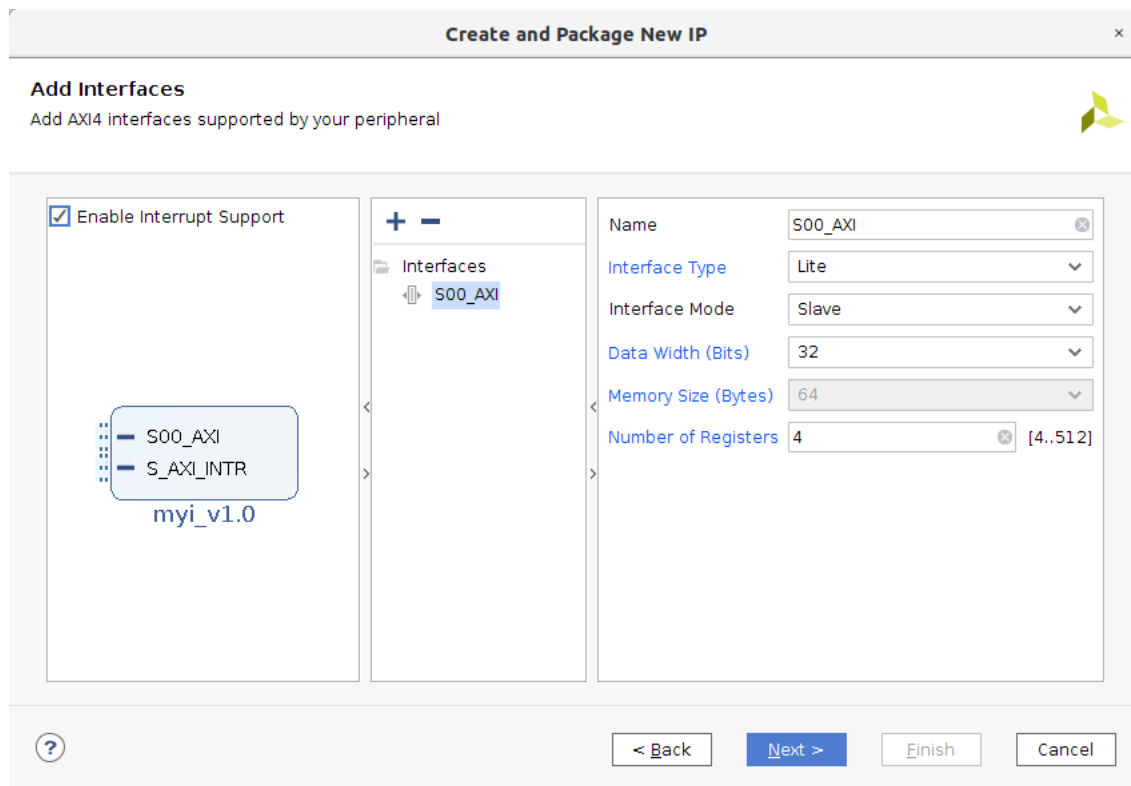


Figure 5:

5. Selezionare **“Edit IP”** e successivamente cliccare su **“Finish”**. Si aprirà una nuova istanza di Vivado in cui è possibile modificare il custom IP.

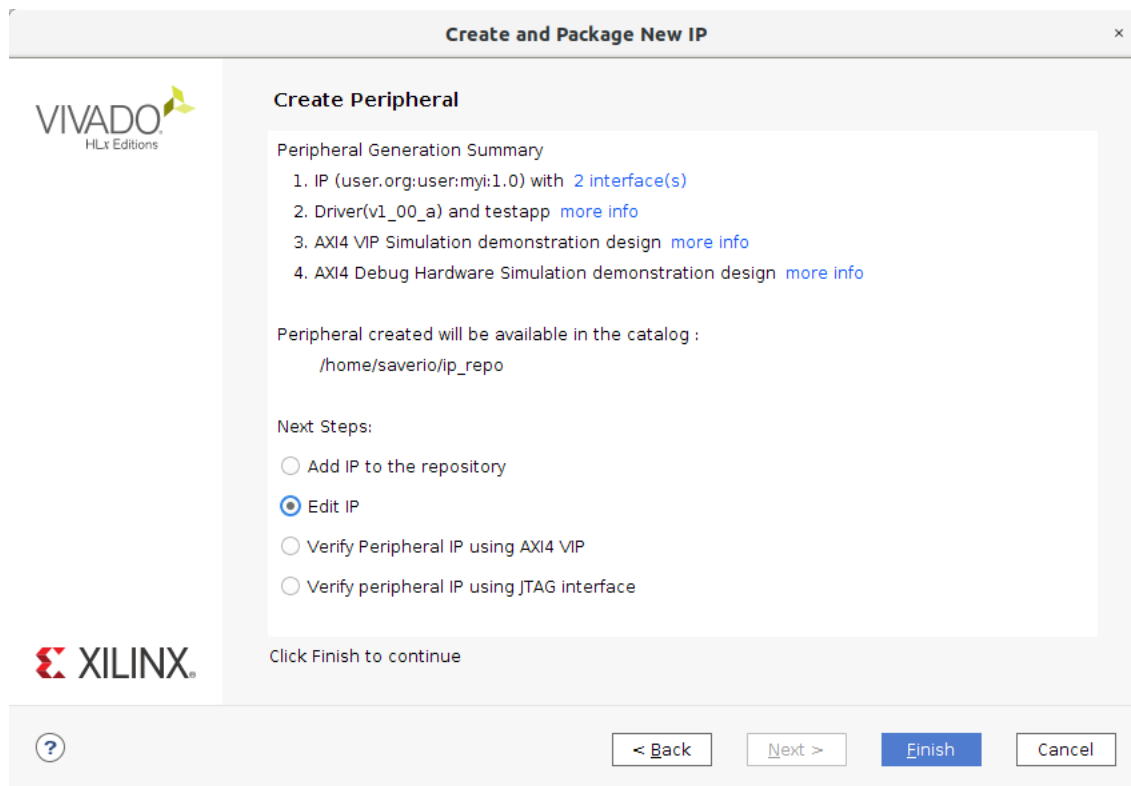


Figure 6:

Nella nuova finestra vediamo che sono stati creati due nuovi file, uno con il nome del nostro custom IP che rappresenta il top module e uno con nome “nomeip_S00_AXI”

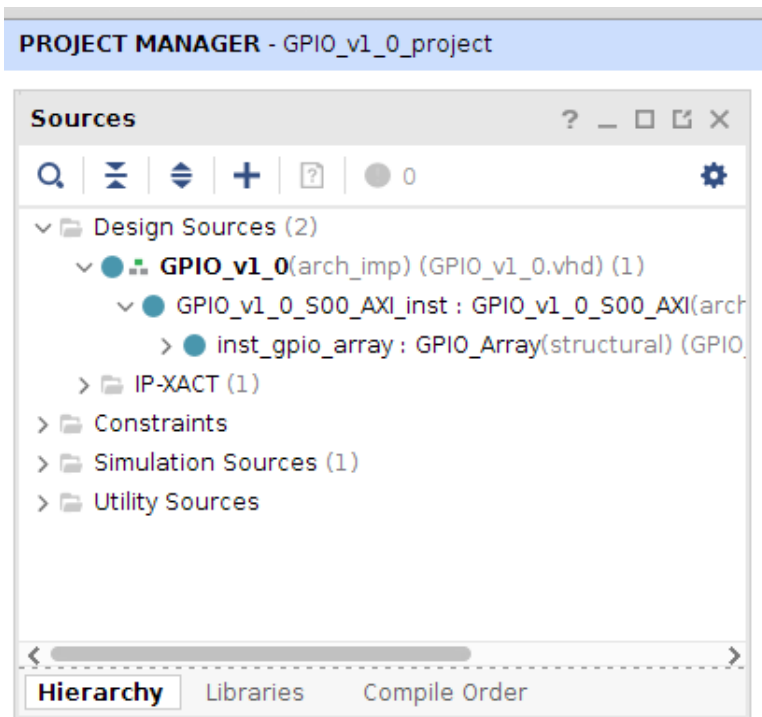


Figure 7:

0.1.3 Modifica Default IP Core

0.1.3.1 Istanziamento del componente GPIO_Array

Il componente istanziato da vivado è un wrapper che implementa l'interfacci AXI, all'interno del quale dovremo inserire il nostro GPIO_Array in modo da permettergli di comunicare sul bus. Uno schematico è riportato in nella seguente figura.

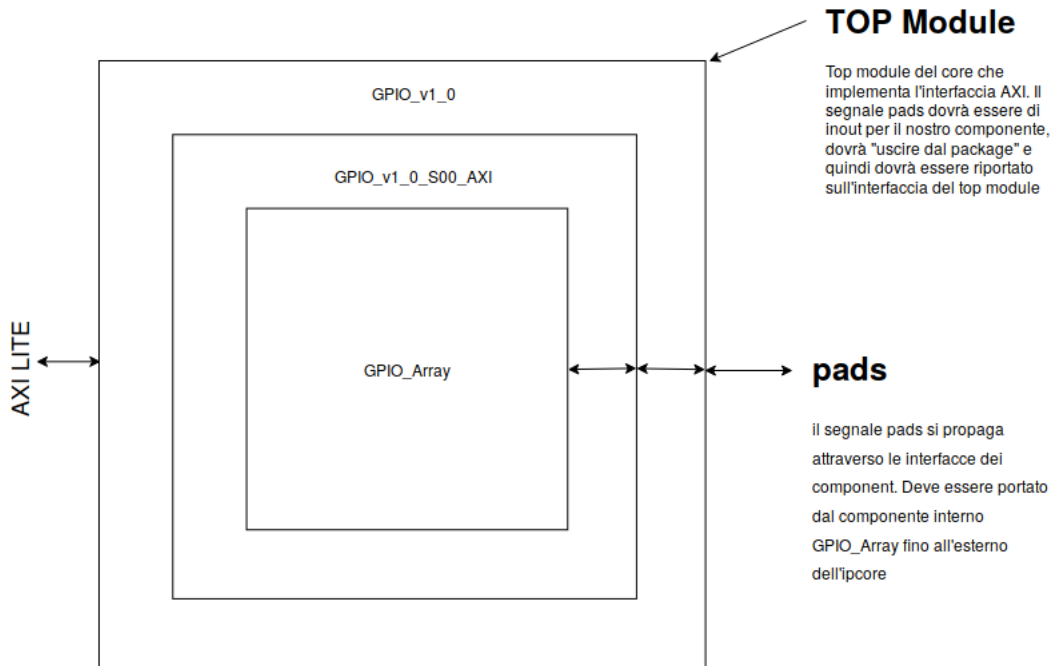


Figure 8: Schema IP

Aggiungiamo al design creato da Vivado il nostro sorgenti vhdl di componenti `GPIO_Array` ed il singolo `GPIO` cliccando su “Add Sources” presente sulla sinistra della interfaccia. Una volta fatto ciò instanziamo il nostro componente (`GPIO_Array`) nel modulo “`GPIO_v1_0_S00_AXI`”. E’ necessario farlo nella sezione indicata da Vivado tra i seguenti commenti

```
1  -- Add user logic here
2  -- User logic ends
```

Per prima cosa è necessario riportare fra le interfacce dei due componenti i segnali che necessitano di comunicare con l'esterno dell'ip e non con il bus axi. Il codice che va aggiunto nelle interfacce dei due componenti è il seguente:

```
1  -- Users to add ports here
2      pads : inout std_logic_vector(width-1 downto 0);
3      interrupt : out std_logic; --! segnale di interrupt
4  -- User ports ends
```

Aggiungiamo i parametri del nostro componente, in questo caso *width*, tra i seguenti commenti.

```
1  -- Users to add parameters here
2      width : integer := 4;
3  -- User parameters ends
```

Si noti che sarà necessario aggiungere i segnali e i parametri sopra riportati anche nell'istanziamento del componente `GPIO_v1_0_S00_AXI` nel top module.

Superata la prima fase è necessario interfacciare correttamente i segnali del nostro `GPIO_Array` con il bus e il processore. Per fare cioè il componente utilizza gli `slave_reg`, dei registri con i quali sarà possibile scrivere\leggere i valori dei segnali di `ENABLE`, `WRITE` e `READ`. Collegheremo dunque i segnali di `READ` e `WRITE` opportunamente nell'istanziamento del componente come segue:

```
1  gpio_inst : GPIO_Array
2  generic map(width => width)
3  port map(    enable => slv_reg0(width-1 downto 0),
4              write => slv_reg1(width-1 downto 0),
5              read  => gpio_read(width-1 downto 0),
```



```
6 | pads =>pads);
```

I segnali di ENABLE e di WRITE vengono pilotati dai registri *slv_reg0* ed *slv_reg1*. Il valore di READ viene salvato nel registro *gpio_read* perchè, per leggerlo, non può essere usato uno degli *slv_reg* generati automaticamente da vivado in quanto READ, essendo un pin di output per il GPIO, forzerebbe dei valori sullo slave reg. Anche il bus axi, tramite un process detto “di scrittura”, forza dei valori sugli slave reg che quindi si ritroverebbero pilotati da due segnali contemporaneamente. Per eliminare questo conflitto viene introdotto un nuovo segnale *gpio_read* che verrà pilotato esclusivamente dal read del nostro GPIO_Array (Figura 0.9).

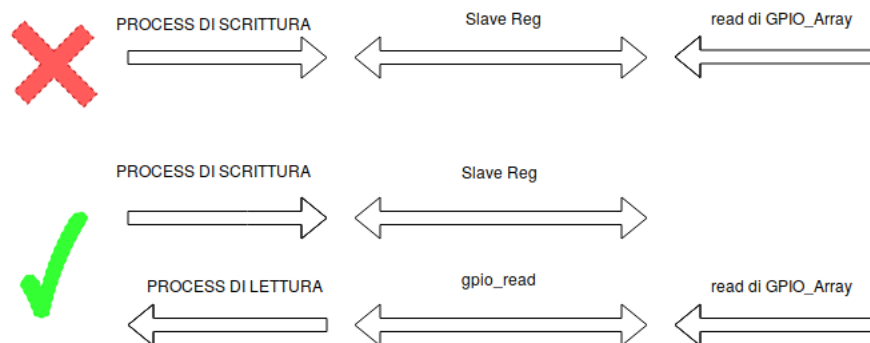


Figure 9: Corretta gestione dei segnali di output del componente GPIO_Array

Segue il codice del process “di lettura” modificato opportunamente per leggere il valore di READ:

```
1 process (slv_reg0, slv_reg1, gpio_read, slv_reg3, slv_reg4, slv_reg5, status_reg_out,
2   slv_reg7_out, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
3   variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
4   begin
5     -- Address decoding for reading registers
6     loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
7     case loc_addr is
8       when b"000" =>
9         reg_data_out <= slv_reg0;
10      when b"001" =>
11        reg_data_out <= slv_reg1;
12      when b"010" =>
13        reg_data_out <= gpio_read;
14      when b"011" =>
15        reg_data_out <= slv_reg3;
16      when b"100" =>
17        reg_data_out <= slv_reg4;
18      when b"101" =>
19        reg_data_out <= slv_reg5;
20      when b"110" =>
21        reg_data_out <= status_reg_out;
22      when b"111" =>
23        reg_data_out <= slv_reg7_out;
24      when others =>
25        reg_data_out <= (others => '0');
26    end case;
27  end process;
```

In questo caso invece di leggere lo *slv_reg2* verrà letto il segnale *gpio_read*. Si noti che questo procedimento va applicato per tutti i segnali di out del componente istanziato.

0.1.3.2 Gestione delle interruzioni

Segue la logica creata per permettere all'ip di lavorare sotto interruzioni. Siamo interessati a generare un evento di interruzione ogni qual volta vi sia una variazione del segnale di READ del GPIO_Array. La variazione deve asserire il segnale di interrupt se e solo se:

- Le interruzioni globali del componente sono abilitate
- La singola linea interna del GPIO_Array è abilitata (*mascherata*) a generare l'interruzione
- Il segnale di READ è pilotato da PADS e non da WRITE

Per catturare le variazioni del segnale READ è stato utilizzato il seguente process:

```
1  gpio_read_sampling : process (S_AXI_ACLK, gpio_read)
2  begin
3  if (rising_edge (S_AXI_ACLK)) then
4      if ( S_AXI_ARESETN = '0' ) then
5          last_stage <= (others => '0');
6          current_stage <= (others => '0');
7      else
8          last_stage <= gpio_read(width-1 downto 0);
9          current_stage <= last_stage;
10     end if;
11 end if;
12 end process;
13
14 changed_bits <= (last_stage xor current_stage) and intr_mask and (not gpio_enable);  --!
15     indica quale bit,                                     se mascherato e pilotato da pads,
16                                                         è cambiato
17 change_detected <= global_intr and (or_reduce(changed_bits));  --! il segnale
18     assume valore 1 se e solo se                             le interruzioni globali sono abilitate e
19                                                         c'è un cambiamento del segnale READ
```

Si noti che siamo interessati a qualunque variazione del segnale read con le relazioni sopra elencate.

Il process viene sintetizzato da vivado come due Flip-Flop collegati in cascata, necessari per campionare il valore di gpio_read con il clock. Al primo colpo di clock il primo flip flop salva il valore del segnale, al secondo colpo di clock il secondo flip flop conterrà il valore precedentemente salvato nel primo. I due segnali sono posti in xor per vedere se ci sono variazioni. Segue uno schematico della soluzione realizzata (non considerare Q negato):

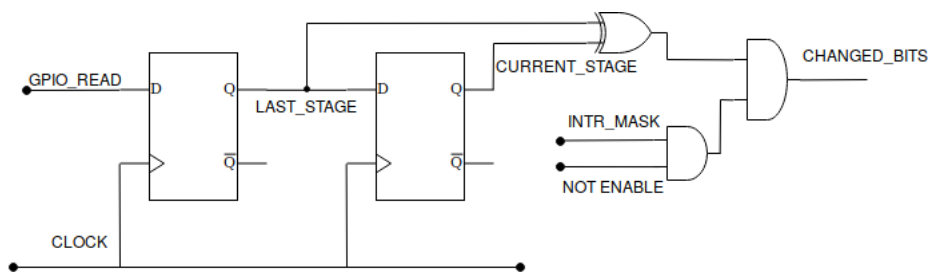


Figure 10:

Un secondo process viene utilizzato per la gestione del registro delle interruzioni pendenti (*pending_intr*):

```
1  pending_intr_tmp <= pending_intr;
2
3  intr_pending : process (S_AXI_ACLK, change_detected, ack_intr)
4  begin
5      if (rising_edge (S_AXI_ACLK)) then
```

```

6         if (change_detected = '1') then
7             pending_intr <= pending_intr_tmp or changed_bits;
8         elsif (or_reduce(ack_intr)='1') then
9             pending_intr <= pending_intr_tmp and (not ack_intr);
10    else
11        pending_intr <= pending_intr_tmp;
12    end if;
13 end if;
14 end process;

```

Il valore del registro pending è dato da il valore di una precedente interruzione non servita oppure ad una interruzione rilevata (changed_bits). Il registro contiene un 1 nella posizione relativa al GPIO che chiede di generare un interrupt. Nello stesso process è gestito anche il meccanismo di ack utilizzato per pulire il relativo bit ne registro pending. Il segnale ack_intr deve contenere un 1 nella posizione relativa al gpio al quale si vuol dare l'ack. Nel caso arrivi un segnale di ack bisogna cambiare il bit relativo al registro pending della interruzione servita. L' ultimo process infine occorre per la generazione del segnale di interrupt che va portato all'esterno dell'ip:

```

1 inst_irq : process(S_AXI_ACLK,pending_intr)
2     begin
3         if (rising_edge (S_AXI_ACLK)) then
4             if ( S_AXI_ARESETN = '0' ) then
5                 interrupt <= '0';
6             else
7                 if (or_reduce(pending_intr) = '1' and global_intr = '1') then
8                     interrupt <= '1';
9                 else
10                    interrupt <= '0';
11                end if;
12            end if;
13        end if;
14    end process;

```

Tale segnale è altro nel caso vi siano interruzioni pendenti e le interruzioni globali siano abilitate, nel caso di reset del bus o non vi siano interruzioni o le stesse siano disabilitate è pari a 0.

Dopo aver instaziato il componente nella top level entity avente il nome del nostro custom IP possiamo procedere con l' impacchettamento del nostro componente.

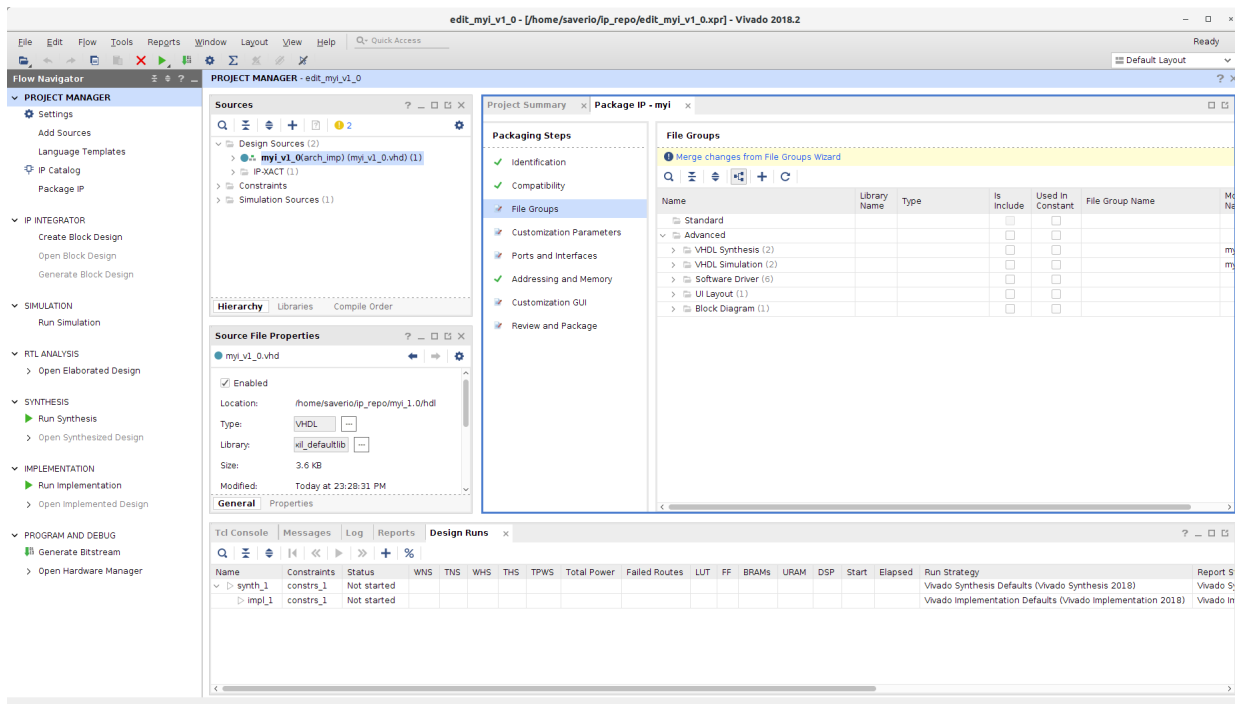


Figure 11:

Nella sezione “**File Groups**” cliccare su “**Merge changes from File Groups Wizard**”, in “**Customization Parameters**” su “**Merge changes from Customization Parameters Wizard**”, selezionare “**Hidden Parameters**” si aprirà tale finestra

Figure 12:

Da qui è possibile rendere visibile questo parametro da configurare cliccando sul box “**Visible in Customization GUI**” e settarne i valori che può assumere cliccando su “**Specify Range**”.

Recarsi infine su “**Review and Package**” cliccare su “**Re-Package IP**” per ottenere il custom ip, facendo chiudere la finestra di vivado.

0.1.4 Creazione del block design

Istanziamo 3 IP ognuno con dimensione 4. Serviranno a controllare rispettivamente switch, led e button. Cliccare sul menù di sinistra “**Create block design**” inserire i parametri desiderati e cliccare su “OK” verrà mostrato questo workspace

Cliccare sul pulsante + ed inserire il custom IP creato insieme al processore ZYNQ, appariranno due pulsanti “**Run Block Automation**” e “**Run Connection Automation**” che cliccati fanno sì che i vari componenti si collegheranno tra di loro automaticamente. Successivamente è necessario rendere esterni i pin che si vuole pilotare dalla board. Cliccare con il tasto destro del mouse su un pin e selezionare “**Make external**”, successivamente selezionare l’icona contrassegnata da un simbolo di spunta. Il risultato è il seguente:

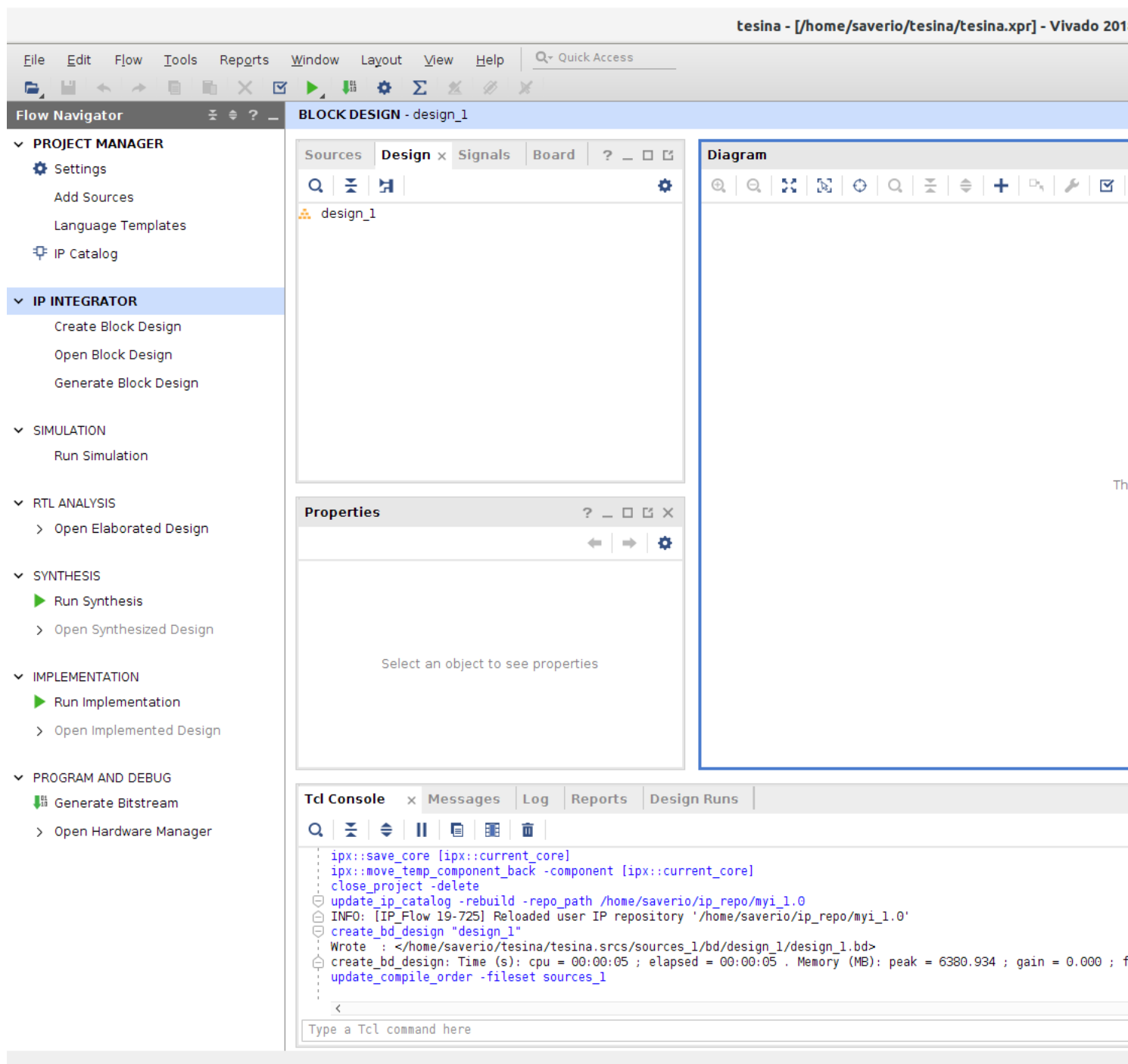


Figure 13:

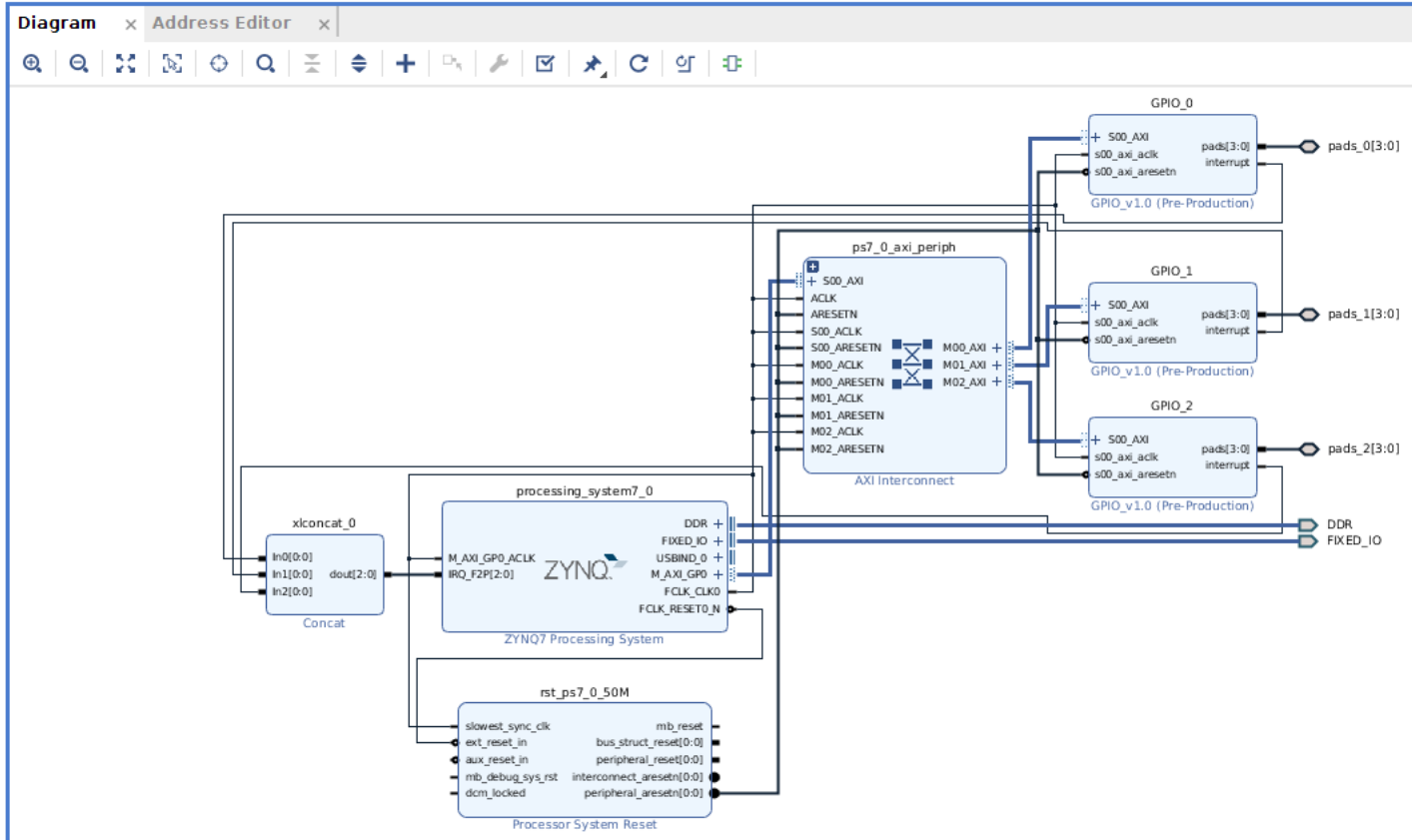


Figure 14: Block design completo

Per permettere alla PL di interrompere la PS è necessario abilitare le interruzioni dello zynq processing system. Cliccare due volte sul componente e recarsi nella sezione Interrupt e abilitare “**PL-PS interrupt port**”.

La linea di interrupt che apparirà sul processing system è unica, quindi è necessario utilizzare il componente Concat per collegare i 3 GPIO_Array. Il size verrà automaticamente aggiornato.

Bisogna generare ora un *wrapper HDL* affinché il block design sia sintetizzabile. Recandosi nella sezione “**Sources**”, tasto destro sul nome del block design e selezionare “**Create HDL Wrapper...**” cliccando “OK” ed essendo sicuri che sia selezionata l’opzione “Let Vivado manage wrapper and auto-update” fatto ciò procedere alla sintesi.

Una volta terminata, selezionare dal sottomenù “Open Synthesized Design” e da un menù a tendina in alto a destra “**I/O Planning**” verrà mostrata la seguente schermata





ZYNQ7 Processing System (5.5)

 Documentation  Presets  IP Location  Import XPS Settings

Page Navigator

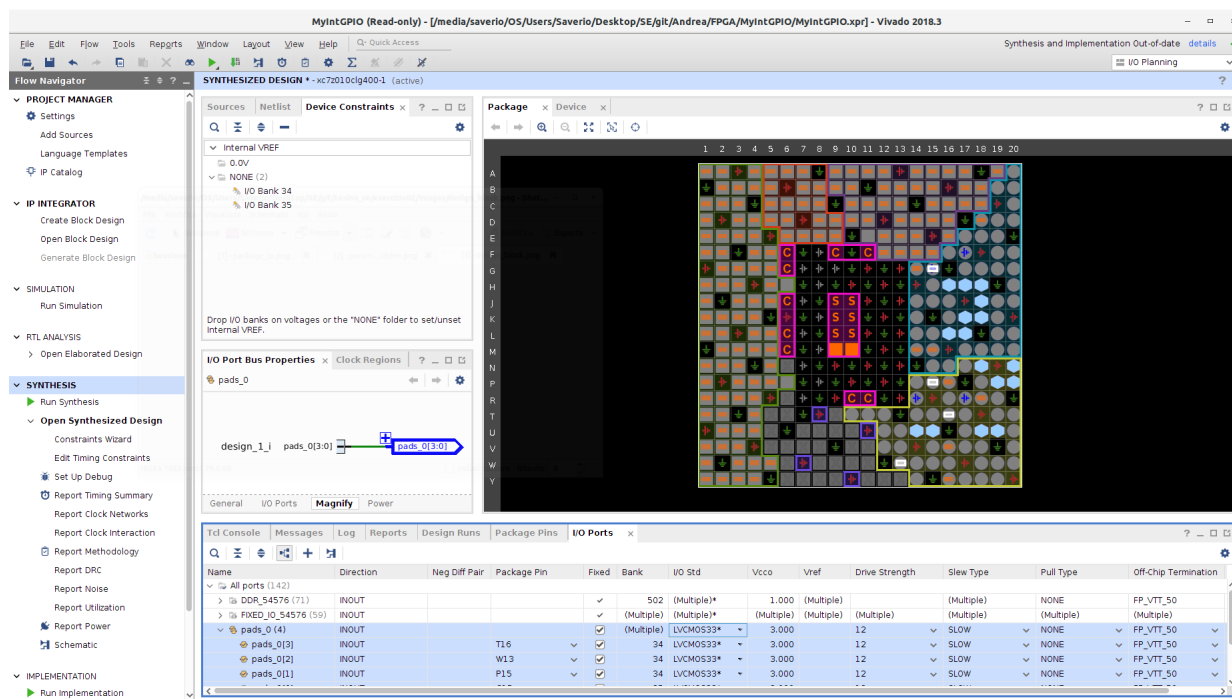
- Zynq Block Design
- PS-PL Configuration
- Peripheral I/O Pins
- MIO Configuration
- Clock Configuration
- DDR Configuration
- SMC Timing Calculation
- Interrupts**

Interrupts

Search:

Interrupt Port	ID	Description
<input checked="" type="checkbox"/> Fabric Interrupts		Enable
▼ PL-PS Interrupt Ports		
<input checked="" type="checkbox"/> IRQ_F2P[15:0]	[91:84],...	Enable
<input type="checkbox"/> Core0_nFIQ	28	Enable
<input type="checkbox"/> Core0_nIRQ	31	Enable
<input type="checkbox"/> Core1_nFIQ	28	Enable
<input type="checkbox"/> Core1_nIRQ	31	Enable
> PS-PL Interrupt Ports		



In basso sono presenti i pin che sono stati resti esterni, bisogna selezionare il tip di “I/O Std” di solito è quello mostrato in figura ed i “Package Pin” una volta fatto salvare i constraints cliccando sull’ icone del floppy blu e dando un nome al file di constraint, una volta fatto si può generare il bitstream.

Terminato il processo possiamo esportarlo dal menù “**File -> Export -> Export Hardware...**” assicurarsi che sia spuntata l’opzione “***Include bitstream***” e selezionare “OK”, si può ora procedere alla creazione del driver linux, lanciando SDK dal menù “File->Launch SDK”.

0.1.5 Driver Standalone

Il driver ora presentato, viene eseguito direttamente dalla sezione PS della board senza il supporto di un sistema operativo.

Per poter procedere alla scrittura del driver, dobbiamo sapere dove i registri del nostro componente hardware sono stati mappati, tale informazione può essere reperita dal file “*xparameters.h*” presente nella directory “**cartella_del_board_s** troveremo una sezione di codice simile alla seguente

```

1  /* Definitions for driver MYINTGPIO */
2  #define XPAR_MYINTGPIO_NUM_INSTANCES 3
3  /* Definitions for peripheral MYINTGPIO_0 */
4  #define XPAR_MYINTGPIO_0_DEVICE_ID 0
5  #define XPAR_MYINTGPIO_0_S00_AXI_BASEADDR 0x43C00000
6  #define XPAR_MYINTGPIO_0_S00_AXI_HIGHADDR 0x43C0FFFF
7  /* Definitions for peripheral MYINTGPIO_1 */
8  #define XPAR_MYINTGPIO_1_DEVICE_ID 1
9  #define XPAR_MYINTGPIO_1_S00_AXI_BASEADDR 0x43C20000
10 #define XPAR_MYINTGPIO_1_S00_AXI_HIGHADDR 0x43C2FFFF
11 /* Definitions for peripheral MYINTGPIO_2 */
12 #define XPAR_MYINTGPIO_2_DEVICE_ID 2
13 #define XPAR_MYINTGPIO_2_S00_AXI_BASEADDR 0x43C40000
14 #define XPAR_MYINTGPIO_2_S00_AXI_HIGHADDR 0x43C4FFFF

```

dove è possibile sapere il numero dei nostri custom IP core istanziati nel progetto HW il loro indirizzo base e quello più alto associato alla periferica.

Oltre a tale informazione dobbiamo anche conoscere di quanto sono stati spiazzati i nostri `slv_reg` rispetto all' indirizzo base, tale informazione reperibile nel file situato in `"cartella_del_board_support_package\ps7_cortexa9_0\libsrc\'nome_neql` quale, una volta aperto, sarà presente una sezione simile:

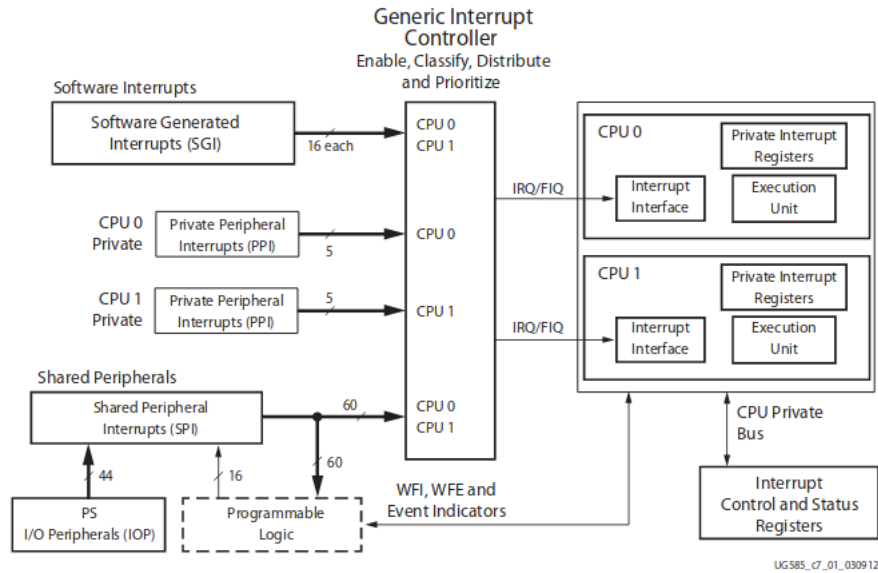


Figure 17:

```

1 #define MYINTGPIO_S00_AXI_SLV_REG0_OFFSET 0
2 #define MYINTGPIO_S00_AXI_SLV_REG1_OFFSET 4
3 #define MYINTGPIO_S00_AXI_SLV_REG2_OFFSET 8
4 #define MYINTGPIO_S00_AXI_SLV_REG3_OFFSET 12

```

Ottenute queste informazioni possiamo leggere e scrivere valori nei registri. Per gli indirizzi base degli ip si consulti l'Address Editor di Vivado.

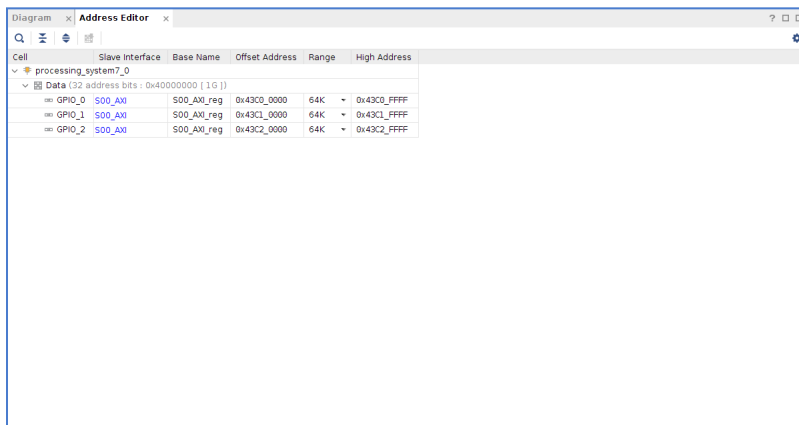


Figure 16:

Possiamo procedere alla creazione del driver, instanziamo un nuovo progetto vuoto dal menù “**File -> New -> Application Project**” e creare un progetto vuoto. La PS della Zynq 7000 è composta da *Cortex-A9* e un *GIC pl390 interrupt controller*. Si mostano per completezza alcune nozioni sul GIC, che non avendo un SO, andrà gestito direttamente. Si mostra uno schema generico del GIC.

L'interruzione che abbiamo bisogno di gestire sono interruzioni provenienti dalla PL e quindi visti dal GIC come interruzioni SPI. Sarebbe possibile indicare, per ogni interruzione, quale dei due Cortex deve gestirla, ma lasciamo questa scelta al GIC. Ogni linea di interrupt è identificata da un ID unico. Per il funzionamento interno riferirsi a **ug585-Zynq-7000-TRM**.

Noi ci interfaceremo solo con il modello di programmazione del GIC utilizzando i driver offerti da Xilinx nella libreria *scugic*.

Il workflow da eseguire per configurare il device è il seguente:

1. Configurare il GIC.

```
1  int Status;
2  /* Istanza del Gic */
3  XScuGic InterruptController;
4
5  /*Istanza della configurazione del Gic*/
6  XScuGic_Config *GicConfig;
7
8  GicConfig = XScuGic_LookupConfig(INTC_DEVICE_ID);
9
10 Status = XScuGic_CfgInitialize(&InterruptController,GicConfig, GicConfig->
    CpuBaseAddress);
11 if ( Status != XST_SUCCESS) return XST_FAILURE;
```

Questa fase di inizializzazione e fissa.

2. Abilitare la gestione delle eccezioni relative al Gic (opzionale ma utile)

```
1  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
2      (Xil_ExceptionHandler)XScuGic_InterruptHandler,&InterruptController);
3  Xil_ExceptionEnable();
```

Questa fase di inizializzazione e fissa.

3. Associare alle 3 le linee di interruzione i relativi handler

```
1  Status = XScuGic_Connect(&InterruptController,XPAR_FABRIC_GPIO_0_INTERRUPT_INTR,
2      (Xil_ExceptionHandler)SwitchISR,(void *)&InterruptController);
3  if ( Status != XST_SUCCESS) return XST_FAILURE;
4
5  Status = XScuGic_Connect(&InterruptController,XPAR_FABRIC_GPIO_1_INTERRUPT_INTR,
6      (Xil_ExceptionHandler)ButtonISR,(void *)&InterruptController);
7  if ( Status != XST_SUCCESS) return XST_FAILURE;
8
9  Status = XScuGic_Connect(&InterruptController,XPAR_FABRIC_GPIO_2_INTERRUPT_INTR,
10     (Xil_ExceptionHandler)LedISR,(void *)&InterruptController);
11 if ( Status != XST_SUCCESS) return XST_FAILURE;
```

La define **XPAR_FABRIC_GPIO_0_INTERRUPT_INTR** indica l'ID della linea di interruzione al quale è collegato l'ip GPIO_0. E' possibile trovarla del file "*xparameters.h*". Il parametro SwitchISR è il nome dell'handler.

4. Enable della linea di interruzione

```
1  XScuGic_Enable(&InterruptController,XPAR_FABRIC_GPIO_0_INTERRUPT_INTR);
2  XScuGic_Enable(&InterruptController,XPAR_FABRIC_GPIO_1_INTERRUPT_INTR);
3  XScuGic_Enable(&InterruptController,XPAR_FABRIC_GPIO_2_INTERRUPT_INTR);
```

Si mostra di seguito una routine esempio di gestione dell' interruzione che:

1. Disabilita le interruzioni globali dell'ip.
2. Verifica quale delle linee chiedono di essere servite;
3. Da l'ack alle linee pendenti
4. Riabilita le interruzioni globali del componente;

```

1 void SwitchISR() {
2
3     XGPIO_GlobalDisableInterrupt (&GPIO_Switch, 0x01);
4     InterruptProcessed = TRUE;
5     print ("\n\n*****ISR SWITCH*****\n\n");
6     uint8_t pendingReg = XGPIO_GetPending (&GPIO_Switch);
7     XGPIO_ACK (&GPIO_Switch, pendingReg);
8     XGPIO_GlobalEnableInterrupt (&GPIO_Switch, 0x01);
9 }

```

0.1.6 Driver Linux

La seguente sezione mostra come scrivere un driver con il supporto di un sistema operativo Linux. In questo ambiente è possibile scrivere un driver come un modulo kernel oppure utilizzando l'Userspace I/O (UIO). In entrambi i casi la prima operazione da effettuare è quella di creare un First Stage Boot Loader e un Device-Tree come mostrato nel primo capitolo.

0.1.6.1 Kernel Mode

Un driver può essere scritto sottoforma di modulo kernel e poi caricato dinamicamente. Questa pratica fornisce più flessibilità rispetto al "build statico" di un modulo all'interno del kernel, in quanto potrebbe risultare non necessario inserire moduli che poi non verranno utilizzati. Per l'astrazione del nostro device GPIO si è realizzata una struct, definita nel file GPIO.h, che contiene tutte le informazioni necessarie per la gestione del dispositivo.

```

1 /**
2  * @brief Struttura che astrae un device GPIO in kernel-mode.
3  * Contiene ciò che è necessario al funzionamento del driver.
4  */
5 typedef struct {
6     /** Major e minor number associati al device (M: identifica il driver associato al device; m
7      : utilizzato dal driver per discriminare il singolo device tra quelli a lui associati)*/
8     dev_t Mm;
9     /** Puntatore a struttura platform_device cui l'oggetto GPIO si riferisce */
10    struct platform_device *pdev;
11    /** Struttura per l'astrazione di un device a caratteri */
12    struct cdev cdev;
13    /** Puntatore alla struttura che rappresenta l'istanza del device*/
14    struct device* dev;
15    /** Puntatore a struttura che rappresenta una vista alto livello del device*/
16    struct class* class;
17    /** Interrupt-number a cui il device è connesso*/
18    uint32_t irqNumber;
19    /** Puntatore alla regione di memoria cui il device è mappato*/
20    struct resource *mreg;
21    /** Device Resource Structure*/
22    struct resource res;
23    /** Maschera delle interruzioni interne attive per il device*/
24    uint32_t irq_mask;
25    /** res.end - res.start; numero di indirizzi associati alla periferica.*/
26    uint32_t res_size;
27    /** Indirizzo base virtuale della periferica*/
28    void __iomem *vrtl_addr;
29    /** wait queue per la sys-call read() */
30    wait_queue_head_t read_queue;
31    /** wait queue per la sys-call poll()*/
32    wait_queue_head_t poll_queue;
33    /** Flag che indica, quando asserito, la possibilità di effettuare una chiamata a read*/
34    uint32_t can_read;

```

```

34  /** Spinlock usato per garantire l'accesso in mutua esclusione alla variabile can_read*/
35      spinlock_t slock_int;
36  } GPIO;

```

Per le funzioni necessarie all'interfacciamento con il device si rimanda alla documentazione interna. Il device è stato gestito come un **character device**.

All'inserimento del modulo viene chiamata la funzione **Probe**, la quale si occupa dell'inizializzazione del driver chiamando la funzione `GPIO_Init()` del device da registrare. Questa deve occuparsi dunque di tutte le operazioni necessarie alla registrazione e all'inserimento di un dispositivo a caratteri all'interno del sistema:

```

1  /**
2   * @brief Inizializza una struttura GPIO per il corrispondente device
3   *
4   * @param   GPIO_device puntatore a struttura GPIO, corrispondente al device su cui operare
5   * @param   owner puntatore a struttura struct module, proprietario del device (THIS_MODULE)
6   * @param   pdev puntatore a struct platform_device
7   * @param   driver_name nome del driver
8   * @param   device_name nome del device
9   * @param   serial numero seriale del device
10  * @param   f_ops puntatore a struttura struct file_operations, specifica le funzioni che
11             agiscono sul device
12  * @param   irq_handler puntatore irq_handler_t alla funzione che gestisce gli interrupt
13             generati dal device
14  * @param   irq_mask maschera delle interruzioni attive del device
15  *
16  * @retval "0" se non si è verificato nessun errore
17  *
18  * @details
19  */
20  int GPIO_Init(    GPIO* GPIO_device,
21                  struct module *owner,
22                  struct platform_device *pdev,
23                  struct class* class,
24                  const char* driver_name,
25                  const char* device_name,
26                  uint32_t serial,
27                  struct file_operations *f_ops,
28                  irq_handler_t irq_handler,
29                  uint32_t irq_mask) {
30
31      int error = 0;
32      struct device *dev = NULL;
33
34      char *file_name = kmalloc(strlen(driver_name) + 5, GFP_KERNEL);
35      sprintf(file_name, device_name, serial);
36
37      GPIO_device->pdev = pdev;
38      GPIO_device->class = class;
39
40      /** Alloca un range di Mj e min numbers per il device a caratteri */
41
42      if ((error = alloc_chrdev_region(&GPIO_device->Mm, 0 , 1, file_name)) != 0) {
43          printk(KERN_ERR "%s: alloc_chrdev_region() ha restituito %d\n", __func__, error);
44          return error;
45      }
46
47      /** Inizializza la struttura cdev specificando la struttura file operations associata al
48          device a caratteri */
49
50      cdev_init (&GPIO_device->cdev, f_ops);
51      GPIO_device->cdev.owner = owner;

```

```

49  /** Crea il device all'interno del filesystem assegnandogli i numbers richiesti in
    precedenza e ne restituisce il puntatore. */
50
51  if ((GPIO_device->dev = device_create(class, NULL, GPIO_device->Mm, NULL, file_name)) ==
    NULL) {
52      printk(KERN_ERR "%s: device_create() ha restituito NULL\n", __func__);
53      error = -ENOMEM;
54      goto device_create_error;
55  }
56
57  /** Aggiunge il device a caratteri al sistema. Se l'operazione va a buon fine sarà possibile
    vedere il device sotto /dev */
58
59
60  if ((error = cdev_add(&GPIO_device->cdev, GPIO_device->Mm, 1)) != 0) {
61      printk(KERN_ERR "%s: cdev_add() ha restituito %d\n", __func__, error);
62      goto cdev_add_error;
63  }
64
65  /** Inizializza la struct resource con il valori recuperati dal device tree corrispondente
    al device */
66
67  dev = &pdev->dev;
68  if ((error = of_address_to_resource(dev->of_node, 0, &GPIO_device->res)) != 0) {
69      printk(KERN_ERR "%s: address_to_resource() ha restituito %d\n", __func__, error);
70      goto of_address_to_resource_error;
71  }
72  GPIO_device->res_size = GPIO_device->res.end - GPIO_device->res.start + 1;
73
74  /** Alloca una quantita res_size di memoria fisica per il dispositivo IO a partire dall'
    indirizzo res.start e ne restituisce l'indirizzo */
75
76  if ((GPIO_device->mreg = request_mem_region(GPIO_device->res.start, GPIO_device->res_size,
    file_name)) == NULL) {
77      printk(KERN_ERR "%s: request_mem_region() ha restituito NULL\n", __func__);
78      error = -ENOMEM;
79      goto request_mem_region_error;
80  }
81
82  /** Mappa la memoria fisca allocata e restituisce l'indirizzo virtuale */
83
84  if ((GPIO_device->vrtl_addr = ioremap(GPIO_device->res.start, GPIO_device->res_size)) ==
    NULL) {
85      printk(KERN_ERR "%s: ioremap() ha restituito NULL\n", __func__);
86      error = -ENOMEM;
87      goto ioremap_error;
88  }
89
90  /** Cerca le specifiche dell'interrupt nel device tree e restituisce il suo numero
    identificativo */
91
92  GPIO_device->irqNumber = irq_of_parse_and_map(dev->of_node, 0);
93  if ((error = request_irq(GPIO_device->irqNumber, irq_handler, 0, file_name, NULL)) != 0)
    {
94      printk(KERN_ERR "%s: request_irq() ha restituito %d\n", __func__, error);
95      goto irq_of_parse_and_map_error;
96  }
97  GPIO_device->irq_mask = irq_mask;
98
99

```

```

100
101 /** Inizializzazione della wait-queue per la system-call read() e poll() */
102
103     init_waitqueue_head(&GPIO_device->read_queue);
104     init_waitqueue_head(&GPIO_device->poll_queue);
105
106 /** Inizializzazione degli spinlock */
107
108     spin_lock_init(&GPIO_device->slock_int);
109     GPIO_device->can_read = 0;
110 /** Abilitazione degli interrupt del device */
111
112     GPIO_GlobalInterruptEnable(GPIO_device);
113     GPIO_PinInterruptEnable(GPIO_device, GPIO_device->irq_mask);
114     goto no_error;
115
116 irq_of_parse_and_map_error:
117     iounmap(GPIO_device->vrtl_addr);
118 ioremap_error:
119     release_mem_region(GPIO_device->res.start, GPIO_device->res_size);
120 request_mem_region_error:
121 of_address_to_resource_error:
122 cdev_add_error:
123     device_destroy(GPIO_device->class, GPIO_device->Mm);
124 device_create_error:
125     cdev_del(&GPIO_device->cdev);
126     unregister_chrdev_region(GPIO_device->Mm, 1);
127
128 no_error:
129
130     printk(KERN_INFO " IRQ registered as %d\n", GPIO_device->irqNumber);
131     printk(KERN_INFO " Driver succesfully probed at Virtual Address 0x%08lx\n", (unsigned long
132         ) GPIO_device->vrtl_addr);
133
134     return error;
135 }

```

Quando invece il modulo viene rimosso viene chiamata la funzione **Remove**, la quale effettua le operazioni inverse chiamando la funzione **GPIO_Destroy**. Sia la Probe che la Remove devono essere ridefinite all'interno del modulo e si utilizza la struttura **platform_driver** per effettuare il matching.

```

1 /**
2  * @brief Definisce le funzioni probe() e remove() da chiamare al caricamento del driver.
3  */
4 static struct platform_driver GPIO_driver = {
5     .driver = {
6         .name = DRIVER_NAME,
7         .owner = THIS_MODULE,
8         .of_match_table = of_match_ptr(__test_int_driver_id),
9     },
10    .probe = GPIO_probe,
11    .remove = GPIO_remove
12 };

```

La funzione **of_match_ptr(__test_int_driver_id)** si occupa di effettuare il matching con i device contenuti all'interno del device-tree. Per ogni device contenente un campo compatible uguale a quello specificato mediante la struttura **of_device_id** verrà chiamata la funzione di Probe per far sì che il driver possa gestire quel device.

```

1 /**
2  * @brief Identifica il device all'interno del device tree
3  */

```

```

4  */
5  static const struct of_device_id __test_int_driver_id[]={
6      {.compatible = "GPIO"},
7      {}
8  };

```

Dato che un driver può gestire più di un singolo device GPIO è stato implementato un meccanismo di gestione tramite lista. La Probe dunque inizializza il corrispondente device GPIO e lo inserisce all'interno della lista, se questa non contiene già il numero massimo consentito di dispositivi. Il device all'interno del sistema operativo Linux è visto come un file, per cui il device driver deve implementare tutte le system-call per l'interfacciamento con un file. La corrispondenza tra queste e la relativa funzione fornita dal driver viene stabilita attraverso la struttura `file_operations`.

```

1  /**
2   * @brief Struttura che specifica le funzioni che agiscono sul device
3   *
4   */
5  static struct file_operations GPIO_fops = {
6      .owner      = THIS_MODULE,
7      .llseek     = GPIO_llseek,
8      .read       = GPIO_read,
9      .write      = GPIO_write,
10     .poll        = GPIO_poll,
11     .open        = GPIO_open,
12     .release     = GPIO_rele
13 };

```

- `owner`: rappresenta puntatore al modulo che è il possessore della struttura. Ha lo scopo di evitare che il modulo venga rimosso quando uno delle funzionalità fornite è in uso. Inizializzato mediante la macro `THIS_MODULE`
- `GPIO_llseek`: sposta l'offset di lettura/scrittura sul file.
- `GPIO_read`: utilizzata per leggere dal device. La chiamata a `GPIO_read` potrebbe avvenire quando il device non ha dati disponibili, in questo caso il processo chiamante deve essere messo in una coda di processi sleeping in modo tale da mascherare all'esterno le dinamiche interne del device. Per far ciò viene utilizzata una variabile "can_read". La funzione read effettua un controllo sullo stato di quest'ultima e se rileva che non è possibile effettuare una lettura mette il processo in sleep. L'ISR avrà il compito di settare la variabile per poter rendere possibile la lettura e risvegliare i processi dalla coda. Per realizzare questo meccanismo sono stati utilizzati `spinlock` e `wait_queue` fornite dal kernel.
- `GPIO_write`: utilizzata per inviare dati al device.
- `GPIO_poll`: utilizzata per verificare se un'operazione di lettura sul device risulti bloccante. Verifica lo stato della variabile `can_read` e in caso sia possibile effettuare una lettura ritorna un'opportuna maschera.
- `GPIO_open`: chiamata all'apertura del file descriptor associato al device. Se alla chiamata viene specificato il flag `O_NONBLOCK` tutte le operazioni di lettura sul file descriptor aperto non risulteranno essere bloccanti.
- `GPIO_release`: chiamata alla chiusura del file descriptor associato al device.

Il codice allegato è diviso in:

- `GPIO.h/GPIO.c` : definizione e implementazione di una struttura che astrae il nostro device GPIO in kernel mode. Contiene ciò che è necessario al funzionamento del driver, compreso lo `spinlock` per l'accesso in mutua esclusione alla variabile `can_read` e le `wait_queue`.
- `GPIO_list.h/GPIO_list.c` : definizione e implementazione di una lista di oggetti GPIO. Fornisce tutte le funzioni necessarie per l'interfacciamento quali inizializzazione, cancellazione, aggiunta oggetto, ricerca.
- `GPIO_kernel_main.c`: rappresenta il vero e proprio modulo kernel che reimplementa le tutte funzioni necessarie all'interfacciamento.

Per compilare il modulo è sufficiente lanciare lo script “prepare_environment.sh” prima di dare il comando make. Segue il Makefile utilizzato per la compilazione:

```
1 obj-m += my_kernel_GPIO.o
2 my_kernel_GPIO-objs :=GPIO_kernel_main.o GPIO.o GPIO_list.o
3
4 all:
5     make -C linux-xlnx/ M=$(PWD) modules
6
7 clean:
8     make -C linux-xlnx/ M=$(PWD) clean
```

Una volta ottenuto il kernel object (.ko) l'ultima operazione da effettuare è quella di inserirlo mediante il comando:

```
1 insmod my_kernel_GPIO.ko
```

```
root@linaro-developer:/# insmod my_kernel_GPIO.ko
```

Se l'operazione è andata a buon fine si visualizzeranno i seguenti messaggi stampando il log del kernel tramite il comando dmesg:

```
[ 35.366499] my_kernel_GPIO: loading out-of-tree module taints kernel.
[ 35.367646] Chiamata GPIO_probe
[ 35.368084] IRQ registered as 46
[ 35.368093] Driver succesfully probed at Virtual Address 0xe0b20000
[ 35.368100] 43c00000.GPIO => GPIO0
[ 35.368270] Chiamata GPIO_probe
[ 35.371049] IRQ registered as 47
[ 35.371059] Driver succesfully probed at Virtual Address 0xe0b40000
[ 35.371066] 43c10000.GPIO => GPIO1
[ 35.371245] Chiamata GPIO_probe
[ 35.371540] IRQ registered as 48
[ 35.371550] Driver succesfully probed at Virtual Address 0xe0b60000
[ 35.371556] 43c20000.GPIO => GPIO2
root@linaro-developer:/#
```

Per mostrare il corretto funzionamento di tutte le funzionalità implementate sono state create due user application: read_block_user_app.c, read_NON_block_user_app.c e poll_user_app.c che sono allegate.

- read_block_user_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua in un loop infinito la chiamata a read per controllare se siano presenti nuovi valori da leggere sul dispositivo selezionato.
- read_NON_block_user_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua in un loop infinito la chiamata a read (distanziata l'una dall'altra di un tempo specificato tramite il parametro TIMEOUT per rendere verificabile il funzionamento) per controllare se siano presenti nuovi valori da leggere sul dispositivo. L'apertura del device è effettuata specificando il flag O_NONBLOCK per cui le chiamate a read non saranno mai bloccanti.
- poll_user_app.c : l'utente specifica tramite linea di comando quale GPIO vuole utilizzare (-s GPIO0 switches; -b GPIO1 buttons; -l GPIO2 leds). Effettua una chiamata a poll con un TIMEOUT specificato: se prima della scadenza di questo vengono rilevati nuovi valori da leggere la funzione ritorna la maschera degli eventi rilevati e viene effettuata una chiamata a read che non sarà bloccante; altrimenti la funzione ritorna il valore 0 e non verrà effettuata la chiamata a read in quanto bloccante.

Per rimuovere il modulo impartire il comando:

```
1 rmmmod my_kernel_GPIO.ko
```

```
root@linaro-developer:/# rmmmod my_kernel_GPIO.ko
```

```
[ 21.903441] Chiamata GPIO_remove
           ptr: de585e00
           name: 43c20000.GPIO
           id: 4294967295
[ 21.904022] Chiamata GPIO_remove
           ptr: de57e000
           name: 43c10000.GPIO
           id: 4294967295
[ 21.904514] Chiamata GPIO_remove
           ptr: de57e200
           name: 43c00000.GPIO
           id: 4294967295
```

0.1.6.2 UIO

L'Userspace I/O (UIO) è un framework che permette di gestire i driver direttamente nell'userspace e fornisce meccanismi di **gestione delle interruzioni a livello utente**. La prima operazione da effettuare prima di scrivere un driver UIO è recarsi all'interno del progetto del device-tree e aggiungere ai bootargs nel file system-top.dts il parametro "uio_pdrv_genirq.of_id=generic-uio" e all'interno del file pl.dtsi impostare il campo compatible dei device GPIO a "generic-uio". Segue il file pl.dtsi:

```
1 / {
2     amba_pl: amba_pl {
3         #address-cells = <1>;
4         #size-cells = <1>;
5         compatible = "simple-bus";
6         ranges ;
7         GPIO_0: GPIO@43c00000 {
8             /* This is a place holder node for a custom IP, user may need to update the entries */
9             clock-names = "s00_axi_aclk";
10            clocks = <&clkc 15>;
11            compatible = "generic-uio";
12            interrupt-names = "interrupt";
13            interrupt-parent = <&intc>;
14            interrupts = <0 29 4>;
15            reg = <0x43c00000 0x10000>;
16            xlnx,s00-axi-addr-width = <0x5>;
17            xlnx,s00-axi-data-width = <0x20>;
18        };
19        GPIO_1: GPIO@43c10000 {
20            /* This is a place holder node for a custom IP, user may need to update the entries */
21            clock-names = "s00_axi_aclk";
22            clocks = <&clkc 15>;
23            compatible = "generic-uio";
24            interrupt-names = "interrupt";
25            interrupt-parent = <&intc>;
26            interrupts = <0 30 4>;
27            reg = <0x43c10000 0x10000>;
28            xlnx,s00-axi-addr-width = <0x5>;
29            xlnx,s00-axi-data-width = <0x20>;
30        };
31        GPIO_2: GPIO@43c20000 {
32            /* This is a place holder node for a custom IP, user may need to update the entries */
33            clock-names = "s00_axi_aclk";
34            clocks = <&clkc 15>;
35            compatible = "generic-uio";
36            interrupt-names = "interrupt";
37            interrupt-parent = <&intc>;
38            interrupts = <0 31 4>;
39            reg = <0x43c20000 0x10000>;
40            xlnx,s00-axi-addr-width = <0x5>;
41            xlnx,s00-axi-data-width = <0x20>;
```

```

42     };
43 };
44 };

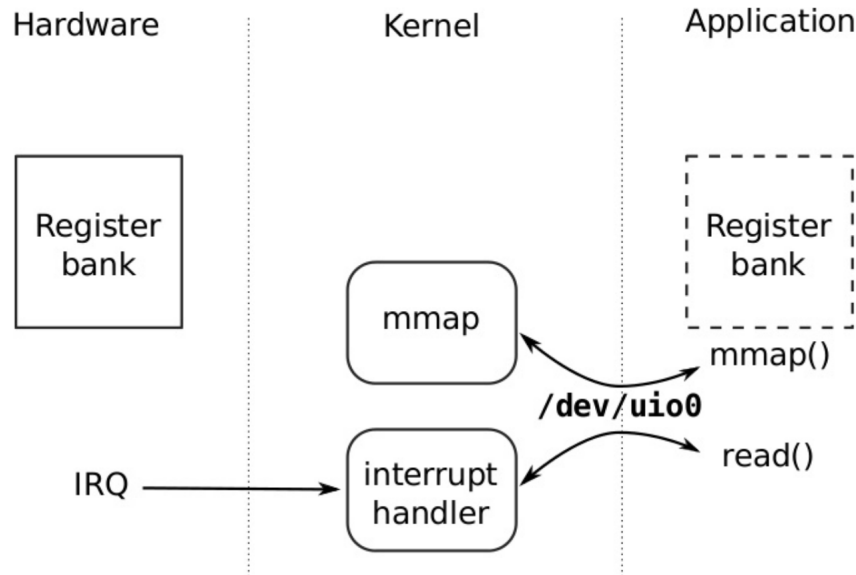
```

A questo punto si ricompila il device-tree generando il file .dtb e lo si sposta nella partizione di BOOT della SD Card. All'avvio del sistema operativo si potranno osservare sotto /dev i tre device. Il driver userspace effettuerà il mapping dei device per poi mettersi in attesa di notifica di interrupt tramite chiamata a read. Segue uno schema generale.

```

root@linaro-developer:~# ls /dev/
block          memory_bandwidth  ram6   tty19  tty39  tty59  vcsa1
char           mmcblk0           ram7   tty2   tty4   tty6   vcsa2
console       mmcblk0p1         ram8   tty20  tty40  tty60  vcsa3
cpu_dma_latency mmcblk0p2         ram9   tty21  tty41  tty61  vcsa4
disk          network_latency   random tty22  tty42  tty62  vcsa5
fd            network_throughput shm    tty23  tty43  tty63  vcsa6
full          null              snd    tty24  tty44  tty7   vcsu
gpiochip0     port              stderr tty25  tty45  tty8   vcsu1
io:device0    ptmx              stdin  tty26  tty46  tty9   vcsu2
initctl       pts               stdout tty27  tty47  ttyPS0 vcsu3
kmsg          ram0              tty    tty28  tty48  uio0   vcsu4
log           ram1              tty0   tty29  tty49  uio1   vcsu5
loop-control  ram10             tty1   tty3   tty5   uio2   vcsu6
loop0         ram11             tty10  tty30  tty50  urandom vga_arbiter
loop1         ram12             tty11  tty31  tty51  vcs    watchdog
loop2         ram13             tty12  tty32  tty52  vcs1   watchdog0
loop3         ram14             tty13  tty33  tty53  vcs2   xconsole
loop4         ram15             tty14  tty34  tty54  vcs3   zero
loop5         ram2              tty15  tty35  tty55  vcs4
loop6         ram3              tty16  tty36  tty56  vcs5
loop7         ram4              tty17  tty37  tty57  vcs6
mem           ram5              tty18  tty38  tty58  vcsa

```



Segue il codice relativo al driver UIO:

```

1  #include <unistd.h>
2  #include <stdlib.h>
3  #include <stdio.h>
4  #include <limits.h>
5  #include <sys/types.h>
6  #include <sys/stat.h>
7  #include <fcntl.h>
8  #include <sys/mman.h>
9  #include <poll.h>
10 #include "GPIO_interrupt_uio_poll.h"
11

```

```

12 #define DIR_OFF      0  // DIRECTION
13 #define WRITE_OFF    4  // WRITE
14 #define READ_OFF     8  // READ
15 #define GLOBAL_INTR_EN 12 // GLOBAL INTERRUPT ENABLE
16 #define INTR_EN      16 // LOCAL INTERRUPT ENABLE
17 #define INTR_ACK_PEND 28 // PENDING/ACK REGISTER
18
19 #define INTR_MASK 15
20
21 #define TIMEOUT 2000
22
23 typedef u_int8_t u8;
24 typedef u_int32_t u32;
25
26 void write_reg(void *addr, unsigned int offset, unsigned int value)
27 {
28     *((unsigned*)(addr + offset)) = value;
29 }
30
31 unsigned int read_reg(void *addr, unsigned int offset)
32 {
33     return *((unsigned*)(addr + offset));
34 }
35
36
37 void wait_for_interrupt(int fd0, int fd1, int fd2, void *addr_0, void *addr_1, void *addr_2)
38 {
39
40     int pending = 0;
41     int reenable = 1;
42     u32 read_value;
43     struct pollfd poll_fds [3];
44     int ret;
45
46     printf("Waiting for interrupts....\n");
47
48     poll_fds[0].fd = fd0;
49     poll_fds[0].events = POLLIN; //The field events is an input parameter, a bit mask
        specifying the
50         //events the application is interested in for the file descriptor fd.
51         //Means that we are interested at the event: there is data to read.
52     poll_fds[1].fd = fd1;
53     poll_fds[1].events = POLLIN;
54
55     poll_fds[2].fd = fd2;
56     poll_fds[2].events = POLLIN;
57
58     // non blocking wait for an interrupt on file descriptors specified in the pollfd
        structure*/
59     ret = poll(poll_fds, 3, TIMEOUT); //timeout of TIMEOUT ms
60     if (ret > 0){
61         if(poll_fds[0].revents && POLLIN){
62
63             read(fd0, (void *)&pending, sizeof(int));
64             write_reg(addr_0, GLOBAL_INTR_EN, 0); //disabilito interruzioni
65             printf("*****ISR SWITCH*****\n");
66             read_value = read_reg(addr_0, READ_OFF);
67             printf("Read value: %08x\n", read_value);
68             write_reg(addr_0, INTR_ACK_PEND, INTR_MASK); //ACK
69             sleep(1);

```

```

70     write_reg(addr_0, INTR_ACK_PEND, 0); //ACK
71     write_reg(addr_0, GLOBAL_INTR_EN, 1); //abiito interruzioni
72     write(fd0, (void *)&reenable, sizeof(int));
73
74 }
75 if(poll_fds[1].revents && POLLIN){
76
77     read(fd1, (void *)&pending, sizeof(int));
78     write_reg(addr_1, GLOBAL_INTR_EN, 0); //disabilito interruzioni
79     printf("*****ISR BUTTON*****\n");
80     read_value = read_reg(addr_1, READ_OFF);
81     printf("Read value: %08x\n", read_value);
82     write_reg(addr_1, INTR_ACK_PEND, INTR_MASK); //ACK
83     sleep(1);
84     write_reg(addr_1, INTR_ACK_PEND, 0); //ACK
85     write_reg(addr_1, GLOBAL_INTR_EN, 1); //abiito interruzioni
86     write(fd1, (void *)&reenable, sizeof(int));
87
88 }
89 if(poll_fds[2].revents && POLLIN){
90
91     read(fd2, (void *)&pending, sizeof(int));
92     write_reg(addr_2, GLOBAL_INTR_EN, 0); //disabilito interruzioni
93     printf("*****ISR LED*****\n");
94     read_value = read_reg(addr_2, READ_OFF);
95     printf("Read value: %08x\n", read_value);
96     write_reg(addr_2, INTR_ACK_PEND, INTR_MASK); //ACK
97     sleep(1);
98     write_reg(addr_2, INTR_ACK_PEND, 0); //ACK
99     write_reg(addr_2, GLOBAL_INTR_EN, 1); //abiito interruzioni
100    write(fd2, (void *)&reenable, sizeof(int));
101
102 }
103 }
104
105 }
106
107 int main(int argc, char *argv[]){
108
109     void *gpio_0_ptr;
110     void *gpio_1_ptr;
111     void *gpio_2_ptr;
112
113     //-----MAPPING GPIO_0-----//
114
115     int fd_gpio_0 = open("/dev/uio0", O_RDWR);
116     if (fd_gpio_0 < 1){
117         printf("Errore nell'accesso al device UIO0.\n");
118         return -1;
119     }
120
121     unsigned dimensione_pag = sysconf(_SC_PAGESIZE);
122
123     gpio_0_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_0, 0);
124
125     write_reg(gpio_0_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
126     write_reg(gpio_0_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
127
128     //-----MAPPING GPIO_1-----//
129

```

```

130 int fd_gpio_1 = open("/dev/uio1", O_RDWR);
131 if (fd_gpio_1 < 1){
132     printf("Errore nell'accesso al device UIO1.\n");
133     return -1;
134 }
135
136 gpio_1_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_1, 0);
137
138 write_reg(gpio_1_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
139 write_reg(gpio_1_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
140
141 //-----MAPPING GPIO_2-----//
142
143 int fd_gpio_2 = open("/dev/uio2", O_RDWR);
144 if (fd_gpio_2 < 1){
145     printf("Errore nell'accesso al device UIO2.\n");
146     return -1;
147 }
148
149 gpio_2_ptr = mmap(NULL, dimensione_pag, PROT_READ|PROT_WRITE, MAP_SHARED, fd_gpio_2, 0);
150
151 write_reg(gpio_2_ptr, GLOBAL_INTR_EN, 1); // abilitazione interruzioni globali
152 write_reg(gpio_2_ptr, INTR_EN, INTR_MASK); // abilitazione interruzioni
153
154
155
156 while (1) {
157     printf("Calling function wait_for_interrupt: ");
158     wait_for_interrupt(fd_gpio_0, fd_gpio_1, fd_gpio_2, gpio_0_ptr, gpio_1_ptr, gpio_2_ptr);
159 }
160
161 // unmap the gpio device
162 munmap(gpio_0_ptr, dimensione_pag);
163 munmap(gpio_1_ptr, dimensione_pag);
164 munmap(gpio_2_ptr, dimensione_pag);
165
166 return 0;
167
168 }

```

La prima operazione del driver, come introdotto all'inizio della sezione, è quella di aprire tre file descriptor sui tre device uio corrispondenti ai tre GPIO. Successivamente calcola la dimensione della pagina e effettua il mapping tramite chiamata a `mmap()`. Si è scelto di non effettuare chiamate a `read()` bloccanti ma di utilizzare la system call `poll()` per verificare se sono disponibili nuovi dati prima di effettuare una lettura. La funzione prende in ingresso un array di strutture `pollfd` composte da tre campi:

1. file descriptor: descrittore del file associato al device.
2. events: maschera di bit che indica gli eventi, relativi al file descriptor, ai quali l'applicazione è interessata.
3. revents: maschera riempita dal kernel contenente gli eventi rilevati.

La chiamata `poll()` prende in ingresso la suddetta struttura, un intero che indica quanti oggetti sono presenti in quest'ultima e un parametro che indica il tempo che il processo deve attendere notifiche di eventi dal device espresso in millisecondi.