UNIMORE
UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Dipartimento di Scienze Fisiche, Informatiche e Matematiche

6. HLS Lab #1

High Performance Computing [262-022]

Dott. Gianluca Brilli
gianluca.brilli@unimore.it
Dott. Alessandro Capotondi
Alessandro.capotondi@unimore.it
Prof. Andrea Marongiu
andrea.marongiu@unimore.it

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

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.

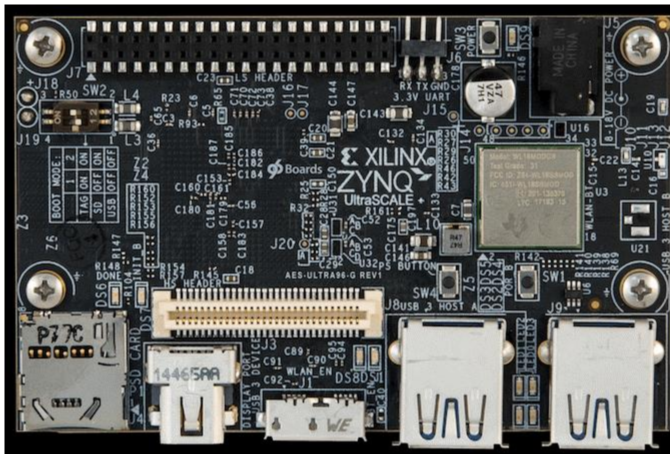
Agenda
Cosa vedremo in queste lezioni

- ***Esempi di accelerazione tramite FPGA:***
  - ***Simulazione di alcuni kernel computazionali, applicando alcune ottimizzazioni HLS.***

- *Test su development board:*
  - *Realizzazione di un design completo e test su piattaforma di sviluppo.*

Piattaforma di sviluppo

➔ *Avnet Ultra96v2*



AVNET®
Reach Further™

➔ *Zynq UltraScale+, SoC ZU3EG:*

➔ *CPU:*

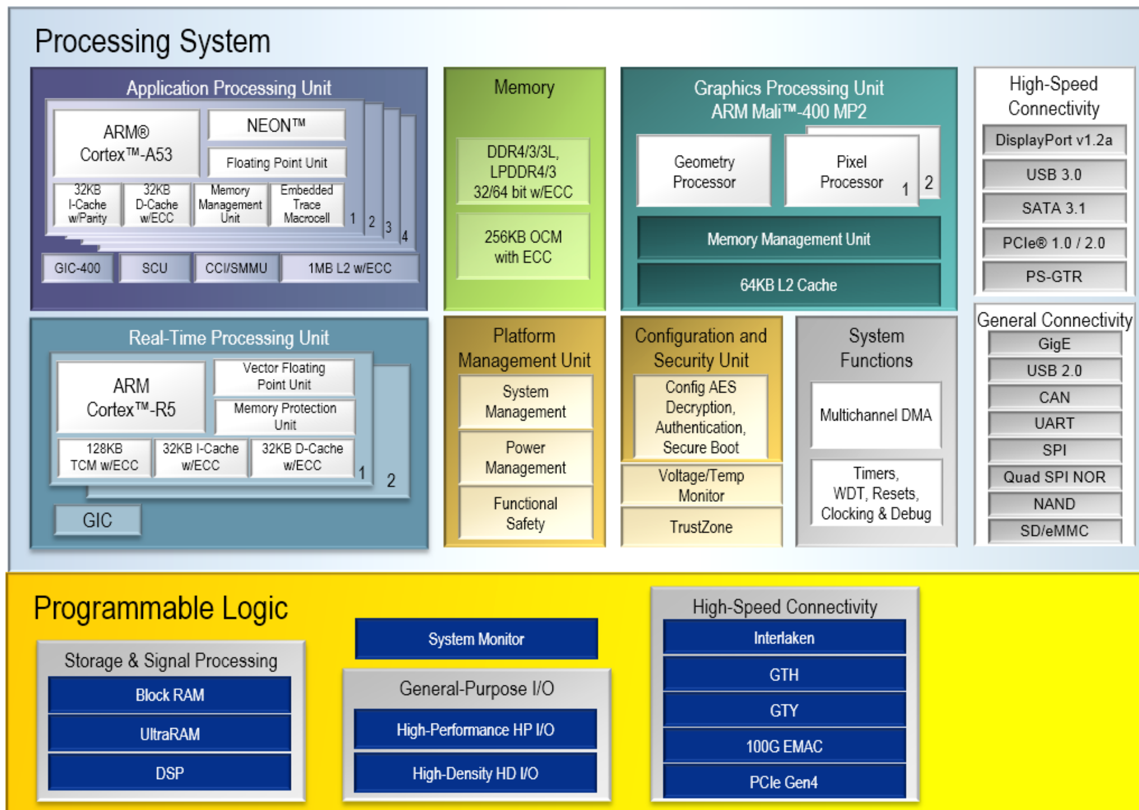
- ➔ *quad-cores ARM Cortex A53*
- ➔ *dual-cores ARM Cortex R5F*

➔ *GPU:*

- ➔ *ARM Mali 400 (solo grafica)*

➔ *16nm FPGA*

Piattaforma di sviluppo



- ➔ *PL: Programmable Logic*
- ➔ *PS: Processing System*

- ➔ *Queste due entità possono comunicare tra di loro per mezzo di un protocollo definito da ARM: AMBA AXI (**A**dvanced **M**icrocontroller **B**us **A**rchitecture **A**dvanced **EX**tensible **I**nterface);*

- ➔ *Disponibile in tre differenti varianti: AXI Lite, AXI Full e AXI Stream.*

Interfacce

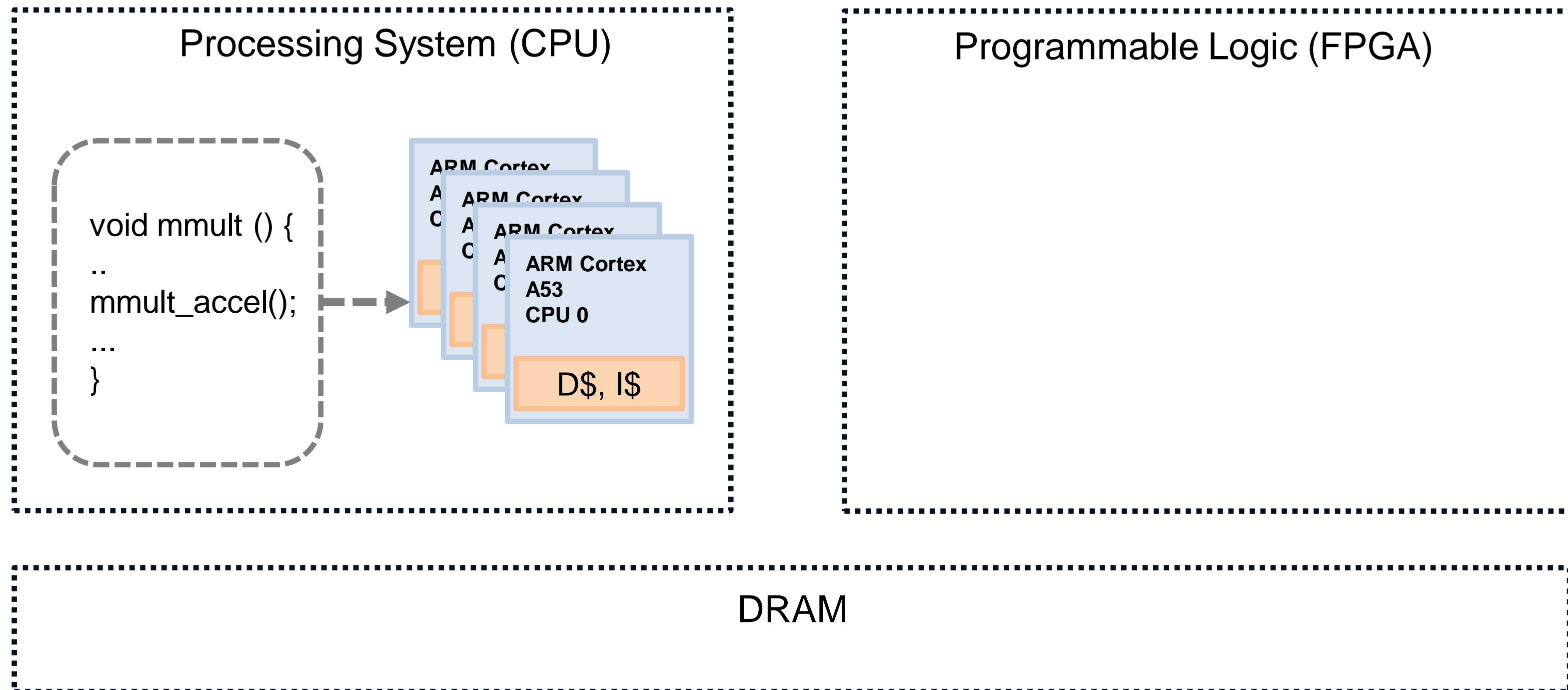
Interfacce AXI4

- *<u>AXI4</u>: 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.*

- *<u>AXI4-Lite</u>: 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.*

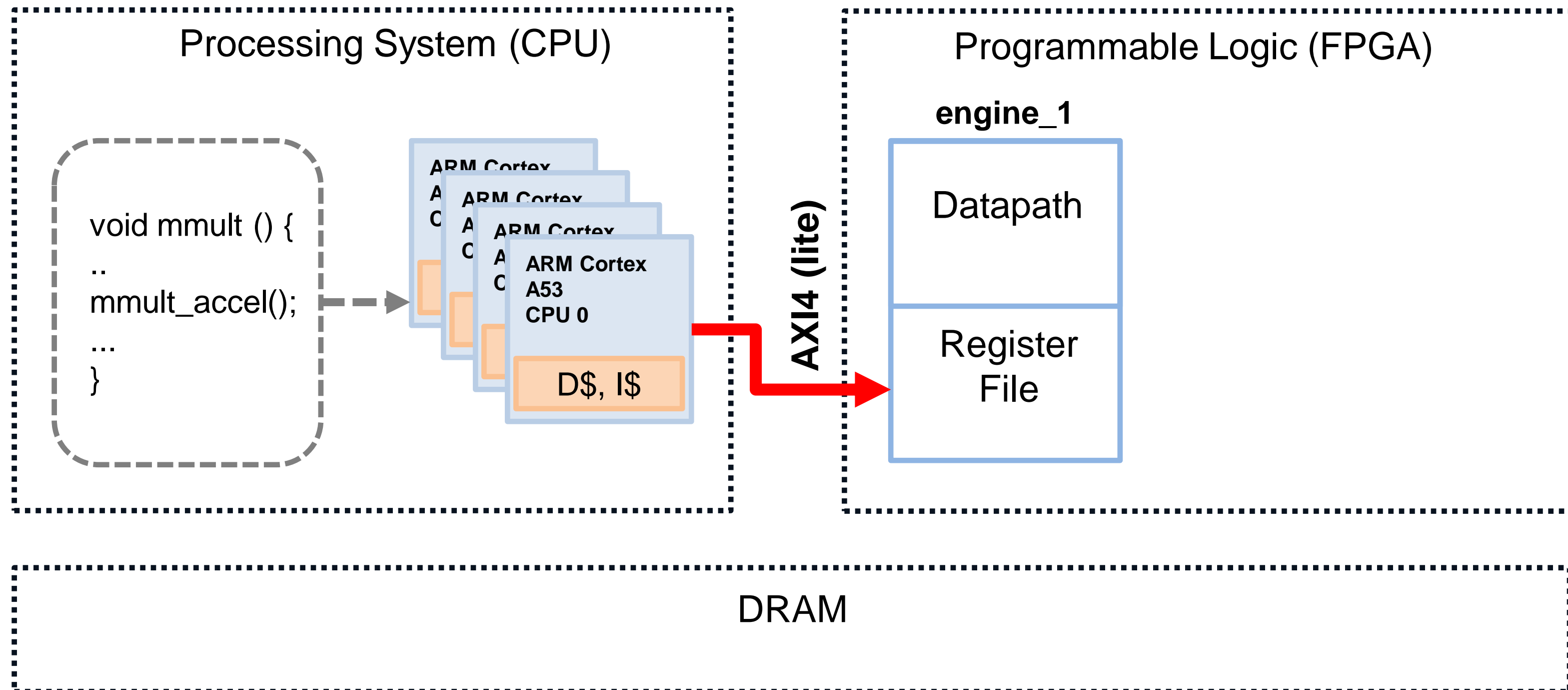
- *<u>AXI4-Stream</u>: Per streaming ad alte prestazioni. Non prevede indirizzamento (i.e. non è di tipo memory mapped). Consente trasferimenti dati di dimensione illimitata. Utilizzato tipicamente, ma non solo, per trasferire flussi di immagini.*

Interfacce AXI4 esempio (1)



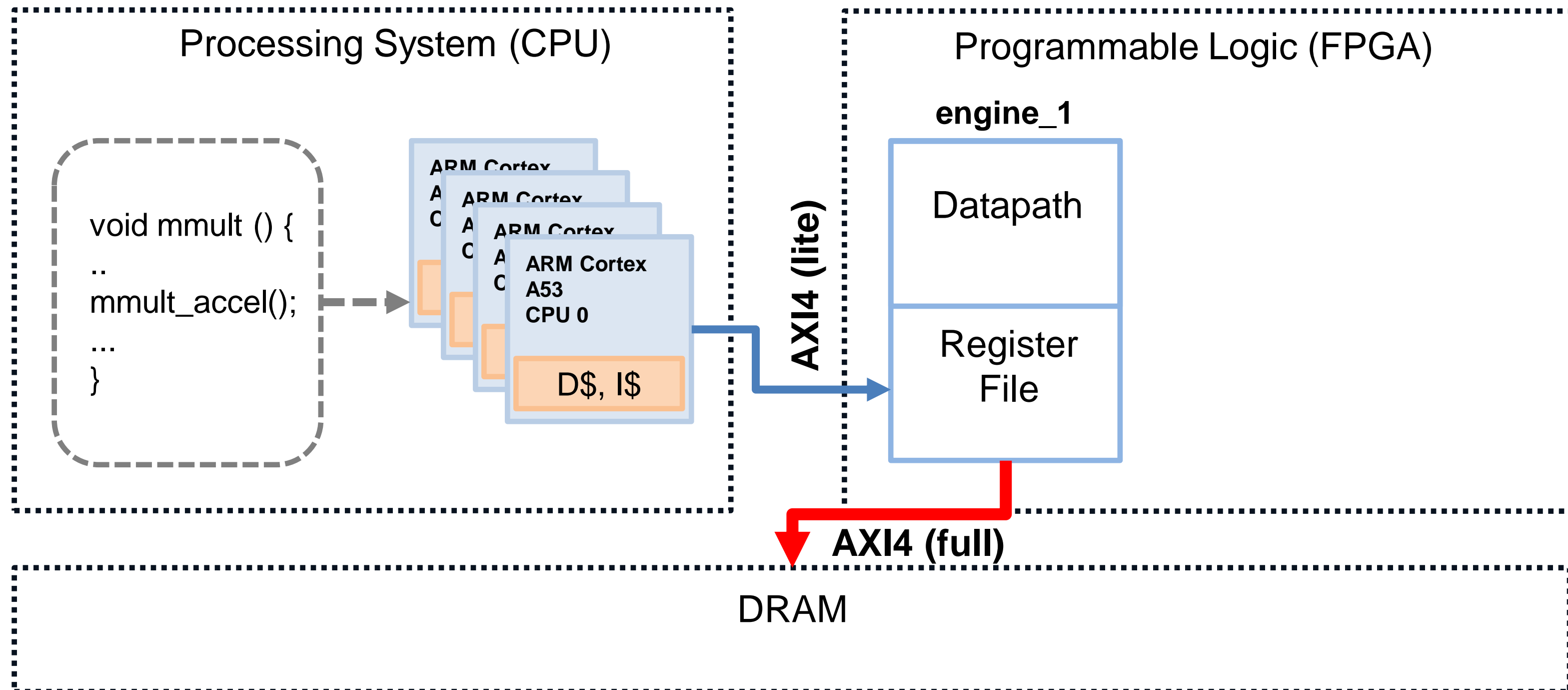
➔ *Contesto: un software in esecuzione sui cores, richiede l'accelerazione FPGA di una funzione (es: mmult).*

Interfacce AXI4 esempio



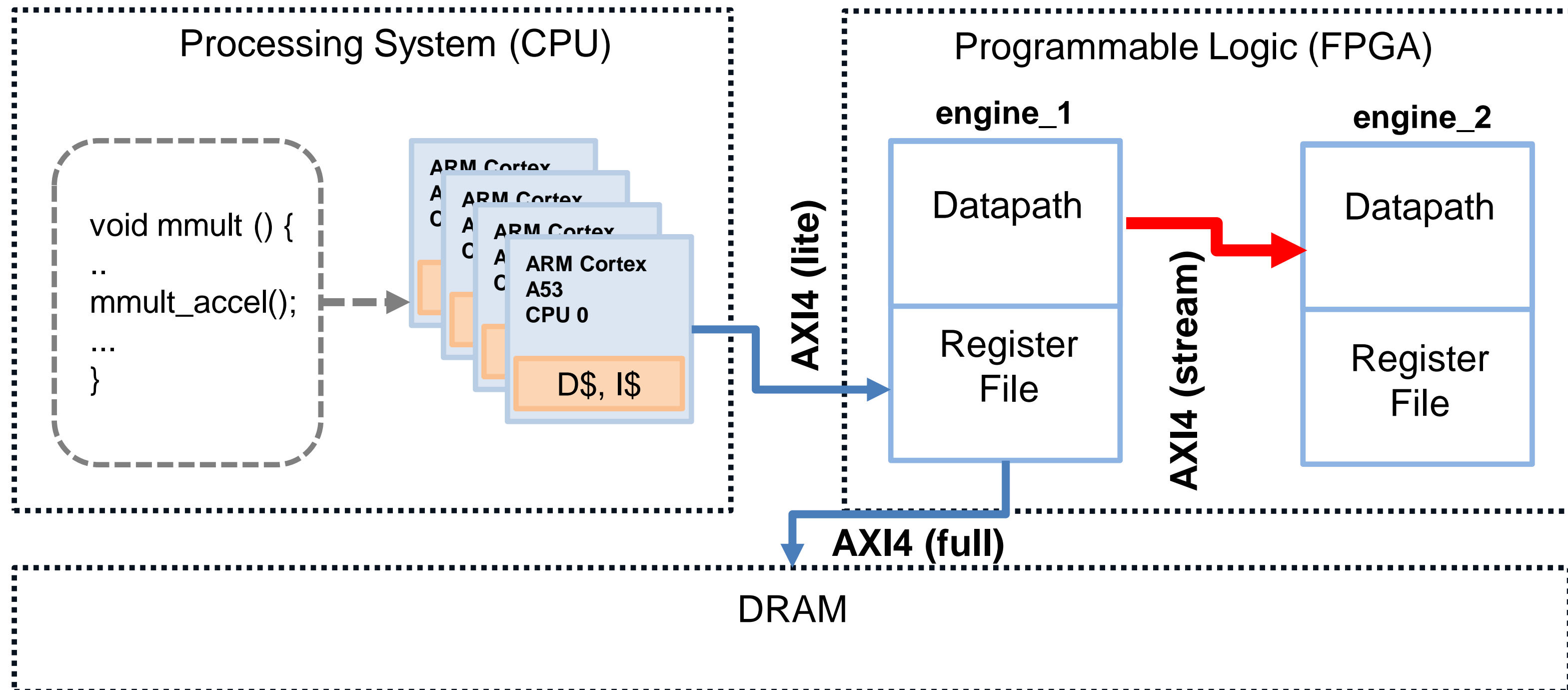
➔ *<u>AXI4 (lite)</u>: tipicamente usato per lettura/scrittura di parametri di configurazione, ad esempio per l'<u>accesso ai registri</u> di un acceleratore.*

Interfacce AXI4 esempio



➔ *<u>AXI4 (full)</u>: link memory-mapped ad elevate prestazioni (accessi burst), ad esempio utilizzato per interconnettere un modulo FPGA alla <u>memoria principale</u>.*

Interfacce AXI4 esempio



➔ *<u>AXI4 (stream)</u>: link <u>punto-punto</u>, molto snello (non necessario effettuare address request) e ad elevate prestazioni. Tipicamente utilizzato per mettere in comunicazione due moduli FPGA.*

Interfacce AXI4 in HLS (1)

```
void  fun(int *a, …){
        #pragma HLS INTERFACE <mode> port=<name> offset=<string> bundle=<string>
        …
}
```

- *m_axi*
- *s_axilite*
- *axis*

- *Riferito al rispettivo parametro della top-function (es: port=a)*

- *In caso di canale m_axi, crea un registro sull'acceleratore contenente l'indirizzo del buffer di memoria. (offset=slave)*

- *Permette il multiplexing di più segnali su bus AXI condivisi.*

Interfacce AXI4 in HLS (2)

Alcuni esempi

```
void  fun(int *a, int length){
            #pragma HLS INTERFACE s_axilite port=length bundle=params
            …
}
```

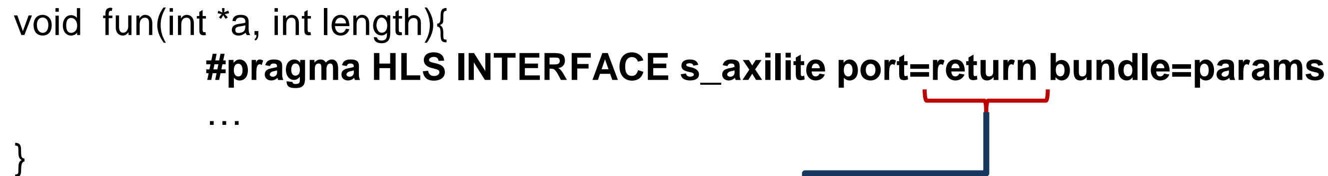
➔ *In questo caso il tool di HLS crea un registro a 32 bit contenente la lunghezza del vettore «a». Tale registro può essere letto/scritto tramite un canale AXI4-Lite.*

```
void  fun(int *a, int length){
            #pragma HLS INTERFACE m_axi port=a offset=slave bundle=mem
            …
}
```

➔ *Viene creata un'interfaccia AXI4 per l'accesso alla zona di memoria contenente il vettore «a». Tramite il parametro «offset=slave», viene creato un registro a 32 bit, per la memorizzazione dell'indirizzo del vettore «a». Tale registro è accessibile tramite un canale AXI4-Lite.*

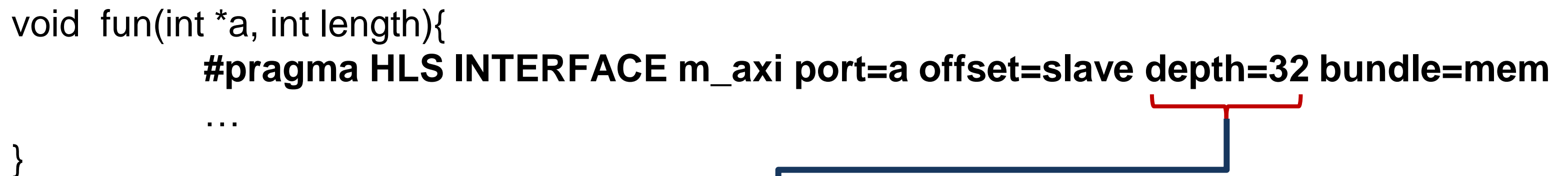
Interfacce AXI4 in HLS (3)

Alcuni casi particolari



```
void  fun(int *a, int length){
        #pragma HLS INTERFACE s_axilite port=return bundle=params
        …
}
```

➔ *In questo caso la keyword return permette la creazione di un canale AXI4-Lite per i segnali di controllo dell'acceleratore generato (es: start, stop, ready, done…).*

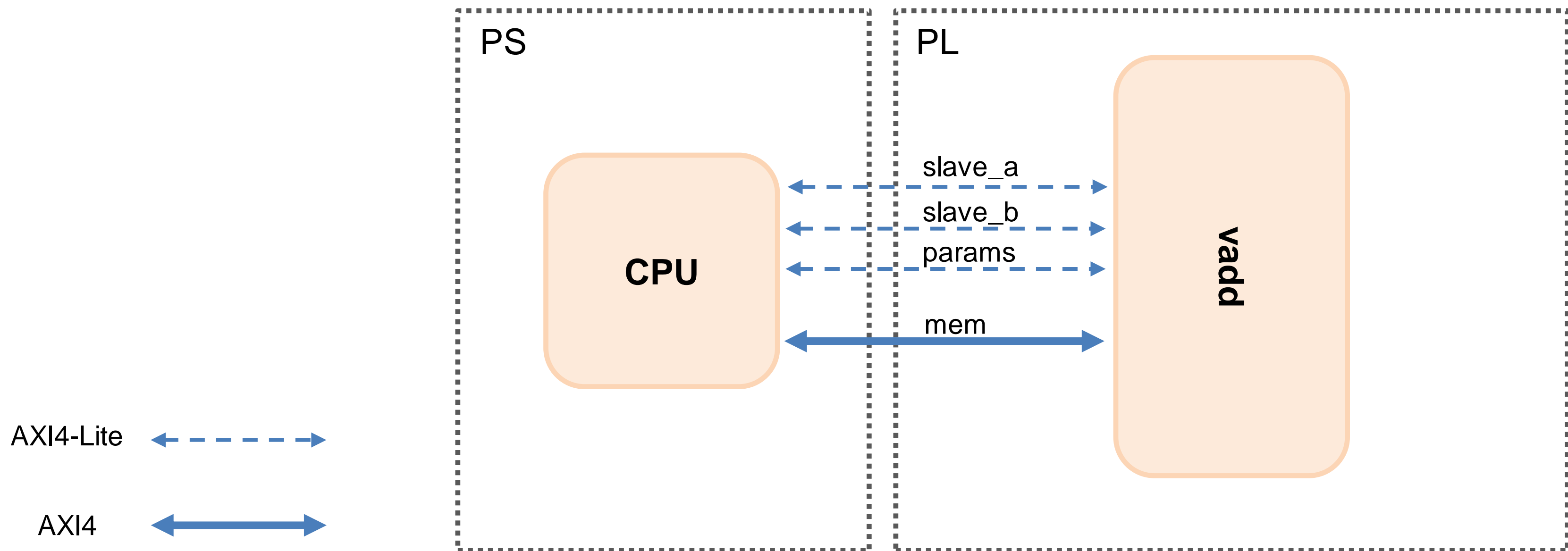


```
void  fun(int *a, int length){
        #pragma HLS INTERFACE m_axi port=a offset=slave depth=32 bundle=mem
        …
}
```

➔ *Il parametro depth permette di settare il quantitativo di memoria accessibile dal canale AXI4. E' opzionale, ma potrebbe essere necessario specificarlo in fase di co-simulazione.*

Interfacce AXI4 in HLS (4)

```
void  vadd(int *a, int * b, int length){
        #pragma HLS INTERFACE m_axi port=a offset=slave bundle=mem
        #pragma HLS INTERFACE m_axi port=b offset=slave bundle=mem
        #pragma HLS INTERFACE s_axilite port=length bundle=params
        #pragma HLS INTERFACE s_axilite port=return bundle=params
        …
}
```



Profiling

Profiling (1)

- *Il tool di HLS, dopo la sintesi fornisce in output un report (report.rpt) con alcune informazioni utili al profiling delle performance, tra cui:*
  - *Stima dei cicli di clock necessari all'esecuzione del modulo FPGA;*
  - *Stima del tempo di esecuzione, sulla base della frequenza di clock di riferimento;*
  - *Utilizzo di risorse FPGA.*
- *Per il profiling delle prestazioni è necessario che il tool di sintesi conosca il numero esatto di iterazioni di ogni loop, ad esempio:*
  - *Numero di iterazioni costante;*
  - *Direttiva #pragma di profiling.*

Profiling (2)

```
void fun(int iter){
        #pragma HLS INTERFACE s_axilite port=iter bundle=ctrl
        for(int i = 0; i < iter; i++) {
                ...
        }
}
```

➔ *Numero di iterazioni <u>non noto</u> a tempo di sintesi.*

```
#define iter <num_iter>

void fun(){
        for(int i = 0; i < iter; i++) {
                ...
        }
}
```

➔ *Ok, ma <u>non configurabile a runtime</u>.*

```
void fun(int iter){
        #pragma HLS INTERFACE s_axilite port=iter bundle=ctrl
        for(int i = 0; i < iter; i++) {
                #pragma HLS LOOP_TRIPCOUNT max=max_iter min=min_iter
                ...
        }
}
```

Esercizio 00 - guidato

exercise_0_solution.cpp

➔ *Implementare in Vivado HLS una somma di vettori.*

➔ *Realizzare i seguenti esperimenti:*

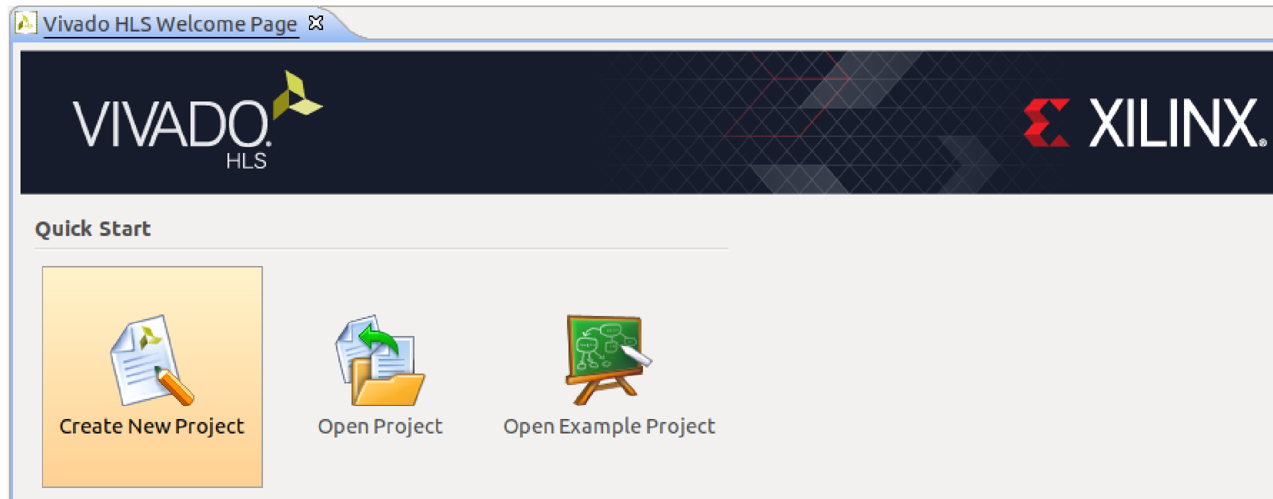
1. *Partire da un'implementazione software della vadd ed inserire le interfacce AXI4.*
2. *Leggere il report di Vivado HLS, in particolare:*
   - ➔ *Latenza: numero di cicli di clock impiegati dall'acceleratore;*
   - ➔ *Risorse: percentuale di risorse FPGA necessarie all'implementazione dell'acceleratore;*
   - ➔ *Interfacce AXI generate.*
3. *Vedere lo schedule viewer di Vivado HLS;*
4. *Realizzare una simulazione C++;*
5. *Realizzare una co-simulazione C++/RTL*

Esercizio 00 - guidato

Step 1

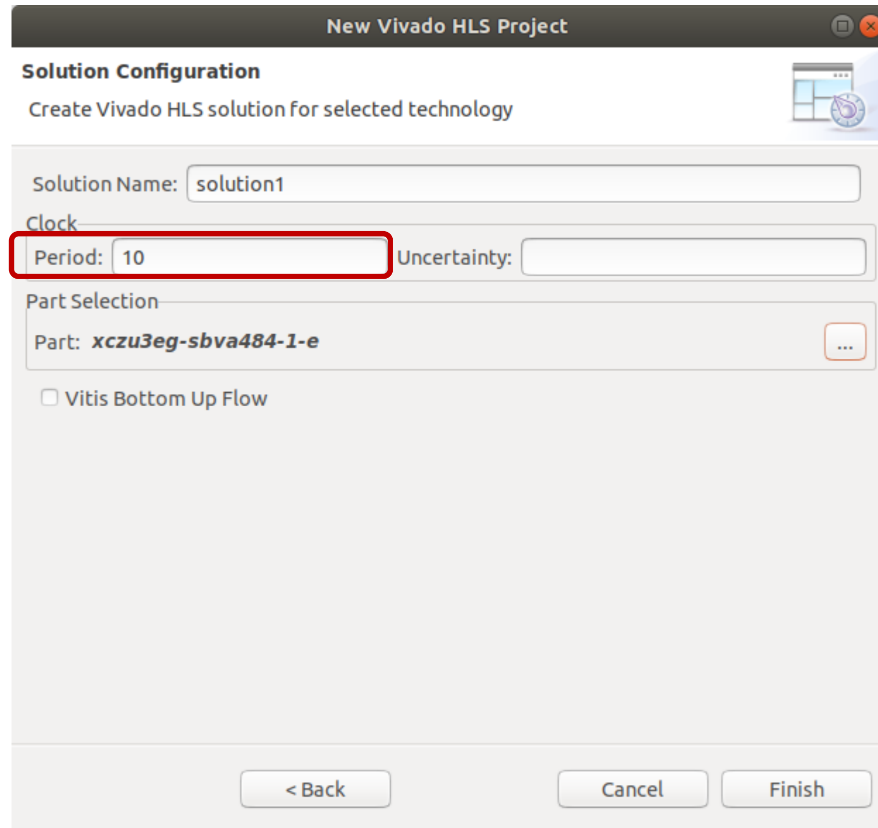
➔ *Creazione di un progetto:*

- ➔ *$ source /tools/Xilinx/Vivado/2020.1/settings64.sh*
- ➔ *$ vivado_hls*



Esercizio 00 - guidato
Step 1

➔ *Cliccare next fino all'ultima schermata:*



➔ *Selezionare il <u>periodo di clock</u> di riferimento;*

➔ *Il default è 10 (10 nanosecondi) che corrisponde ad una <u>frequenza di 100MHz</u>.*

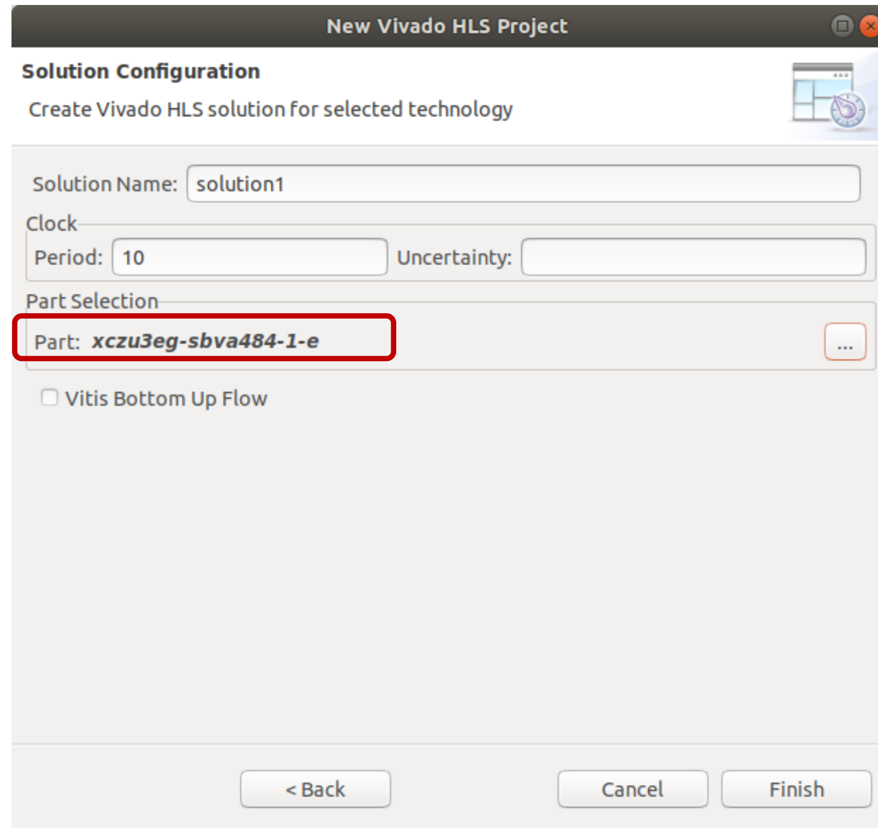
➔ *In questo modo verrà generato un acceleratore che indicativamente dovrà avere un clock in input di circa 100MHz.*

➔ *In caso di overclock è necessario ri-generare il modulo HLS, settando il periodo corretto.*

Esercizio 00 - guidato

Step 1

- ➔ *Cliccare next fino all'ultima schermata:*



- ➔ *Selezionare il System-on-Chip corretto.*
- ➔ *Nel nostro caso selezionare il part:*
  - ➔ *xczu3eg-sbva484-1-e*
- ➔ *Che corrisponde al SoC presente sulla dev-board Avnet Ultra96.*

Esercizio 00 - guidato

Step 1

```
#define TEST_DATA_SIZE 4194304 // 2^22

const unsigned int c_dim = TEST_DATA_SIZE;

void vadd_accel(int *a, int *b, int *c, const int len)
{
#pragma HLS INTERFACE m_axi port=a offset=slave bundle=mem
#pragma HLS INTERFACE m_axi port=b offset=slave bundle=mem
#pragma HLS INTERFACE m_axi port=c offset=slave bundle=mem
#pragma HLS INTERFACE s_axilite port=len bundle=params
#pragma HLS INTERFACE s_axilite port=return bundle=params

    vadd: for(int i = 0; i < len; i++) {
        #pragma HLS LOOP_TRIPCOUNT min=c_dim max=c_dim
        c[i] = a[i] + b[i];
    }
}
```

- ➔ *In questo modo l'acceleratore avrà un bus AXI4 per l'accesso alla memoria principale;*
- ➔ *Un bus AXI4-lite per l'accesso al register file da parte della CPU.*

Esercizio 00 - guidato
Step 2

Synthesis Report for 'vadd_accel'

General Information

Date: Tue May 11 13:36:11 2021
Version: 2020.1 (Build 2897737 on Wed May 27 20:21:37 MDT 2020)
Project: vadd_baseline
Solution: solution1
Product family: zynquplus
Target device: xczu3eg-sbva484-1-e

Performance Estimates

Timing

Summary

Clock	Target	Estimated	Uncertainty
ap_clk	10.00 ns	8.750 ns	1.25 ns

Latency

Summary

Latency (cycles)		Latency (absolute)		Interval (cycles)		
min	max	min	max	min	max	Type
50331654	50331654	0.503 sec	0.503 sec	50331654	50331654	none

Detail

Instance

Loop

	Latency (cycles)			Initiation Interval			
Loop Name	min	max	Iteration Latency	achieved	target	Trip Count	Pipelined
- vadd	50331648	50331648	12	-	-	4194304	no

- *Diverse informazioni tra cui:*
  - *Chip utilizzato;*
  - *Latenza dell'acceleratore in termini di cicli di clock e tempo di esecuzione;*
  - *Latenza dei loop dell'acceleratore.*

Esercizio 00 - guidato

Step 2

Utilization Estimates

Summary

Name	BRAM_18K	DSP48E	FF	LUT	URAM
DSP	-	-	-	-	-
Expression	-	-	0	175	-
FIFO	-	-	-	-	-
Instance	2	-	700	876	-
Memory	-	-	-	-	-
Multiplexer	-	-	-	162	-
Register	-	-	422	-	-
Total	2	0	1122	1213	0
Available	432	360	141120	70560	0
Utilization (%)	~0	0	~0	1	0

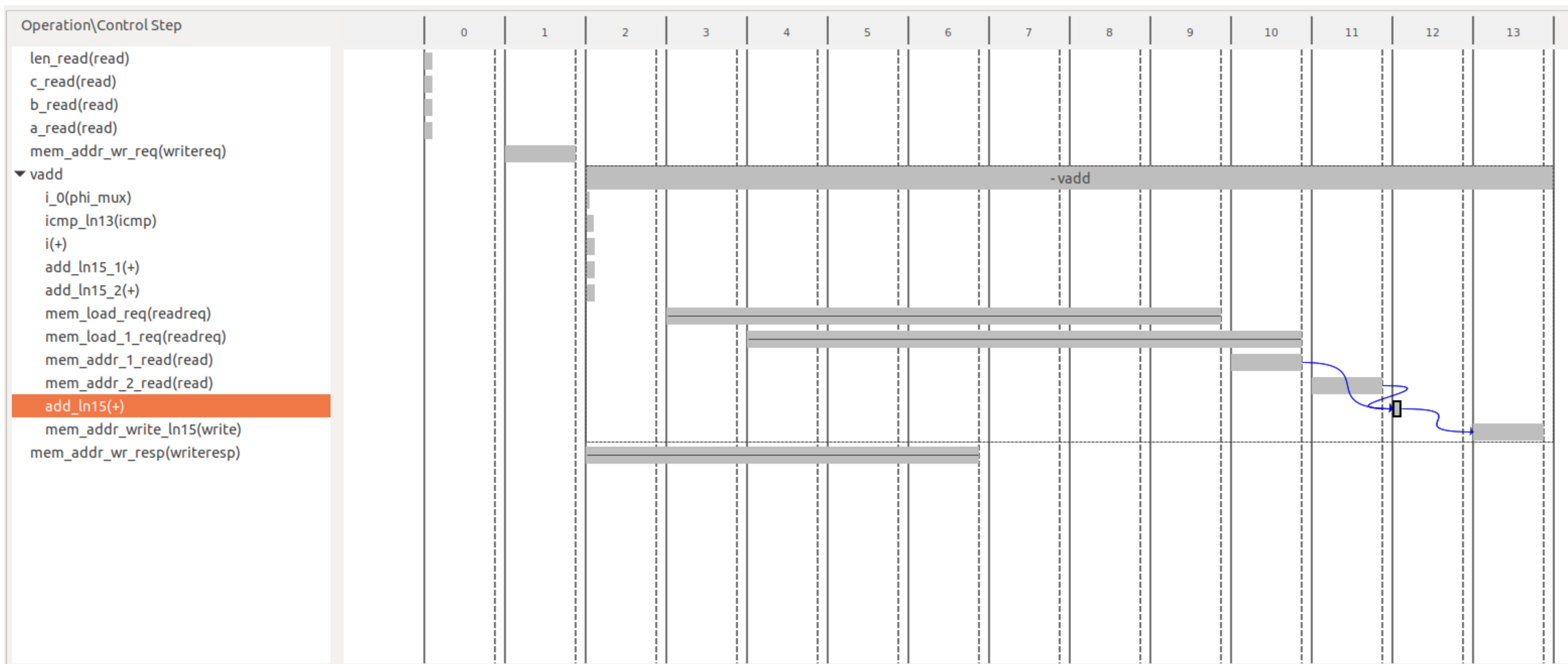
Detail

- **Instance**
- **DSP48E**
- **Memory**
- **FIFO**
- **Expression**
- **Multiplexer**
- **Register**

➔ *Utilizzo di risorse programmabili utilizzate dall'acceleratore.*

Esercizio 00 - guidato
Step 3

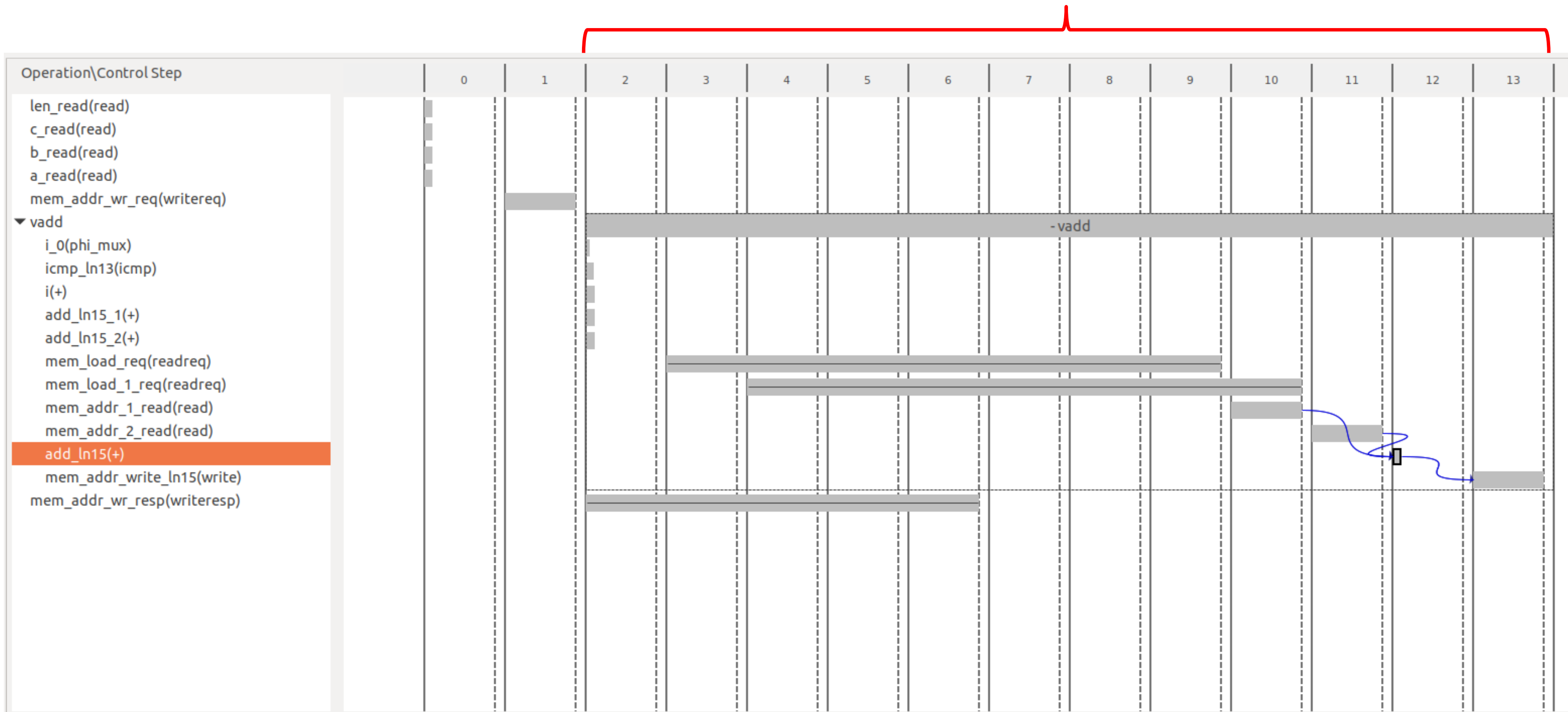
➔ *Cicli di clock necessari. Per il loop si considera una sola iterazione.*



Esercizio 00 - guidato

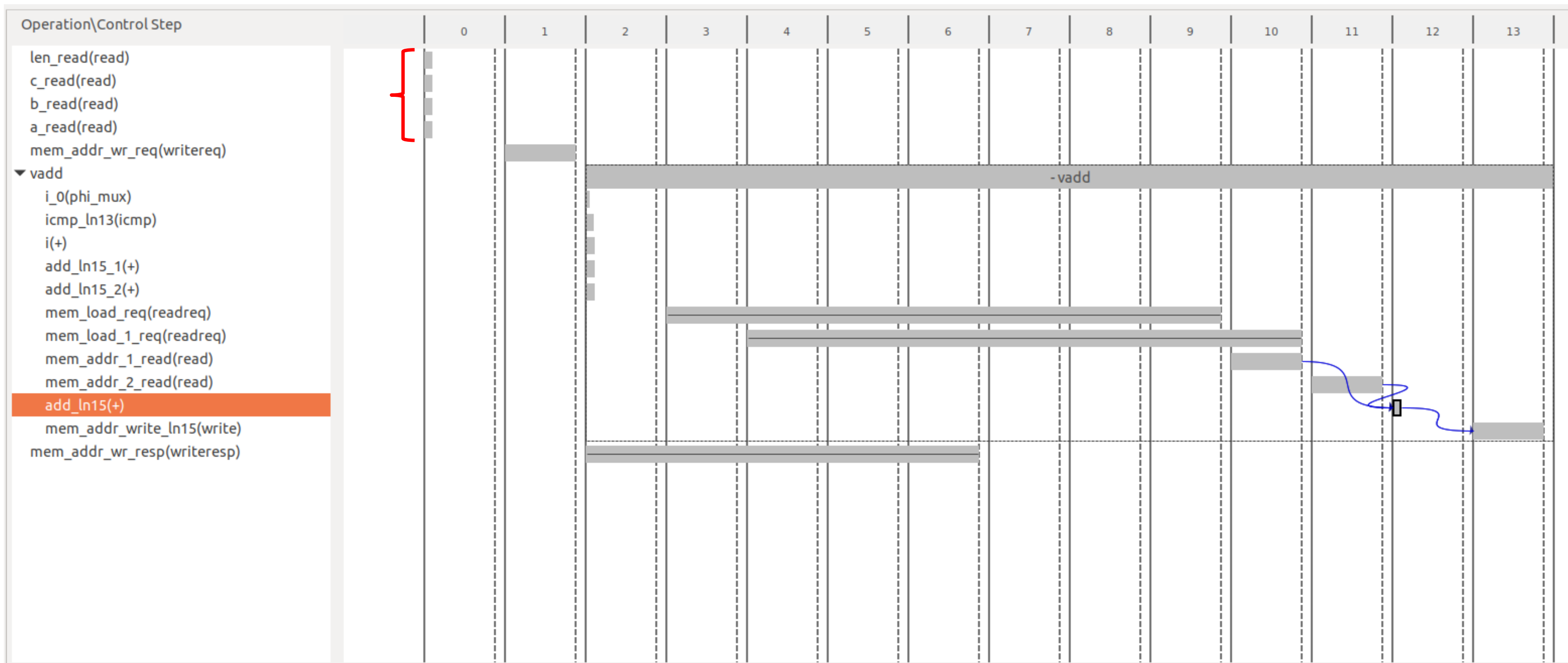
Step 3

➔ *Cicli di clock necessari per il solo loop for.*



Esercizio 00 - guidato
Step 3

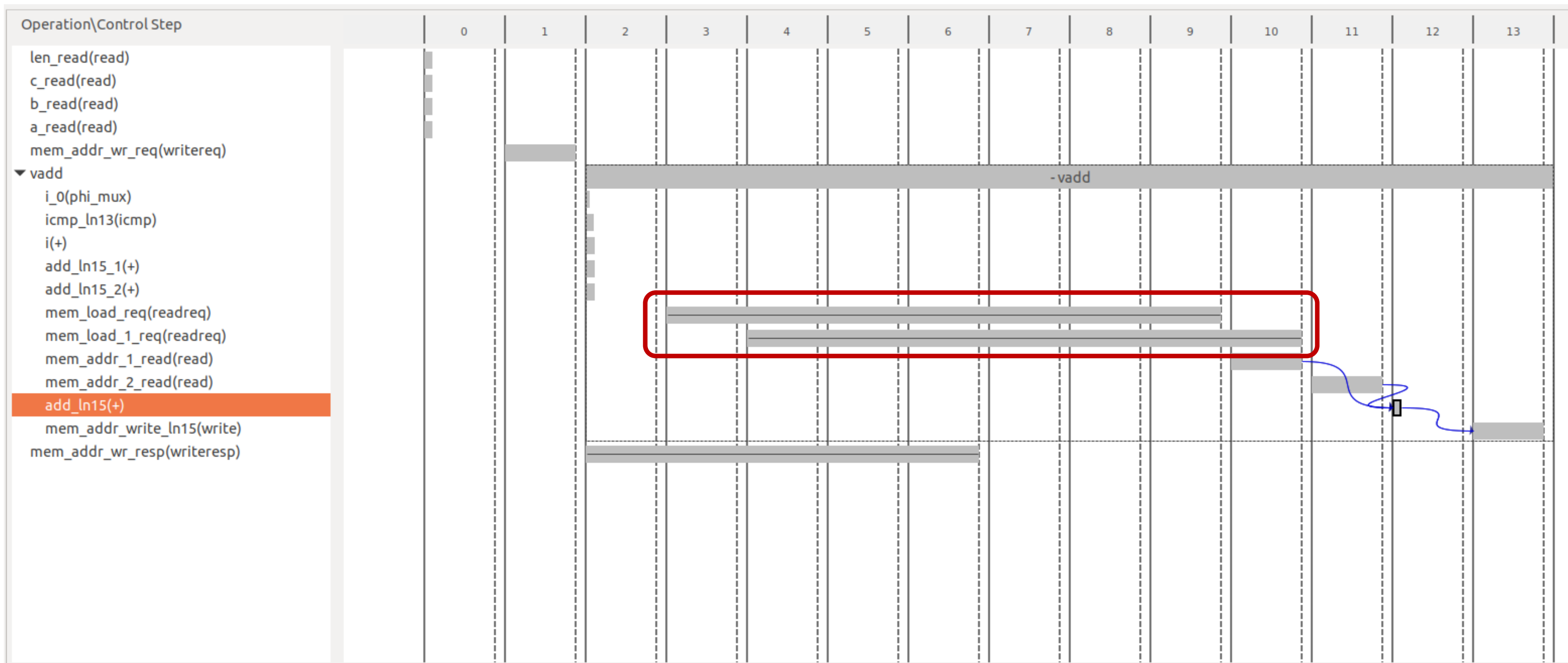
➔ *Lettura dei registri dell'acceleratore.*



Esercizio 00 - guidato

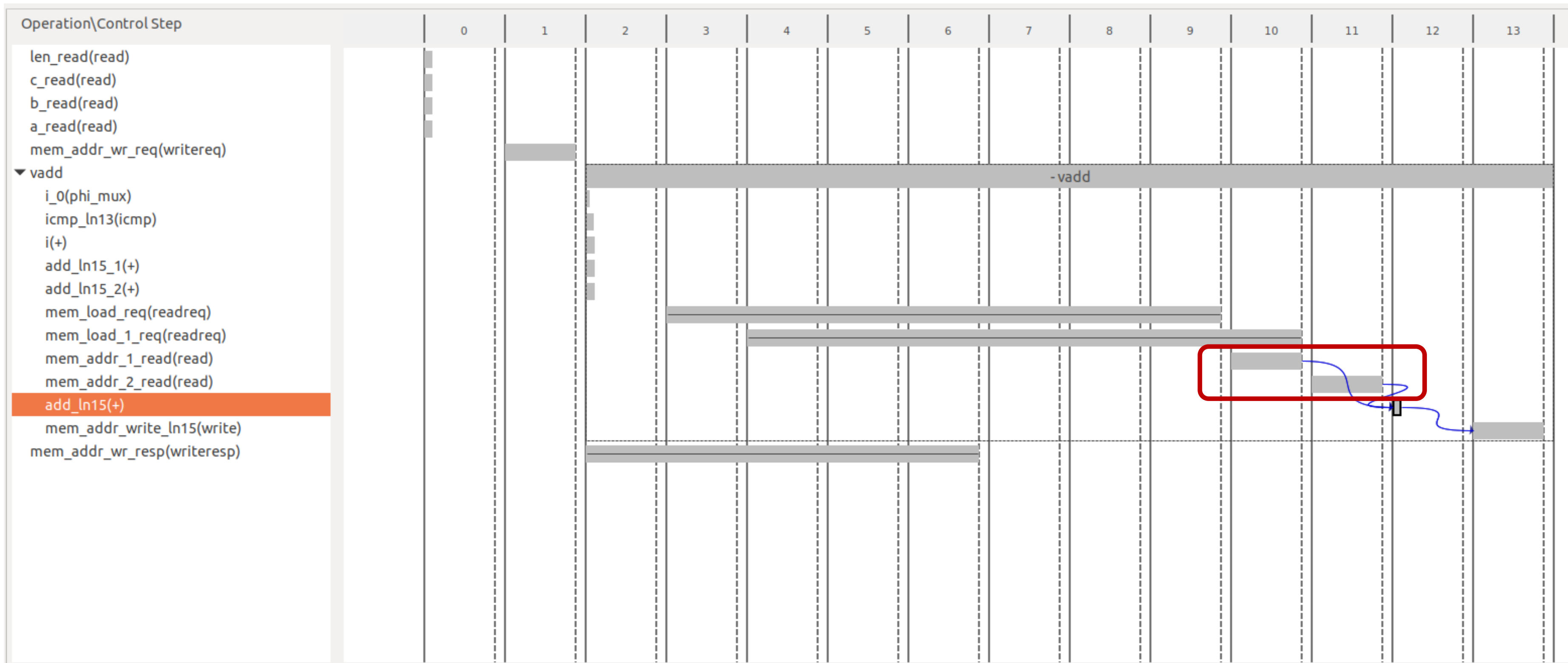
Step 3

➔ *Apertura dei canali AXI4 per la lettura degli elementi dei due vettori*



Esercizio 00 - guidato
Step 3

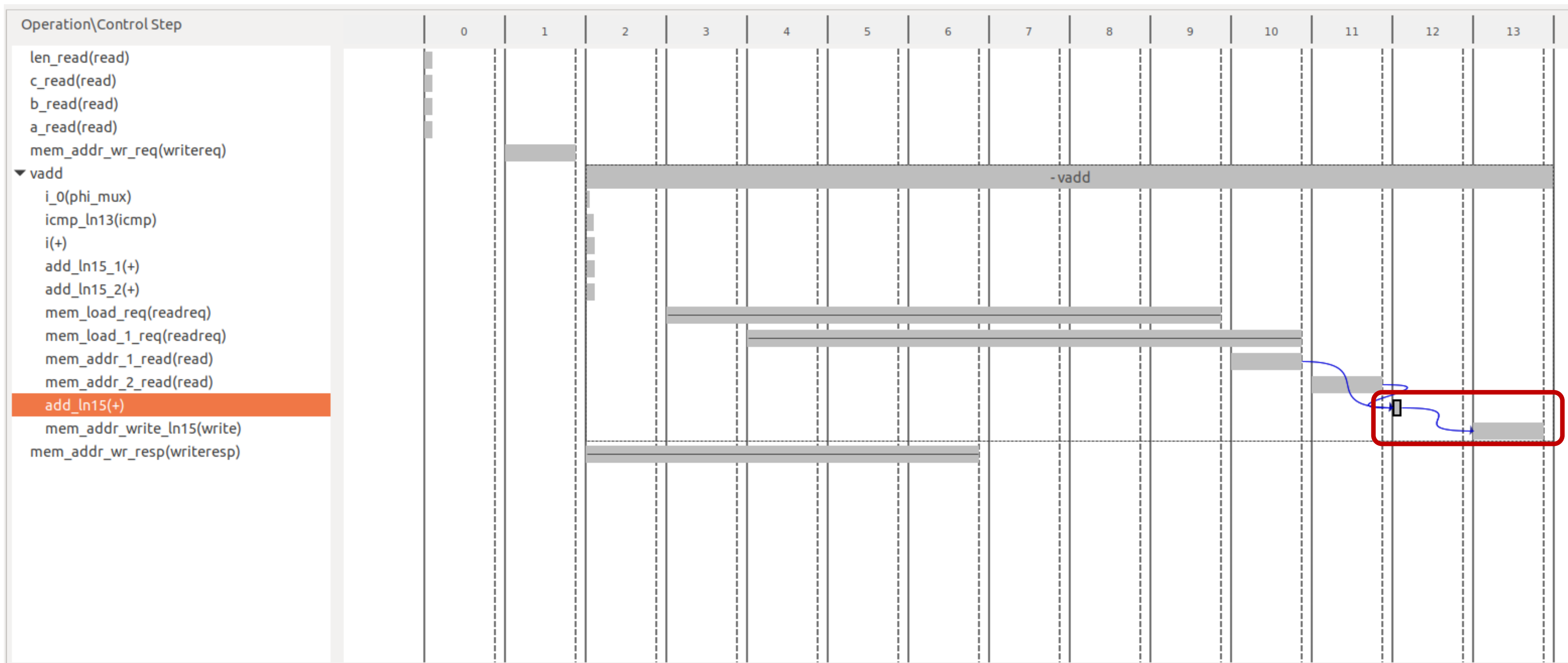
➔ *Lettura effettiva dei due dati da sommare per l'iterazione corrente. <u>Notiamo che le letture sono sequenziali</u>.*



Esercizio 00 - guidato

Step 3

➔ *Somma degli elementi i-esimi dei due vettori e successiva scrittura in memoria del dato.*



Esercizio 00 - guidato

Step 4

- ➔ *Aggiungiamo un file main.cpp per emulare in software l'esecuzione dell'acceleratore.*
- ➔ *Utile in fase di debugging.*

- ➔ *Test Bench > new file > main.cpp*

```
#include <stdlib.h>
#include <stdio.h>

#include "vadd.h"

int main () {

        int n = max_elem;

        int * a = (int *) malloc(n*sizeof(int));
        int * b = (int *) malloc(n*sizeof(int));
        int * c = (int *) malloc(n*sizeof(int));

        for (int i = 0; i < n; i++) {
                a[i] =  i;
                b[i] = 2*i;
        }

        for (int i = 0; i < n; i++) {
                c[i] = a[i] + b[i];
        }

        sum (a, b, c, n);

        FILE *fp = fopen("ground_truth.txt", "r");

        for (int i = 0; i < n; i++) {

                int buff;

                fscanf(fp, "%d", &buff);

                if (buff != c[i]) {
                        return 1;
                }
        }

        fclose(fp);

        free(a);
        free(b);
        free(c);

        return 0;
}
```

Esercizio 00 - guidato
Step 5

- *Testiamo una co-simulazione C++/RTL. In questo caso viene effettuata un'<u>emulazione C++ e Verilog/VHDL</u>, dove vengono comparati i risultati ottenuti.*
- *Se il risultato della co-simulazione è corretto, allora il comportamento dell'acceleratore RTL è lo stesso dell'algoritmo C/C++;*

- *La co-simulazione è utile quando non siamo certi di come una porzione di codice C/C++ venga tradotta in RTL.*

Cosimulation Report for 'vadd_accel'

Result

		Latency			Interval		
RTL	Status	min	avg	max	min	avg	max
VHDL	NA	NA	NA	NA	NA	NA	NA
Verilog	Pass	280	280	280	NA	NA	NA

Esercizio 00 - guidato

Step 5

- *Nel nostro caso potrebbe essere utile ridurre la dimensione massima dell'acceleratore per ridurre i tempi di co-simulazione (es: TEST_DATA_SIZE = 32).*

- *Potrebbe essere necessario inserire il parametro «depth» nelle direttive m_axi:*

```
#pragma HLS INTERFACE m_axi port=a offset=slave depth=c_dim bundle=mem
#pragma HLS INTERFACE m_axi port=b offset=slave depth=c_dim bundle=mem
#pragma HLS INTERFACE m_axi port=c offset=slave depth=c_dim bundle=mem
```

- *Tale parametro è utilizzato solo in co-simulazione ed indica la dimensione della porzione di memoria accessibile da un determinato bus.*

Ottimizzazioni

Ottimizzazioni

#01 Parallelismo sulle Interfacce AXI4

- ➔ *Come abbiamo visto dallo schedule viewer, la lettura dei due vettori avviene in maniera sequenziale.*
- ➔ *Questo è dovuto al fatto che i vettori «a», «b» e «c» condividono lo stesso bus AXI4. Possiamo gestire i bus AXI4 tramite il parametro «bundle» della #pragma HLS INTERFACE:*

```
void fun(int *a, …)
{
        #pragma HLS INTERFACE m_axi port=a offset=slave depth=c_dim bundle=a_mem
        …
}
```

- ➔ *Differenziando questo parametro è possibile quindi utilizzare dei bus concorrenti.*

Ottimizzazioni

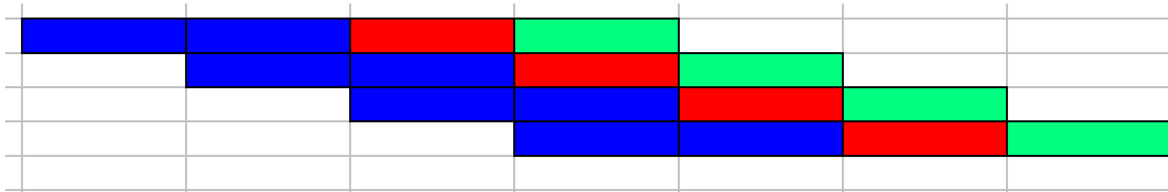
#02 Loop Pipelining

- *Introduciamo la prima ottimizzazione, che consiste nel creare una pipeline sulle iterazioni di un 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
#02 Loop Pipelining

- *Il pipelining riduce il tempo di attesa di un'iterazione di un loop, affinchè inizi la sua esecuzione.*
- *L'effetto è quello di incrementare il throughput del modulo FPGA.*

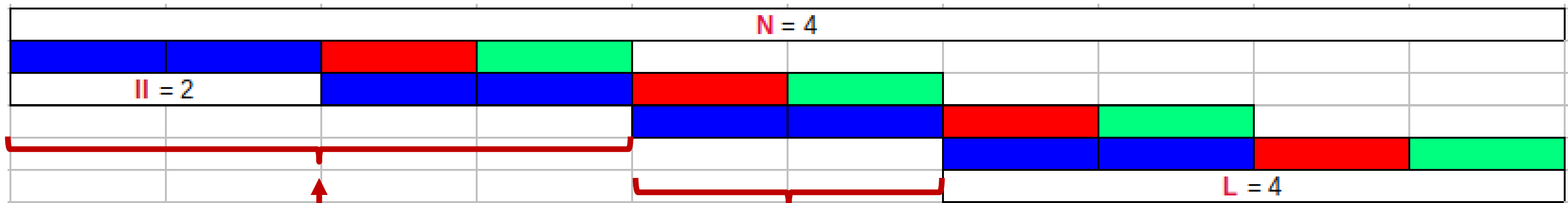


- *In HLS si può utilizzare la seguente direttiva:*

```
loop: for(int i = 0; i < size; i++) {
        #pragma HLS PIPELINE
        ...
}
```

Ottimizzazioni

#02 Loop Pipelining



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

Esercizio 01
exercise_1.cpp

➔ *Testiamo l'ottimizzazione del pipelining e del parallelismo sulle interfacce, tramite un esempio di somma di due vettori.*

➔ *Svolgere i seguenti esperimenti:*

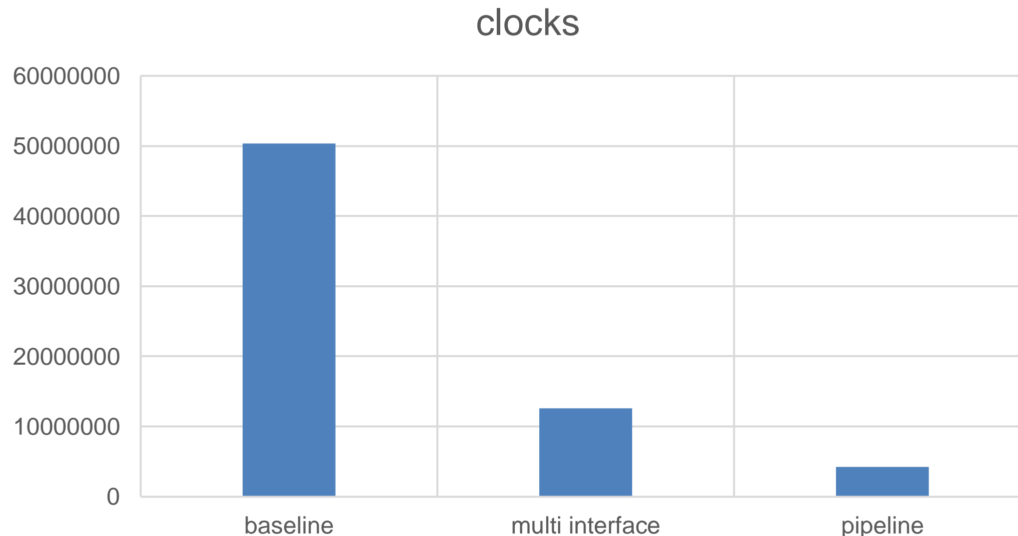
1. *Implementare una versione utilizzando un bus AXI4 per ogni vettore;*
2. *Modificare il codice ed inserire la direttiva di pipelining;*
3. *Calcolare lo speedup delle due versioni rispetto al caso senza pipelining e singolo bus AXI4..*

Esercizio 01
Comparazione - speedup

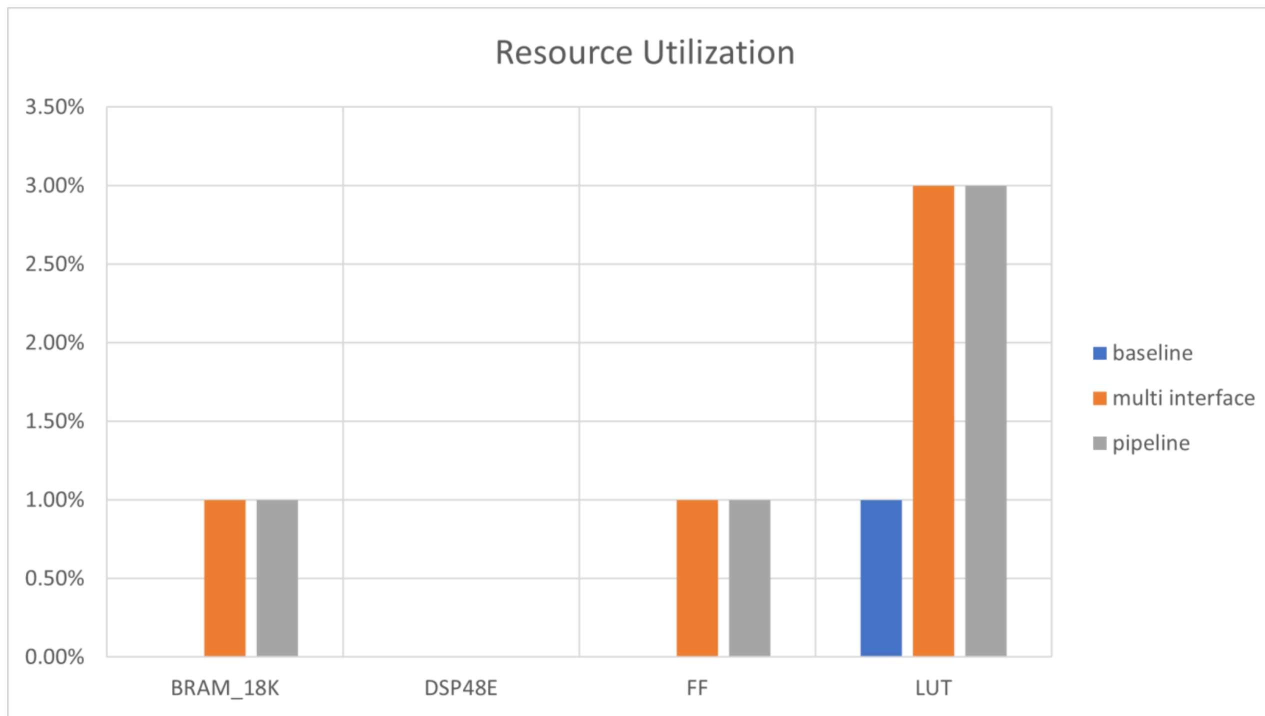
	baseline	multi interface	pipeline
clocks	50331654	12582924	4194319

	baseline	multi interface	pipeline
speedup	1	3.999996662	11.99996

- ➔ *Tipo di dato int;*
- ➔ *n = 2^22 elementi;*
- ➔ *11.9x di speedup ottenuto.*



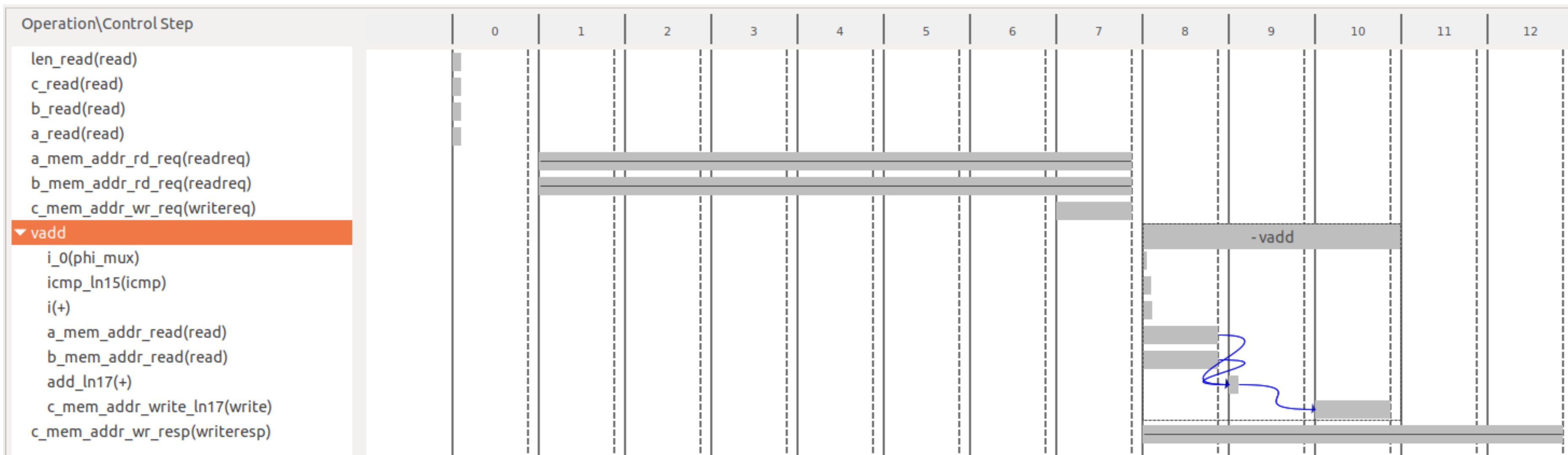
Esercizio 01
Comparazione – utilizzo di risorse



➔ *Quasi nessun incremento dal punto di vista delle risorse necessarie.*

Esercizio 01
Comparazione – schedule viewer

- ➔ *Notevoli differenze rispetto alla versione baseline, in particolare:*
- ➔ *L'apertura dei canali AXI4 di lettura viene fatta una volta sola prima di entrare nel loop.*
- ➔ *Le letture del dato all'iterazione corrente vengono svolte in parallelo nello stesso ciclo di clock.*

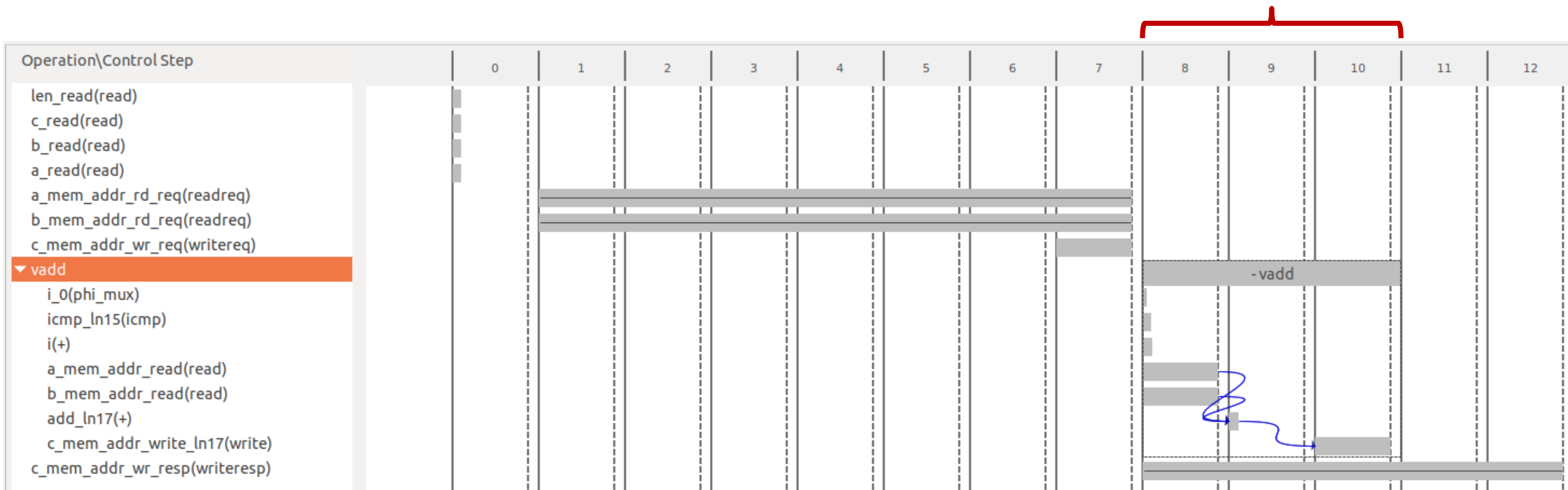


Esercizio 01

Comparazione – schedule viewer

➔ *Inoltre il <u>costo di una singola iterazione</u> (in termini di cicli di clock) è notevolmente ridotto:*

➔ *<u>3 cicli di clock necessari</u>, rispetto ai 12 cicli di clock della versione precedente.*



Ottimizzazioni

#03 Loop Fusion

- ➔ *In certi casi dei loop adiacenti possono essere fusi in un unico loop al fine di ridurre l'overhead di gestione del loop ed incrementare le performance.*
- ➔ *In ambito hardware può essere efficacemente <u>combinato con la direttiva pipeline</u>.*

```
for (i = 0; i < 300; i++)
  a[i] = a[i] + 3;

for (i = 0; i < 300; i++)
  b[i] = b[i] + 4;
```

```
for (i = 0; i < 300; i++)
 {
   a[i] = a[i] + 3;
   b[i] = b[i] + 4;
 }
```

Esercizio 02

exercise_2.cpp

- *Implementare l'algoritmo <u>Nearest Neighbor</u> testare le ottimizzazioni pipelining e loop fusion.*
- *L'algoritmo data una matrice «points» ed un vettore «search_points» di input permette di determinare il vettore all'interno di «points» più vicino a «search_points».*

$$search_points = \begin{bmatrix} 1 \\ 2 \\ 0 \\ 3 \\ 4 \end{bmatrix} \quad points = \begin{bmatrix} 2 & 3 & 0 & 1 & 4 & 5 & 7 & 1 \\ 0 & . & . & . & . & . & . & . \\ 1 & . & . & . & . & . & . & . \\ 3 & . & . & . & . & . & . & . \\ 2 & . & . & . & . & . & . & . \end{bmatrix}$$

Esercizio 02

exercise_2.cpp

➔ *Svolgere i seguenti esperimenti:*

1. *Scaricare l'implementazione dell'algoritmo nearest neighbor (exercise_2.cpp);*
2. *Fondere i due loop in uno unico;*
3. *Inserire la direttiva pipeline;*
4. *Calcolare lo speedup ottenuto.*

Ottimizzazioni

#03 Loop Fusion (in Vivado HLS)

- ➔ *Direttiva che permette di fondere **loop_1** con tutti i loop superiori nella gerarchia;*
- ➔ *Loop fusion spesso automaticamente aggiunta nei loop pipelined;*

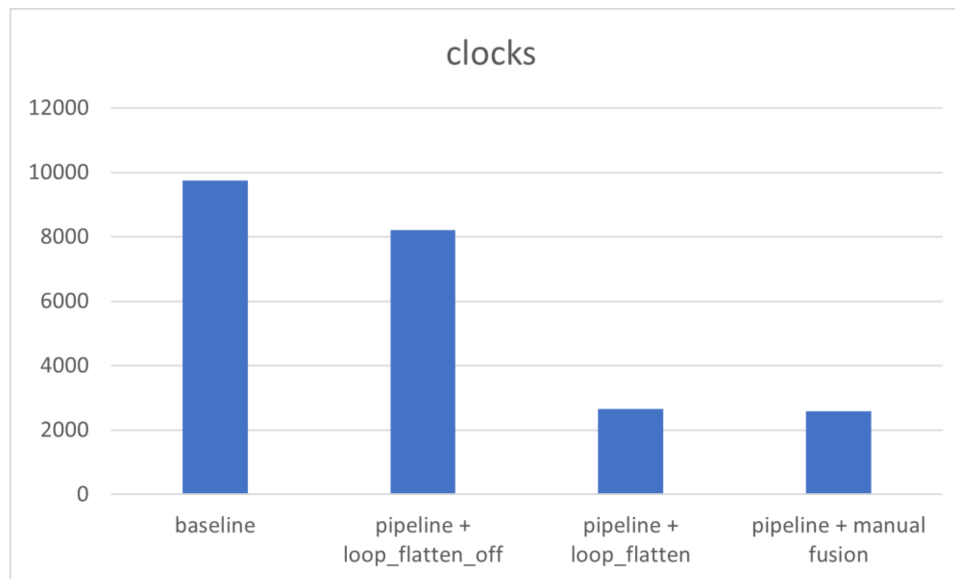
- ➔ ***#pragma HLS LOOP_FLATTEN off**, disabilita la fusion automatica del loop.*

```
void foo (num_samples, ...) {
  ...
  loop_1: for(int i = 0; i < N; i++) {
      ...
      #pragma HLS LOOP_FLATTEN
      ...
    }
}
```

Esercizio 02
Comparazione - speedup

	baseline	pipeline	pipeline + loop_flatten	pipeline + manual fusion
clocks	9751	8212	2657	2589

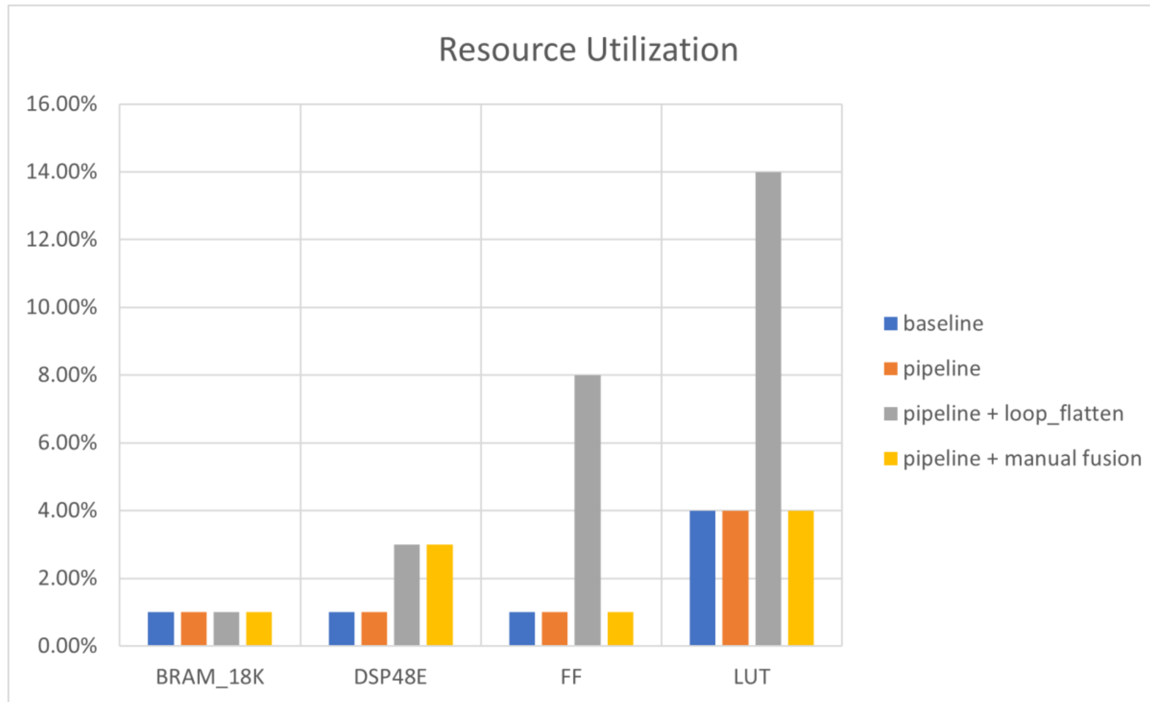
	baseline	pipeline	pipeline + loop_flatten	pipeline + manual fusion
speedup	1.00	1.19	3.67	3.77



- ➔ *Tipo di dato int;*
- ➔ *NUM_DIMS 5*
- ➔ *NUM_POINTS 512*
- ➔ *3.77x di speedup ottenuto*

Esercizio 02

Comparazione – utilizzo di risorse



➔ *Leggero incremento sull'utilizzo di DSP e Flip-Flop.*

Esercizio 03
exercise_3.cpp

➔ *Implementare l'algoritmo di moltiplicazione tra due matrici.*

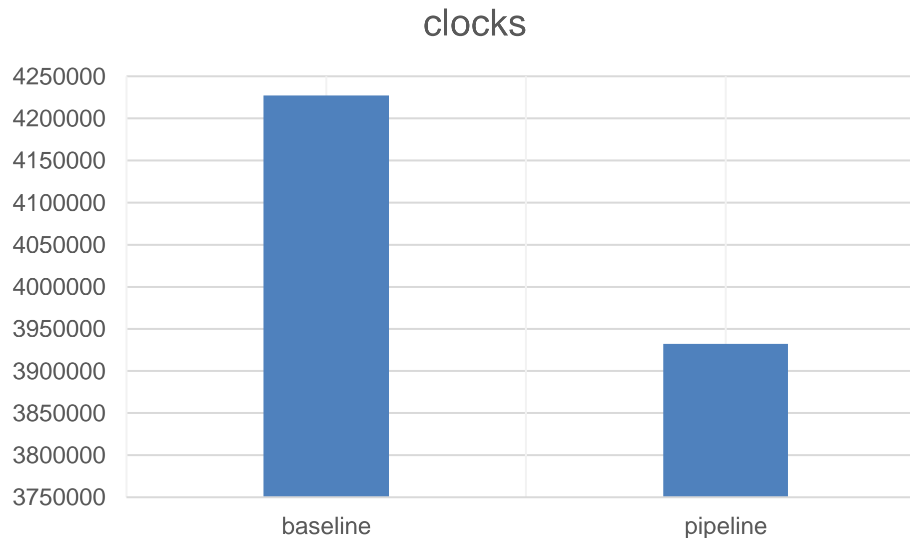
➔ *Svolgere i seguenti esperimenti:*

1. *Scaricare e sintetizzare una versione baseline, senza ottimizzazioni (exercise_3.cpp);*
2. *Inserire la direttiva pipeline nel loop più interno;*
3. *Calcolare lo speedup ottenuto.*

Esercizio 03
Comparazione - speedup

	baseline	pipeline
clocks	4227201	3932262

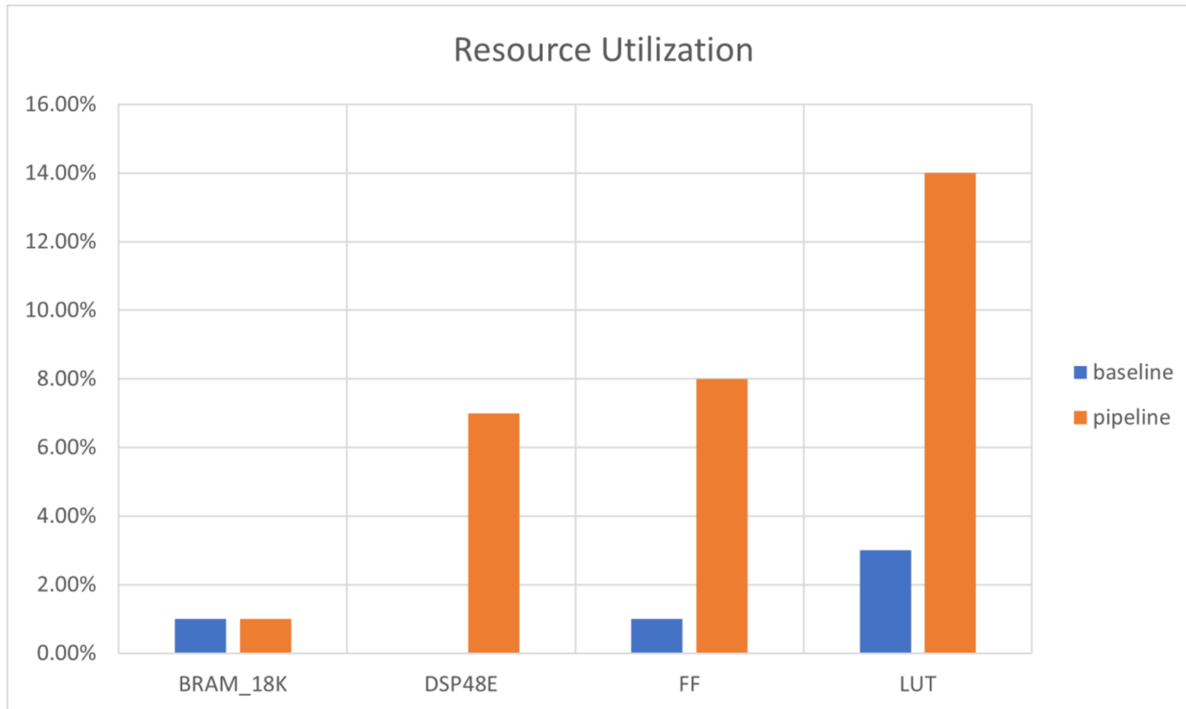
	baseline	pipeline
speedup	1	1.075004921



- ➔ *Tipo di dato int;*
- ➔ *N = 64*
- ➔ *1.07x di speedup ottenuto*

Esercizio 03

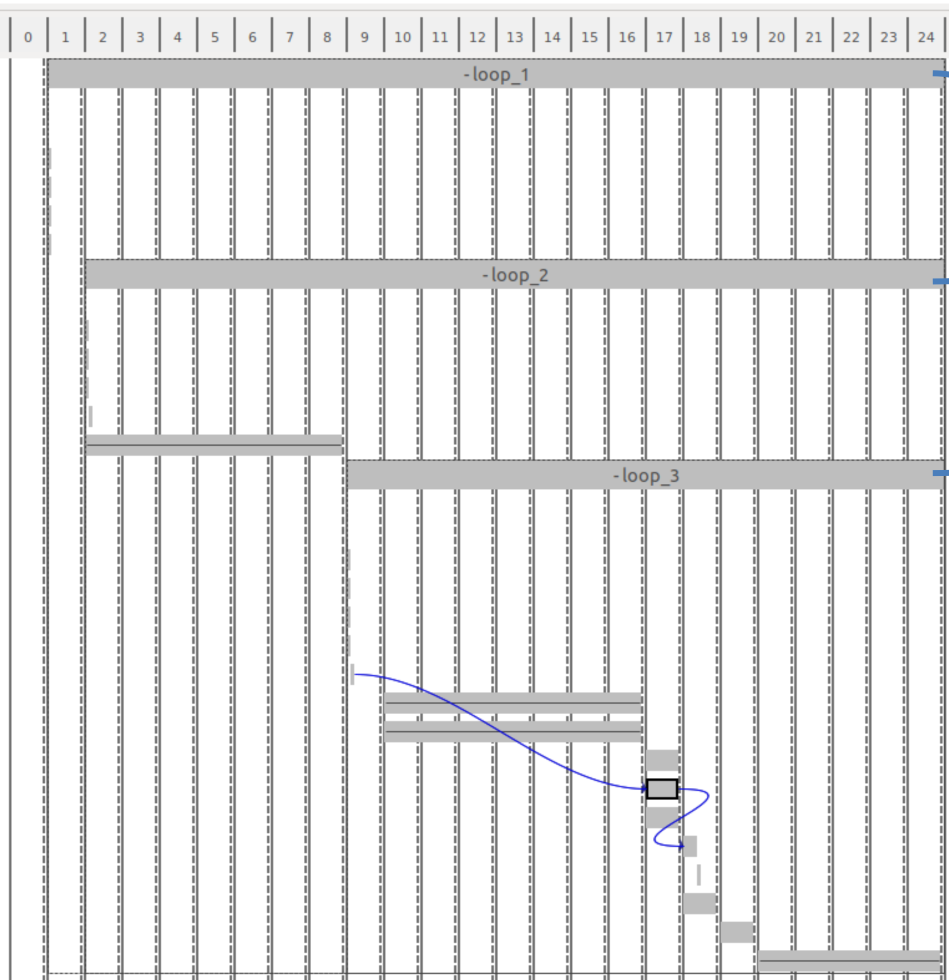
Comparazione – utilizzo di risorse



➔ *Rispetto agli esempi precedenti, l'introduzione del pipelining comporta un overhead maggiore.*

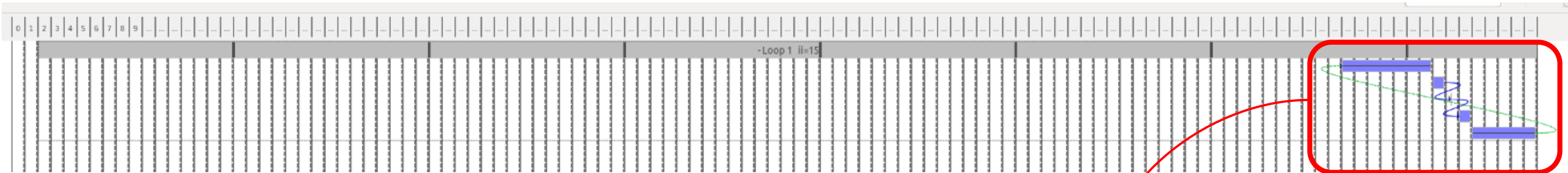
Esercizio 03

Comparazione – scheduler




```
loop_1: for (int i = 0; i < dim; i++){

#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

        loop_2: for (int j = 0; j < dim; j++){

#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

                loop_3: for (int k = 0; k < dim; k++){

            //TODO: insert pipeline directive
#pragma HLS LOOP_TRIPCOUNT max=max_size min=max_size

            out[i * dim + j] += in1[i * dim + k] * in2[k * dim  + j];
        }
    }
}
```

Esercizio 03
Comparazione – scheduler



- ➔ *Versione **Pipelined**;*
- ➔ *<u>Loop fusion automatico</u> dei tre loop innestati.*

- ➔ *Vivado HLS evidenzia il ciclo di letture/scritture da DRAM, come operazione che ha causato un II maggiore di 1 nel loop pipelined;*
- ➔ *Lo step successivo sarà l'ottimizzazione degli accessi in memoria.*

Ottimizzazioni

#04 Block RAM

- *Le Block RAM (BRAM) sono delle piccole memorie sparse all'interno del FPGA, sono molto veloci, ma rappresentano una risorsa limitata.*
- *Possono essere utilizzate per implementare delle piccole scratchpad accessibili dai kernels FPGA, in modo da ottimizzare gli accessi in memoria.*

- *In HLS si possono utilizzare istanziando degli array interni ai moduli, ad esempio:*

```
void fun(int * in1) {

    #pragma HLS INTERFACE m_axi port=in1 offset=slave bundle=in1_mem

    int v1_buffer[MAX_SIZE_LOC];                                    // Block RAM
    memcpy(v1_buffer, in1, MAX_SIZE_LOC*sizeof(int));

    ...
    ...
}
```

Esercizio 04
exercise_4.cpp

➔ *Partire dal codice pipelined della mmult ed implementare una versione basata su BRAM.*

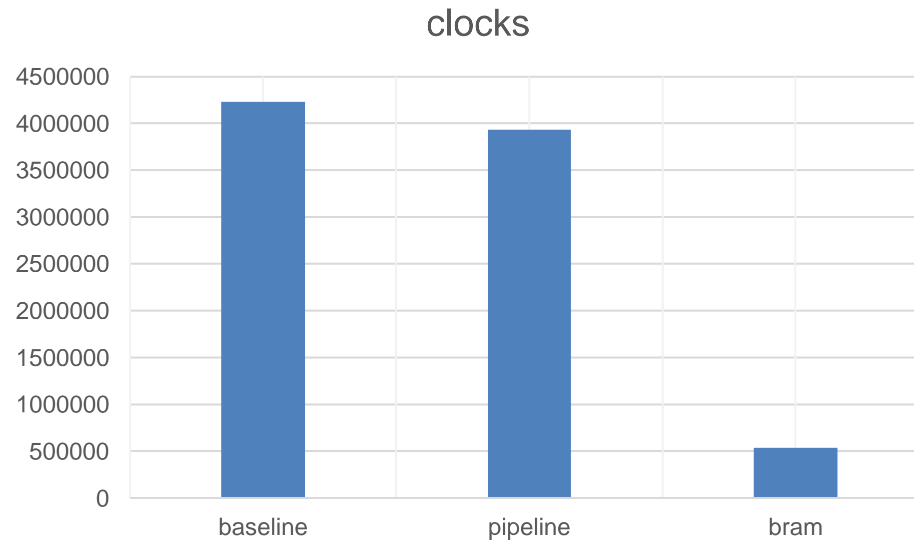
➔ *Svolgere i seguenti esperimenti:*

1. *Realizzare tre matrici in BRAM;*
2. *Caricare i valori da DRAM a BRAM nelle due matrici di input, per mezzo di due memcpy;*
3. *Dopo il calcolo scrivere la matrice di output in DRAM;*
4. *Calcolare lo speedup ottenuto.*

Esercizio 04
Comparazione - speedup

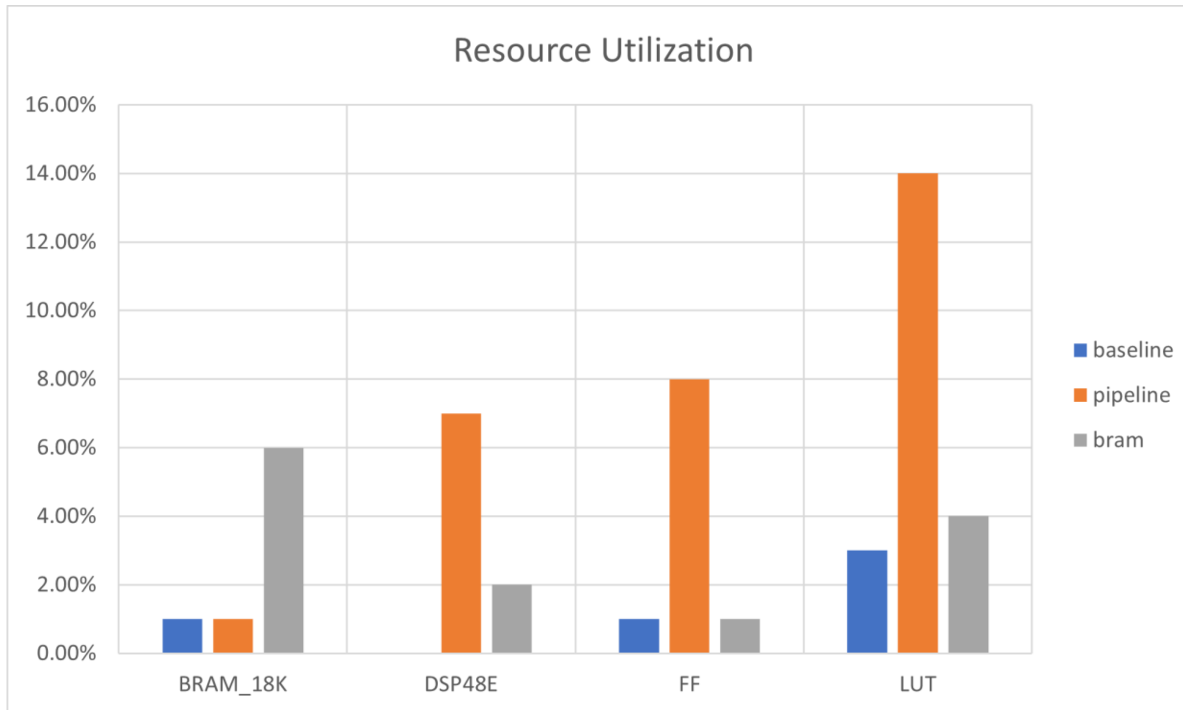
	baseline	pipeline	bram
clocks	4227201	3932262	536605

	baseline	pipeline	bram
speedup	1	1.075004921	7.877677249



- *Tipo di dato int;*
- *N = 64*
- *7.87x di speedup ottenuto*

Esercizio 04
Comparazione – utilizzo di risorse

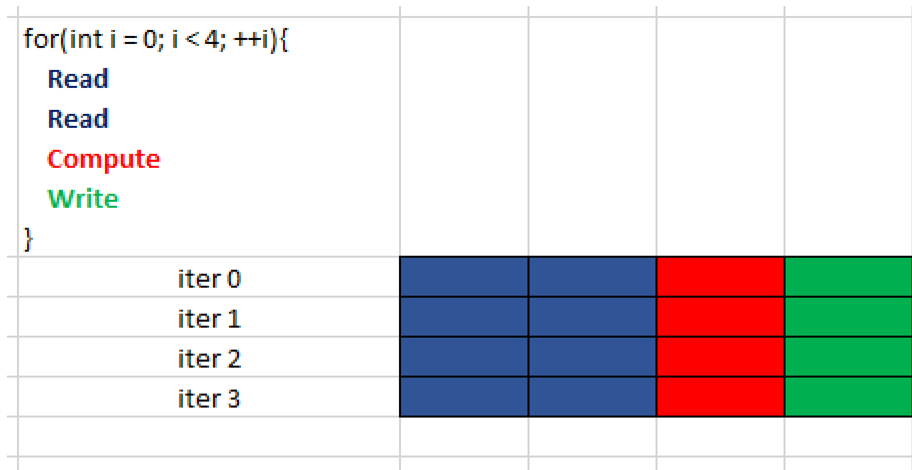


- ➔ *Leggero incremento sull'utilizzo di BRAM;*
- ➔ *In questo caso riduzione di tutte le altre risorse.*

Ottimizzazioni

#05 Loop Unrolling

➔ *Tramite il Loop Unrolling l'hardware che esegue l'iterazione del loop viene replicato per ogni iterazione.*



➔ *Il numero di iterazioni del loop* ***deve essere costante***.

```
loop: for(int i = 0; i < size; i++) {
            #pragma HLS UNROLL
            ...
}
```

Ottimizzazioni
#05 Loop Unrolling

- *L’unrolling può essere anche svolto implicitamente andando inserire una direttiva Pipeline.*
- *A tutti i loop interni viene applicata la direttiva Unroll.*

- *Esempio:*

```
loop: for(int i = 0; i < size; i++) {

        #pragma HLS PIPELINE

        loop: for(int j = 0; j < size; j++) {
        #pragma HLS UNROLL
                ...
        }
}
```

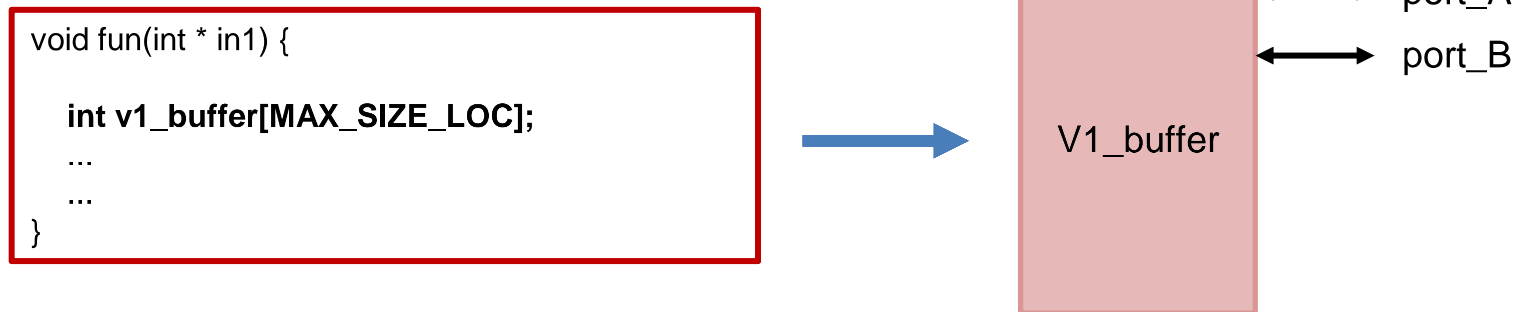
(`#pragma HLS UNROLL` is struck through on the slide; an arrow points to the inner loop.)

- *Automaticamente unrollato.*

Ottimizzazioni
#06 Array Partitioning

- ➔ *L'utilizzo del Loop Unrolling e della Block RAM viene molto spesso utilizzato assieme.*
- ➔ *Tuttavia l'accesso parallelo delle iterazioni unrollate di un loop può causare slowdown dovuti a contesa sulle porte delle BRAM:*

- ➔ *Esempio:*



- ➔ *Una BRAM dichiarata in questo modo ha solamente due porte.*

Ottimizzazioni

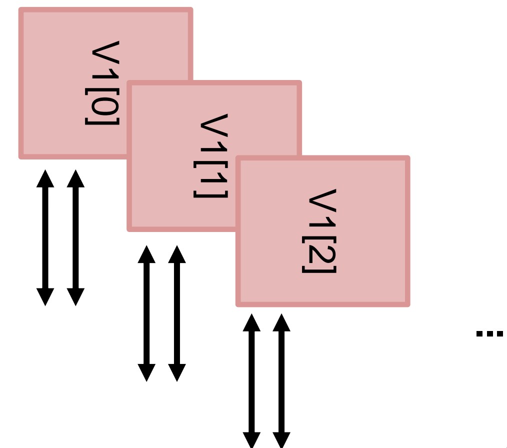
#06 Array Partitioning

- ➔ *L'idea dell'Array Partitioning è quella di sparpagliare l'array su più BRAM distinte.*

- ➔ ***pro**:*
  - ➔ *Aumenta notevolmente il parallelismo di accesso alla memoria.*

- ➔ ***contro**:*
  - ➔ *Le Block RAM non sono utilizzate al 100%, perciò l'utilizzo di BRAM sale notevolmente.*

```
void fun(int * in1) {

    int v1_buffer[MAX_SIZE_LOC];

    #pragma HLS ARRAY_PARTITION variable=v1_buffer complete dim=1
    ...
}
```



Esercizio 05

exercise_5.cpp

- *Partire dall'ultima versione del codice relativo alla matrix multiplication, quello con BRAM*

- *Svolgere i seguenti esperimenti:*
  1. *Implementare una versione con Loop Unrolling sul loop più interno;*
  2. *Implementare una versione con Array Partitioning sulle due matrici di input:*
     - *Matrice in1_loc: acceduta per colonne, partitioning su dim=2.*
     - *Matrice in2_loc: acceduta per righe, partitioning su dim=1.*
  3. *Inserire la direttiva Pipelining sul loop intermedio;*
  4. *Calcolare lo speedup ottenuto delle tre versioni realizzate.*

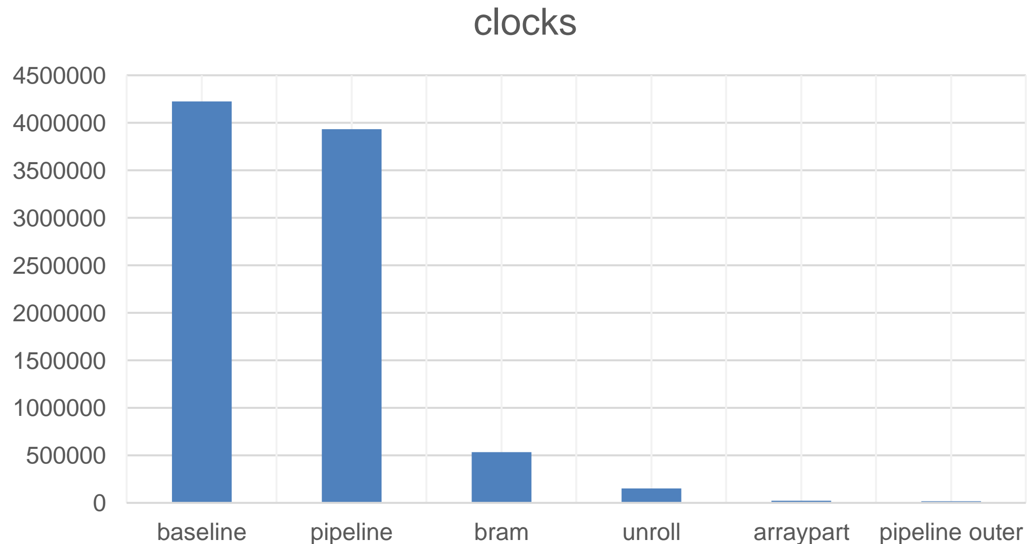
Esercizio 05

Comparazione - speedup

	baseline	pipeline	bram	unroll	arraypart	pipeline outer
clocks	4227201	3932262	536605	153755	24795	16413

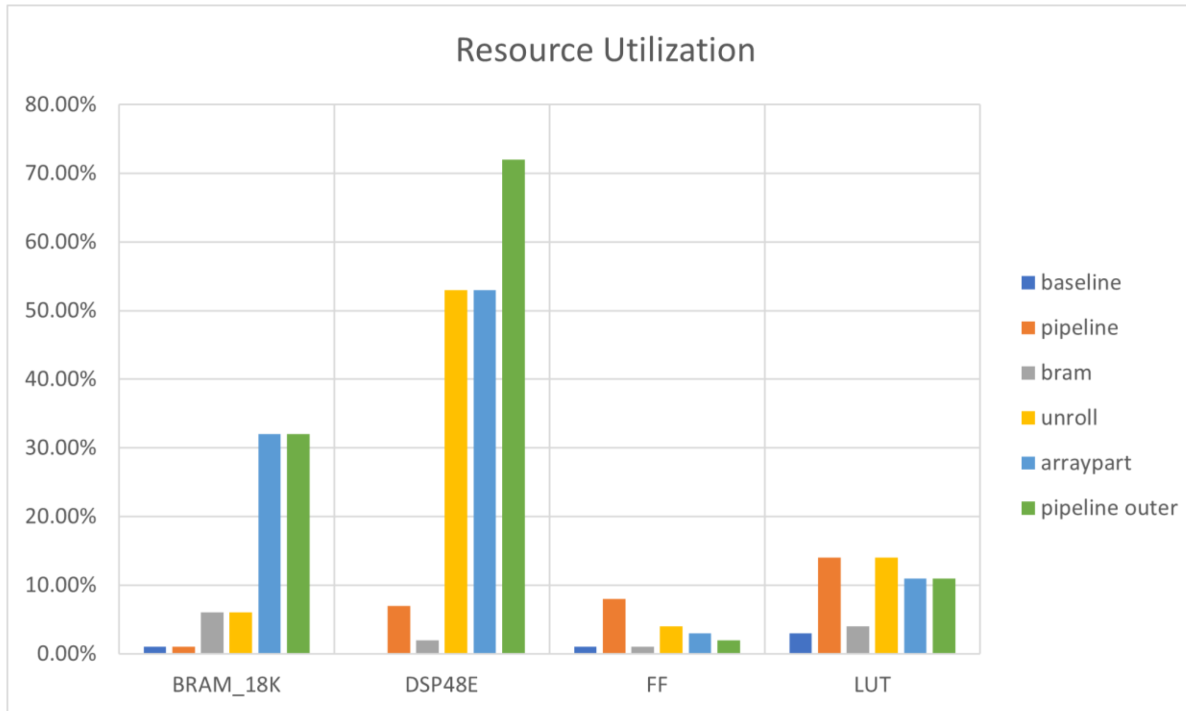
	baseline	pipeline	bram	unroll	arraypart	pipeline outer
speedup	1	1.075004921	7.877677249	27.4931	170.486	257.5520015

- ➔ *Tipo di dato int;*
- ➔ *N = 64*
- ➔ *257.5x di speedup ottenuto*



Esercizio 05

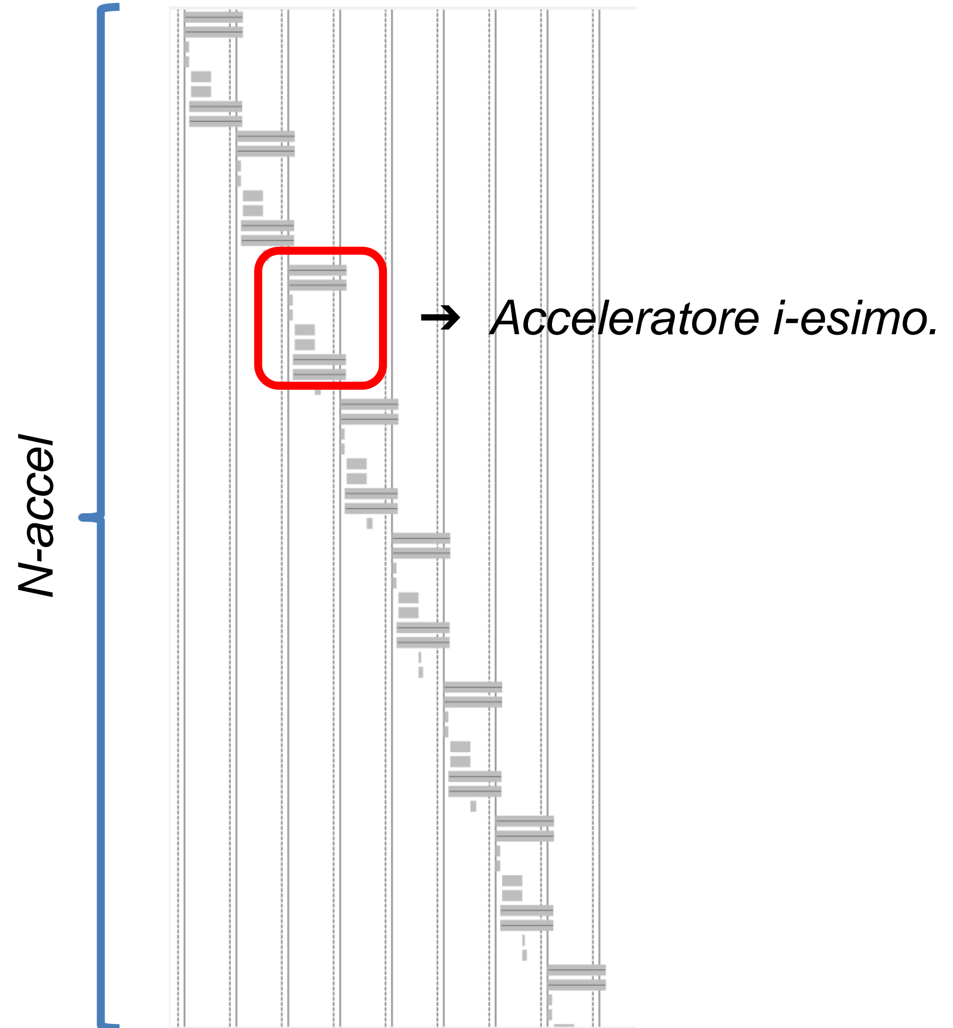
Comparazione – utilizzo di risorse



- ➔ *L'__unroll__ comporta un notevole aumento dei DSP (~**50%**);*
- ➔ *L'__arraypart__ comporta un incremento di **~25%** sull'utilizzo di BRAM, a causa del sotto-utilizzo;*
- ➔ *Il __pipelining__ comporta un ulteriore incremento di DSP.*

Esercizio 03
Comparazione – scheduler

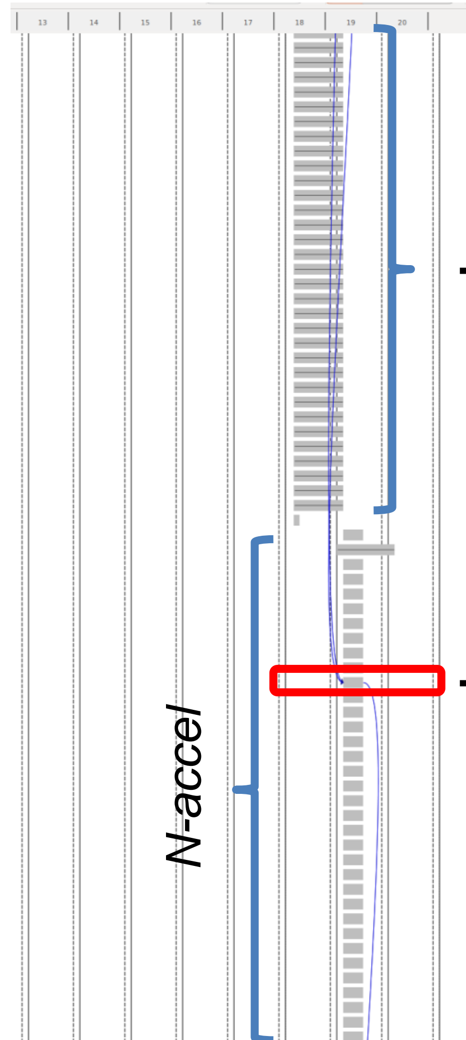
- ➔ *Versione **Unrolled**;*
- ➔ ***N-acceleratori paralleli**;*
- ➔ *In questo caso il ridotto numero di porte sulle BRAM, causa contesa sugli accessi;*
- ➔ *Nonostante siano presenti N-acceleratori, l'esecuzione non avviene effettivamente in parallelo*



Esercizio 03

Comparazione – scheduler

- ➔ *Versione* ***Unrolled*** *con* ***array partitioning****;*
- ➔ ***N-acceleratori paralleli****;*
- ➔ *In questo caso gli acceleratori sono schedulati in parallelo, sfruttando tutte le porte sulle BRAM.*



- ➔ *Le letture/scritture avvengono effettivamente in parallelo.*
- ➔ *Acceleratore i-esimo.*

Esercizio 06 - 07

Exercise_6.cpp

➔ *Rivediamo le ottimizzazioni andando ad implementare un Finite Impulse Response Filter (FIR Filter).*

$$y[n] = \sum_{i=0}^{N} b_i x[n-i]$$

$x[n]$ input signal

$y[n]$ output signal

N filter order

b_i *i*th filter coefficient

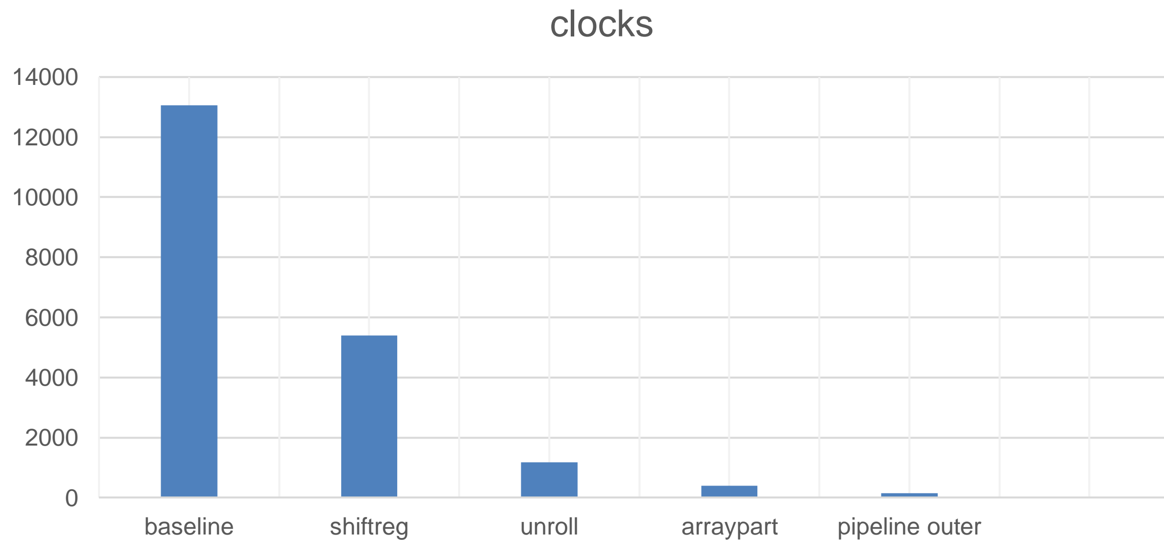
1. *Implementazione di una versione baseline, senza ottimizzazioni (exercise_6.cpp).*
2. *Implementazione di una versione del FIR basata su shift-register:*
   - ➔ *Scaricare il file exercise_7.cpp;*
   - ➔ *Comprenderne il funzionamento;*
   - ➔ *Calcolare lo speedup rispetto alla versione baseline;*
3. *Inserire una direttiva di Unrolling sui due loop interni;*
4. *Partizionare lo shift register tramite la direttiva Array Partitioning.*
5. *Implementare una Pipeline sul loop più esterno.*

Esercizio 06

Comparazione - speedup

	baseline	shiftreg	unroll	arraypart	pipeline outer
clocks	13062	5397	1173	405	152

	baseline	shiftreg	unroll	arraypart	pipeline outer
speedup	1	2.420233463	11.13555	32.25185	85.93421



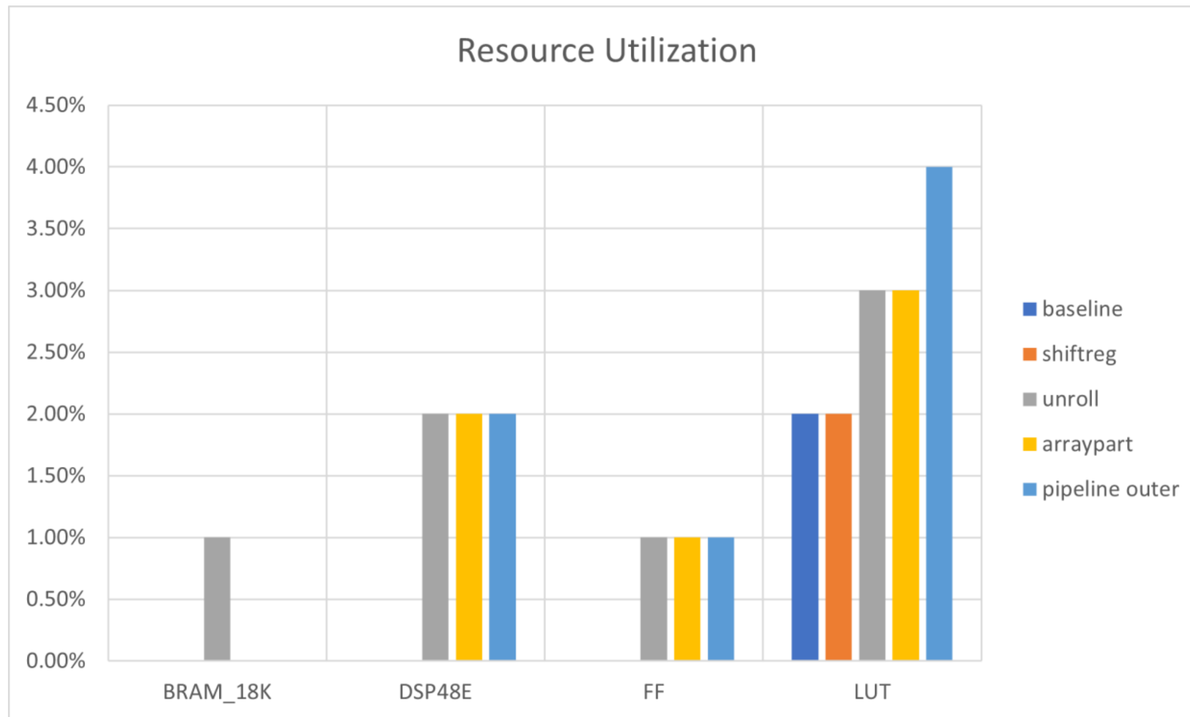
- *Tipo di dato int;*

- *N = 10*
- *SIZE = 128*

- *85.9x di speedup ottenuto*

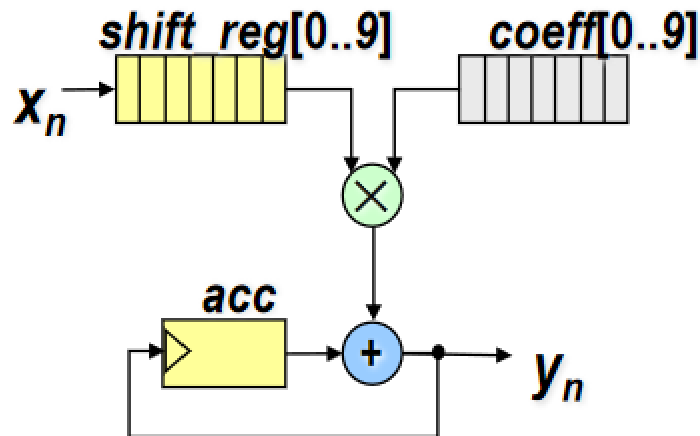
Esercizio 06

Comparazione – utilizzo di risorse



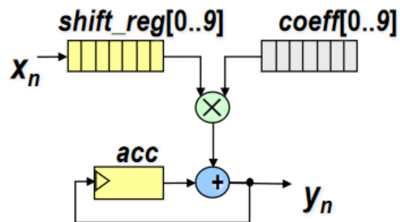
- ➔ *Il trend è simile agli altri esempi;*
- ➔ *Il consumo di risorse in questo caso è molto limitato.*

Esercizio 06
Comparazione – Soluzione 1

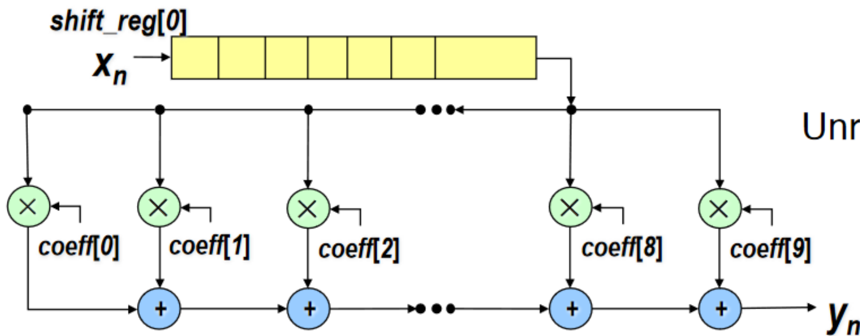


- ➔ *Versione con shift register;*
- ➔ *In questo caso l'input viene bufferizzato su una BRAM;*
- ➔ *Comporta tempi di accesso minori.*

Esercizio 06
Comparazione – Soluzione 2



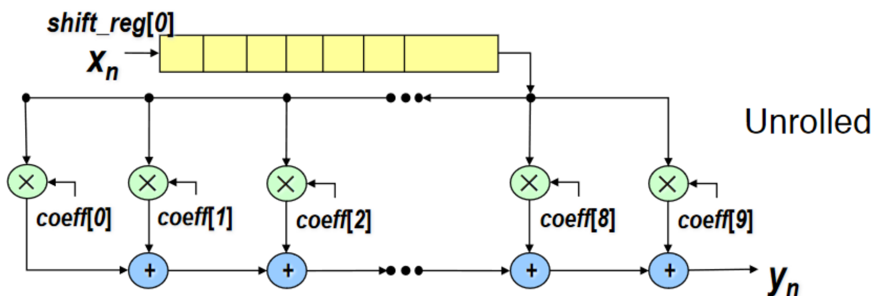
Default



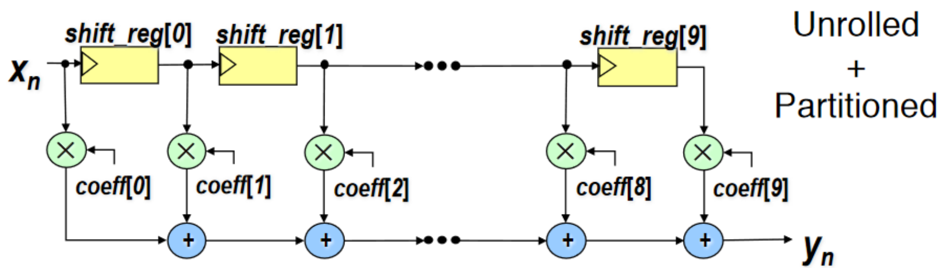
Unrolled

➔ *L'unrolling comporta la creazione di adder paralleli sulle iterazioni del loop.*

Esercizio 06
Comparazione – Soluzione 2



Unrolled

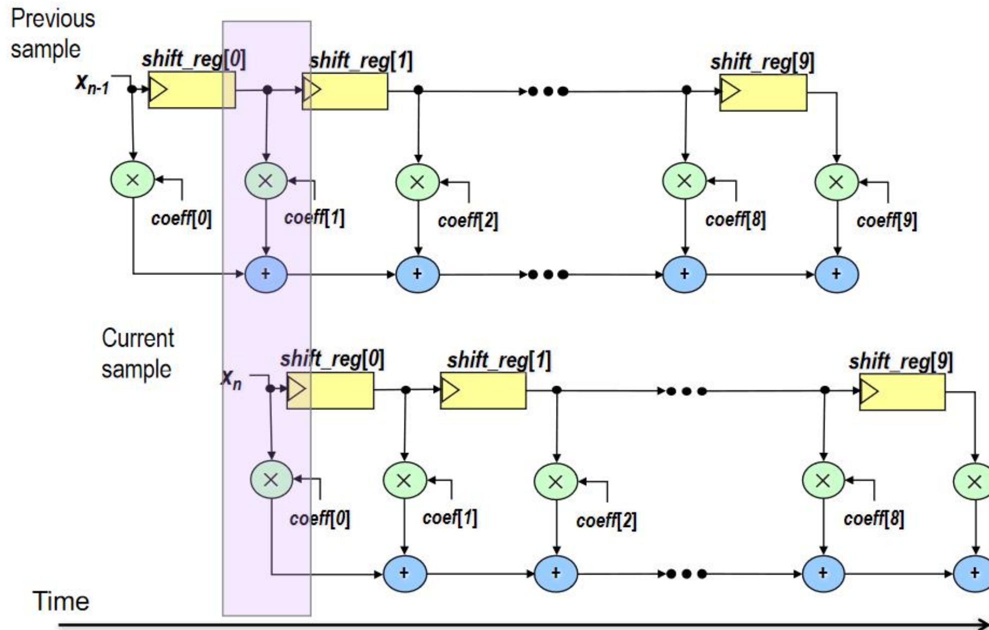


Unrolled + Partitioned

➔ *In questo caso l'array partitioning comporta maggior parallelismo sull'accesso allo shift register.*

Esercizio 06

Comparazione – Soluzione 2



➔ *Il pipelining finale infine permette di esprimere parallelismo anche sul loop esterno.*