RIASSUNTO SISTEMI DIGITALI

Sommario

MODULO 1

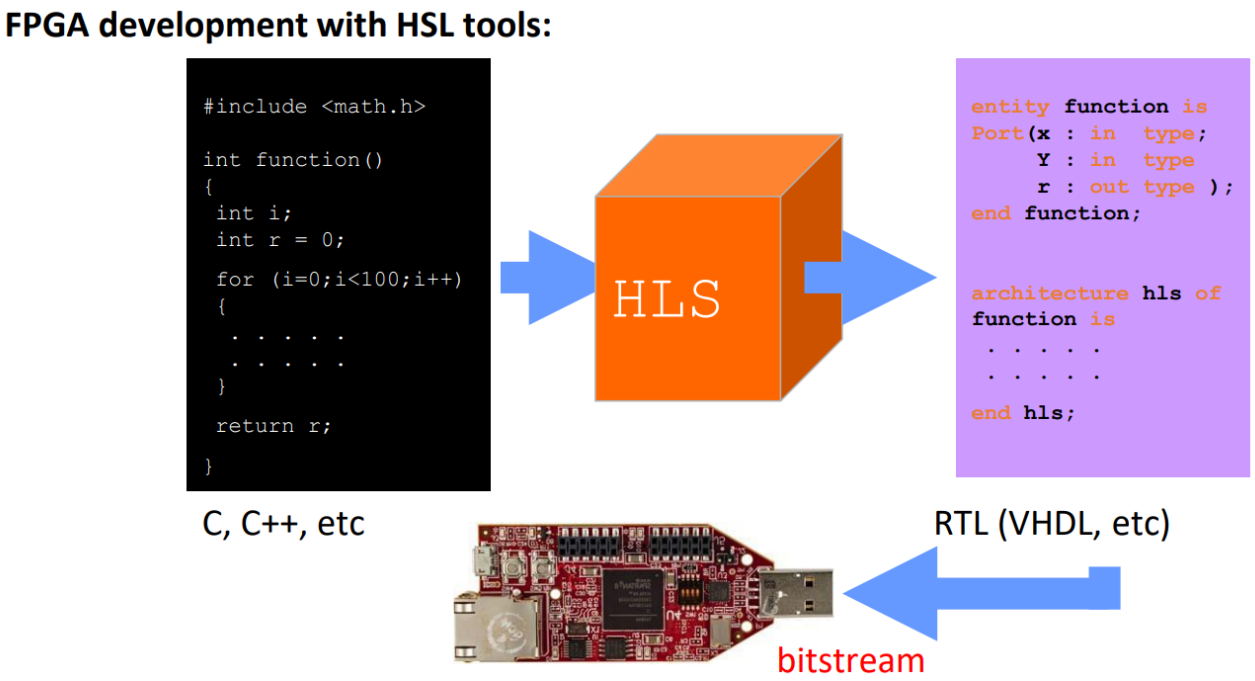
FPGA

Anni 70 = tutto si faceva con reti logiche, scomodo!

Anni 80= nascono i dispositivi programmabili (es. EPROM)

Anni 90= FPGA

Schede elettroniche (evaluation board) contenenti una parte di logica riprogrammabile (PL, ossia la FPGA propriamente detta), un processore (PS), e delle parti aggiuntive come memorie, pulsanti, led, periferiche eccetera. Un esempio sono la Zynq e la Pynq, entrambe prodotte da Xilinx, equipaggiate con due processori ARM e contenti una memoria condivisa tra PS e PL per permettere la comunicazione (che però può avvenire anche tramite appositi canali, vedi "bus protocols"). La FPGA (PL) può essere configurata per implementare una rete logica a piacere, ottimo per creare ottimizzazioni hardware ad hoc (mappo una rete logica sulla FPGA tramite software). L'utilizzo di queste board è il seguente: programmo una rete logica nella FPGA per creare un'ottimizzazione hw a piacere, sul processore faccio girare il mio codice (magari python) che richiama la FPGA a bisogno. La FPGA mi permette di creare accelerazioni specifiche. La FPGA in genere viene programmata tramite HDL (Hardware description language). I linguaggi di HDL (come Verilog o VHDL) sono molto a basso livello. Per ovviare a ciò esistono tool di sintesi ad alto livello (HLS) che permettono di programmare una FPGA con linguaggi come il C. In questo caso il C non viene usato come siamo abituati. Si specifica in una funzione C quello che sarà il comportamento della rete logica che verrà installata nella FPGA. Il codice C verrà tradotto in HDL e l'HDL in bitstream (il sw che contiene la descrizione della rete logica). A fare ciò è un opportuno compilatore speciale.

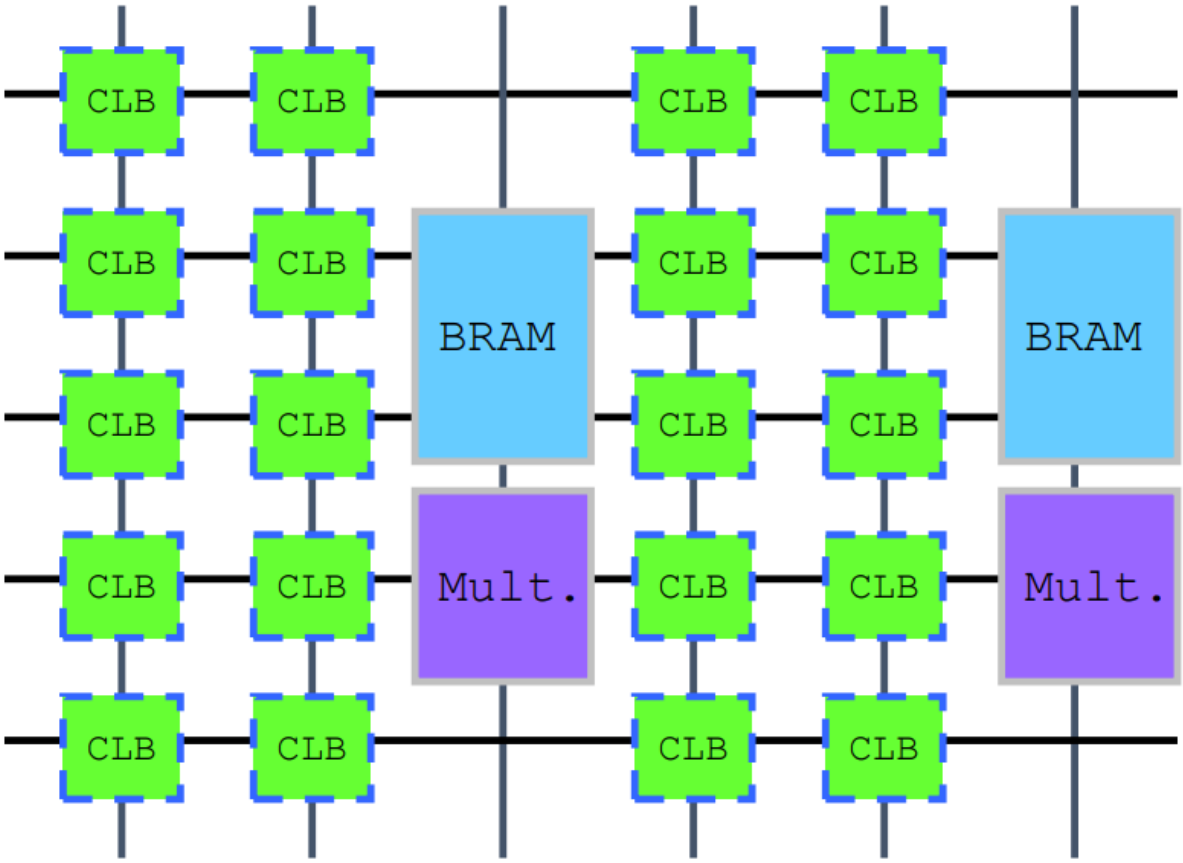


Il bitstream ha sempre una dimensione fissa.

Per creare blocchi custom si può usare il tool Vivado HLS. Per unire blocchi custom e/o blocchi già esistenti in una rete logica finale e convertire tutto in bitstream da installare su FPGA si usa il tool Vivado. Per programmare la CPU ARM si usa VivadoSDK.

Le board sono a basso consumo e ideali per sistemi embedded. Per risparmiare energia hanno una frequenza di calcolo minore rispetto alle CPU dei PC desktop. Per non perdere performance le board sono fatte in modo da sfruttare al meglio il parallelismo delle operazioni.

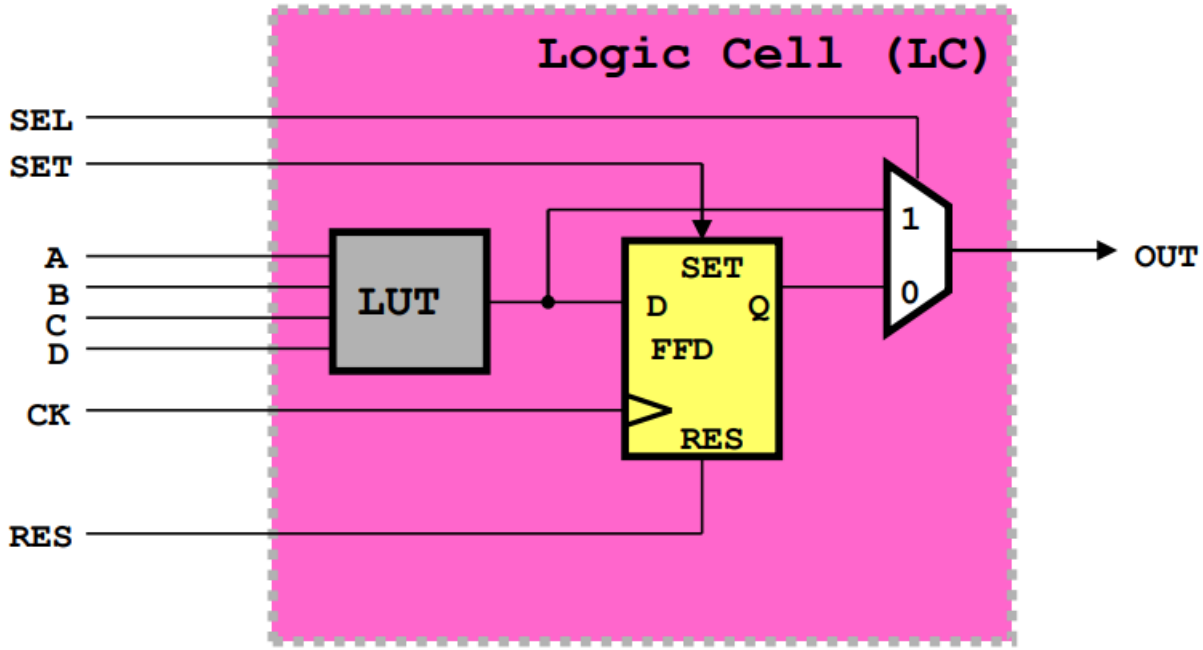
Le FPGA sono composte da CLB, blocchi elementari. Sono blocchi logici configurabili cioè programmabili all'infinito (a differenza delle board ASIC che sono programmabili una sola volta, sono ancora più economiche di una FPGA e si usano per la produzione su larga scala, mentre le FPGA si usano per la prototipazione). I CLB sono interconnessi tra loro. Compito del progettista è quello di specificare (via sw HDL/HLS) il comportamento dei CLB e delle interconnessioni. Le FPGA consumano poco, sono economiche e si usano per progettare/prototipare. Nella FPGA posso implementare non solo reti logiche ma veri e propri algoritmi. I CLB sono semplici e presenti a migliaia in una FPGA. La programmazione di una FPGA è ripetibile quante volte vuoi. Nelle board sono presenti anche periferiche di I/O (connesse ai pin della FPGA e configurabili a piacere per qualsiasi tipo di segnale) e della RAM. Questa RAM è distribuita in tanti piccoli blocchetti di memoria sparsi nella FPGA chiamati BLOCK RAM. Ciascuno blocco è indipendente (accessi multipli consentiti). Inoltre, nella FPGA troviamo unità di calcolo specializzate riconfigurabili (si occupano di un compito specifico e lo fanno velocemente → ottimizzazione). In genere queste unità sono moltiplicatori. Difatti, la moltiplicazione è un'operazione che capita di fare spesso e notoriamente rappresenta un collo di bottiglia. Ha senso quindi per non sprecare CLB investire un poco di silicio in più per ottenere ottimizzazioni particolari. Ad ogni modo sono anche questi riconfigurabili come quasi tutto in una FPGA.



Esistono molti modelli, produttori e tecnologie differenti di FPGA. In genere sono due gli aspetti che discriminano le FPGA:

- Struttura dei CLB
- Tecnologia usata per le connessioni:
    - Fusibili (programmabili una sola volta, scomodi, usati solo nello spazio perché immuni alle radiazioni)
    - Flash mem
    - SRAM

L'unità elementare dei CLB è la CELLA LOGICA.

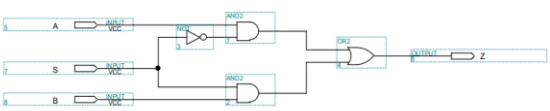


La LUT è una LOOK-UP TABLE. Implementa una funzione combinatoria. L'uso di tabelle è comodo perché le tabelle hanno un tempo di accesso costante. Può essere usata anche come RAM o shift register. Può essere vista come una EPROM dove l'ingresso (in foto ABCD) rappresenta l'indirizzo del dato da mandare in output.

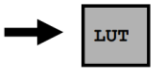
Il FFD è la parte sequenziale sincrona (le FPGA sono usate per implementare reti combinatorie o sincrone, NO RETI ASINCRONE!). Se SEL=1 → SOLO COMBINATORIA (FFD INUTILIZZATO). Il FFD può essere visto come una memoria (così come la LUT). Si viene quindi a creare una gerarchia di cache:

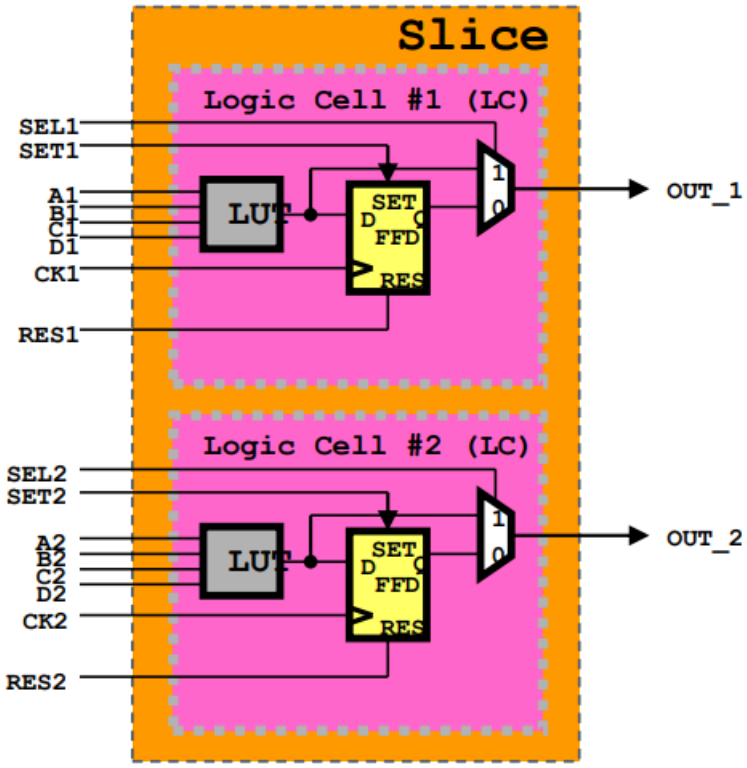
- Eventuale memoria esterna (se presente usata di solito per caricare dati nelle block ram)
- Block ram
- FFD/LUT

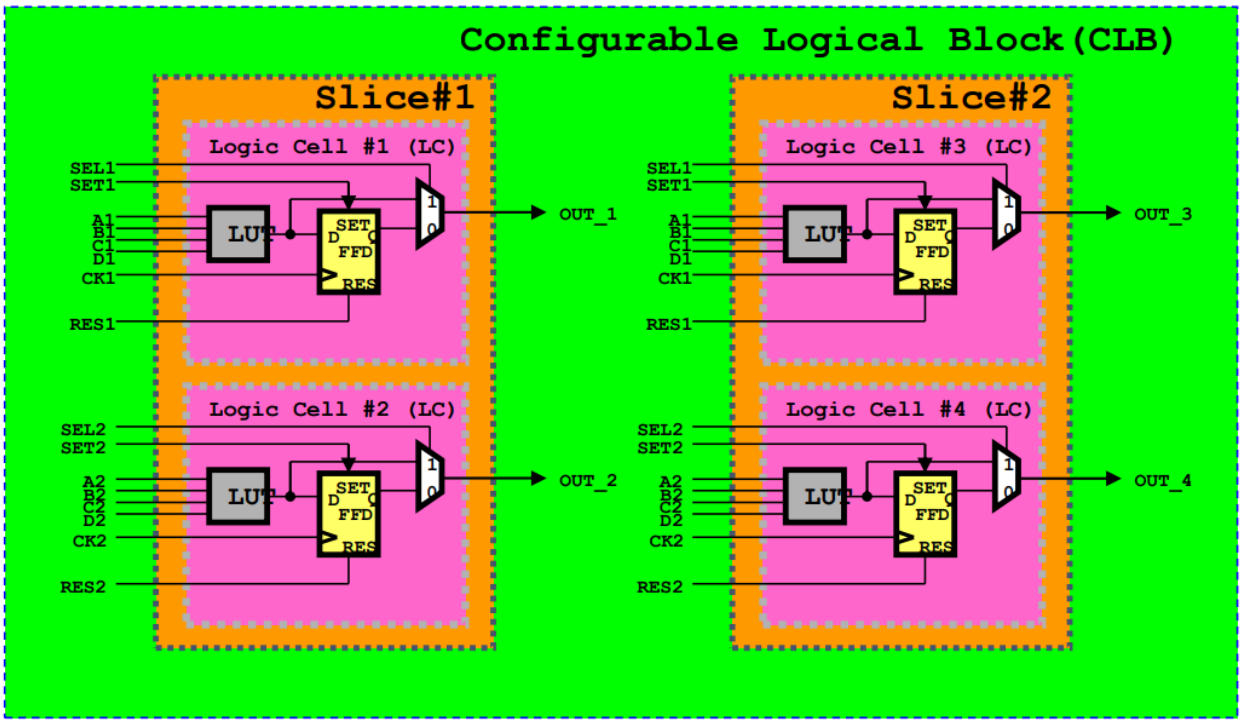
Dove man mano che si scende aumenta la velocità (obv).



S	A	B	Z
0	0	0	0
0	0	1	0
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	0
1	1	1	1







Le celle logiche sono raggruppate in SLICE. I CLB contengono più slice.

Nelle FPGA vi è almeno un segnale di clock. In genere vi è un segnale di clock per ogni unità funzionale (occhio al passaggio di dati tra clock domains distinti). Ci sono due problemi legati al clock:

- SKEW: segnale anticipa o ritarda il clock (problema di fase)
- JITTER: segnale cambia frequenza dinamicamente.

Skew e jitter non sono mutualmente esclusivi (purtroppo). Per generare segnali stabili a una data frequenza si usano PLL o DCM. Non sono altro che moltiplcatori/divisori. Servono per generare un segnale stabile a partire da un segnale periodico esterno (cambiandone la frequenza volendo), riducono skew e jitter, generano segnali shiftati a partire da un certo segnale di riferimento. DCM e PLL sono esempi di IPCORE. Gli IPCORE sono l'equivalente di una libreria sw nel mondo hw. Gli IPCORE sono l'unità di rilascio. In una FPGA di solito oltre a DCM/PLL si trovano anche altri IPCORE come:

- Memory controller
- Serdes
- Communication controller.

BUS PROTOCOLS

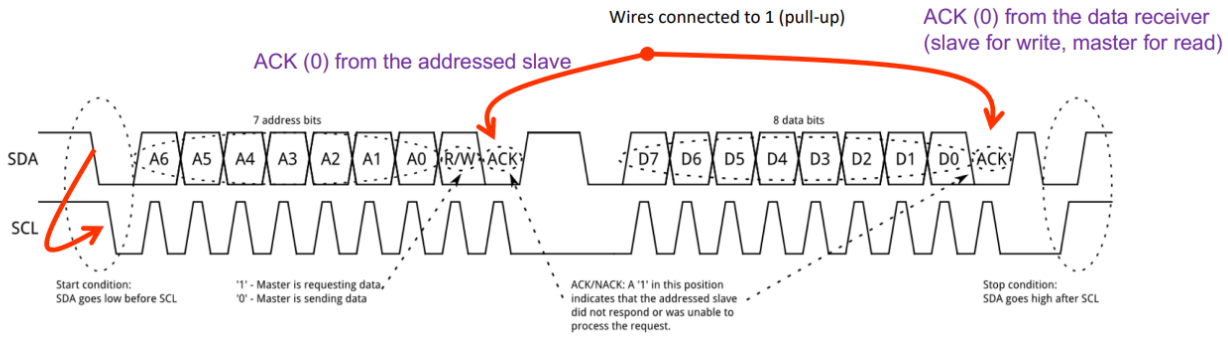
Per la comunicazione FPGA/mondo esterno o FPGA/ARM. Protocolli per periferiche e memorie.

Protocolli: I2C, ARM AXI, ARM AXILITE, ARM AXI Stream

I2C

Protocollo anni 80, molto diffuso perché semplice e leggero. Usa solo due fili: uno per il clock (SCL) e uno per i dati (SCL). I dati viaggiano in modo SERIALE (1 solo filo). Il suo acerrimo nemico è SPI (che però usa 4 fili che scarso!). Protocollo di tipo master/slave. Gli indirizzi sono a 7 bit, i dati a 8 bit. Ogni ciclo di bus è diviso in due parti:

- Address frame (il master specifica indirizzo dello slave con cui vuole interagire e dice se sarà lettura o scrittura)
- Data frame (scambio dati)



All'inizio SDA e SCL sono pullappati a VCC. Poi prima SDA e poi SCL vanno a 0 (start condition). Gli slave sono in allerta. Il master emette l'indirizzo a 7 bit dello slave aiutandosi con il clock (emesso sempre dal master) per indicare quando un bit è da considerarsi significativo. Segue un ottavo bit che indica se sarà una lettura o scrittura (R=1, W=0). Se lo slave è presente risponde con un ACK (stop bit) ossia porta a 0 SDA. Poi SDA=1 e SCL=0. Fine prima fase. Poi SDA=0. Inizio seconda fase. Scambio dati. Il clock è sempre emesso dal master. Anche qui alla fine stop bit di ACK (SDA=0). Poi SDA=1 e SCL=0. Poi SDA=0. Poi stop condition: SCL=1 e poi anche SDA=1. Fine ciclo di bus.

AXI (by ARM)

- AXI = memory mapped, veloce, burst transfer
- AXI LITE = memory mapped, economico, meno veloce
- AXI Stream = address less

AXI

Complesso e veloce, master/slave, frequenza settabile, memory mapped

Indirizzi 32 bit, Parallelismo dati settabile tra 32 e 1024 bit

Segnali di controllo (ack da parte dello slave in caso di scrittura)

Supporta i burst transfer → trasferimenti (lettura o scrittura) in sequenza di blocchi contigui. Se per esempio devo leggere un array quello che dovrei normalmente fare per ogni dato è EMETTO INDIRIZZO e poi LEGGO specificando quindi ogni volta l'indirizzo e creando un certo overhead. Con il ciclo burst quello che faccio è specificare solo la prima volta da quale indirizzo leggere (o scrivere) e per quanto, poi per tutti i cicli successivi non è necessario specificare l'indirizzo. Axi lite non supporta i cicli burst. Per la presenza dei cicli burst Axi è meglio per le memorie. Mentre Axi Lite è meglio per le periferiche perché richiede meno logica. I cicli burst sono ottimi per i line feed della cache.

Axi e axilite usabili sia in arm che in fpga

AXI STREAM

Master/slave, no memory mapped, leggero, semplice, posso settare il parallelismo dati, lo posso usare solo nella fpga

Non ha indirizzi (address-less): i dati vengono passati come un unico flusso sempre nello stesso ordine in modo sequenziale. No random access! Ottimo per quei dati che so che devono essere letti sempre nello stesso modo (es immagini). Siccome le CPU capiscono solo indirizzi non posso usare AXI STREAM nella PS. Se ho proprio bisogno degli indirizzi quello che posso fare è ricevere i dati e metterli in una struttura con indirizzi e poi andare a leggere da lì.

Devo sapere quando inizia il flusso → sincronizzazione → HANDSHAKING

HLS

Vivado HLS → tool eclipse-based per sintesi ad alto livello di moduli custom (da esportare e usare su Vivado) sequenziali sincroni (c'è il clock ma non si vede)

Variabili volatile per evitare ottimizzazioni da parte del compilatore

Uso del C per descrivere comportamento di una rete

Header file in cui dichiaro la firma della funzione di top level + tipi di dato che mi servono (includendo "ap_int.h" posso creare tipi di dato custom, es. 1 bit se devo pilotare un solo filo).

File .cpp in cui scrivo la logica

La logica che specifico nella funzione di top level verrà implementata in una rete. Non sto scrivendo codice da eseguire ma sto descrivendo comportamento → CAMBIO DI PARADIGMA

Se ho bisogno di mantenere valori tra più interazioni (es. conteggio) → uso di variabili static

Per entrare meglio nella mentalità dobbiamo immaginarci che la funzione venga richiamata a ogni ciclo di clock → NB: tecnicamente però la funzione non viene "chiamata" da nessuno, non viene nemmeno eseguita, semplicemente verrà tradotta in HDL!!!!!

Testbench → codice che scrivo per verificare che tutto funzioni

In vivado hls posso associare a un segnale un protocollo:

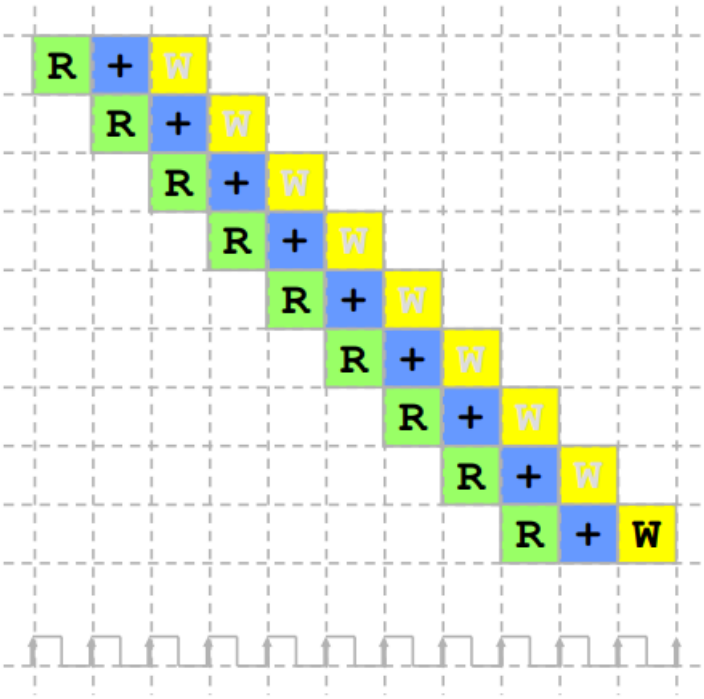
- ap_none: nessun aggiunta
- ap_stable: come prima ma per ingressi che cambiano di rado
- ap_ack: per gli input aggiunge un ACK di output che va alto quando l'input è letto, per gli output aggiunge ACK di input che dopo ogni scrittura bisogna aspettare vada alto prima di proseguire
- ap_vld: aggiunge segnale di validità
- ap_ovld: come prima ma solo x output
- ap_hs: handshaking semplice, robusto, affidabile, veloce
- ap_memory: x le memorie (protocollo costoso)
- bram: come prima ma per le bram
- ap_fifo: come ap_memory ma più semplice, address-less, coda fifo
- ap_bus: generic bus interface, ottimo x bus bridge, supporta lettura/scrittura sia singola sia burst
- axis: axi stream
- s_axilite: interfacce per slave axi lite
- m_axi: per master axi

Latency: è il numero di cicli di clock necessari al modulo per generare il risultato a fronte di un nuovo input
Interval: numero di cicli di clock necessari prima di poter elaborare un nuovo dato

Latency e interval non sono la stessa cosa! Si pensi al DLX: nel caso del DLX sequenziale Latency e Interval sono coincidenti mentre nel caso del DLX pipelined la Latency è 5 clock (senza stalli) mentre Interval è pari a 1. Simili metodologie si applicano per rendere i moduli HLS più efficienti mediante delle opportune direttive.

OTTIMIZZAZIONI IN VIVADO

PIPELINING



Per ottimizzare i loop posso ricorrere al pipelining → separo il loop in vari stadi che possono eseguire in contemporanea

Lo si fa aggiungendo una semplice riga di codice #pragma HLS PIPELINE II=1 → molto ad alto livello → si occupa di tutto il compilatore

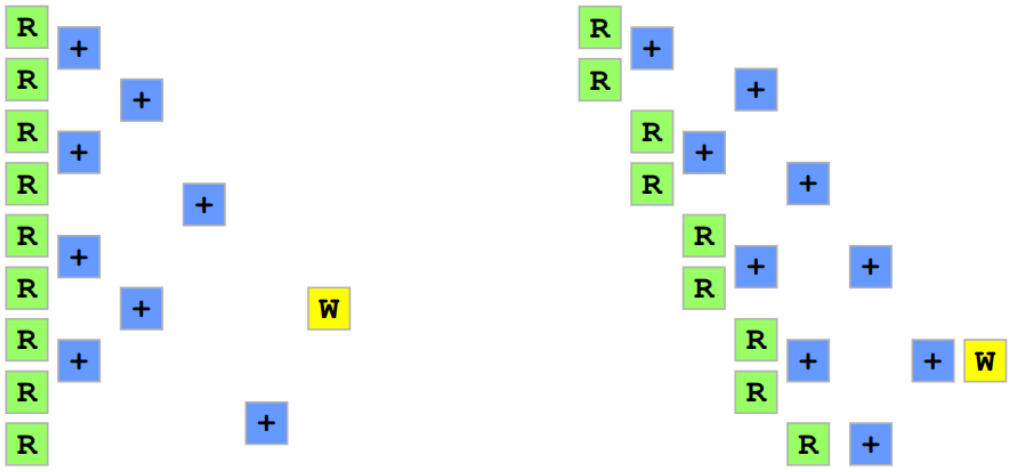
Il parametro II indica l' INTERATION INTERVAL → valore di interval minimo desiderato (in genere sempre 1) → possibilmente voglio elaborare 1 dato a clock (non è detto che il compilatore ci riesca, potrebbe essere maggiore il valore effettivo).

LOOP UNROLLING

Accumuli parziali → faccio più somme parziali in parallelo → necessità di avere più risorse allocate → replicazione delle unità funzionali

Ottimizzazioni in Vivado HLS: loop unrolling

Anche per il loop unrolling non è necessaria alcuna modifica al codice, è sufficiente una #pragma



Inoltre sto imponendo il vincolo che i dati supportino l'accesso multiplo (non sempre possibile) → vincolo forte e costoso → servono strutture dati ad hoc e maggior uso di unità funzionali

MODULO 2

PYTHON

Python è un linguaggio interpretato, object-oriented, portabile, flessibile e dinamicamente tipato. È pensato principalmente per lo scripting ma può essere usato anche per fare applicativi. Una volta installato l'interprete si può già iniziare a lavorare in python. Molto facile da imparare ed estremamente diffuso anche in ambito di ricerca. È dinamicamente tipato quindi i tipi esistono ma non si vedono. Non serve esplicitare il tipo di una variabile perché viene dedotto dall'interprete. Inoltre, una variabile può cambiare tipo, cioè è possibile assegnarle un valore di tipo diverso dal tipo del valore corrente. Bisogna comunque verificare la compatibilità di tipi per evitare errori a runtime. Non è necessario dichiarare le variabili, basta usarle al volo. Una variabile definita dentro una funzione è locale ad essa, cioè non visibile al di fuori della funzione. Le variabili global sono accedibili in sola lettura dentro le funzioni (a meno del costrutto global).

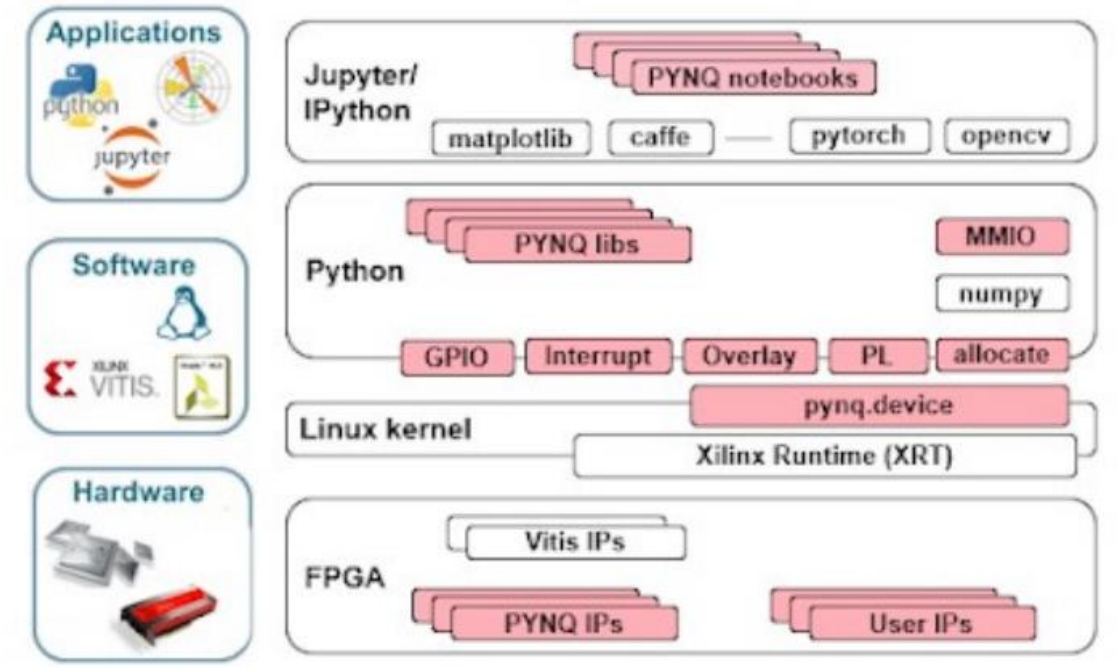
Esistono due principali versioni di Python: 2 e 3. La 2 non è più supportata. Sono molto simili ma comunque i codici potrebbero non essere compatibili.

Le stringhe sono sequenze di caratteri immutabili e accedibili in sola lettura.

Vi sono tre tipi di collezioni: tuple, liste e dizionari. Le tuple sono collezioni immutabili e omogenee (dati tutti dello stesso tipo). Le liste, dualmente, sono mutabili e possono contenere dati di tipo diverso. I dizionari sono mappe (coppie chiave-valori).

Python è object-oriented. Supporta classi ed oggetti. Prevede ereditarietà multipla. È possibile definire solo un costruttore chiamato __init__. Tutti i metodi (costruttore compreso) per essere tali devono contenere come primo parametro la keyword self (equivalente del this in java). Gli attributi privati hanno il nome preceduto da due underscore (__nomeAttributoPrivato).

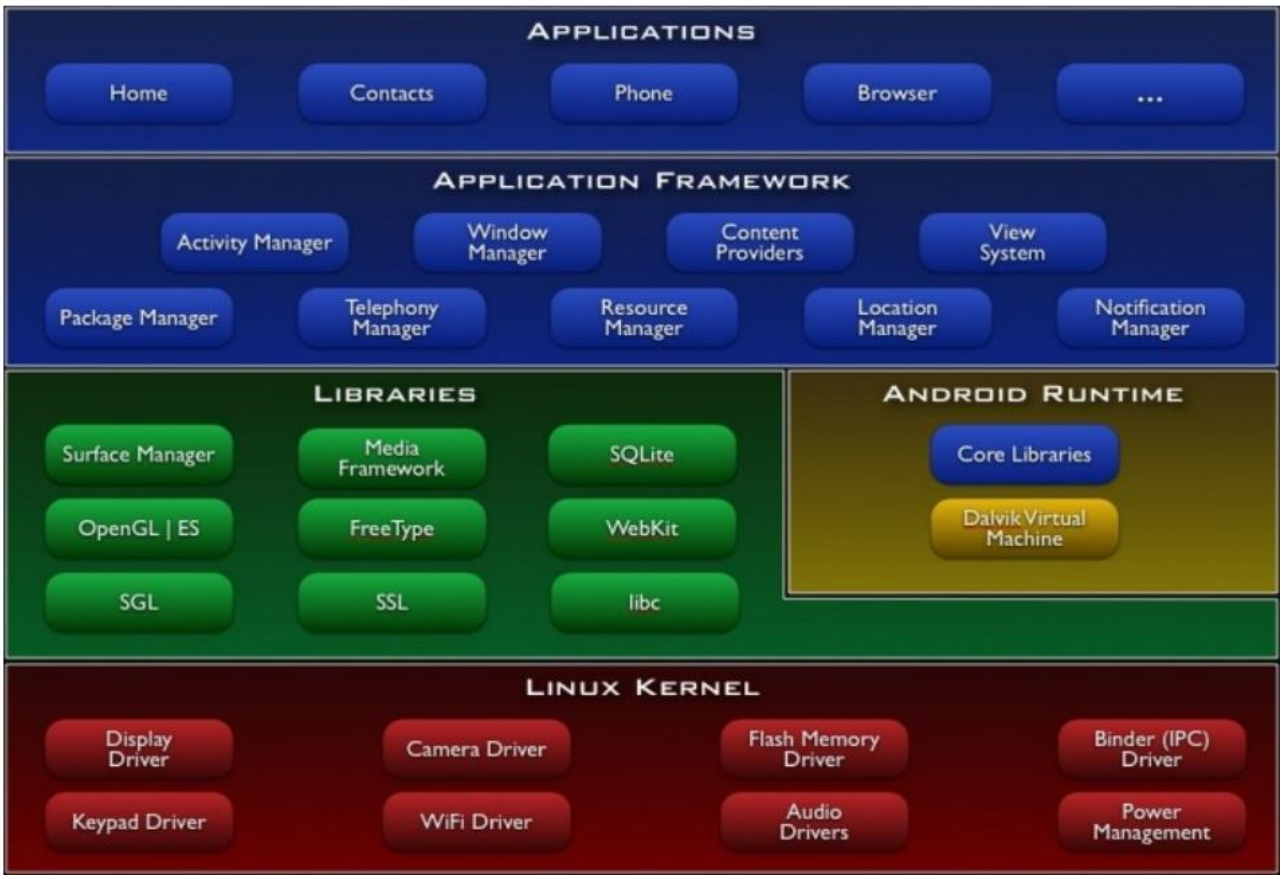
PYNQ



Pynq è un progetto opensource di Xilinx che vuole portare la programmazione FPGA ad un maggiore livello di astrazione. Pensata per Python.

Molto simile alla Zynq. Equipaggiata con un processore dual core ARM (PS). Dotata di una FPGA (PL) e di memorie, periferiche, interruttori eccetera. La memoria è condivisa tra PS e PL. Lo Xilinx Runtime (XRT) permette le comunicazione tra PS e PL. La PS durante l'esecuzione di codice può richiamare funzionalità della PL tramite le interfacce degli overlay. Gli overlay sono interfacce di livello intermedio che incapsulano un IPCORE permettendone l'accesso alle sue funzionalità. Tramite le interfacce overlay il codice Python può chiamare l'esecuzione hw sulla FPGA. Gli overlay tramite oggetti driver (che gestiscono l'interazione tra overlay e PL) permettono l'accesso alle funzionalità di un bitstream. Gli overlay vengono chiamati a livello python. Sono scritti da specialisti hw, ma per usarli non è richiesta alcuna conoscenza specifica da parte del programmatore. L'overlay può essere visto come una libreria sw. Gli overlay presentano una firma da invocare e costituiscono un punto di accesso di alto livello all'FPGA. Possono essere importati o caricati dinamicamente.

ANDROID



Android è un sistema operativo linux-based per sistemi embedded. Mette a disposizione uno stack sw per lo sviluppo di app. Diverse versioni, diversi set di API.

A livello kernel troviamo Linux e i driver per gestire l'hw. A livello librerie troviamo le librerie di più basso livello che implementano funzionalità basilari e non eseguono su una JVM. Troviamo anche l'Android Runtime che gestisce le VM Dalvik. Difatti, android usa una versione particolare della JVM, più leggera, chiamata Dalvik. Per motivi di sicurezza ogni app esegue nella propria privata istanza di Dalvik. Ne consegue che questa VM deve essere leggera, veloce da istanziare e devono poter coesistere più istanze in parallelo senza consumare troppe risorse. Di tutto ciò se ne occupa l'Android Runtime.

Il livello Application Framework mette a disposizione le API utili agli sviluppatori su cui si appoggiano le app del livello superiore. Mette a disposizione diverse astrazioni per gestire l'hw e le risorse.

Il livello Application contiene le app sviluppate dai programmatori.

CAMERA

Ogni telefono ha un hw diverso e generalizzare può essere complicato. Questa cosa era particolarmente vera nelle prime due versioni delle API android per gestire la camera, Camera e Camera2. Erano molto a basso livello e hw-dependent. Le cose sono migliorate con CameraX, che offre API più ad alto livello. Oggi molti telefoni hanno la camera posteriore costituita da più camere (in genere 3). Queste camere aggiuntive servono per elaborazioni parallele che permettono di migliorare la messa fuoco, la calibrazione della luce, grandangolo, migliorare la qualità dell'immagine e cose di questo tipo. Non è ancora possibile accedere singolarmente a una di queste camere, non si pensa possa servire allo sviluppatore. Manca supporto standard alle multicamere.

CameraX offre molti use case di utilità. Tra cui:

- Preview: apre la camera mostrando il flusso video (a basso costo) di ciò che la camera sta inquadrando
- Image Analysis: esegue un dato algoritmo su di una immagine
- Image capture: salva una immagine
- Video capture: salva un video

Chaquopy è un tool che permette l'integrazione di Python in progetti Android. Consente di chiamare codice Python da Java/Kotlin e viceversa. È un Python SDK per sistemi Android. Si occupa di molte cose di basso livello. È integrato con il build gradle di Android Studio e importarlo è piuttosto facile. Non permette l'accesso a ottimizzazioni hw (es la gpu del telefono). È un prodotto sotto licenza a pagamento. È molto lento e ingombrante.

DEEP LEARNING

Insieme di metodologie che realizzano reti neurali che tramite apprendimento possono generalizzare qualsiasi dato. Esiste dagli anni 90 ma esplosione solo recentemente (servono potenza di calcolo e grandi disponibilità di dati).

Una rete neurale è un sistema di calcolo che ricalca l'architettura del cervello umano. Si vuole modellare la strategia di elaborazione della biologia del cervello. In particolare, per poter imparare dagli esempi come fanno gli animali (che permette anche di gestire errori e situazioni nuove). Si è modellato il cervello umano grazie alla conoscenze della biologia e della neuroscienza. Il cervello umano è composto da tantissimi neuroni, unità di elaborazione elementare. I neuroni sono interconnessi tra loro da sinapsi che fanno viaggiare i "dati" tra un neurone e l'altro. Tramite le sinapsi ogni neurone riceve molteplici input da varie fonti (come altri neuroni o periferiche) e in output manda la propria elaborazione ad altri neuroni. I segnali che entrano in un neurone vengono sommati tramite una somma pesata, dove i pesi fanno parte del neurone (e vengono imparati con l'addestramento). Quindi se al neurone arrivano degli input Xi, per ognuno di questi ci sarà un peso Wi e si ottiene la somma di tutti i prodotti XiWi. A tale somma pesata si aggiunge poi un offset chiamato BIAS. Il Bias è un peso costante (anch'esso imparato) che si aggiunge alla fine della somma pesata. Il valore risultante va in ingresso a una funzione di attivazione, una funzione non lineare che ha il compito di smorzare le features deboli lasciando passare solo quelle forti. L'output della funzione è l'output del neurone: $o = F_{att}(\sum_i x_i w_i \ + b)$.

Una rete neurale è fatta di tanti livelli. Ogni livello è formato da tanti "neuroni". Ogni livello aggiunge uno strato di elaborazione, ricevendo l'input dal livello precedente e mandando l'output al livello successivo.

I pesi vengono determinati durante l'addestramento tramite la backpropagation, un procedimento in cui iterativamente tramite degli esempi noti si aggiornano i pesi per minimizzare l'errore della rete. Si utilizza il gradiente di una loss function, una funzione (che deve essere derivabile) che quantifica lo iato tra l'output predetto e l'output atteso. Ci sono vari modi per calcolare la loss function tra cui la somma dei quadrati delle differenze. Se si conoscono le label si parla di metodo supervised, altrimenti si parla di metodo unsupervised. Di norma in ogni iterazione si evita che l'intensità del cambiamento dei pesi sia troppo elevata per evitare l'overfit. Difatti un'intensità elevata significherebbe che si sta prendendo in considerazione solo l'ultimo esempio dimenticandosi dei precedenti, cosa da evitare.

Il forward pass è quando si percorrono in avanti i livelli della rete usando i pesi per sintetizzare un nuovo output. Il backward pass è quando si ripercorre al contrario la rete per aggiornare i pesi.

Quando si usa un dataset per inizializzare la rete, una porzione di questo dataset costituisce il training set che viene usato in addestramento. L'altra parte è il testing set, ossia i dati che vengono usati per i test dopo l'addestramento. Il forward pass avviene sia in training che in testing, il backward solo durante il training.

Le CNN sono reti a cui si aggiunge una rete di front-end che ha il compito di scremare i dati. Difatti, molto spesso i dati possono essere eccessivi e contenere informazioni superflue che vanno tolte. Si aggiungono quindi vari livelli preliminari di filtraggio. Anche questi filtri sono reti neurali con i loro pesi da addestrare. Nel caso delle immagini si parla di filtri convoluzionali che riducono la dimensione della foto. Nelle CNN vi sono 3 fasi di scrematura:

- Convoluzione: si applica un kernel di convoluzione (somma pesata), fase onerosa
- Attivazione: si applica una funzione di attivazione
- Pooling: si fondono i dati, si fa un'aggregazione di informazioni

Nel machine learning ci sono 3 casi:

- Classificazione → assegno a un dato una label presa da un set predefinito (le label sono note)
- Regressione → predizione di una funzione (ti do x e la rete mi da f(x))

- Clustering → come classificazione ma label non sono note (le deve intuire la rete stessa)

TP= true positive, FP=fake positive, TN=true negative, FN=fake negative

Precisione: quante predizioni positive sono corrette $P = \frac{TP}{TP+FN}$

Recall: quante volte ho detto positivo e ho fatto bene a farlo $R = \frac{TP}{TP+FP}$

F1score: riassunto di P e R (varie implementazioni tra cui $F_1 = \frac{2PR}{P+R} = \frac{2TP}{2TP+FN+FP}$)

Scikitlearn → libreria python opensource numpy-based per il machine learning

TENSORFLOW

Framework open-source per il deep learning by Google

Questa libreria utilizza come struttura dati per i pesi della rete i TENSORI, cioè array NDIM molto simili agli array di numpy.

Due principali versioni: 1 e 2.

Nella 1 le fasi di definizione delle operazioni e di esecuzione sono separate. La prima avviene tramite la creazione di un grafo computazione, un oggetto che rappresenta le operazioni da fare.difatti, possiamo vedere la rete come un grafo unidimensionale che si può navigare con i tensori. Il grafo ha NODI (cioè le operazioni) e ARCHI (cioè i tensori che fluiscono tra nodi).

Nella versione 2 queste due fasi collassano in una sola tramite la EAGER EXECUTION (come anche in pytorch).

I tensori possono essere variabili, costanti o placeholder (solo v1). I placeholder hanno il ruolo di segnaposto. Nella versione 1 infatti è necessario riempire i valori vuoti del grafo con valori segnaposto di input che poi verranno sostituiti nella esecuzione. A questo servono i tensori placeholder che ovviamente non esistono nella versione 2. L'esecuzione nella v1 avviene tramite la SESSIONE, che non esiste nella 2.

Tensorflow rappresenta un backend. Tensorflow mette a disposizione anche un frontend: Keras. Tramite le API ad alto livello di Keras si possono gestire con maggiore astrazione le funzionalità di Tensorflow.

PYTORCH

Stessa roba però di Meta. Anche qui tensori e grafo (dinamico). Una sola fase come in TF2. Non c'è un'equivalente di Keras anche se ci sono classi specifiche come Module che permette di definire una rete. Tramite Sequential si può definire lo stack dei layer della rete. Tramite il metodo forward di Module si possono definire le operazione da fare durante l'inferenza (come la call in keras).

MOBILE DEEP LEARNING

Tensorflow e Pytorch permettono di convertire i loro modelli in formato lite per l'esecuzione su sistemi embedded (TFLite e PyTorch Mobile). Il train avviene ancora su pc desktop, mentre inferenza e test possono avvenire su smartphone.

Quando si converte un modello l'obiettivo è rendere le operazioni più efficiente e ridurre il costo computazionale della rete. Vi sono varie strategie tra cui la QUANTIZZAZIONE, che consiste nell'approssimare la rete riducendo il num di bit dei pesi (e vi sono vari modi per farlo). Questo permette di risparmiare memoria e rendere le operazioni più veloci. Vi sono due tipi di quantizzazione:

- POST-TRAINING → dopo l'addestramento
- AWARE-TRAINING → durante l'addestramento tengo conto dell'approssimazione

Tensorflow mette a disposizione 3 strategie di Post-Training. PyTorch ha in comune le prime due strategie di post-training con tf e addizionalmente offre una terza strategia di quantizzazione aware-training.

Riducendo il numero di bit si va a ridurre la granularità dei pesi. Si riduce l'insieme dei possibili valori. Per fare questo bene di solito si calcola prima il range di interesse. Difatti, se operiamo con valori float32 verosimilmente non staremo usando tutto lo spettro di possibili valori, ma i nostri valori effettivi saranno tutti all'interno di un certo range. È bene tenerne conto durante la quantizzazione in modo tale che con i bit ridotti si copra il range (evitando di tagliare fuori porzioni di range e/o avere porzioni inutili che peggiorerebbero la quantizzazione).

Tensorflow mette a disposizione queste tre strategie:

- **Dynamic range**
  Conversione statica solo per i pesi, i quali passano da float32 a int (floating to fixed). All'inferenza i pesi tornano in float32 e poi vengono cachati per motivi di efficienza. Per gli operatori dinamici questi vengono quantizzati con un range dinamico ed eseguiti come operazioni a 8bit.
- **Full integer (quantizzazione statica)**
  Solo conversione statica laddove possibile. Tutto convertito in int8. Per i tensori statici no problem: basta calibrare il range staticamente. Per quelli dinamici niente (a meno che non sia possibile stimare il range staticamente se si ha un set rappresentativo con cui farlo)
- **Float16**
  Tutto in float16 (no fixed solo floating). Minor perdita di precisione e minor guadagno di velocità.

Pytorch ha le prime due in comune. Poi aggiunge una strategia di quantizzazione aware-training in cui si tiene conto dell'addestramento delle approssimazioni. Durante l'addestramento i pesi vengono anche convertiti (da appositi layer di QUANT e DEQUANT) e si addestrano anche i pesi convertiti per aver una maggior precisione. Aumenta l'accuratezza dell'approssimazione ma allunga i tempi di addestramento (è come se dovessi fare due addestramenti).