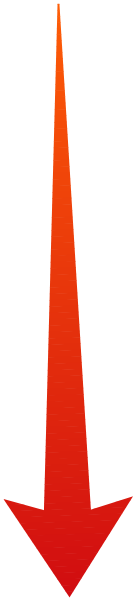
Seminario

FPGAs in practice

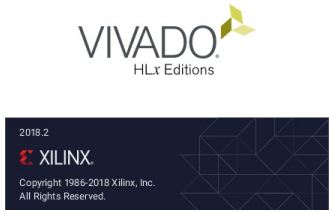
Gianluca Brilli
gianluca.brilli@unimore.it

Design su FPGA

- Principali strumenti di lavoro:



- *Vivado HLS:* permette la creazione di IP VHDL/Verilog a partire da codice C/C++.

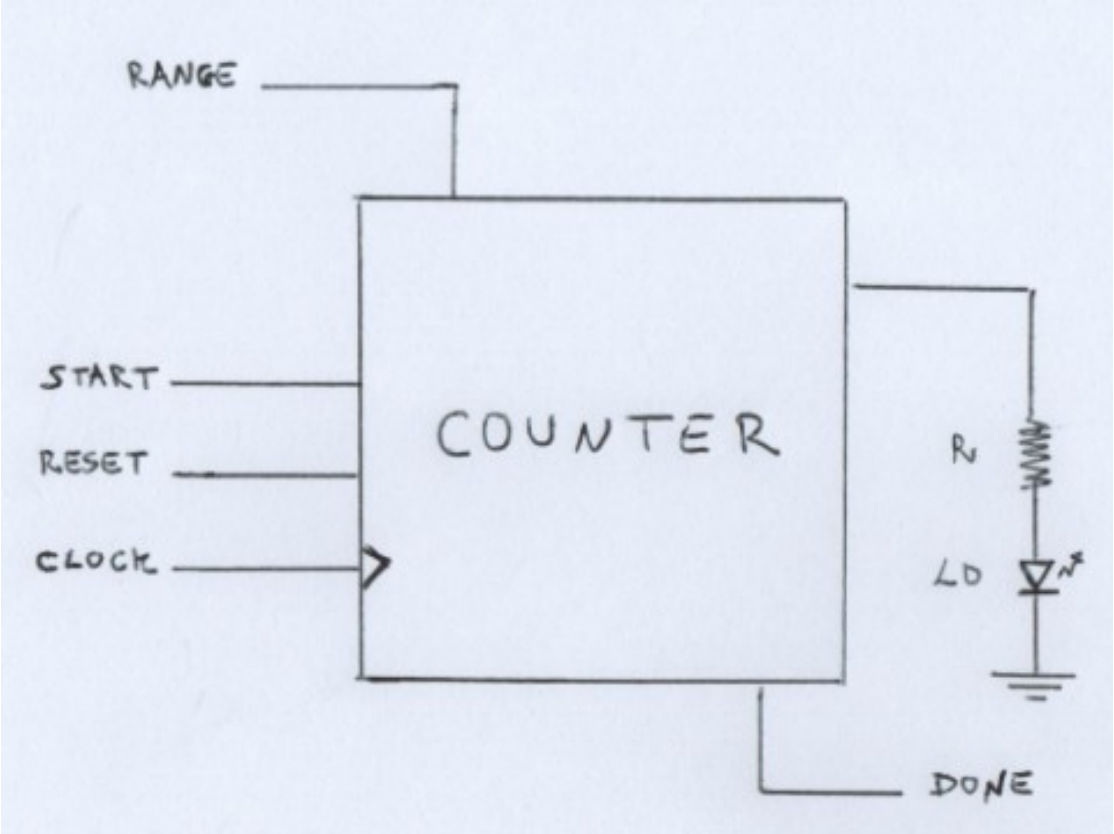


- *Vivado HLx:* Creazione del design hardware su FPGA, integrazione delle IP e configurazione dei cores ARM.

- *Xilinx SDK:* ambiente di cross-compilazione del codice ARM.

Contatore in FPGA, design HLS

- Logica dell'IP-core:



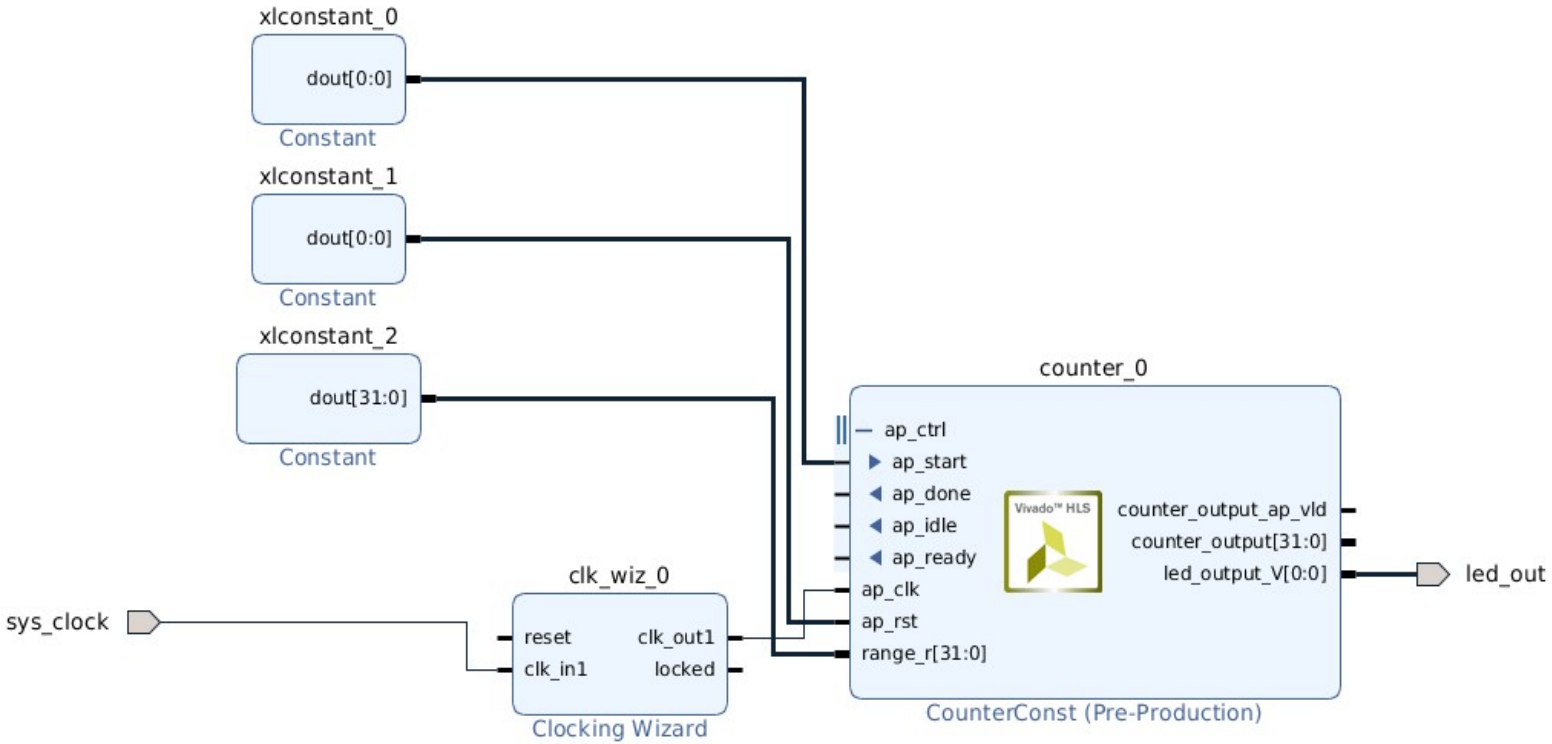
Contatore in FPGA – IP design (1)

- Logica dell'IP-core:
  - Scrivere un contatore modulo range, che abbassa la frequenza in ingresso.
  - Portare l'uscita del contatore in ingresso ad un Led.
  - Per il momento usiamo solamente l'FPGA, i cores ARM restano spenti.

Contatore in FPGA – IP design (2)

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

Layout Vivado



Principali Componenti

- Elementi Fondamentali:
  - *Counter (pre-production):* IP contatore sviluppato in Vivado HLS.
  - *Clocking Wizard:* Tramite il pin Y9 della ZedBoard permette di fornire all'IP una sorgente di clock configurabile.
  - *Costanti:* Rispettivamente fanno partire il modulo *(ap_start),* mantengono il reset in off *(ap_rst) e* forniscono l'input per il range *(range_r).*

Mapping dell I/O sulla ZedBoard

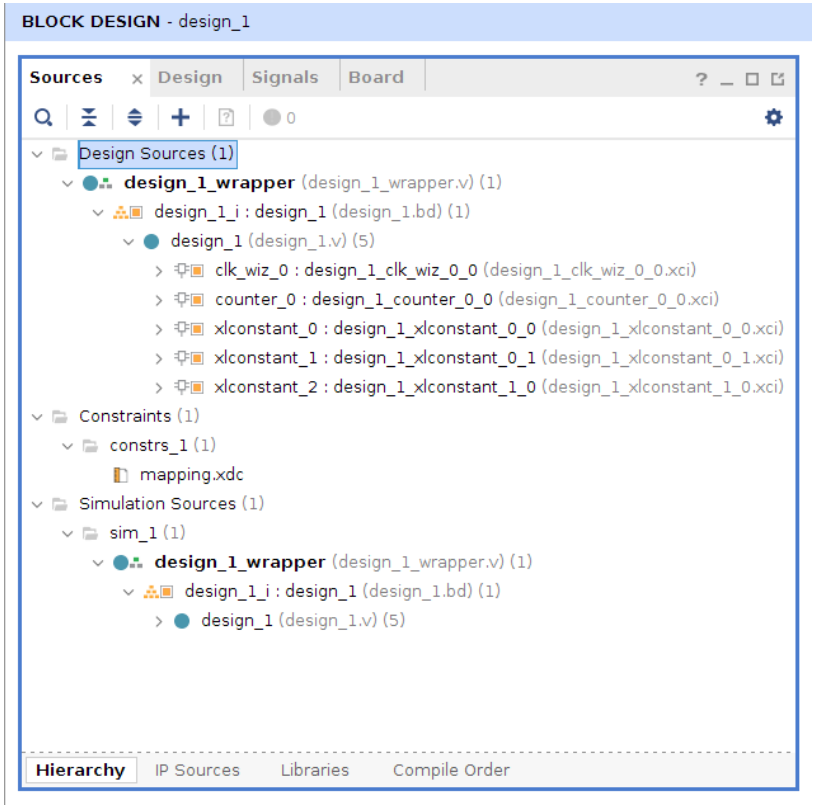
- Assegnamo le nostre porte di ingresso e uscita (*led_out e sys_clock)*, ad un led reale sulla board ed impostiamo una corretta tensione di alimentazione:

```
1 set_property PACKAGE_PIN      T22 [get_ports {led_out[0]}]
2 set_property IOSTANDARD  LVCMOS33 [get_ports {led_out[0]}]
3 set_property IOSTANDARD  LVCMOS33 [get_ports { sys_clock}]
4
```

- *"Generate HDL wrapper"* per andare ad instanziare i componenti del nostro design.

Generazione del Bitstream

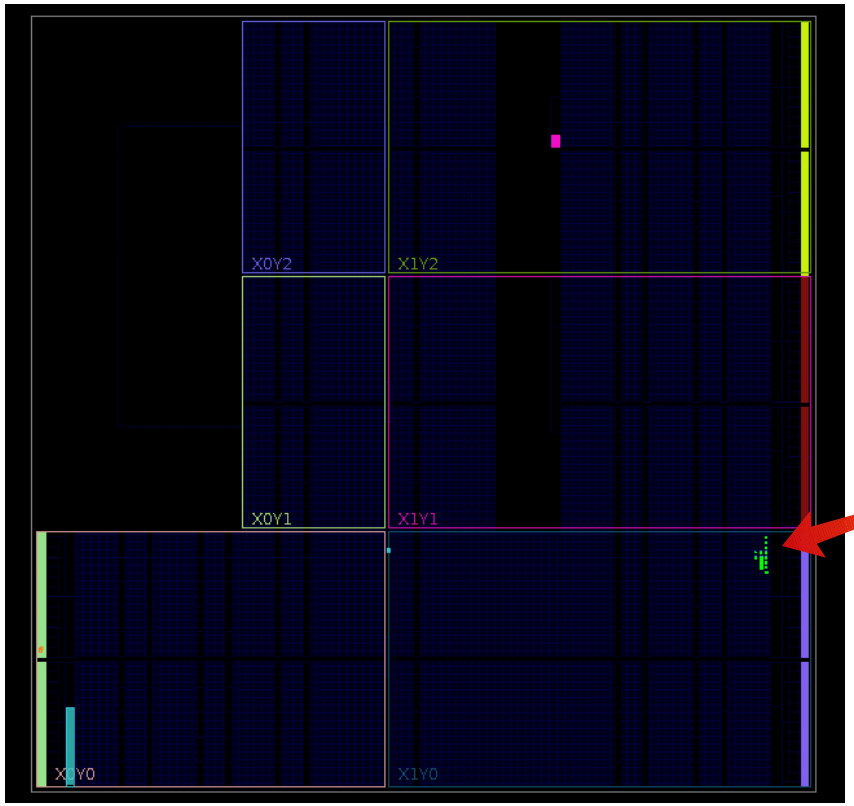
- Dopodiché dovremmo avere un file Verilog contenente i nostri componenti:



- Infine:
  - Run Synthesis
  - Run Implementation
  - Generate Bitstream

Contatore – Area Utilizzata

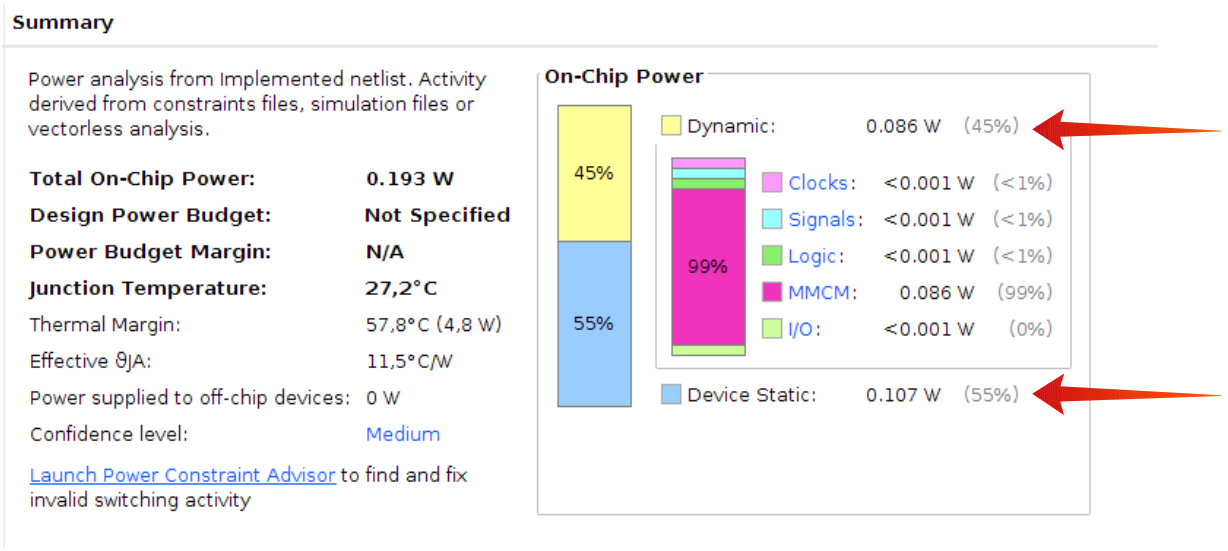
- Dopodiché possiamo vedere alcune cose interessanti sul bitstream generato:



Area programmabile utilizzata.

Contatore – Potenza Dissipata

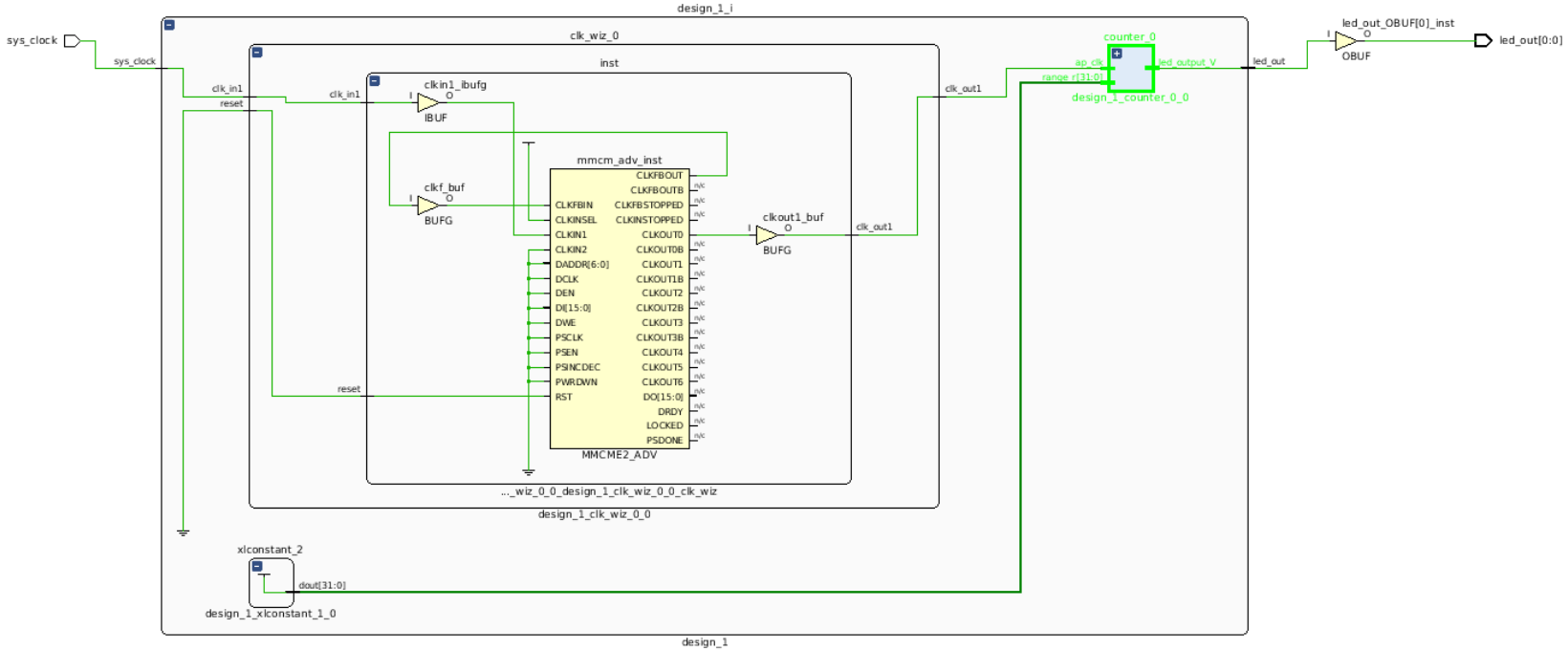
- Dopodiché possiamo vedere alcune cose interessanti sul bitstream generato:



Stima dei consumi di potenza

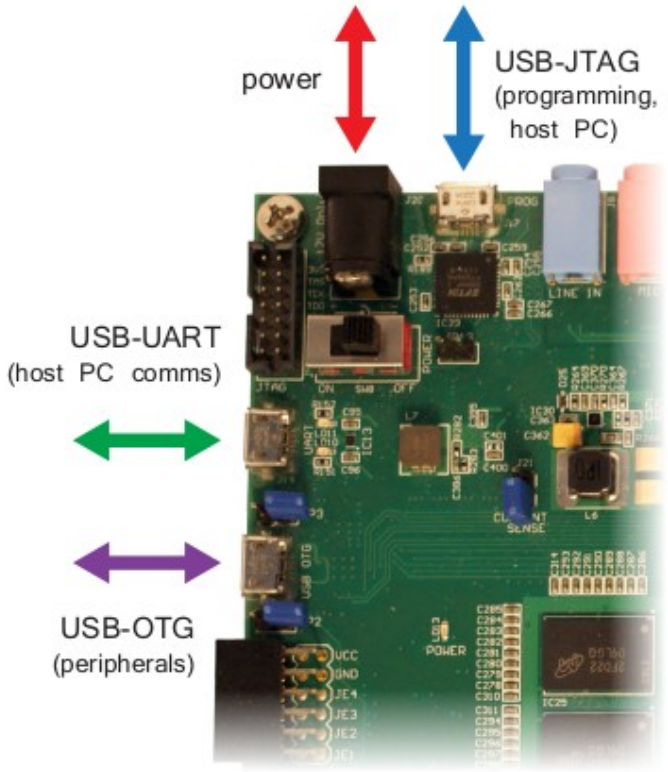
Contatore - Schematico

- Schematico del circuito su FPGA, autogenerato da Vivado:

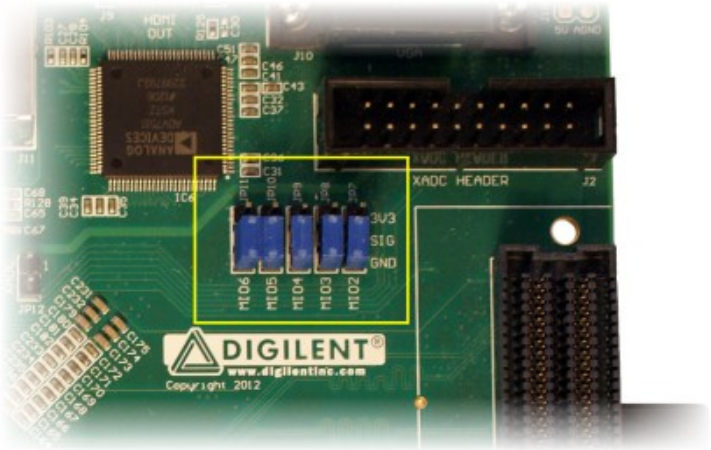


Test su ZedBoard (1)

- Colleghiamo alimentazione, UART e JTAG.

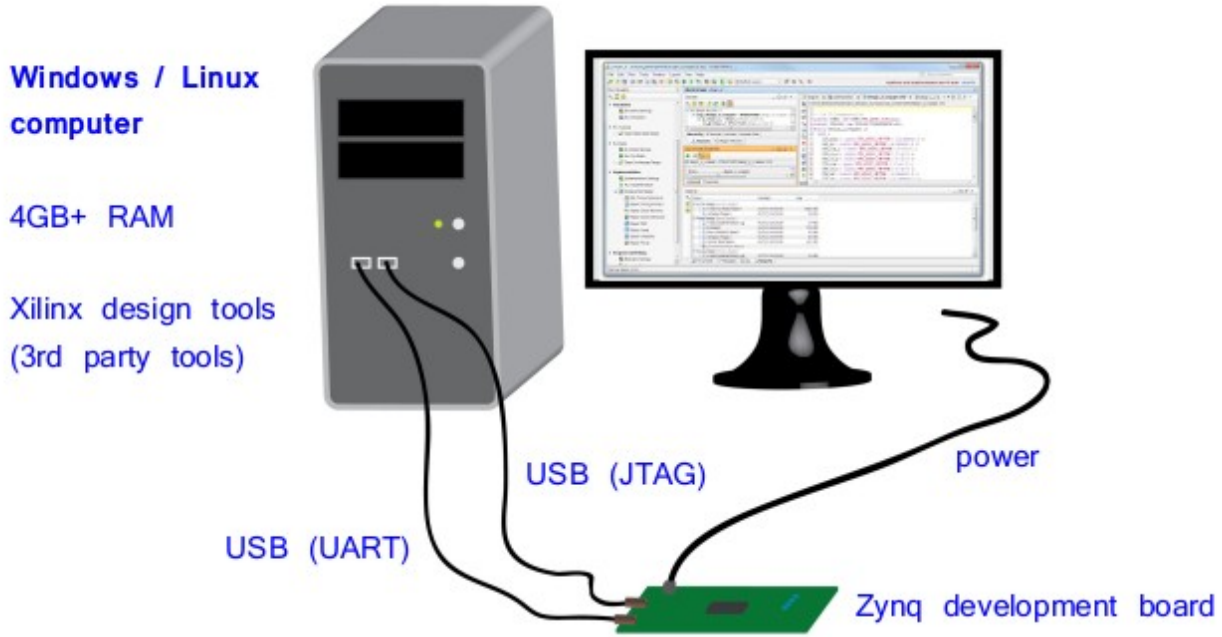


- Impostiamo il boot da JTAG



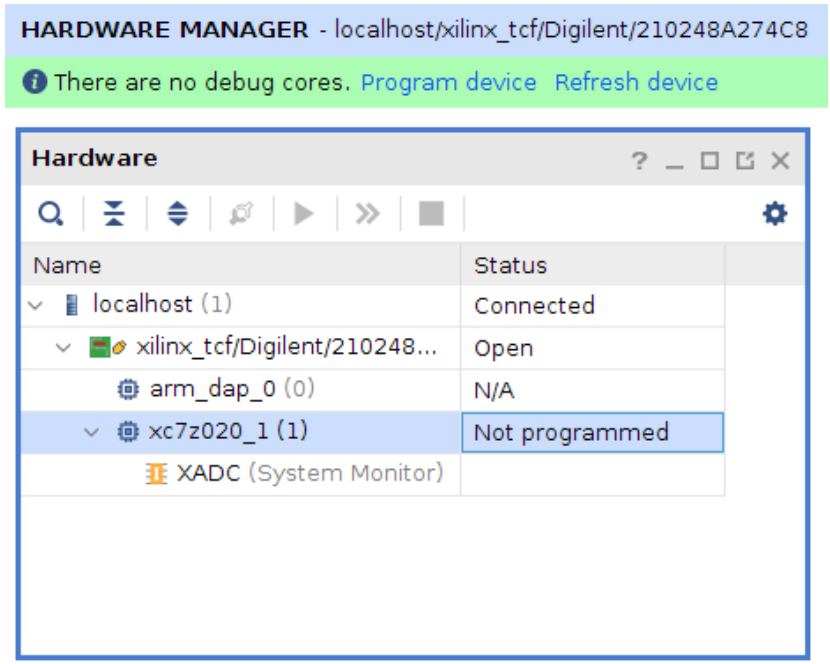
Test su ZedBoard (2)

- Collegamento:



Test su ZedBoard – Trasferimento Bitstream

- Su Vivado: *Open Hardware Manager > Open Target > Auto Connect > Program Device*



Contatore in FPGA controllo Software

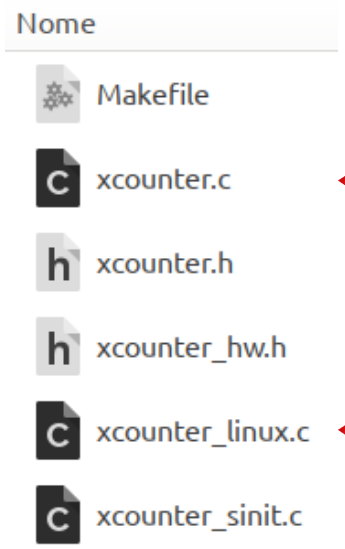
- Logica dell'IP-core:
  - Scrivere un contatore modulo range, che accende e spegne un led.
  - Interfaccia *AXI-Lite* per la connessione PL-PS, per quanto riguarda il range di conteggio.
  - Modulo *AXI-GPIO* per il controllo del bit di start (*ap_start*).
  - Gestione degli interrupt generati dalla PL (*ap_done*), tramite *IRQ_F2P*.

Contatore in FPGA – IP design (1)

```
1  #include "contatore.h"
2
3  void counter(
4          volatile unsigned int range,
5          volatile ap_uint<1> *led_output
6          ) {
7
8      #pragma HLS INTERFACE s_axilite port=range bundle=range
9      #pragma HLS INTERFACE ap_none port=led_output
10
11     unsigned int counter_value   = 0;
12     static ap_uint<1> led_status = 0;
13
14     while(counter_value < range){
15         counter_value ++;
16     }
17
18     led_status  = not(led_status);
19     *led_output = led_status;
20
21     return;
22 }
23
```

Contatore in FPGA – IP design (2)

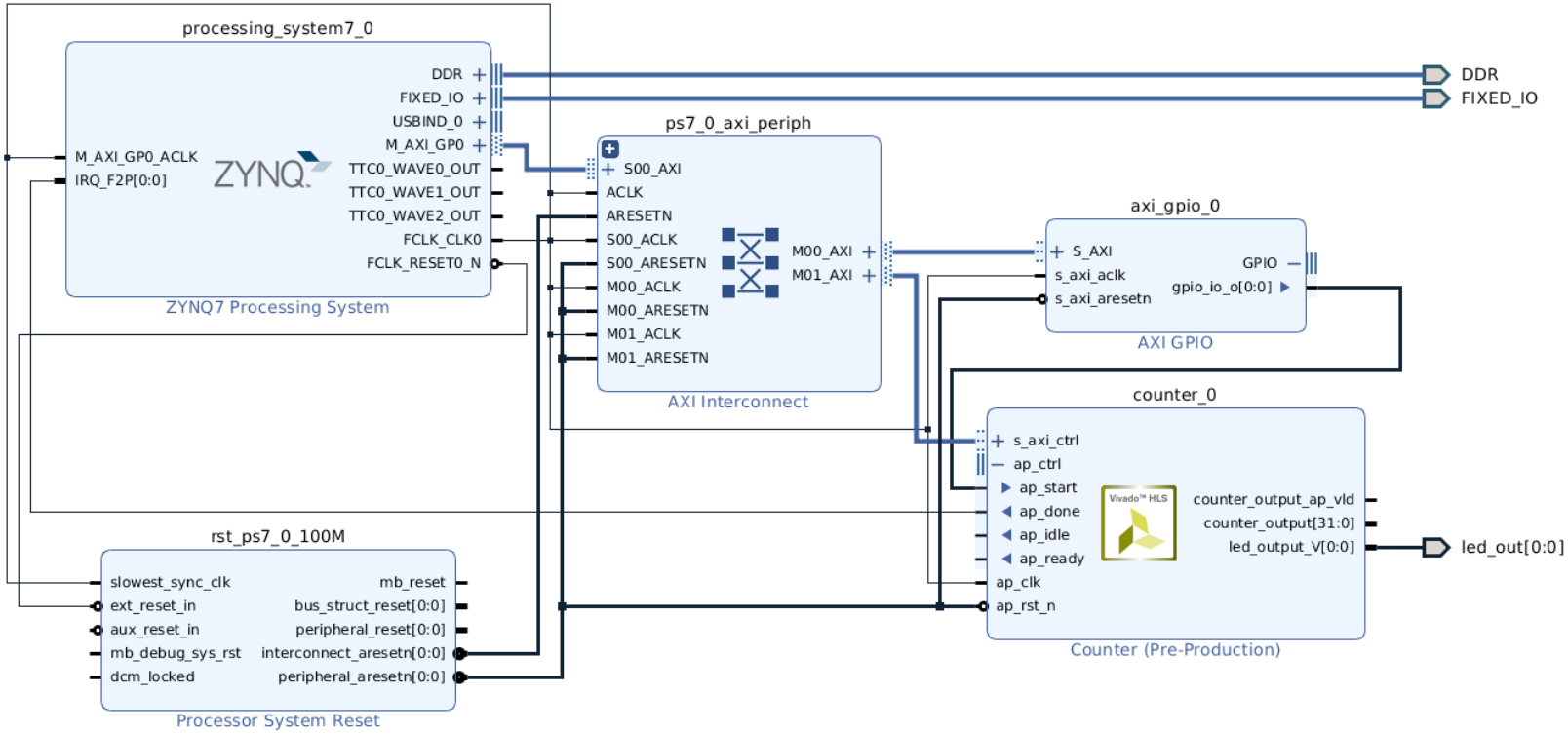
- In questo caso Vivado HLS va a creare una cartella contenente i drivers per utilizzare la periferica generata da software:
- *VivadoHLS-workspace/contatore/contatore_prj/solution1/impl/ip/drivers/counter_v1_4/src*



Driver per Baremetal.

Driver per Linux.

Layout Vivado



Componenti Utilizzati

- Elementi Fondamentali:
  - *Counter (pre-production):* IP contatore sviluppato in Vivado HLS.
  - *Processing-System7:* IP per la gestione del processore (*ARM cortex A9 dual core*); fornisce il clock all'esterno, invia alla PL tramite AXI e riceve gli interrupt.
  - *Processor System Reset: s*i occupa del reset della PS e delle periferiche.
  - *AXI-Interconnect:* permette l'utilizzo di più interfacce AXI.

Mapping degli Indirizzi

- La memoria è condivisa tra i due *ARM A9* e l'*FPGA*.
- E' presente un unico *Memory Controller*.
- Assegnamento indirizzi per le interfacce AXI:

Diagram | IP Catalog | **Address Editor**

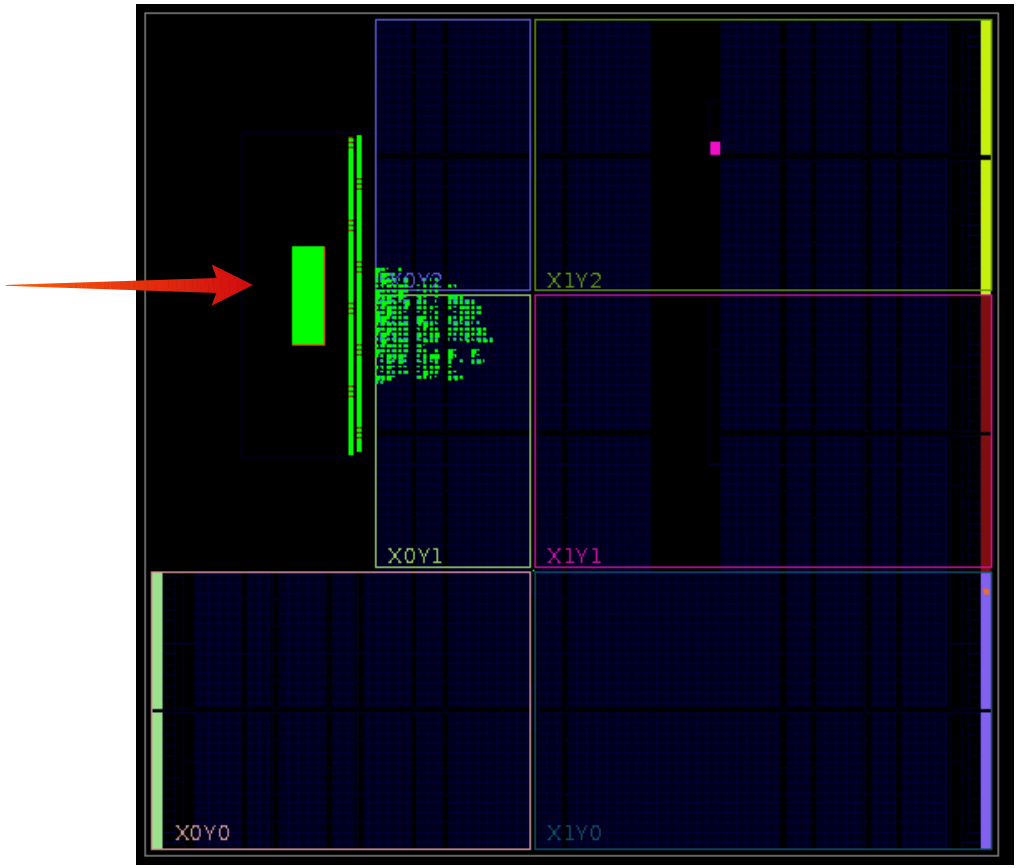
Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G])					
counter_0	s_axi_range	Reg	0x43C0_0000	64K	0x43C0_FFFF
axi_gpio_0	S_AXI	Reg	0x4120_0000	64K	0x4120_FFFF

Generazione Bitstream

- Dopodiché anche in questo caso andiamo a generare il wrapper in Verilog e mappiamo la variabile *led_out* su un led a nostra scelta ed assegnamo una tensione di *3.3V.*
- Infine come prima lanciamo *Sintesi, Implementazione* e *Generazione Bitstream.*

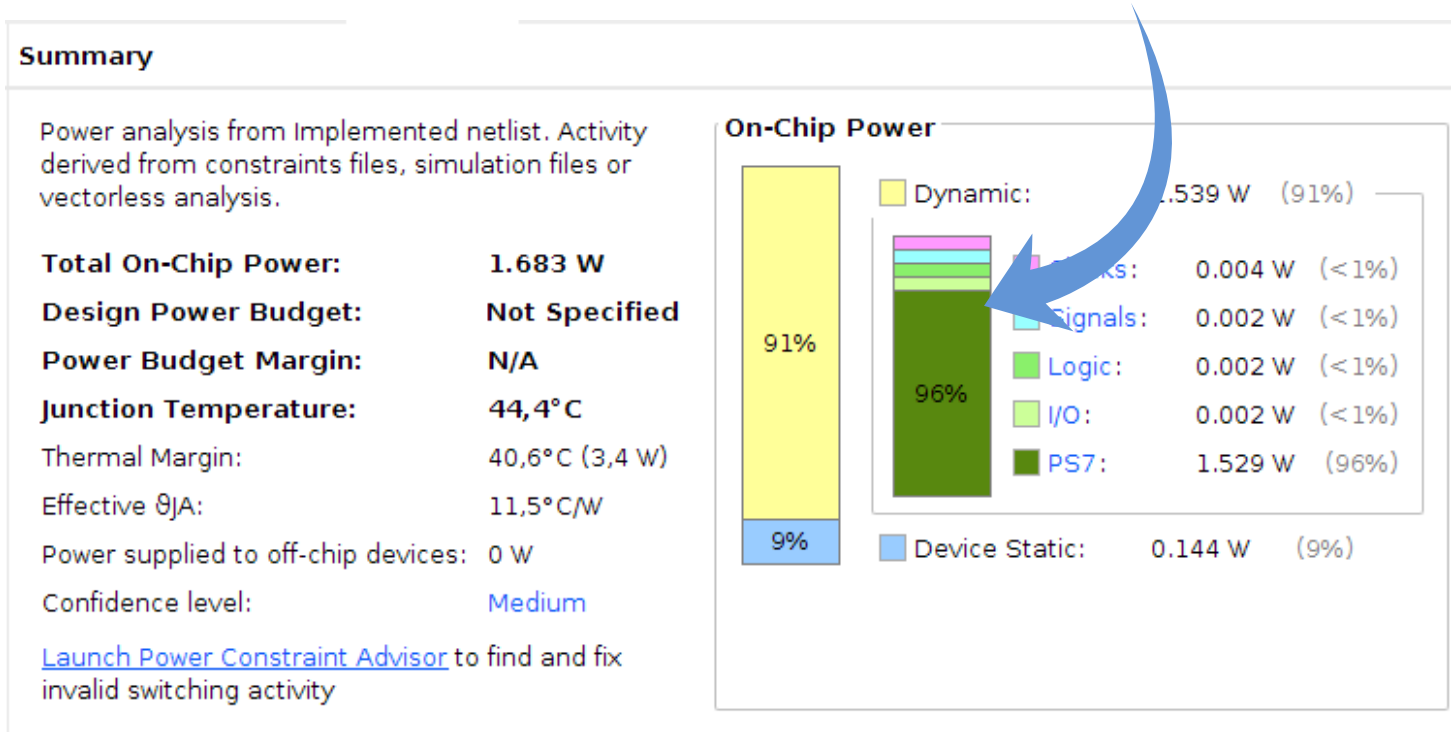
Contatore – Area Utilizzata

- Notiamo la differenza del consumo di CLBs rispetto alla versione precedente.



Contatore – Potenza Dissipata

- In questo caso i consumi sono dominati dalla *componente dinamica*, in particolare dalla CPU.



Esportazione dell'Hardware

- Una volta sintetizzato il Bitstream, andiamo ad esportare l'hardware e ad avviare XSDK:
  - *File > Export > Export Hardware*
  - *File > Launch SDK*
- Una volta avviato XSDK, andiamo a creare una nuova applicazione per *Baremetal* (*Standalone*), selezionamo C++.

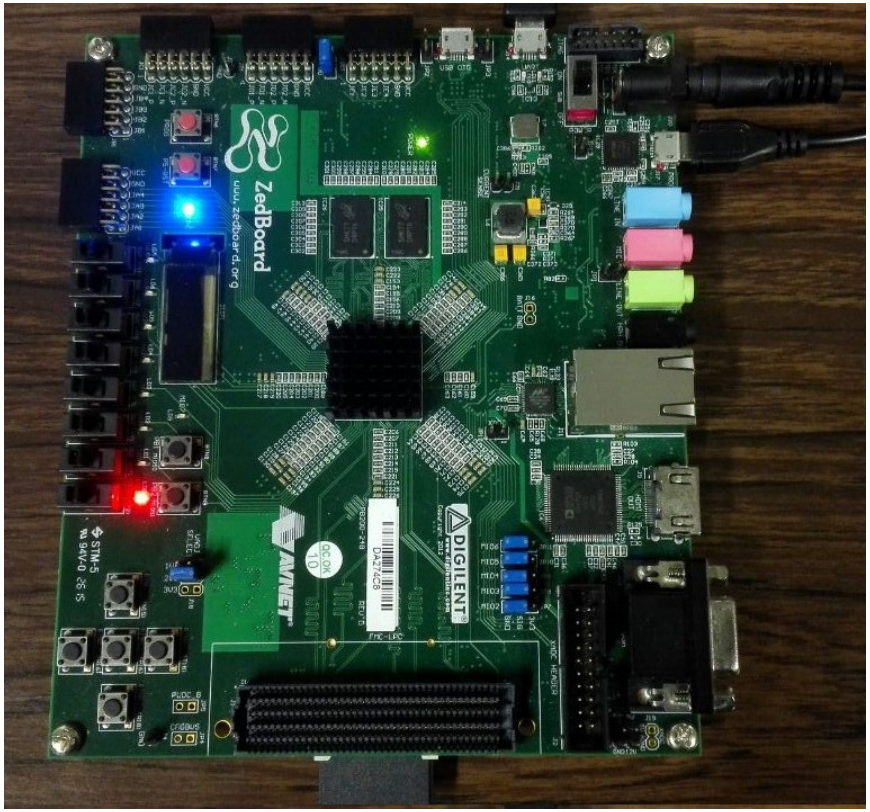
Contatore in FPGA – Controllo Software (1)

```c
#include <stdio.h>
#include <pthread.h>
#include <xscugic.h>
#include <xcounter.h>

#include "platform.h"
#include "xil_printf.h"

#define AXI_GPIO_ADDR       0x41200000
#define INTC_INTERRUPT_ID   61

XCounter counterInstance;

XScuGic Intc;
XScuGic_Config *IntcConf;

void IntCallback(void *InstancePtr);
int SetupInterruptSystem();

void XCounter_Start() { Xil_Out8(AXI_GPIO_ADDR, 1); }
void XCounter_Stop()  { Xil_Out8(AXI_GPIO_ADDR, 0); }

int main() {

    init_platform();

    SetupInterruptSystem();

    XCounter_Initialize(&counterInstance, 0);           // inizializza periferica
    XCounter_Start();                                   // abilita la periferica
    XCounter_Set_range_r(&counterInstance, 100000000);  // trasferisce il range tramite AXILite

    // spinna all'infinito
    while(true);

    cleanup_platform();

    return 0;
}
```

Contatore in FPGA – Controllo Software (2)

```
void IntCallback(void *InstancePtr) {

    // interrupt service routine here
    printf("Il contatore ha terminato!\n");
}

int SetupInterruptSystem() {
  /// Initialize the Interrupt controller driver so that is ready to use
  if(!(IntcConf = XScuGic_LookupConfig(XPAR_PS7_SCUGIC_0_DEVICE_ID))){
      return 1;
  }

  /// Initialize the SCU and GIC, specify edge sensitivity & register callback
  if(XScuGic_CfgInitialize(&Intc,IntcConf,IntcConf->CpuBaseAddress)){
      return 1;
  }

  XScuGic_SetPriorityTriggerType(&Intc, INTC_INTERRUPT_ID, 0xA0, 0x3);

  if(XScuGic_Connect(&Intc, INTC_INTERRUPT_ID, //connect interrupt handler
        (Xil_ExceptionHandler) IntCallback, (void *)&Intc)){
      return 1;
  }

  XScuGic_Enable(&Intc, INTC_INTERRUPT_ID);    // Enable IRQ F2P interrupts

  /// Initialize exception table & register interrupt controller handler with it
  Xil_ExceptionInit();
  Xil_ExceptionRegisterHandler(XIL_EXCEPTION_ID_INT,
      (Xil_ExceptionHandler)XScuGic_InterruptHandler, &Intc);
  Xil_ExceptionEnable();

  printf("GIC Inizializzato\n");

  return 0;
}
```

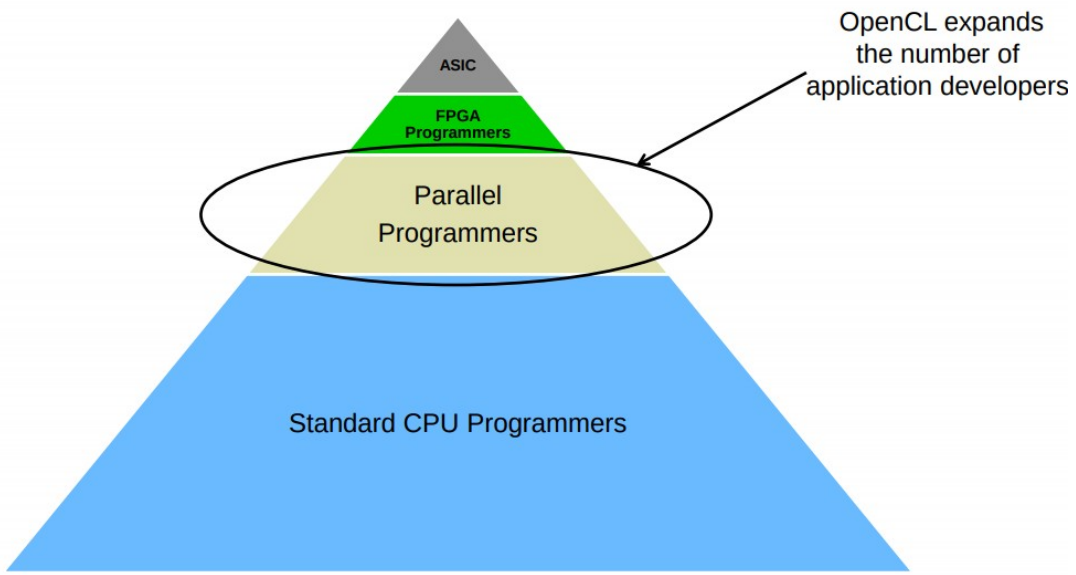
Contatore in FPGA - Funzionamento




```
/bin/bash
/bin/bash
GIC Inizializzato
Interrupt 1
Interrupt 2
Interrupt 3
```

Programmazione FPGA in OpenCL

- *OpenCL:* standard sviluppato dal gruppo *Khronos*, per la programmazione di sistemi eterogenei. Propone un programming model ed un'astrazione hardware unificata per tutti i devices che lo adottano.
- Simile a CUDA.



Programmazione FPGA in OpenCL – Lato Device

- In questo caso, idealmente, viene creato un moltiplicatore in FPGA per ogni elemento dell'array.
- Parallelismo esplicito.
- Il codice OpenCL è convertito in Verilog/VHDL.

Traditional loops

```
void
trad_mul(int n,
         const float *a,
         const float *b,
         float *c)
{
  int i;
  for (i=0; i<n; i++)
    c[i] = a[i] * b[i];
}
```

Parallelism Must be Inferred

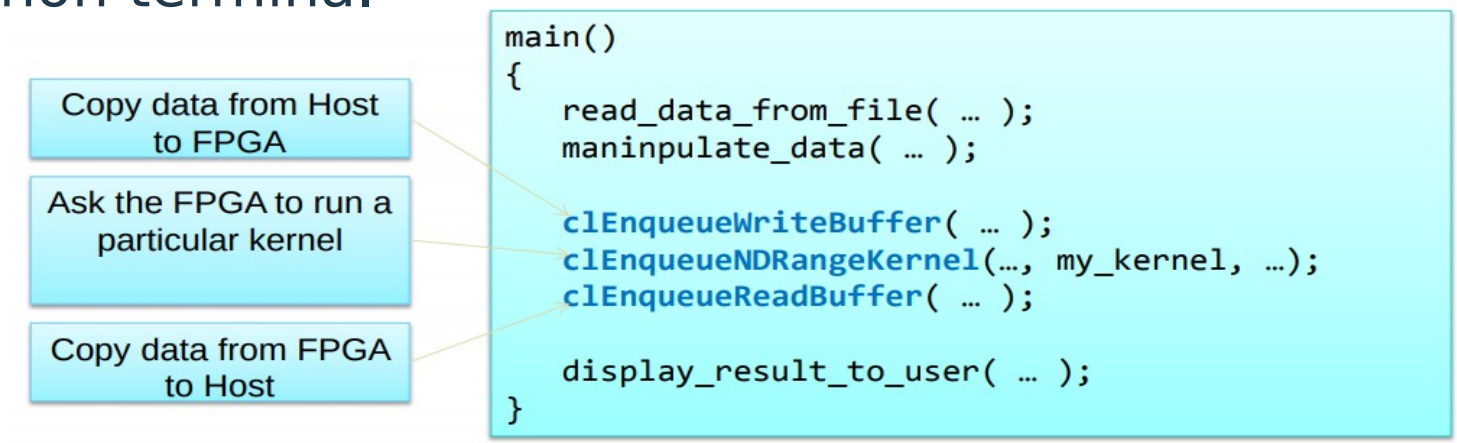
Data Parallel OpenCL

```
kernel void
dp_mul(global const float *a,
       global const float *b,
       global float *c)
{
  int id = get_global_id(0);

  c[id] = a[id] * b[id];

} // execute over "n" work-items
```

Parallelism is Explicit

Programmazione FPGA in OpenCL – Lato Host

- Copia dei dati PS-PL e viceversa tramite API OpenCL.
  - Non è necessario gestire esplicitamente le interconnessioni HW (es: *AXILite*, *AXI Stream* ecc).
- L'Host si blocca sulla *ReadBuffer*, fino a che l'FPGA non termina.



Programmazione FPGA in OpenCL - Compilazione

- Necessari due step di compilazione:
  - *xocc*: (*Xilinx OpenCL Compiler*) per la sintesi del bitstream.
  - *gcc*: per il software che invoca i kernel.
- XOCC Common Options:
  - *--platform*: per specificare la piattaforma target (es. Zed, ZCU102, ...)
  - *--kernel-frequency*: per settare la frequenza del modulo in FPGA.
  - *--target*: per specificare simulazione, emulazione ecc

Programmazione FPGA in OpenCL – Compilazione (2)

- In questo caso, *a runtime* è necessario caricare il file binario generato dal compilatore *xocc.*

```
1.          fp = fopen(fileName, "r");
2.          if (!fp) {
3.                  fprintf(stderr, "Failed to load kernel.\n");
4.                  exit(1);
5.          }
6.          binary_buf = (char *)malloc(MAX_BINARY_SIZE);
7.          binary_size = fread(binary_buf, 1, MAX_BINARY_SIZE, fp);
8.          fclose(fp);
```