



UNIMORE
UNIVERSITÀ DEGLI STUDI DI
MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche,
Informatiche e Matematiche

Xilinx toolchain

Seminario Informatica Industriale

Corso di Laurea in Ing. Informatica
(D.M.270/04) [16-262]
Anno accademico 2020/2021

Dott. Gianluca Brilli
gianluca.brilli@unimore.it
prof. Paolo Burgio
Paolo.burgio@unimore.it

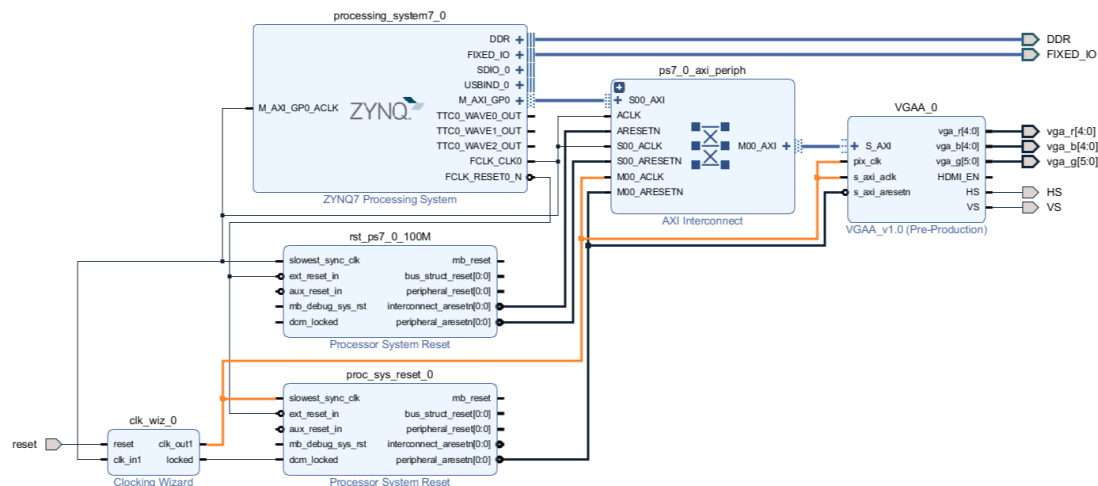
È vietata la copia e la riproduzione dei contenuti e immagini in qualsiasi forma.

È inoltre vietata la redistribuzione e la pubblicazione dei contenuti e immagini non autorizzata espressamente dall'autore o dall'Università di Modena e Reggio Emilia.

Toolchain

Strumenti di Sviluppo

→ La progettazione FPGA consiste nella creazione di uno schema a blocchi, come in figura:



→ I blocchi possono funzionare in maniera standalone, o essere controllati da una CPU.

Strumenti di Sviluppo

- Vitis: *programmazione software.*
- Vivado: *design hardware.*



Vitis HLS

Overview

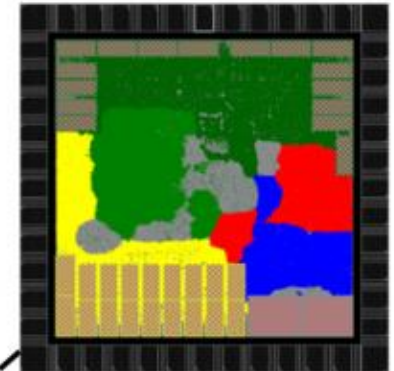
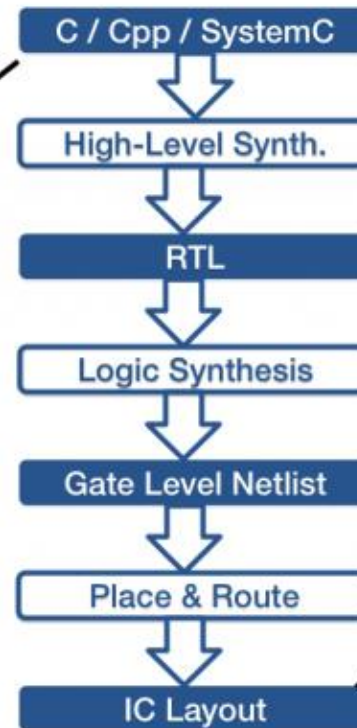
→ *Strumento che vi permette di sintetizzare un IP a partire da un linguaggio di alto livello come il C/C++.*

```
void polarDecode (int *llrs, int* decodedBits)
{
    // Initialize polar decoder with LLRs
    for (int i = 0; i < numBits; i++) {
        polarDecoderArray[log(n)-1][i].llr = llrs[i];
    }

    // run successive cancellation
    for (int i = 0; i < channelCode->n; i++) {
        calcElement* currentElement;
        currentElement= &polarDecoderArray[0][i];

        if (currentElement->isG == 0) {
            calcF(currentElement);
        } else {
            calcG(currentElement);
        }

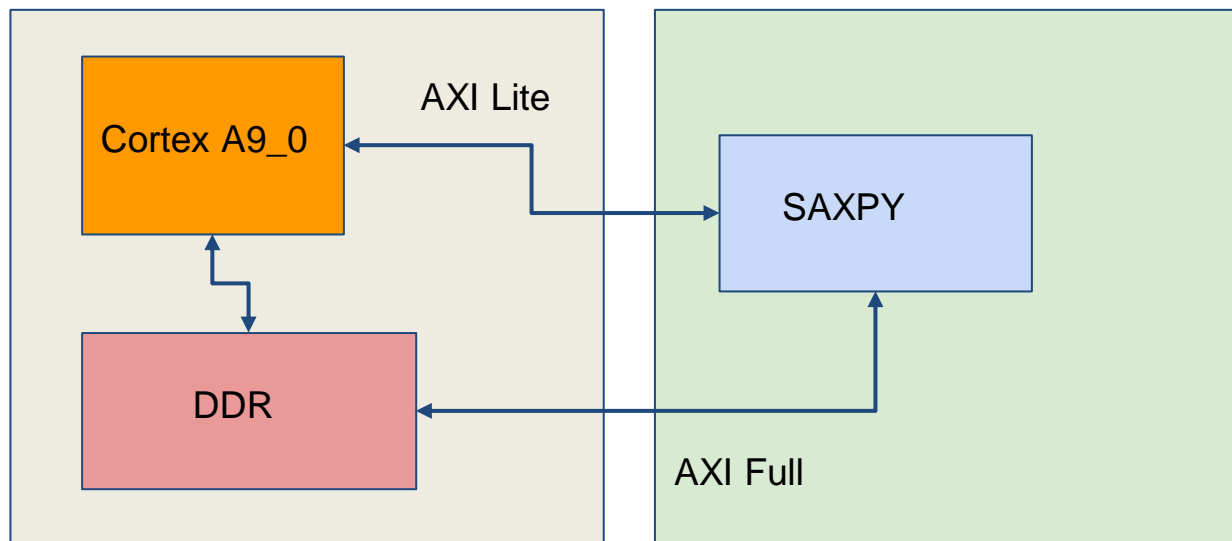
        if (currentElement->L >= 0) {
            decoded[i] = 0;
        } else {
            decoded[i] = 1;
        }
    }
}
```



Accelerazione FPGA

Esempio

- Paradigma host-acceleratore: un applicativo in esecuzione sul processore richiede ad un engine FPGA di accelerare una determinata funzione. (es. convoluzione o moltiplicazione di matrici).
- Supponiamo di voler realizzare il design seguente:



Esempio

→ Partiamo quindi da un'implementazione software di una somma di vettori:

```
6 typedef int data_t;  
7  
8 void add_pipe(data_t *x, data_t* y, data_t alpha, int n) {  
9  
10     loop_ddd: for(int i = 0; i < n; ++i) {  
11         y[i] = alpha*x[i] + y[i];  
12     }  
13 }
```

→ Convertito in un modulo hardware:



→ Per prima cosa andiamo a creare i bus di comunicazione tra l'engine e il core.

Vitis HLS

Interfacce AXI4

- 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.

Vitis HLS

Passaggi da seguire

```
6 typedef int data_t;
7
8 void add_pipe(data_t *x, data_t* y, data_t alpha, int n) {
9
10     #pragma HLS INTERFACE m_axi port=x offset=slave bundle=m_ddr
11     #pragma HLS INTERFACE m_axi port=y offset=slave bundle=m_ddr
12
13     #pragma HLS INTERFACE s_axilite port=n      bundle=s_data
14     #pragma HLS INTERFACE s_axilite port=alpha  bundle=s_data
15     #pragma HLS INTERFACE s_axilite port=return bundle=ctrl
16
17     loop_ddr: for(int i = 0; i < n; ++i) {
18         #pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000
19         y[i] = alpha*x[i] + y[i];
20     }
21 }
```

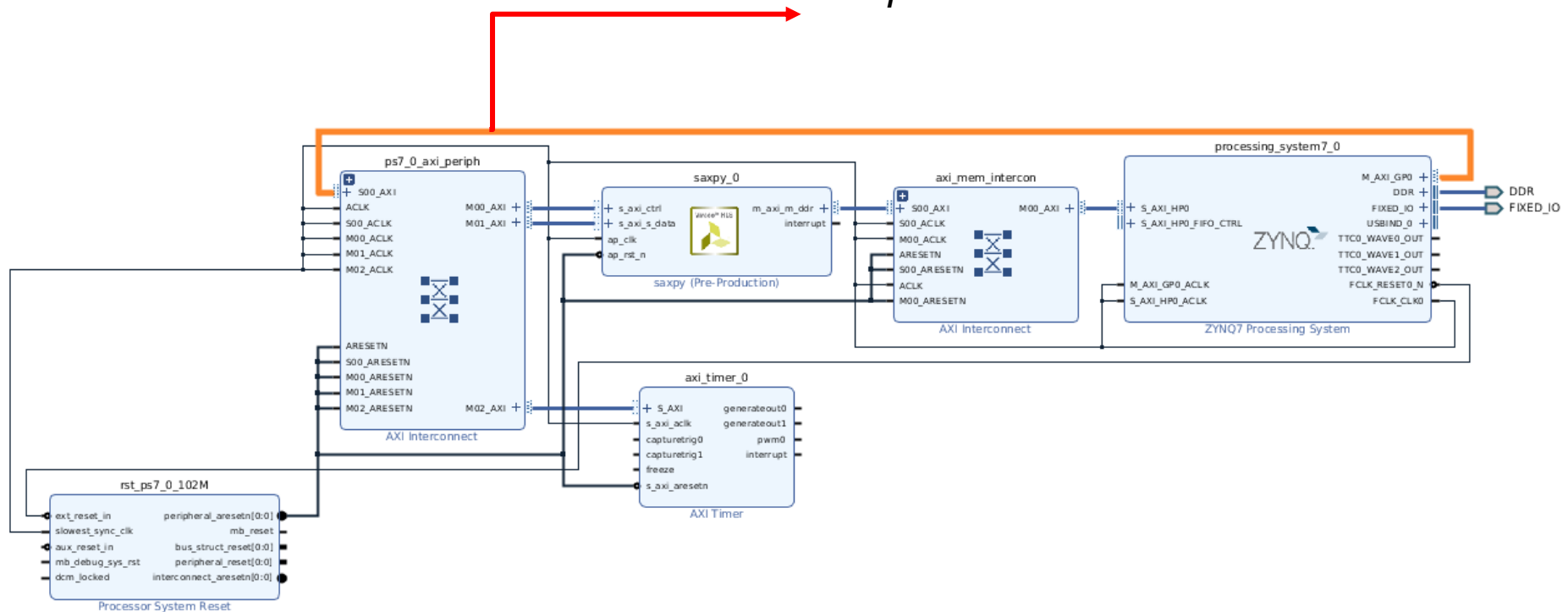
→ Interfacce Slave AXI Lite, le utilizziamo per trasmettere parametri di configurazione alla nostra top function.

→ Interfacce AXI Master, le utilizziamo per accedere alla DDR dall'IP.

Vivado

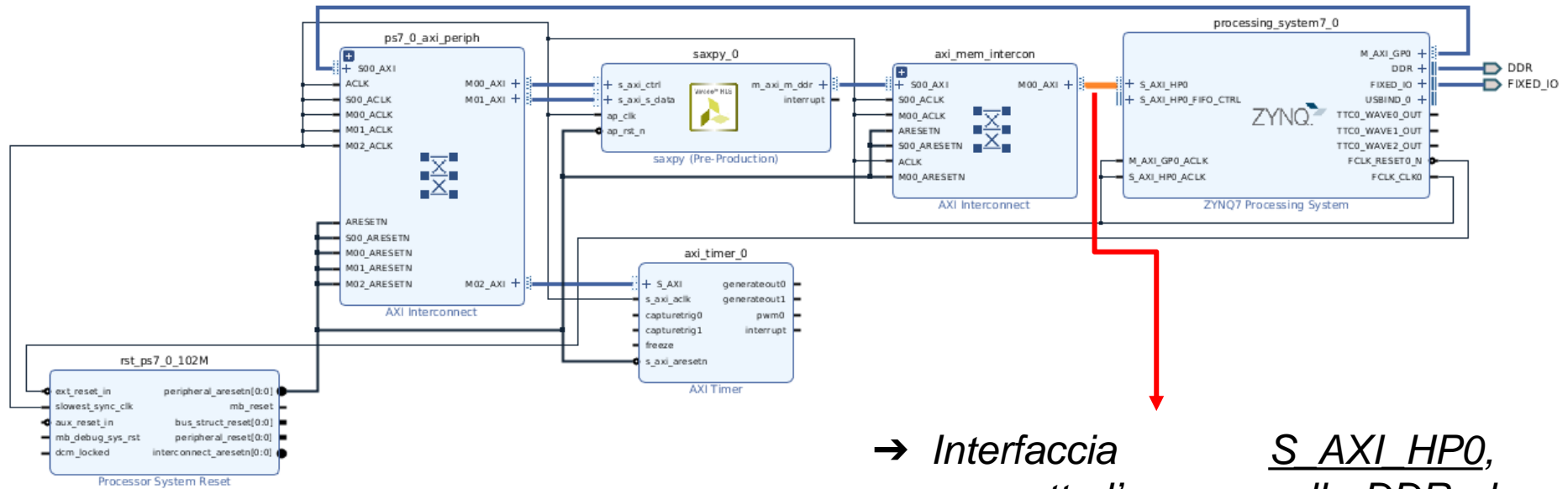
Design Complessivo

→ Interfaccia M AXI GP0,
permette il trasferimento dei
parametri tramite AXI Lite.



Vivado

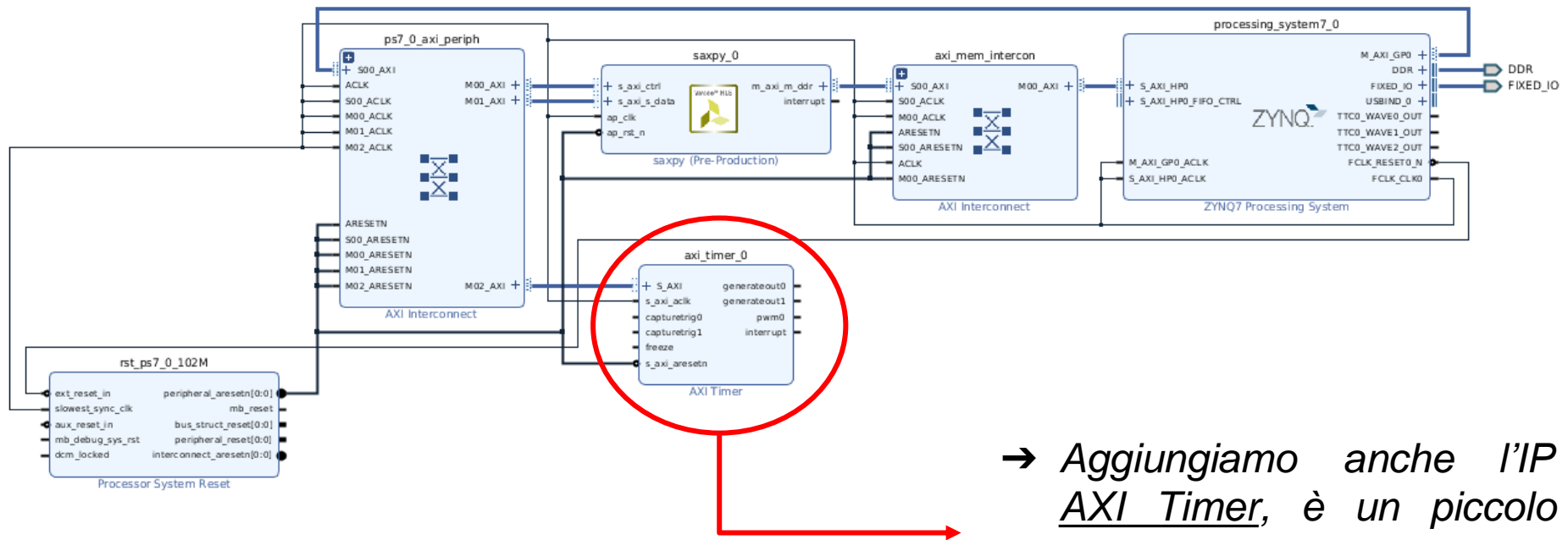
Design Complessivo



→ *Interfaccia S AXI HP0,
permette l'accesso alla DDR al
modulo HLS.*

Vivado

Design Complessivo

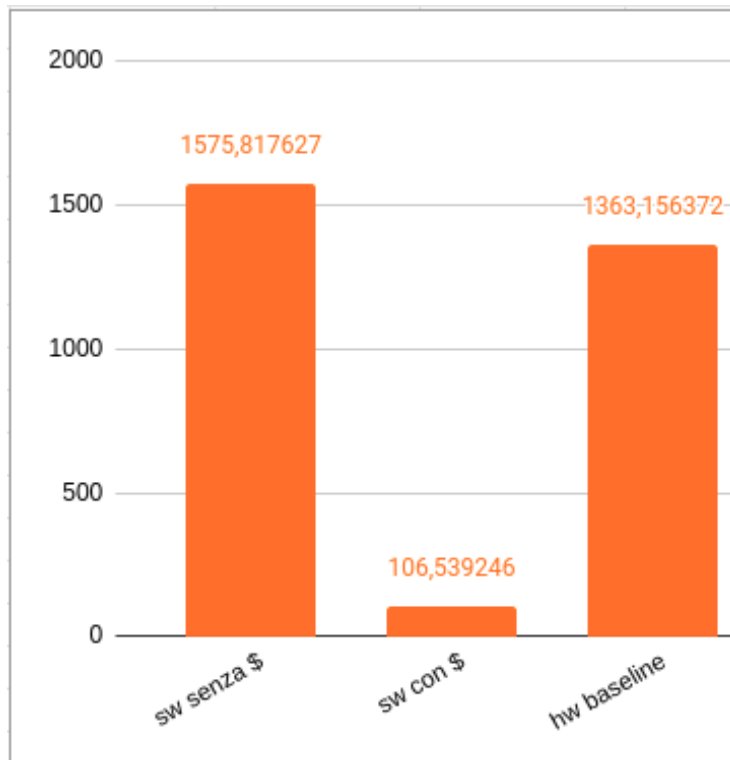


→ Aggiungiamo anche l'IP AXI Timer, è un piccolo timer in PL che ci permetterà di misurare i tempi di esecuzione.

Vivado

Comparazioni HW - SW

→ Proviamo a svolgere qualche comparazione tra l'IP generato e la rispettiva controparte software.



- I tempi sono riportati in millisecondi con un numero di elementi pari a 2097152 (2^{21}).
- In questo caso notiamo che realizzare la SAXPY in HW comporta uno slowdown di un fattore di circa 12x.
- I tempi ottenuti in sw sono stati misurati anche disabilitando le cache.

Ottimizzazioni

#01 Loop Pipelining

- Introduciamo la prima ottimizzazione, che consiste nel creare una pipeline sulle iterazioni del loop.
- Supponiamo il seguente frammento di pseudocodice:




- La latenza del ciclo for è pari al numero di cicli di clock moltiplicato per il numero di iterazioni.
- L'esecuzione seriale di un algoritmo, che gira su hardware a bassa frequenza di clock (100MHz) e senza l'utilizzo di memorie cache, comporta pessime prestazioni.

Ottimizzazioni

#01 Loop Pipelining

- I problemi che impediscono il corretto utilizzo di una pipeline sono di due tipologie:
 - dipendenze inter-iterations: si ha quando una iterazione dipende dal risultato della precedente.
 - contese di interfacce: quando si cerca di accedere a risorse hardware con un numero limitato di interfacce.
- Nel nostro caso abbiamo contesa sull'accesso alla porta y che viene acceduta sia in lettura che in scrittura:



```
y[i] = alpha*x[i] + y[i];
```

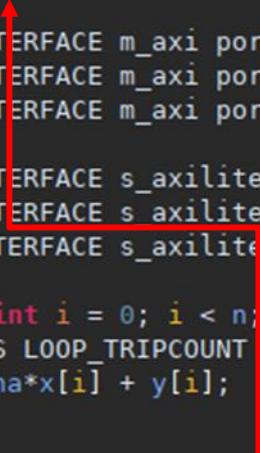
- Questo comporta una latenza totale di **15002 cicli di clock** supponendo di avere un loop da **1000 iterazioni**.

Ottimizzazioni

#01 Loop Pipelining

→ Per risolvere questo problema inseriamo una terza porta (z) che vada a memorizzare il risultato della SAXPY

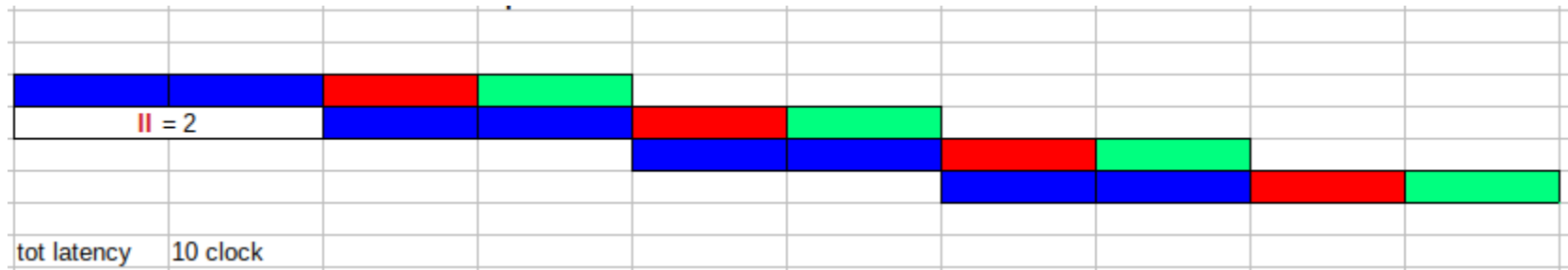
```
void add_pipe(data_t *z, data_t *x, data_t* y, data_t alpha, int n) {  
  
    #pragma HLS INTERFACE m_axi port=x offset=slave bundle=ddr  
    #pragma HLS INTERFACE m_axi port=y offset=slave bundle=ddr  
    #pragma HLS INTERFACE m_axi port=z offset=slave bundle=ddr  
  
    #pragma HLS INTERFACE s_axilite port=n      bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=alpha bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=return bundle=ctrl  
  
    loop_ddr: for(int i = 0; i < n; ++i) {  
        #pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000  
        z[i] = alpha*x[i] + y[i];  
    }  
}
```



Ottimizzazioni

#01 Loop Pipelining

→ Questa ottimizzazione comporta un notevole incremento di performance, andando a realizzare una pipeline con $II = 2$.



→ Con una conseguente latenza totale di **2009 cicli di clock** supponendo di avere un loop da **1000 iterazioni**.

Ottimizzazioni

#01 Loop Pipelining

- Notiamo infine che è presente ancora un punto di contesa, in questo caso in lettura tra $x[i]$ e $y[i]$:

```
void add_pipe(data_t *z, data_t *x, data_t* y, data_t alpha, int n) {  
  
    #pragma HLS INTERFACE m_axi port=x offset=slave bundle=ddr  
    #pragma HLS INTERFACE m_axi port=y offset=slave bundle=ddr  
    #pragma HLS INTERFACE m_axi port=z offset=slave bundle=ddr  
  
    #pragma HLS INTERFACE s_axilite port=n      bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=alpha bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=return bundle=ctrl  
  
    loop_ddr: for(int i = 0; i < n; ++i) {  
        #pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000  
        z[i] = alpha*x[i] + y[i];  
    }  
}
```

- Questo dovuto al fatto che abbiamo accorpato tutti i bus in un unico bundle.

Ottimizzazioni

#01 Loop Pipelining

→ Notiamo infine che è presente ancora un punto di contesa, in questo caso in lettura tra $x[i]$ e $y[i]$:

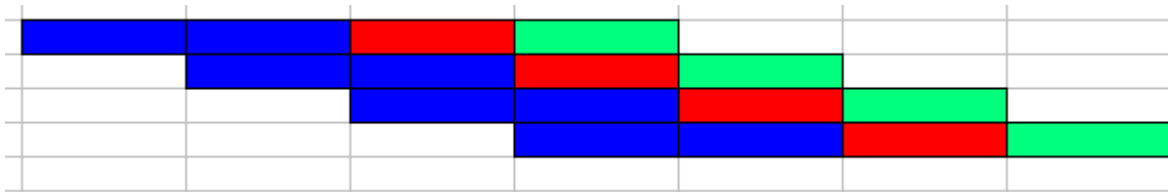
```
void add_pipe(data_t *z, data_t *x, data_t* y, data_t alpha, int n) {  
  
    #pragma HLS INTERFACE m_axi port=x offset=slave bundle=x_ddr  
    #pragma HLS INTERFACE m_axi port=y offset=slave bundle=y_ddr  
    #pragma HLS INTERFACE m_axi port=z offset=slave bundle=z_ddr  
  
    #pragma HLS INTERFACE s_axilite port=n      bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=alpha bundle=s_data  
    #pragma HLS INTERFACE s_axilite port=return bundle=ctrl  
  
    loop_ddr: for(int i = 0; i < n; ++i) {  
        #pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000  
        z[i] = alpha*x[i] + y[i];  
    }  
}
```

→ Questo dovuto al fatto che abbiamo accorpato tutti i bus in un unico bundle.

Ottimizzazioni

#01 Loop Pipelining

→ *In questo modo arriviamo infine alla seguente pipeline:*



→ *Con una conseguente latenza finale di **1004 cicli di clock** supponendo di avere un loop da **1000 iterazioni**.*

Ottimizzazioni

#01 Loop Pipelining

→ *Resoconto finale delle tre varianti della pipeline, secondo i report di Vitis HLS:*

Loop							
		Latency (cycles)		Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
-loop_ddr	15002	15002	18	15	1	1000	yes

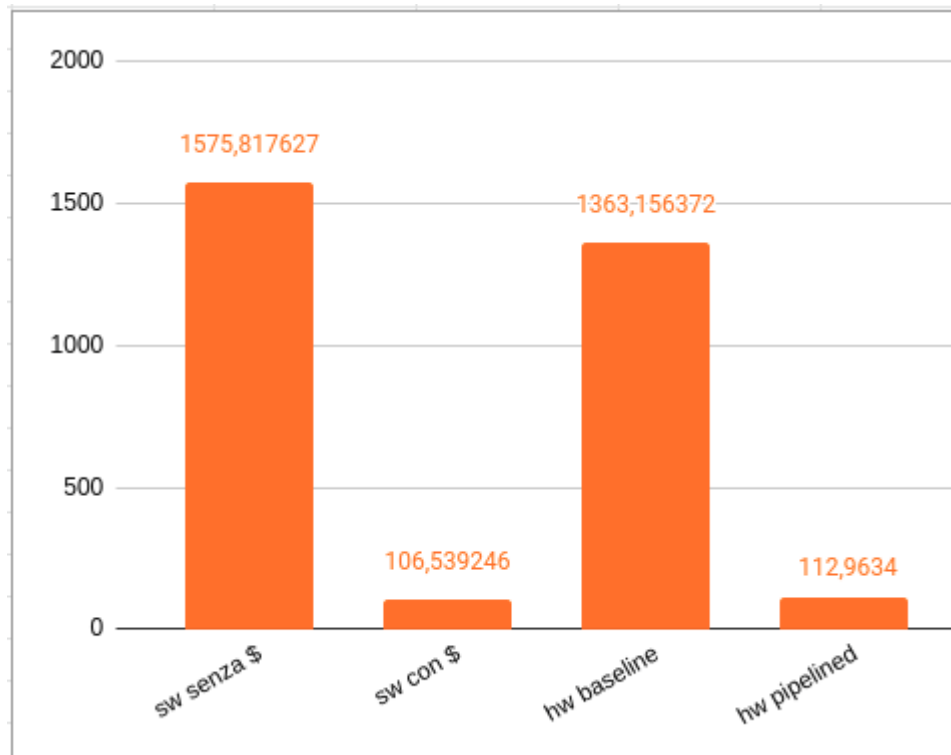
Loop							
		Latency (cycles)		Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
-loop_ddr	2010	2010	13	2	1	1000	yes

Loop							
		Latency (cycles)		Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
-loop_ddr	1004	1004	6	1	1	1000	yes

Ottimizzazioni

#01 Loop Pipelining

→ Testando infine questa ottimizzazione sulla Zedboard otteniamo le seguenti prestazioni:



→ Vediamo il notevole distacco prestazionale ottenuto rispetto alla versione sw senza cache.

→ Necessario ottimizzare gli accessi in memoria.

Ottimizzazioni

#02 BRAM e Accessi Burst

- *L'idea è quella di svolgere il calcolo della SAXPY portando una parte dei dati su memorie molto più veloci della DDR e più vicine al modulo HW che stiamo realizzando.*
- *Sfruttiamo le BRAM, sparse all'interno della PL. Per fare questo ci basta definire degli array all'interno dell'IP, che verranno automaticamente sintetizzati sfruttando delle BRAM. Questi possiamo vederli come delle piccole "scratchpad", "una sorta di memoria cache" che vengono gestite manualmente dal programmatore / progettista HW.*
- *Nello specifico variabili singole vengono sintetizzate tramite dei registri o dei singoli FlipFlop, mentre gli array tramite delle BRAM.*
- *Cerchiamo infine di ottimizzare anche gli spostamenti di memoria da DDR alla scratchpad, andando a trasferire chunk più grandi, sfruttando gli accessi burst di AXI4.*

Ottimizzazioni

#02 BRAM e Accessi Burst

- Per prima cosa definiamo dei blocchi di BRAM per gli array x, y e z ed un registro per alpha:

```
data_t z_loc[MAX_SIZE_LOC];  
data_t x_loc[MAX_SIZE_LOC];  
data_t y_loc[MAX_SIZE_LOC];  
  
data_t alpha_loc;
```

- Il numero di elementi deve essere noto a tempo di sintesi.

```
#define MAX_SIZE      4194304  
#define DIVIDER      128  
#define MAX_SIZE_LOC MAX_SIZE/DIVIDER
```

- Fissiamo quindi un numero massimo di elementi per i nostri blocchi di BRAM.

Ottimizzazioni

#02 BRAM e Accessi Burst

- *Andiamo poi a realizzare un ciclo for esterno per gestire il caricamento in burst dalla DDR alla BRAM:*

```
for(int j = 0; j < DIVIDER; ++j ) {  
  
    memcpy(x_loc, &x[j]*MAX_SIZE_LOC, MAX_SIZE_LOC*sizeof(data_t));  
    memcpy(y_loc, &y[j]*MAX_SIZE_LOC, MAX_SIZE_LOC*sizeof(data_t));  
  
    // implementare il calcolo della SAXPY locale  
  
    memcpy(&z[j]*MAX_SIZE_LOC, z_loc, MAX_SIZE_LOC*sizeof(data_t));  
}
```

- *La memcpy in HLS può essere utilizzata per il caricamento da DDR a memoria locale all'IP e tramite questa il tool di sintesi realizza automaticamente degli accessi in burst.*
- *Dal momento che la Zynq7000 ha un ridotto quantitativo di BRAM, dobbiamo stare attenti a non sforare con il quantitativo di memoria utilizzata.*

Ottimizzazioni

#02 BRAM e Accessi Burst

- Implementiamo infine il loop più interno che va a lavorare sui dati locali. Successivamente al termine del loop più interno viene svolta una scrittura burst da BRAM a DDR per salvare i risultati parziali.

```
for(int j = 0; j < DIVIDER; ++j ) {  
  
    memcpy(x_loc, &x[j*MAX_SIZE_LOC], MAX_SIZE_LOC*sizeof(data_t));  
    memcpy(y_loc, &y[j*MAX_SIZE_LOC], MAX_SIZE_LOC*sizeof(data_t));  
  
    for(int i = 0; i < MAX_SIZE_LOC; ++i) {  
#pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000  
        z_loc[i] = alpha_loc*x_loc[i] + y_loc[i];  
    }  
    memcpy(&z[j*MAX_SIZE_LOC], z_loc, MAX_SIZE_LOC*sizeof(data_t));  
}
```

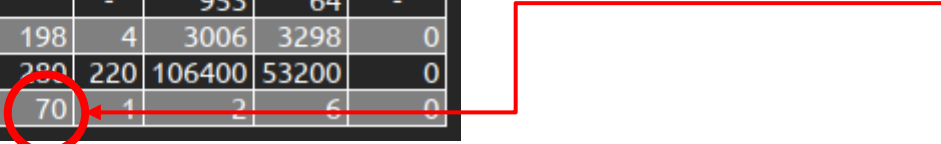
Ottimizzazioni

#02 BRAM e Accessi Burst

→ *Utilizzo di risorse finale:*

Summary

Name	BRAM_18K	DSP	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	113	-
FIFO	-	-	-	-	-
Instance	6	4	2053	2741	-
Memory	192	-	0	0	-
Multiplexer	-	-	-	380	-
Register	-	-	953	64	-
Total	198	4	3006	3298	0
Available	280	220	106400	53200	0
Utilization (%)	70	1	2	6	0



Ottimizzazioni

#02 BRAM e Accessi Burst

```
#include <stdint.h>
#include <stdlib.h>
#include <string.h>

#define MAX_SIZE 4194304
#define DIVIDER 128
#define MAX_SIZE_LOC MAX_SIZE/DIVIDER

typedef int data_t;

void add_mem_opt(data_t *z, data_t *x, data_t *y, data_t alpha, int n) {

#pragma HLS INTERFACE m_axi port=z offset=slave bundle=z_ddr
#pragma HLS INTERFACE m_axi port=x offset=slave bundle=x_ddr
#pragma HLS INTERFACE m_axi port=y offset=slave bundle=y_ddr

#pragma HLS INTERFACE s_axilite port=n bundle=s_data
#pragma HLS INTERFACE s_axilite port=alpha bundle=s_data
#pragma HLS INTERFACE s_axilite port=return bundle=ctrl

    data_t z_loc[MAX_SIZE_LOC];
    data_t x_loc[MAX_SIZE_LOC];
    data_t y_loc[MAX_SIZE_LOC];

    data_t alpha_loc;

    alpha_loc = alpha;

    for(int j = 0; j < DIVIDER; ++j) {

        memcpy(x_loc, &x[j*MAX_SIZE_LOC], MAX_SIZE_LOC*sizeof(data_t));
        memcpy(y_loc, &y[j*MAX_SIZE_LOC], MAX_SIZE_LOC*sizeof(data_t));

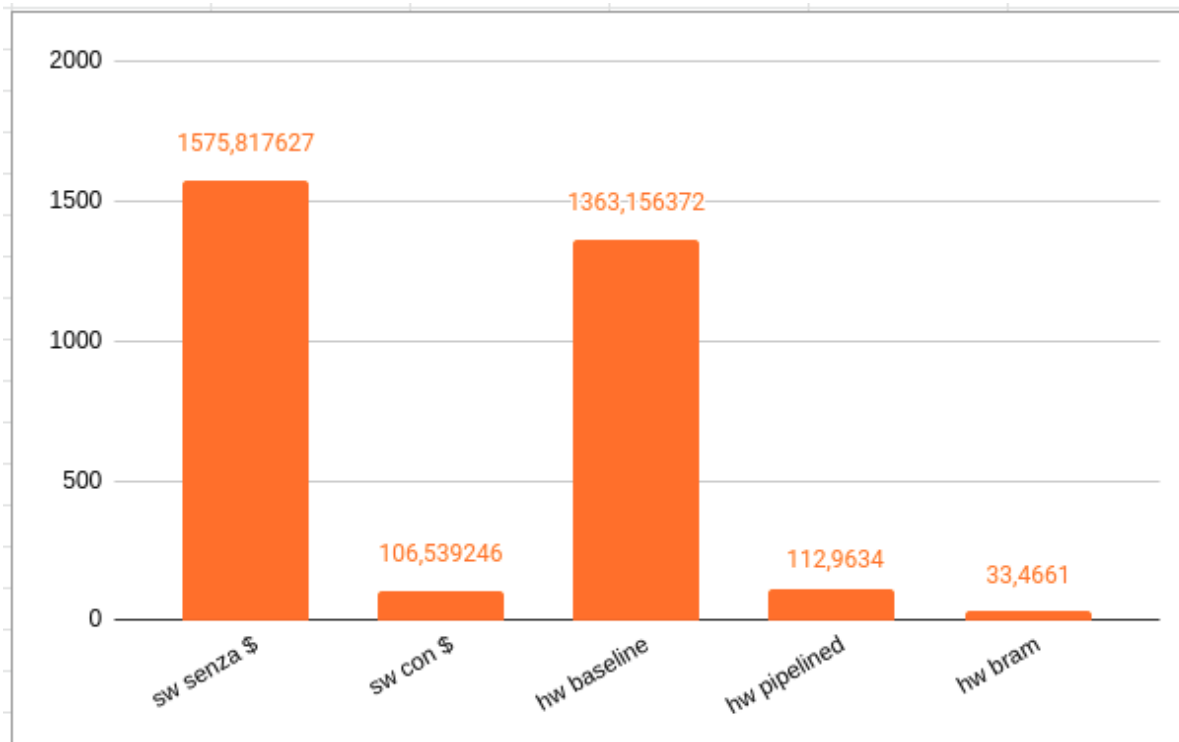
        for(int i = 0; i < MAX_SIZE_LOC; ++i) {
#pragma HLS LOOP_TRIPCOUNT MAX=1000 MIN=1000
            z_loc[i] = alpha_loc*x_loc[i] + y_loc[i];
        }
        memcpy(&z[j*MAX_SIZE_LOC], z_loc, MAX_SIZE_LOC*sizeof(data_t));
    }
}
```

→ Codice finale pronto
per essere
sintetizzato.

Ottimizzazioni

#02 BRAM e Accessi Burst

→ Aggiungendo anche questa ottimizzazione arriviamo al seguente risultato prestazionale:



Altre ottimizzazioni

Precisione ridotta e aritmetica Fixed Point

- Possibile ridurre il numero di bit utilizzare per rappresentare i nostri dati. In questo caso abbiamo utilizzato degli interi a 32 bit. E' possibile utilizzare una precisione ridotta andando a sfruttare l'header file **ap_int.h**, andando a definire dei dati di tipo (ap_int o ap_uint), ad esempio:
 - `ap_int <5> val` → questo definisce rispettivamente un intero signed/unsigned a 5 bit.
 - `ap_uint<5> val`
- Per quanto riguarda l'utilizzo di numeri reali, è possibile passare ad una rappresentazione in virgola fissa, utilizzando l'header file **ap_fixed.h**, ad esempio:
 - `ap_fixed <9, 5, AP_RND_CONV, AP_SAT> val`
 - questo definisce un numero reale avente in totale 9 bit, di cui 5 per la parte intera e 4 per la parte decimale.

Altre ottimizzazioni

Loop Unrolling

- *In questo caso l'hardware che svolge l'operazione viene replicato per ogni iterazione del loop.*
- *Aumenta incredibilmente il numero di risorse utilizzate ed i tempi di sintesi.*

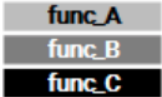
```
loop_1: for(int i = 0; i < N; i++) {  
    #pragma HLS unroll  
    a[i] = b[i] + c[i];  
}
```

Altre ottimizzazioni

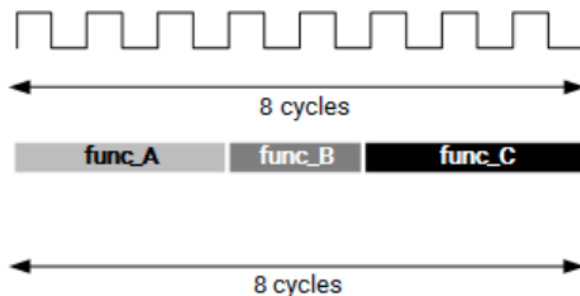
Dataflow

- Permette di eseguire in parallelo dei task all'interno della top function, andando a realizzare una pipeline a livello di funzione.
- Supponendo questo frammento di codice:

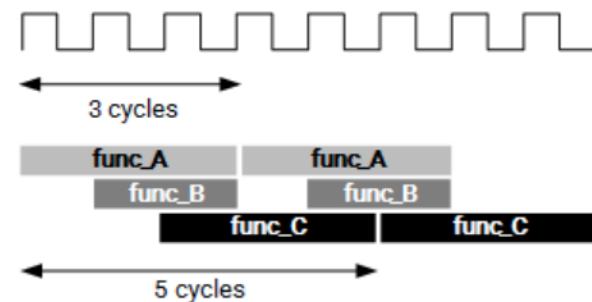
```
void top (a,b,c,d) {  
    ...  
    func_A(a,b,i1);  
    func_B(c,i1,i2);  
    func_C(i2,d)  
  
    return d;  
}
```



→ Senza Dataflow



→ Con Dataflow



Altre ottimizzazioni

Approfondimenti

→ *Alcuni link utili che approfondiscono le ottimizzazioni ottenibili in HLS:*

- https://www.xilinx.com/html_docs/xilinx2017_4/sdaccel_doc/okr1504034364623.html
- http://home.mit.bme.hu/~szanto/education/vimima15/heterogen_xilinx_hls.pdf
- https://www.xilinx.com/support/documentation/sw_manuals/xilinx2019_2/ug902-vivado-high-level-synthesis.pdf#nameddest=xApplyingOptimizationDirectives

Programmazione SW

Piattaforme software

Diverse alternative

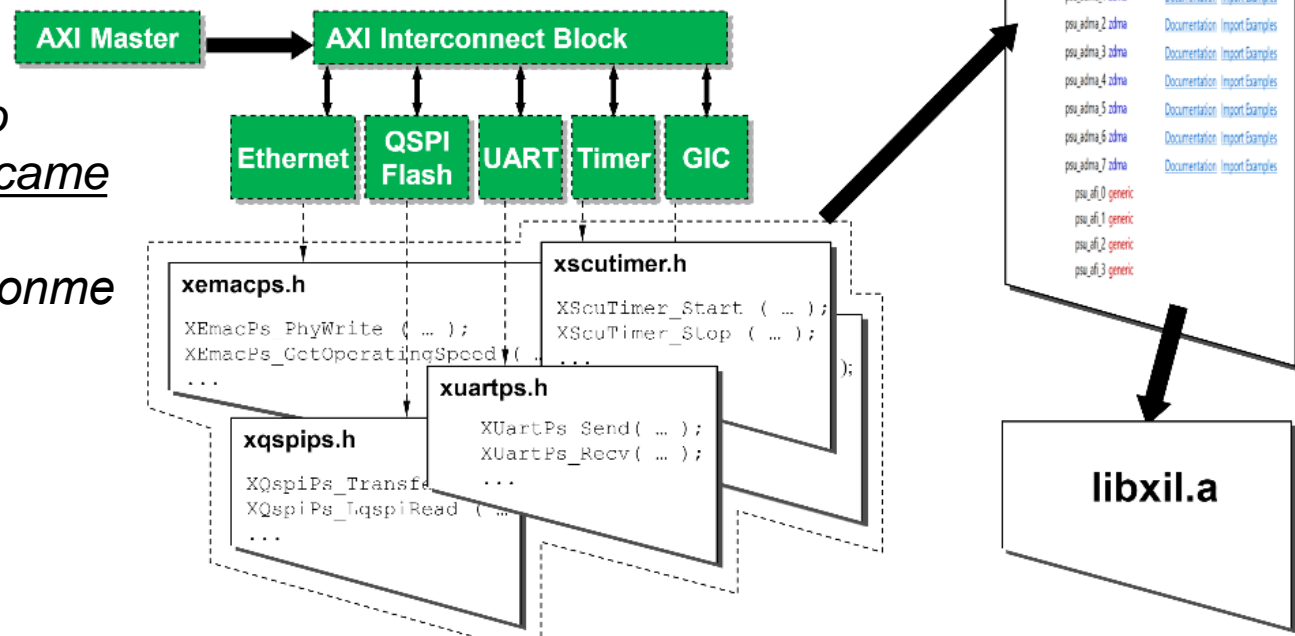
- Standalone: l'applicazione gira direttamente sopra al core (o quasi);
 - Librerie di basso livello pre-fornite da Xilinx;
 - I drivers per l'accesso al modulo HLS, generati automaticamente da Vitis.
- SO Real-Time: esempio **FreeRTOS**, come baremetal, ma implementato il supporto al multitasking. Implementati scheduler real-time.
- Distro Linux: necessario implementare un modulo kernel per accedere al modulo HLS.

Piattaforme software

Standalone

→ *Presente uno strato software chiamato Board Support Package (BSP), che si interpone tra l'hardware e l'applicazione.*

→ *Generato automaticamente dall'environment.*



Piattaforme software

Standalone

- *Il main inizializza i parametri e chiama la «accel_saxpy»;*
- *La «accel_saxpy» effettua chiamate ai drivers;*
- *La coerenza delle cache, tra CPU e FPGA deve essere gestita dal programmatore.*

```
1 void accel_saxpy(int *x, int* y, int alpha, int n) {
2
3     XAdd adder;
4     XAdd_Initialize(&adder, XPAR_ADD_PIPE_0_DEVICE_ID);
5
6     XAdd_Set_x(&adder, (u32)x);
7     XAdd_Set_y(&adder, (u32)y);
8     XAdd_Set_alpha(&adder, alpha);
9     XAdd_Set_n(&adder, n);
10
11     XAdd_Start(&adder);
12
13     while(!XAdd_IsDone(&adder));
14
15     Xil_DCacheInvalidate();
16 }
```

```
18 int main() {
19
20     u32 n = 16;
21
22     int* x = (int*)0x10000000;
23     int* y = x + n*4;
24
25     for(u32 i = 0; i < n; ++i) {
26         x[i] = i;
27         y[i] = 6;
28     }
29
30     int alpha = 2;
31
32     Xil_DCacheFlush();
33
34     accel_saxpy(x, y, alpha, n);
35
36     return 0;
37 }
```

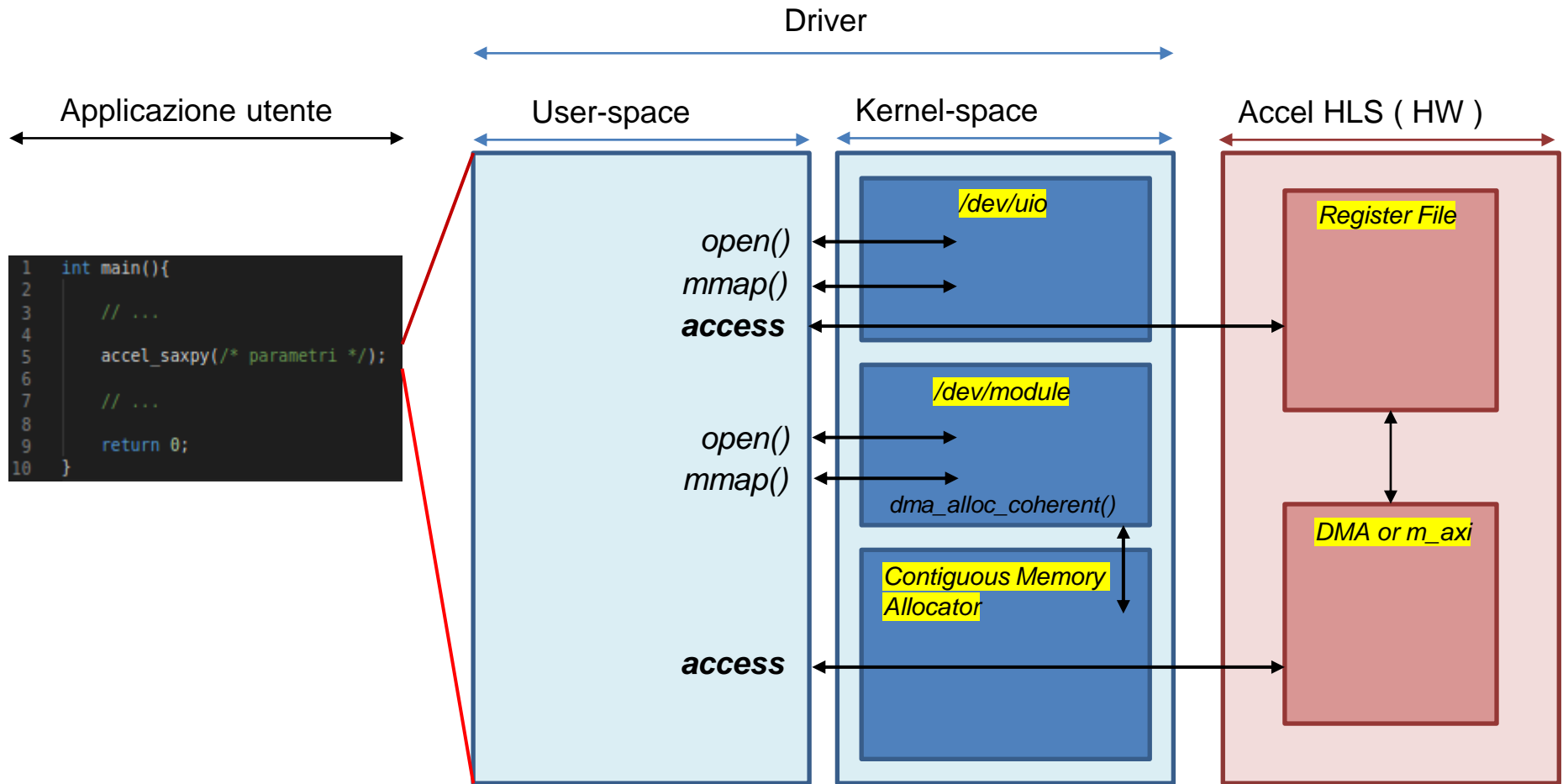
Piattaforme software

Linux

- PetaLinux: toolchain per la creazione di distribuzioni Linux, pensate per sistemi Xilinx.
 - Basato sul progetto Yocto, un insieme di tool che permettono il build di sistemi basati su Linux.
 - Semplifica notevolmente il processo di compilazione del kernel;
 - Permette la creazione di un root filesystem personalizzato, in base alle necessità dell'utilizzatore.

Piattaforme software

Linux



Grazie per l'attenzione