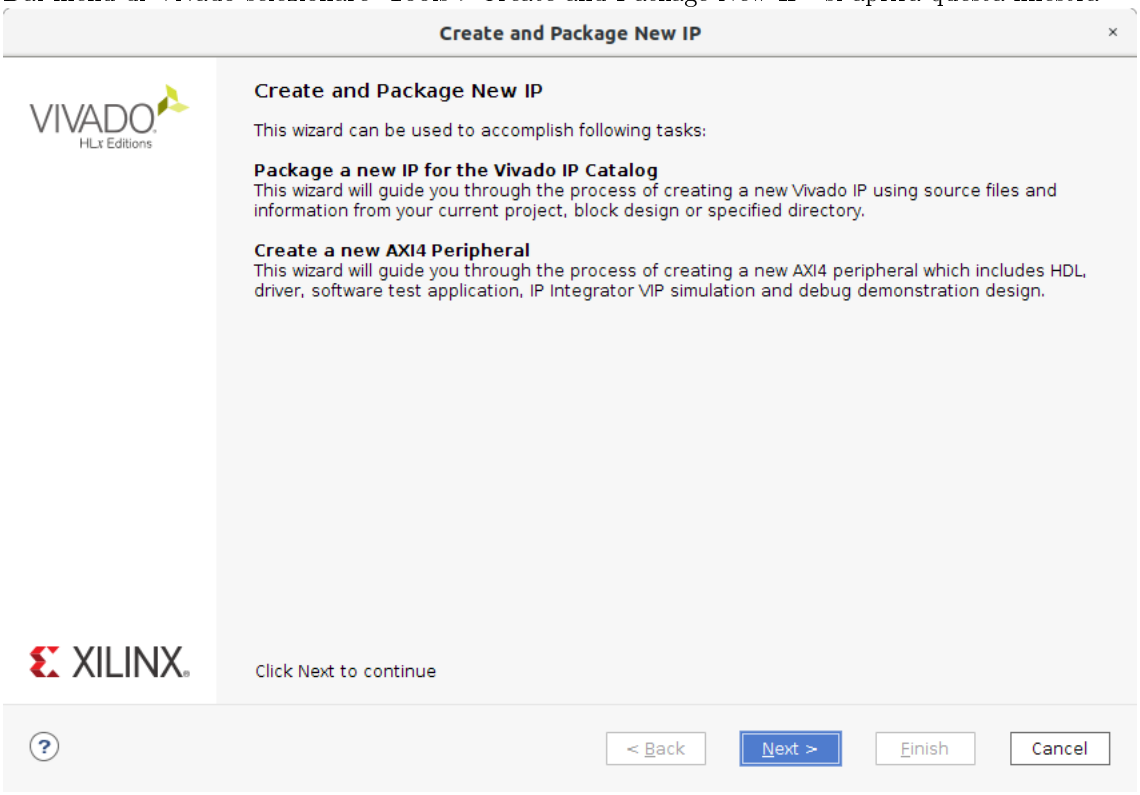
0.1 Soluzione

0.1.1 Descrizione GPIO

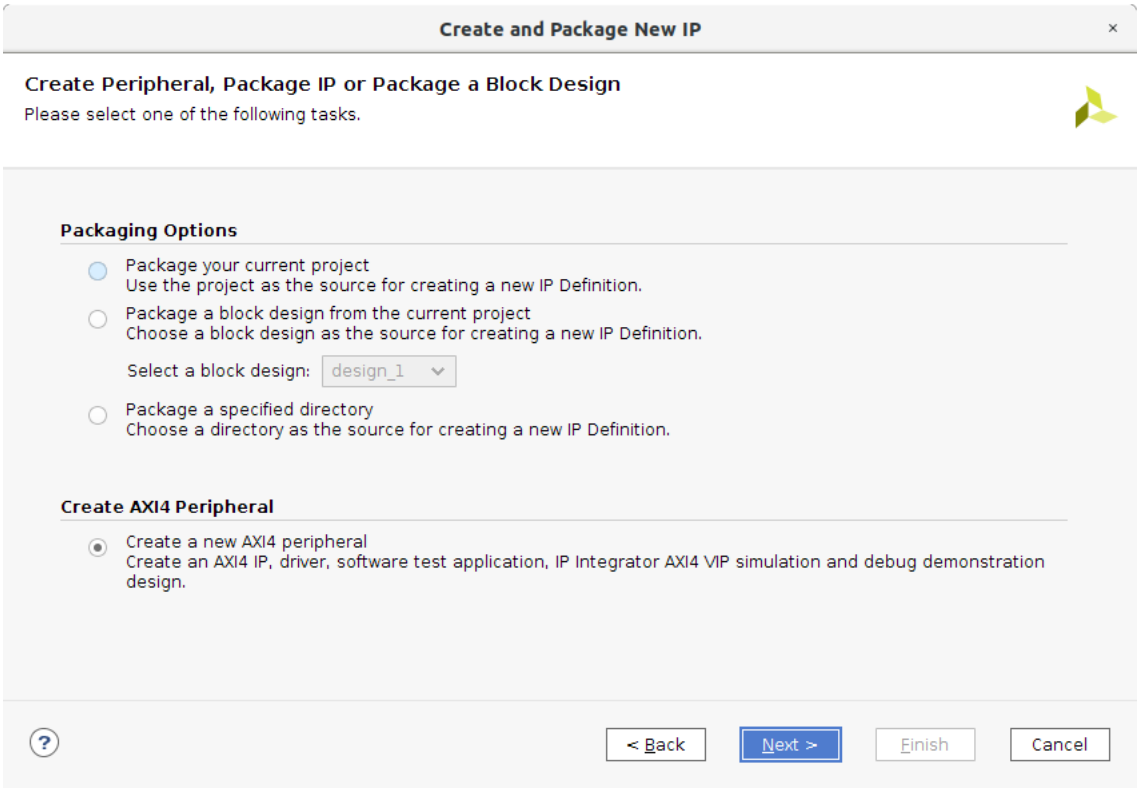
Il general purpose input output è un interfaccia che permette di leggere il valore proveniente da un pin oppure di scrivere un valore.
Nel nostro design il segnale di enable decide il comportamento dell' interfaccia, nel caso questo venga settato ad uno il valore del bit write viene forzato sul pin pad, altrimenti il valore del pin al quale è collegato pad viene letto tramite il pin read.

0.1.2 Creazione Custom IP

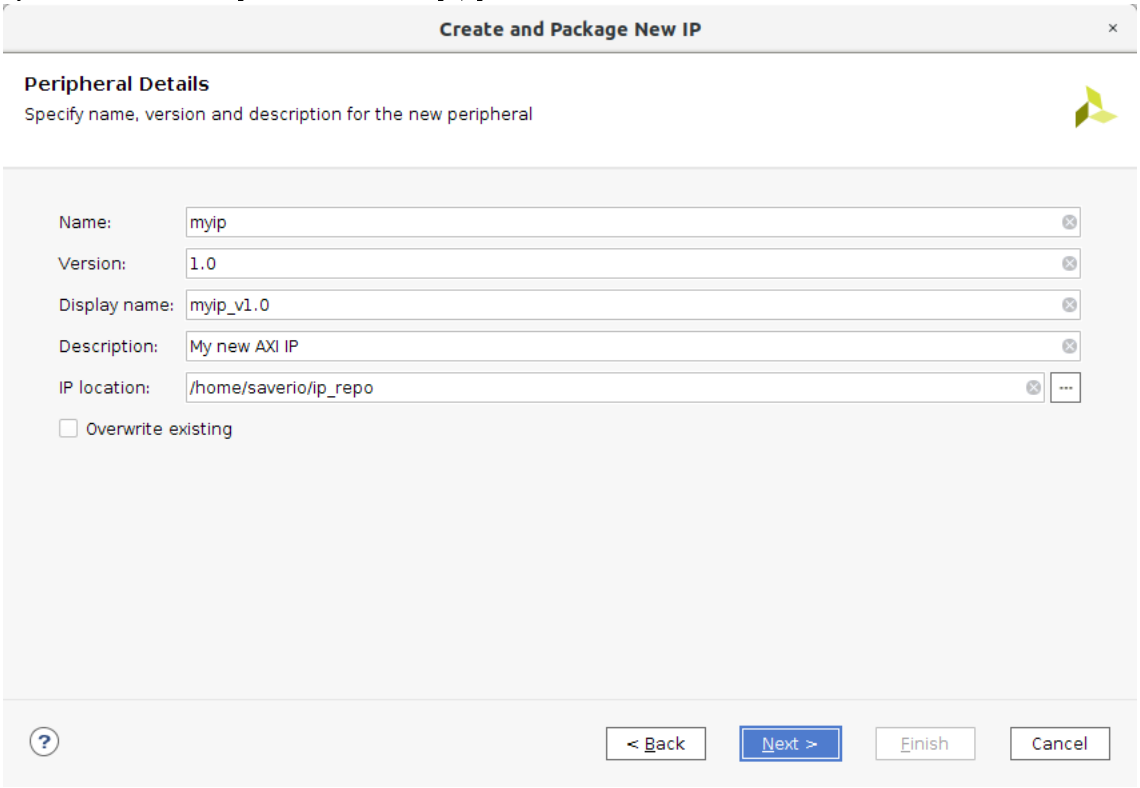
Dal menù di Vivado selezionare "Tools->Create and Package New IP" si aprirà questa finestra



Cliccare su "Next", selezionare "Create a New AXI4 peripheral", di nuovo "Next"

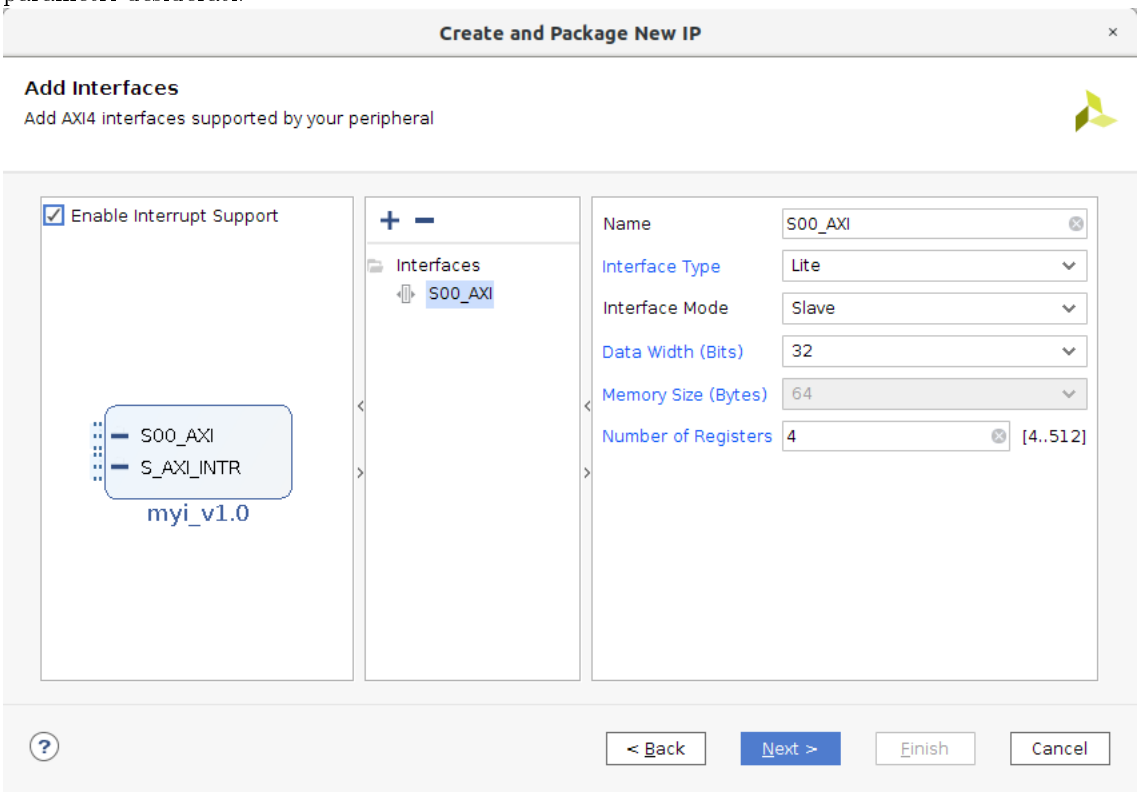


Riempire i vari campi con le informazioni dell' IP, in questa finestra è importante il parametro "IP location" che permette di selezionare in quale cartella salveremo il nostro custom IP.
Quando si avrà compilato tutti i campi, procedere cliccando su "Next"

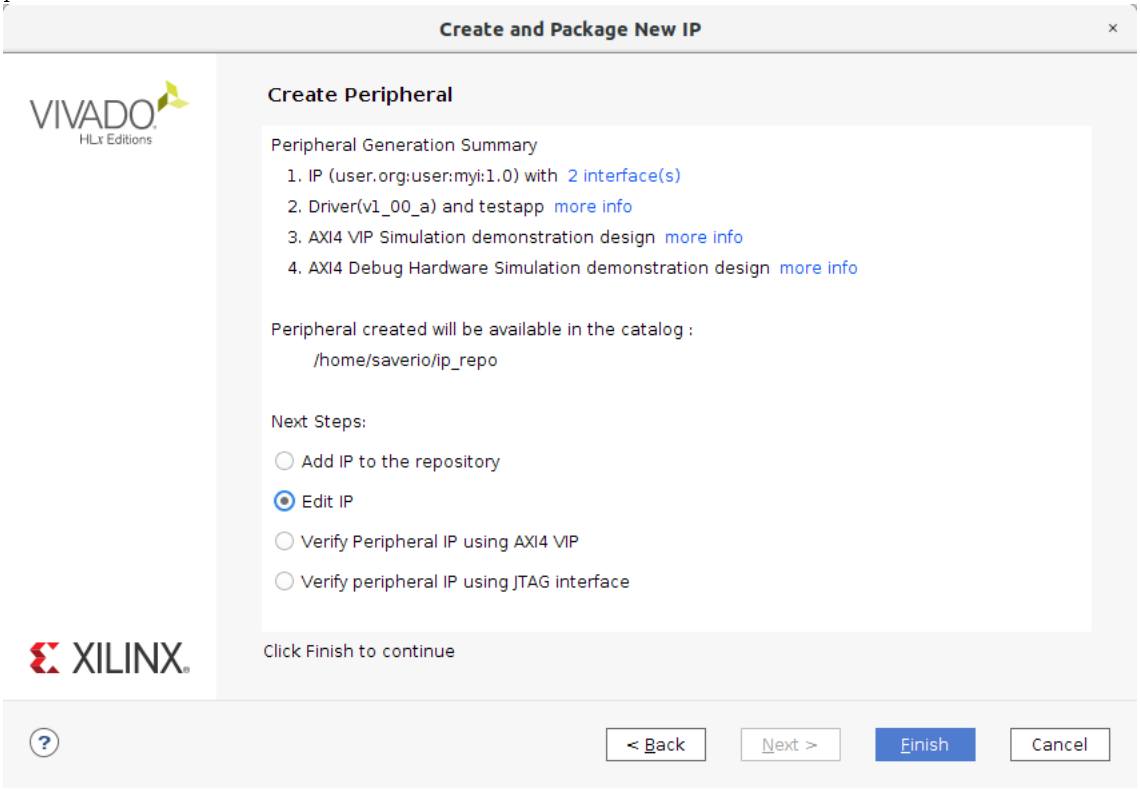


Nella finestra successiva è possibile configurare il tipo di interfaccia del nostro componente, nel nostro caso LITE, che tipo di device è, Slave dato che non gestisce le transizioni del bus AXI, la dimensione dei registri in cui andremo a scrivere ed a leggere ed il numero di registri che vogliamo utilizzare, clicchiamo su "Next" una volta settati i

parametri desiderati.



Selezionare "Edit IP" e successivamente cliccare su "Finish" verrà creata una nuova finestra di Vivado in cui è possibile modificare il custom IP.



Nella nuova finestra vediamo che sono stati creati due nuovi file, uno con il nome del nostro custom IP che rappresenta l' interfaccia del nostro componente ed un altro con il "nome del nostro IP _nome settato nella generazione del componente".

0.1.3 Modifica Default IP Core

Aggiungiamo al design creato da Vivado il nostro GPIO array ed il singolo GPIO, cliccando su "Add Sources" presente sulla sinistra della interfaccia, una volta fatto ciò instanziamo il nostro componente nel file "nome del nostro IP _nome settato nella generazione del componente", è buona norma farlo nella sezione indicata da Vivado tra i seguenti commenti

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

Il componente instanziato è il seguente

```
gpio_inst : GPIO_Array
generic map(width => width)
port map(    enable => slv_reg0(width-1 downto 0),
       write => slv_reg1(width-1 downto 0),
       read => gpio_read(width-1 downto 0),
       pads =>pads);
```

Il valore di enable e di write vengono gestiti dai valori presenti nei registri slv_reg0 ed 1, invece il valore di read viene salvato nel registro gpio_read, non può essere usato uno degli slv_reg generato automaticamente da vivado poiché read vuole forzare dei valori essendo un pin di output, ma anche gli slv_reg forzano dei valori essendo i loro valori forzati da un process che ne permette la scrittura dei valori, per tale motivo viene introdotto in questo caso il segnale gpio_read che non viene scritto ma può essere solo letto modificando il process di scrittura nel seguente modo

```
process (slv_reg0, slv_reg1, gpio_read, slv_reg3, slv_reg4, slv_reg5, status_reg_out,
    slv_reg7_out, axi_araddr, S_AXI_ARESETN, slv_reg_rden)
  variable loc_addr :std_logic_vector(OPT_MEM_ADDR_BITS downto 0);
  begin
      -- Address decoding for reading registers
      loc_addr := axi_araddr(ADDR_LSB + OPT_MEM_ADDR_BITS downto ADDR_LSB);
      case loc_addr is
        when b"000" =>
          reg_data_out <= slv_reg0;
        when b"001" =>
          reg_data_out <= slv_reg1;
        when b"010" =>
          reg_data_out <= gpio_read;
        when b"011" =>
          reg_data_out <= slv_reg3;
        when b"100" =>
          reg_data_out <= slv_reg4;
        when b"101" =>
          reg_data_out <= slv_reg5;
        when b"110" =>
          reg_data_out <= status_reg_out;
        when b"111" =>
          reg_data_out <= slv_reg7_out;
        when others =>
          reg_data_out  <= (others => '0');
      end case;
  end process;
```

In questo caso invece di leggere lo slv_reg3 verrà letto il segnale gpio_read, ma i valori di tale registro non vengono forzati da nessuno se non dal pin read.
Aggiungiamo i parametri del nostro componente in questo caso width tra i seguenti commenti

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

ed i segnali che si devono esporre nella seguente sezione

```
-- Users to add ports here
          pads : inout std_logic_vector(width-1 downto 0); --! se GPIO in modalità lettura
              mostra il valore letto, altrimenti forza un valore in uscita
      interrupt : out std_logic; --! segnale di interrupt
-- User ports ends
```

Per catturare le variazione del componente GPIO affinché da avviare l' interruzione è stato utilizzato il seguente process

```
gpio_read_sampling : process (S_AXI_ACLK, gpio_read)
    begin
    if (rising_edge (S_AXI_ACLK)) then
        if ( S_AXI_ARESETN = '0' ) then
            last_stage <= (others => '0');
            current_stage <= (others => '0');
        else
            last_stage <= gpio_read(width-1 downto 0);
            current_stage <= last_stage;
         end if;
        end if;
    end process;
```

Tale frammento di codice occorre per determinare quando vi è una variazione dei segnali gpio_read.
Essenzialmente è costituito da tre flip-flop collegati in cascata, i primi due occorrono per sincronizzare il valore di gpio_read con il clock, poiché a noi interessa cattuare cambiamenti di tale segnale solo in presenza di fronti, di salita nel nostro caso, difatti al primo colpo di clock il primo flip flop salva il valore del segnale, al secondo colpo di clock il secondo flip flop salva il valore precedentemente salvato, così da rispettare la successione degli eventi del segnale da campionare, mentre il primo flip flop campione di nuovo il segnale in ingresso; il terzo flip flop occorre per problemi di metastabilità che possono occorrere in un circuito reale (ad esempio il valore del primo flip flop da alto passa a basso e il secondo flip flop campiona quando tale valore si trova a metà tra i due livelli, non permettendo al flip flop di salvare un valore stabile, che può essere corretto quando il terzo flip flop campiona il segnale del secondo, ovviamente tale situazione di meta stabilità può continuare anche con più flip flop, ma la percentuale con cui può capitare avendo tre flip flop e molto bassa.
Un secondo process viene utilizzato per la gestione del registro delle interruzioni pending

```
intr_pending : process (S_AXI_ACLK, change_detected, ack_intr)
    begin
    if (rising_edge (S_AXI_ACLK)) then
        if (change_detected = '1') then
            pending_intr <= pending_intr_tmp or changed_bits;
        else
            if (or_reduce(ack_intr)='1') then
                pending_intr <= pending_intr_tmp and (not ack_intr);
            end if;
           end if;
       end if;
    end process;
```

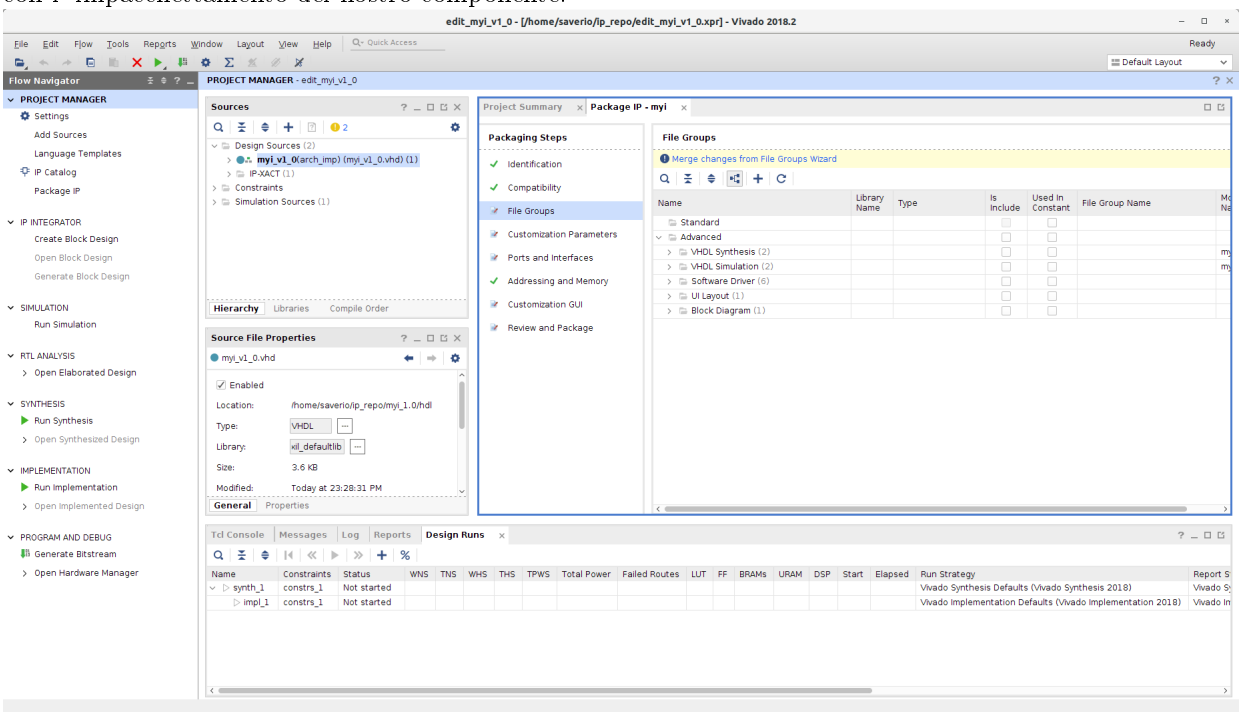
Qui si determina quando il valore del registro pending debba essere alto o basso, osserviamo che nel caso in una variazione del segnale change_detected il cui valore dipende da tale espressione

```
change_detected <= global_intr and (or_reduce(changed_bits and intr_mask and (not
   gpio_enable)));
```

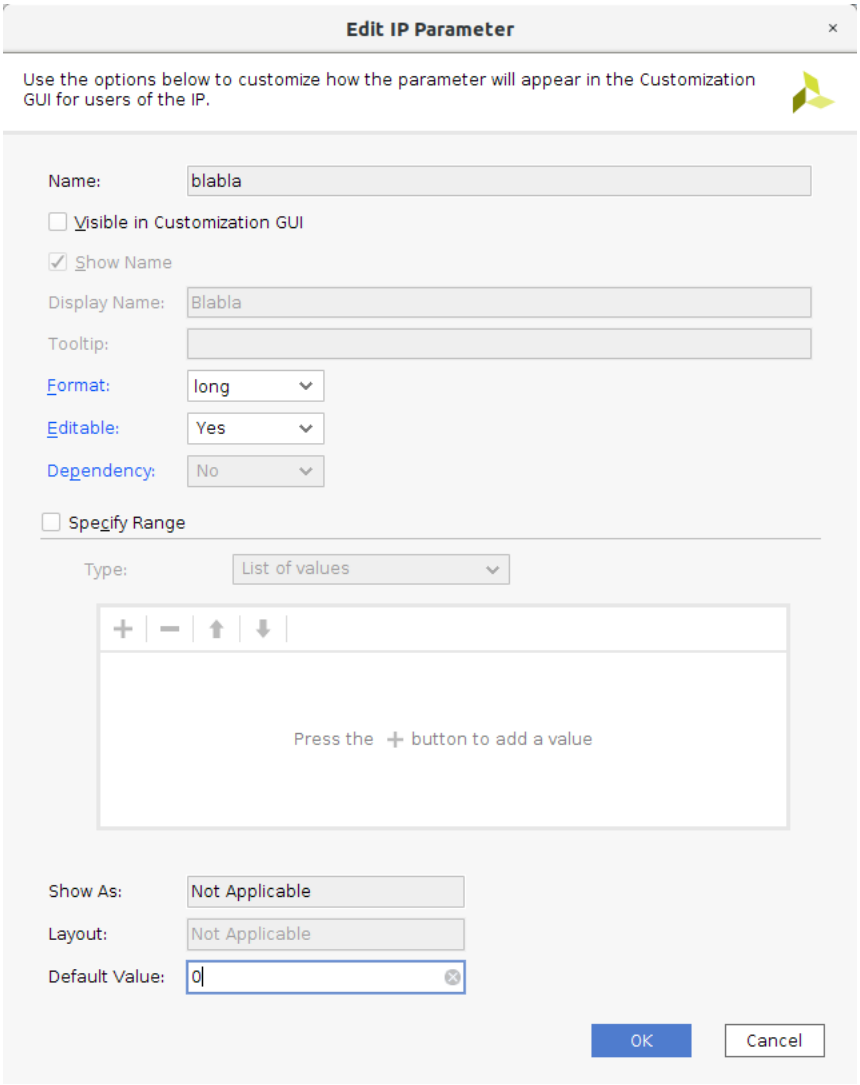
cioè ci dice che vi è una variazione solamente se le interruzioni globali sono attive ed è stata rilevato un interrupt abilitato dalla maschera intr_mash e tale pin di GPIO sia attivo, il valore del registro pending è dato da il valore di una precedente interruzione non servita oppure ad una interuzione rilevata (changed_bits).
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

```
inst_irq : process(S_AXI_ACLK,pending_intr)
    begin
        if (rising_edge (S_AXI_ACLK)) then
            if ( S_AXI_ARESETN = '0' ) then
                    interrupt <= '0';
            else
                if (or_reduce(pending_intr) = '1') then
                   interrupt <= '1';
                else
                    interrupt <= '0';
                end if;
            end if;
        end if;
    end process;
```

tale segnale è uno nel caso vi siano interruzioni pendenti, nel caso di reset del bus o non vi siano interruzioni 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.



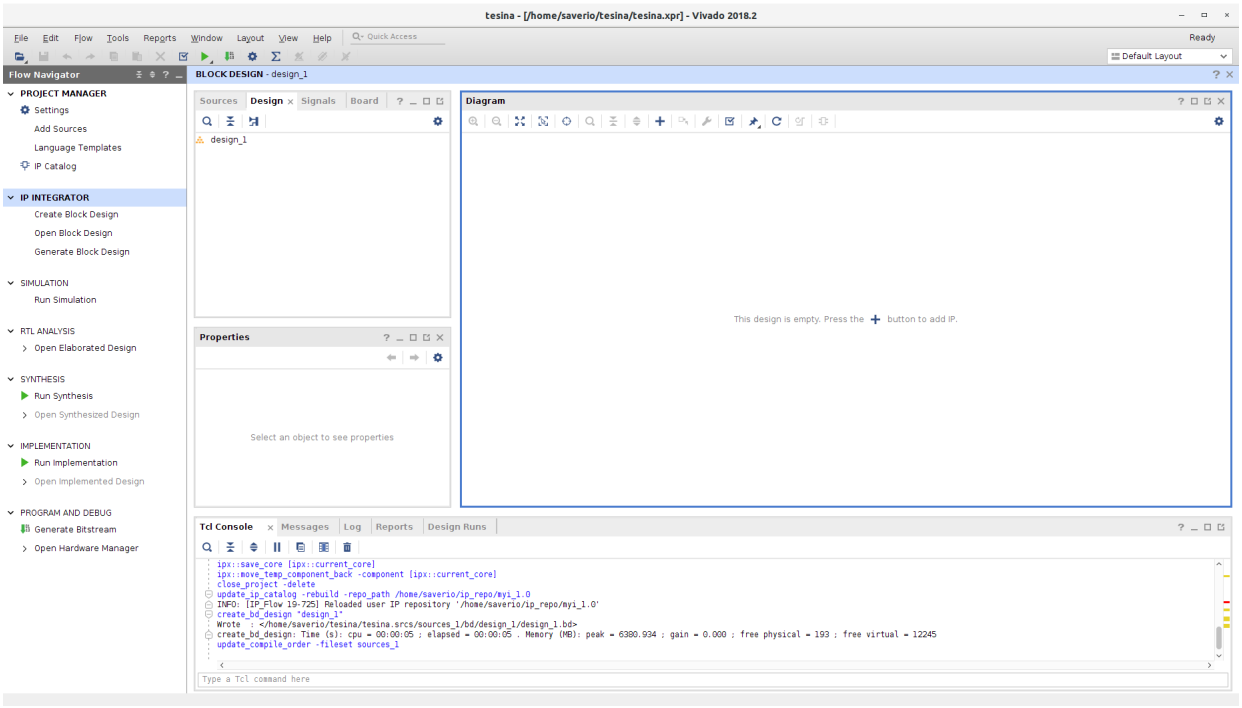
Nella sezione “File Groups” cliccare su “Merge changes from File Groups Wizard”, in “Customization Parameters” su “Merge changes from Customization Parameters Wizard”, selezionare “Hidden Parameters” si aprirà tale finestra



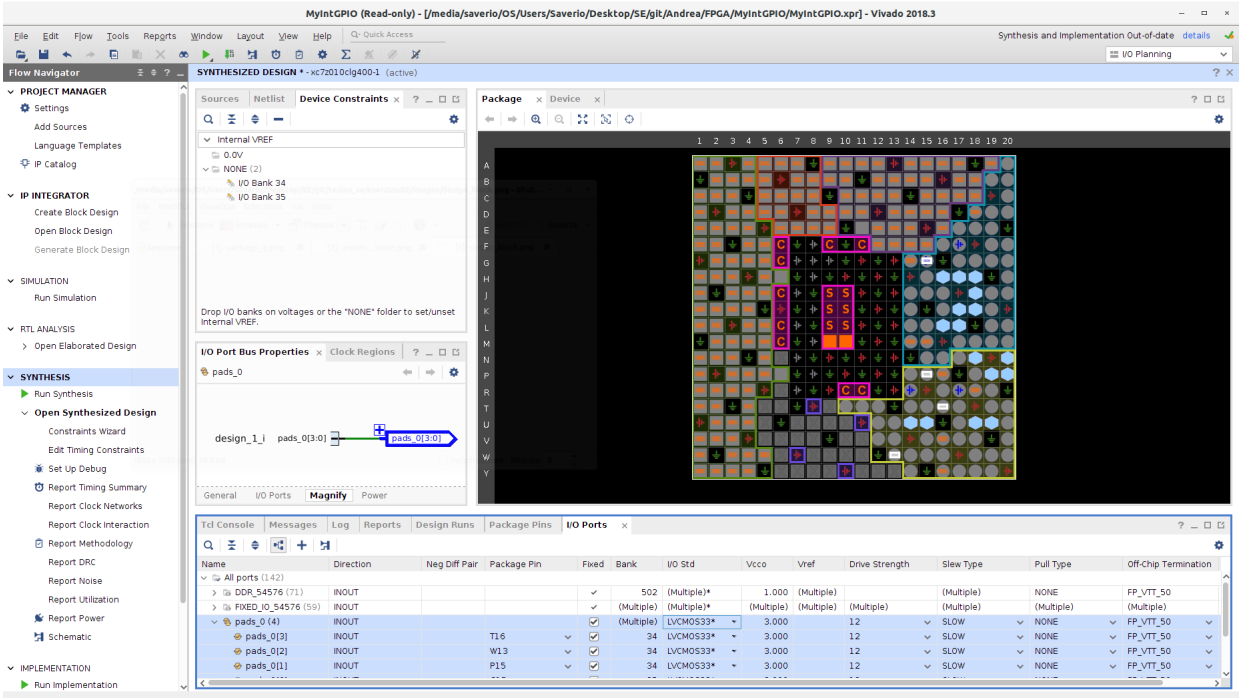
Da qui è possibile rendere visibile questo parametro da configurare cliccando sul box "Visible in Customization GUI" e settarne i valori che può assumere cliccando su "Specify Range".
Recarsi infine su "Review and Package" cliccare su "Re-Package IP" per ottenere il custom ip, facendo chiudere la finestra di vivado.

0.1.4 Creazione del block design

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" i vari componenti si collegheranno tra di loro automaticamente, una volta fatto ciò bisogna rendere esterni i pin che si vuole pilotare dalla board cliccando con il tasto destro del mouse su un pin e selezionando "Make external" una volta fatto selezionare l' icona contrassegnata da un simbolo di spunta ed è possibile chiudere il block design.
Bisogna generare ora un wrapper HDL affinchè 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



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

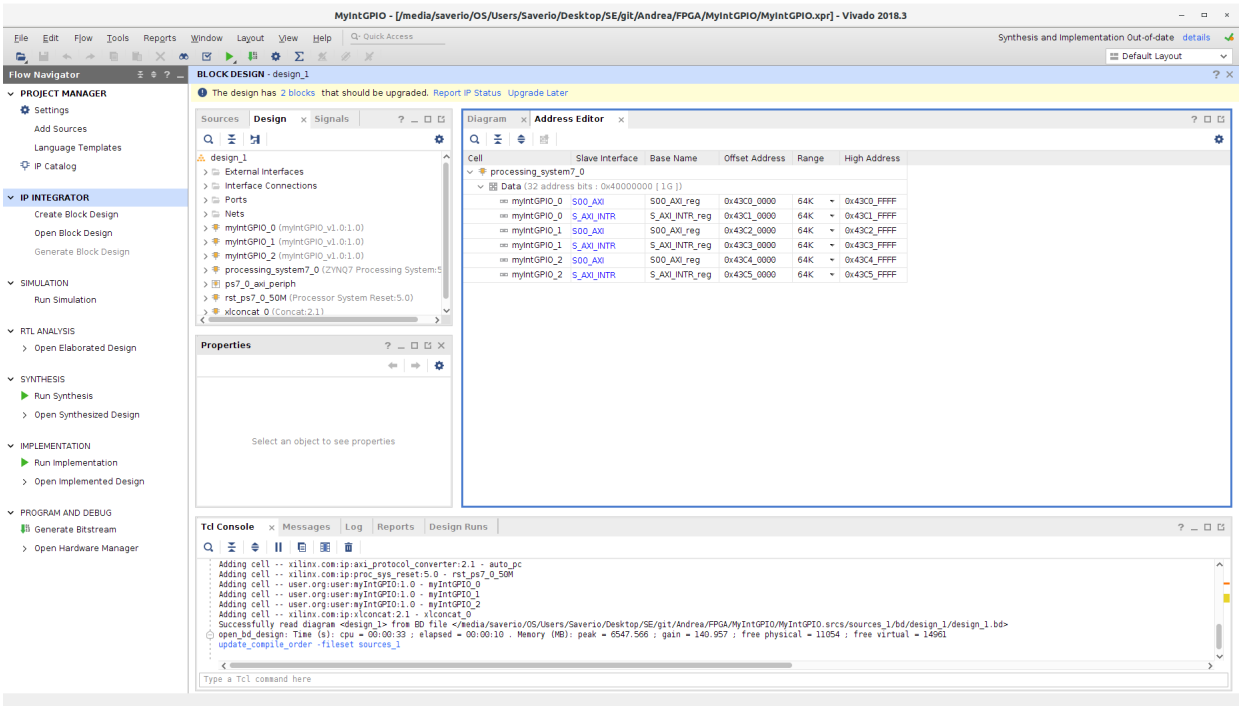
Il driver ora presentato, viene eseguito direttamente dalla sezione PS della board senza il supporto di un sistema operativo linux.
Per poter procedere alla scrittura del driver, dobbiamo sapere dove i registri del nostro componente hardware sono stati mappati, tale informazione può essere reperita dal file "xparameters.h" presente nella directory "'cartella_del_board_support troveremo una sezione di codice molto simile

```
/* Definitions for driver MYINTGPIO */
#define XPAR_MYINTGPIO_NUM_INSTANCES 3
/* Definitions for peripheral MYINTGPIO_0 */
#define XPAR_MYINTGPIO_0_DEVICE_ID 0
#define XPAR_MYINTGPIO_0_S00_AXI_BASEADDR 0x43C00000
#define XPAR_MYINTGPIO_0_S00_AXI_HIGHADDR 0x43C0FFFF
/* Definitions for peripheral MYINTGPIO_1 */
#define XPAR_MYINTGPIO_1_DEVICE_ID 1
#define XPAR_MYINTGPIO_1_S00_AXI_BASEADDR 0x43C20000
#define XPAR_MYINTGPIO_1_S00_AXI_HIGHADDR 0x43C2FFFF
/* Definitions for peripheral MYINTGPIO_2 */
#define XPAR_MYINTGPIO_2_DEVICE_ID 2
#define XPAR_MYINTGPIO_2_S00_AXI_BASEADDR 0x43C40000
#define XPAR_MYINTGPIO_2_S00_AXI_HIGHADDR 0x43C4FFFF
```

dove è possibile sapere il numero dei nostri custom IP core instanziati 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 spiazziati 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_ una volta aperto sarà presente una sezione simile in alto nel file

```
#define MYINTGPIO_S00_AXI_SLV_REG0_OFFSET 0
#define MYINTGPIO_S00_AXI_SLV_REG1_OFFSET 4
#define MYINTGPIO_S00_AXI_SLV_REG2_OFFSET 8
#define MYINTGPIO_S00_AXI_SLV_REG3_OFFSET 12
```

Ottenute queste informazioni possiamo leggere e scrivere valori nei registri.
Per la gestione delle interruzioni invece abbiamo bisogno dato che il nostro componente sfrutta le interruzioni dobbiamo conoscere dove sono mappati i registri, creati da vivado, che gestiscono e ci danno informazioni sulle interruzioni che avvengono, tali indirizzi sono visibile aprendo il block design e recandoci nella sezione address editor, visibile in figura.



Per conoscere invece di quanto sono spaziati i registri che controllano l' evoluzione della gestione dell' interrupt dall' indirizzo base nel file "nome del componente _S_AXI_INTR" presente all' interno del nostro custom IP è presente un process simile

```
process (reg_global_intr_en, reg_intr_en, reg_intr_sts, reg_intr_ack, reg_intr_pending,
    axi_araddr, S_AXI_ARESETN, intr_reg_rden)
    variable loc_addr :std_logic_vector(2 downto 0);
      begin
        if S_AXI_ARESETN = '0' then
      reg_data_out  <= (others => '0');
    else
    -- Address decoding for reading registers
    loc_addr := axi_araddr(4 downto 2);
    case loc_addr is
      when "000" =>
        reg_data_out <= x"0000000" & "000" & reg_global_intr_en(0);
      when "001" =>
        reg_data_out <= reg_intr_en;
      when "010" =>
        reg_data_out <= reg_intr_sts;
            when "011" =>
        reg_data_out <= reg_intr_ack;
      when "100" =>
                reg_data_out <= reg_intr_pending;
      when others =>
        reg_data_out  <= (others => '0');
    end case;
    end if;
end process;
```

Vediamo che sono presenti cinque registri di dimensione pari a trentadue bit aventi il seguente compito:

1. il primo abilita il componente a poter generare interrupt;
2. il secondo dice quali segnali di interrupt vogliamo abilitare;
3. il terzo ci dice quali interruzioni sono avvenute;

4. il quarto quando viene scritto si asserisce che l' interruzione è stata gestita;
5. il quinto le interruzioni pendenti;

La differenza del registro di stato è quello pendenti sta che, una volta abilitate tutte le interruzioni quello di stato mostra tutte le interruzioni che avvengono nel componente, invece quello pendenti mostra solamente le interruzioni avvenute e che sono abilitate.
Possiamo procedere alla creazione del driver, instanziamo un nuovo progetto vuoto dal menù "File -> New -> Application Project" e creare un progetto vuoto, sul progetto appena creato andremo ad inserire la logica del nostro driver, di seguito verrà fornita un' idea di come funziona il nostro driver, per dettagli sulla implementazione riferirsi al codice.
Dobbiamo abilitare lo scugic per essere sensibili alle interruzioni, chiamando prima la funzione

```
XScuGic_LookupConfig("id scugic");
```

per conoscere la configurazione attuale dello scugic e successivamente

```
XScuGic_CfgInitialize("variabile di tipo XScuGic","variabile configurazione","indirizzo base
    della CPU");
```

Ora possiamo abilitare le interruzioni del nostro componente programmabile:

1. Abilitiamo l' interruzione scrivendo un uno nel registro reg_intr_en;
2. Abilitiamo le interruzioni globali del componente un uno nel registro reg_global_intr_en;
3. Settiamo la routine che gestisce le interruzioni;
4. Connettiamo l' handler fornito da Xilinx con la logica di handling fornita dal processore con la primitiva

```
Xil_ExceptionRegisterHandler
```

5. Assegnamo alla ricezione di un determinato interrupt la routine da noi creata per la sua gestione con la primitiva

```
XScuGic_Connect
```

   così che il gic sappia quale routine lanciare alla ricezione di quella interruzione
6. Abilitiamo le eccezioni nel processore ARM con

```
Xil_ExceptionEnable
```

7. Settiamo la maschera del nostro componente detector per dire quali pin del GPIO vogliamo campionare
8. Infine abilitiamo lo scugic con

```
XScuGic_Enable
```

Nella routine di gestione dell' interruzione :

1. Disabilitiamo l' interuzione scrivendo uno zero nel registro reg_intr_en;
2. Scriviamo un uno nel registro reg_intr_ack per dire al componente che l' interruzione è stata catturata;
3. Riabilitiamo le interruzioni sul componente;

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.
All'inserimento del modulo viene chiamata la funzione **Probe**, la quale si occupa dell'inizializzazione del driver. Quando invece il modulo viene rimosso viene chiamata la funzione **Remove**. Entrambe vengono devono essere ridefinite all'interno del modulo e si utilizza la struttura platform_driver per effettuare il matching.

```
/**
 * @brief Definisce le funzioni probe() e remove() da chiamare al caricamento del driver.
 */
static struct platform_driver GPIO_driver = {
  .driver = {
        .name = DRIVER_NAME,
        .owner = THIS_MODULE,
        .of_match_table = of_match_ptr(__test_int_driver_id),
    },
  .probe = GPIO_probe,
  .remove = GPIO_remove
};
```

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.

```
/**
 * @brief Identifica il device all'interno del device tree
 *
 */
static const struct of_device_id __test_int_driver_id[]={
  {.compatible = "GPIO"},
  {}
};
```

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.

```
/**
 * @brief Struttura che specifica le funzioni che agiscono sul device
 *
 */
static struct file_operations GPIO_fops = {
    .owner    = THIS_MODULE,
    .llseek   = GPIO_llseek,
    .read     = GPIO_read,
    .write    = GPIO_write,
    .poll     = GPIO_poll,
    .open     = GPIO_open,
    .release  = GPIO_rele
};
```

- owner: rappresenta puntatore al modulo che è il possessore della struttura. Ha lo scopo di evitare che il modulo venga rimosso quando uno delle funzionalià 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:

```
obj-m += my_kernel_GPIO.o
my_kernel_GPIO-objs :=GPIO_kernel_main.o GPIO.o GPIO_list.o

all:
  make -C linux-xlnx/ M=$(PWD) modules

clean:
  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:

```
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:
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 (distanziate 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 rimuove il modulo impartire il comando:

```
rmmod my_kernel_GPIO.ko
```

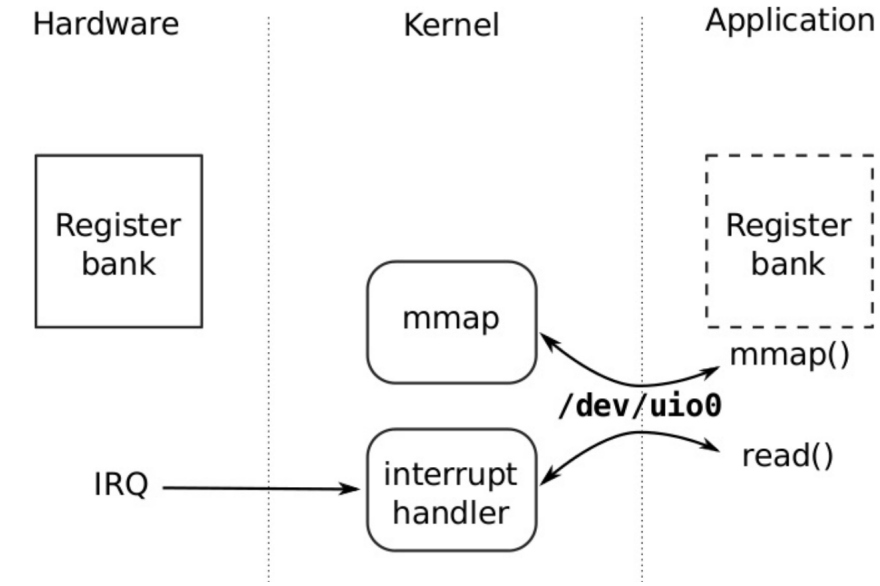
0.1.6.2 UIO

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:

```
/ {
  amba_pl: amba_pl {
    #address-cells = <1>;
    #size-cells = <1>;
    compatible = "simple-bus";
    ranges ;
    GPIO_0: GPIO@43c00000 {
      /* This is a place holder node for a custom IP, user may need to update the entries */
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "generic-uio";
      interrupt-names = "interrupt";
      interrupt-parent = <&intc>;
      interrupts = <0 29 4>;
      reg = <0x43c00000 0x10000>;
      xlnx,s00-axi-addr-width = <0x5>;
      xlnx,s00-axi-data-width = <0x20>;
    };
    GPIO_1: GPIO@43c10000 {
      /* This is a place holder node for a custom IP, user may need to update the entries */
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "generic-uio";
      interrupt-names = "interrupt";
      interrupt-parent = <&intc>;
      interrupts = <0 30 4>;
      reg = <0x43c10000 0x10000>;
      xlnx,s00-axi-addr-width = <0x5>;
      xlnx,s00-axi-data-width = <0x20>;
    };
    GPIO_2: GPIO@43c20000 {
      /* This is a place holder node for a custom IP, user may need to update the entries */
      clock-names = "s00_axi_aclk";
      clocks = <&clkc 15>;
      compatible = "generic-uio";
      interrupt-names = "interrupt";
      interrupt-parent = <&intc>;
      interrupts = <0 31 4>;
      reg = <0x43c20000 0x10000>;
      xlnx,s00-axi-addr-width = <0x5>;
      xlnx,s00-axi-data-width = <0x20>;
    };
```

```
  };
};
```

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.



Segue il codice relativo al driver UIO:

```c
#include <unistd.h>
#include <stdlib.h>
#include <stdio.h>
#include <limits.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <sys/mman.h>
#include <poll.h>
#include "GPIO_interrupt_uio_poll.h"

#define DIR_OFF      0  // DIRECTION
#define WRITE_OFF    4  // WRITE
#define READ_OFF     8  // READ
#define GLOBAL_INTR_EN  12 // GLOBAL INTERRUPT ENABLE
#define INTR_EN         16 // LOCAL INTERRUPT ENABLE
#define INTR_ACK_PEND   28 // PENDING/ACK REGISTER

#define INTR_MASK 15

#define TIMEOUT 2000

typedef u_int8_t u8;
typedef u_int32_t u32;

void write_reg(void *addr, unsigned int offset, unsigned int value)
{
  *((unsigned*)(addr + offset)) = value;
}

unsigned int read_reg(void *addr, unsigned int offset)
```

```
{
  return *((unsigned*)(addr + offset));
}


void wait_for_interrupt(int fd0, int fd1, int fd2, void *addr_0, void *addr_1, void *addr_2)
{

  int pending = 0;
  int reenable = 1;
  u32 read_value;
  struct pollfd poll_fds [3];
  int ret;

  printf("Waiting for interrupts....\n");

  poll_fds[0].fd = fd0;
  poll_fds[0].events = POLLIN;  //The field events is an input parameter, a bit mask specifying the
                  //events the application is interested in for the file descriptor fd.
                 //Means that we are interested at the event: there is data to read.
  poll_fds[1].fd = fd1;
  poll_fds[1].events = POLLIN;

  poll_fds[2].fd = fd2;
  poll_fds[2].events = POLLIN;

  // non blocking wait for an interrupt on file descriptors specified in the pollfd structure*/
  ret = poll(poll_fds, 3, TIMEOUT); //timeout of TIMEOUT ms
  if (ret > 0){
      if(poll_fds[0].revents && POLLIN){

        read(fd0, (void *)&pending, sizeof(int));
        write_reg(addr_0, GLOBAL_INTR_EN, 0); //disabilito interruzioni
        printf("**********ISR SWITCH**********\n");
        read_value = read_reg(addr_0, READ_OFF);
        printf("Read value: %08x\n", read_value);
        write_reg(addr_0, INTR_ACK_PEND, INTR_MASK); //ACK
        sleep(1);
        write_reg(addr_0, INTR_ACK_PEND, 0); //ACK
        write_reg(addr_0, GLOBAL_INTR_EN, 1); //abiito interruzioni
        write(fd0, (void *)&reenable, sizeof(int));

      }
      if(poll_fds[1].revents && POLLIN){

        read(fd1, (void *)&pending, sizeof(int));
        write_reg(addr_1, GLOBAL_INTR_EN, 0); //disabilito interruzioni
        printf("**********ISR BUTTON**********\n");
        read_value = read_reg(addr_1, READ_OFF);
        printf("Read value: %08x\n", read_value);
        write_reg(addr_1, INTR_ACK_PEND, INTR_MASK); //ACK
        sleep(1);
        write_reg(addr_1, INTR_ACK_PEND, 0); //ACK
        write_reg(addr_1, GLOBAL_INTR_EN, 1); //abiito interruzioni
        write(fd1, (void *)&reenable, sizeof(int));

      }
      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 desctiptor: 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.