

High Performance Computing

01. Xilinx Vivado HLx Design Suite

Gianluca Brilli
(Gianluca.Brilli@Unimore.it)
AA 2018-2019

FPGA Design Overview



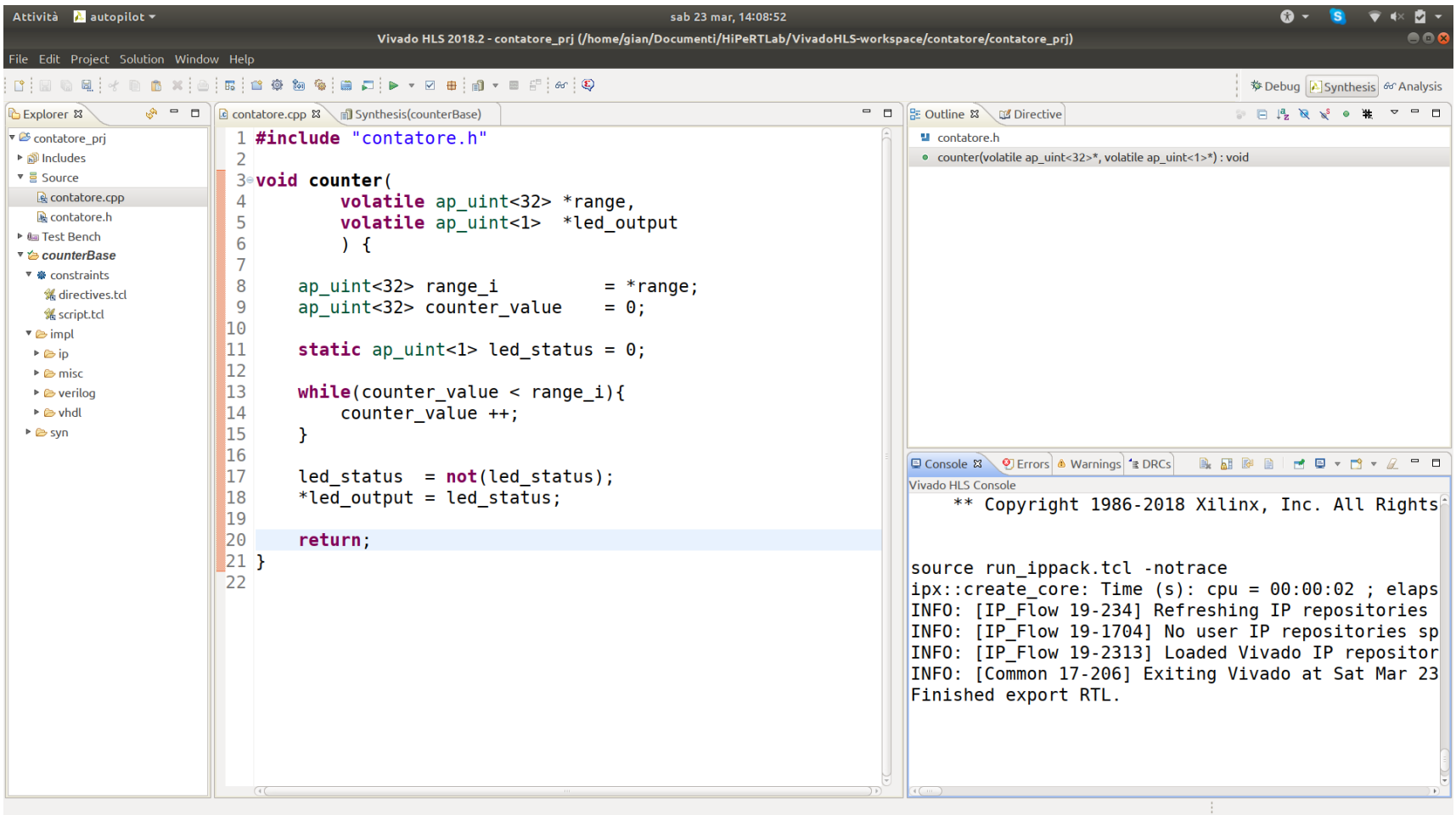
- Strumento principale sviluppato da Xilinx, utilizzato per la programmazione FPGA.
- Suite composta essenzialmente da tre tool:

- Vivado HLS
- Vivado HLX
- Xilinx SDK



Vivado HLS - Interfaccia

- Una versione di Eclipse modificata.



Vivado HLS - Interfaccia

The screenshot displays the Vivado HLS 2018.2 interface. The main editor shows the C++ source file `contatore.cpp` with the following code:

```
1 #include "contatore.h"
2
3 void counter(
4     volatile ap_uint<32> *range,
5     volatile ap_uint<1> *led_output
6 ) {
7
8     ap_uint<32> range_i = *range;
9     ap_uint<32> counter_value = 0;
10
11     static ap_uint<1> led_status = 0;
12
13     while(counter_value < range_i){
14         counter_value ++;
15     }
16
17     led_status = not(led_status);
18     *led_output = led_status;
19
20     return;
21 }
22
```

A red arrow points to the `void counter` function signature. A large red text overlay reads: **Top Function, è la funzione che sarà convertita in un modulo hardware, i parametri della funzione saranno i segnali di ingresso e uscita che ritroveremo nell'IP sintetizzato.**

The right-hand side of the interface shows the `Outline` pane with the function signature `counter(volatile ap_uint<32>*, volatile ap_uint<1>*) : void` and the `Console` pane displaying the following output:

```
Vivado HLS Console
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
ipx::create_core: Time (s): cpu = 00:00:02 ; elaps
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories sp
INFO: [IP_Flow 19-2313] Loaded Vivado IP repositor
INFO: [Common 17-206] Exiting Vivado at Sat Mar 23
Finished export RTL.
```

Vivado HLS - Interfaccia

The screenshot displays the Vivado HLS 2018.2 interface. The main editor shows the C++ source file `contatore.cpp` with the following code:

```
1 #include "contatore.h"
2
3 void counter(
4     volatile ap_uint<32> *range,
5     volatile ap_uint<1> *led_output
6 ) {
7     // Sintesi, esegue la traduzione da codice HLS
8     // in codice HDL.
9     ap_uint<32> range_i = *range;
10    ap_uint<32> counter_value = 0;
11
12    static ap_uint<1> led_status = 0;
13
14    while(counter_value < range_i){
15        counter_value ++;
16
17        led_status = not(led_status);
18        *led_output = led_status;
19
20    }
21 }
22
```

The synthesis button (a play icon) in the top toolbar is highlighted with a red arrow. The console window at the bottom right shows the output of the synthesis process:

```
Vivado HLS Console
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
ipx::create_core: Time (s): cpu = 00:00:02 ; elaps
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories sp
INFO: [IP_Flow 19-2313] Loaded Vivado IP repositor
INFO: [Common 17-206] Exiting Vivado at Sat Mar 23
Finished export RTL.
```

Vivado HLS - Interfaccia

The screenshot displays the Vivado HLS 2018.2 interface. The main editor shows the C++ source file `contatore.cpp` with the following code:

```
1 #include "contatore.h"
2
3 void counter(
4     volatile ap_uint<32> *range,
5     volatile ap_uint<1> *led_output
6 ) {
7
8     ap_uint<32> range_i = *range;
9     ap_uint<32> counter_value = 0;
10
11     static ap_uint<1> led_status = 0;
12
13     while(counter_value < range_i){
14         counter_value ++;
15     }
16
17     led_status = not(led_status);
18     *led_output = led_status;
19
20     return;
21 }
22
```

A red arrow points to the `return;` statement on line 20. A large red text overlay reads: **Export RTL, permette di esportare il codice Verilog / VHDL generato e creare un pacchetto importabile da Vivado.**

The console window at the bottom right shows the following output:

```
Vivado HLS Console
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
ipx::create_core: Time (s): cpu = 00:00:02 ; elaps
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories sp
INFO: [IP_Flow 19-2313] Loaded Vivado IP repositor
INFO: [Common 17-206] Exiting Vivado at Sat Mar 23
Finished export RTL.
```

Vivado HLS - Interfaccia

The screenshot displays the Vivado HLS 2018.2 interface. The main window shows the C++ source file `contatore.cpp` with the following code:

```
1 #include "contatore.h"
2
3 void counter(
4     volatile ap_uint<32> *range,
5     volatile ap_uint<1> *led_output
6 ) {
7
8     ap_uint<32> range_i = *range;
9     ap_uint<32> counter_value = 0;
10
11     static ap_uint<1> led_status = 0;
12
13     while(counter_value < range_i){
14         counter_value++;
15     }
16
17     led_status = not(led_status);
18     *led_output = led_status;
19
20     return;
21 }
22
```

The file explorer on the left shows the project structure, including the `Test Bench` directory. The outline window on the right shows the function signature: `counter(volatile ap_uint<32>*, volatile ap_uint<1>): void`. The console window at the bottom right displays the following output:

```
** Copyright 1986-2018 Xilinx, Inc. All Rights Reserved.

source run_ippack.tcl -notrace
ipx::create_core: Time (s): cpu = 00:00:02 ; elaps
INFO: [IP_Flow 19-234] Refreshing IP repositories
INFO: [IP_Flow 19-1704] No user IP repositories sp
INFO: [IP_Flow 19-2313] Loaded Vivado IP repositor
INFO: [Common 17-206] Exiting Vivado at Sat Mar 23
Finished export RTL.
```

Two red arrows point from the text to the `Test Bench` directory in the file explorer and the `void counter` function definition in the code editor.

Permette di eseguire un TestBench, ovvero una funzione main che va a richiamare la top function che abbiamo scritto.

Vivado HLS – Report

General Information

Date: Thu Mar 7 21:50:46 2019
Version: 2018.2 (Build 2258646 on Thu Jun 14 20:25:20 MDT 2018)
Project: Convolution_prj
Solution: solution1
Product family: zynq
Target device: xc7z020clg484-1

Performance Estimates

Timing (ns)

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00	9.634	1.25

Latency (clock cycles)

Summary

Latency		Interval		
min	max	min	max	Type
844870	844870	844870	844870	none

Detail

Instance

Loop

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	9	-	-
Expression	-	-	0	784
FIFO	-	-	-	-
Instance	6	-	1680	2140
Memory	128	-	0	0
Multiplexer	-	-	-	623
Register	0	-	1059	192
Total	134	9	2739	3739
Available	280	220	106400	53200
Utilization (%)	47	4	2	7

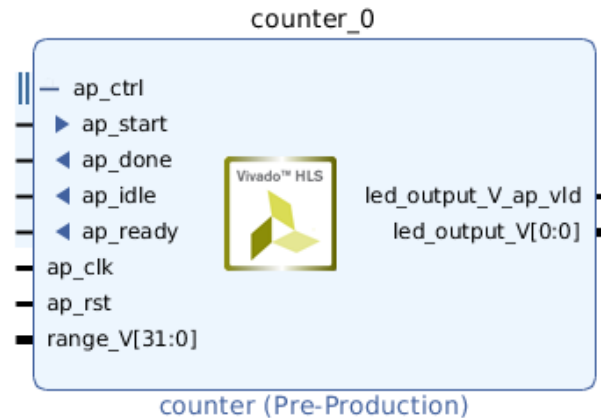
- Generato in automatico dopo la fase di sintesi, ci fornisce alcune informazioni utili sul modulo hardware generato.

Numero di cicli di clock necessari ad eseguire il modulo sintetizzato

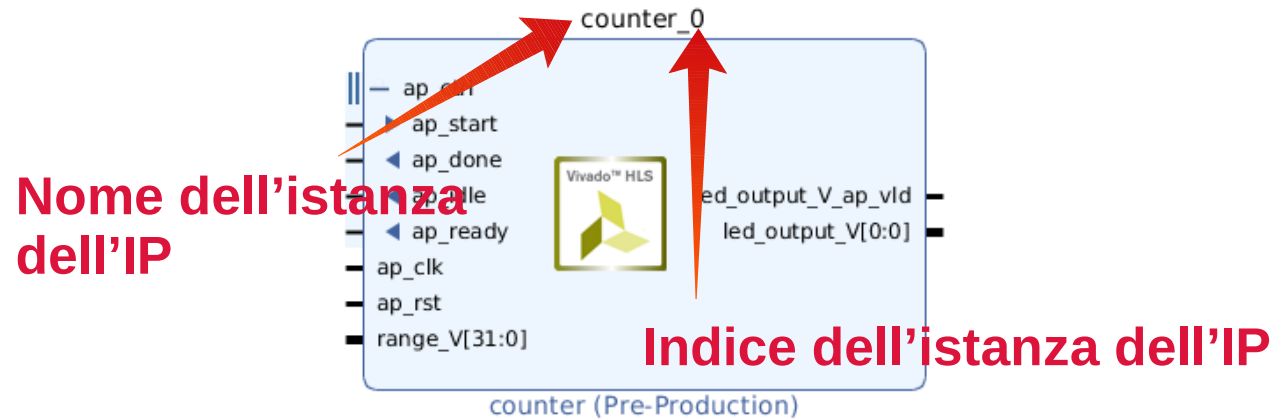
Utilizzo di risorse dell'FPGA, in termini di BRAM, DSP, Flip-Flops e Look-Up-Tables (LUTs)

Vivado HLS – Struttura di un IP

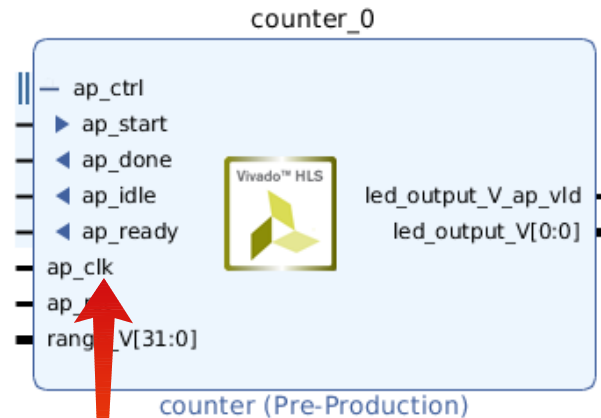
- Un IP generato con Vivado HLS oltre alle porte di ingresso e uscita ha una struttura standard.
- Analizziamo insieme la struttura dell'IP.



Vivado HLS – Struttura di un IP

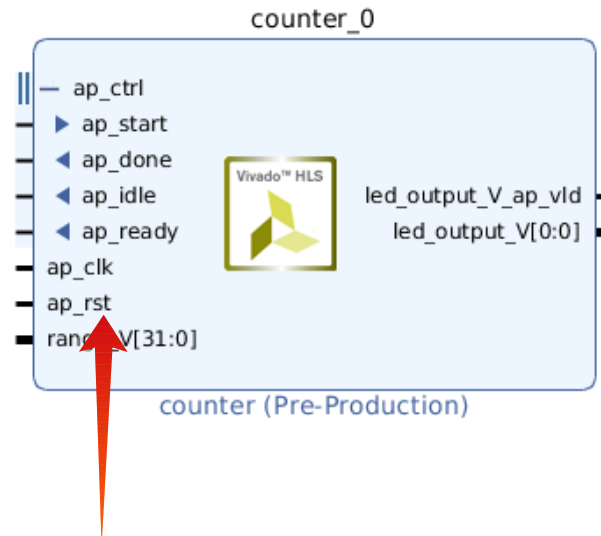


Vivado HLS – Struttura di un IP



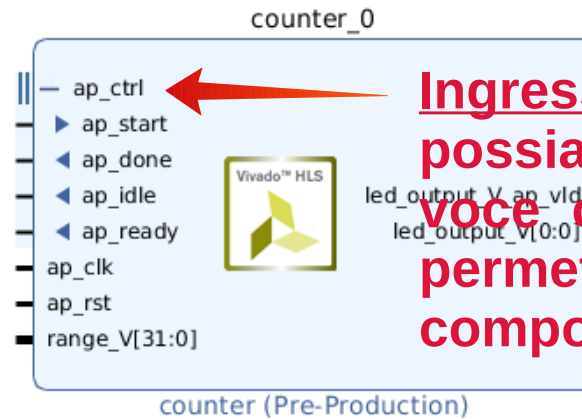
Clock in ingresso, deve essere collegato ad una sorgente esterna. Tipicamente si usa un clock a 100MHz.

Vivado HLS – Struttura di un IP



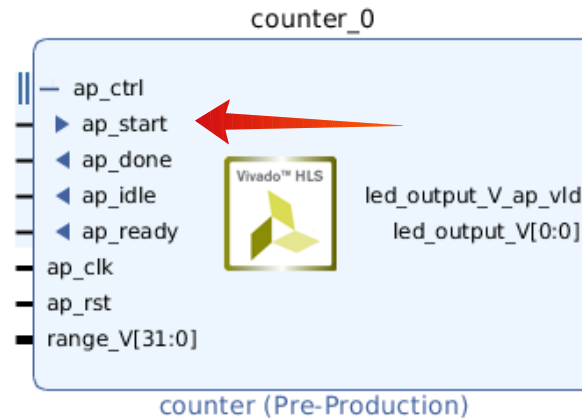
Reset attivo basso, Indispensabile per ogni logica sequenziale porta lo stato del modulo ad un valore noto.

Vivado HLS – Struttura di un IP



Ingressi di controllo,
possiamo espandere la
voce cliccando sul “+”,
permette di controllare il
comportamento dell’IP.

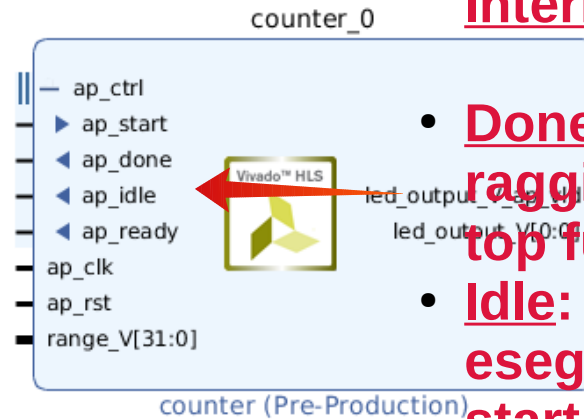
Vivado HLS – Struttura di un IP



**Bit di start, se
settato al livello
logico "1", il modulo
inizia ad eseguire.**

Vivado HLS – Struttura di un IP

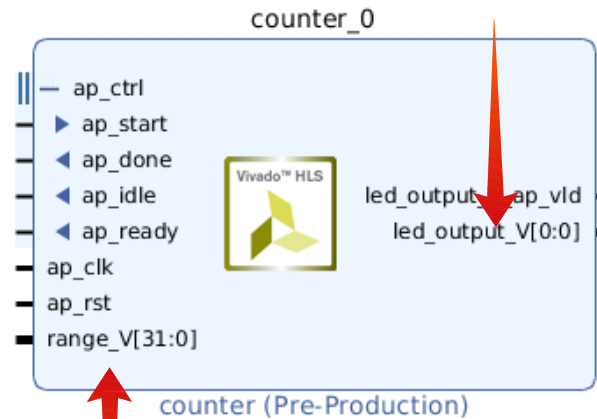
Bit di stato, indicano lo stato dell'IP e possono essere usati come Interrupt.



- Done: Il modulo ha raggiunto il termine della top function.
- Idle: Il modulo non sta eseguendo, (segnale di start non settato).
- Ready: Il modulo è pronto per ri eseguire la propria top function.

Vivado HLS – Struttura di un IP

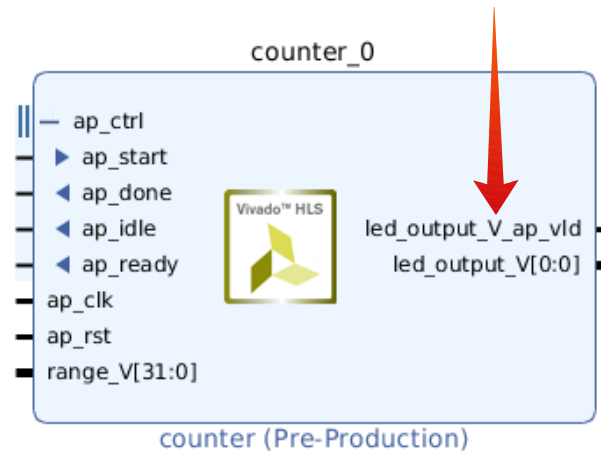
Output, che abbiamo specificato in Vivado HLS.



Input, che abbiamo specificato in Vivado HLS.

Vivado HLS – Struttura di un IP

**Bit di validità, quando è “1”
l’uscita contiene un valore
valido (non indispensabile).**



Vivado HLx – Block Design

The screenshot displays the Vivado 2018.2 Block Design environment. The interface is divided into several panes:

- Flow Navigator:** Shows the project structure with sections for PROJECT MANAGER, IP INTEGRATOR, SIMULATION, RTL ANALYSIS, SYNTHESIS, IMPLEMENTATION, and PROGRAM AND DEBUG.
- Source:** Lists design sources including design_1, design_1_exercise_0_0, design_1_clk_wiz_0, design_1_rst_clk_wiz_100M_0, exercise, and constraints.
- Diagram:** A block diagram showing the interconnection of components: 'reset_rtl_0' and 'reset_rtl_1' connect to 'clk_wiz' (Clocking Wizard). 'clk_wiz' outputs 'clk_out1' to 'rst_clk_wiz_100M' (Processor System Reset). 'rst_clk_wiz_100M' outputs 'mb_reset' and 'interconnect_aresetn[0:0]' to 'exercise_0' (exercise_v1_0). 'exercise_0' has a 'ready' output.
- Properties:** A pane for selecting an object to view its properties.
- Tcl Console:** Shows the command history and execution results, including commands like 'update_compile_order -fileset sources_1', 'open_bd_design', 'add_cell', and 'open_bd_design: Time (s): cpu = 00:00:10; elapsed = 00:00:12 . Memory (MB): peak = 6278.863; gain = 5.637; free physical = 1898; free virtual = 8703'.

A red arrow points to the diagram area, highlighting the graphical interconnection of hardware components.

Permette la creazione di uno schema a blocchi andando a interconnettere tra loro diversi componenti hardware per via grafica.

Vivado HLx - IP Catalog

The screenshot displays the Vivado 2018.2 software interface. On the left, the 'IP INTEGRATOR' tab is selected in the 'PROJECT MANAGER' pane. The 'BLOCK DESIGN - design_1' window shows a hierarchy of design sources, including 'exercise' and 'rst_clk_wiz_100M'. The 'Diagram' window shows a block diagram with components like 'Processor System Reset' and 'exercise_0'. The 'Tcl Console' at the bottom shows the following commands and output:

```
update_compile_order -fileset sources_1
open_bd_design {/home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd}
Adding cell -- xilinx.com:module_ref:exercise:1.0 -- exercise_0
Adding cell -- xilinx.com:ip:clk_wiz:6.0 -- clk_wiz
Adding cell -- xilinx.com:ip:proc_sys_reset:5.0 -- rst_clk_wiz_100M
Successfully read diagram <design_1> from BD file </home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd>
open_bd_design: Time (s): cpu = 00:00:10 ; elapsed = 00:00:12 . Memory (MB): peak = 6278.863 ; gain = 5.637 ; free physical = 1898 ; free virtual = 8703
```

Two red arrows point from the 'IP INTEGRATOR' tab and the 'exercise' component in the hierarchy to the red text overlay.

Lista di tutte le IP disponibili, può essere visto come l'equivalente di un insieme di librerie software.

Vivado HLx – Struttura Progetto

The screenshot displays the Vivado 2018.2 interface. On the left, the 'Flow Navigator' shows the project structure under 'BLOCK DESIGN - design_1'. The 'Design Sources' section lists: design_1 (design_1.bd) (3), design_1_exercise_0_0 (Module Reference), design_1_clk_wiz_0 (design_1_clk_wiz_0.v), design_1_rst_clk_wiz_100M_0 (design_1_rst_clk_wiz_100M.v), and exercise (exercise.v). The 'Simulation Sources' section lists: sim_1 (2), design_1 (design_1.bd) (3), and exercise (exercise.v). A red arrow points from the 'exercise' module in the Design Sources to its corresponding block in the diagram.

The 'Diagram' window shows a block diagram with the following components and connections:

- reset_rtl_0** and **reset_rtl_1** connect to the **reset** input of the **clk_wiz** block.
- sys_clock** connects to the **clk_in1** input of the **clk_wiz** block.
- The **clk_wiz** block has two outputs: **clk_out1** and **locked**.
- The **locked** output of **clk_wiz** connects to the **dcm_locked** input of the **rst_clk_wiz_100M** block.
- The **clk_out1** output of **clk_wiz** connects to the **slowest_sync_clk** input of the **rst_clk_wiz_100M** block.
- The **rst_clk_wiz_100M** block has several outputs: **mb_reset**, **bus_struct_reset[0:0]**, **peripheral_reset[0:0]**, **interconnect_aresetn[0:0]**, and **peripheral_aresetn[0:0]**.
- The **exercise_0** block has two inputs: **start** and **clk**.
- The **mb_reset** output of **rst_clk_wiz_100M** connects to the **start** input of **exercise_0**.
- The **peripheral_reset[0:0]** output of **rst_clk_wiz_100M** connects to the **clk** input of **exercise_0**.
- The **exercise_0** block has an output **ready**.

The 'Tcl Console' at the bottom shows the following commands and output:

```
update_compile_order -fileset sources_1
open_bd_design {/home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd}
Adding cell -- xilinx.com:module_ref:exercise:1.0 - exercise_0
Adding cell -- xilinx.com:ip:clk_wiz:6.0 - clk_wiz
Adding cell -- xilinx.com:ip:proc_sys_reset:5.0 - rst_clk_wiz_100M
Successfully read diagram <design_1> from EDI file </home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd>
open_bd_design: Time (s): cpu = 00:00:10 ; elapsed = 00:00:12 . Memory (MB): peak = 6278.863 ; gain = 5.637 ; free physical = 1898 ; free virtual = 8703
```

Qui sono elencati tutti i file di cui il progetto si compone. E' presente un file che descrive i componenti istanziati dal block design (HDL Wrapper). Sono inoltre presenti anche file constraints.

Vivado HLx - Sintesi

The screenshot displays the Vivado 2018.2 interface during the synthesis phase. The top status bar indicates 'Synthesis Complete'. The left sidebar shows the 'Run Synthesis' option under the 'SYNTHESIS' section, highlighted with a red arrow. The center pane shows a block diagram of the design, including a 'Clocking Wizard' (clk_wiz), a 'Processor System Reset' (rst_clk_wiz_100M), and an 'exercise_0' block. The bottom pane shows the Tcl Console with the following log output:

```
update_compile_order -fileset sources_1
-- design -- /home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd
Adding cell -- xilinx.com:module_ref:exercise_0:exercise_0
Adding cell -- xilinx.com:ip:clk_wiz:clk_wiz_100M:clk_wiz_100M
Adding cell -- xilinx.com:ip:processor_system_reset:rst_clk_wiz_100M:rst_clk_wiz_100M
Successfully read diagram <design_1> from EDI file </home/gian/Documents/HiPeRTLab/Vivado-workspace/test/test.srcs/sources_1/bd/design_1/design_1.bd>
open_bd_design: Time (s): cpu = 00:00:10 ; elapsed = 00:00:12 . Memory (MB): peak = 6278.863 ; gain = 5.637 ; free physical = 1898 ; free virtual = 8703
```

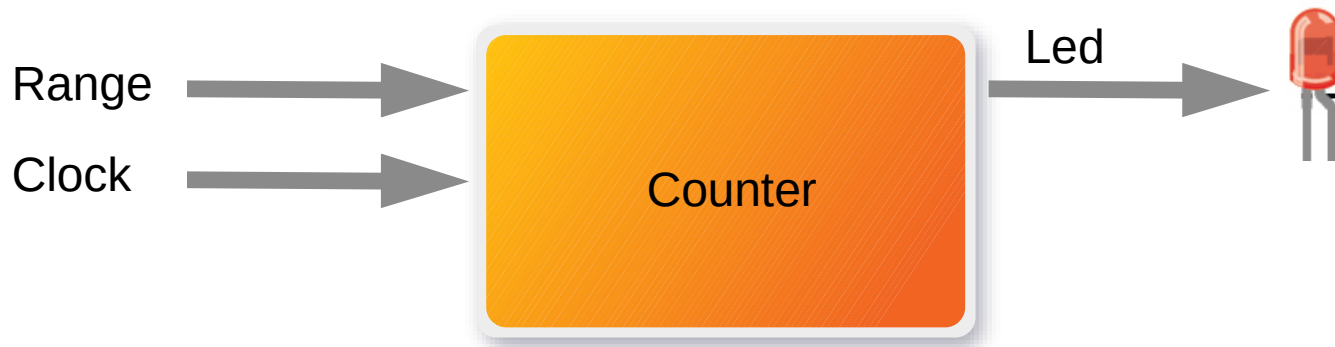
Avviamo rispettivamente la fase di sintesi, implementazione e generazione del bitstream. Che corrispondono alla generazione della Netlist, al piazzamento del circuito in FPGA e alla generazione del Bitstream.

Vivado HLx - Hardware Manager

The screenshot displays the Vivado 2018.2 software interface. The top menu bar includes File, Edit, Flow, Tools, Reports, Window, Layout, View, and Help. The Flow Navigator on the left shows the project structure, including PROJECT MANAGER, IP INTEGRATOR, SIMULATION, RTL ANALYSIS, SYNTHESIS, IMPLEMENTATION, and PROGRAM AND DEBUG. The Source window shows the design hierarchy, and the Diagram window displays a block diagram with components like 'clk_wiz', 'rst_clk_wiz_100M', and 'exercise_0'. The Tcl Console at the bottom shows the command 'open_bd_design' and its output. A red arrow points from the Tcl Console to the 'Open Hardware Manager' option in the Flow Navigator. A large red text overlay reads 'Permette la programmazione dell'FPGA.'

HLS – Esercitazione 01

- Provare a realizzare in FPGA un modulo contatore che dopo N cicli di clock vada ad invertire lo stato di un led, mappato sulla ZedBoard.



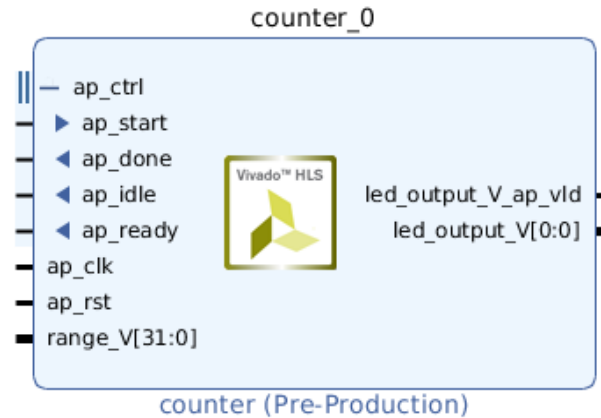
HLS – Esercitazione 01

- Provare a sintetizzare il seguente codice C:

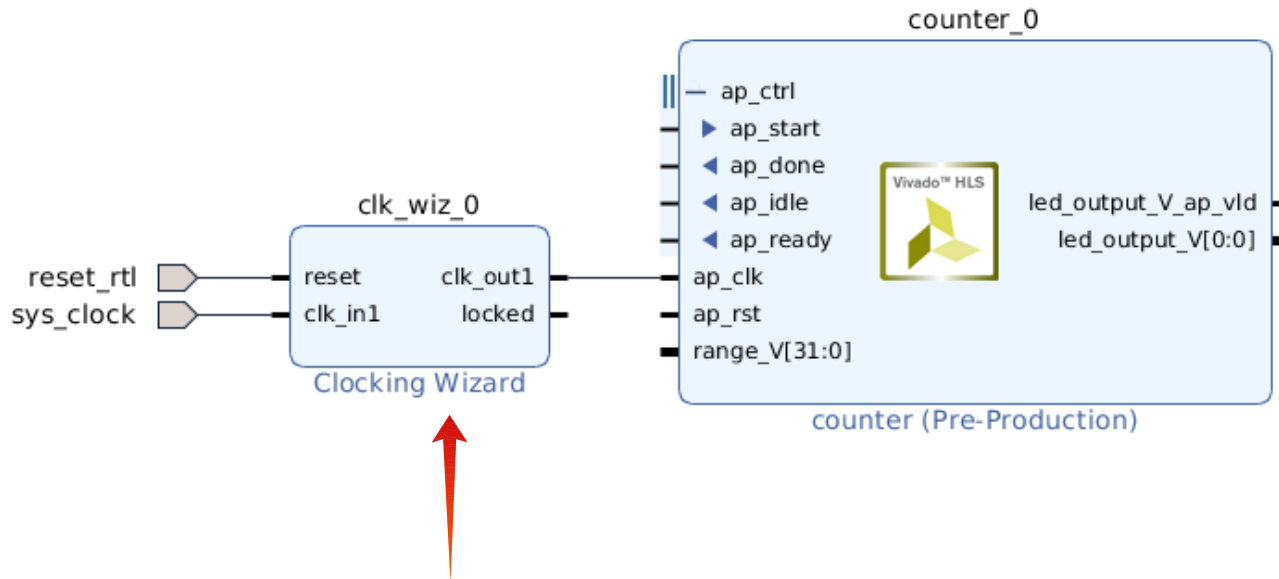
```
1
2 void counter(
3     volatile int range,
4     volatile bool *led_out) {
5
6     static bool led_status = 0;
7     volatile int temp_count = 0;
8
9     while (temp_count < range) {
10        temp_count = temp_count + 1;
11    }
12
13    led_status = not(led_status);
14
15    *led_out = led_status;
16 }
```


Creazione Block Design

- Andiamo a realizzare lo schema a blocchi del nostro sistema a partire dall'IP che abbiamo creato.



Creazione Block Design - Clock



Clocking Wizard: fornisce un uscita di clock configurabile che possiamo utilizzare nel nostro design.

Creazione Block Design - Clock

- Configuriamo il Clocking Wizard, in modo tale che abbia la più bassa frequenza possibile.
- In questo modo vediamo più chiaramente gli effetti sul led.

Component Name: clk_wiz_0

Board | Clocking Options | **Output Clocks** | MMCM Settings | Summary

The phase is calculated relative to the active input clock.

Output Clock	Port Name	Output Freq (MHz) Requested	Actual	Phase (degrees) Requested	Actual	Duty Cycle (%) Requested
<input checked="" type="checkbox"/>	clk_out1	5	5.000	0.000	0.000	50.000
<input type="checkbox"/>	clk_out2	100.000	N/A	0.000	N/A	50.000
<input type="checkbox"/>	clk_out3	100.000	N/A	0.000	N/A	50.000
<input type="checkbox"/>	clk_out4	100.000	N/A	0.000	N/A	50.000
<input type="checkbox"/>	clk_out5	100.000	N/A	0.000	N/A	50.000
<input type="checkbox"/>	clk_out6	100.000	N/A	0.000	N/A	50.000
<input type="checkbox"/>	clk_out7	100.000	N/A	0.000	N/A	50.000

USE CLOCK SEQUENCING

Clocking Feedback

Output Clock	Sequence Number
clk_out1	1
clk_out2	1
clk_out3	1
clk_out4	1
clk_out5	1
clk_out6	1
clk_out7	1

Source

Automatic Control On-Chip
 Automatic Control Off-Chip
 User-Controlled On-Chip
 User-Controlled Off-Chip

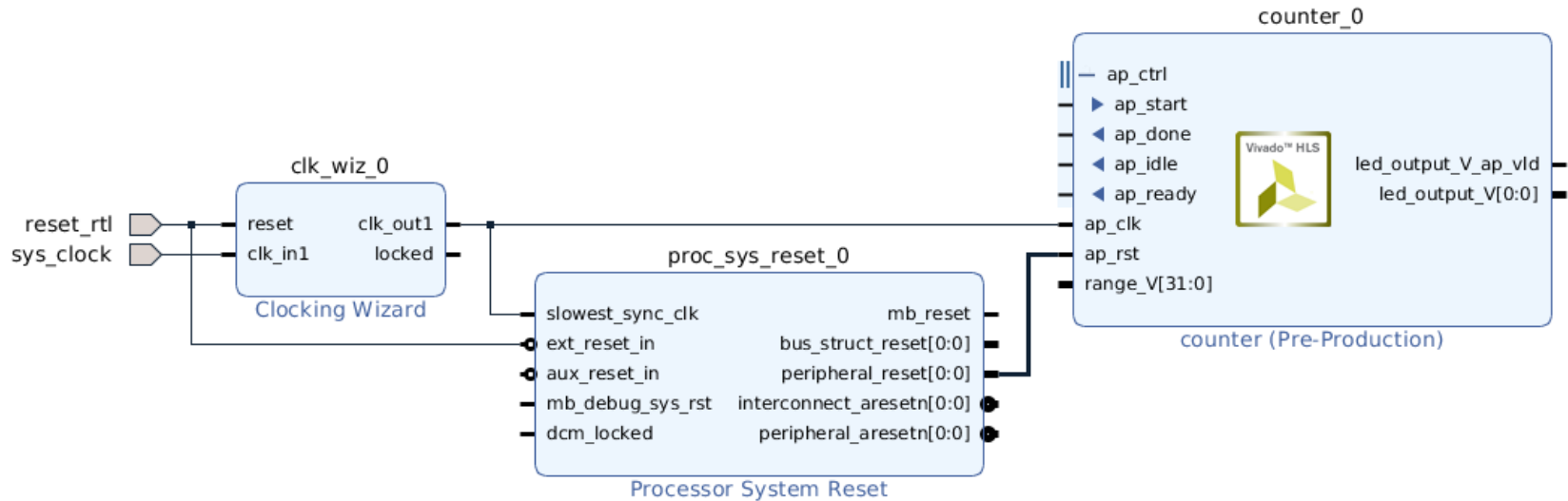
Signaling

Single-ended
 Differential

OK Cancel

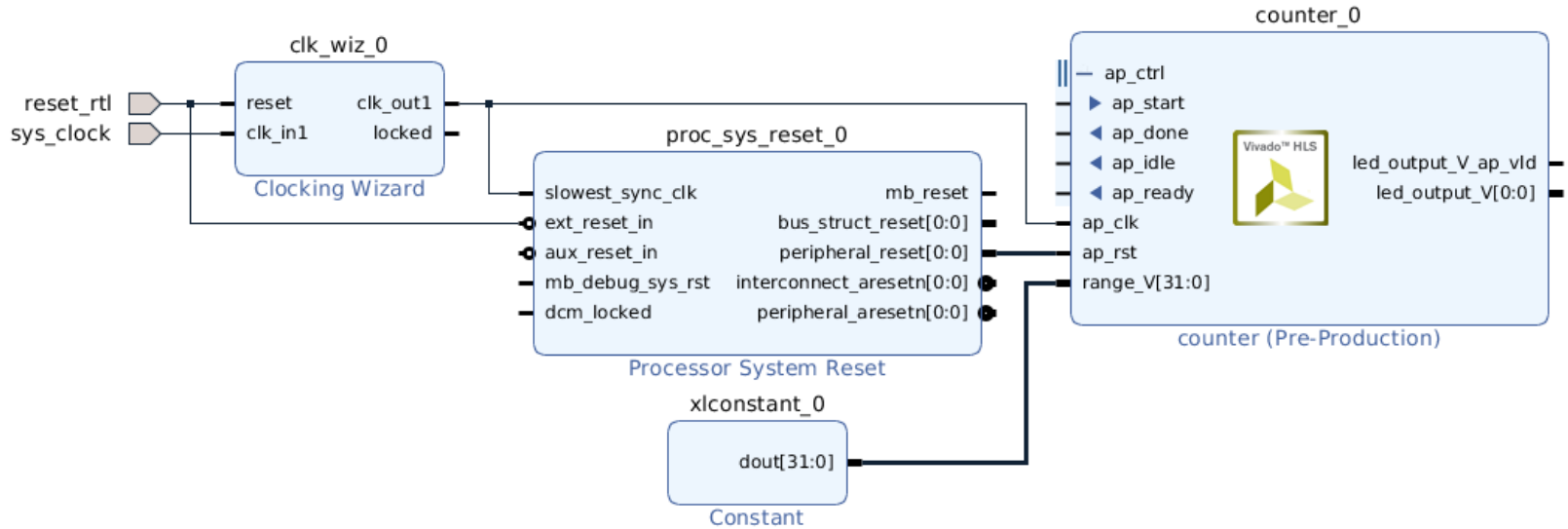
- Impostiamo un clock molto basso, in modo da veder lampeggiare il led.
- Ad esempio 5 MHz.

Creazione Block Design - Reset



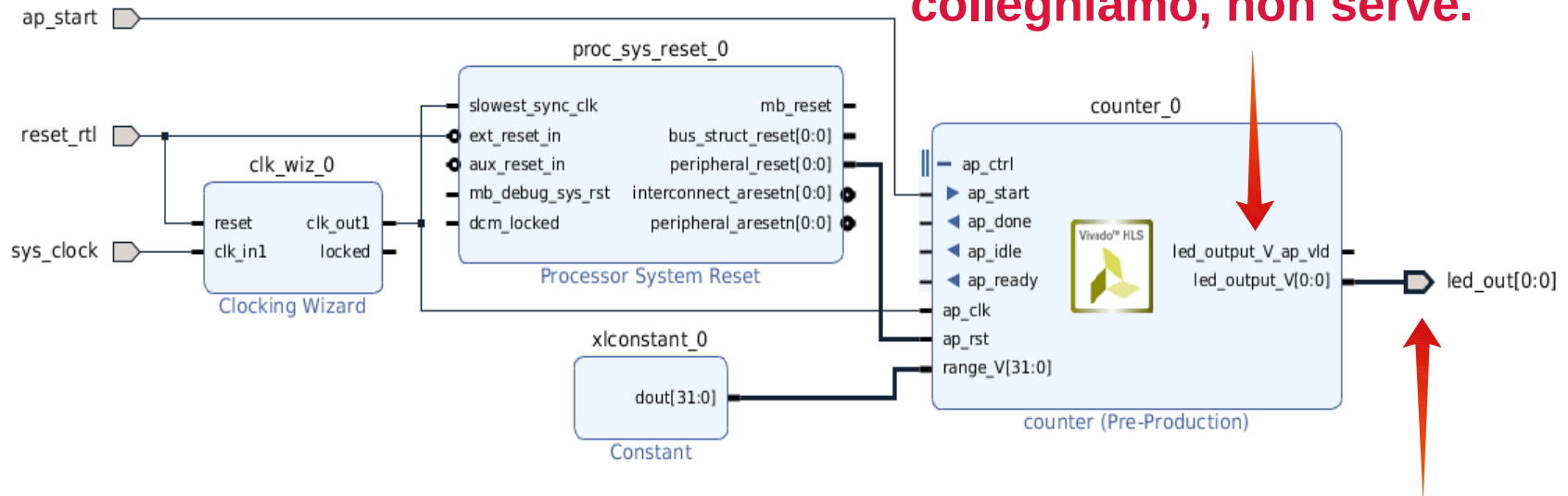
Processor System Reset: gestisce il reset della PS e delle periferiche IP.

Creazione Block Design - Range



Constant: permette di cablare un valore fissato nel design.

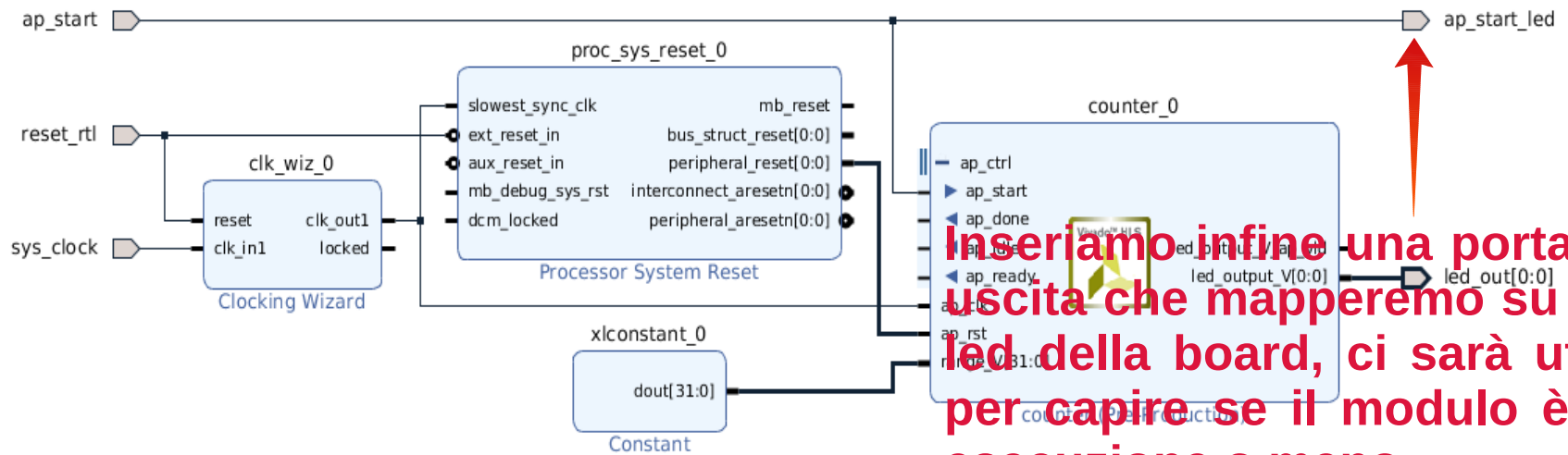
Creazione Block Design - Output



Bit di validità: Non lo colleghiamo, non serve.

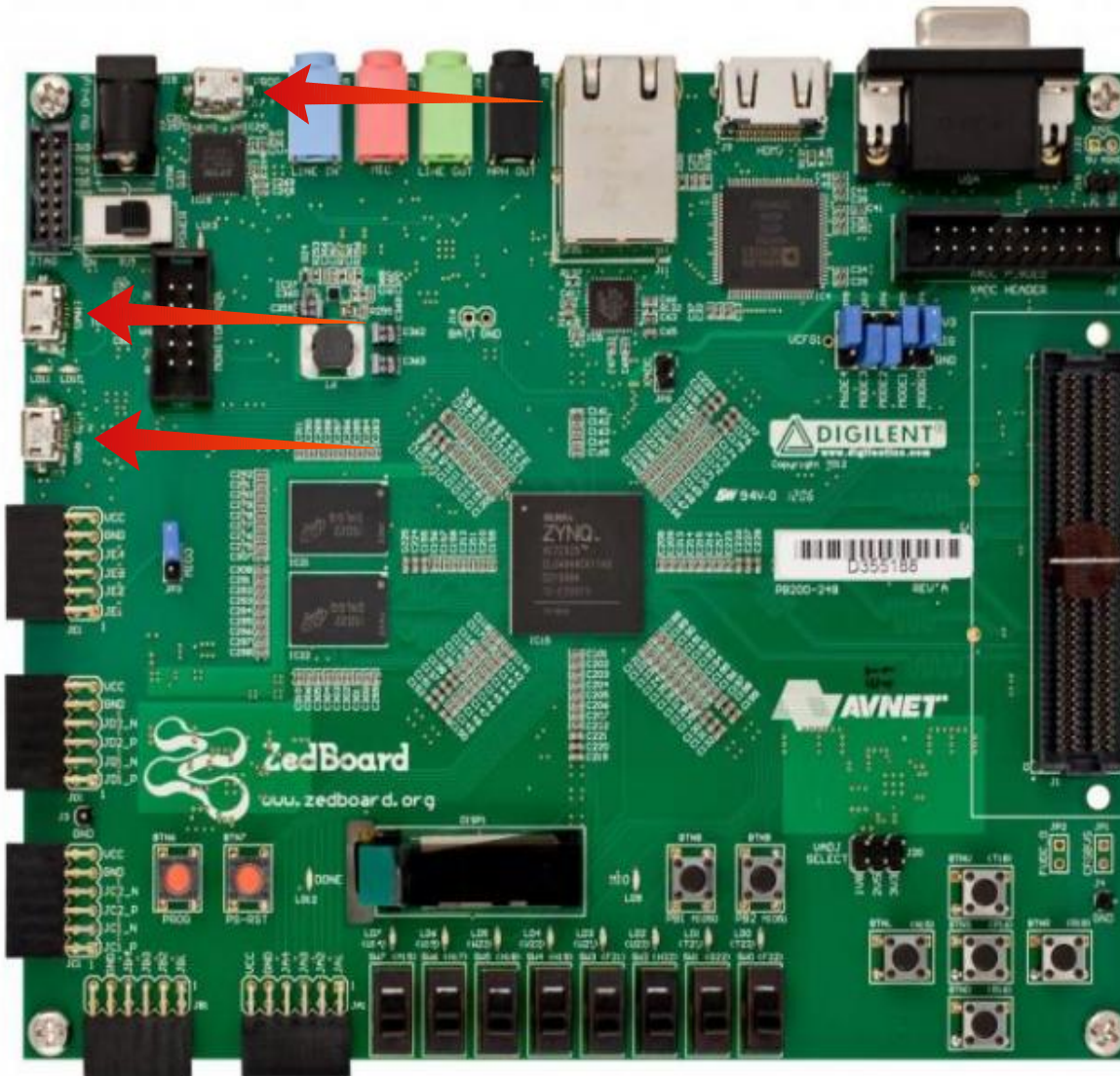
Output: mappiamo l'uscita su un led della ZedBoard.

Creazione Block Design - Output



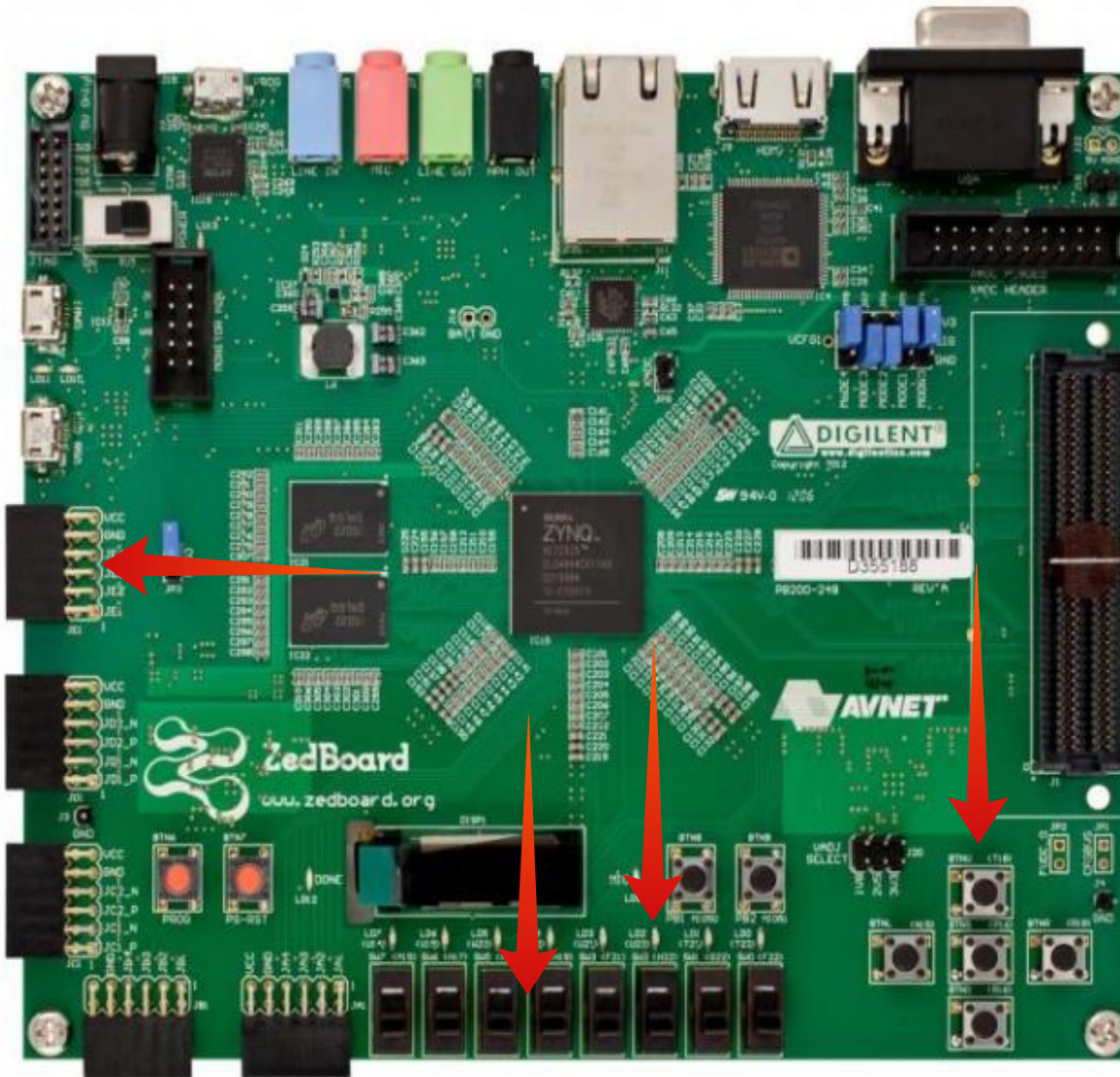
Inseriamo infine una porta di uscita che mapperemo su un led della board, ci sarà utile per capire se il modulo è in esecuzione o meno.

ZedBoard - Interfacce USB



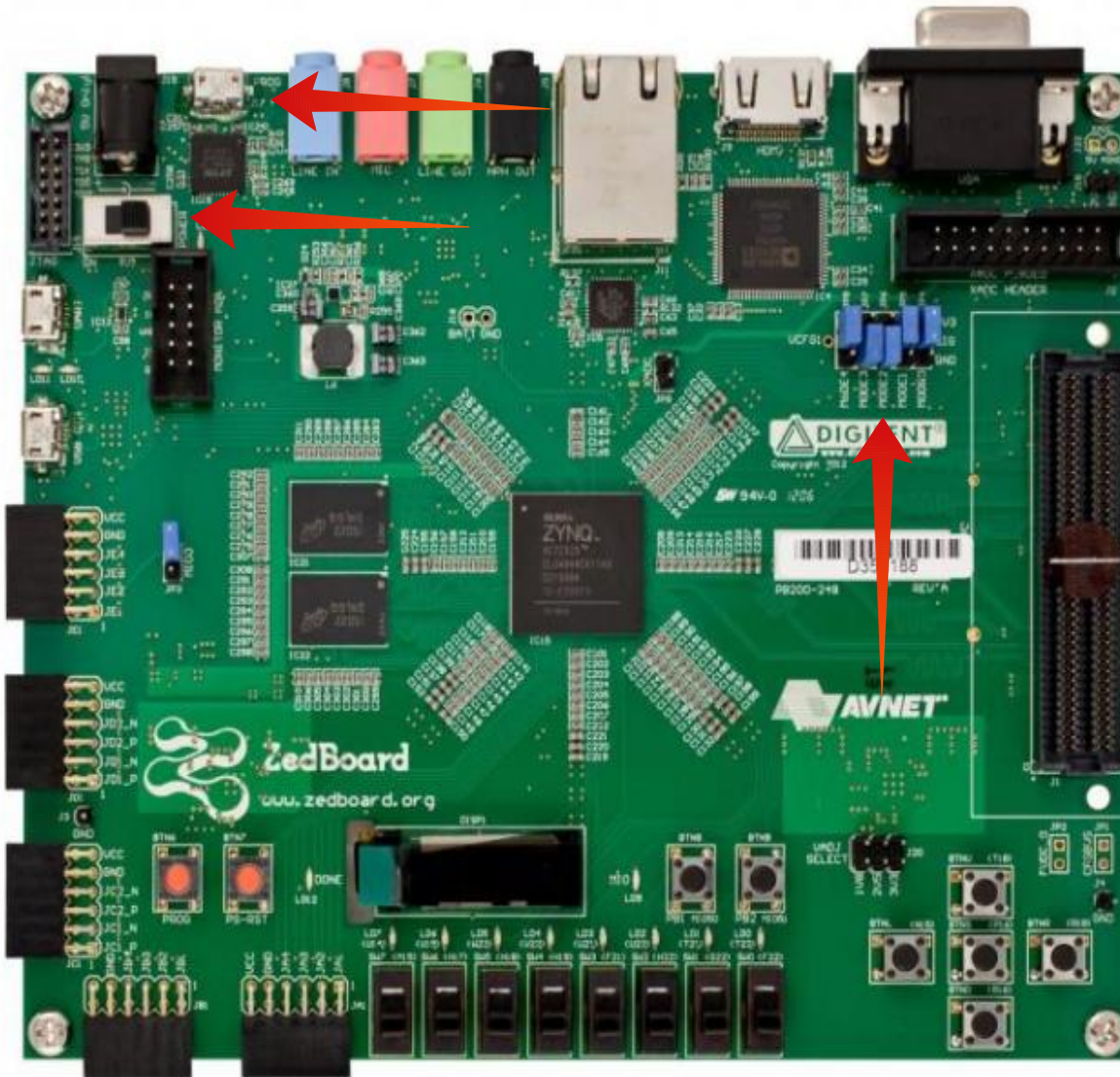
- 1. USB-JTAG:** permette di trasferire il Bitstream e di leggere il contenuto della DDR.
- 2. USB-UART:** permette di ricevere e trasmettere dati in seriale.
- 3. USB-OTG:** permette di collegare delle periferiche esterne.

ZedBoard – IO Programmabili



1. Pmod: ingressi e uscite programmabili classiche GPIO.
2. Switch: interruttori che programmabili come ingressi.
3. Pulsanti: altri input programmabili.
4. Led: possibile utilizzarli come uscite generiche.

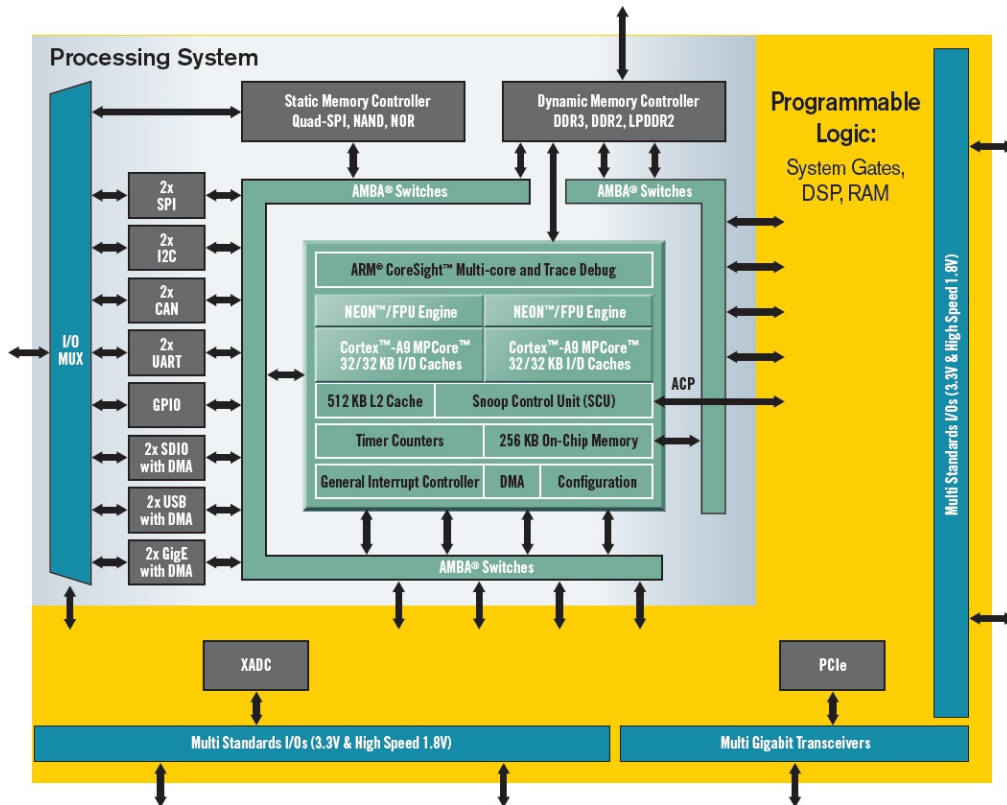
ZedBoard – Boot (JTAG)



1. Collegare il cavo JTAG al proprio computer.
2. Portare i 5 jumper nella posizione bassa (come il secondo e il terzo).
3. Accendere la board.

Hardware-Software Codesign

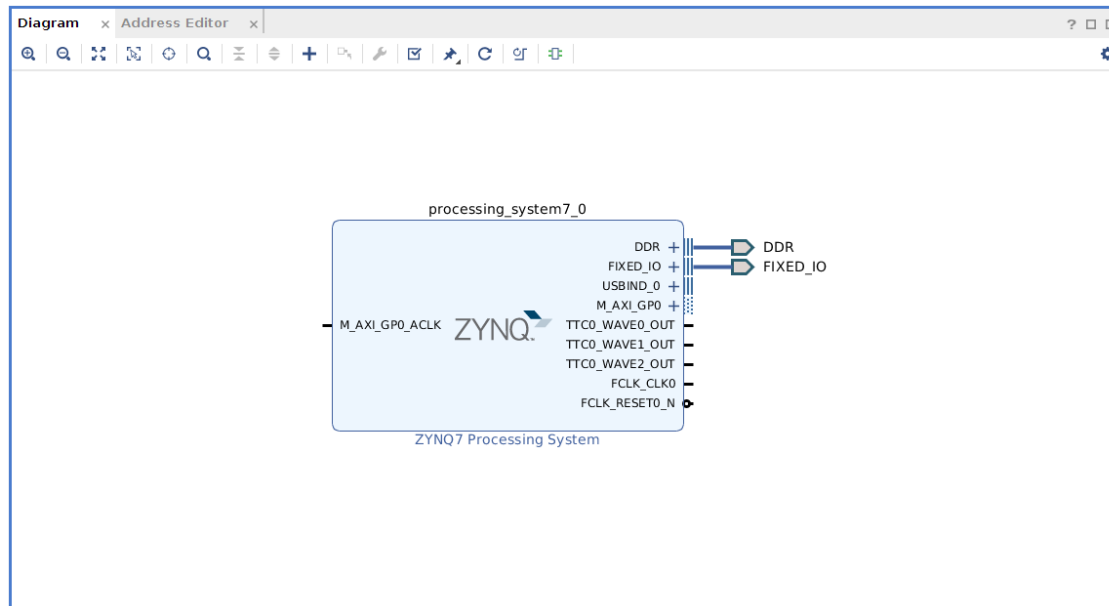
- Nei dispositivi FPGA-based moderni oltre alla logica programmabile sono presenti diverse entità: Cluster di CPU, DSPs, moduli ASIC ecc.



- **PL:** Programmable Logic;
- **PS:** Processing System.
- Queste due entità possono comunicare per mezzo di un protocollo definito da ARM: **AMBA AXI**.
- Disponibile in diverse versioni: **AXI Lite**, **AXI Full**, **AXI Stream**.

Configurazione PS

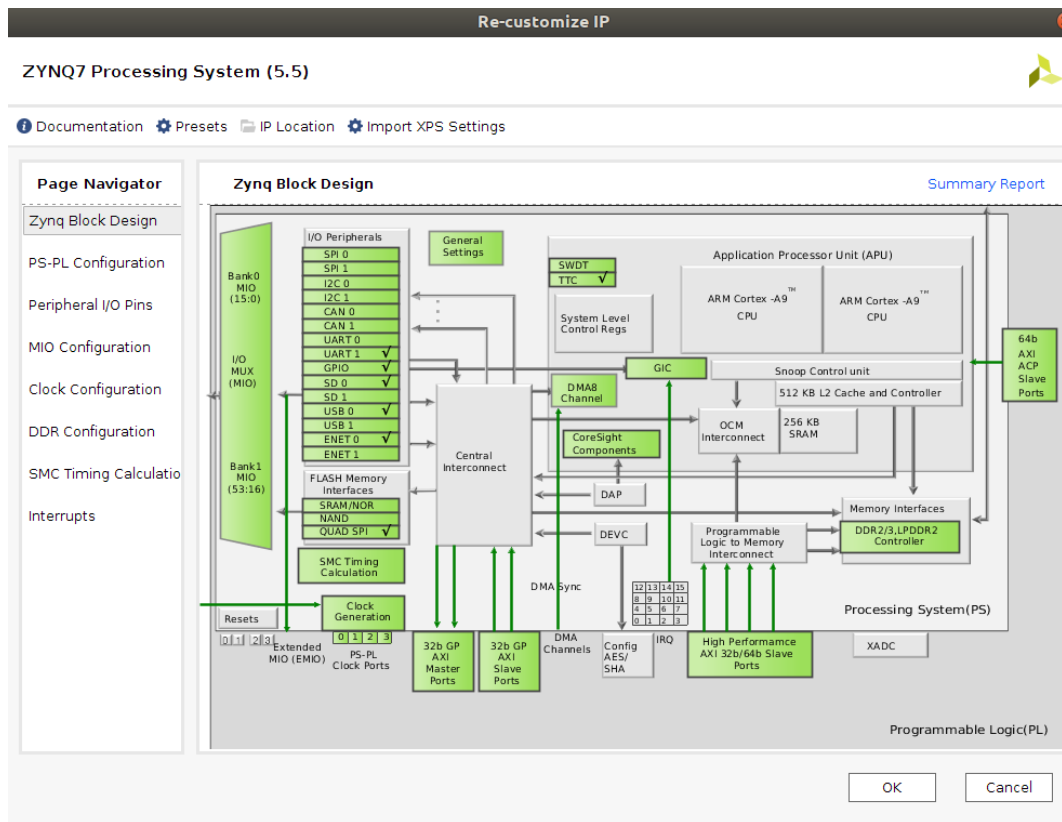
- Configurazione del Processing System tramite Vivado:
 - Aggiungere l'IP Zynq7 Processing System



- Clicchiamo su Run connect automation, in modo da collegarlo ad alcune periferiche necessarie: DDR e Fixed IO.

Configurazione PS

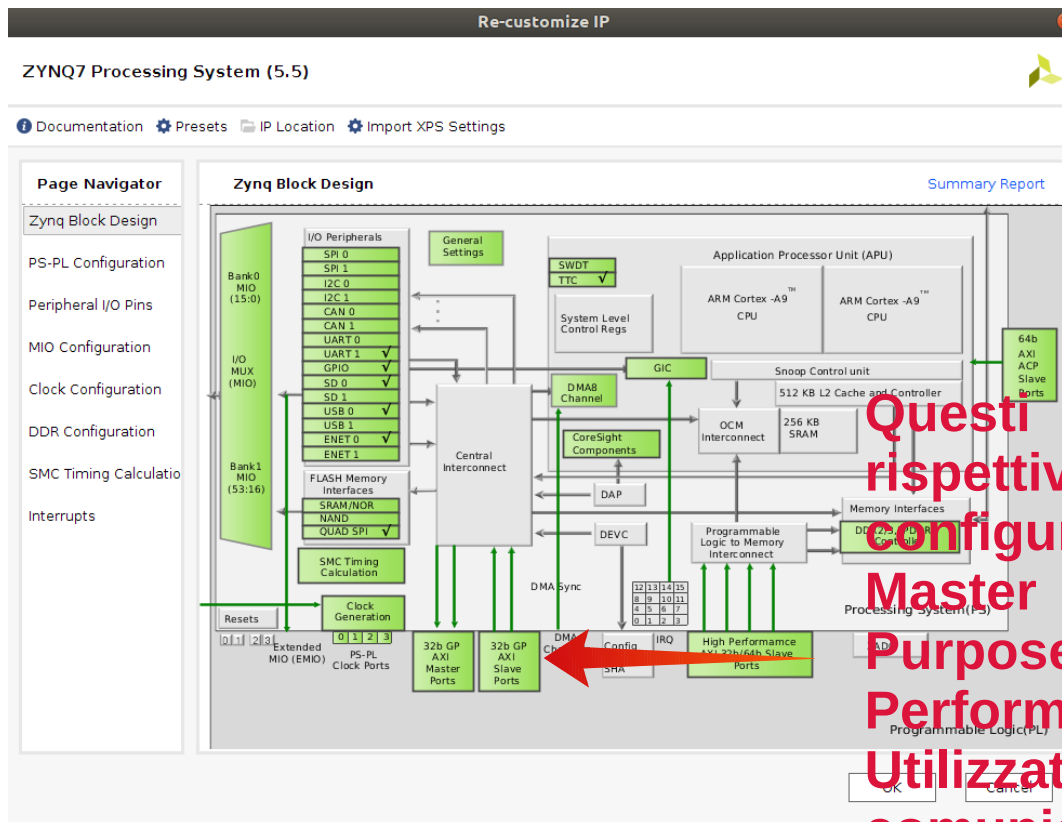
- Doppio Click sul modulo, per entrare nelle impostazioni:



- Lo schema in figura è lo schema a blocchi del PS.
- Tutto ciò che è riportato in verde è configurabile.
- Vediamo solamente le cose fondamentali.

Configurazione PS - GP/HP Ports

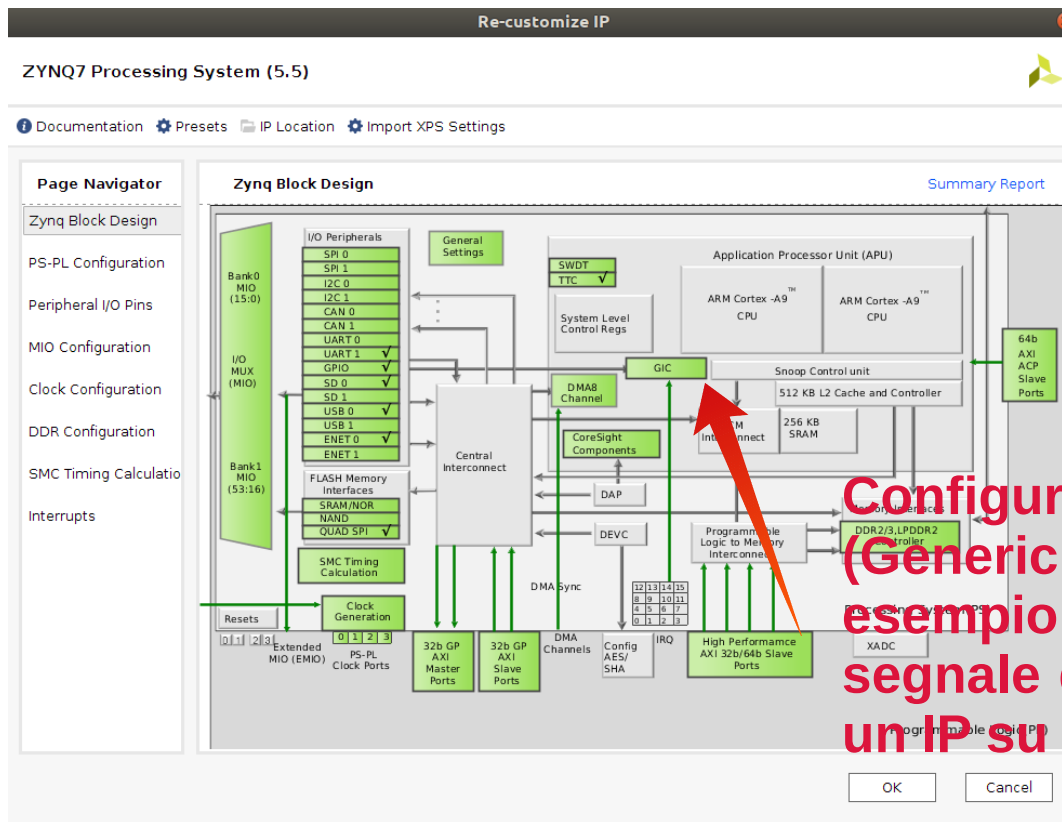
- Doppio Click sul modulo, per entrare nelle impostazioni:



Questi due blocchi
rispettivamente sono per la
configurazione di porte
Master / Slave General
Purpose (GP Ports) e High
Performance (HP Ports).
Utilizzate da AXI per
comunicare con la PL.

Configurazione PS - GIC

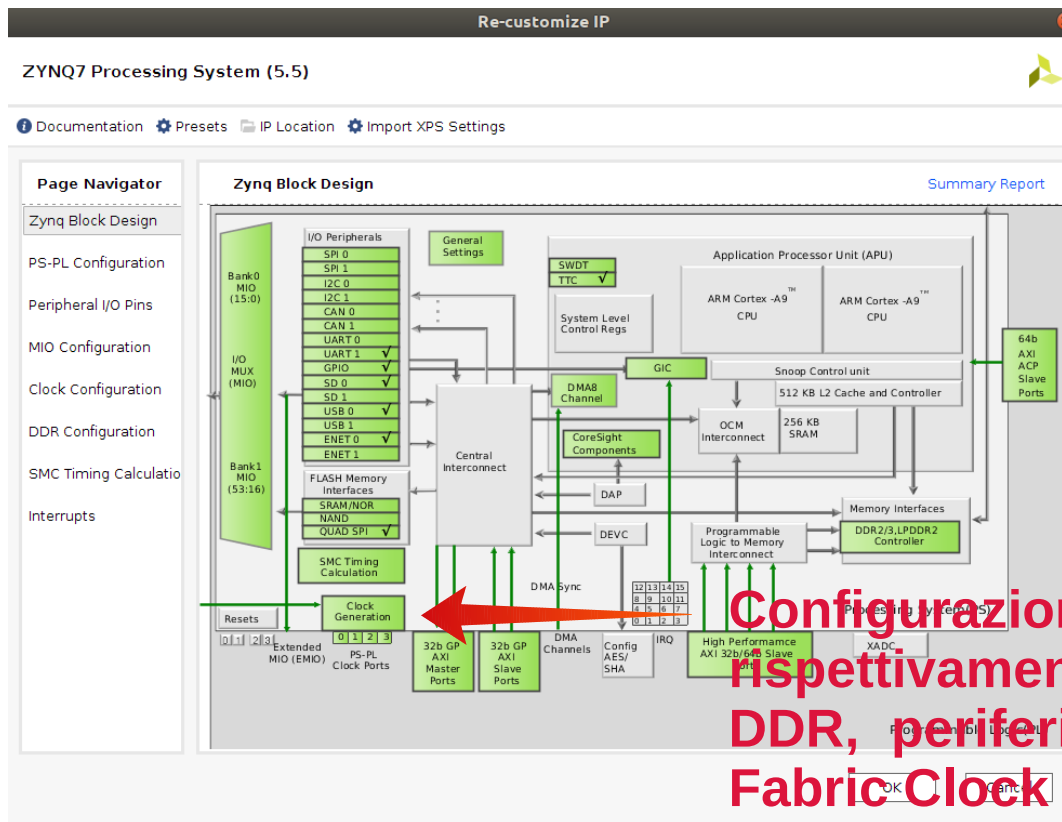
- Doppio Click sul modulo, per entrare nelle impostazioni:



Configurazione del GIC (Generic Interrupt Controller), esempio per ricevere un segnale di termine da parte di un IP su FPGA.

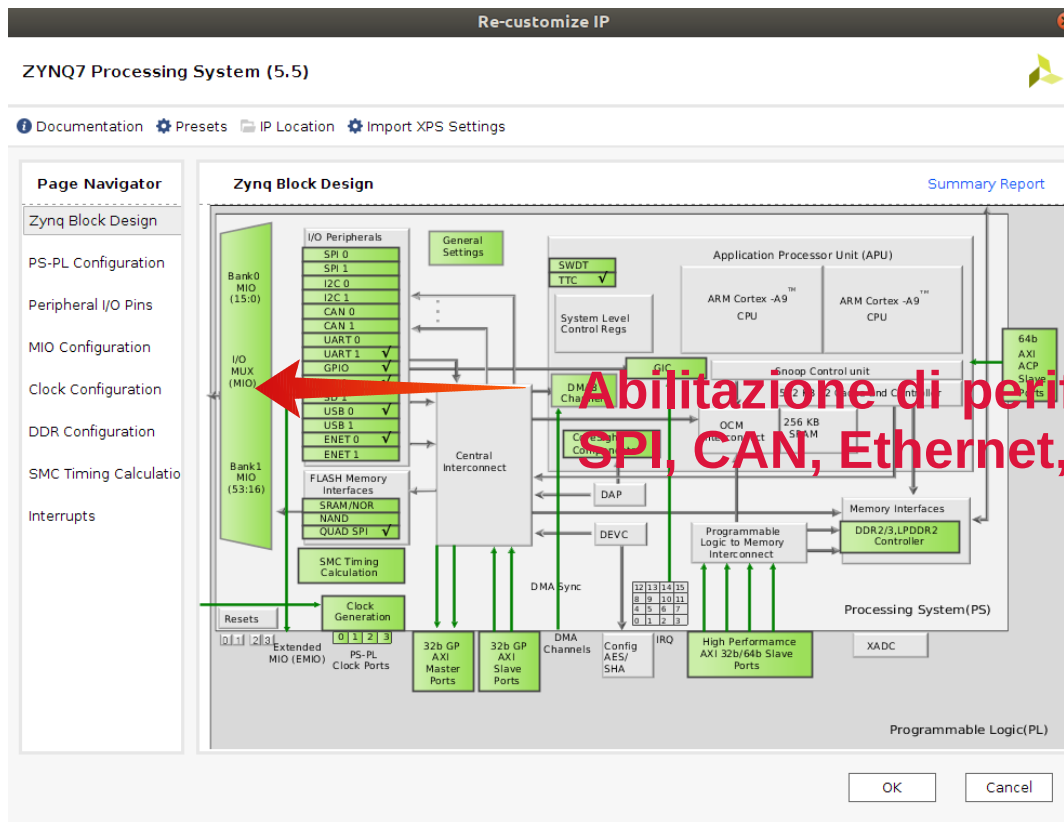
Configurazione PS - Clock

- Doppio Click sul modulo, per entrare nelle impostazioni:



Configurazione PS - Periferiche

- Doppio Click sul modulo, per entrare nelle impostazioni:



Abilitazione di periferiche, esempio: I2C, SPI, CAN, Ethernet, USB, ...

Comunicazione PS-PL: AMBA AXI

- interfacce di comunicazione, definite da ARM, e ora standard per comunicazioni on-chip.
- **AXI4**: Fornisce elevate prestazioni per accessi di tipo **memory mapped**. Permette di eseguire fino a 256 trasferimenti con singolo indirizzamento (burst) e altre caratteristiche avanzate per ottenere **alte prestazioni**.
- **AXI4-Lite**: Consente di eseguire singoli (no burst) trasferimenti di tipo **memory mapped** che richiedono **basso livello di throughput**. Spesso utilizzato per la scrittura e la lettura di registri di stato e di controllo.
- **AXI4-Stream**: Per streaming ad **alte prestazioni**. **Non prevede indirizzamento** (i.e. non è di tipo memory mapped). Consente trasferimenti dati di dimensione illimitata. Utilizzato tipicamente, ma non solo, per trasferire flussi di immagini.

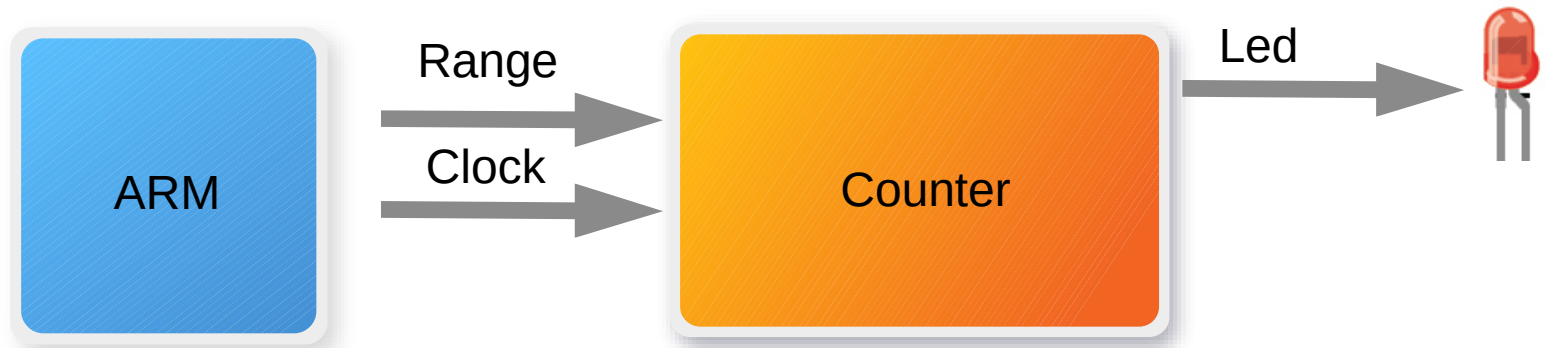
Comunicazione PS-PL: AMBA AXI

- Le comunicazioni AXI4 sono di tipo **Master-Slave** bidirezionali, nel senso che la comunicazione è iniziata da una periferica master.
- Per utilizzare AXI4 tramite HLS, è necessario marcare una particolare porta di ingresso tramite le seguenti pragma:

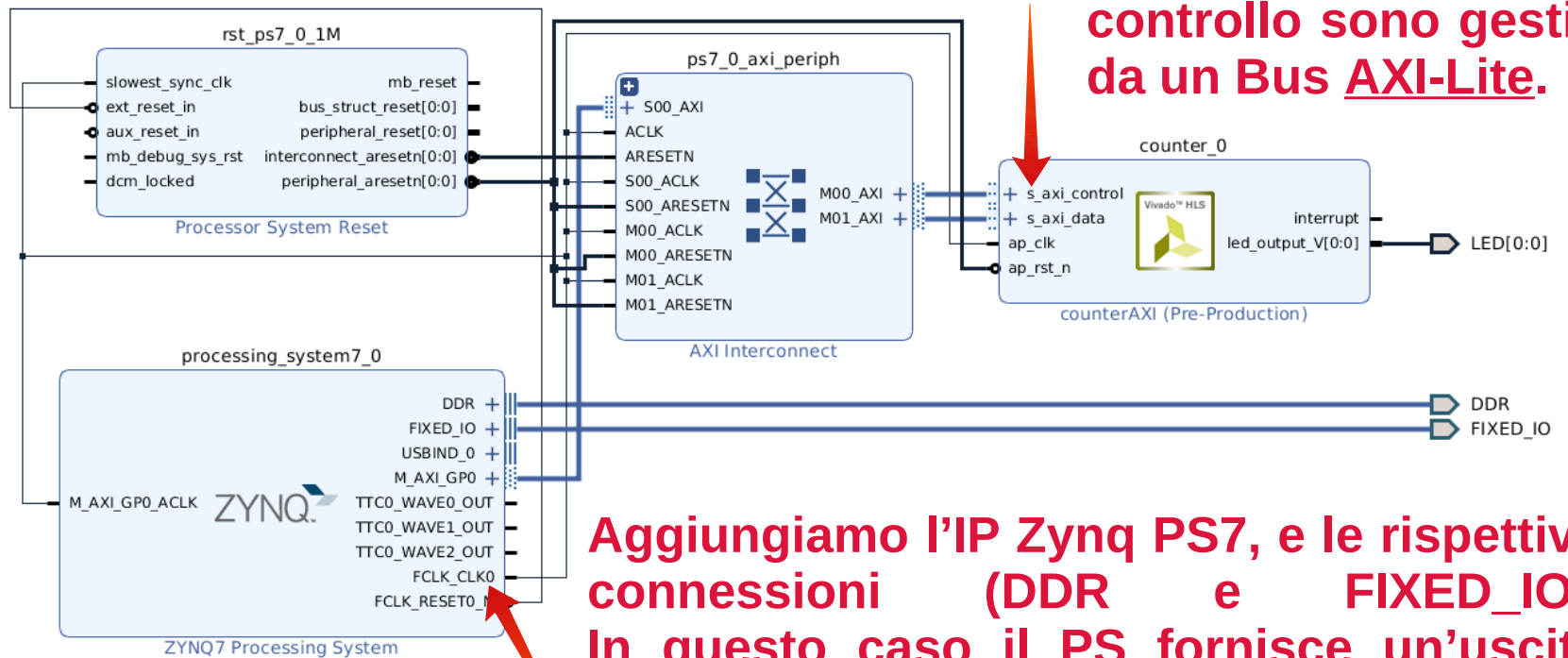
```
1 #pragma HLS INTERFACE s_axilite port=myport bundle=mybundle
2 #pragma HLS INTERFACE m_axi      port=myport bundle=mybundle
3 #pragma HLS INTERFACE axis       port=myport bundle=mybundle
```

HLS – Esercitazione 02

- Provare a realizzare in FPGA un modulo contatore che dopo N cicli di clock vada ad invertire lo stato di un led, mappato sulla ZedBoard.
- Estendiamo la precedente esercitazione in modo tale che sia il PS a passare il range al modulo FPGA, attraverso AXI-Lite:



Creazione Block Design - Differenze



Tutti i segnali di controllo sono gestiti da un Bus AXI-Lite.

Aggiungiamo l'IP Zynq PS7, e le rispettive connessioni (DDR e FIXED_IO). In questo caso il PS fornisce un'uscita configurabile di clock (FCLK_CLK0), quindi il Clocking Wizard non è più necessario.

Esportiamo l'Hardware

- Dopo aver generato il bitstream, invece che programmare direttamente il dispositivo andiamo ad esportare l'hardware generato:
- File > Export > Export Hardware.
- File > Launch SDK
- In questo modo verrà lanciata un'istanza del software **Xilinx SDK** avente come workspace la directory del progetto Vivado che abbiamo creato.
- Tramite XSDK andiamo a creare il software per ARM (Baremetal) che ci servirà per controllare il modulo FPGA.

XSDK Overview - BSP

The screenshot displays the Xilinx SDK IDE interface. The Project Explorer on the left shows a project named 'design_1_wrapper_hw_platform_0' with a sub-folder 'drivers' containing files like 'design_1_wrapper.bit', 'ps7_init_gpl.c', 'ps7_init_gpl.h', 'ps7_init.c', 'ps7_init.h', 'ps7_init.html', 'ps7_init.tcl', and 'system.hdf'. An orange arrow points from the text to the 'system.hdf' file. The Register Overview window in the center shows a table of registers with columns for Cell, Base Address, High Address, Base Width, High Width, and Mem/Reg. The SDK Log at the bottom shows messages from 16:00:50.

File design_1_wrapper.bit è il bitstream che configura l'FPGA. La cartella driver contiene dei file .c contenenti i drivers dei moduli FPGA utilizzati.

Cell	Base Addr	High Addr	Base Width	High Width	Mem/Reg
ps7_intc_usr	0xf8001000	0xf8001fff			REGISTER
counter_0	0x43c10000	0x43c1ffff	s_axi_data		REGISTER
ps7_gp0_0	0xe000a000	0xe000afff			REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f006ff			REGISTER
ps7_slc0_0	0xf8000000	0xf80000ff			REGISTER
ps7_scuwdt_0	0xf8000600	0xf80006ff			REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff			REGISTER
ps7_sce_0	0xf8000000	0xf80000ff			REGISTER
ps7_qspi_mcal_0	0xe0000000	0xe0000fff			REGISTER
ps7_pmu_0	0xf8003000	0xf8003fff			REGISTER
ps7_afi_1	0xf8009000	0xf8009fff			REGISTER
ps7_afi_0	0xf8008000	0xf8008fff			REGISTER
ps7_qspi_0	0xe000d000	0xe000dfff			REGISTER
ps7_usb_0	0xe0002000	0xe0002fff			REGISTER
ps7_afi_3	0xf800b000	0xf800bfff			REGISTER
ps7_afi_2	0xf800a000	0xf800afff			REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff			REGISTER
counter_0	0x43c00000	0x43c0ffff	s_axi_control		REGISTER
ps7_dma_s	0xf8003000	0xf8003fff			REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff			REGISTER
ps7_xadc_0	0xf8007100	0xf8007120			REGISTER
ps7_ddr_0	0x00100000	0x1ffffff			MEMORY
ps7_ddrc_0	0xf8006000	0xf8006fff			REGISTER
ps7_ocmc_0	0xf800c000	0xf800cfff			REGISTER
ps7_pl310_0	0xf8f02000	0xf8f02fff			REGISTER
ps7_uart_1	0xe0001000	0xe0001fff			REGISTER

16:00:50 INFO : Registering command handlers for SDK TCF services
16:00:50 INFO : Launching XSCT server: xsct -n -interactive /home/aian/Doc

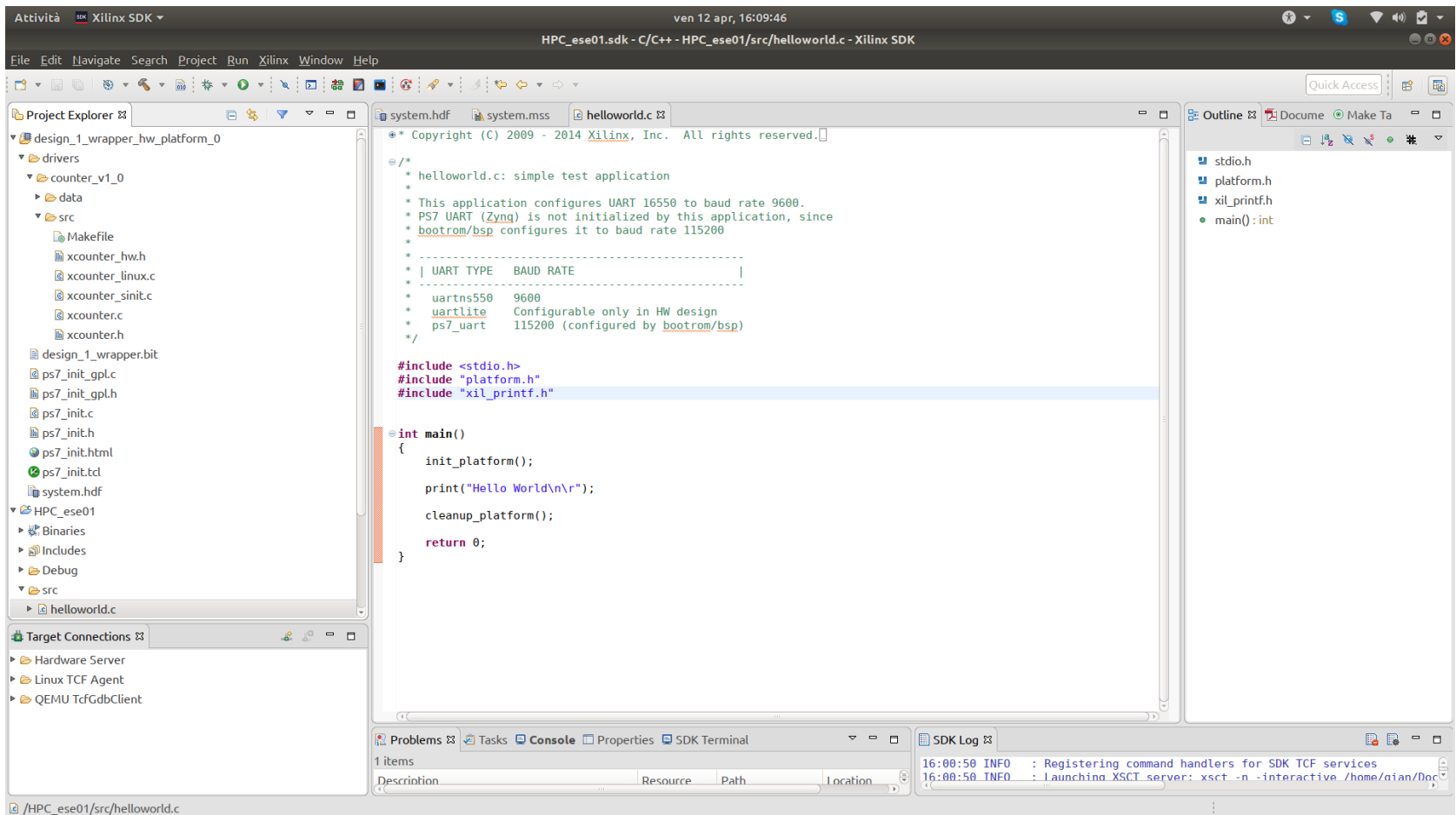
XSDK Overview - BSP

Tabella degli indirizzi mappati in RAM, che possiamo vedere da software. Notiamo gli indirizzi relativi alle interfacce AXI-Lite.

Cell	Base Addr	High Addr	Slave I/f	Mem/Reg
ps7_intc_dist_0	0xf8f01000	0xf8f01fff		REGISTER
counter_0	0x43c10000	0x43c1ffff	s_axi_data	REGISTER
ps7_gpio_0	0xe000a000	0xe000afff		REGISTER
ps7_scutimer_0	0xf8f00600	0xf8f0061f		REGISTER
ps7_slcr_0	0xf8000000	0xf8000fff		REGISTER
ps7_scuwdt_0	0xf8f00620	0xf8f006ff		REGISTER
ps7_l2cachec_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_scuc_0	0xf8f00000	0xf8f000fc		REGISTER
ps7_qspi_linear_0	0xfc000000	0xfcffffff		FLASH
ps7_pmu_0	0xf8893000	0xf8893fff		REGISTER
ps7_afi_1	0xf8009000	0xf8009fff		REGISTER
ps7_afi_0	0xf8008000	0xf8008fff		REGISTER
ps7_qspi_0	0xe000d000	0xe000dfff		REGISTER
ps7_usb_0	0xe0002000	0xe0002fff		REGISTER
ps7_afi_3	0xf800b000	0xf800bfff		REGISTER
ps7_afi_2	0xf800a000	0xf800afff		REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff		REGISTER
counter_0	0x43c00000	0x43c0ffff	s_axi_control	REGISTER
ps7_dma_s	0xf8003000	0xf8003fff		REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff		REGISTER
ps7_xadc_0	0xf8007100	0xf8007120		REGISTER
ps7_ddr_0	0x00100000	0x1ffffff		MEMORY
ps7_ddrc_0	0xf8006000	0xf8006fff		REGISTER
ps7_ocmc_0	0xf800c000	0xf800cfff		REGISTER
ps7_pl310_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_uart_1	0xe0001000	0xe0001fff		REGISTER

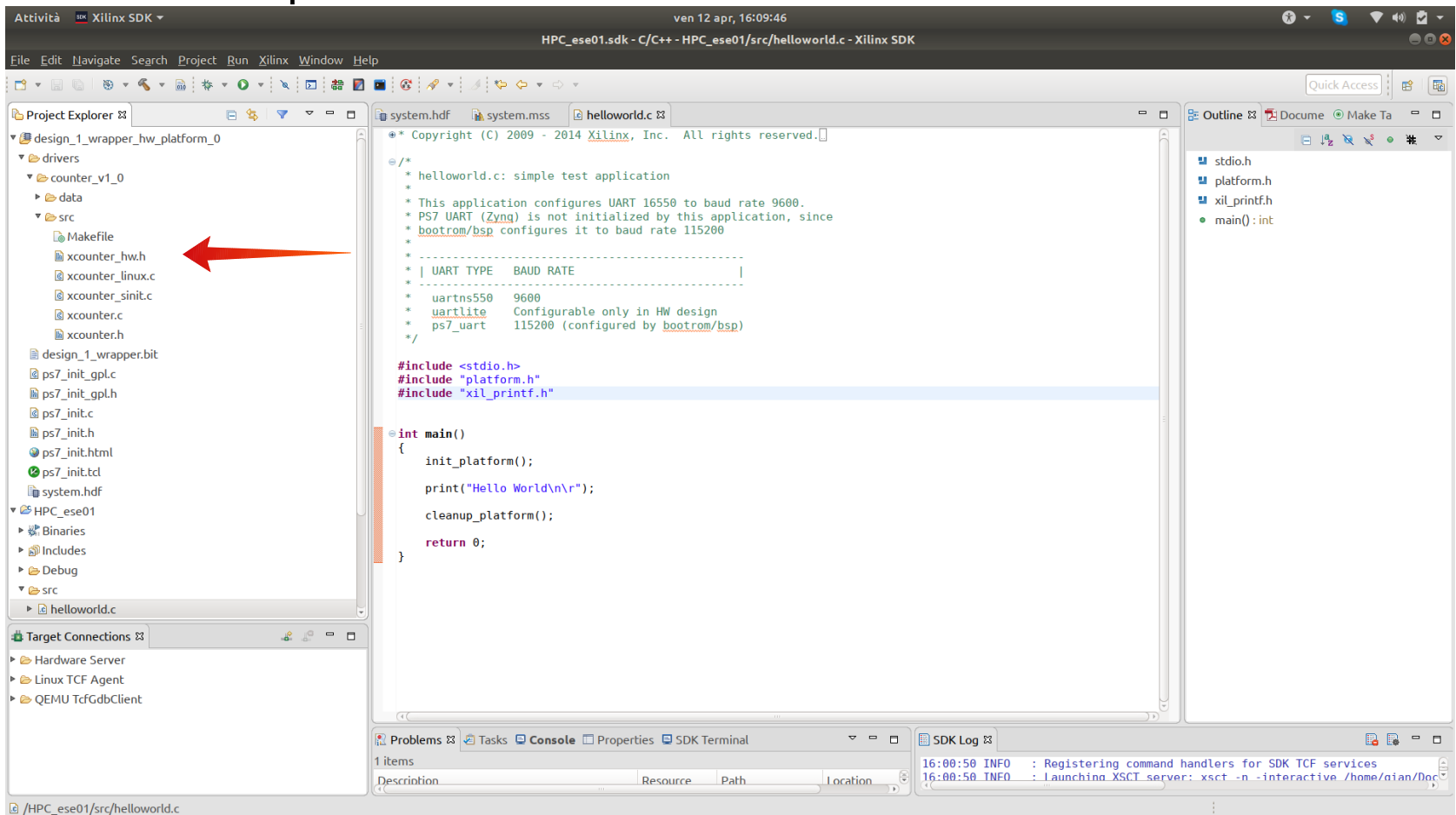
XSDK Overview - Progetto

- Creiamo un nuovo “Application Project”, avente come template “Hello World”



XSDK Overview - Drivers

- Notiamo i file include all'interno della cartella "Drivers", dovremo includerli per usare il modulo contatore.



The screenshot displays the Xilinx SDK IDE interface. On the left, the Project Explorer shows a project named 'design_1_wrapper_hw_platform_0' with a sub-directory 'drivers' containing a 'counter_v1_0' sub-directory. Inside 'counter_v1_0', there is a 'src' directory containing files like 'xcounter_hw.h', 'xcounter_linux.c', 'xcounter_sinit.c', 'xcounter.c', and 'xcounter.h'. A red arrow points from the 'xcounter.h' file in the Project Explorer to the '#include "xil_printf.h"' line in the main source file.

The main editor window shows the source file 'helloworld.c' with the following content:

```
/* Copyright (C) 2009 - 2014 Xilinx, Inc. All rights reserved. */
/*
 * helloworld.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 *-----
 * | UART TYPE   BAUD RATE |
 *-----
 * uarts550     9600
 * uartlite     Configurable only in HW design
 * ps7_uart     115200 (configured by bootrom/bsp)
 */

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"

int main()
{
    init_platform();

    print("Hello World\n");

    cleanup_platform();

    return 0;
}
```

The right-hand side of the IDE shows the Outline view, listing the included files: 'stdio.h', 'platform.h', 'xil_printf.h', and 'main(): int'.

At the bottom, the Console window shows the following log output:

```
16:00:50 INFO : Registering command handlers for SDK TCF services
16:00:50 INFO : Launching XSCT server: xsct -n -interactive /home/aian/Doc
```

XSDK Overview - Drivers

- Drivers principali:

The screenshot displays the Xilinx SDK IDE interface. The main window shows the source code for `xcounter.h`. The code defines various macros and function prototypes for the Xcounter driver. The macros include `Xil_AssertVoid`, `Xil_AssertNonvoid`, `XST_SUCCESS`, `XST_DEVICE_NOT_FOUND`, `XST_OPEN_DEVICE_FAILED`, and `XIL_COMPONENT_IS_READY`. The function prototypes include `Xcounter_Initialize`, `Xcounter_CfgInitialize`, `Xcounter_Initialize` (with `InstanceName`), `Xcounter_Release`, `Xcounter_Start`, `Xcounter_IsDone`, `Xcounter_IsIdle`, `Xcounter_IsReady`, `Xcounter_EnableAutoRestart`, `Xcounter_DisableAutoRestart`, `Xcounter_Set_range_r`, `Xcounter_Get_range_r`, `Xcounter_InterruptGlobalEnable`, `Xcounter_InterruptGlobalDisable`, `Xcounter_InterruptEnable`, `Xcounter_InterruptDisable`, `Xcounter_InterruptClear`, `Xcounter_InterruptGetEnabled`, and `Xcounter_InterruptGetStatus`. The code also includes conditional compilation for `__cplusplus`.

```
#define Xil_AssertVoid(expr)    assert(expr)
#define Xil_AssertNonvoid(expr) assert(expr)

#define XST_SUCCESS            0
#define XST_DEVICE_NOT_FOUND  2
#define XST_OPEN_DEVICE_FAILED 3
#define XIL_COMPONENT_IS_READY 1
#endif

/***** Function Prototypes *****/
#ifndef __linux__
int Xcounter_Initialize(XCounter *InstancePtr, u16 DeviceId);
XCounter_Config* Xcounter_LookupConfig(u16 DeviceId);
int Xcounter_CfgInitialize(XCounter *InstancePtr, XCounter_Config *ConfigPtr);
#else
int Xcounter_Initialize(XCounter *InstancePtr, const char* InstanceName);
int Xcounter_Release(XCounter *InstancePtr);
#endif

void Xcounter_Start(XCounter *InstancePtr);
u32 Xcounter_IsDone(XCounter *InstancePtr);
u32 Xcounter_IsIdle(XCounter *InstancePtr);
u32 Xcounter_IsReady(XCounter *InstancePtr);
void Xcounter_EnableAutoRestart(XCounter *InstancePtr);
void Xcounter_DisableAutoRestart(XCounter *InstancePtr);

void Xcounter_Set_range_r(XCounter *InstancePtr, u32 Data);
u32 Xcounter_Get_range_r(XCounter *InstancePtr);

void Xcounter_InterruptGlobalEnable(XCounter *InstancePtr);
void Xcounter_InterruptGlobalDisable(XCounter *InstancePtr);
void Xcounter_InterruptEnable(XCounter *InstancePtr, u32 Mask);
void Xcounter_InterruptDisable(XCounter *InstancePtr, u32 Mask);
void Xcounter_InterruptClear(XCounter *InstancePtr);
u32 Xcounter_InterruptGetEnabled(XCounter *InstancePtr);
u32 Xcounter_InterruptGetStatus(XCounter *InstancePtr);

#ifdef __cplusplus
}
#endif
#endif
```

The Project Explorer on the left shows the project structure, including the `drivers` directory and the `xcounter.h` file. The Target Connections pane at the bottom left shows the hardware server and QEMU TcfGdbClient. The SDK Log at the bottom right shows the registration of command handlers for SDK TCF services.

XSDK Overview - Drivers

- I driver solitamente sono nominati `x<nome-modulo>.h / .c`
- Nel nostro caso i principali sono **xcounter** e **xcounter_linux**, che sono rispettivamente utilizzati in caso di Baremetal e Linux.
- Nel caso di Linux, gestiscono la traduzione da indirizzo fisico a indirizzo virtuale, ovviamente in caso di progetto Linux, dovremo anche inserire questi indirizzi nel Device Tree del dispositivo.
- Per semplicità noi lavoreremo in Baremetal.

XSDK Overview – Codice ARM

- Realizziamo quindi il software di controllo come da questo esempio:

```
48 #include <stdio.h>
49 #include "platform.h"
50 #include "xil_printf.h"
51 #include "xcounter.h"
52
53 int main() {
54     init_platform();
55
56     XCounter counterInstance;
57
58     u32 range = 100000;
59
60     XCounter_Initialize(&counterInstance, 0);
61
62     printf("Counter initialized, range: %d\n\r", range);
63
64     XCounter_Set_range_V(&counterInstance, range);
65
66     printf("%x\n", Xil_In32(0x0));
67
68     while(1) {
69         XCounter_Start(&counterInstance);
70
71         while(!XCounter_IsDone(&counterInstance));
72         printf("Counter Done!\n\r");
73     }
74     cleanup_platform();
75
76     return 0;
77
78 }
79
```

HLS – Esercitazione 02.5

- Nella precedente esercitazione il controllo del modulo FPGA è gestito secondo una logica di **Polling**, ovvero il processore deve svolgere dei cicli nel quale va a controllare lo stato della periferica.
- Provare a modificare il design dell'esercitazione precedente, in modo tale da sfruttare l'uscita interrupt del modulo FPGA e il Generic Interrupt Controller della PS.
- Un esempio di configurazione del GIC è possibile trovarlo sul sito del corso.

