High Level Synthesis of Digital Systems
2023-2024
Prof.ssa PERRI
Prof. FRUSTACI
FIR Filter Analysis

Giorgio Ubbriaco
247284
bbrgrg00h11d086x@studenti.unical.it

Maggio 2024

Index

Listings

List of Figures

List of Tables

1 Introduction

1.1 LTI Systems

Un **sistema a tempo discreto (TD) SLS o LTI** è un sistema che risulta essere contemporaneamente lineare e stazionario. In particolare, un sistema si dice **lineare** se ad una combinazione lineare degli ingressi corrisponde la combinazione lineare delle uscite (*Principio di Sovrapposizione degli Effetti*):

$$\alpha x_1 + \beta x_2 \xrightarrow{\tau} \alpha y_1 + \beta y_2 \tag{1}$$

Invece, un sistema TD si dice **stazionario** (o tempo-invariante) se ritardando/anticipando l'ingresso di una quantità n_0 l'uscita viene ritardata/anticipata della stessa quantità:

$$x[n - n_0] \xrightarrow{\tau} y[n - n_0] \tag{2}$$

Un sistema SLS-TD è caratterizzato dalla sua risposta impulsiva $h[n]$. Essa rappresenta l'uscita del sistema quando all'ingresso viene applicato un impulso discreto.

Conoscendo la risposta impulsiva h[n] e applicando le proprietà di linearità e stazionarietà, è possibile calcolare l'uscita $y[n]$ corrispondente ad un qualunque ingresso $x[n]$:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k] = x[n] * h[n] \tag{3}$$

Pertanto, l'uscita si ottiene effettuando la convoluzione tra l'ingresso e la risposta impulsiva.

1.2 Causal Systems

Un sistema (non necessariamente SLS) si dice causale se l'uscita all'istante n dipende solo dai valori precedenti dell'ingresso:

$$y[n] = \sum_{k=-\infty}^{+\infty} x[k]h[n-k] = \sum_{k=-\infty}^{+\infty} h[k]x[n-k] \tag{4}$$

Affinché un sistema SLS-TD sia causale è necessario e sufficiente che:

$$h[k] = 0 \quad \text{per ogni} \quad k < 0 \tag{5}$$

1.3 FIR Filter

I principali filtri digitali sono i filtri FIR (Finite Impulse Response) e IIR (Infinite Impulse Response). In particolare, un filtro FIR è un sistema LTI causale con risposta finita all'impulso, la cui funzione di trasferimento risulta essere un polinomio in z^{-1}. In particolare, considerando $h[n]$, cioè la sequenza di coefficienti del filtro in questione, e la sequenza di ingresso $x[n]$, l'obiettivo è calcolare l'uscita $y[n]$ mediante convoluzione:

$$y[n] = \sum_{k=0}^{N-1} h[k]x[n-k] \tag{6}$$

1.4 High Level Synthesis

Il tool **Xilinx® Vivado® High-Level Synthesis (HLS)** permette di trasformare le specifiche C in un'implementazione RTL (Register Transfer Level) che può essere sintetizzata e implementata in un FPGA. Pertanto, mediante linguaggi di programmazione ad alto livello come C, C++ e SystemC, è possibile descrivere in maniera più semplice e veloce un'architettura hardware complessa rispetto ad un linguaggio descrittivo dell'hardware come il VHDL o il Verilog. Ovviamente, tale vantaggio si traduce in una minore precisione ed efficienza della descrizione hardware in questione rispetto a quella che si otterrebbe mediante linguaggi HDL. Pertanto, la sintesi ad alto livello dei sistemi digitali risulta essere una congiunzione tra il mondo hardware e quello software. In particolare, tale approccio comporta i seguenti **vantaggi**:

- **Maggiore produttività per i progettisti hardware**
  I progettisti di hardware possono lavorare a un livello di astrazione superiore mentre creano hardware ad alte prestazioni.
- **Migliori prestazioni del sistema per i progettisti di software**
  Gli sviluppatori di software possono accelerare le parti computazionalmente intensive dei loro algoritmi su un nuovo target di compilazione, cioè l'FPGA.

Inoltre, l'utilizzo di una metodologia di progettazione di sintesi di alto livello consente di:

- **Sviluppare algoritmi tramite linguaggi ad alto livello come C e C++**
  Lavorare a un livello astratto dai dettagli di implementazione, che consumano tempo di sviluppo.
- **Verificare il codice mediante compilatori ad alto livello**
  Convalidare la correttezza funzionale del progetto in modo più rapido rispetto ai linguaggi di descrizione hardware tradizionali come VHDL e Verilog.
- **Controllare il processo di sintesi C attraverso le direttive di ottimizzazione**
  Creare implementazioni hardware specifiche ad alte prestazioni.
- **Creare più implementazioni dal codice sorgente C utilizzando le direttive di ottimizzazione**
  Esplorare diversi design per lo stesso progetto permette di trovare implementazioni più efficienti e ottimali.
- **Sfruttare tutti i vantaggi di portabilità e ri-usabilità del codice ad alto livello**
  Utilizzare tutte le metodologie ad alto livello basate sull'ingegneria del software e, quindi, risparmiare tempo e costi di progettazione software ulteriori.

Per quanto riguarda l'**HLS Design Flow**, questo si articola nei seguenti passi:

1. Compilare, eseguire (simulare) ed eseguire il debug del codice C o C++.
2. Sintetizzare il codice C in un'implementazione RTL, eventualmente utilizzando direttive di ottimizzazione (pragma).
3. Generare e analizzare i report corrispondenti
4. Packaging dell'implementazione RTL per effettuare l'esportazione dell'IP corrispondente.

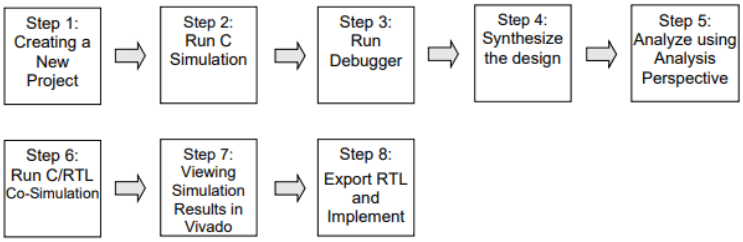


Figure 1: HLS Design Flow

2 Tasks to be performed

Prendendo come riferimento la struttura del filtro FIR, citato precedentemente, e considerando il tool di sintesi ad alto livello per sistemi digitali, fornito da Xilinx® Vivado®, cercare di ottimizzare secondo le caratteristiche principali il circuito in questione. In particolare, utilizzare diverse tecniche di progettazione ad alto livello e sfruttare, se possibile, le direttive di ottimizzazazione (pragma) per ottenere valori ottimali in termini di latenza, dissipazione di potenza e utilizzazione delle risorse. Fondamentalmente, implementare le seguenti tecniche nella maniera opportuna:

- **Unoptimized Solution**
  Rappresenta la soluzione iniziale (non ottimizzata) del filtro FIR in questione.
- **Operation Chaining Solution**
  Rappresenta una soluzione basata sul cambiamento del periodo del clock.
- **Code Hoisting Solution**
  Rappresenta una soluzione basata su un'alternativa gestione delle condizioni di controllo.
- **Loop Fission Solution**
  Rappresenta una soluzione basata sulla scissione di cicli di operazioni complesse in più loop semplici.
- **Loop Unrolling Solution**
  Rappresenta una soluzione basata sul parallelismo delle operazioni.
- **Loop Pipelining Solution**
  Rappresenta una soluzione basata sulla scissione di operazioni combinatorie complesse in più procedure semplici.
- **Bitwidth Optimization Solution**
  Rappresenta una soluzione basata sull'ottimizzazione del numero di bit utilizzati per ogni tipo di dato.
- **AXI Solution**
  Rappresenta una soluzione basata sull'utilizzo dell'interfaccia AXI-Stream.

3 Definitions

Qui di seguito vengono riportate le MACRO e le intestazioni dei metodi corrispondenti alle soluzioni implementate per il filtro FIR in questione. In particolare, ogni definizione presenta la documentazione associata.

```
#include "ap_int.h"
#include <ap_axi_sdata.h>
#include <hls_stream.h>
#include <stdio.h>



#ifndef DEFINITIONS_H


/**
 * Represents FIR filter size
 * @param SIZE FIR filter size
 */
#define SIZE 11

/**
 * Represents the data type used to store the coefficients of the FIR filter
 * @param coeffsType data type for FIR filter coefficients
 */
typedef int coeffsType;

/**
 * Represents the data type used to store the filter input samples
 * @param samplesType data type for FIR filter input samples
 */
typedef int samplesType;

/**
 * Represents the data type used to store the intermediate accumulator when calculating the
    filter output.
 * @param accType data type for FIR filter intermediate accumulator
 */
typedef int accType;

/**
 * Unoptimized Design.
 * @param inputFilter
 * @param outputFilter
 */
void firConvolutionUnoptimized(samplesType inputFilter, samplesType* outputFilter);

/**
 * Operation Chaining Design.
 * @param inputFilter
 * @param outputFilter
 */
void firConvolutionOperationChaining(samplesType inputFilter, samplesType* outputFilter);

/**
 * Code Hoisting Design.
 * @param inputFilter
 * @param outputFilter
 */
void firConvolutionCodeHoisting(samplesType inputFilter, samplesType* outputFilter);

/**
 * Loop Fission Design.
 * @param inputFilter
 * @param outputFilter
 */
void firConvolutionLoopFission(samplesType inputFilter, samplesType* outputFilter);

```

```
63 /**
64  * Loop Unrolling Factor=2 Design.
65  * @param inputFilter
66  * @param outputFilter
67  */
68 void firConvolutionLoopUnrollingFactor2(samplesType inputFilter, samplesType* outputFilter);
69
70 /**
71  * Loop Unrolling Factor=2 with Unrolling Pragma Design.
72  * @param inputFilter
73  * @param outputFilter
74  */
75 void firConvolutionLoopUnrollingFactor2Pragma(samplesType inputFilter, samplesType*
        outputFilter);
76
77 /**
78  * Loop Unrolling Factor=2 with Unrolling Pragma and Partitioning Pragma Design.
79  * @param inputFilter
80  * @param outputFilter
81  */
82 void firConvolutionLoopUnrollingFactor2PP(samplesType inputFilter, samplesType* outputFilter
        );
83
84 /**
85  * Loop Unrolling Factor=4 Design.
86  * @param inputFilter
87  * @param outputFilter
88  */
89 void firConvolutionLoopUnrollingFactor4(samplesType inputFilter, samplesType* outputFilter);
90
91 /**
92  * Loop Unrolling Factor=4 with Unrolling Pragma Design.
93  * @param inputFilter
94  * @param outputFilter
95  */
96 void firConvolutionLoopUnrollingFactor4Pragma(samplesType inputFilter, samplesType*
        outputFilter);
97
98 /**
99  * Loop Unrolling Factor=4 with Unrolling Pragma and Partitioning Pragma Design.
100  * @param inputFilter
101  * @param outputFilter
102  */
103 void firConvolutionLoopUnrollingFactor4PP(samplesType inputFilter, samplesType* outputFilter
        );
104
105 /**
106  * Loop Pipelining Design.
107  * @param inputFilter
108  * @param outputFilter
109  */
110 void firConvolutionLoopPipelining(samplesType inputFilter, samplesType* outputFilter);
111
112 /**
113  * Bitwidth Optimization Design.
114  * @param inputFilter
115  * @param outputFilter
116  */
117 void firConvolutionBitwidthOptimization(ap_int<8> inputFilter, ap_int<18+SIZE>* outputFilter
        );
118
119 /**
120  * Represents the AXI type for AXI-stream interface.
121  * @param AXI_TYPE AXI type
122  */
123 typedef ap_axiu<32, 2, 5, 6> AXI_TYPE;
124
125 /**
```

```
126  * AXI Design.
127  * @param inputFilter
128  * @param outputFilter
129  */
130 void firConvolutionAXI(hls::stream<AXI_TYPE> &inputStreamFilter, hls::stream<AXI_TYPE> &
        outputStreamFilter);
131
132
133 #endif
```

Bisogna, inoltre, specificare che ogni solution (tranne per alcune come verrà appositamente specificato) è stata progettata inizialmente in Xilinx® Vivado® High-Level Synthesis (HLS) e poi, successivamente, dopo aver esportato l'IP, è stata prevista sintesi e implementazione in Xilinx® Vivado®.
Oltre a ciò, si specifica che per ogni implementazione, effettuata in Xilinx® Vivado®, è stato previsto un constraint di clock associato e corrispondente alla solution progettata in Xilinx® Vivado® High-Level Synthesis (HLS) e, in aggiunta, ogni design implementato è stato simulato considerando input random così da ottenere una dati di potenza accurati mediante generazione di file *.saif*.

4 C Simulations

Qui di seguito viene riportato il file testbench per la *C Simulation* in HLS e il corrispondente output ottenuto. Bisogna precisare che ogni risultato ottenuto è stato verificato dal punto di vista numerico.

```
#include "definitions.h"
#include <stdio.h>
#include "ap_int.h"
#include <iostream>


void firConvolutionUnoptimized(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionOperationChaining(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionLoopFission(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionCodeHoisting(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionLoopUnrollingFactor2(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionLoopUnrollingFactor2Pragma(samplesType inputFilter, samplesType*
    outputFilter);
void firConvolutionLoopUnrollingFactor2PP(samplesType inputFilter, samplesType* outputFilter
    );
void firConvolutionLoopUnrollingFactor4(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionLoopUnrollingFactor4Pragma(samplesType inputFilter, samplesType*
    outputFilter);
void firConvolutionLoopUnrollingFactor4PP(samplesType inputFilter, samplesType* outputFilter
    );
void firConvolutionLoopPipelining(samplesType inputFilter, samplesType* outputFilter);
void firConvolutionBitwidthOptimization(ap_int<8> inputFilter, ap_int<18+SIZE>* outputFilter
    );
void firConvolutionAXI(hls::stream<AXI_TYPE> &inputStreamFilter, hls::stream<AXI_TYPE> &
    outputStreamFilter);


int main() {

    samplesType inputFilter[SIZE] = {1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1};
    samplesType outputFilter;

    printf("\n*******firConvolutionUnoptimized*******\n");
    printf("%-20s%-20s%-20s\n", "iteration", "inputFilter[i]", "outputFilter[i]");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionUnoptimized( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionOperationChaining*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionOperationChaining( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionCodeHoisting*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionCodeHoisting( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopFission*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopFission( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopUnrollingFactor2*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor2( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

```

```
    printf("\n*******firConvolutionLoopUnrollingFactor2Pragma*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor2Pragma( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopUnrollingFactor2PragmaPartitioning*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor2PP( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopUnrollingFactor4*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor4( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopUnrollingFactor4Pragma*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor4Pragma( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopUnrollingFactor4PragmaPartitioning*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopUnrollingFactor4PP( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    printf("\n*******firConvolutionLoopPipelining*******\n");
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionLoopPipelining( inputFilter[i], &outputFilter );
        printf("%-20d%-20d%-20d\n", i, inputFilter[i], outputFilter);
    }

    ap_int<8> inputFilterAp[SIZE] = {42, 7, 58, 120, -64, 15, -89, -75, 43, 43, -14};
    ap_int<18+SIZE> outputFilterAp;
    std::cout << "\n*******firConvolutionBitwidthOptimization*******\n";
    for( int i=0; i<SIZE; ++i ) {
        firConvolutionBitwidthOptimization( inputFilterAp[i], &outputFilterAp );
        std::cout << std::left << std::setw(20) << i
                  << std::left << std::setw(20) << inputFilterAp[i]
                  << std::left << std::setw(20) << outputFilterAp << "\n";
    }

    hls::stream<AXI_TYPE> inputStreamFilter;
    hls::stream<AXI_TYPE> outputStreamFilter;
    AXI_TYPE inFilter, outFilter;
    for (int i=0; i<SIZE; i++) {
        inFilter.data=inputFilter[i];
        inFilter.strb = -1;
        inFilter.keep = 15;
        inFilter.user = 0;
        inFilter.last = (i==SIZE-1) ? 1 : 0;
        inFilter.id = 0;
        inFilter.dest = 0;
        inputStreamFilter << inFilter;
    }
    printf("\n*******firConvolutionAXI*******\n");
    firConvolutionAXI( inputStreamFilter, outputStreamFilter );
    for( int i=0; i<SIZE; ++i ) {
        outFilter = outputStreamFilter.read();
        printf("%-20d%-20d%-20d\n", i, inFilter.data.to_int(), outFilter.data.to_int());
    }

    return 0;

```

```
}
```

```
INFO: [SIM 2] *************** CSIM start ***************
INFO: [SIM 4] CSIM will launch GCC as the compiler.
   Compiling ../../../fir_tb.cpp in debug mode
   Compiling ../../../fir_axi.cpp in debug mode
   Compiling ../../../fir_bitwidth_optimization.cpp in debug mode
   Compiling ../../../fir_code_hoisting.cpp in debug mode
   Compiling ../../../fir_loop_fission.cpp in debug mode
   Compiling ../../../fir_loop_pipelining.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor2.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor2_pp.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor2_pragma.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor4.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor4_pp.cpp in debug mode
   Compiling ../../../fir_loop_unrolling_factor4_pragma.cpp in debug mode
   Compiling ../../../fir_operation_chaining.cpp in debug mode
   Compiling ../../../fir_unoptimized.cpp in debug mode
   Generating csim.exe

*******firConvolutionUnoptimized*******
iteration           inputFilter[i]      outputFilter[i]
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050

*******firConvolutionOperationChaining*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050

*******firConvolutionCodeHoisting*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050

*******firConvolutionLoopFission*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
```

```
7                    1                    1088
8                    1                    997
9                    1                    997
10                   1                    1050

*******firConvolutionLoopUnrollingFactor2*******
0                    1                    53
1                    1                    53
2                    1                    -38
3                    1                    -38
4                    1                    275
5                    1                    775
6                    1                    1088
7                    1                    1088
8                    1                    997
9                    1                    997
10                   1                    1050

*******firConvolutionLoopUnrollingFactor2Pragma*******
0                    1                    53
1                    1                    53
2                    1                    -38
3                    1                    -38
4                    1                    275
5                    1                    775
6                    1                    1088
7                    1                    1088
8                    1                    997
9                    1                    997
10                   1                    1050

*******firConvolutionLoopUnrollingFactor2PragmaPartitioning*******
0                    1                    53
1                    1                    53
2                    1                    -38
3                    1                    -38
4                    1                    275
5                    1                    775
6                    1                    1088
7                    1                    1088
8                    1                    997
9                    1                    997
10                   1                    1050

*******firConvolutionLoopUnrollingFactor4*******
0                    1                    53
1                    1                    53
2                    1                    -38
3                    1                    -38
4                    1                    275
5                    1                    775
6                    1                    1088
7                    1                    1088
8                    1                    997
9                    1                    997
10                   1                    1050

*******firConvolutionLoopUnrollingFactor4Pragma*******
0                    1                    53
1                    1                    53
2                    1                    -38
3                    1                    -38
4                    1                    275
5                    1                    775
6                    1                    1088
7                    1                    1088
8                    1                    997
9                    1                    997
```

```
10                  1                   1050

*******firConvolutionLoopUnrollingFactor4PragmaPartitioning*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050

*******firConvolutionLoopPipelining*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050

*******firConvolutionBitwidthOptimization*******
0                   42                  2226
1                   7                   371
2                   58                  -748
3                   120                 5723
4                   -64                 4476
5                   15                  13066
6                   -89                 35907
7                   -75                 63411
8                   43                  64678
9                   43                  18722
10                  -14                 -48096

*******firConvolutionAXI*******
0                   1                   53
1                   1                   53
2                   1                   -38
3                   1                   -38
4                   1                   275
5                   1                   775
6                   1                   1088
7                   1                   1088
8                   1                   997
9                   1                   997
10                  1                   1050
INFO: [SIM 1] CSim done with 0 errors.
INFO: [SIM 3] *************** CSIM finish ***************
```

5 Solutions

Di seguito verranno illustrate e analizzate diverse implementazioni del filtro FIR. In particolare, verranno mostrati i report generati da HLS e Vivado così da effettuare ulteriori considerazioni a riguardo. Bisogna precisare che la prima soluzione presentata, cioè quella iniziale non ottimizzata, verrà discussa approfonditamente così da affrontare e spiegare i parametri di confronto che verranno utilizzati nelle successive sezioni.

5.1 Unoptimized Solution

Qui di seguito viene riportato il codice relativo alla soluzione iniziale (non ottimizzata) del fitro FIR in questione.

```
#include "definitions.h"

void firConvolutionUnoptimized(samplesType inputFilter, samplesType* outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loop: for( i=SIZE-1; i>=0; --i ) {
        if( i==0 ) {
            accumulator += inputFilter * coefficientsFilter[0];
            shiftRegister[0] = inputFilter;
        } else {
            shiftRegister[i] = shiftRegister[i-1];
            accumulator += shiftRegister[i] * coefficientsFilter[i];
        }
    }
    *outputFilter = accumulator;
}
```

In particolare, si può notare come l'input, l'output e i coefficienti utilizzati risultano essere definiti rispettivamente tramite i tipi *samplesType*, *samplesType* e *coeffsType*, cioè tutti e tre definiti come tipi di dato *int* come descritto in *definitions.h*. Bisogna specificare che l'operazione di convoluzione è stata ottenuta mediante il loop qui sopra definito. Tale operazione è data dallo shifting dei valori di input al filtro e, successivamente, dall'accumulo. Quest'ultimo è ottenuto mediante la somma, tra la corrente calcolata e la precedente accumulata, e il prodotto, tra il valore shiftato nella precedente iterazione e il coefficiente *i-esimo* corrispondente.
Infine, il valore accumulato nella variabile *accumulator* viene assegnato all'output atteso dalla procedura.
Effettuando la sintesi è possibile evidenziare il seguente report:

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.510	1.25

Table 1: HLS Unoptimized Solution Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
23	45	23	45

Table 2: HLS Unoptimized Solution Latency Summary (clock cycles)

Si può notare come il periodo di clock previsto è pari a 10 ns, come impostato durante la creazione della solution in questione, mentre quello stimato è minore. Questo è dovuto al fatto che il tool stima un'incertezza (assimilabile ad un margine) che permette di calcolare il periodo di clock effettivo utilizzato dal processo di sintesi.

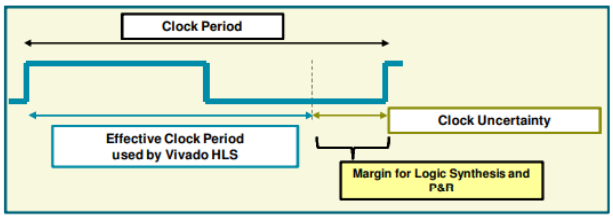


Figure 2: HLS Clock Period from Synthesis Report

In particolare, il periodo di clock effettivo, calcolato dal tool, non deve essere considerato come parametro di progettazione ma come un margine di sicurezza previsto da HLS per il successivo place and route. Nel caso in cui si volesse operare sul periodo di clock della solution come design parameter, si dovrebbe impostare un periodo di clock differente durante la creazione della solution e poi far calcolare automaticamente l'incertezza ed il periodo del clock effettivo dallo stesso tool.

Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
Name	min	max	min	max	achieved	target	
- loop	22	44	2	4	-	-	11

Table 3: HLS Unoptimized Solution Latency Loops Summary

Considerando, invece, il loop descritto nel file sorgente, si può notare come esso presenti 11 iterazioni (corrispondenti alla capienza di processing del filtro corrispondente) e una iteration latency (IL), cioè latenza per ogni iterazione, compresa tra 2 e 4 cicli di clock. Si deduce che la latenza totale del loop in questione sia compresa tra 22 e 44 cicli di clock.

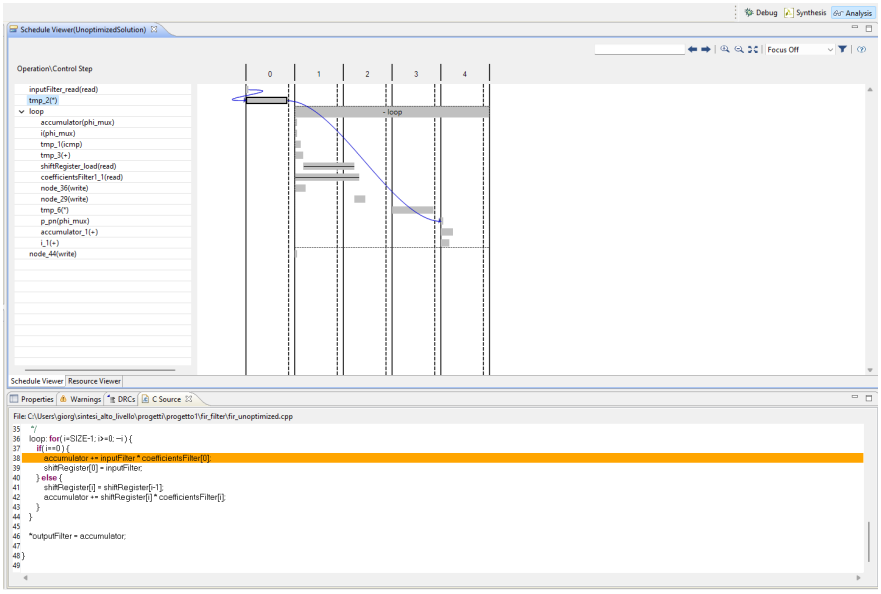


Figure 3: HLS Unoptimized Solution If Statement Analysis

In particolare, si può notare come il ciclo di clock in più, previsto nella latenza totale dell'architettura, sia dovuto alla verifica, al calcolo e all'assegnazione in corrispondenza dell'iterazione 0 all'interno del loop. È come se l'if all'interno del ciclo for venga trattato al di fuori dello stesso loop. Questo può essere notato facendo riferimento alle operazioni svolte all'interno del loop.

Inoltre, si può notare come le operazioni di lettura previste all'interno del loop siano svolte in parallelo (come mostrato nel primo gruppo di figure allegato), mentre quelle di scrittura schedulate in differente

maniera (come mostrato nel secondo gruppo di figure allegato).

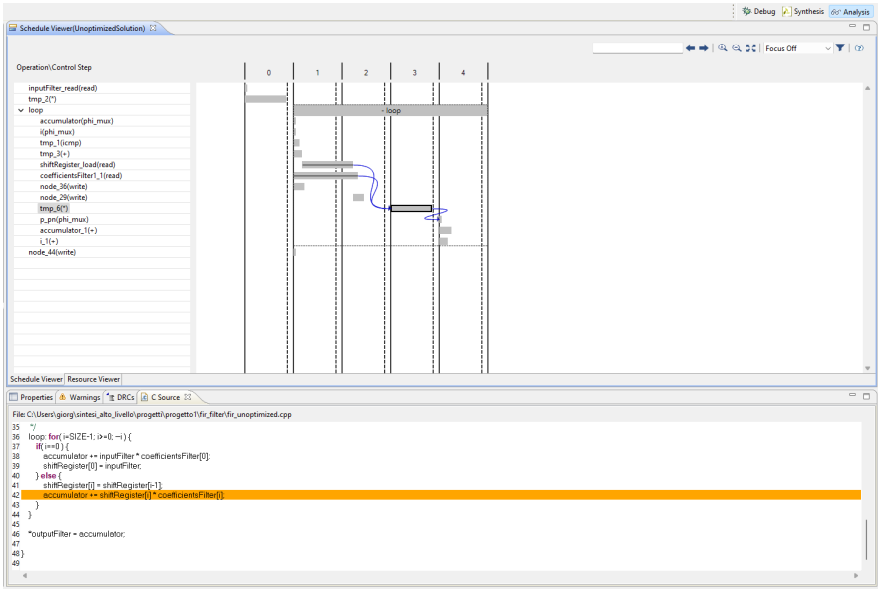


Figure 4: HLS Unoptimized Solution Else Statement Analysis

Infatti, si può notare come la latenza associata alle operazioni previste nelle rimanenti iterazioni del loop venga correlata alla latency associata al ciclo stesso.

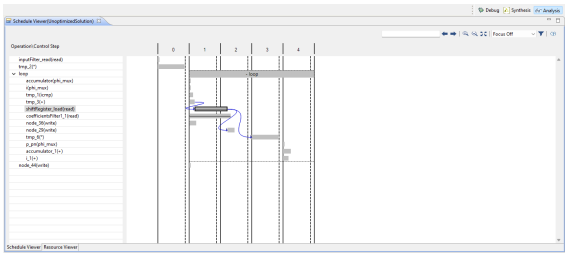


Figure 5: HLS Unoptimized Solution Read Operations Analysis

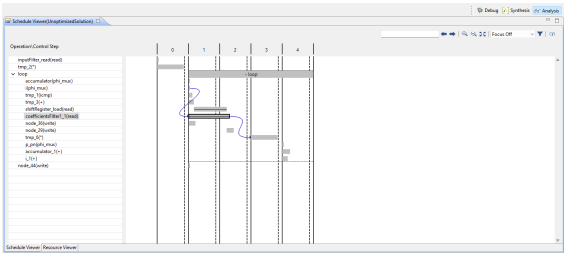


Figure 6: HLS Unoptimized Solution Read Operations Analysis

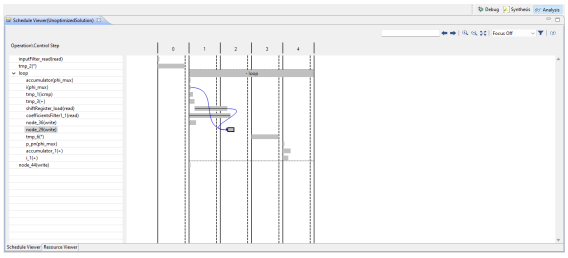


Figure 7: HLS Unoptimized Solution Write Operations Analysis

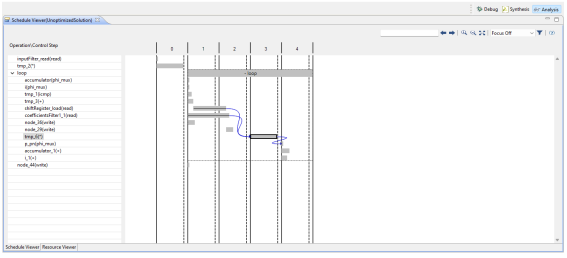


Figure 8: HLS Unoptimized Solution Write Operations Analysis

Qui di seguito, viene allegato l'utilizzazione delle risorse stimata dal processo di sintesi.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	4	0	105
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	120
Register	-	-	213	-
Total	0	4	287	233
Available	280	220	106400	53200
Utilization (%)	0	1	∼0	∼0

Table 4: HLS Unoptimized Solution Utilization Estimates Summary

Successivamente effettuando la C/RTL Cosimulation è possibile evidenziare il seguente report:

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	43	43	44	43	43	44

Table 5: HLS Unoptimized Solution C/RTL Cosimulation Summary

Questa tabella mostra quanti cicli di clock sono necessari affinché venga fornito l'ingresso e l'uscita successivi.

Dopodiché si può procedere con l'Export RTL.

Resource	VHDL
SLICE	81
LUT	275
FF	160
DSP	2
BRAM	0
SRL	0

Table 6: HLS Unoptimized Solution Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	6.410

Table 7: HLS Unoptimized Solution Export RTL Final Timing

Si può notare come il numero di Flip Flop e di DSP sia diminuito rispetto a quello stimato nel report di sintesi. Questo è dovuto al fatto che, l'Export RTL non è una vera e propria implementazione ma si tratta di un processo durante il quale il tool effettua dei miglioramenti.
Successsivamente a tale fase si procede con l'importazione del'IP in Vivado.

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity firConvolutionUnoptimized_top is
    port (
        ap_clk : IN STD_LOGIC;
        ap_rst : IN STD_LOGIC;
        ap_start : IN STD_LOGIC;
        ap_done : OUT STD_LOGIC;
        ap_idle : OUT STD_LOGIC;
        ap_ready : OUT STD_LOGIC;
        inputFilter : IN STD_LOGIC_VECTOR (31 downto 0);
        outputFilter : OUT STD_LOGIC_VECTOR (31 downto 0);
        outputFilter_ap_vld : OUT STD_LOGIC
```

```
    );
end firConvolutionUnoptimized_top;

architecture Behavioral of firConvolutionUnoptimized_top is
    component firConvolutionUnoptimized_0 is
        port (
            ap_clk : IN STD_LOGIC;
            ap_rst : IN STD_LOGIC;
            ap_start : IN STD_LOGIC;
            ap_done : OUT STD_LOGIC;
            ap_idle : OUT STD_LOGIC;
            ap_ready : OUT STD_LOGIC;
            inputFilter : IN STD_LOGIC_VECTOR (31 downto 0);
            outputFilter : OUT STD_LOGIC_VECTOR (31 downto 0);
            outputFilter_ap_vld : OUT STD_LOGIC
        );
  end component;

    begin
        firConvolutionUnoptimized_IP : firConvolutionUnoptimized_0 port map(
            ap_clk => ap_clk,
            ap_rst => ap_rst,
            ap_start => ap_start,
            ap_done => ap_done,
            ap_idle => ap_idle,
            ap_ready => ap_ready,
            inputFilter => inputFilter,
            outputFilter => outputFilter,
            outputFilter_ap_vld => outputFilter_ap_vld
    );
end Behavioral;
```

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity firConvolutionUnoptimized_tb is
end firConvolutionUnoptimized_tb;

architecture Behavioral of firConvolutionUnoptimized_tb is
    component firConvolutionUnoptimized_top is
        port (
            ap_clk : IN STD_LOGIC;
            ap_rst : IN STD_LOGIC;
            ap_start : IN STD_LOGIC;
            ap_done : OUT STD_LOGIC;
            ap_idle : OUT STD_LOGIC;
            ap_ready : OUT STD_LOGIC;
            inputFilter : IN STD_LOGIC_VECTOR (31 downto 0);
            outputFilter : OUT STD_LOGIC_VECTOR (31 downto 0);
            outputFilter_ap_vld : OUT STD_LOGIC
        );
  end component;

    constant clk_period : Time := 10 ns;
    signal ap_clk :  STD_LOGIC := '0';
    signal ap_rst :  STD_LOGIC;
    signal ap_start :  STD_LOGIC;
    signal ap_done :  STD_LOGIC;
    signal ap_idle :  STD_LOGIC;
    signal ap_ready :  STD_LOGIC;
    signal inputFilter :  STD_LOGIC_VECTOR (31 DOWNTO 0);
    signal outputFilter :  STD_LOGIC_VECTOR (31 DOWNTO 0);
    signal outputFilter_ap_vld :  STD_LOGIC;
    signal cycles: integer := 44;

    begin
        uut : firConvolutionUnoptimized_top port map(
            ap_clk,
```

```
            ap_rst,
            ap_start,
            ap_done,
            ap_idle,
            ap_ready,
            inputFilter,
            outputFilter,
            outputFilter_ap_vld
    );

    clk_process : process
            begin
                wait for clk_period/2;
                ap_clk <= not ap_clk;
        end process clk_process;

    uut_process : process
            begin
                ap_rst <= '1';
        ap_start <= '0';
        wait for 250ns;
        ap_rst <= '0';
        wait for clk_period;
        ap_start <= '1';
                inputFilter <= "01101100011111000000011010001110";
                wait for cycles*clk_period;
                inputFilter <= "10101100001011101011101101100010";
                wait for cycles*clk_period;
                inputFilter <= "11110000011100010010110001111100";
                wait for cycles*clk_period;
                inputFilter <= "01101101000111100000010100000100";
                wait for cycles*clk_period;
                inputFilter <= "11000100010011111001010011011001";
                wait for cycles*clk_period;
                inputFilter <= "01101001100110010111101010010010";
                wait for cycles*clk_period;
                inputFilter <= "10110101110111110100000101110011";
                wait for cycles*clk_period;
                inputFilter <= "11110011001101000000001100001010";
                wait for cycles*clk_period;
                inputFilter <= "01010001001110100111010101001001";
                wait for cycles*clk_period;
                inputFilter <= "10100010011110110010000010011000";
                wait for cycles*clk_period;
                inputFilter <= "11101000110011101001011101010001";
                wait for cycles*clk_period;
                wait;
    end process uut_process;
end Behavioral;
```

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

In particolare, impostando un constraint di clock pari a 10 ns (come impostato anche in HLS) e considerando un numero di ciclo di clock pari a quelli ottenuti nel report di C/RTL Cosimulation, è possibile eseguire il processo di sintesi e implementazione.

LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
275	32	160	0	2	71	1

Table 8: Vivado Unoptimized Solution Utilization Report [#]

Cycles [#]	**Clock Constraint** [ns]	**WNS** [ns]	**Maximum Clock Frequency** [Mhz]
44	10	3.654	157.5795777

Table 9: Vivado Unoptimized Solution Timing Report

Pertanto, dopo aver effettuato la Post-Implementation Timing Simulation e aver inserito il file *.saif* all'interno dell'Implementation Power Report, è possibile analizzare con dettaglio la potenza dinamica e l'energia per singola operazione associata.

BRAM	**Clock Enable**	**Clocks**	**DSP**	**Logic**	**Set/Reset** [mW]	**Data**
0	0.454227469	1.215484925	0.335011806	0.921480532	3.57E-03	1.007059589

Table 10: Vivado Unoptimized Solution Dynamic Power Report [mW]

Dynamic Total
3.936829417

Table 11: Vivado Unoptimized Solution Dynamic Power Report [mW]

Energy Single Operation
39.36829417

Table 12: Vivado Unoptimized Solution Energy Single Operation Report [pJ]

Quindi, dopo aver generato i report associati alla soluzione iniziale non ottimizzata, si può procedere con le successive soluzioni che cercheranno di migliorare diversi aspetti dell'architettura appena proposta. In particolare, nelle successive sezioni verranno commentati i risultati in termini di potenza, timing e risorse facendo riferimento alla soluzione iniziale o a quella precedente su cui sarà basata la corrente.

5.2 Operation Chaining Solution

Considerando, ora, la stessa architettura software precedentemente esposta, si potrebbe fare un ragionamento riguardo il periodo di clock considerato. In particolare, si potrebbe pensare di utilizzare un periodo minore per analizzare come il tool gestisce la situazione associata. L'idea è quella di diminuire tale periodo senza andare oltre un margine. Questo vuol dire considerare lo slack associato alla soluzione iniziale e ridurlo fin quando è possibile, cioè fin quando i constraint di timing non sono violati. Ciò che dovremmo aspettarci è che il tool riesca a gestire la stessa architettura con un periodo di clock minore, rispetto a quello della soluzione iniziale non ottimizzata, garantendo in alcuni casi delle ottimizzazioni effettuate dallo stesso tool. L'ideale sarebbe ottenere una soluzione che mi permette di avere un giusto trade-off tra latenza (intesa come cicli di clock per ottenere un risultato), potenza e risorse utilizzate.
Qui di seguito verranno presentati i report associati alla sintesi, C/RTL Cosimulation ed Export RTL considerando un range di periodo di clock pari a $[10ns, 3ns]$. In particolare, le soluzioni analizzate non sono state considerate con un periodo del clock minore a 3ns dato che, come verrà mostrato e analizzato successivamente, in corrispondenza di un periodo del clock pari a 3ns, la soluzione hardware implementata all'interno di Vivado presenta un WNS negativo comportando, di conseguenza, la violazione dei constraint di timing.

Solution	**Clock**	**Target**	**Estimated**	**Uncertainty**
clk=10ns	ap_clk	10.00	8.510	1.25
clk=9ns	ap_clk	9.00	6.912	1.12
clk=8ns	ap_clk	8.00	6.912	1
clk=7ns	ap_clk	7.00	5.745	0.88
clk=6ns	ap_clk	6.00	4.644	0.75
clk=5ns	ap_clk	5.00	4.321	0.62
clk=4ns	ap_clk	4.00	3.254	0.5
clk=3ns	ap_clk	3.00	2.552	0.38

Table 13: HLS Operation Chaining Solution Timing Summary (ns)

Solution	**Latency**		**Interval**	
	min	max	min	max
clk=10ns	23	45	23	45
clk=9ns	24	57	24	57
clk=8ns	24	57	24	57
clk=7ns	25	69	25	69
clk=6ns	27	93	27	93
clk=5ns	27	93	27	93
clk=4ns	40	139	40	139
clk=3ns	40	139	40	139

Table 14: HLS Operation Chaining Solution Latency Summary (clock cycles)

Solution	Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
	Name	min	max	min	max	achieved	target	
clk=10ns	- loop	22	44	2	4	-	-	11
clk=9ns	- loop	22	55	2	5	-	-	11
clk=8ns	- loop	22	55	2	5	-	-	11
clk=7ns	- loop	22	66	2	6	-	-	11
clk=6ns	- loop	22	88	2	8	-	-	11
clk=5ns	- loop	22	88	2	8	-	-	11
clk=4ns	- loop	33	132	3	12	-	-	11
clk=3ns	- loop	33	132	3	12	-	-	11

Table 15: HLS Operation Chaining Solution Latency Loops Summary

Si può notare come al diminuire del periodo di clock, la latenza totale e la latenza per ogni iterazione aumentano rispettivamente. Questo è dovuto al fatto che, considerando un periodo di clock sempre minore, il tool cerca di garantire il minor tempo di latenza affinché le operazioni presenti all'interno del loop vengano eseguite correttamente. Pertanto, se il periodo del clock diminuisce, questo si traduce in un aumento dei cicli di clock richiesti per ogni iterazione. Bisogna notare che, in corrispondenza di alcune soluzioni, alcuni parametri assumono valori uguali. In particolare, tale situazione si verifica in corrispondenza delle soluzioni aventi periodo del clock pari a 9ns e 8ns e anche in corrispondenza di 6ns e 5ns e, infine, 4ns e 3ns. Questa peculiarità potrebbe essere sfruttata successivamente, considerando ulteriori design parameters di confronto,

per la scelta di un trade-off opportuno. Dal momento che il tool cerca di garantire il soddisfacimento di tali operazioni complesse con un minore periodo di clock, quello che ci si aspetta è una utilizzazione delle risorse stimate e, in particolare, un aumento di quest'ultime al diminuire del periodo di clock.

Solution	**BRAM**	**DSP**	**FF**	**LUT**
clk=10ns	0	4	287	233
clk=9ns	0	4	619	301
clk=8ns	0	4	619	301
clk=7ns	0	4	623	305
clk=6ns	0	4	725	221
clk=5ns	0	4	725	221
clk=4ns	0	4	832	286
clk=3ns	0	4	832	286

Table 16: HLS Operation Chaining Solution Utilization Estimates [#]

Si evidenzia quanto citato precedentemente e, inoltre, si può notare come, anche in questo caso, in corrispondenza di alcune soluzioni hardware, l'utilizzazione delle risorse risulta essere la medesima. In particolare, è possibile già effettuare una riflessione riguardo i risultati ottenuti dopo il processo di sintesi. Si può notare come, in corrispondenza delle soluzioni hardware con clk=9ns e clk=8ns si ha lo stessa latency, iteration latency e utilizzazione delle risorse. Ma soprattutto, facendo riferimento alla soluzione con periodo di clock pari a 7ns, la latency e l'iteration latency aumentano di un fattore pari a 11 (rispetto alle altre soluzioni dove i valori sono di gran lunga maggiori) e l'utilizzazione di FF e LUT (dal momento che il numero di BRAM e DSP utilizzati è il medesimo) risulta essere aumentata rispettivamente del 0.6% e del 1.32%. Pertanto, a fronte del processo di sintesi, la soluzione hardware avente periodo di clock pari a 7ns si potrebbe già considerare come candidata al trade-off sopra citato.

Solution	RTL	Status	**Latency**			**Interval**		
			min	avg	max	min	avg	max
clk=10ns	VHDL	Pass	43	43	44	43	43	44
clk=9ns	VHDL	Pass	54	54	55	54	54	55
clk=8ns	VHDL	Pass	54	54	55	54	54	55
clk=7ns	VHDL	Pass	65	65	66	65	65	66
clk=6ns	VHDL	Pass	87	87	88	87	87	88
clk=5ns	VHDL	Pass	87	87	88	87	87	88
clk=4ns	VHDL	Pass	130	130	131	130	130	131
clk=4ns	VHDL	Pass	130	130	131	130	130	131
clk=3ns	VHDL	Pass	130	130	131	130	130	131

Table 17: HLS Operation Chaining Solution C/RTL Cosimulation Report

Solution	SLICE	LUT	FF	DSP	BRAM	CP required	CP achieved post-synthesis	CP achieved post-implementation
clk=10ns	81	275	160	2	0	10	5.745	6.41
clk=9ns	88	276	226	2	0	9	5.745	6.059
clk=8ns	89	276	226	2	0	8	5.745	6.034
clk=7ns	75	186	222	2	0	7	5.745	5.692
clk=6ns	76	186	241	2	0	6	4.823	4.364
clk=5ns	96	186	258	2	0	5	4.823	4.282
clk=4ns	114	188	376	2	0	4	3.667	3.724
clk=3ns	107	192	377	4	0	3	3.667	3.627

Table 18: HLS Operation Chaining Solution Export RTL Report

Si può notare come la soluzione ottimale precedentemente proposta risulta essere la migliore tra quelle citate. Infatti, è possibile evidenziare che rispetto alle altre soluzioni presenta il minor numero di risorse stimate dopo aver effettuato l'Export RTL in HLS.
Pertanto, si può affermare che il tool, in corrispondenza di un periodo di clock pari a 7ns, è riuscito ad effettuare delle ottimizzazioni che hanno permesso di ottenere un minimo aumento della latenza e una diminuzione delle risorse utilizzate.
Quindi, considerando constraint di clock differenti per ogni IP importato in Vivado, si procede con la sintesi, l'implementazione, la Post-Implementation Timing Simulation e la generazione del file .saif per ogni soluzione corrispondente.

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

```
create_clock -period 9.000 -name myclk -waveform {0.000 4.500} [get_ports ap_clk]
```

```
create_clock -period 8.000 -name myclk -waveform {0.000 4.000} [get_ports ap_clk]
```

```
create_clock -period 7.000 -name myclk -waveform {0.000 3.500} [get_ports ap_clk]
```

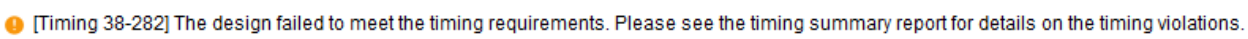
```
create_clock -period 6.000 -name myclk -waveform {0.000 3.000} [get_ports ap_clk]
```

```
create_clock -period 5.000 -name myclk -waveform {0.000 2.500} [get_ports ap_clk]
```

```
create_clock -period 4.000 -name myclk -waveform {0.000 2.000} [get_ports ap_clk]
```

```
create_clock -period 3.000 -name myclk -waveform {0.000 1.500} [get_ports ap_clk]
```

In particolare, qui di seguito verranno mostrati e analizzati soltanto i report corrispondenti alle soluzioni avente periodo di clock compreso nell'intervallo $[10ns, 4ns]$. Infatti, è possibile notare come, in corrrispondenza del periodo di clock pari a 3ns, i constraint di timing siano violati.



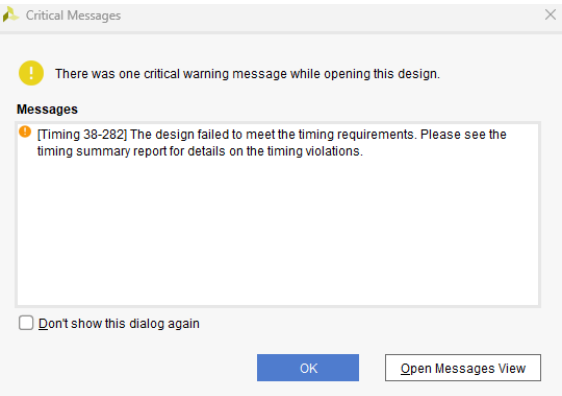


Figure 9: Vivado Operation Chaining Solution clk=3ns Timing Constraint Violated

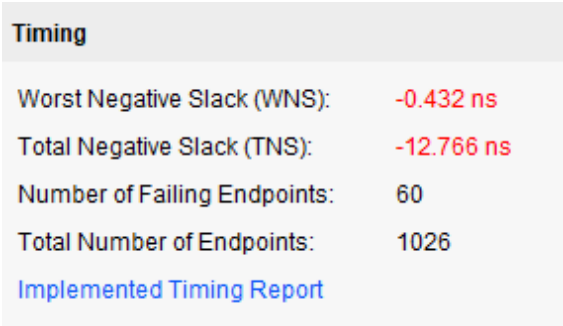


Figure 10: Vivado Operation Chaining Solution clk=3ns Timing Constraint Violated

Pertanto, si procede con l'analisi dei report delle soluzioni hardware implementate in Vivado aventi periodo di clock coompreso nell'intervallo $[10ns, 4ns]$.

Solution	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
clk=10ns	275	32	160	0	2	71	1
clk=9ns	276	32	226	0	2	71	1
clk=8ns	276	32	226	0	2	71	1
clk=7ns	186	32	222	0	4	71	1
clk=6ns	186	32	241	0	4	71	1
clk=5ns	186	32	258	0	4	71	1
clk=4ns	188	32	376	0	4	71	1

Table 19: Vivado Operation Chaining Solution Utilization Report [#]

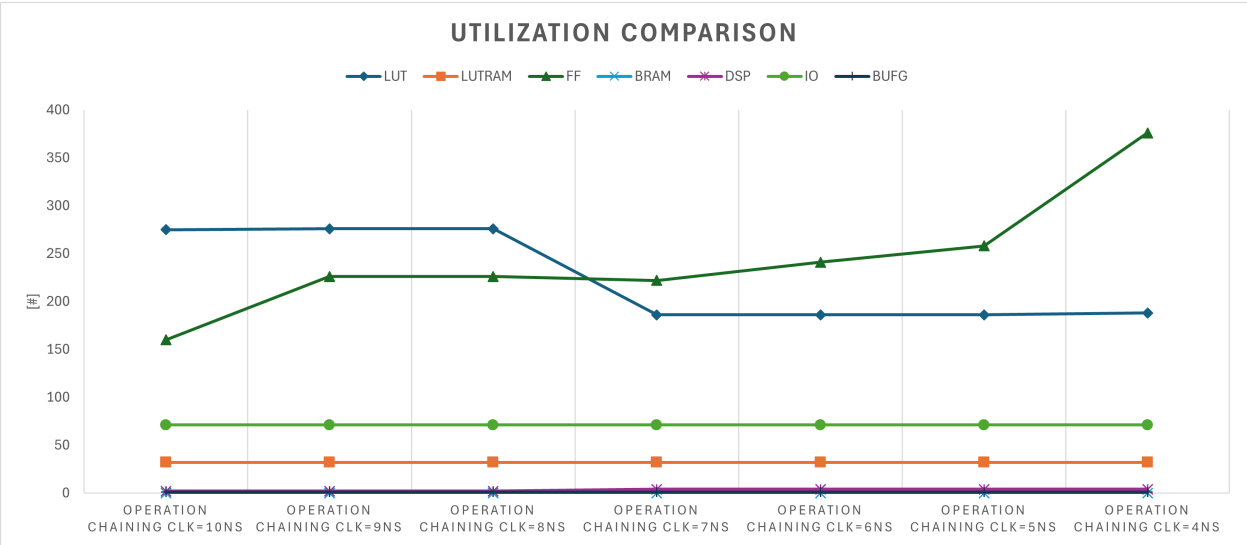


Figure 11: Vivado Operation Chaining Utilization Plot

Solution	**Cycles** [#]	**Clock Constraint** [ns]	**WNS** [ns]	**Maximum Clock Frequency** [Mhz]
clk=10ns	44	10	3.654	157.5795777
clk=9ns	55	9	3.06	168.3501684
clk=8ns	55	8	2.277	174.7335314
clk=7ns	66	7	1.33	176.366843
clk=6ns	88	6	1.573	225.8866049
clk=5ns	88	5	0.374	216.1694769
clk=4ns	131	4	0.454	282.0078962

Table 20: Vivado Operation Chaining Solution Timing Report

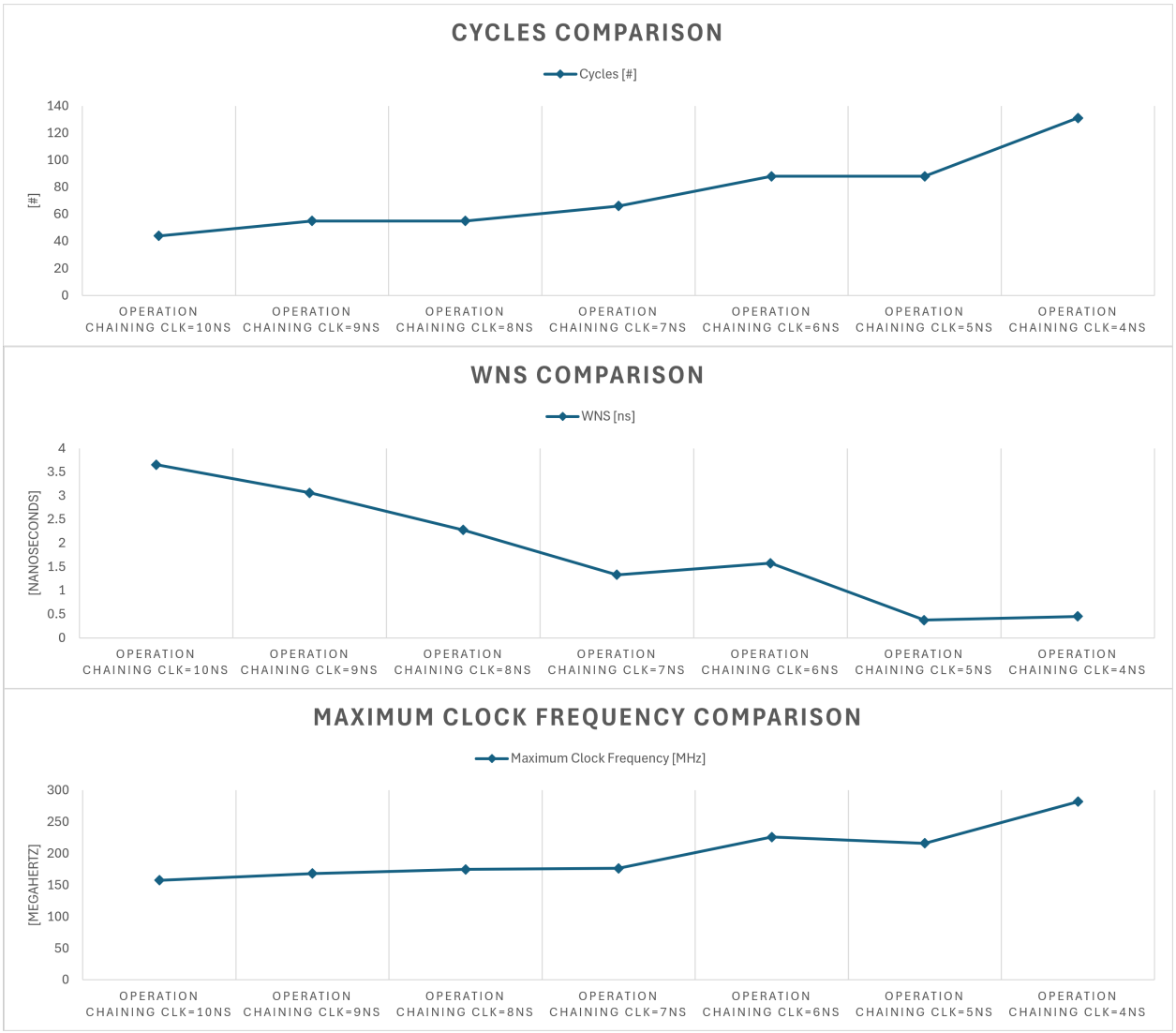


Figure 12: Vivado Operation Chaining Timing Plot

Si può notare come in corrispondenza dell'architettura hardware a cui corrisponde un periodo di clock pari a 7ns si ha l'utilizzazione delle risorse minore a meno del numero di LUT e FF che risulta essere aumentato rispettivamente di un'unità e di 66 unità. Questo aumento di risorse può essere giustificato dal momento

che, rispetto alla soluzione iniziale non ottimizzata a 10ns, il periodo del clock risulta essere diminuito e, pertanto, il tool prevede una gestione delle istruzioni in maniera più complessa comportando un'aumento delleee risorse stesse. Tanto è vero che al diminuire del periodo di clock, si ha un aumento delle risorse utilizzate e dei cicli di clock considerati per ottenere un nuovo risultato. In particolare, per quanto riguarda la massima frequenza di clock associata ad ogni soluzione proposta in questa sezione, si può notare come quella a cui corrisponde un periodo di clock pari a 7ns, presenta una maximum clock frequency maggiore rispetto a quelle avente periodo maggiore. Questo aumento lo si può, inoltre, evidenziare nelle successive soluzioni aventi periodo di clock ancora minore. Tanto è vero che il WNS, al diminuire del clock constraint, anch'esso diminuisce.

Solution	**Clock Enable**	**Clocks**	**DSP**	**Logic**	**Set/Reset**	**Data**
clk=10ns	0.455035944	1.215706812	0.343570428	0.925468281	0.00349783	1.014495501
clk=9ns	0.371109403	1.601127908	0.308091694	0.871003722	0.003504661	0.845550094
clk=8ns	0.432465255	1.790320966	0.364705513	1.03614782	0.003386509	1.085355412
clk=7ns	0.4179838	1.991891069	0.305155962	0.881534419	0.002575182	0.757947506
clk=6ns	0.367850007	2.092533279	0.244985771	0.788475969	0.00273619	0.666663051
clk=5ns	0.556948828	3.174162703	0.291668839	0.943421735	0.004532358	0.906153
clk=4ns	0.57213963	3.804846667	0.253423554	0.799402245	0.002476984	0.756326073

Table 21: Vivado Operation Chaining Solution Dynamic Power Report [mW]

Solution	**Dynamic Total**
clk=10ns	3.957774796
clk=9ns	4.000387481
clk=8ns	4.712381475
clk=7ns	4.357087937
clk=6ns	4.163244267
clk=5ns	5.876887463
clk=4ns	6.188615153

Table 22: Vivado Operation Chaining Solution Dynamic Power Report [mW]

Solution	**Energy Single Operation**
clk=10ns	39.57774796
clk=9ns	36.00348733
clk=8ns	37.6990518
clk=7ns	30.49961556
clk=6ns	24.9794656
clk=5ns	29.38443731
clk=4ns	24.75446061

Table 23: Vivado Operation Chaining Solution Energy Single Operation Report [pJ]

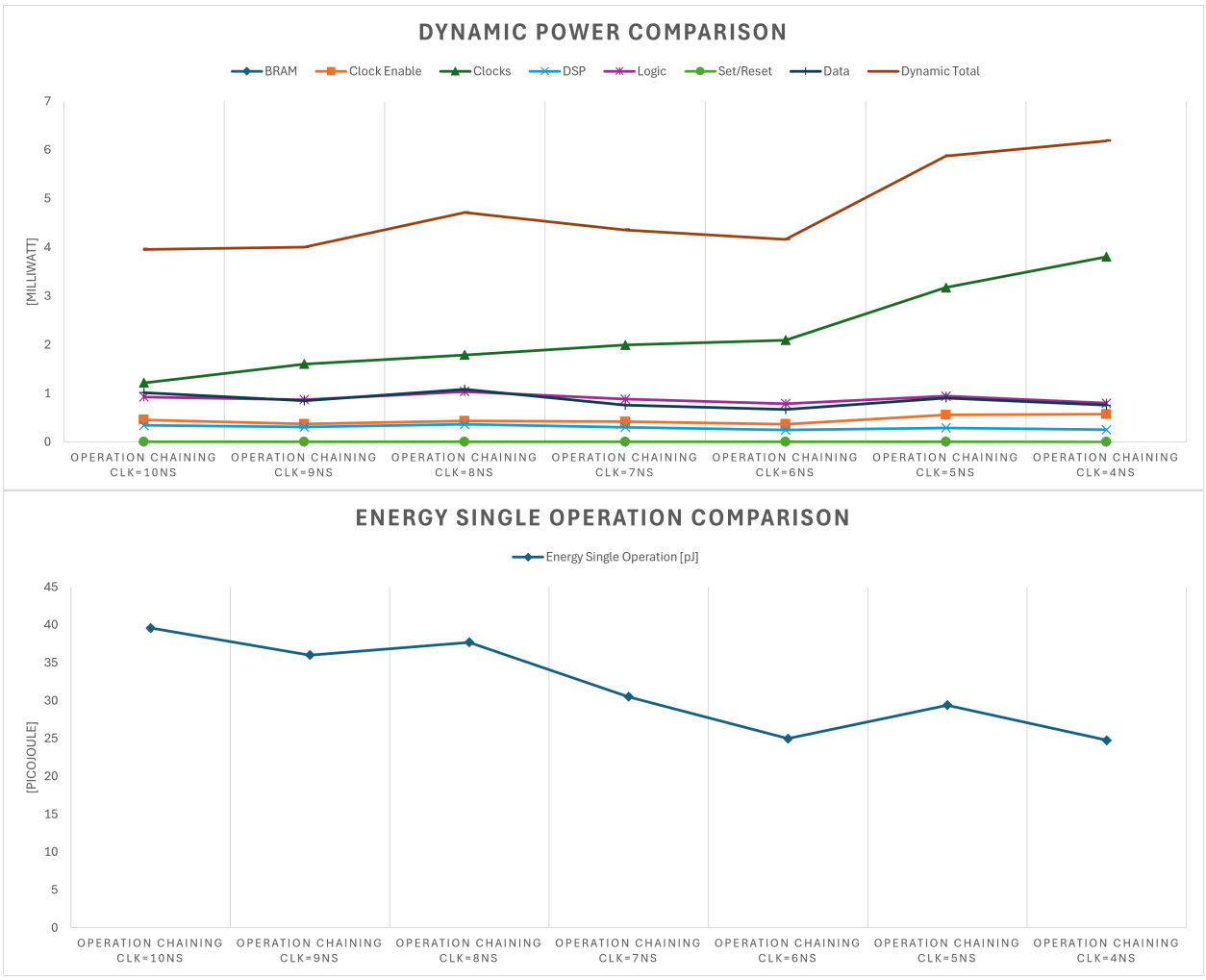


Figure 13: Vivado Operation Chaining Power Plot

Si può notare come la potenza dinamica aumenti al diminuire del periodo di clock considerato. Questo si evince dal fatto che l'utilizzazione delle risorse (in particolar modo per quello che riguarda i FF) aumenta al diminuire del constraint di clock. Tanto è vero che la potenza dinamica associata all'albero di distribuzione del clock (BUFG) e del clock enable sia di gran lunga maggiore rispetto alla soluzione iniziale dove è stato considerato un constraint pari a 10ns. In particolare, questo aumento di potenza dinamica può essere associato al maggior numero di Flip Flop utilizzato nelle soluzioni aventi periodo di clock sempre minore. Pertanto, considerando i report di utilizzazione delle risorse, di timing, di potenza dinamica ed energia per singola operazione, si può evincere che la soluzione hardware che permette un trade-off ottimale è quella ottenuta in corrispondenza del periodo di clock pari a 7ns.

5.3 Code Hoisting Solution

Considerando, ora, la stessa architettura software precedentemente utilizzata, si potrebbe mettere il luce un possibile problema: impiego di risorse aggiuntive a causa di istruzioni logiche condizionali. In particolare, questo problema è dovuto al fatto che, all'interno del ciclo for, è prevista l'istruzione condizionale $if(i == 0)$ che comporta l'utilizzo di risorse hardware addizionali. Bisogna precisare che, istruzioni logiche quali if, *else* e *then* rendono i loop inefficienti limitando le performance dal punto di vista hardware. Fondamentalmente, l'istruzione condizionale $if(i == 0)$ a livello hardware si traduce nel leggere il valore dell'indice i in una certa struttura dati ed effettuare il confronto, tramite un comparatore, tra tale valore appena citato e il valore zero.

```
#include "definitions.h"

void firConvolutionCodeHoisting(samplesType inputFilter, samplesType* outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loop: for( i=SIZE-1; i>0; --i ) {
        shiftRegister[i] = shiftRegister[i-1];
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    accumulator += inputFilter * coefficientsFilter[0];
    shiftRegister[0] = inputFilter;
    *outputFilter = accumulator;
}
```

Pertanto, si potrebbe pensare di eliminare l'if in questione e spostarlo al di fuori del loop. In questo modo, anche l'istruzione condizionale *else*, che ricopriva il resto delle iterazioni, verrà gestita direttamente all'interno del loop avente quest'ultimo un'iterazione in meno dovuto ai motivi appena citati. Pertanto, quello che ci si aspetta è una diminuzione dal punto di vista della latenza totale, dal punto di vista delle risorse e per ciò che riguarda la potenza dinamica associata alla parte di logica dell'architettura hardware in questione.
Pertanto, si mostrano qui di seguito i report, generati da HLS, riguardo la sintesi, la C/RTL Cosimulation e l'Export RTL.

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.510	1.25

Table 24: HLS Code Hoisting Solution Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
42	42	42	42

Table 25: HLS Code Hoisting Solution Latency Summary (clock cycles)

Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
Name	min	max	min	max	achieved	target	
- loop	40	40	4	4	-	-	10

Table 26: HLS Code Hoisting Solution Latency Loops Summary

Si può notare come il numero di iterazioni del loop, in questo caso, sia pari a 10 tale che la latenza totale del ciclo risulta essere pari a 40.

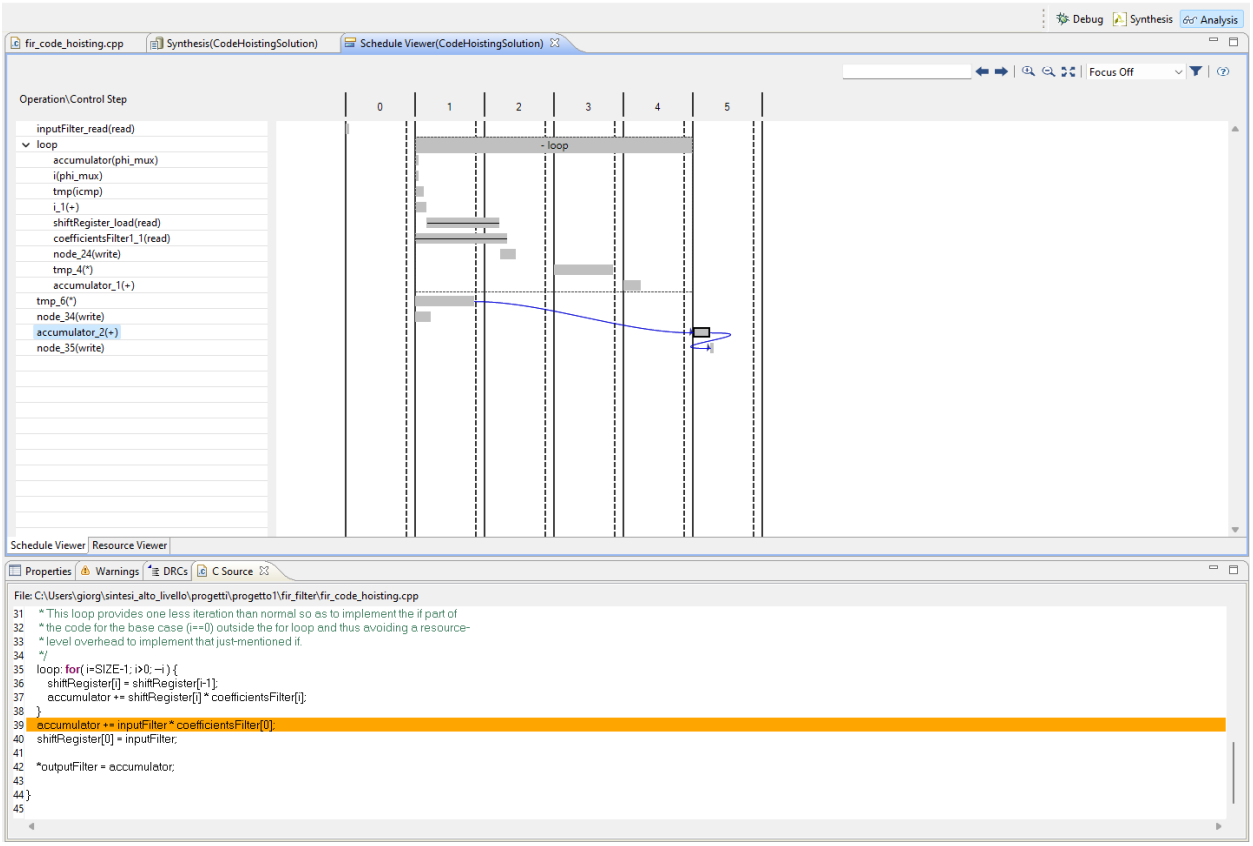


Figure 14: HLS Code Hoisting Solution Trip Count Analysis

In particolare, è opportuno fare un'attenta riflessione riguardo la latenza totale dell'architettura hardware in questione. Il report di sintesi, effettivamente, ha restituito una stima di tale valore pari a 42 dal momento bisogna considera una latenza totale del loop pari a 40, un ciclo di latency ulteriori dovuti alla lettura iniziale dell'input (come avviene per ogni soluzione progettata) e, come ultimo, un altro ciclo finale dovuto alle due operazioni in parallelo effettuate: accumulo associato al coefficiente in posizione 0 e scrittura del dato in uscita. Precedentemente, nella soluzione iniziale non ottimizzata, tale latenza finale non era prevista dal momento che il caso $i == 0$ veniva gestito all'interno del loop mediante istruzione condizionale e la scrittura del dato in uscita veniva effettuata in parallelo all'operazione appena citata.

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
DSP	-	-	-	-
Expression	-	4	0	140
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	92
Register	-	-	156	-
Total	0	4	230	240
Available	280	220	106400	53200
Utilization (%)	0	1	~0	~0

Table 27: HLS Code Hoisting Solution Utilization Estimates Summary

Si può notare come il report di C/RTL Cosimulation evidenzi un ciclo di latenza in meno rispetto alla

soluzione hardware iniziale non ottimizzata. Questo è dovuto al fatto che, come precedentemente citato, il loop presenta un'iterazione in meno tale per cui il numero di cicli di clock, affinché venga processato si abbia un risultato in uscita, è pari a 43 rispetto ai 44 corrispondenti alla soluzione non ottimizzata.

RTL	Status	**Latency**			**Interval**		
		min	avg	max	min	avg	max
VHDL	Pass	42	42	43	42	42	43

Table 28: HLS Code Hoisting Solution C/RTL Cosimulation Summary

Inoltre, si può notare come il numero di risorse relative alla soluzione hardware in questione risulta essere pressocché il medesimo tranne per il numero di FF che risulta essere ridotto di circa il 18%. In aggiunta, si evidenzia come, rispetto alla soluzione iniziale, il periodo di clock raggiunto nella fase di post-implementation (stimato) risulta essere maggiore e, pertanto, avente una corrispondente incertezza minore.

Resource	**VHDL**
SLICE	75
LUT	270
FF	131
DSP	2
BRAM	0
SRL	0

Table 29: HLS Code Hoisting Solution Export RTL Resource Usage

Timing	**VHDL**
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	6.847

Table 30: HLS Code Hoisting Solution Export RTL Final Timing

Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

LUT	**LUTRAM**	**FF**	**BRAM**	**DSP**	**IO**	**BUFG**
270	32	134	0	2	71	1

Table 31: Vivado Code Hoisting Solution Utilization Report [#]

Dal momento che il WNS è diminuito, si può notare come la maximum clock frequency sia diminuita rispetto alla soluzione non ottimizzata.

Cycles [#]	**Clock Constraint** [ns]	**WNS** [ns]	**Maximum Clock Frequency** [Mhz]
43	10	3.074	144.3834825

Table 32: Vivado Code Hoisting Solution Timing Report

Per quanto riguarda la potenza dinamica e l'energia per singola operazione, si evidenzia un aumento rispetto alla soluzione iniziale. Si può ipotizzare che, dal punto di vista della potenza, il tool riesca meglio ad ottimizzare direttamente tutte le operazioni relative ad accumulo e shifting nello stesso ciclo, anziché averne 10 nel ciclo ed una al di fuori.

BRAM	**Clock Enable**	**Clocks**	**DSP**	**Logic**	**Set/Reset** [mW]	**Data**
0	0.370487687	1.756788697	0.41467679	0.838255044	3.35E-03	1.381990616

Table 33: Vivado Code Hoisting Solution Dynamic Power Report [mW]

Dynamic Total
4.765546238

Table 34: Vivado Code Hoisting Solution Dynamic Power Report [mW]

Energy Single Operation
47.65546238

Table 35: Vivado Code Hoisting Solution Energy Single Operation Report [pJ]

Pertanto, se da una parte si è ottenuto una piccola diminuzione delle risorse avendo gestito l'overhead di risorse dovuto alle istruzioni condizionali presenti all'interno del loop, dall'altra parte si è ottenuto un aumento di circa il 21% della potenza dinamica rispetto alla soluzione iniziale noon ottimizzata.

5.4 Loop Fission Solution

Considerando la soluzione precedente, si è risolto la possibile problematica di overhead delle risorse dovuta alle istruzioni condizionali presenti all'interno del loop. Bisogna notare, però, che all'interno di tale ciclo for siano presenti istruzioni potenzialmente complesse dal punto di vista computazionale. In particolare, il tool, durante la compilazione del codice, cerca in qualche modo di apportare qualche tipologia di ottimizzazione al loop in questione. Pertanto, si potrebbe pensare di implementare due loop: uno per l'operazione di shifting ed un altro per l'operazione di accumulo. Quindi, effettuando la scissione del loop, relativo alla soluzione precedente, si potrebbe pensare che il tool possa effettuare tali ottimizzazioni in maniera più efficiente dal momento che le ottimizzazioni verrebbero effettuate su due cicli distinti. Quello che ci si aspetta, però, è un aumento significativo della latenza totale dovuto al fatto che, in questo caso, bisogna considerare la latenza di due loop.

```
#include "definitions.h"

void firConvolutionLoopFission(samplesType inputFilter, samplesType* outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>0; --i ) {
        shiftRegister[i] = shiftRegister[i-1];
    }
    shiftRegister[0] = inputFilter;
    accumulator = 0;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

Effettuando la sintesi, la C/RTL Cosimulation ed Export RTL si ottengono i seguenti report.

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.510	1.25

Table 36: HLS Loop Fission Solution Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
66	66	66	66

Table 37: HLS Loop Fission Solution Latency Summary (clock cycles)

Si può notare come la latenza sia aumentata rispetto alle soluzioni precedenti in cui era presente un solo loop. Questo può essere, inoltre, analizzato nella tabella "Table 38: HLS Loop Fission Solution Latency Loops Summary" sotto allegata dove è possibile osservare che i due loop vengono riconosciuti come due loop indipendenti tra loro e con una latenza associata. In particolare, "loopShifting" presenta un trip count pari a 10 dal momento che il caso $i == 0$ viene gestito secondo la tecnica del code hoisting, cioè al di fuori del ciclo stesso.

Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
Name	min	max	min	max	achieved	target	
- loopShifting	20	20	2	2	-	-	10
- loopAccumulator	44	44	4	4	-	-	11

Table 38: HLS Loop Fission Solution Latency Loops Summary

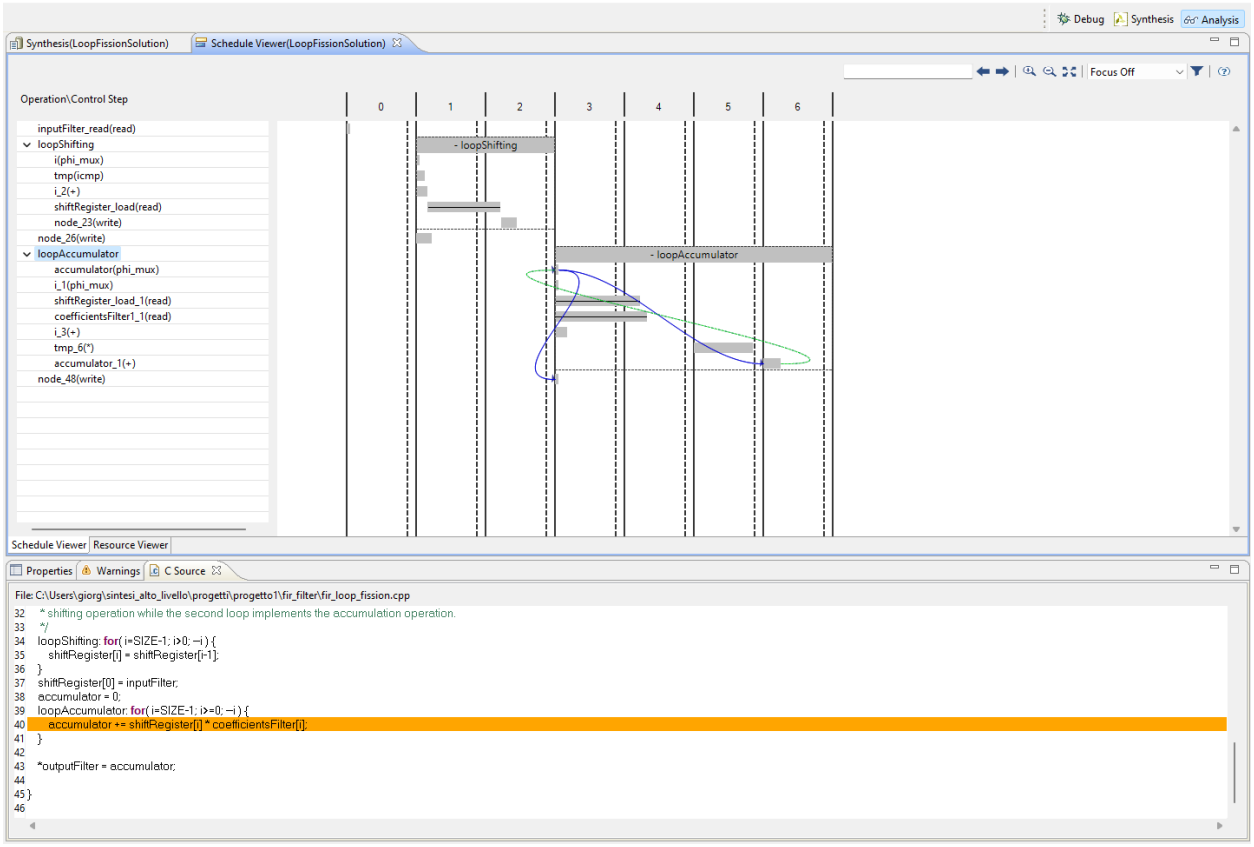


Figure 15: HLS Loop Fission Solution Analysis

La scissione del loop nei cicli "loopShifting" e "loopAccumulator" la si può notare, inoltre, nella figura sopra allegata che mostra la finestra di *Analysis*. Bisogna di nuovo precisare che, tale tecnica ha l'obiettivo di favorire il tool di ottimizzare i due loop in maniera indipendente effettuando magari uno scheduling delle operazioni differente, attraverso procedure proprietarie, rispetto alle soluzioni precedenti dove era previsto un ciclo for unico.

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
DSP	-	-	-	-
Expression	-	2	0	96
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	74	8
Multiplexer	-	-	-	110
Register	-	-	131	-
Total	0	2	205	214
Available	280	220	106400	53200
Utilization (%)	0	~0	~0	~0

Table 39: HLS Loop Fission Solution Utilization Estimates Summary

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	66	66	67	66	66	67

Table 40: HLS Loop Fissiong Solution C/RTL Cosimulation Summary

Resource	VHDL
SLICE	44
LUT	157
FF	106
DSP	2
BRAM	0
SRL	0

Table 41: HLS Loop Fissiong Solution Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	6.363

Table 42: HLS Loop Fissiong Solution Export RTL Final Timing

Si può evidenziare come, dopo aver effettuato l'Export RTL, il numero di cicli di clock affinché un risultato venga processato in uscita sia aumentato a 67 rispetto a quello ottenuto nella soluzione precedente pari a 43. Di fondamentale importanza è la diminuzione dell'utilizzazione delle risorse. Infatti, si può notare, rispetto alla soluzione hardware basata sul code hoisting, come il numero di SLICE sia diminuito di circa il 41%, quello delle LUT di circa il 42% e quello dei FF di circa il 19%.
Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
158	32	106	0	2	71	1

Table 43: Vivado Loop Fission Solution Utilization Report [#]

Si può notare come l'utilizzazione delle risorse sia minore (-42% LUT e -21% FF) rispetto alla solution precedente e, inoltre, la maximum clock frequency sia aumentata (dato dal fatto che il WNS è maggiore). Ovviamente, a tali vantaggi corrisponde, come già anticipato precedentemente, un aumento dei cicli di clock necessari ($+55.8\%$) affinchè un risultato venga processato in uscita.

Cycles [#]	Clock Constraint [ns]	WNS [ns]	Maximum Clock Frequency [Mhz]
67	10	3.464	152.998776

Table 44: Vivado Loop Fission Solution Timing Report

BRAM	Clock Enable	Clocks	DSP	Logic	Set/Reset [mW]	Data
0	0.292603218	1.098937588	0.321194879	0.590288255	0.004160448	0.937705627

Table 45: Vivado Loop Fission Solution Dynamic Power Report [mW]

Dynamic Total
3.244890014

Table 46: Vivado Loop Fission Solution Dynamic Power Report [mW]

Energy Single Operation
32.44890014

Table 47: Vivado Loop Fission Solution Energy Single Operation Report [pJ]

Si evidenzia che, la potenza dinamica totale e l'energia per singola operazione risultano essere diminuite, rispetto alla soluzione hardware precedente, di circa il 32%. In particolare, le maggiori diminuzioni si hanno in corrispondenza dei seguenti contributi: Clocks (-38%), Logic (-30%) e Data (-32%). Infatti, bisogna ricordare che l'utilizzazione delle risorse, come già precedentemente citato, è anch'essa notevolmente diminuita rispetto alla solution precedente.

5.5 Loop Unrolling Solution

Considerando l'architettura hardware precedente, è possibile, effettivamente, implementare un approccio più veloce dal punto di vista delle operazioni, cioè una soluzione basata su parallelismo. In particolare, allocando un maggior numero di risorse, è possibile eseguire delle operazioni in parallelo riducendo di conseguenza la latenza, cioè il tempo di esecuzione totale.
Tale approccio è possibile implementarlo in due modi differenti:

- **Manuale**
  Unrolling ottenuto tramite rimodulazione software
- **Automatico**
  Unrolling ottenuto tramite direttiva proprietaria (pragma)

In particolare, sia l'unrolling manuale sia quello automatico, sono stati ottenuti considerando sia un fattore di parallelismo pari a 2 sia pari a 4. Inoltre, bisogna specificare che, per quanto riguarda l'unrolling automatico (tramite pragma), è stata realizzata un'ulteriore soluzione hardware, tenendo conto sia del fattore 2 sia del fattore 4, dove è stato considerato sia il pragma di unrolling sia il pragma di partitioning. Il partizionamento serve per risolvere un problema tipicamente causato dagli array. Gli array sono implementati come BRAM, solitamente progettate per un dual-port massimo. Questo può limitare il throughput di un algoritmo ad alta intensità di read/write. La larghezza di banda può essere migliorata dividendo l'array (una singola BRAM) in array più piccoli (più BRAM), aumentando di fatto il numero di porte. Gli array vengono partizionati utilizzando la direttiva ARRAY_PARTITION. Vivado HLS offre tre tipi di partizionamento degli array. I tre tipi di partizionamento sono:

- **block**]
  L'array originale viene suddiviso in blocchi di uguali dimensioni di elementi consecutivi dell'array originale.
- **cyclic**
  L'array originale viene suddiviso in blocchi di uguali dimensioni che interlacciano gli elementi dell'array originale.
- **complete**
  L'operazione predefinita consiste nel dividere l'array nei suoi singoli elementi. Ciò corrisponde alla risoluzione di una memoria in registri.

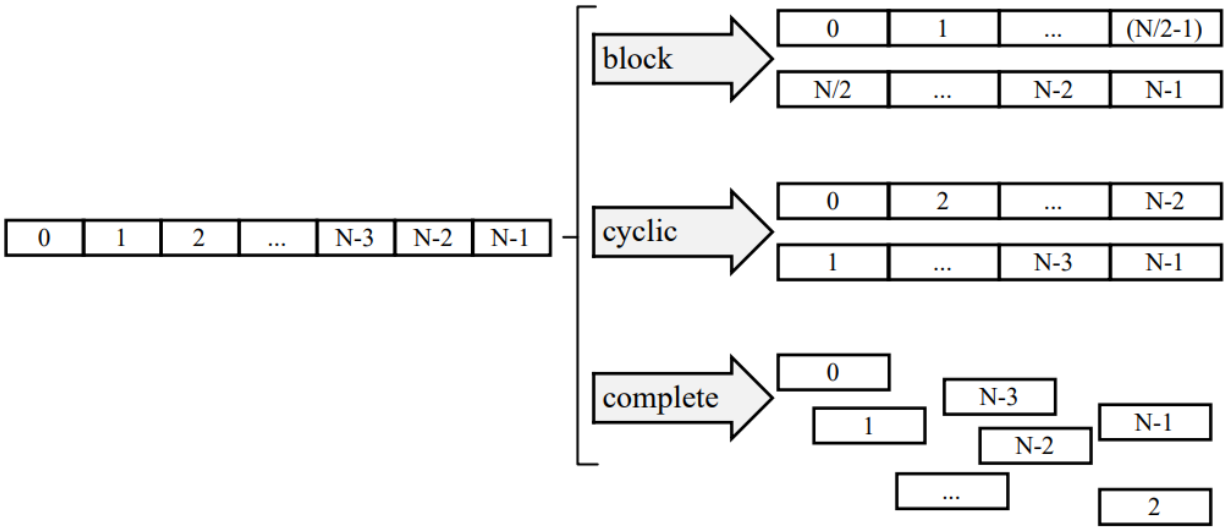


Figure 16: HLS Array Partitioning

Pertanto, è possibile elencare qui di seguito le soluzioni hardware progettate per l'approccio del parallelismo:

- Unrolling Manuale Fattore=2
- Unrolling Manuale Fattore=4
- Unrolling Automatico Fattore=2
- Unrolling Automatico Fattore=4
- Unrolling Automatico con Partitioning Fattore=2
- Unrolling Automatico con Partitioning Fattore=4

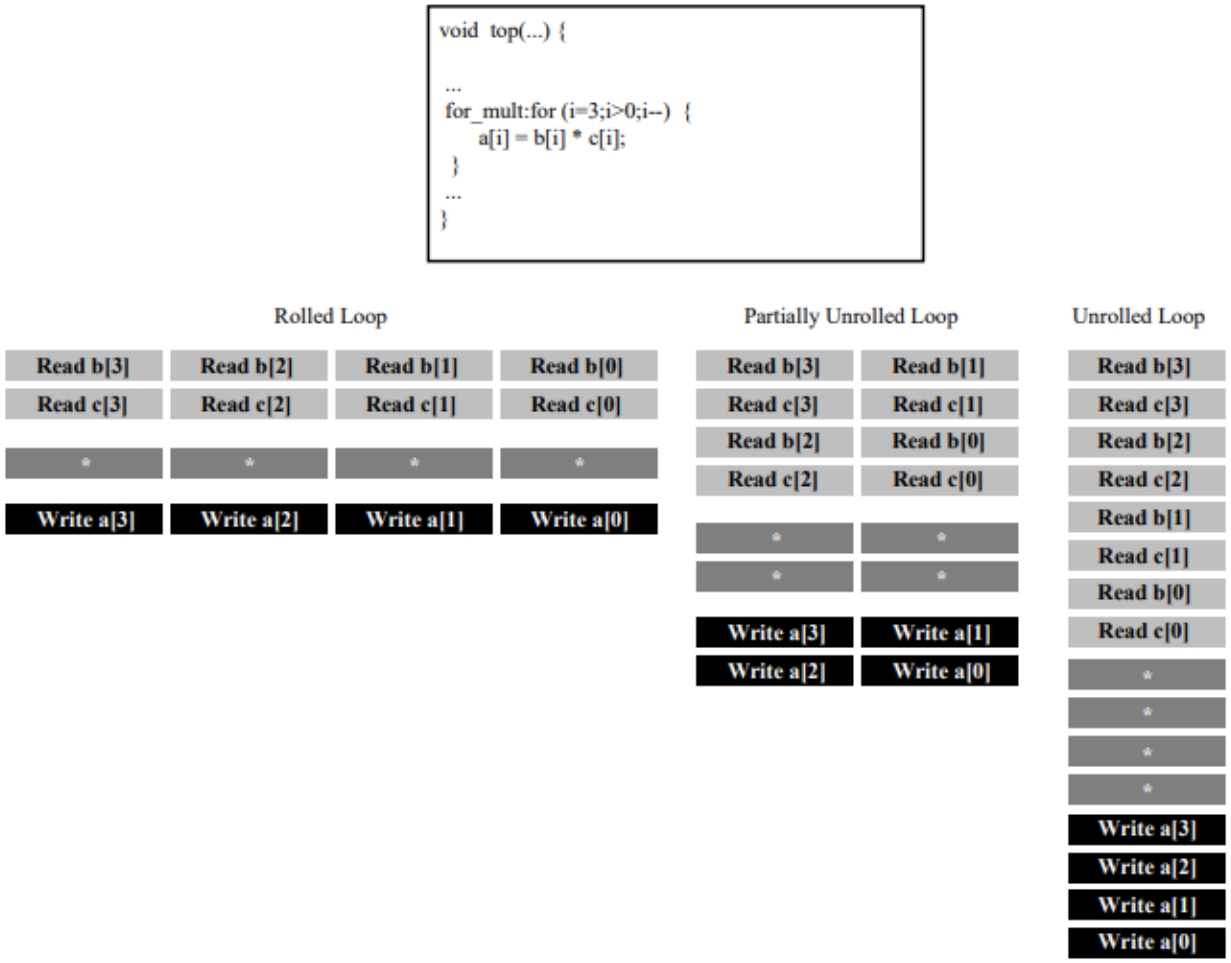


Figure 17: HLS Loop Unrolling

A questo proposito, verranno analizzati i report considerando le soluzioni hardware aventi stesso fattore così da poter effettuare confronti adeguati tra le varie implementazioni. Bisogna precisare, che in questo caso è stato effettuando un loop unrolling parziale, cioè tenendo conto della figura sopra allegata è stato considerato un fattore 2 o un fattore 4 anziché utilizzare un fattore che potesse *ricoprire* la totalità degli elementi processabili (capacità del filtro). Ovviamente, la figura sopra allegata è solo un esempio per rappresentare meglio il concetto di unrolling e partitioning.

5.5.1 Loop Unrolling Factor=2

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor2(samplesType inputFilter, samplesType* outputFilter)
    {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>1; i=i-2 ) {
        shiftRegister[i] = shiftRegister[i-1];
        shiftRegister[i-1] = shiftRegister[i-2];
    }
    if( i==1 ) {
        shiftRegister[1] = shiftRegister[0];
    }
    shiftRegister[0] = inputFilter;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor2Pragma(samplesType inputFilter, samplesType*
    outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>0; --i ) {
        #pragma HLS unroll factor=2
        shiftRegister[i] = shiftRegister[i-1];
    }
    shiftRegister[0] = inputFilter;
    accumulator = 0;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor2PP(samplesType inputFilter, samplesType* outputFilter
    ) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>0; --i ) {
        #pragma HLS array_partition variable=shiftRegister complete
        #pragma HLS unroll factor=2
        shiftRegister[i] = shiftRegister[i-1];
    }
    shiftRegister[0] = inputFilter;
    accumulator = 0;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

Solution	Clock	Target	Estimated	Uncertainty
Manuale	ap_clk	10.00	8.510	1.25
Automatico	ap_clk	10.00	8.510	1.25
Automatico con Partitioning	ap_clk	10.00	8.510	1.25

Table 48: HLS Loop Unrolling Factor=2 Solution Timing Summary (ns)

Si può notare come la latenza associata alla soluzione hardware basata su unrolling manuale sia la medesima di quella basata su unrolling automatico (effettuato mediante pragma). Questo è dovuto al fatto che l'implementazione è la stessa: nel primo caso è effettuato manualmente mentre nel secondo caso è effettuato mediante direttiva proprietaria del tool. Invece, nel caso della soluzione hardware basata su unrolling automatico e partitioning, si riscontra una latenza maggiore. Questo risultato è dovuto al fatto che, tramite il pragma di partizionamento, si è dovuto gestire, inoltre, letture e scritture in parallelo.

Solution	Latency		Interval	
	min	max	min	max
Manuale	56	56	56	56
Automatico	56	56	56	56
Automatico con Partitioning	61	66	61	66

Table 49: HLS Loop Unrolling Factor=2 Solution Latency Summary (clock cycles)

Si può evidenziare come, nel caso della soluzione hardware con urolling automatico e partizionamento, l'aumento della latenza si ha soltanto in corrispondenza del loop di shifting, cioè proprio dove è stato collocato il pragma di partitioning.

Solution	Loop Name	Latency		Iteration Latency		Initiation Interval		Trip
		min	max	min	max	achieved	target	Count
Manuale	- loopShifting	10	10	2	2	-	-	5
	- loopAccumulator	44	44	4	4	-	-	11
Automatico	- loopShifting	10	10	2	2	-	-	5
	- loopAccumulator	44	44	4	4	-	-	11
Automatico	- loopShifting	15	20	3	4	-	-	5
con Partitioning	- loopAccumulator	44	44	4	4	-	-	11

Table 50: HLS Loop Unrolling Factor=2 Solution Latency Loops Summary

Aprendo la sezione Analysis del tool si può vedere meglio nel dettaglio che, all'interno del loop di shifting, sono presenti 2 shifting in parallelo (dal momento che è stato considerato un fattore di parallelismo pari a 2) sia peer la soluzione hardware di unrolling manuale sia per quella di unrolling automatico (mediante pragma).

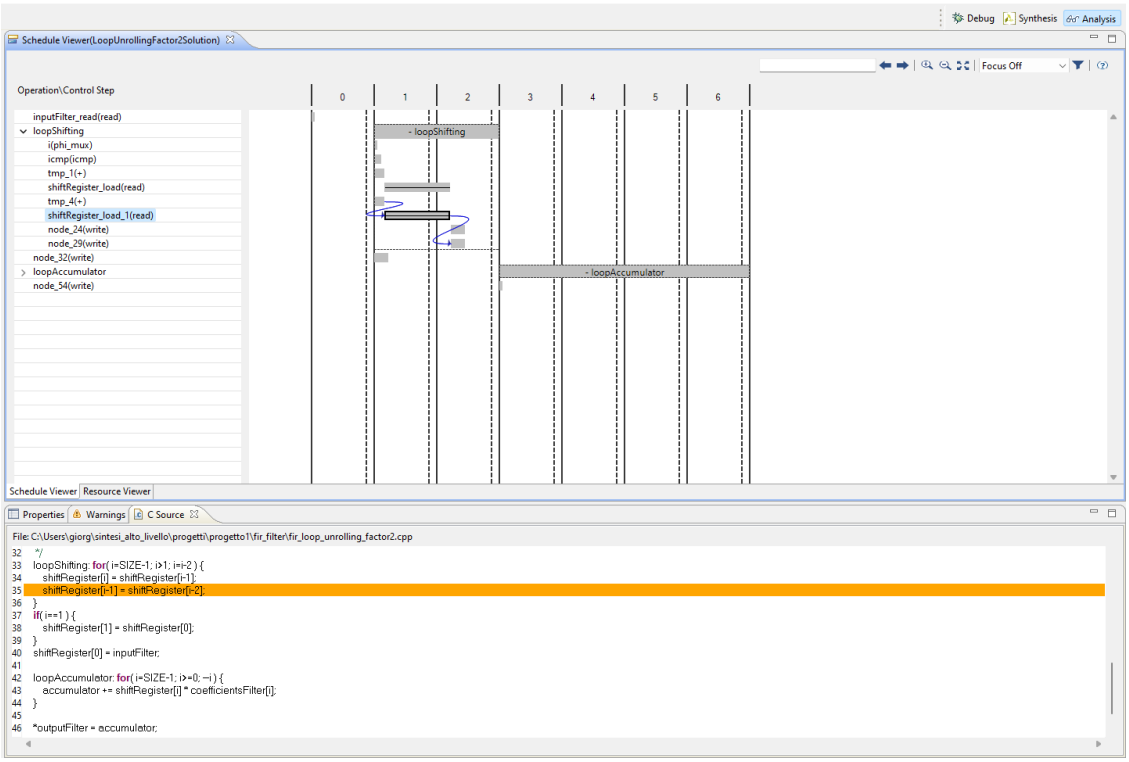


Figure 18: HLS Loop Unrolling Manual Factor=2 Analysis

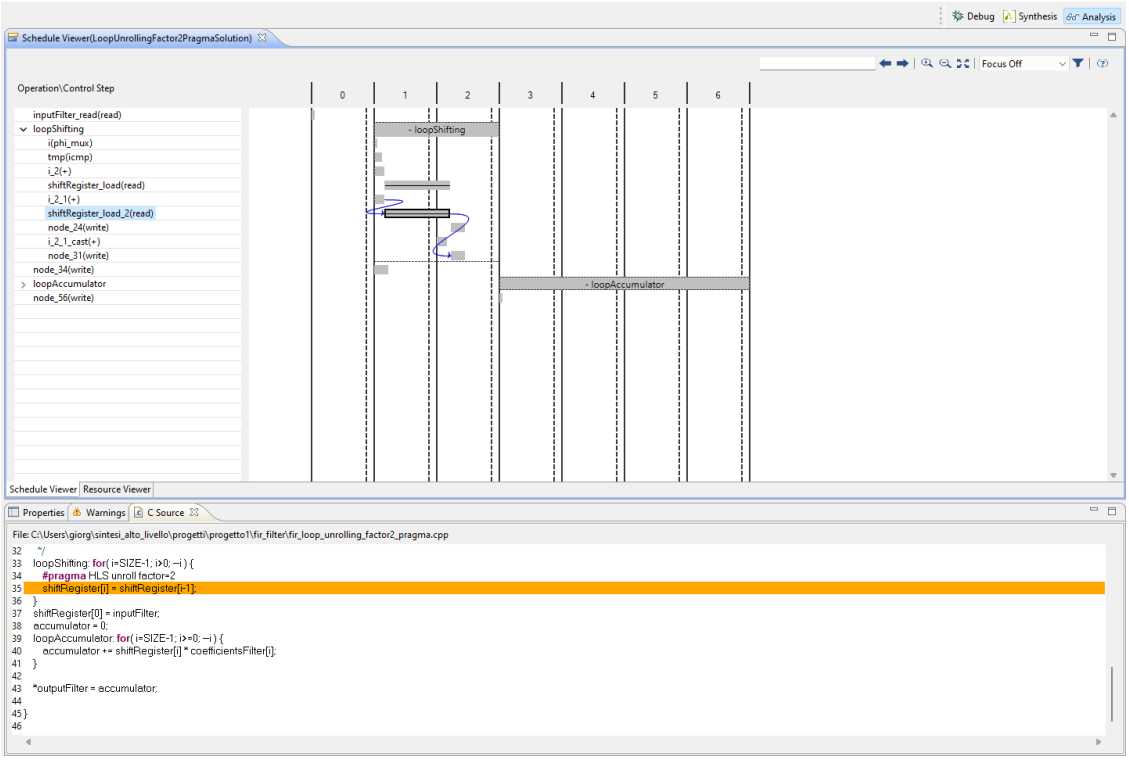


Figure 19: HLS Loop Unrolling Automatic Factor=2 Analysis

Analizzando nel dettaglio il report si può notare come il numero di DSP e LUT sia pressocché il medesimo per tutte e tre le soluzioni hardware proposte. Per quanto riguarda, invece, il numero di BRAM utilizzate, le soluzioni basate su unrolling manuale e automatico presentano un'utilizzazione pari a 2, mentre quella basta su unrolling automatico e partitioning presenta un numero di BRAM utilizzato pari a 0. Questo è dovuto al fatto che il pragma di partizionamento, al fine di garantire accessi di read/write in memoria in parallelo, impone di non utilizzare le BRAM dual-port (che limiterebbero tali accessi). In particolare, impostando la tipologia di partizionamento *complete*, la direttiva fa in modo che la soluzione hardware impieghi come memoria dei singoli registri, cosí che siano garantiti gli accessi in memoria in parallelo. Tanto è vero che si può notare un aumento considerevole dei FF in corrispondenza dell'architettura basata su unrolling automatico e partizionamento.

Solution	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
Manuale	2	2	145	236
Automatico	2	2	141	251
Automatico con Partitioning	0	2	912	289

Table 51: HLS Loop Unrolling Factor=2 Solution Utilization Estimates [#]

Qui di seguito vengono riportati i report relativi alla C/RTL Cosimulation, dove è possibile analizzare il numero di cicli di clock che servono per ottenere un risultato in uscita, e quello relativo a Export RTL.

Solution	RTL	Status	**Latency**			**Interval**		
			min	avg	max	min	avg	max
Manuale	VHDL	Pass	56	56	57	56	56	57
Automatico	VHDL	Pass	56	56	57	56	56	57
Automatico con Partitioning	VHDL	Pass	65	65	66	65	65	66

Table 52: HLS Loop Unrolling Factor=2 Solution C/RTL Cosimulation Report

Solution	**SLICE**	**LUT**	**FF**	**DSP**	**BRAM**	**CP required**	**CP achieved post-synthesis**	**CP achieved post-implementation**
Manuale	31	97	72	2	2	10	5.745	5.692
Automatico	29	97	72	2	2	10	5.745	5.692
Automatico con Partitioning	294	413	843	2	0	10	5.745	6.188

Table 53: HLS Loop Unrolling Factor=2 Solution Export RTL Report

Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
1 create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

Analizzando il report di utilizzazione delle risorse generato da Vivado dopo aver effettuato il processo di implementazione, è possibile notare come il numero di risorse stimato dal tool HLS è leggermente mutato. Però, si può evidenziare come il cambiamento delle risorse tra le varie soluzioni hardware è il medesimo, cioè il numero di LUT e FF nel caso dell'implementazione con unrolling automatico e partitioning sia aumentato considerevolmente e il numero di BRAM si è ridotto a zero.

Solution	LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
Manuale	98	0	72	1	2	71	1
Automatico	98	0	72	1	2	71	1
Automatico con Partitioning	413	0	843	0	2	71	1

Table 54: Vivado Loop Unrolling Factor=2 Solution Utilization Report [#]

Si è, inoltre, analizzato l'utilizzazione delle risorse effettuando un confronto con la soluzione hardware basata sulla scissione del loop dal momento che tali architetture basate su unrolling sono basate sulla prima citata. In particolare, si può notare come il numero di risorse utilizzate sia pressoché il medesimo tra la soluzione hardware basta su loop fission e quelle basate rispettivamente sull'unrolling manuale e automatico di fattore 2. Più nello specifico, considerando le implementazioni basate sul parallelismo manuale e automatico rispetto a quella basta sul loop fission, il numero di LUT è diminuito di circa il 40%, il numero di FF è diminuito di circa il 32% e il numero di BRAM è aumentato di un'unità. Invece, effettuando un confronto tra quella basata su partitioning e quella basata sulla scissione del loop, il numero di LUT è aumentato di circa il 161% e il numero di FF di circa il 695%.

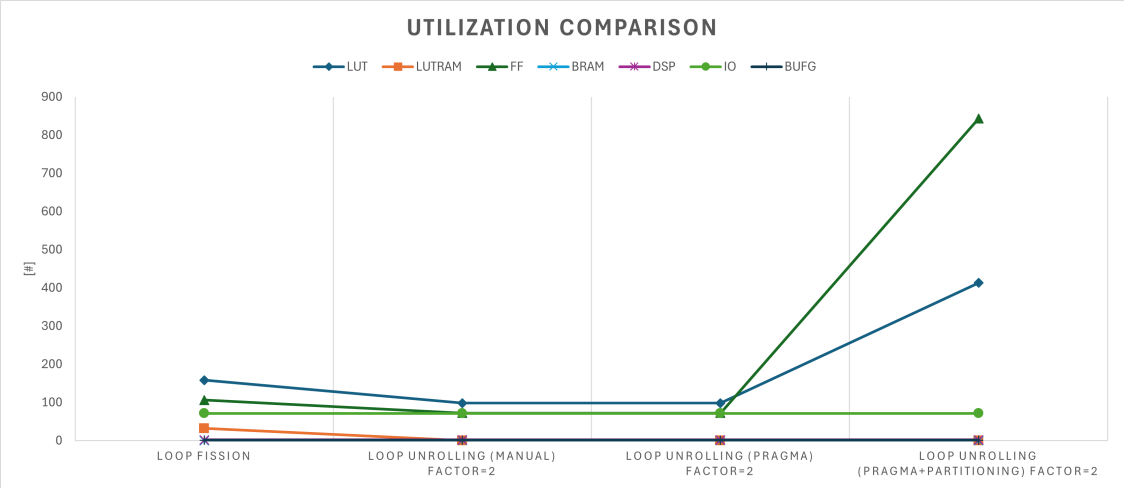


Figure 20: Vivado Loop Unrolling Factor=2 Utilization Plot

Infine, per quanto riguarda il timing, è possibile notare come il numero di cicli sia pressocché il medesimo tra la soluzione basata su unrolling manuale e quella basata su unrolling automatico. Invece, come precedentemente citato, la soluzione basata su partizionamento presenta un numero di cicli di clock, tali per garantire un risultato, maggiore.

Solution	Cycles [#]	Clock Constraint [ns]	WNS [ns]	Maximum Clock Frequency [MHz]
Manuale	57	10	4.33	176.366843
Automatico	57	10	4.33	176.366843
Automatico con Partitioning	66	10	3.469	153.1159087

Table 55: Vivado Loop Unrolling Factor=2 Solution Timing Report

Effettuando un confronto grafico e tenendo conto anche della soluzione basata sulla scissione del loop, è possibile evidenziare un numero di cicli di clock, tali per garantire un risultato, un WNS e una maximum clock frequency pressoché uguali tra l'implementazione basata su loop fission e quella basata su partizionamento. Questo accade dal momento che la soluzione hardware basata su unrolling automatico e partitioning fa

diminuire il numero di cicli di clock tramite il parallelismo e lo fa aumentare allo stesso tempo a causa del partizionamento.

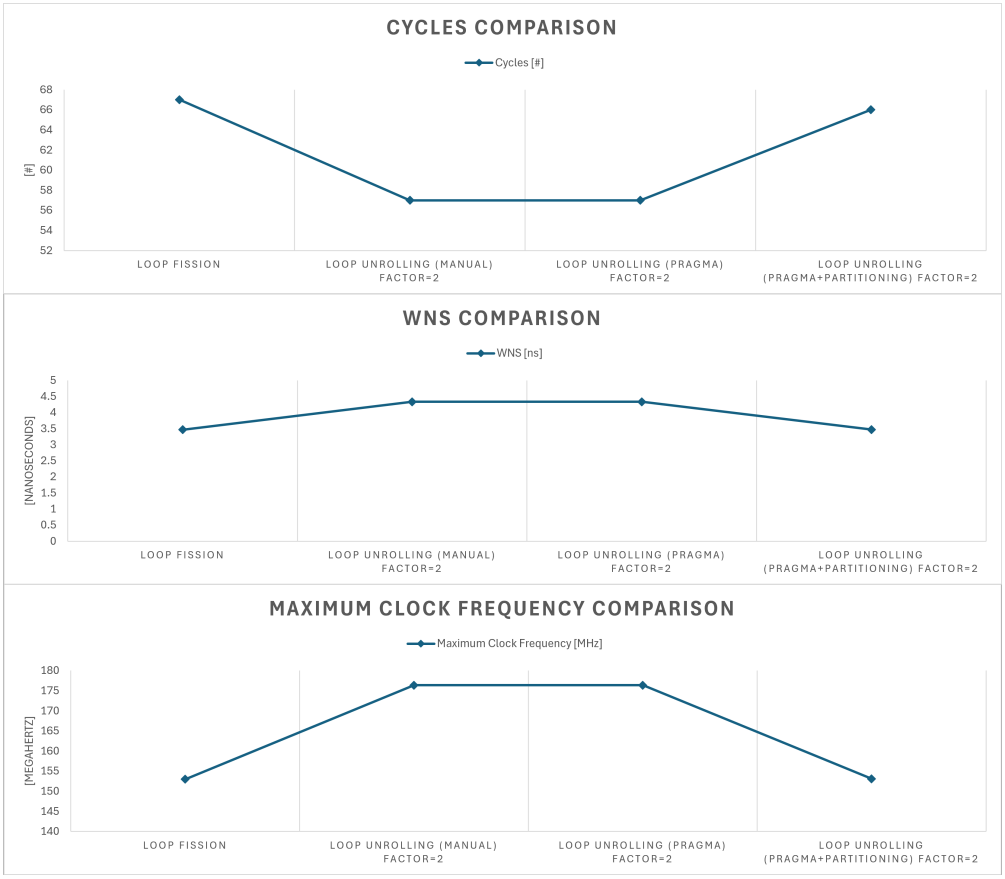


Figure 21: Vivado Loop Unrolling Factor=2 Timing Plot

Per quanto riguarda, invece, la potenza dinamica associata alle tre soluzioni hardware proposte in questa sezione, è possibile notare come i contributi associati al Clock Enable e Clocks risultano essere notevolmente più grandi in corrispondenza dell'implementazione basata su partizionamento. Questo potrebbe essere causato dal notevole aumento dell'utilizzazione dei FF per questa solution.

Solution	**BRAM**	**Clock Enable**	**Clocks**	**DSP**	**Logic**	**Set/ Reset**	**Data**
Manuale	1.250551548	0.096387172	0.900532817	0.268251897	0.260709843	0.003146866	0.423992984
Automatico	1.240851358	0.082524632	0.960682868	0.272355421	0.266662013	0.00428147	0.425589533
Automatico con Partitioning	0	0.325270201	2.352835611	0.263715046	0.575191109	0.007010513	0.750690058

Table 56: Vivado Loop Unrolling Factor=2 Solution Dynamic Power Report [mW]

Infatti, è possibile riscontrare un aumento della potenza dinamica totale e dell'energia per singola operazione in corrispondenza dell'architettura basata su unrolling e partitioning. Invece, per quanto riguarda le altre due solution (unrolling manuale e automatico), entrambi i parametri appena citati risultano essere i medesimi.

Solution	Dynamic Total
Manuale	3.203573127
Automatico	3.252947294
Automatico con Partitioning	4.274712538

Table 57: Vivado Loop Unrolling Factor=2 Solution Dynamic Power Report [mW]

Solution	Energy Single Operation
Manuale	32.03573127
Automatico	32.52947294
Automatico con Partitioning	42.74712538

Table 58: Vivado Loop Unrolling Factor=2 Solution Energy Single Operation Report [pJ]

Analizzando graficamente queste tre soluzioni e in aggiunta anche l'implementazione basata sulla scissione del loop, è possibile notare come quella basasta su loop fission presenta un medesimo valore di potenza dinamica totale ed energia per singola operazione rispetto alle due soluzioni di unrolling manuale e automatico.

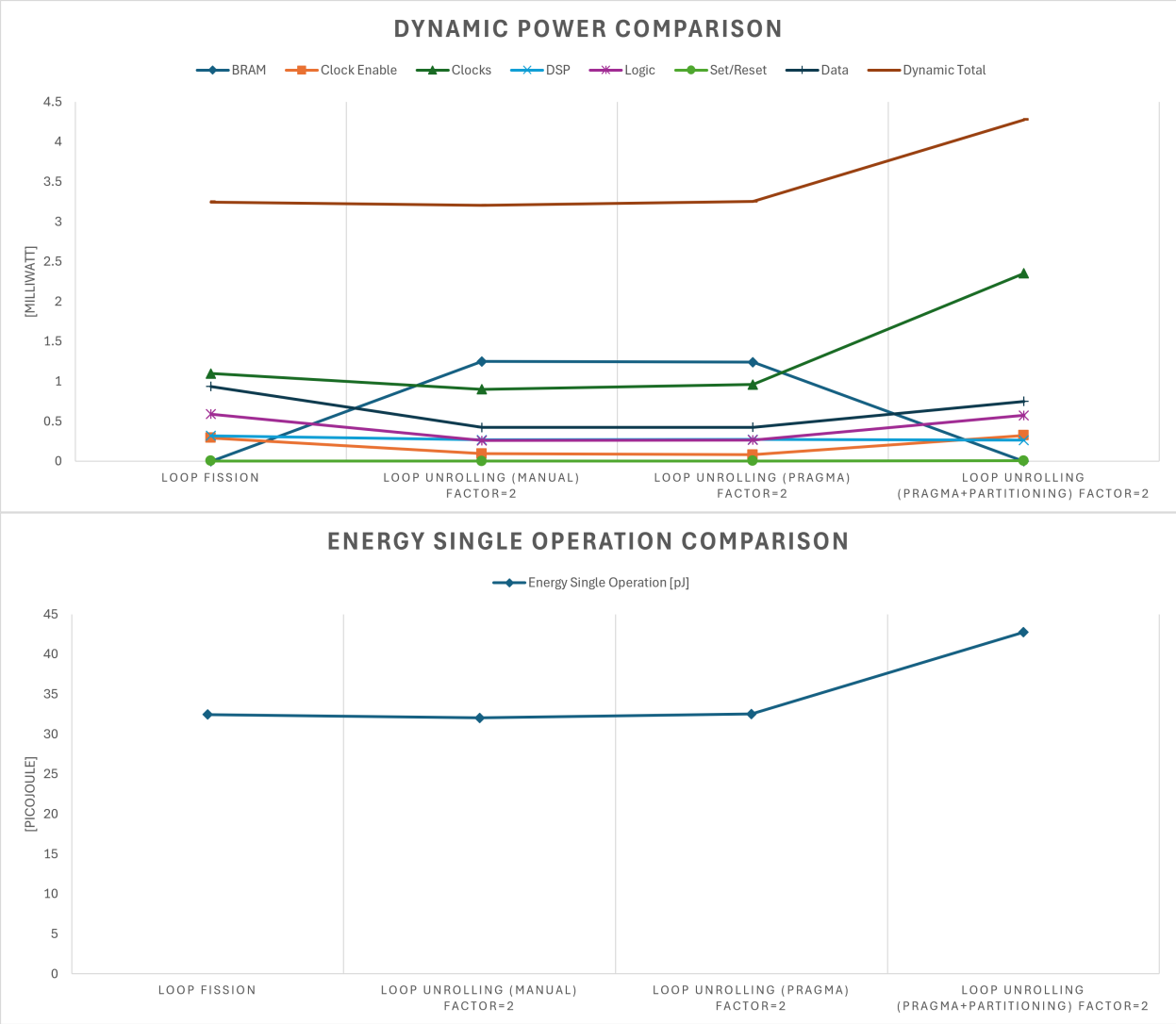


Figure 22: Vivado Loop Unrolling Factor=2 Dynamic Power Plot

5.5.2 Loop Unrolling Factor=4

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor4(samplesType inputFilter, samplesType* outputFilter)
    {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>1; i=i-4 ) {
        shiftRegister[i] = shiftRegister[i-1];
        shiftRegister[i-1] = shiftRegister[i-2];
        shiftRegister[i-2] = shiftRegister[i-3];
        shiftRegister[i-3] = shiftRegister[i-4];
    }
    if( i==1 ) {
        shiftRegister[1] = shiftRegister[0];
    }
    shiftRegister[0] = inputFilter;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor4Pragma(samplesType inputFilter, samplesType*
    outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>0; --i ) {
        #pragma HLS unroll factor=4
        shiftRegister[i] = shiftRegister[i-1];
    }
    shiftRegister[0] = inputFilter;
    accumulator = 0;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
}
```

```
#include "definitions.h"

void firConvolutionLoopUnrollingFactor4PP(samplesType inputFilter, samplesType* outputFilter
    ) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loopShifting: for( i=SIZE-1; i>0; --i ) {
        #pragma HLS array_partition variable=shiftRegister complete
        #pragma HLS unroll factor=4
        shiftRegister[i] = shiftRegister[i-1];
    }
    shiftRegister[0] = inputFilter;
    accumulator = 0;
    loopAccumulator: for( i=SIZE-1; i>=0; --i ) {
        accumulator += shiftRegister[i] * coefficientsFilter[i];
    }
    *outputFilter = accumulator;
```

```
20 }
```

Solution	Clock	Target	Estimated	Uncertainty
Manuale	ap_clk	10.00	8.510	1.25
Automatico	ap_clk	10.00	8.510	1.25
Automatico con Partitioning	ap_clk	10.00	8.510	1.25

Table 59: HLS Loop Unrolling Factor=4 Solution Timing Summary (ns)

Solution	Latency		Interval	
	min	max	min	max
Manuale	58	58	58	58
Automatico	62	66	62	66
Automatico con Partitioning	62	64	62	64

Table 60: HLS Loop Unrolling Factor=4 Solution Latency Summary (clock cycles)

Solution	Loop Name	Latency		Iteration Latency		Initiation Interval		Trip Count
		min	max	min	max	achieved	target	
Manuale	- loopShifting	12	12	4	4	-	-	3
	- loopAccumulator	44	44	4	4	-	-	11
Automatico	- loopShifting	15	18	5	5	-	-	3
	- loopAccumulator	44	44	4	4	-	-	11
Automatico con Partitioning	- loopShifting	15	17	5	5	-	-	3
	- loopAccumulator	44	44	4	4	-	-	11

Table 61: HLS Loop Unrolling Factor=4 Solution Latency Loops Summary

In questo caso si può notare come il trip count sia, ovviamente, il medesimo per tutte e tre le soluzioni. Ciò che cambia effettivamente, come nel caso del fattore pari a 2, è la latency associata ai loop. In particolare, analizzando più nel dettaglio, tramite l'interfaccia Analysis, si può evidenziare come il loopShifting, in cui viene implementato un parallelismo pari a 4, viene gestito in maniera differente. Infatti, nel caso del loop unrolling manuale, vengono effettuate in parallelo due read alla volta e poi successivamente vengono eseguite in parallelo due write alla volta. Invece, nel caso della soluzione hardware basata su unrolling automatico, vengono effettuate una read e una write in parallelo per ogni ciclo di latenza. Nello specifico, considerando quattro shifting per ogni iterazione, viene effettuata la read al primo ciclo di latenza e, successivamente, viene eseguita la write corrispondente in parallelo alla read relativo al secondo shifting e così via. Di conseguenza, si avranno cinque colpi di latenza per ogni iterazione come riportato dal valore di iteration latency del report di sintesi.

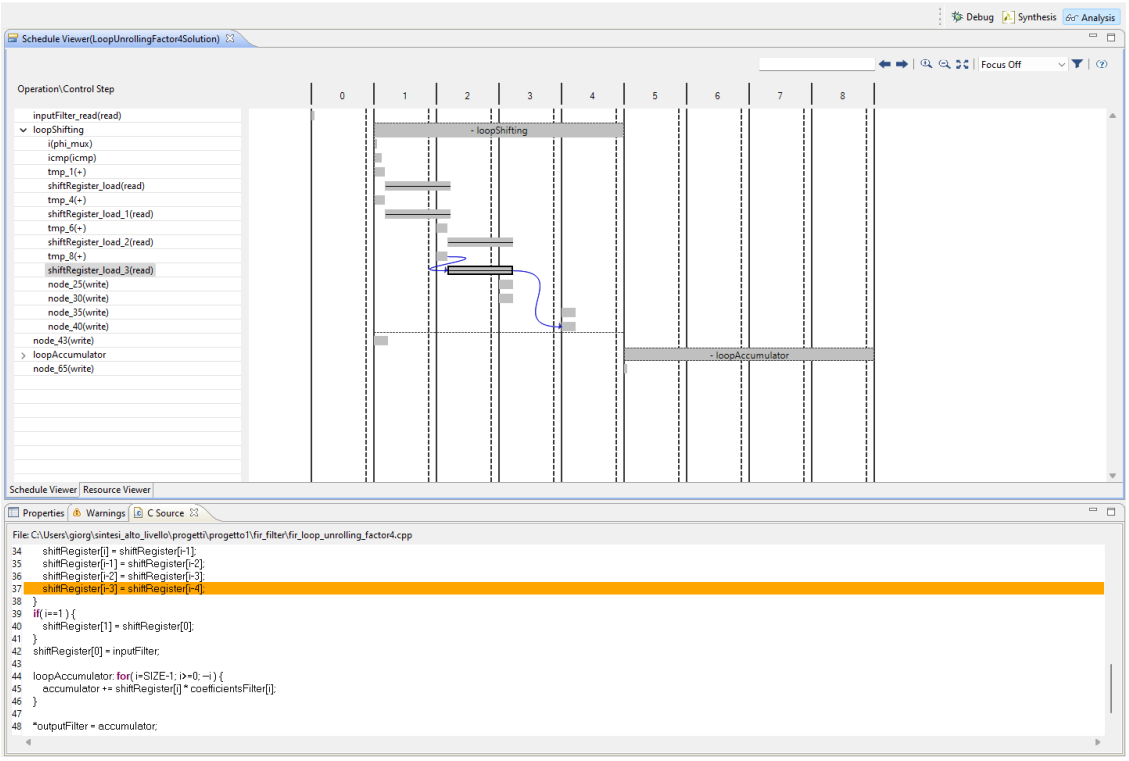


Figure 23: HLS Loop Unrolling Manual Factor=4 Analysis

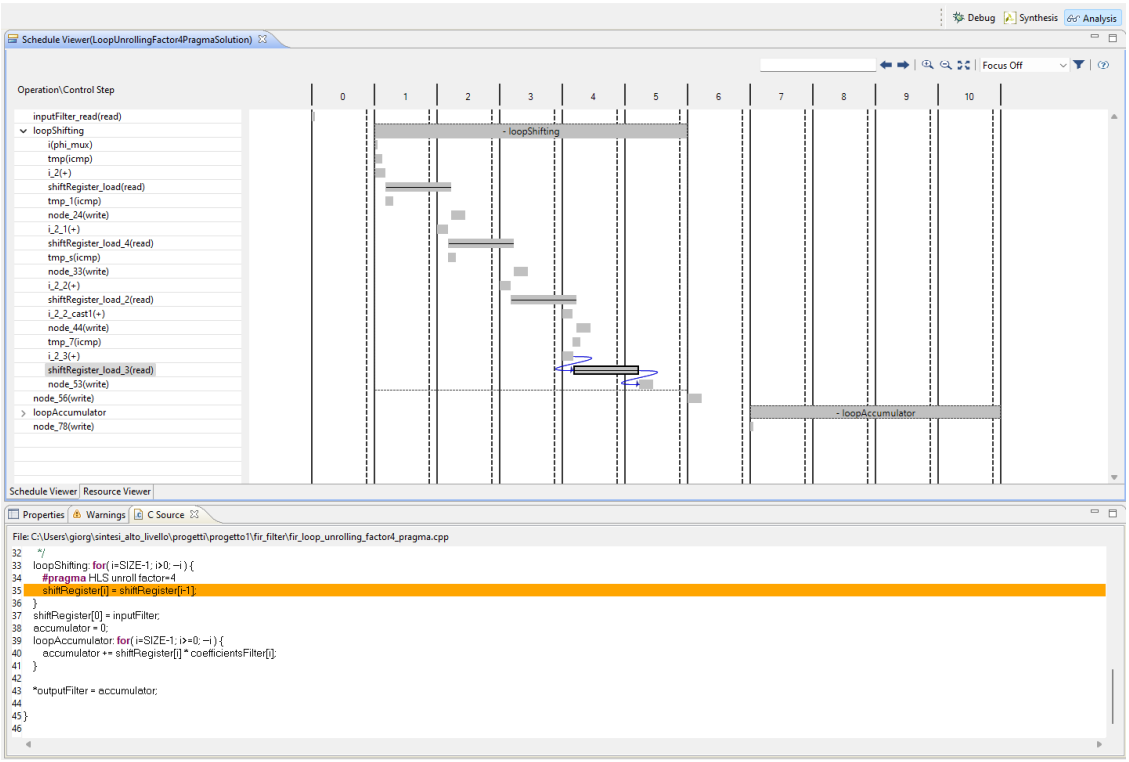


Figure 24: HLS Loop Unrolling Automatic Factor=4 Analysis

Analogamente a quanto è stato analizzato e descritto nel caso di parallelismo di fattore pari a 2, anche in questo caso si può notare come l'utilizzazione delle risorse sia quasi la medesima tra il loop unrolling manuale e automatico. Invece, per quanto riguarda il loop unrolling automatico con partitioning, si ha un notevole aumento dell'utilizzazione sia di FF sia di LUT.

Solution	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
Manuale	2	2	221	318
Automatico	2	2	194	367
Automatico con Partitioning	0	2	928	748

Table 62: HLS Loop Unrolling Factor=4 Solution Utilization Estimates [#]

Qui di seguito vengono riportati i report relativi alla C/RTL Cosimulation, dove è possibile analizzare il numero di cicli di clock che servono per ottenere un risultato in uscita, e quello relativo a Export RTL.

Solution	RTL	Status	**Latency**			**Interval**		
			min	avg	max	min	avg	max
Manuale	VHDL	Pass	58	58	59	58	58	59
Automatico	VHDL	Pass	60	60	61	60	60	61
Automatico con Partitioning	VHDL	Pass	58	58	59	58	58	59

Table 63: HLS Loop Unrolling Factor=4 Solution C/RTL Cosimulation Report

Solution	**SLICE**	**LUT**	**FF**	**DSP**	**BRAM**	**CP required**	**CP achieved post-synthesis**	**CP achieved post-implementation**
Manuale	46	160	147	2	2	10	5.745	5.692
Automatico	40	145	93	2	2	10	5.745	5.692
Automatico con Partitioning	412	1145	864	2	0	10	5.745	7.109

Table 64: HLS Loop Unrolling Factor=4 Solution Export RTL Report

Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

Analogamente all'unrolling di fattore pari a 2, anche in questo caso in corrispondenza della soluzione hardware basata su unrolling e partizionamento si ha un notevole aumento di utilizzazione delle risorse. Inoltre, si può notare come il numero di FF utilizzati per la solution di unrolling automatico risulta essere circa il 37% in meno rispetto a quella basata su unrolling manuale. Evidentemente il tool, tramite la direttiva proprietaria, è riuscito ad effettuare delle ottimizzazione tali da garantire una diminuzione dei Flip Flop.

Solution	**LUT**	**LUTRAM**	**FF**	**BRAM**	**DSP**	**IO**	**BUFG**
Manuale	159	0	147	1	2	71	1
Automatico	145	0	93	1	2	71	1
Automatico con Partitioning	1145	0	864	0	2	71	1

Table 65: Vivado Loop Unrolling Factor=4 Solution Utilization Report [#]

Effettuando un confronto grafico e tenendo conto dei dati precedentemente ottenuti per la soluzione basata su scissione del loop, si può notare come l'utilizzazione delle risorse sia pressocché la medesima tra loop fission e loop unrolling automatico.

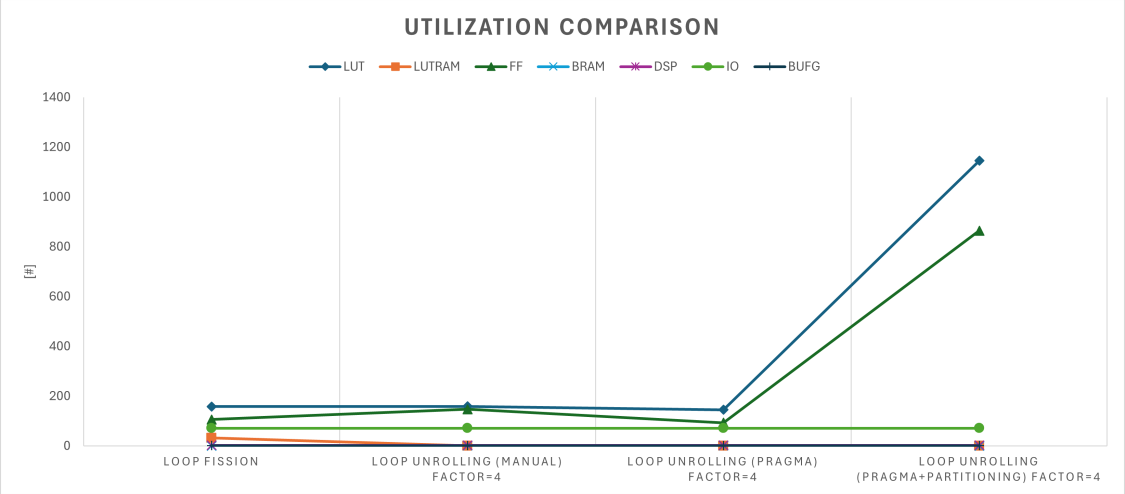


Figure 25: Vivado Loop Unrolling Factor=4 Utilization Plot

Per quanto riguarda il report di timing, è possibile notare, analogamente a quanto successo per l'unrolling di fattore pari a 2, una diminuzione della maximum clock frequency in corrispondenza della soluzione basata su loop unrolling automatico con partizionamento.

Solution	**Cycles** [#]	**Clock Constraint** [ns]	**WNS** [ns]	**Maximum Clock Frequency** [MHz]
Manuale	59	10	4.257	174.1250218
Automatico	61	10	4.33	176.366843
Automatico con Partitioning	59	10	3.097	144.8645516

Table 66: Vivado Loop Unrolling Factor=4 Solution Timing Report

Per quanto riguarda, invece, il numero di cicli di clock per garantire un risultato in uscita, in questo caso, per le tre soluzioni basate su unrolling di fattore pari a 4, si è ottenuto un valore minore rispetto alla soluzione basata su scissione del loop.

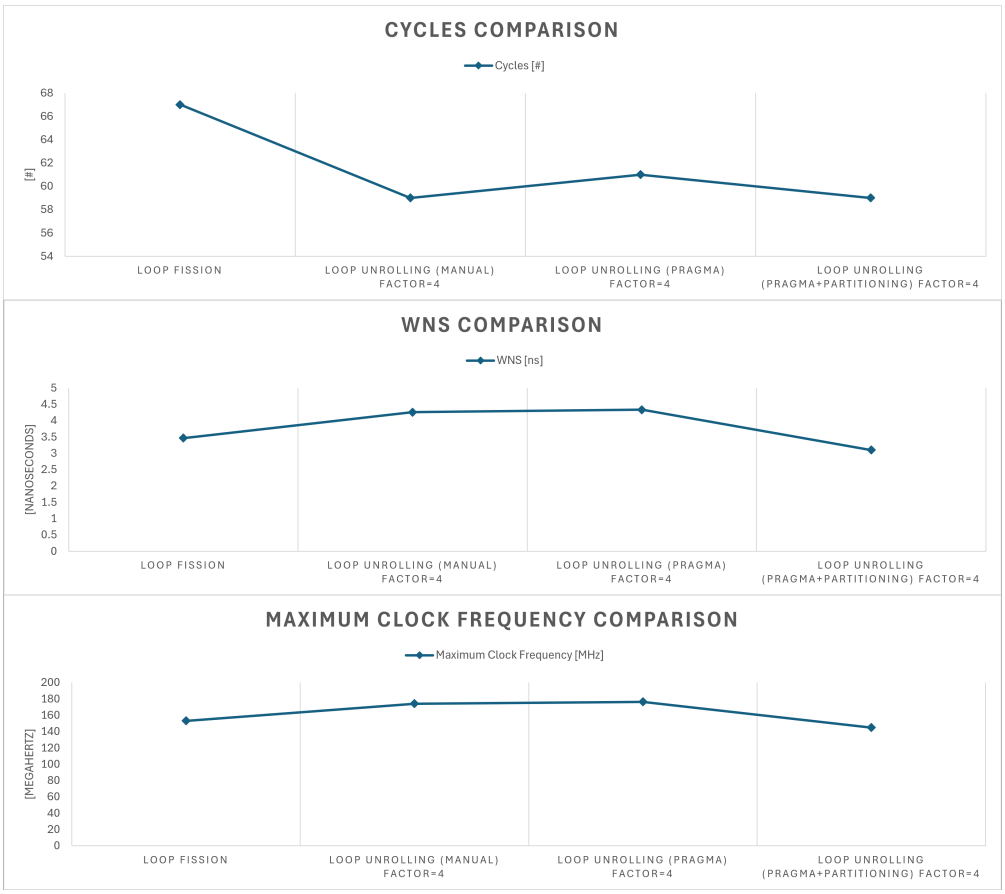


Figure 26: Vivado Loop Unrolling Factor=4 Timing Plot

Analogamente all'unrolling di fattore pari a 2, anche in questo caso in corrispondenza della soluzione hardware basata su unrolling e partizionamento si ha un notevole aumento del contributo di potenza dinamica relativo al *Clocks* comportando un aumento della potenza dinamica totale.

Solution	**BRAM**	**Clock Enable**	**Clocks**	**DSP**	**Logic**	**Set/ Reset**	**Data**
Manuale	1.32976193	0.382625905	0.253616716	0.921033497	0.543549308	0.010887122	0.585376518
Automatico	1.23103871	0.106978332	0.972227077	0.247065967	0.258892891	0.002585625	0.41881192
Automatico con Partitioning	0	0.328624592	2.560390625	0.298048515	1.146363211	0.0030065	1.324957004

Table 67: Vivado Loop Unrolling Factor=4 Solution Dynamic Power Report [mW]

Solution	**Dynamic Total**
Manuale	4.026850996
Automatico	3.237600523
Automatico con Partitioning	5.661390447

Table 68: Vivado Loop Unrolling Factor=4 Solution Dynamic Power Report [mW]

Solution	**Energy Single Operation**
Manuale	40.26850996
Automatico	32.37600523
Automatico con Partitioning	56.61390447

Table 69: Vivado Loop Unrolling Factor=4 Solution Energy Single Operation Report [pJ]

Bisogna notare un aspetto molto interessante. La potenza dinamica totale e l'energia per singola operazione associata alla soluzione hardware basata su loop unrolling automatico di fattore pari a 4 risultano essere pressocché le medesime di quelle relative all'unrolling automatico di fattore pari a 2. Questo potrebbe essere dovuto a ottimizzazione effettuate dal tool tramite la direttiva proprietaria dell'unrolling.

5.6 Loop Pipelining Solution

Il pipelining consente di eseguire operazioni in modo simultaneo: ogni fase di esecuzione non deve completare tutte le operazioni prima di iniziare quella successiva. Pertanto, questo approccio permette di scindere le operazioni complesse in più operazioni semplici. In questo modo si può far lavorare l'architettura con dati temporalmente differenti. Ad esempio, tenendo conto della figura sotto allegata, questa funzione senza pipeline dovrebbe eseguire tutte le iterazioni previste in maniera sequenziale e, quindi, eseguendo read, compute, write, read, compute, write e così via. Considerando, invece, un approccio mediante pipeline, quello che succede è che alla prima iterazione viene eseguita la prima micro-operazione, alla seconda iterazione viene eseguita la seconda micro-operazione e in contemporanea si può eseguire la prima micro-operazione che nel caso precedente senza pipeline era prevista per il quarto ciclo di clock. L'idea è quella di far lavorare l'architettura sfruttando la scissione della logica complessa in più micro-operazioni.

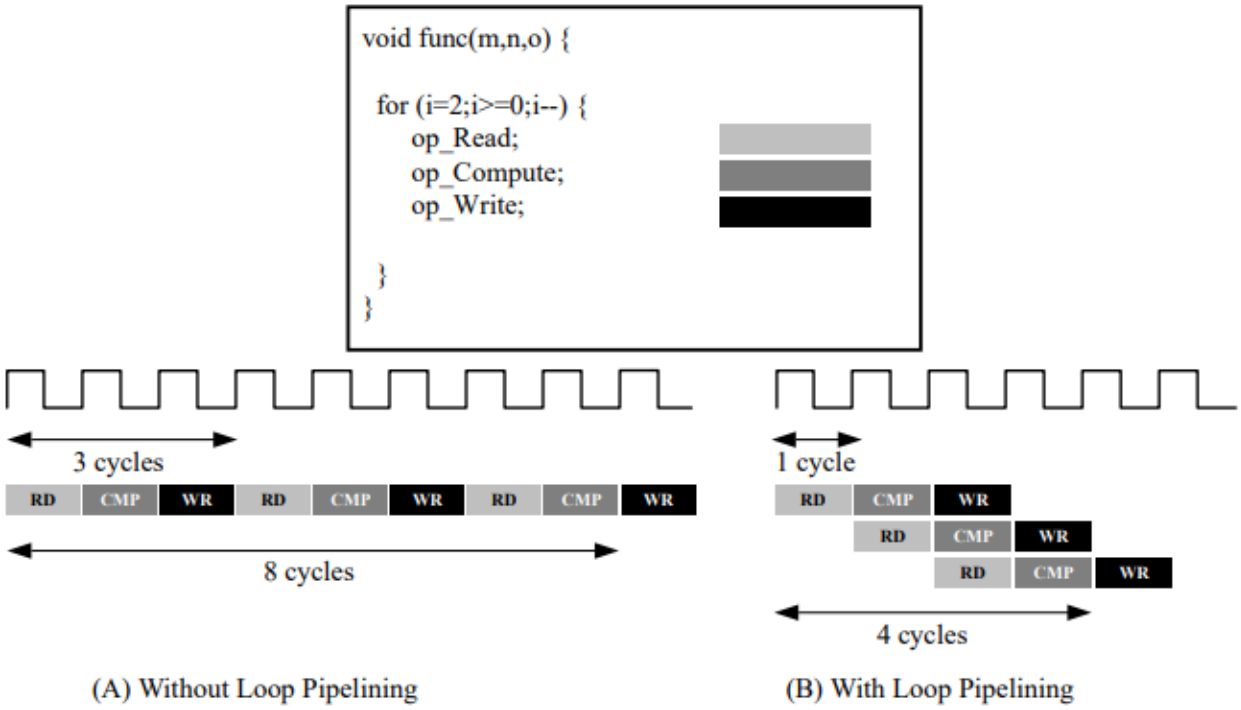


Figure 27: HLS Loop Pipelining

In particolare, è possibile evidenziare due parametri fondamentali per questa tipologia di approccio:

- **Iteration Latency (IL)**
  Numero di cicli che servono per eseguire un'iterazione del corpo del ciclo.
- **Initiation Interval (II) := Throughput**
  Numero di cicli di clock fino alla prossima iterazione del loop.

Idealmente, ci si aspetta un throughput pari a 1 perchè corrisponderebbe al caso migliore per il pipelining. Ovviamente non sempre è possibile ottenere un throughput pari a 1 ma in questo caso l'obiettivo è quello di ottenerlo.
Il pipelining in HLS è possibile implementarlo mediante una direttiva proprietaria che permette, inoltre, di specificare l'Initiation Interval desiderato. Tanto è vero che all'interno del report di sintesi viene specificato l'II target, cioè quello specificato nella direttiva, e l'II raggiunto dal tool. Teoricamente, si dovrebbe ottenere II_target=II_achieved. Se così non fosse allora il tool non è riuscito a raggiungere l'obiettivo prefissato. Pertanto, si potrebbe modificare il codice per vedere se effettivamente il problema può essere risolto nella

seguente maniera. Se così non fosse, evidentemente non dipende dal codice ma dal tipo di applicazione che si sta cercando di implementare che non lo permette.

```
#include "definitions.h"

void firConvolutionLoopPipelining(samplesType inputFilter, samplesType* outputFilter) {
    coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
    static samplesType shiftRegister[SIZE];
    accType accumulator;
    accumulator = 0;
    int i;
    loop: for( i=SIZE-1; i>=0; --i ) {
        #pragma HLS pipeline II=1
        #pragma HLS array_partition variable=shiftRegister complete
        if( i==0 ) {
            accumulator += inputFilter * coefficientsFilter[0];
            shiftRegister[0] = inputFilter;
        } else {
            shiftRegister[i] = shiftRegister[i-1];
            accumulator += shiftRegister[i] * coefficientsFilter[i];
        }
    }
    *outputFilter = accumulator;
}
```

In questo caso, la soluzione hardware presa come riferimento è stata quella iniziale non ottimizzata. In particolare, all'interno del loop è stata definita sia la direttiva di pipeline, specificando un II=1, sia la direttiva di partizionamento. Nello specifico, è stato considerato il pragma di partitioning al fine di realizzabilità della soluzione in questione. Questo è dovuto al fatto che, senza considerare tale direttiva, si otterrebbe un II_achieved differente da un II_target. Infatti, utilizzare lo shift register vuol dire non poter accedere ai dati intermedi ma soltanto a quelli finali. Pertanto, per far fronte a questo problema si potrebbe pensare di realizzare tali shift register mediante Flip Flop (tramite la direttiva di partizionamento).
Effettuando la sintesi si ottengono i seguenti report.

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.510	1.25

Table 70: HLS Loop Pipelining Solution Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
15	15	15	15

Table 71: HLS Loop Pipelining Solution Latency Summary (clock cycles)

Si può notare come la latenza totale del loop sia notevolmente diminuita rispetto alla soluzione iniziale non ottimizzata. Inoltre, si può evidenziare come l'Initiation Interval raggiunto sia il medesimo di quello target, cioè pari a 1. Questo è stato possibile per i motivi precedentemente citati.
In particolare, considerando il valore di Iteration Latency, il valore del Trip Count e il valore di Initiation Interval, è possibile ricavare il valore della latenza relativa al loop:

$$IL + [(N-1) * II] - 1 = 4 + [(11-1) * 1] - 1 = 4 + [10 * 1] - 1 = 4 + 10 - 1 = 13 \tag{7}$$

Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
Name	min	max	min	max	achieved	target	
- loop	13	13	4	4	1	1	11

Table 72: HLS Loop Pipelining Solution Latency Loops Summary

Per quanto riguarda l'utilizzazione stimata delle risorse, si può notare come il numero di FF sia aumentato rispetto alla soluzione non ottimizzata dal momento che è stato utilizzato il pragma di partizionamento tale da partizionare shiftRegister in diverse strutture da un certo numero di FF ciascuna anziché utilizzare BRAM.

Name	BRAM_18K	DSP48E	FF	LUT
DSP	-	-	-	-
Expression	-	4	0	480
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	10	2
Multiplexer	-	-	-	75
Register	-	-	700	64
Total	0	4	710	621
Available	280	220	106400	53200
Utilization (%)	0	1	∼0	1

Table 73: HLS Loop Pipelining Solution Utilization Estimates Summary

In particolare, si può notare come la direttiva proprietaria di partitioning ha partizionato shiftRegister in 10 strutture da 32 FF ciascuna.

Name	FF	LUT	Bits	Const Bits
accumulator_reg_114	32	0	32	0
ap_CS_fsm	3	0	3	0
ap_enable_reg_pp0_iter0	1	0	1	0
ap_enable_reg_pp0_iter1	1	0	1	0
ap_enable_reg_pp0_iter2	1	0	1	0
ap_enable_reg_pp0_iter3	1	0	1	0
ap_phi_reg_pp0_iter1_p_pn_reg_138	32	0	32	0
ap_phi_reg_pp0_iter2_p_pn_reg_138	32	0	32	0
ap_phi_reg_pp0_iter3_p_pn_reg_138	32	0	32	0
coefficientsFilter1_1_reg_458	10	0	10	0
i_reg_127	5	0	5	0
shiftRegister_0	32	0	32	0
shiftRegister_1	32	0	32	0
shiftRegister_2	32	0	32	0
shiftRegister_3	32	0	32	0
shiftRegister_4	32	0	32	0
shiftRegister_5	32	0	32	0
shiftRegister_6	32	0	32	0
shiftRegister_7	32	0	32	0
shiftRegister_8	32	0	32	0
shiftRegister_9	32	0	32	0
shiftRegister_load_p_reg_453	32	0	32	0
tmp_1_reg_426	1	0	1	0
tmp_2_reg_417	32	0	32	0
tmp_4_reg_430	4	0	4	0
tmp_6_reg_463	32	0	32	0
tmp_reg_422	1	0	1	0
tmp_1_reg_426	64	32	1	0
tmp_reg_422	64	32	1	0
Total	700	64	574	0

Table 74: HLS Loop Pipelining Solution Register Detail Estimates Summary

Effettuando la C/RTL Cosimulation ed Export RTL si ottengono i seguenti report.

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	15	15	16	15	15	16

Table 75: HLS Loop Pipelining Solution C/RTL Cosimulation Summary

Resource	VHDL
SLICE	204
LUT	311
FF	455
DSP	2
BRAM	0
SRL	0

Table 76: HLS Loop Pipelining Solution Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	5.745
CP achieved post-implementation	6.512

Table 77: HLS Loop Pipelining Solution Export RTL Final Timing

Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
1 create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

Si può notare come, rispetto alla soluzione iniziale non ottimizzata, si è ottenuto un aumento di circa il 13% delle LUT e un aumento di circa il 186% dei FF.

LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
311	0	458	0	2	71	1

Table 78: Vivado Loop Pipelining Solution Utilization Report [#]

Inoltre, si può evidenziare come la maximum clock frequency sia aumentata di circa $15MHz$ rispetto alla soluzione non ottimizzata di riferimento. Tanto è vero che il WNS, relativo alla soluzione realizzata mediante approccio di pipelining, risulta essere leggermente maggiore.

Cycles [#]	**Clock Constraint** [ns]	**WNS** [ns]	**Maximum Clock Frequency** [Mhz]
16	10	4.208	172.6519337

Table 79: Vivado Loop Pipelining Solution Timing Report

In particolare, si può notare come, rispetto alla soluzione hardware iniziale non ottimizzata, i contributi di potenza dinamica sono aumentati tale da far aumentare l'energia per singola operazione. Nello specifico, si ha un aumento del contributo *Clocks* di circa il 46% dovuto principalmente all'aumento delle risorse associato alla solution in questione.

BRAM	Clock Enable	Clocks	DSP	Logic	Set/Reset [mW]	Data
0	0.196236724	1.778833685	0.814738101	1.275097136	0.012051522	1.206569896

Table 80: Vivado Loop Pipelining Solution Dynamic Power Report [mW]

Dynamic Total
5.283527065

Table 81: Vivado Loop Pipelining Solution Dynamic Power Report [mW]

Energy Single Operation
52.83527065

Table 82: Vivado Loop Pipelining Solution Energy Single Operation Report [pJ]

5.7 Bitwidth Optimization Solution

In questa sezione verrà discussa il possibile sovradimensionamento riguardo i tipi di dato relativi alle variabili utilizzate nelle varie architetture proposte. Effettivamente, nelle soluzioni hardware precedentemente citate, è stato principalmente utilizzato il tipo di dato *int* anche in casi in cui bastava utilizzare un numero di bit notevolmente minore. Quello che si potrebbe fare è utilizzare tipi di dato aventi un numero di bit inferiore come, ad esempio, *char* o *short*. In alcuni casi, potrebbe accadere che il numero di bit richiesto è ulteriormente inferiore a questi tipi di dato disponibili negli ambienti *C-like*. A tale proposito, nel contesto dei linguaggi ad alto livello C e C++, è disponibile una libreria, **ap_int.h** che permette di modellare il tipo di dato *int* scegliendo opportunamente il numero di bit da riservare per una determinata variabile. Pertanto, si potrebbe implementare una *Bitwidth Optimization Solution* dove venga efficientemente allocato un numero di bit per le variabili al fine di ridurre l'utilizzazione delle risorse corrispondente. Ovviamente, utilizzare un numero ben preciso di bit per ogni variabile, vuol dire avere una soluzione custom per un certo range di bit e, nel caso in cui si debba utilizzare, ad esempio, un input che non sia incluso nel range di bit previsto, si dovrebbe, pertanto, implementare una nuova architettura con un numero di bit differente per le variabili corrispondenti. Ovviamente, in generale, si è considerato un numero di bit maggiorato di un'unità, dal momento che *ap_int.h* è una libreria che prevede un tipo di dato con segno.

```
#include "definitions.h"
#include "ap_int.h"

void firConvolutionBitwidthOptimization(ap_int<8> inputFilter, ap_int<18+SIZE>* outputFilter
    ) {
  ap_int<10> coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
  static ap_int<8> shiftRegister[SIZE];
  ap_int<18+SIZE> accumulator;
  accumulator = 0;
  ap_int<5> i;
  loop: for( i=SIZE-1; i>0; --i ) {
    shiftRegister[i] = shiftRegister[i-1];
    accumulator += shiftRegister[i] * coefficientsFilter[i];
  }
  accumulator += inputFilter * coefficientsFilter[0];
  shiftRegister[0] = inputFilter;
  *outputFilter = accumulator;
}
```

In particolare, la soluzione hardware proposta prevede un input ad 8 bit ed un output corrispondente a $18 + SIZE = 18 + 11 = 29$ bit. La scelta del numero di bit relativi all'*inputFilter* è arbitrario. Ovviamente, al variare di tale valore, il numero di bit previsto per le variabili interne cambierà di conseguenza.
Per quanto riguarda, invece, il numero di bit previsto per gli elementi presenti in *coefficientsFilter*, sono stati considerati il valore massimo e minimo previsti all'interno dell'array. Riguardo l'array *shiftRegister*, la bitwidth scelta è la medesima dell'input dal mommento che tale struttura dati viene utilizzata soltanto per lo shifting dei valori in ingresso all'architettura. Viceversa, la variabile *accumulator* prevede un numero di bit uguale a quello dell'uscita del metodo. Infatti, tale variabile prevede, per ogni iterazione all'interno del ciclo for, una moltiplicazione e una somma. In particolare, la moltiplicazione richiede un numero di bit fisso, cioè 8+10 bit, mentre la somma, che viene effettuata sempre sulla medesima variabile in questione, richiede un numero di bit sempre maggiore per ogni iterazione. Quindi, tenendo conto che le iterazioni sono pari a $SIZE = 11$, allora il numero di bit previsto per la variabile *accumulator* è pari a $8 + 10 + SIZE = 8 + 10 + 11 = 29$. Infine, per quanto riguarda il numero di bit associato alla variabile indice i è relazionato al numero di iterazioni da effettuare che sono pari a $SIZE = 11$.
In particolare, l'architettura di riferimento utilizzata è basata sulla *Code Hoisting Solution*. Pertanto, considerando i risultati ottenuti in corrispondenza della soluzione hardware citata ed effettuando la sintesi, la C/RTL Cosimulation e l'Export RTL della *Bitwidth Optimization Solution*, è possibile effettuare i seguenti confronti.
Si può notare come, dopo aver effettuato la sintesi, il numero di risorse, rispetto alla soluzione basata su Code Hoisting, presenti una diminuzione di circa il 65% dei FF e di circa il 55% delle LUT. Il numero dei DSP stimati e diminuito di 2 unità. Inoltre, si può notare come la latenza stimata è diminuita di 10 cicli rispetto alla soluzione presa come riferimento. Ovviamente il Trip Count è il medesimo dal momento

che l'architettura proposta nella *Bitwidth Optimization Solution* è equivalente a quella proposta nel *Code Hoisting Solution*.

Solution	**Clock**	**Target**	**Estimated**	**Uncertainty**
Code Hoisting	ap_clk	10.00	8.510	1.25
Bitwidth Opt.	ap_clk	10.00	6.38	1.25

Table 83: HLS Bitwidth Optimization Solution Timing Summary (ns)

Solution	**Latency**		**Interval**	
	min	max	min	max
Code Hoisting	42	42	42	42
Bitwidth Opt.	31	31	31	31

Table 84: HLS Bitwidth Optimization Solution Latency Summary (clock cycles)

Solution	Loop Name	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
		min	max	min	max	achieved	target	
Code Hoisting	- loop	40	40	4	4	-	-	10
Bitwidth Opt.	- loop	30	30	3	3	-	-	10

Table 85: HLS Bitwidth Optimization Solution Latency Loops Summary

Solution	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
Bitwidth Opt.	0	4	230	240
Code Hoisting	0	2	81	107

Table 86: HLS Bitwidth Optimization Solution Utilization Estimates [#]

Solution	RTL	Status	**Latency**			**Interval**		
			min	avg	max	min	avg	max
Code Hoisting	VHDL	Pass	42	42	43	42	42	43
Bitwidth Opt.	VHDL	Pass	31	31	32	31	31	32

Table 87: HLS Bitwidth Optimization Solution C/RTL Cosimulation Report

Analizzando il report dell'Export RTL si può notare come l'utilizzazione delle risorse è notevolmente diminuita. Tanto è vero che il numero delle LUT e dei FF è diminuito di circa l'82%. Questo decremento dell'utilizzazione delle risorse prevista è dovuto al fatto che nella *Bitwidth Optimization Solution* sono stati utilizzati dei tipi di dato con un numero di bit custom per la soluzione in questione.

Solution	**SLICE**	**LUT**	**FF**	**DSP**	**BRAM**	**CP required**	**CP achieved post-synthesis**	**CP achieved post-implementation**
Code Hoisting	75	270	131	2	0	10	5.745	6.847
Bitwidth Opt.	15	50	24	2	0	10	5.745	5.692

Table 88: HLS Bitwidth Optimization Solution Export RTL Report

6 AXI4-Stream

In questa sezione verrà discussa una possibile implementazione di una soluzione hardware basata su un approccio non ottimizzato (medesimo a quello iniziale) in cui è stato considerato il protocollo AXI4-Stream. Pertanto, tale solution non è da considerare come ottimizzazione rispetto alle precedenti ma come caso d'uso dello standard in corrispondenza di una delle solution precedentemente citate. In particolare, come già citato per semplicità si farà riferimento alla *Unoptimized Solution*.
Il protocollo AXI4-Stream è utilizzato come interfaccia standard per collegare componenti che desiderano scambiare dati. L'interfaccia può essere utilizzata per collegare un singolo master, che genera dati, a un singolo slave, che riceve dati. Il protocollo può essere utilizzato anche per collegare un numero maggiore di componenti master e slave. Lo standard supporta flussi di dati multipli che utilizzano lo stesso insieme di bus condivisi, consentendo di costruire un'interconnessione generica. In particolare, con l'interfaccia Stream si garantisce un nuovo dato in input ad ogni colpo di clock. In questo caso specifico, quindi, ci si aspetta che lo standard garantisca 11 dati in input per 11 colpi di clock così da essere processati ed essere messi in uscita.

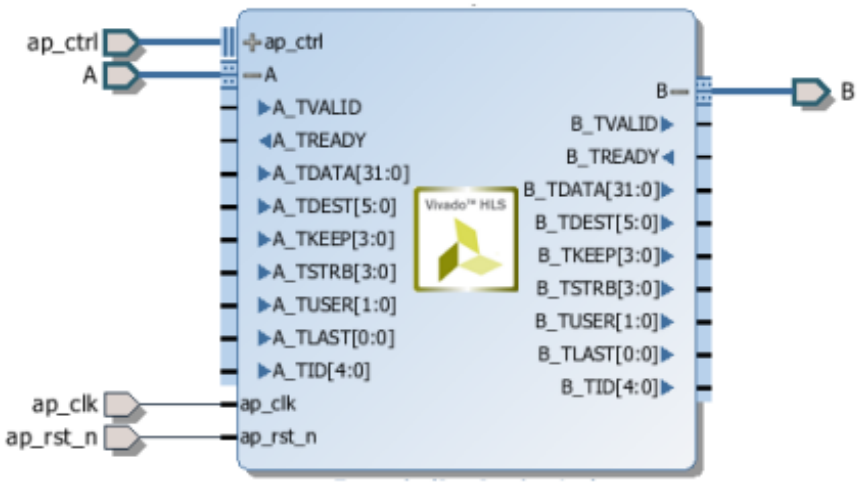


Figure 28: AXI4-Stream Protocol

Nello specifico, il tipo di pacchetto includerà il segnale corrispondente al dato effettivo e tutti gli altri segnali che servono per lo standard corrispondente.

```
#include "definitions.h"

void firConvolutionAXI(hls::stream<AXI_TYPE> &inputStreamFilter, hls::stream<AXI_TYPE> &
    outputStreamFilter) {
  #pragma HLS INTERFACE ap_ctrl_none port=return
  #pragma HLS INTERFACE axis port=inputStreamFilter
  #pragma HLS INTERFACE axis port=outputStreamFilter
  AXI_TYPE inFilter, outFilter;
  coeffsType coefficientsFilter[SIZE] = {53, 0, -91, 0, 313, 500, 313, 0, -91, 0, 53};
  static samplesType shiftRegister[SIZE];
  while( inputStreamFilter.read_nb(inFilter) ) {
    #pragma HLS pipeline II=1
    accType accumulator;
    accumulator = 0;
    loop: for( int i=SIZE-1; i>=0; i-- ) {
      if( i==0 ) {
        accumulator += inFilter.data*coefficientsFilter[0];
        shiftRegister[0] = inFilter.data;
      } else {
        shiftRegister[i] = shiftRegister[i-1];
        accumulator += shiftRegister[i] * coefficientsFilter[i];
      }
```

```
    }
    outFilter.data=(accumulator);
    outFilter.strb = 15;
    outFilter.keep = 15;
    outFilter.user = 1;
    outFilter.last = inFilter.last;
    outFilter.id = 0;
    outFilter.dest = 0;
    outputStreamFilter.write(outFilter);
  }
}
```

In particolare, *inputStreamFilter* e *outputStreamFilter* sono rispettivamente lo stream di ingresso e lo stream di uscita all'architettura.
Per quanto riguarda, invece, le direttive specificate inizialmente, la prima fa riferimento al fatto che il tool non deve sintetizzare i segnali *ap_ctrl* tranne quella relativa a *inputStreamFilter* e *outputStreamFilter* e i segnali relativi allo standard, mentre la seconda e la terza rispettivamente fanno riferimento al fatto che *inputStreamFilter* e *outputStreamFilter* devono essere mappati secondo lo standard Stream.
Inoltre, si può notare come, all'interno del ciclo while, sia stato specificato un approccio *non blocking*, cioè in riferimento al fatto che non si sa a priori la quantità di dati entranti nel buffer.
Ovviamente, bisogna specificare che il segnale di *TVALID* sarà pari a 1 soltanto nel momento in cui si ha il risultato finale, cioè in corrispondenza dell'accumulo finale.
Effettuando la sintesi, la C/RTL Cosimulation e l'Export RTL si ottengono i seguenti report.

Clock	**Target**	**Estimated**	**Uncertainty**
ap_clk	10.00	8.742	1.25

Table 89: HLS AXI Solution Timing Summary (ns)

Latency		**Interval**	
min	max	min	max
?	?	?	?

Table 90: HLS AXI Solution Latency Summary (clock cycles)

Si può notare come la latenza totale e quella relativa al ciclo for presentano un valore pari a *?*, cioè un valore di latenza non noto, dovuto al fatto che l'architettura presenta un approccio *non blocking*. Inoltre, dal momento che è stata prevista una direttiva di pipeline, si può notare come sia stato raggiunto un Initiation Interval pari a 1.

Loop	**Latency**		**Iteration Latency**		**Initiation Interval**		**Trip Count**
Name	min	max	min	max	achieved	target	
- loop	?	?	4	4	1	1	?

Table 91: HLS AXI Solution Latency Loops Summary

Name	**BRAM_18K**	**DSP48E**	**FF**	**LUT**
DSP	-	-	-	-
Expression	-	14	0	380
FIFO	-	-	-	-
Instance	-	-	-	-
Memory	0	-	-	-
Multiplexer	-	-	-	171
Register	-	-	767	32
Total	0	14	767	583
Available	280	220	106400	53200
Utilization (%)	0	6	∼0	1

Table 92: HLS AXI Solution Utilization Estimates Summary

RTL	Status	Latency			Interval		
		min	avg	max	min	avg	max
VHDL	Pass	13	13	13	NA	NA	NA

Table 93: HLS AXI Solution C/RTL Cosimulation Summary

Resource	VHDL
SLICE	231
LUT	739
FF	699
DSP	0
BRAM	0
SRL	0

Table 94: HLS AXI Solution Export RTL Resource Usage

Timing	VHDL
CP required	10.000
CP achieved post-synthesis	7.269
CP achieved post-implementation	8.202

Table 95: HLS AXI Solution Export RTL Final Timing

Pertanto, importando l'IP in Vivado e impostando un clock constraint pari a 10ns è possibile analizzare i seguenti report di risorse, timing, potenza dinamica ed energia per singola operazione.

```
create_clock -period 10.000 -name myclk -waveform {0.000 5.000} [get_ports ap_clk]
```

In particolare, si può effettuare un confronto rispetto alla soluzione iniziale non ottimizzata, considerata come riferimento per questa implementazione, per valutare gli effetti dell'introduzione del protocollo AXI nell'architettura in questione. Si può notare un aumento dell'utilizzazione delle LUT circa pari al 169% e un aumento del numero di FF circa pari al 337%.

LUT	LUTRAM	FF	BRAM	DSP	IO	BUFG
739	0	699	0	0	93	1

Table 96: Vivado AXI Solution Utilization Report [#]

Si può evidenziare come il numero di cicli previsto per questa soluzione sia pari a 11 come ipotizzato inizialmente.

Cycles [#]	Clock Constraint [ns]	WNS [ns]	Maximum Clock Frequency [Mhz]
11	10	1.369	115.8614297

Table 97: Vivado AXI Solution Timing Report

Per quanto riguarda la potenza dinamica totale e l'energia per singola operazione, si può notare un aumento rispetto alla soluzione dove non è presenta la logica di controllo relativa al protocollo AXI. In particolare, notevoli aumenti si hanno in corrispondenza dei contributi di *Clocks*, *Logic* e *Data*.

BRAM	Clock Enable	Clocks	DSP	Logic	Set/Reset [mW]	Data
0	0.361682673	2.409646753	0	7.963255048	0.008282737	5.53872576

Table 98: Vivado AXI Solution Dynamic Power Report [mW]

Dynamic Total
16.28159297

Table 99: Vivado AXI Solution Dynamic Power Report [mW]

Energy Single Operation
162.8159297

Table 100: Vivado AXI Solution Energy Single Operation Report [pJ]

7 Conclusions

Qui di seguito verranno riportati i plot relativi alle caratteristiche più importanti, relative alle implementazioni effettuate in Vivado, di tutte le solutioni hardware analizzate.

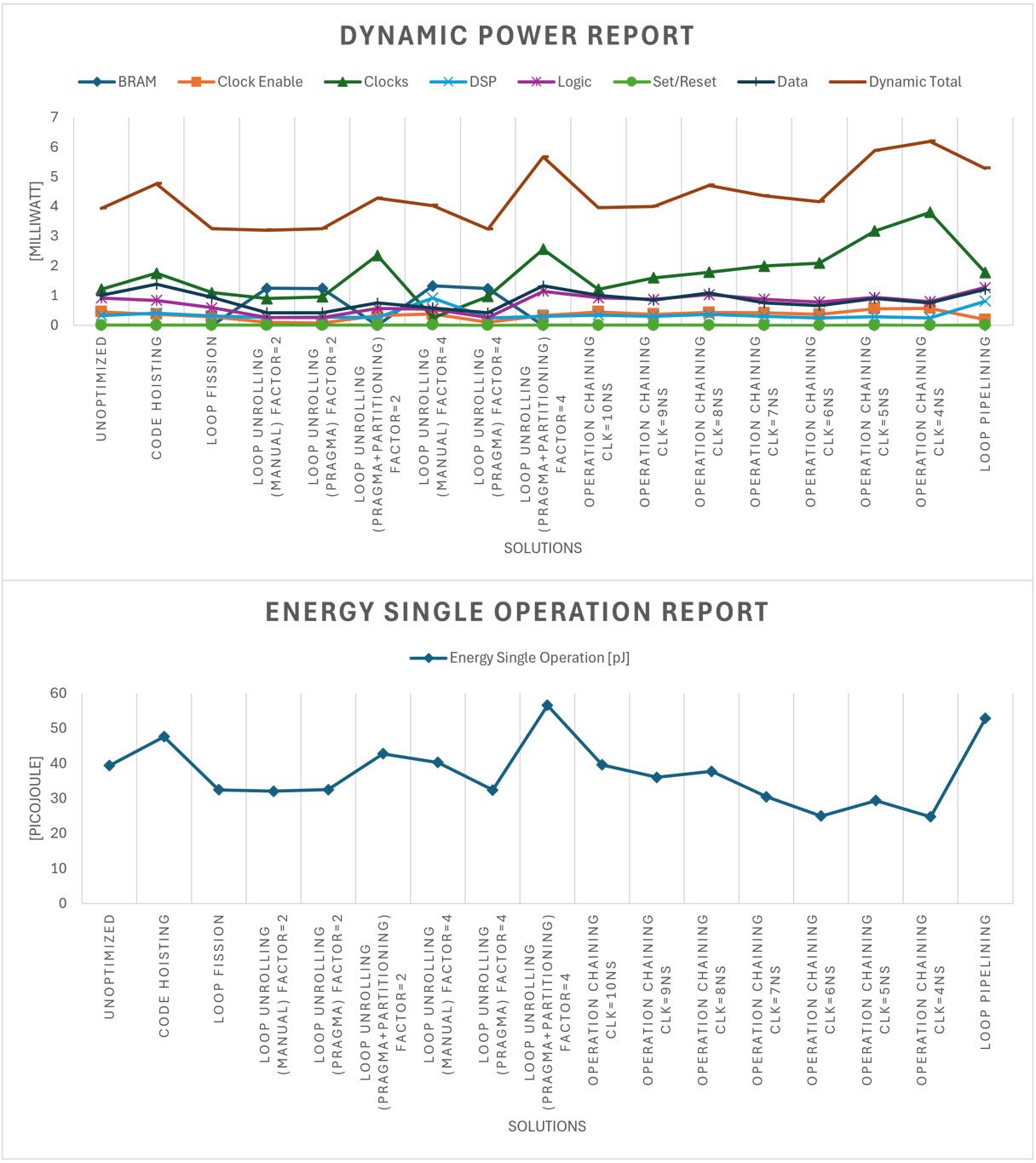


Figure 29: Vivado Power Report

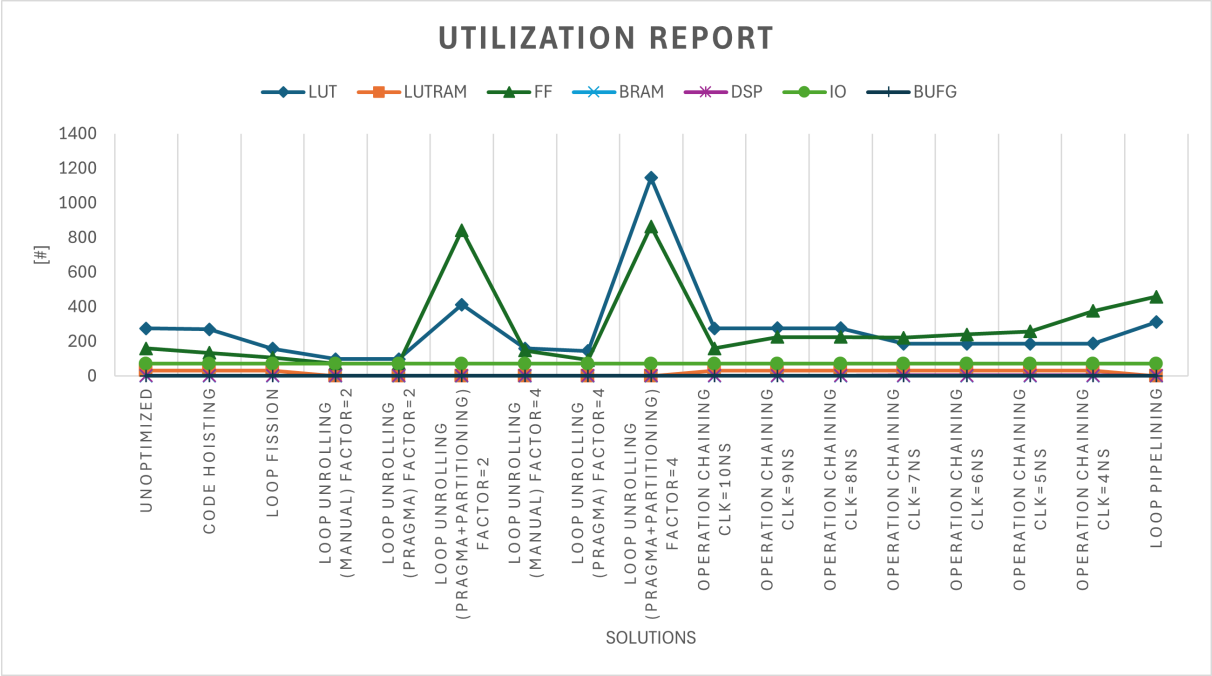


Figure 30: Vivado Utilization Report

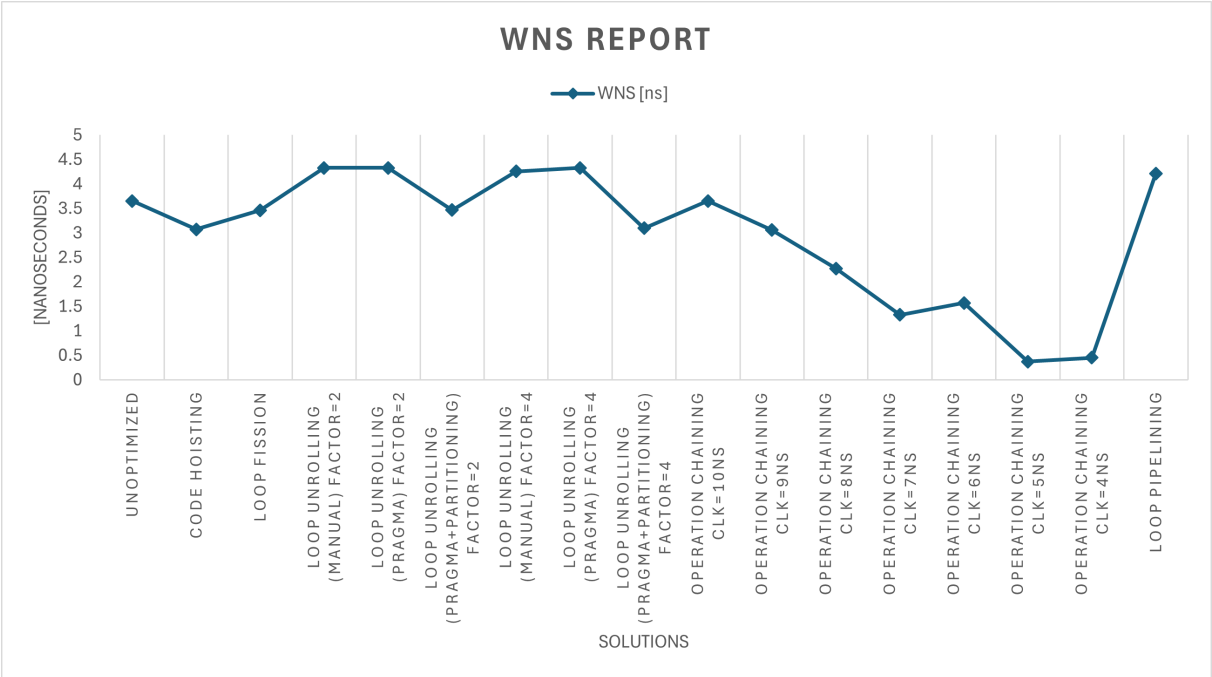


Figure 31: Vivado WNS Report

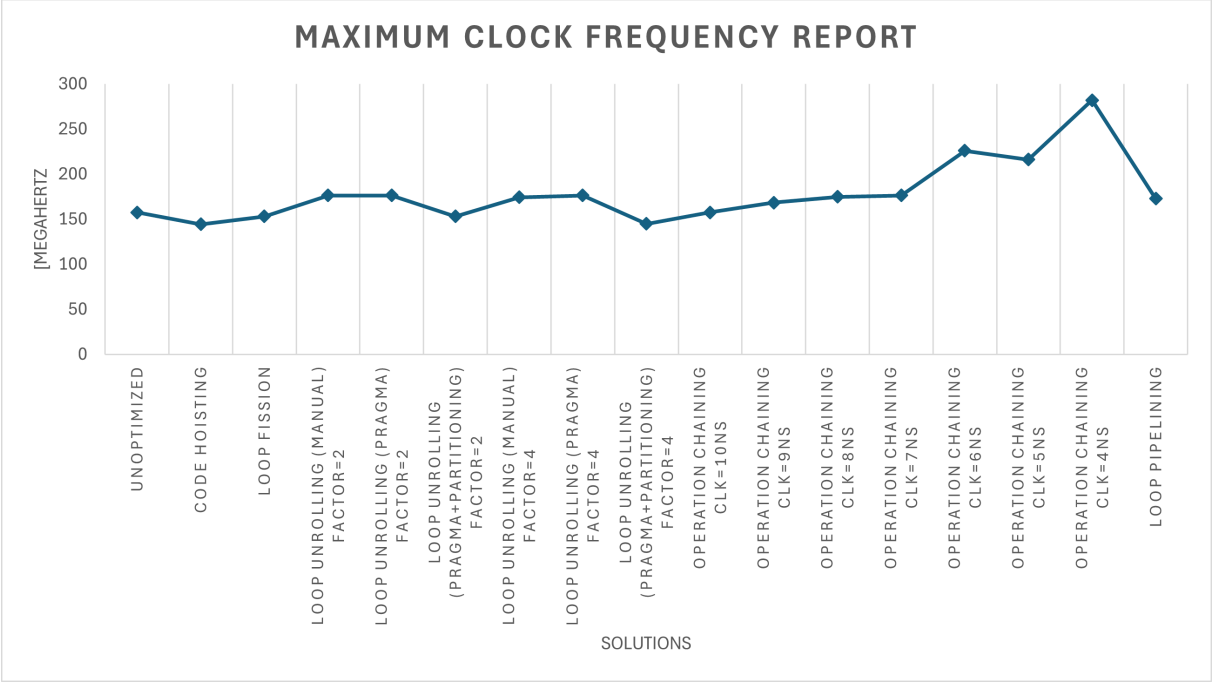


Figure 32: Vivado Maximum Clock Frequency Report