Indice

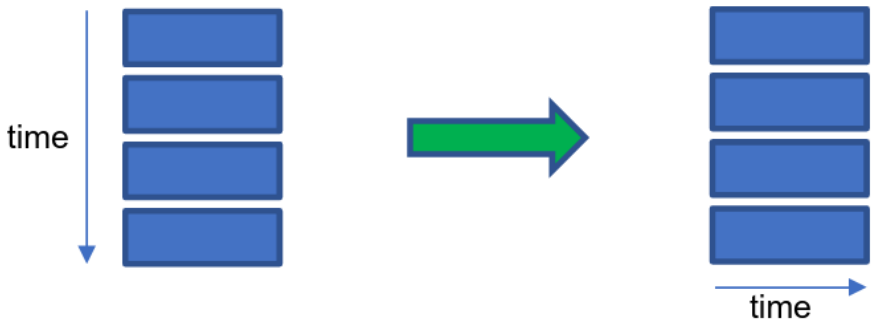
1 — Introduzione al calcolo parallelo

Most real world problems need fast, often real time, solutions.

How to speed up computations

There are multiple strategies, tipically not mutually exclusive, to speed-up computations. One of the most effective ones consists of executing multiple operations simultaneously, in **parallel** or concurrently.

Although there are multiple techniques to achieve parallel computing, the main goal is to overlap the execution of multiple operations or tasks.



1.1 Dipendenza dei dati e corsa sui dati

É necessario introdurre concetti base utili durante l'implementazione di operazioni concorrente o parallele.

Dipendenza dei dati

La **dipendenza dei dati** accade quando un'operazione è dipendente dall'output di un'altra, nel caso di esecuzione sequenziale, questo non è un problema, ma lo diventa in operazioni parallele in quanto una operazione deve rimanere bloccata fino a quando non ha tutti i dati necessari per essere eseguita, magari fornita da operazioni concorrenti.

Corsa sui dati

Quando thread multipli accedono alla stessa memoria in modo concorrente, e almeno uno la modifica, si può incorrere in una **corsa sui dati**. Il risultato non è predicibil, in quanto è influenzato dal timing tra i thread.

1.1.1 Soluzioni

Esistono diverse strategie, che agiscono a livello hardware o software, ma nella maggior parte dei casi, è il design dell'hardware a porre limiti al software.

Hardware

- aumentare la frequenza
- sfruttare pipelining o altre tecniche
- permettere operazioni multiple in una singola istruzone
- aumentare il numero di core

Software

- eseguire operazioni multiple nella stessa istruzione
- distribuire la computazione su CPU multiple
- ridurre la dimensione dei dati o effettuare calcoli meno intensi

1.2 Strategie di design hardware

1.2.1 Aumenare la frequenza

La soluzione più semplice consiste nell'aumentare la frequenza del clock della CPU.

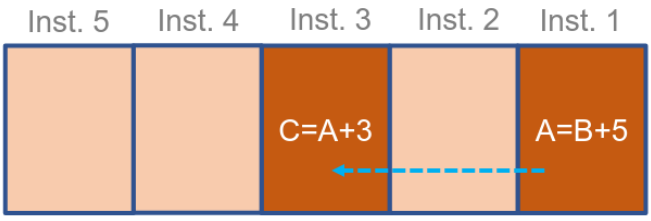
L'idea è semplice: più alta la frequenza, minore il tempo di esecuzione; tuttavia questo comporta un maggiore consumo di energia, deve quindi essere usata cautamente.

1.2.2 Pipelining

É una soluzione molto efficace che permette di eseguire istruzioni multiple concorrenti, anche se non allo stesso "stadio" di esecuzione (non effettivamente in parallelo).

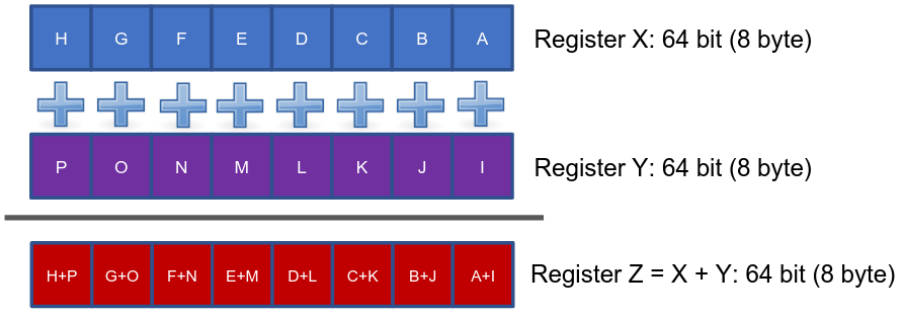
Si ottiene un CPI (Cycles Per Instruction) minore rispetto alla stessa CPU senza pipelining, tuttavia la sua implementazione richiede più componenti.

In questo caso la dipendenza dei dati diventa un problema, risolvibile però facilmente dalla forwarding unit.



1.2.3 Operazioni multiple in una singola istruzione

Questa strategia consiste nell'eseguire operazioni (identiche) multiple in una singola istruzione, un singolo registro contiene dati multipli sui quali eseguire l'operazione in parallelo.



Single Instruction Multiple Data (SIMD) è una tecnica molto potente e al contempo leggera a livello di implementazione.

1.2.4 Aumentare il numero di core

Consiste nel distribuire il calcolo su più unità di esecuzioni, che siano core o complete CPU.

Questa strategia può essere più efficace e meno costosa rispetto all'aumento della frequenza, tuttavia non è sempre implementabile e sono frequenti i problemi relativi ai dati.

1.3 Strategie di design software

1.3.1 Istruzioni SIMD

Un buon compilatore è capace di inferire dal codice le possibili computazione parallele per istruzioni SIMD, tuttavia non è facile.

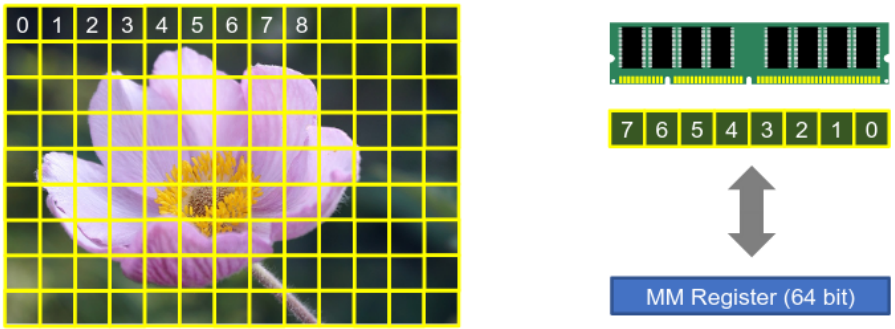
La migliore performance può essere ottenuta organizzando le strutture dati per le SIMD e aggiungere codice assembly nel linguaggio ad alto livello utilizzato, è consigliato applicare questa strategia soltanto alle porzioni più intense del codice.

Questa strategia è efficace quando:

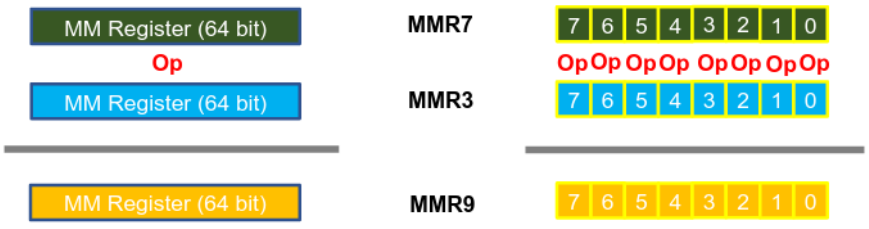
- i dati sono organizzati in modo adeguato in memoria
- la stessa operazione è applicabile a dati multipli

Adatta principalmente all'elaborazione di immagini, contenuti multimediali e segnali, anche se può essere utile anche in altri contesti, come gli algoritmi di sorting.

Se i dati sono appropriamente organizzati in memoria, una singola istruzione può caricare più operandi in un singolo registro multimediale.



Una volta caricati dati impacchettati in registri multimediali, una singola operazione può essere applicata a tutti gli operandi.



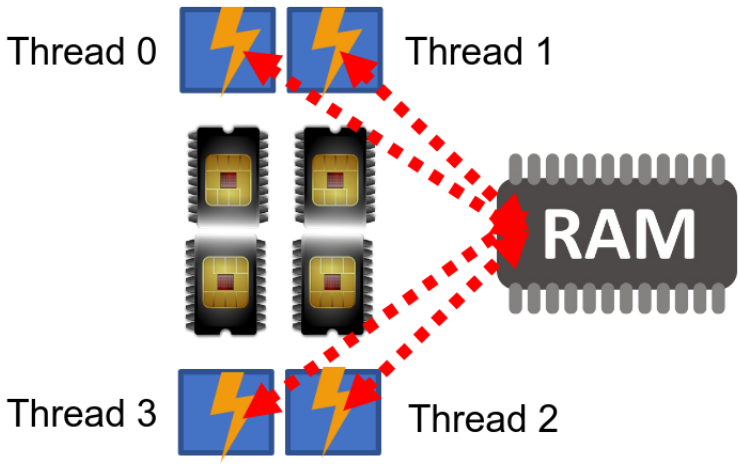
Ovviamente, un MM register più grande significa un maggiore incremento nella velocità (Intel/AMD up to 512 bit).

1.3.2 Sfruttare core multipli

Invece di agire a livello dell'isruzione/dato è possibile aumentare il parallelismo a livello dei thread, eseguendo sottotask multiple in modo concorrente.

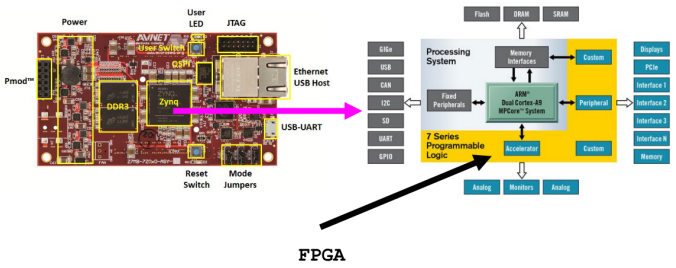
Sfortunatamente, eseguire su core multipli in modo concorrente può risultare in dipendenza dei dati e corsa sui dati portando a risultati non predicibili.

Il motivo risiede nel fatto che ogni thread lavora in modo indipendenta ma su una memoria condivisa che potrebbe contere input, risultati intermedi e risultati di operazioni.



2 — Field Programmable Gate Array

2.1 Introduzione



Una FPGA è composta da **Configurable Logic Blocks** (CLB) che possono essere connessi con linguaggi ad alto livello (HDL o HDS).

Sono riprogrammabili sul campo e hanno un costo molto contenuto, sono supportate inoltre da linguaggio di alto livello come C, C++ o Python.

Sono la soluzione ideale per lo sviluppo veloce di prototipi e introdurli sul mercato in poco tempo. Un'altro grande vantaggio delle FPGA consiste nel fatto che hanno un consumo energetico molto basso.

Permettono in generale un alto livello di astrazione, consentendo l'implementazione di reti logiche e algoritmi a livello hardware.

2.2 Architettura FPGA

Una FPGA è composta da migliaia di CLB, le funzioni e connessioni di ognuno di questi possono essere completamente progettate e riadattate sul campo dallo sviluppatore, che può generare nuove architetture molteplici volte.

Altre caratteristiche:

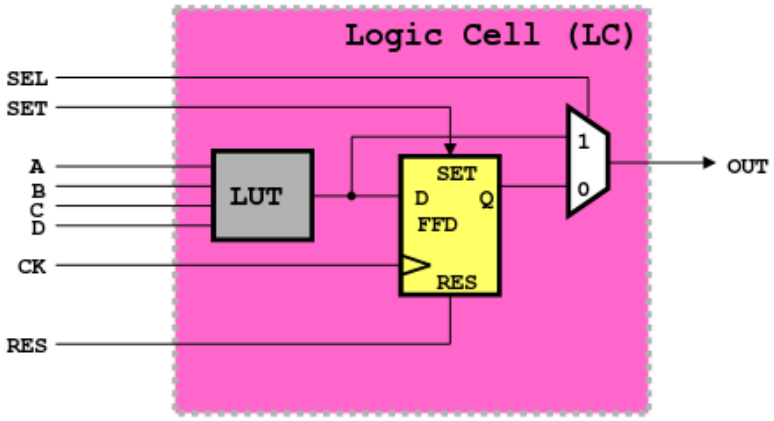
- spesso le FPGA contengono anche RAM integrata (qualche KB)
- alcuni CLB sono dedicate alla gestione dell'I/O

- numerosi adder, multiplier etc.

FPGA prodotte da diversi produttori si distinguono grazie a due aspetti:

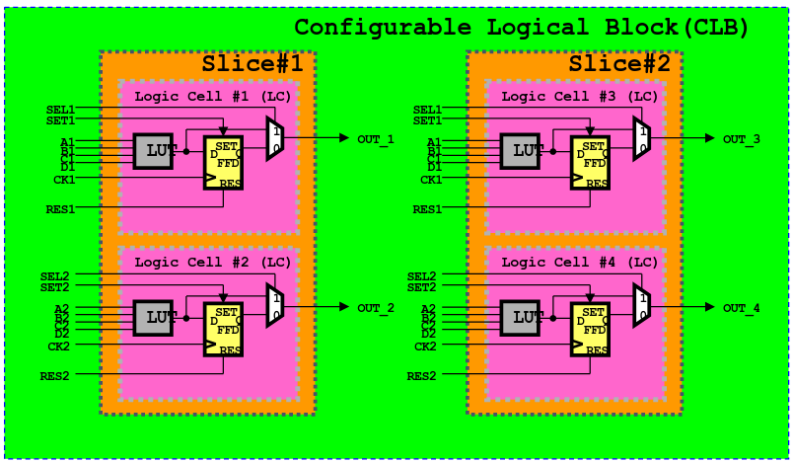
- Tecnologia usata per le connessioni
  - fusibili
  - memorie flash
  - memorie SRAM
- Struttura dei CLB

La figura sotto mostra la rete logica di una **Logic Cell** che implementa un CLB.

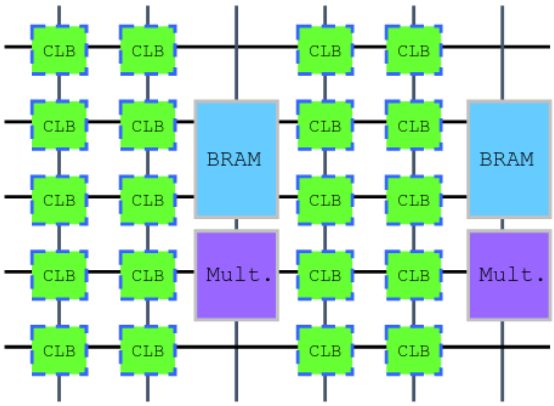


Il blocco LUT è una rete combinatoria programmabile, può essere configurate per agire come uno shift-register o una memoria (RAM distribuita).

Tipicamente le Logic Cell sono raggruppate in **slice**, queste slice possono poi essere organizzate ulteriormente in gruppi per andare a formare un completo CLB.



La struttura effettiva di una FPGA, oltre ai CLB, comprende anche altre unità come la RAM e i multiplier:



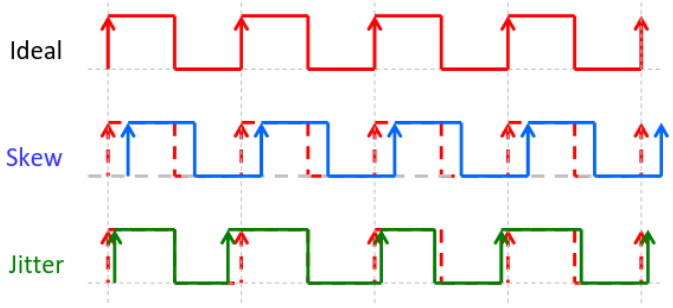
2.2.1 Clock

Le FPGA sono solitamente configurate per implementare reti sincrone, quindi almeno un segnale di clock è necessario, usato per Sequential Synchronous Networks.

Spesso più domini di clock possono coesistere in uno stesso progetto, ad esempio ognuno dedicate a un particolare modulo, questo tuttavia crea problemi in caso sia necessario comunicare tra domini. Per generare questi segnali clock a una frequenza prefissata, i produttori mettono a disposizione componenti specifici, chiamati **DCM** e **PLL**.

Il clock è afflitto da due imprecisioni principali: lo **skew** e il **jitter**. Lo *skew* produce uno sfasamento del del segnale mentre il *jitter* causa una imprecisione dinamica nella frequenza del clock.

Figura 2.1: Skew e jitter



Per ovviare a questi problemi si utilizzano i sopracitati DCM e PLL, aiutano a gestire il segnale del clock permettendo di:

- generare segnali stabili, partendo da un segnale periodico esterno
- ridurre skew e jitter
- generare segnali con uno shift dato rispetto al segnale di reference

2.2.2 IP-core

Nelle FPGA è possibile utilizzare dispositivi comuni tramite gli **IP-core**, la controparte hardware delle librerie software).

Spesso questi IP-core sono basati su funzioni logice cablate nei circuiti dell'FPGA dai produttori (DCM e PLL sono alcuni di questi), alcuni esempi:

- Memory controllers, permettono di trasferire dati con dispositivi di memoria esterni
- Serializer/Deserializer (SERDES), dispositivi per convertire segnali ad alta frequenza da seriale a parallelo (e viceversa)
- Communication controllers, dispositivi per gestire trasferimenti ad alta larghezza di banda

Un IP-core configura una FPGA per implementare una specifica funzionalità, come per il software possono essere a pagamento, possono essere copiati tramite attacchi (SRAM) ma esistono comunque meccanismi di protezione.

2.2.3 Frequenza e performance

Le FPGA non hanno frequenze di clock alte, ottengono pero ottime performance grazie a

3 — Protocolli bus

3.1 Protocolli di comunicazione

I protocolli di comunicazione standard comprendono protocolli per periferiche e dispositivi di memoria, sono stati principalmente proposti da ARM per applicazione industriali e molti di questi si riferiscono a un dominio di applicazione specifico.

- I2C
- ARM AXI
- ARM AXI Lite
- ARM AXI Stream (address less)

3.1.1 Protocollo I2C (I^2C)

Il protocollo I2C, Inter-Integrated Circuit, è stato inizialmente proposto da Philips negli anni 80.

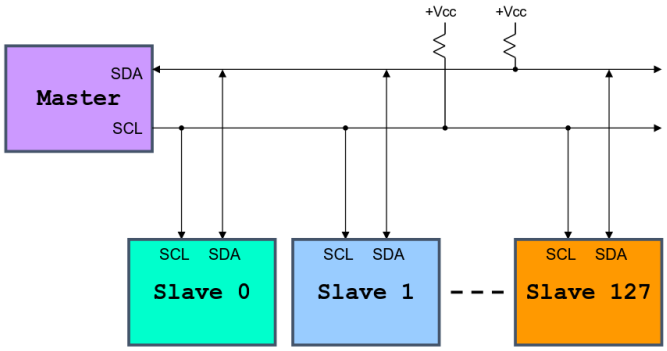
Questo protocollo è altamento diffuso in vari contesti (videocamere, sensori, CPU...) per via della sua semplicità:

- protocollo seriale
- richiede solo 2 fili
- frequenza 100KHz (ora a 400KHz o 1+ MHz)
- semplice e lightweight
- predecessore di SPI

Funzionamento

É presente un **master**, che *inizia sempre la comunicazione*, si hanno un collegamento per i **dati** (**SDA**) e uno per il **clock** (**SCL**). Si possono avere al massimo 127 slave.

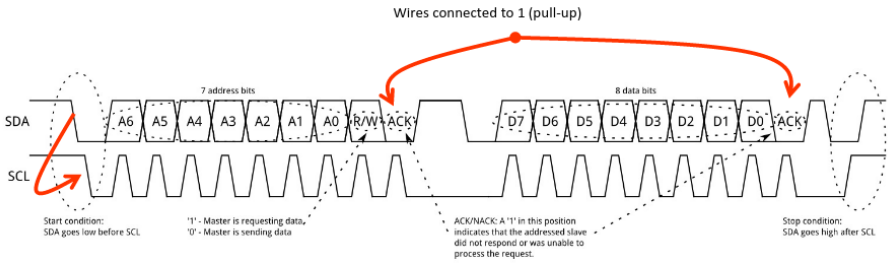
Figura 3.1: Schema di collegamento



Ogni ciclo di bus consiste di una sequenza di *due* sezioni:

- **Address Frame**
  - *master* trasmette l'**indirizzo** dello *slave* coinvolto
  - *master* specifica se è una lettura o una scrittura
- **Data Frame**
  - in caso di scrittura, *master* invia i dati allo *slave*
  - in caso di lettura, *master* legge dati dallo *slave*

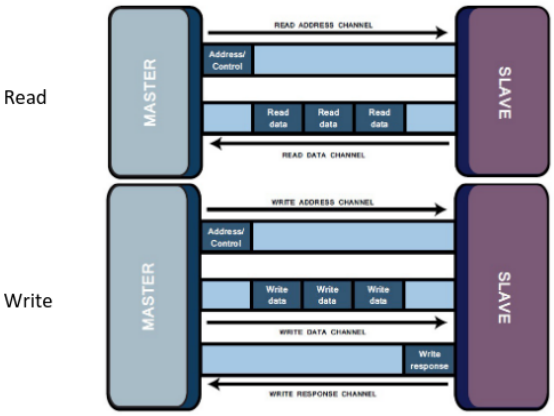
Figura 3.2: Schema di funzionamento



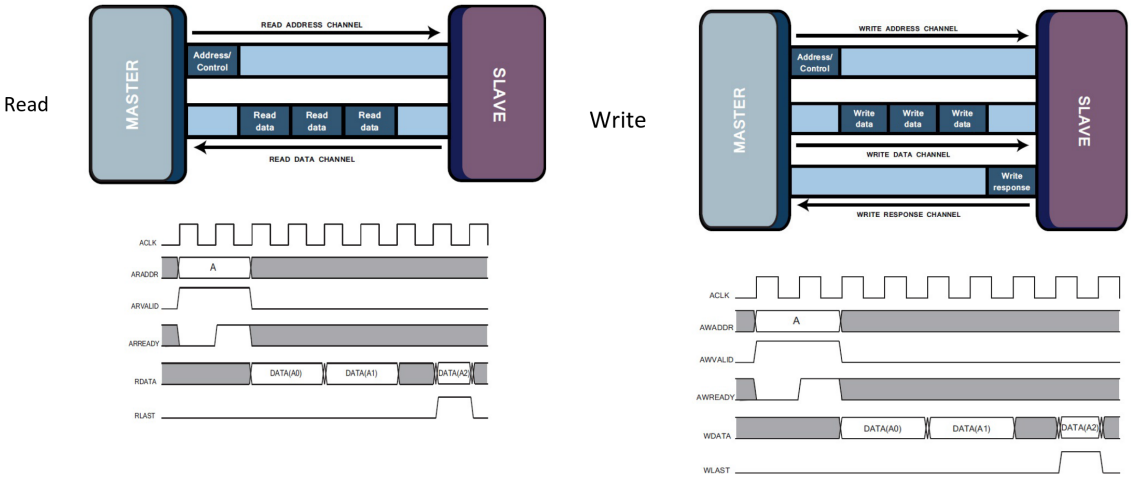
3.1.2 Protocolli on-chip su Zynq

- AXI4: adatto per comunicazioni con dispositivi **memory-mapped** ad alte prestazioni, può gestire 256 trasferimenti dato un singolo indirizzo (*burst transfer*)
- AXI4-Lite: abilita un singolo trasferimento per dispositivi **memory-mapped**.
- AXI4-Stream: adatto a per trasferimenti di stream di dati ad alte prestazioni con dispositivi **non memory-mapped**, spesso utilizzato per gestire stream di immagini o segnali.

Comunicazioni con dispositivi memory-mapped

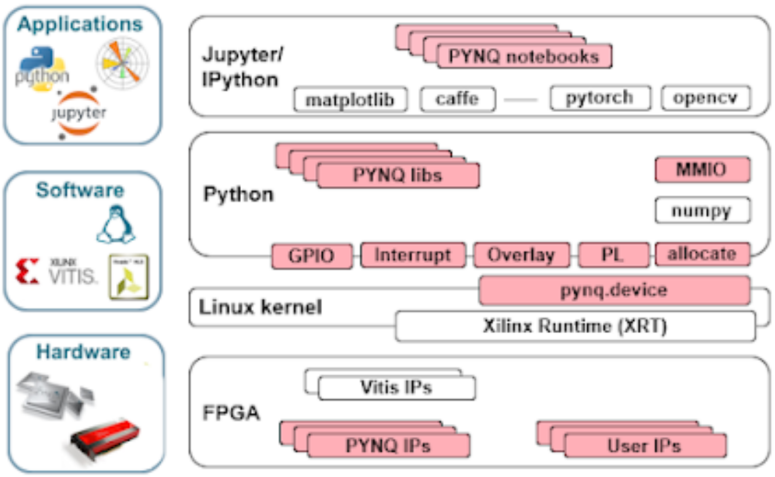


Lettura e scrittura burst



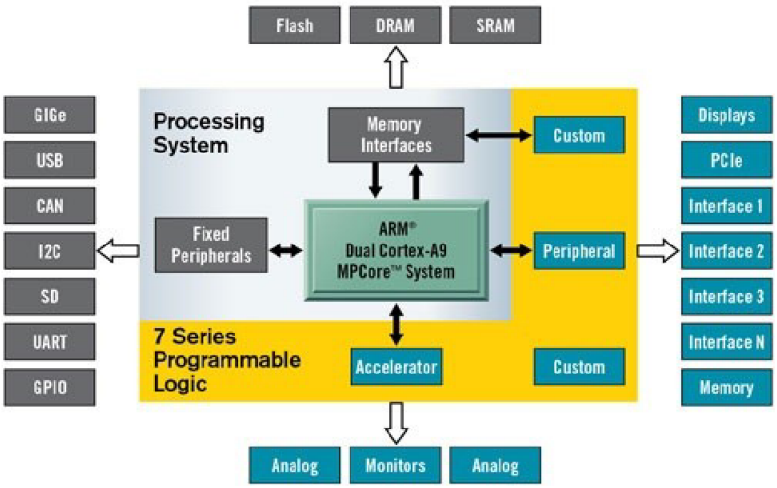
4 — Introduzione a PYNQ board

PYNQ è un progetto open-source gestito da Xilinx che porta le funzionalità di una FPGA a linguaggi ad alto livello.



La PYNQ board monta un processore dual-core ARM Cortex-A9, chiamato **Processing System** (PS), e una FPGA (**Porgrammable Logic**, PL).

Il PS può richiedere l'esecuzione di alcune funzione al PL, utilizzando chiamate a librerie hardware specifiche, chiamate **overlays**.



Possono essere caricate alla FPGA dinamicamente e chiamate quando necessario, in questo gli overlay sono similari a librerie software.

É resa disponibile una interfaccia Python, utile per controllare overlays caricati nel PL da codice Python eseguito sul PS.

Primi passi

La board è provvista di una scheda SD con una versione di PYNQ, una distro linux custom.

Può essere collegata direttamente a un PC, e una volta accesa e collegata a un router, è possibile accedere a un web server, solitamente all'indirizzo `http://pynq:9090` (se connesso tramite ethernet) o a `http://192.168.2.99:9090`.

4.1 Overlay

Gli overlay permettono di utilizziare IP di basso livello tramite Python. Un overlay carica un **bitstream** per permettere l'accesso alle sue funzionalità: per questo si utilizzano oggetti **driver**.

Esempio

Ad esempio, si considera una semplice funzione HLS che restituisce la somma di due int a 32 bit.

```
void add(int a, int b, int &c) {
    #pragma HLS INTERFACE ap_ctrl_none port=return
    #pragma HLS INTERFACE s_axilite port=a
    #pragma HLS INTERFACE s_axilite port=b
    #pragma HLS INTERFACE s_axilite port=c

    c = a + b;
}
```

Dopo aver progettato l'IP è possibile importarla in python creando un overlay tramite il pacchetto `pynq`, usato per caricare l'IP dal bitstream:

```
from pynq import Overlay

overlay = Overlay('/home/cilinx/tutorial_1.bit')
add_ip = overlay.scalar_add
help(add_ip)
```

Con la funzione help è possibile ottenere informazione sull'oggetto IP estratto dall'overlay.

L'IP core è caricato come oggetto base `DefaultIP`, un driver che espone API di `read` e `write`.

Per l'esempio dell'add, secondo la documentazione, è necessario scrivere i due parametri della funzione agli offset `0x10` e `0x18` e leggere il risultato da `0x20`.

```
add_ip.write(0x10, 4)
add_ip.write(0x18, 5)
add_ip.read(0x20) # 9
```

```
class DefaultIP(builtins.object)
 |  Driver for an IP without a more specific driver
 |
 |  This driver wraps an MMIO device and provides a base class
 |  for more specific drivers written later. It also provides
 |  access to GPIO outputs and interrupts inputs via attributes. More specific
 |  drivers should inherit from `DefaultIP` and include a
 |  `bindto` entry containing all of the IP that the driver
 |  should bind to. Subclasses meeting these requirements will
 |  automatically be registered.
 |
 |  Attributes
 |  ----------
 |  mmio : pynq.MMIO
 |      Underlying MMIO driver for the device
 |  _interrupts : dict
 |      Subset of the PL.interrupt_pins related to this IP
 |  _gpio : dict
 |      Subset of the PL.gpio_dict related to this IP
 |
 |  Methods defined here:
 |
 |  __init__(self, description)
 |      Initialize self.  See help(type(self)) for accurate signature.
 |
 |  read(self, offset=0)
 |      Read from the MMIO device
 |
 |      Parameters
 |      ----------
 |      offset : int
 |          Address to read
 |
 |  write(self, offset, value)
 |      Write to the MMIO device
 |
 |      Parameters
 |      ----------
 |      offset : int
 |          Address to write to
 |      value : int or bytes
 |          Data to write
 |
 |  ----------------------------------------------------------------------
 |  Data descriptors defined here:
 |
 |  __dict__
 |      dictionary for instance variables (if defined)
 |
 |  __weakref__
 |      list of weak references to the object (if defined)
```

Il driver di defualt non è user-friendly, una soluzione migliore è quella di creare una classe che estende il driver defualt e espone un metodo `add(a, b)` che implementa le operazioni necessarie.

```
class AddDriver(DefaultIP):
    def __init__(self, description):
        super().__init__(description=description)

    bindto = ['xilinx.com:hls:add:1.0']

    def add (self, a, b):
        self.write(0x10, a)
        self.write(0x18, b)
        return self.read(0x20)
```

Ora è possibile accedere l'IP usando una singola `add` API, resa disponibile dal driver custom.

Driver inclusi

- AXI GPIO
- AXI DMA
- AXI VDMA
- AXI Interrupt Controller
- Pynq-Z1 Audio IP

- Pynq-Z1 HDMI IP
- Color convert IP
- Pixel format conversion
- HDMI input and output frontends
- Pynq Microblaze program loading
- ...

4.2 PYNQ Hello LED

Gli overlay base che controllano le periferiche presenti sulla PYNQ, disponibili caricando il `BaseOverlay` da `pynq.overlay.base`:

```
import time
from pynq.overlays.base import BaseOverlay
base = BaseOverlay('base.bit')
```

É possibile, ad esempio, accedere a singoli LED e accenderli/spegnerli.

```
led0 = base.leds[0]
led1 = base.leds[1]
led2 = base.leds[2]
led3 = base.leds[3]

led0.on()
led0.off()
```

É anche possibile utilizzare switch per interagire con i led, per prima cosa si ottengono i LED, switch e tasti disponibili.

```
MAX_LED = 4
MAX_SWITCHES = 2
MAX_BUTTONS = 4

leds = [0] * MAX_LEDS
switches = [0] * MAX_SWITCHES
buttons = [0] * MAX_BUTTONS

for i in range(MAX_LEDS):
    leds[i] = base.leds[i]
for i in range(MAX_SWITCHES):
    leds[i] = base.switches[i]
for i in range(MAX_BUTTONS):
    leds[i] = base.buttons[i]
```

INSERIRE25-26

4.3 HDMI Stream

La board mette a disposizione driver per gestire le porte HDMI:

```
from pynq.overlays.base import BaseOverlay
from pynq.lib.video import *

base = BaseOverlay('base.bit')
hdmi_in = base.video.hdmi_in
hdmi_out = base.video.hdmi_out

# HDMI handlers configuration
hdmi_in.configure()
hdmi_out.configure(hdmi_in.mode)

hdmi_in.start()
hdmi_out.start()
```

Come primo passo, è possibile semplicemente replicare l'input stream sull'HDMI output utilizzando `hdmi_in.tie(hdmi_out)`.

Con lo stesso risultato, è possibile inoltrare frame per frame da input ad ouput, utilizzando i metodi `readframe` e `writeframe`.

```
import time

numframes = 600
start = time.time()

for \_ in range(numframes):
    f = hdmi_in.readframe()
    hdmi_out.writeframe(f)

end = time.time()
print("Frames per second: " + str(numframes / (end - start)))
```

Questo approccio permette di elaborare singoli frame prima di inviarli all'output.

```
import cv2
import numpy as np

numframes = 10
grayscale = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                       dtype=np.uint8)
result = np.ndarray(shape=(hdmi_in.mode.height, hdmi_in.mode.width),
                    dtype=np.uint8)
```

```
start = time.time()

for _ in range(numframes):
    inframe = hdmi_in.readframe()
    cv2.cvtcolor(inframe, cv2.COLOR_BGR2GRAY, dst=grayscale)
    inframe.freebuffer()
    cv2.Laplacian(grayscale, cv2.CV_8U, dst=result)

    outframe = hdmi_out.newframe()
    cv2.cvtcolor(result, cv2.COLOR_GRAY2BGR, dst = outframe)
    hdmi_out.writeframe(outframe)

end = time.time()
print("Frames per second: " + str(numframes / (end - start)))
```

Ricordare di chiudere gli stream

```
hdmi_out.close()
hdmi_in.close()
```