



Politecnico di Torino

Analysis and Design of an FIR filter obtained with High Level Synthesis

Candidate: Federico Parodi

Supervisors: Mario Casu

Luciano Lavagno

A mia madre e a mio padre

Un ringraziamento speciale va a tutta la mia famiglia che mi ha supportato durante questi 5 anni, mia mamma Costanza e mio papà Marco senza i quali niente sarebbe stato possibile, dal primo Open-day al Politecnico, fino a questo importante traguardo.

Ringrazio i miei fratelli Eugenia, Beatrice, Alessandra e Filippo per l'affetto che dimostrano ogni giorno. Spero, nonostante tutti i miei difetti, di aver lasciato un buon esempio, *Duc in altum*.

Fonte di esempio per me sono stati nonno Giorgio e nonna Laura, nonno Elio e nonna Giò, zia Luisa a cui va un grande abbraccio.

Vorrei ringraziare dal profondo colei che durante questi anni, a causa della lontananza, ha più sofferto, nonostante questo mi è sempre stata vicino, incoraggiandomi a non mollare mai per raggiungere il mio sogno. Giulia, la dimostrazione che *Amor vincit omnia*.

Infine ma non per importanza i miei amici con i quali ho condiviso vittorie e sconfitte lungo il percorso, cinque anni che sono volati e che non dimenticherò mai, anche grazie a voi, in particolare Simone, come un fratello, Francesco, Ymer, Sergio e Jovana.

Contents

| | | |
|----------|---|-----------|
| 1 | Introduction | 1 |
| 1.1 | Work introduction | 1 |
| 1.2 | Definition of High Level Synthesis | 2 |
| 1.3 | Brief history of HLS | 5 |
| 1.4 | Catapult HLS | 9 |
| 1.4.1 | Catapult flow | 10 |
| 2 | State of Art | 17 |
| 2.1 | Previous works on filter synthesis through HLS | 17 |
| 3 | Design of filter | 21 |
| 3.1 | Upsampling FIR | 21 |
| 3.2 | The original Matlab model | 22 |
| 3.3 | Translation from Matlab to C++ of the algorithm | 24 |
| 3.4 | The synthesizable filter | 25 |
| 3.5 | Bit-Accurate filter | 27 |
| 3.6 | The testbench | 30 |
| 3.7 | Results | 33 |
| 4 | Architecture optimizations | 34 |
| 4.1 | Performances of SM filter | 34 |
| 4.2 | Code changes | 35 |
| 4.3 | The ac_channel Class Definition | 40 |
| 4.4 | Results given by the optimized code | 41 |
| 5 | Catapult optimizations | 47 |
| 5.1 | Recap of the GUI parameters | 47 |
| 5.1.1 | Clusters | 48 |
| 5.2 | Unrolling of the SHIFT LOOP | 48 |

| | | |
|----------|--|-----------|
| 5.3 | Pipelining of MAC Loops | 52 |
| 5.3.1 | Testbench and verification | 55 |
| 5.3.2 | Back annotation and switching activity | 57 |
| 5.4 | Solutions with low latency | 59 |
| 6 | FPGA implementation | 62 |
| 6.1 | AXI bus | 63 |
| 6.2 | From Catapult to Vivado | 63 |
| 6.3 | Block diagram generation | 65 |
| 6.3.1 | Filter IP creation | 66 |
| 6.3.2 | FIFO IP creation | 67 |
| 6.3.3 | IP with Filter and FIFOs | 68 |
| 6.3.4 | IP with AXI interface | 69 |
| 6.3.5 | Final block diagram | 72 |
| 7 | Conclusions | 74 |
| A | C++ codes | 77 |
| B | VHDL codes | 82 |
| C | Testbench | 86 |
| D | C application for the FPGA | 89 |

List of Figures

| | | |
|------|---|----|
| 1.1 | Gajski-Kuhn chart [1] | 2 |
| 1.2 | Gajski-Kuhn chart [1] | 3 |
| 1.3 | HLS vs Logic synthesis [13] | 4 |
| 1.4 | Different type of design in the history [12] | 5 |
| 1.5 | The original CMU Design System [4] | 6 |
| 1.6 | High Level Synthesys deployment 2011 [9] | 7 |
| 1.7 | Types of hardware being designed 2014 [11] | 8 |
| 1.8 | Catapult High-Level Synthesis Platform [15] | 9 |
| 1.9 | NVIDIA Closes Design Complexity Gap with High-Level Synthesis [17] | 10 |
| 1.10 | The Catapult design flow [14] | 10 |
| 1.11 | The Catapult design flow | 11 |
| 1.12 | Hierarchy settings | 12 |
| 1.13 | Architecture constraints on Module | 13 |
| 1.14 | Architecture constraints on Core | 13 |
| 1.15 | Architecture constraints on Loops | 14 |
| 1.16 | Gantt chart | 15 |
| 1.17 | Summary table with different designs | 16 |
| 2.1 | Table of performances [6] | 18 |
| 2.2 | Table of development time [6] | 18 |
| 2.3 | Table of development time [7] | 19 |
| 2.4 | Quality of results and design effort of HLS compared to hand-written RTL in several cases, work of [20], citation of [21], [22], [23], [24], [25], [26], [27], [28] | 20 |
| 3.1 | Frequency response of the filter | 22 |
| 3.2 | FIR | 22 |
| 3.3 | Matlab model of the filter | 23 |
| 3.4 | Gen_dig_stim function | 23 |

| | | |
|------|--|----|
| 3.5 | Architecture of the project | 26 |
| 3.6 | Type definition [18] | 28 |
| 3.7 | Range of different types [18] | 28 |
| 3.8 | Exact bits for operations [18] | 29 |
| 3.9 | Rounding | 30 |
| 3.10 | ScVerify | 32 |
| 3.11 | Test result | 32 |
| 3.12 | Results of the first synthesis | 33 |
| 3.13 | Results of the first synthesis | 33 |
| 4.1 | Area report for SM filter | 35 |
| 4.2 | Power report for SM filter | 36 |
| 4.3 | FIR with order 12 | 37 |
| 4.4 | FIR with order 6 | 38 |
| 4.5 | FIR with order 6 and symmetric coefficients | 38 |
| 4.6 | FIR with order 7 | 39 |
| 4.7 | FIR with order 7 and symmetric coefficients | 39 |
| 4.8 | Data_in and data_out frequency | 40 |
| 4.9 | Summary table of Rolled Loop solution | 42 |
| 4.10 | Complete list of objects instantiated | 42 |
| 4.11 | Two independent ROMs | 43 |
| 4.12 | Statistics on the area | 43 |
| 4.13 | Complete list of objects instantiated with area estimation | 45 |
| 4.14 | Overall RTL of the Rolled Loop | 46 |
| 5.1 | Different shift-unrolling performances | 49 |
| 5.2 | Shift unrolling area comparison | 49 |
| 5.3 | Scheduling of the Shift full unrolled filter | 50 |
| 5.4 | Gantt chart for the Shift full unrolled filter | 51 |
| 5.5 | Timing of the Shift full unrolled filter | 51 |
| 5.6 | Overall RTL of the Rolled Loop | 52 |
| 5.7 | Pipelining | 52 |
| 5.8 | Pipelining | 53 |
| 5.9 | Pipelining | 54 |
| 5.10 | Interfaces differences | 55 |
| 5.11 | Valid_in behaviour and wait handshake | 57 |
| 5.12 | Start of the timing and valid_in behaviour | 57 |

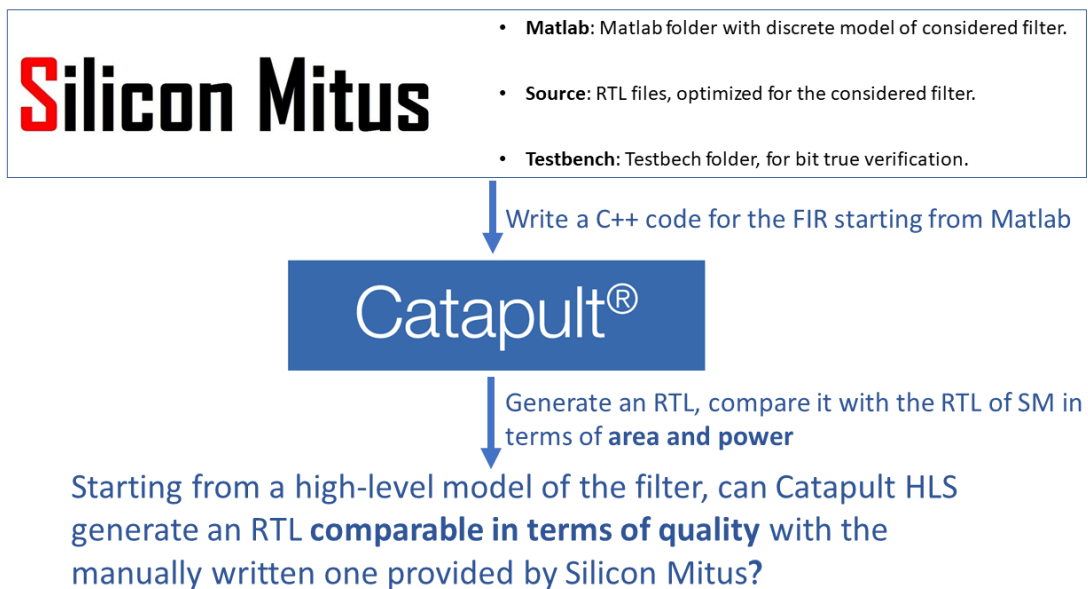
| | | |
|------|---|----|
| 5.13 | General functioning of the input/output sync | 58 |
| 5.14 | Backannotation process | 59 |
| 5.15 | Nets Switching activity and Toggle rate | 61 |
| 6.1 | Zynq architecture[30] | 62 |
| 6.2 | Differences between the Memory-mapped interface and the Streaming interface[31] . . | 64 |
| 6.3 | Setting for the compatible board | 64 |
| 6.4 | AXI Interface of the block with FIR and FIFOs | 66 |
| 6.5 | Window for creating a new IP | 66 |
| 6.6 | Filter with FIFOs IP | 68 |
| 6.7 | Interconnections between the AXI interface and the Filter+FIFOs IP | 69 |
| 6.8 | Timing for the verification of the AXI_FIR_FIFOs IP | 72 |
| 6.9 | Interconnections between the Zynq PS and the AXI IP through other blocks | 72 |
| 6.10 | Screen of Vivado SDK with the UART Terminal | 73 |

CHAPTER 1

Introduction

1.1 Work introduction

This Master thesis work is done in collaboration with Silicon Mitus. Objective of this work is to demonstrate that a tool for High Level Synthesis, taking as input an high level description of a FIR filter, can automatically generate an RTL architecture that is competitive in terms of area and power with respect to a product developed with the traditional HDL design.



The company provides a Matlab source of the FIR filter that generates the golden outputs which are taken as reference point. Moreover is provided also an RTL optimized description of the filter written in System-Verilog that is useful to better understand the architecture that it must be recreated and to have a comparison in terms of area and power. Lastly Silicon Mitus, as is schematized in Fig.1.1, gives

also a testbench file, written in System-Verilog, that will be necessary to construct a new testbench for the architecture generated with High Level Synthesis.

The first objective of the thesis is to write a C++ code, based on the above Matlab code that can be synthesized by Catapult HLS tool. Then, after an RTL description in Verilog has been generated, it is possible to give it as input to Synopsys Design Compiler. This tool performs another synthesis from the RTL level to the gate level. Now it is possible to have a better knowledge on the area occupied and on the two types of power, static and dynamic.

A comparison between the synthesized code from the Catapult RTL and the synthesized code from Silicon Mitus RTL is done and finally the conclusions about the quality of results can be extracted.

1.2 Definition of High Level Synthesis

High Level Synthesis (HLS), also known as electronic system-level synthesis (ESL), describes an automatic process executed by a tool to translate an algorithmic description into a hardware description. A set of constraints and goals must be specified together with the high level description and they affect the behaviour of the final hardware, that is the way the system interacts with the environment. The description of the whole hardware is also called structure and it is made by a set of interconnected components (from the more complex ones like CPUs and memories to the simplest ones like flip flop and logic ports). Structure must be mapped into a physical domain. The smallest unit of the physical domain is the transistor. Behavioural, structural and physical domains are three different ways of description of an hardware system, each of them could be then divided in levels of abstractions as shown in Fig.1.1.

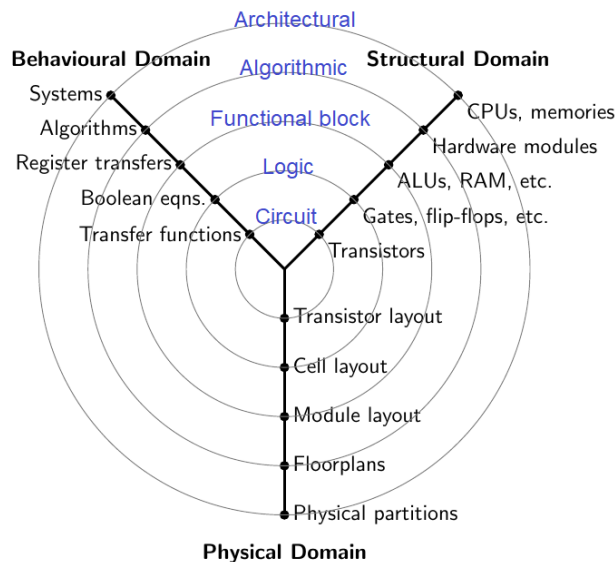


Figure 1.1: Gajski-Kuhn chart [1]

The Y diagram called *Gajski-Kuhn chart*, was proposed for the first time in 1983 as an indication of the different perspectives in VLSI hardware design. More precisely, the graph reported is the refinement of Robert Walker and Donald Thomas made in 1985 [1]. On this graph the authors drew which should have been the best path for designer engineers (Fig1.2). So starting from a high level

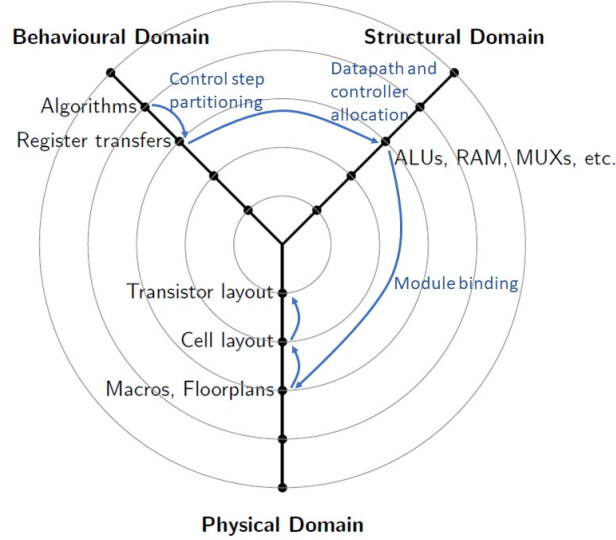


Figure 1.2: Gajski-Kuhn chart [1]

of Behavioral Domain the first passage is to step down to a lower level of abstraction in the same domain, then move to the Structural Domain. Here are defined the datapath and the control unit of the architecture. These two passages are made, in this thesis, by Catapult, the HLS tool that arrives to a HDL description of the filter starting from a C++ code. Then, to complete the design, there must be others steps towards and inside the Physical Domain. All of these steps could be described as *few-to-many* because, in general, many solutions are available to move from the starting point to the end point of any arrow. For example there may be many structural implementations of a particular behaviour.

High Level Synthesis, as we use the term, means to move from an Algorithmic level description of a digital system to a Register-transfer level (RTL) structure that implements its behaviour. From the input specifications, it is produced a datapath that contains functional units, registers, multiplexers and buses. If the datapath does not include the control block, as often happens, the synthesis tool produces also the control part like microcode, PLA profiles or random logic.

From a single algorithm usually many different RT implementations could be produced. The work of the designer is to guide the tool to the best hardware for his scope. It may happen that there are constraints for example on the final floorplan area or on the maximum clock frequency or on the dissipated power or even on the costs of production. HLS gives to the designer a better control over the optimization phase to efficiently build and verify the hardware.

HLS must be distinguished from other type of synthesis, like RTL-synthesis or logic-synthesis, because all of them could be used after the high level one. The register-transfer starts from an already specified set of registers and functional blocks and the interconnections between them is largely or completely known. Logic synthesis, on the opposite, works from a description of the system through logic equation and the final result is a technology that fits in the best way the given equations. It is used, once the HDL description of the system is available, to synthesize the gate level as clearly shown in Fig1.3.

In the work of McFarland, Parker and Camposano [3] are described which are the main reasons of

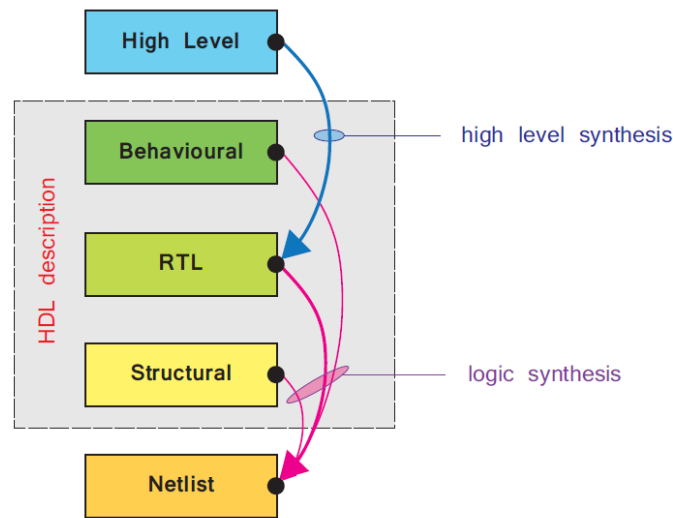


Figure 1.3: HLS vs Logic synthesis [13]

the High Level Synthesis growing trend:

- Shorter design cycle: If a large part of the design is automated, the time-to-market is consequently reduced. Furthermore also the costs decrease because one of the expensive part of the chip is its design.
- Fewer errors: In HLS synthesis product can be easily verified.
- Ability to search the design space: It is possible to produce a synthesis reasonably quickly and this leads to the possibility of exploring a large number of different designs. Moreover an automated tool can suggest tradeoffs.
- Documenting the design process: The decisions carried on by the tool are tracked and it is possible to know which are the consequences of those decisions.
- Availability of IC technology to more people: It is easier for inexperienced RTL-designers to produce their own chip.

High Level Synthesis is nowadays introduced in the history of design of digital architectures. Starting from the 70's years the growing complexity of integrated circuits (Moore's Law) leads to an

even growing necessity to abstraction to gain in productivity. The compulsive integration of billion of transistors on the same chip and nowadays also the addition of various technologies (SoC) gave a big boost to the search of new methodologies of design. The HLS is one of the most promising way to keep up with respect to the integration growth. Fig.1.4 shown different design methodologies that were used during the last decades.

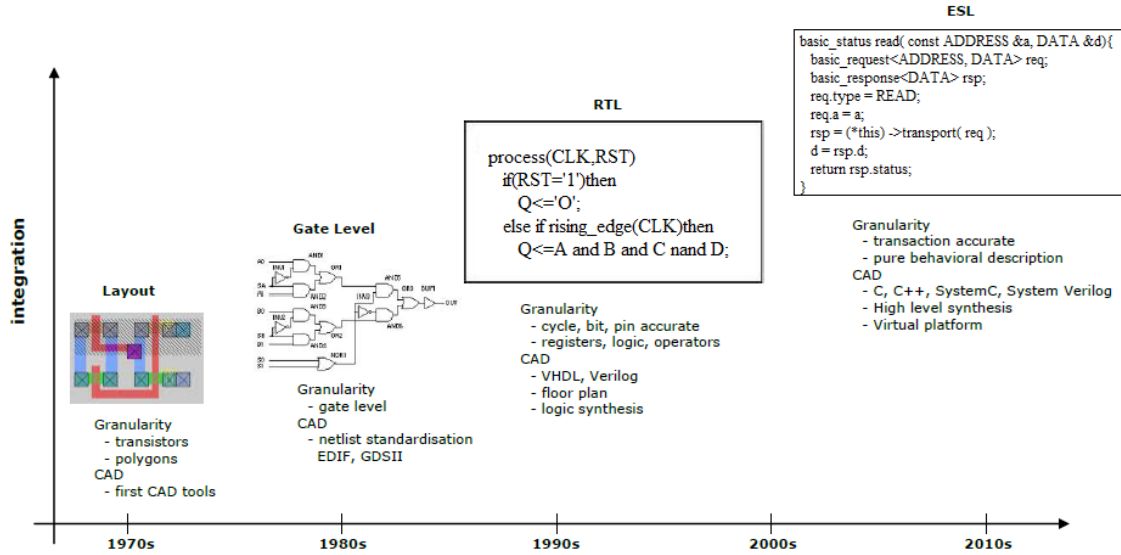


Figure 1.4: Different type of design in the history [12]

1.3 Brief history of HLS

Experts on this subject [2] are used to divide the history of the High Level Synthesis in three periods that represent three different evolutions of this new kind of approach, from hereinafter we can call them *generations*.

The first period covers years 70s-80s (Generation 0) and 80s-90s (Generation 1) when most of the works were made in the research field. To a more in-depth analysis on the differences between these two generations please refers to the work of G.Martin and G.Smith [2], while here are treated as a unique period due to the similar contents. One of the main topic was the optimization of layout, that is part of the lower level of a hierarchy. Noticeable contribution came from the Carnegie-Mellon University of Pittsburgh where the Expl system was invented. It is the first that explores the possible optimizations in the design working on the trade-off between series structures and parallel structures. Main limitation of this system was that it works below an algorithmic level using standard register-transfer modules to optimize the design. All the blocks are written in a hardware description language called ISPL. In the late 70s the same research group guided by Alice Parker developed a new automation system called CMU-DA (Carnegie Mellon University Design Automation) [4] which input is a functional

specification so focused on describing very well the interfaces (input and output behaviour) while not necessarily describing the internal structure. The original design system of the CMU-DA is reported in Fig.1.5 to have a term of comparison with the today's design tools.

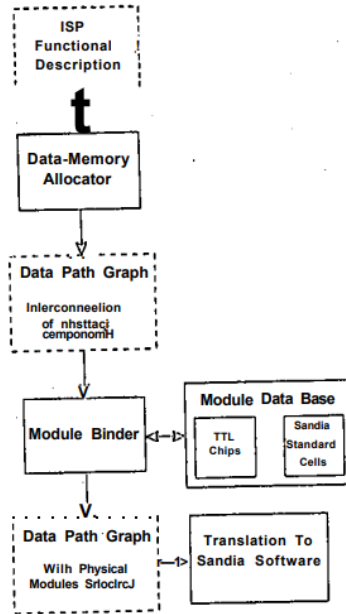


Figure 1.5: The original CMU Design System [4]

The foundations of the contemporary HLS were laid during those years but, from commercialization of this new technology point of view, this generation faced a terrible failure. As reported in the work of G.Martin and G.Smith [2] the four mainly reasons, that we are going to analyse better later, of the failure were: wrong historical period, input languages, quality of results and domain specialization.

During those years many new revolutionary technologies were introduced such as automatic placement and routing and RTL-synthesis and probably behavioural-synthesis were put aside. Also because hardware description languages became very popular in those years while HLS tools are programmed to accept as input source unlikely languages like Silage that will never be used again. Third cause is referred to the roughness of those early tools with expensive allocation and primitive scheduling. The final cause was probably the focusing on DSP design rather than the ASIC one, that was in vogue at the time.

Second period or Generation 2, must be defined till 2000s when major companies started to develop and commercialize their first tools. The main example could be *Behavioural Compiler* introduced by Synopsys in 1994 that uses Verilog and VHDL as possible input languages.

The use of a behavioural HDLs was one of the main causes of failure for this generation because it enters in competition with the existing RTL synthesis tool. This deters the interest of all algorithm

and software designers to learn this new methodology. Instead RTL synthesis users did not find a substantial improvement in terms of area and performance with respect to the tool they were already using.

Some of the mistakes of this generation were the poor quality of the results and the wrong input language, because, as we said, algorithmic designers were waiting for higher level programming languages. The quality in particular was not satisfying talking about synthesis of circuit designed for control and not for dataflow or signal processing. Other causes can be found in a lack of attention in the synthesis of the interfaces and the impossibility, at that time, to have tool for the formal verification of the HLS results.

A third generation, starting from early 2000s, introduced the big innovation of high level programming languages as C, C++ or SystemC. The paradigm of possible solutions increased incredibly and all of the market's fields (ASIC, ASSP, FPGA, DSP, control) were covered by new tools. In this period Mentor Graphics *Catapult C Synthesis* was born which is the predecessor of the tool used in this work. In this generation finally most of the critic points of the previous ones were fixed, leading to a reasonable success of the HLS tools in particular in Japan and Europe. Going further in details, the right input languages like C or Matlab, comfortable for designers, gave a significant boost to this generation. Moreover the wider domain of application and the improved quality of the results are important keys. During these years many of the most famous companies turn to the use of simpler hardware blocks, generated with HLS, to accelerate their algorithms. For example Catapult was used by Nokia to generate hardware implementation of DSP for wireless communications starting from a Matlab code [2]. The same tool was used by Alcatel Aerospace, Ericsson, Fujitsu and Toshiba.

In the Calypto Design System's 3rd annual independent worldwide survey executed during January 2011 the results of the overall uses of HLS tools is show in Fig2.1. This anonymous survey was emailed to several thousand SoC/IC design professionals worldwide. The same survey in 2014 evaluate which

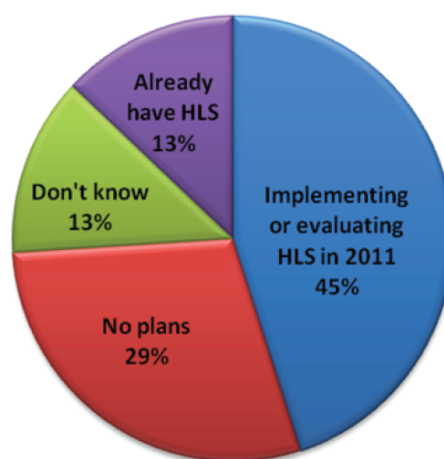


Figure 1.6: High Level Synthesis deployment 2011 [9]

are the type of hardware developed with usage of HLS. The result highlights a broad spectrum of hardware being designed, including wireless, video, imaging, graphics and switch/routers.

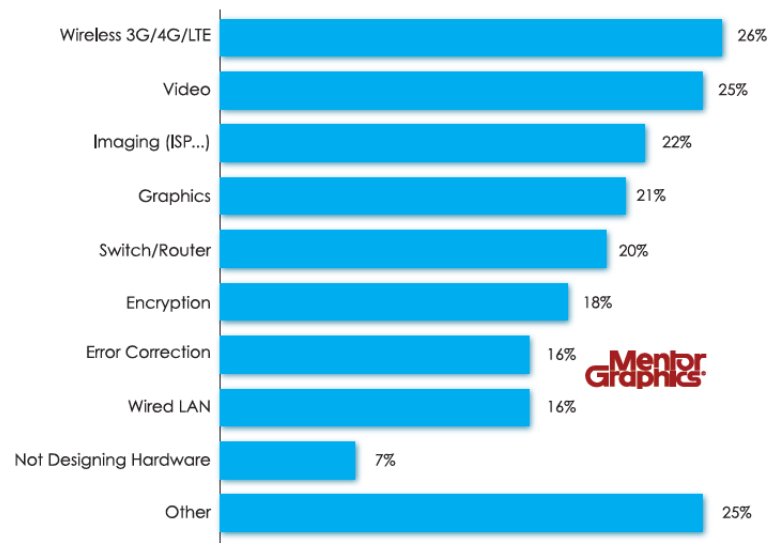


Figure 1.7: Types of hardware being designed 2014 [11]

1.4 Catapult HLS

In this section it will be described how the HLS tool chosen for our purpose works. The Catapult C Synthesis tool automatically generates control and algorithmic RTL designs from both C++ and SystemC sources. This process gives designers the possibility to move up to an higher abstraction level for both design and verification of ASICs and FPGAs. Both time-to-market and freedom to automatically explore different solutions are boosted and a fully optimized and error-free hardware implementation is quickly achieved. Catapult has integrated High-Level Verification (HLV) tools and methodologies that enable designers to complete their verification signoff at the C++ level with fast closure for RTL.

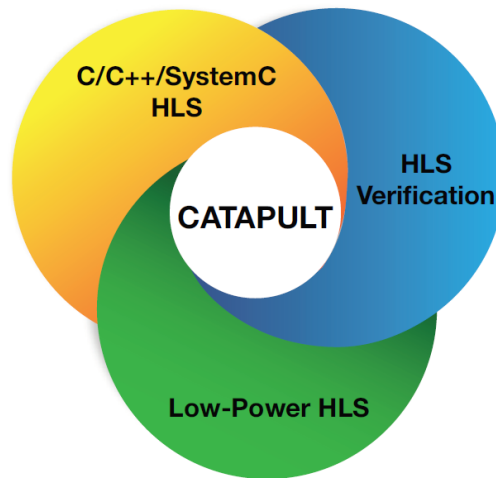


Figure 1.8: Catapult High-Level Synthesis Platform [15]

Catapult C has been recognized as the High Level Synthesis market leader by Gary Smith EDA for 3 years in a row [16].

As we already said, HLS simplifies the traditional RTL generation by automating the design. Using C++/SystemC reduces the numbers of code lines up to 80% making it easier to write, modify because of changes at the end of the design, retarget to a different technology and debug. In Fig1.9 is reported a study made by Nvidia Corporation [17] about the time and the resources consumed by the traditional RTL design and Catapult HLS one. As is it possible to understand using the HLS there is a gain both in terms of time and in terms of resources occupied. The highly interactive Catapult workflow provides full visibility and control of the synthesis process, enabling designers to rapidly converge to the best implementation for performance, area, and power. After the RTL has been synthesized, Catapult automates a complete verification infrastructure reusing the original C++ or SystemC testbench to exercise the generated RTL.

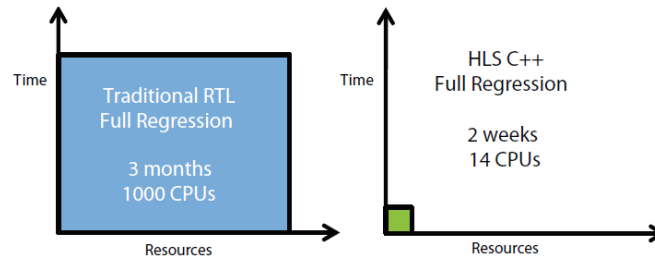


Figure 1.9: NVIDIA Closes Design Complexity Gap with High-Level Synthesis [17]

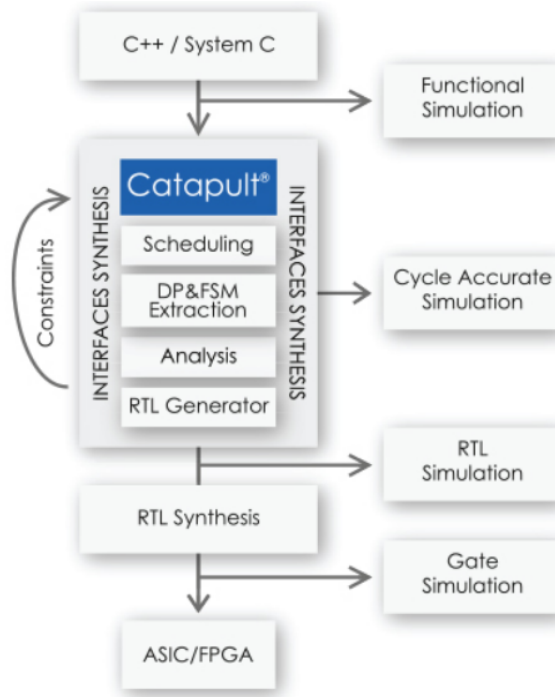


Figure 1.10: The Catapult design flow [14]

1.4.1 Catapult flow

To produce the desired output, Catapult provides the user with a friendly interface where are presented all the tasks that can be used to steer the tool. This panel is shown in Fig1.11. Now we will explain all of the single task.

- **Input files:** Clicking on this first option, the user can insert all of the input files needed for the synthesis. All of the coding files must be specified but the headers files are not necessary, they must be simply present in the same directory. An important tip is to remove the tick at those files that must be included in the project but must not be synthesized, as for example the testbench file.
- **Hierarchy:** All of the functions described in the code are shown here and for each of them

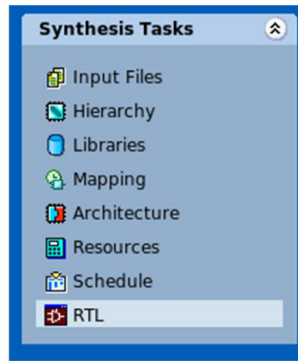


Figure 1.11: The Catapult design flow

a hierarchy must be specified. Possibles status may be: Top, Block and Inline. Only one function could be specified as 'Top' of the hierarchy, this function should make all the calls to the remaining functions that are signed as 'Block'. All other functions and labels are designated 'Inline', meaning they are inside one of the hierarchy blocks. Labels are typically used to emphasize loops and mark them as sub-blocks. Typically the hierarchy is designated in the source code files by inserting the definition *hls_design pragma* followed by one of the three types, just above the function as in Fig.1.12.a . We must be careful because the Hierarchy Constraints Window of the GUI Fig.1.12.b has a higher priority than the directives specified in the code, so if the settings are conflicting, the code one will not be considered. If no hierarchy constraints are specified, all of the functions and labels are considered to be 'Inline' by default.

- **Libraries:** At this step the technology with which we want to synthesize must be defined. To specify the technology, you first select a target RTL synthesis tool and a device. Based on your selections, an associated list of IP libraries appears in the Compatible Libraries field on the right. The set of libraries consists of a base library and some additional IP libraries (i.e. for RAMs and ROMs). In this work the RTL tool is the *OasysRTL* and the target library is the *Nangate 45nm*. Here is also possible to generate a new memory with the specific command starting from a VHDL/Verilog description.
- **Mapping:** Now the clock parameters are set. There are basic parameters like frequency and period, duty cycle, offset and uncertainty, furthermore is also possible to implement automatically advanced signals like two types of reset (synchronous or asynchronous) or the enable signal which function is also clock gating. A signal called "Transaction done" can be enabled. This is useful for synchronization of the input/output signals and, in case there is not specified handshake, every boundary signal has its own *triosy* signal that indicate completion of I/O transactions. In this work, the I/O is managed by the *ac_channel* interface as will be more clear in Chapter 4. The *triosy* signals are not necessary with this interface type because it already has a ready/valid handshake, called *wait handshake*.

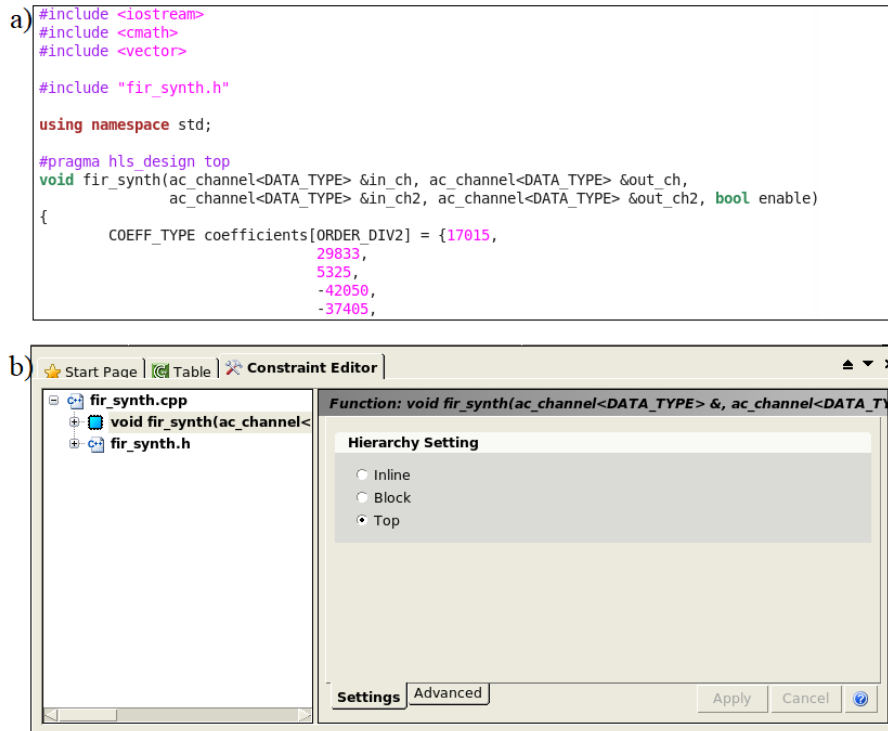


Figure 1.12: Hierarchy settings

- Architecture: This is probably the main task of all the design flow because here the constraints on loops, memories, input and output interfaces and arrays are set. For the 'module' *fir_synth* that is the 'Top' block the parameters that are flexible are reported in Fig1.13. The effort level can be setted as medium or high, moreover the user can modify the input and output delays and others options. Step down in the architecture, we have at the same level 'Interfaces' (which kind of protocol to be used, in this thesis the wait protocol will be used, as specified above), 'Constant array' (if present, optimize the memories) and 'Core'. We do not enter in all of the possible settings for the first two. The interesting part is the optimization of the 'Core' that is presented in Fig.1.14.

Besides the possibility to change the effort, there is the option *Design Goal* that can be setted on *latency* or *area* which is important for the scope of this work. Furthermore it is possible to specify an expected value of area and maximum latency, if the tool cannot meet the specified constraints, generates a warning message. It is also possible to change the share allocation time that is the percentage of clock period reserved for the logic needed to share components. The default value of this parameter is 20% of the clock cycle. Another step down brings to a lower level where are present the memories inside the 'Core', called 'Arrays', and finally the loops that form the central part of the design. 'Arrays' can be modelled as a set of registers or as a memory, taking into account that the memories that can be selected in this step are the ones

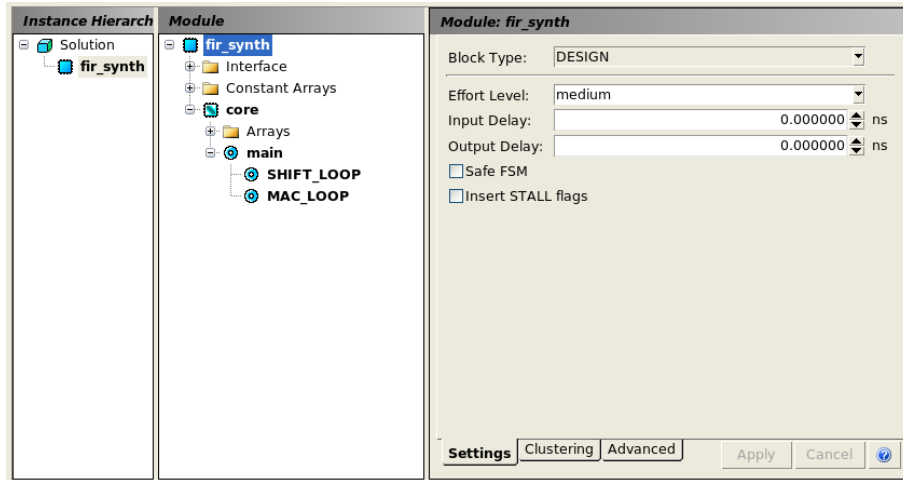


Figure 1.13: Architecture constraints on Module

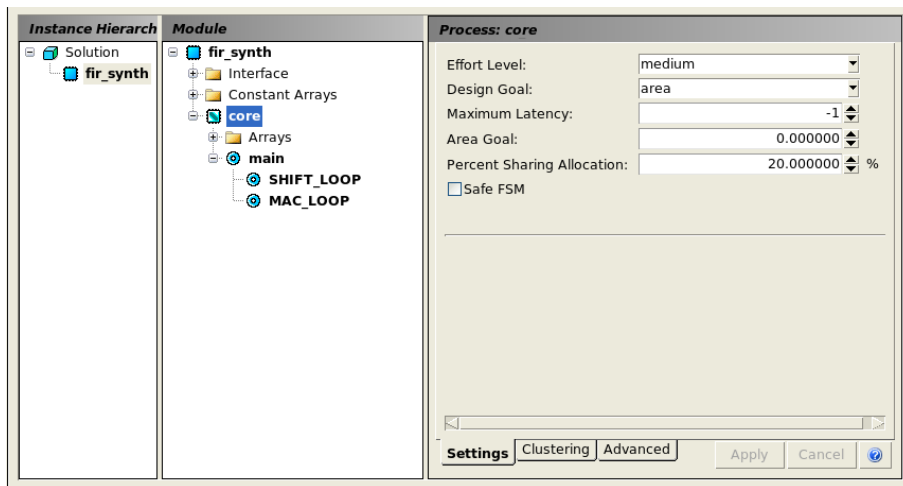


Figure 1.14: Architecture constraints on Core

included in the libraries step. The optimization of the loops is the part that has most weight inside the overall design because loops can be easily modified following different strategies. The two main techniques that Catapult provides are *Pipelining* and *Unrolling* as shown in Fig1.15 that improve the hardware function's performance by exploiting the parallelism between loop iterations.

Here they are briefly described:

- Unroll: If we check the box of "unrolling", it creates multiple copies of the loop body and adjust the loop iteration counter accordingly. By default the loop will be completely unrolled, which allows the architecture to perform all the operations in only one clock cycle. This is possible replicating the hardware a number of times equal to the initially iteration counter. The area becomes very wide. In the schedule step, it will put as many iterations

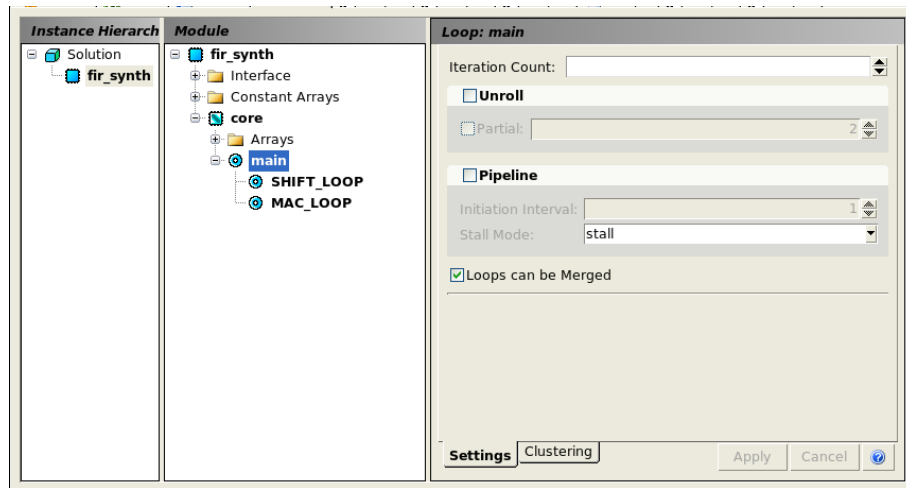


Figure 1.15: Architecture constraints on Loops

as possible in a clock cycle instead of an iteration in each clock cycle. It is possible to unroll it partially. The number set in the "Partial" field specifies how many times the loop body is copied. Here is an example of code:

```

1  int sum = 0;
2  for(int i = 0; i < 10; i++) {
3      #pragma HLS unroll factor=2
4      sum += a[i];
5  }

```

That is the same as writing:

```

1  int sum = 0;
2  for(int i = 0; i < 10; i+=2) {
3      sum += a[i];
4      sum += a[i+1];
5  }

```

- Pipeline: This pragma reduces the *Initiation interval* (II) which is the number of clock cycles between the start times of consecutive loop iterations. This is possible thanks to the concurrent execution of operations. The default initiation interval for the pipeline pragma is 1, which processes a new input every clock cycle. It is important to know that the loops nested inside of a pipelined loop are automatically pipelined too.

It is important to mention also the option that allows the tool to merge the loop with other loops or, on the other hand, maintain one loop independent from the rest of the loops. Another important advanced option is to introduce clusters, there is an entire window called "Clustering" to do that. Clusters are standard blocks that implement a typical operation, for example a MAC, and they are already optimized. There will be a deeper analysis on this type of optimization in

Chapter 4.

- **Resources:** Click on this button shows which resources will be utilized in the RTL composition, so it is possible to choose which adder or multiplexer use. This step can be skipped passing directly to the scheduling. In this case Catapult automatically will choose the best components following the directives of the constraints at the previous steps.
- **Schedule:** Catapult generates the Gantt chart that is a graph where the design was analyzed by the timed point of view. With the chart is possible to know which operations are done in each clock cycle and which one consumes more execution time. The Gantt chart graphs the number of control steps (C-steps) in each loop and the sequence of the operations scheduled within the C-steps. Operations are shown in blue in a box proportional to the operation delay. Drawback

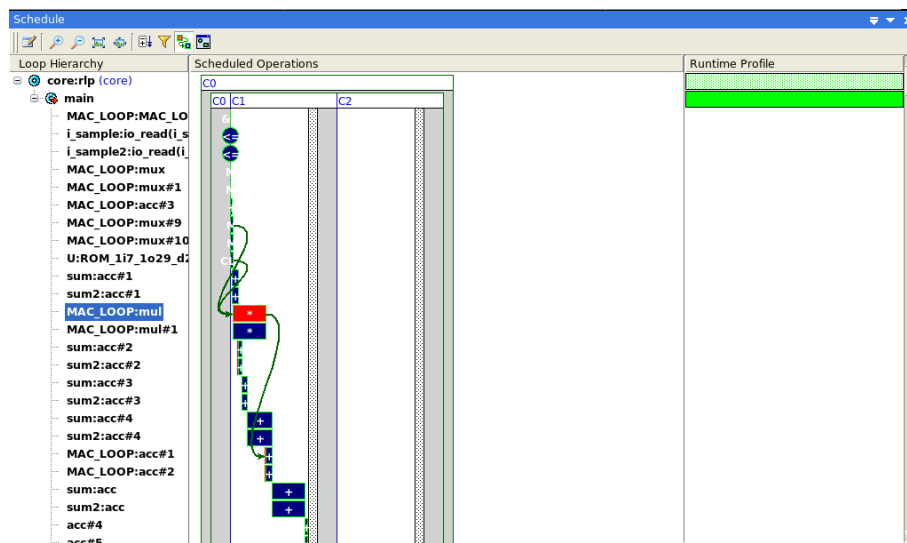


Figure 1.16: Gantt chart

is that the user can not see the pipeline stages in a clear way and this makes it difficult to check where the pipeline stages have been placed in the design. Selecting a data object in the Schedule window highlights the object in all columns and displays arrows in the Gantt chart to show dependencies between the selected data object and other operations. Different coloured arrows indicate different types of dependency paths. In Fig1.16 it is shown a MAC operation in the Gantt chart with a data dependency that is coloured in green.

- **RTL:** This is the last step when the HDL code is generated and written on different output files both in Verilog and VHDL. Catapult automatically generates also an RTL schematic where the user can easily find the critical path and some reports. These lasts are about the resources used in the design (*Bill Of Materials*) and useful information of how the design has been characterized.

After the generation of the RTL of the design, some fundamental characteristics are reported in a

summary table that shown for every synthesized design version its performances in terms of latency, throughput and area, as shown for example in Fig.1.17. Are reported either the Latency and Throughput in terms of clock cycles or in terms of time ($n^{\circ}clock_cycles \cdot clock_period$) and finally the Slack that is the difference in time between the clock period and the travel time of the information along the longest combinatorial path.




| Table | | | | | | |
|---|----------------|----------------|-------------------|-----------------|-----------------|-------------|
| Report: General | | | | | | |
| Solution / | Latency Cycles | Latency Time | Throughput Cycles | Throughput Time | Total Area | Slack |
|  fir_synth.v1 (extract) | 388 | 5051.76 | 389 | 5064.78 | 231983.34 | 0.46 |
|  fir_synth.v2 (extract) | 128 | 1666.56 | 128 | 1666.56 | 92549.31 | 0.06 |
|  fir_synth.v3 (extract) | 130 | 1692.60 | 130 | 1692.60 | 87065.50 | 0.12 |

Figure 1.17: Summary table with different designs

CHAPTER 2

State of Art

2.1 Previous works on filter synthesis through HLS

High Level Synthesis is an automated process having a big impact on the design of digital circuits as already said in Chapter 1 of this work. There are a lot of academic works that during these years have investigated and established the quality of this new process with respect to the traditional design flow.

Works that finally arrive to a positive or negative judgment on High Level Synthesis as a technique to design elements for DSP, like the digital filters, are of particular importance and will be reported in this chapter.

The paper whose title is "*Fast FIR filter implementation using High-Level Synthesis tool*" [5] is only apparently connected to the scope of this work but it is interesting to analyse. The work of T.Ognunfunmi and S.Desai has been written in the far 1994 and the tool utilized was one of the first versions of *Synopsys*. To confirm what we have said in the HLS's history section, the input language of the HLS tool is Verilog hardware description language. This work can be placed in the middle of *Generation 2* because the tool needs an hardware description input language rather than a C-like language as required by the next generations. The tool's results were very impressive for that time, as described in the paper, but are a bit far from the results that nowadays tools can produce.

Among contemporary works, FIR filter implementation has been analysed in 2015 by Hanbo, Shaojun and Yigang [6]. After a description of the advantages of HLS on the traditional RTL design, they present an interesting implementation of an FIR with three different tools: Vivado HLS, LabVIEW FPGA and DSP Builder. Unfortunately, Catapult HLS (the tool used in this work) is not part of the comparison. The circuit analyzed is a 20-order low-pass filter with input data on 16bits, unfortunately, very different to the one of SM, that has an order of 128 and 32bits of parallelism.

The paper does not enter into the details of how to write a correct code for describing a filter in high

level languages, but is more focused on the different flows of project that every tool has. However this part is not interesting from this work's point of view, so we directly analyse the results, reported hereinafter.

The three tools, cited above, are compared with two other tools that are used for a traditional design, also called IP (Intellectual Property) core design. There are a lots of IP, that are basic or hierarchical blocks already described and most of them are fully parameterized. This meets the needs of the majority of the designers but it also makes more complex the writing of an efficient code.

| Development Tools | Resources Occupation | | Synthesis Time (to RTL) | Optimization Options | The Highest Frequency | Latency |
|----------------------|----------------------|------|-------------------------|----------------------|-----------------------|---------|
| | LUTs | DSPs | | | | |
| VIVADO HLS | 236 | 44 | 25s | Exist, plenty | 229.86MHz | 15 |
| VIVADO (IP core) | 89 | 1 | 65s | Exist, plenty | 222.22MHz | 44 |
| DSP Builder | 111 | 2 | 14s | Null | 180.54MHz | 48 |
| Quartus II (IP core) | 12 | 19 | 26s | Exist, few | 116.25MHz | 78 |
| LabVIEW | 9623 | 5 | 572s | Null | 68.68MHz | - |

Figure 2.1: Table of performances [6]

Above all, what emerges from this table is that HLS tools in general require more hardware than an implementation through IP core. Moreover, in general HLS leads to a faster synthesis of the filter, with many tools the time is more than halved (Vivado HLS and DSP Builder with respect to IP core). The performances are better also for the maximum frequency and the latency in the case of HLS. The quality of results and the pooriness of possible optimizations that LabVIEW offers can be explained if we take a look to the second image (Fig.2.2).

| Developments Tools | Development Time(day) |
|--------------------|-----------------------|
| VIVADO HLS | ≈ 8 |
| DSP Builder | ≈ 4 |
| LabVIEW FPGA | ≈ 3 |

Figure 2.2: Table of development time [6]

Looking at this, it is possible to understand that a project in LabVIEW has a development time that is shorter. Because of this it is more useful to those designers that want to explore a lot of different solutions quickly or for those who are beginners of the HLS technique. As opposed to this a complete tool, like Vivado HLS, it has plenty of possible optimizations, it is based on a C language but it has an higher development time.

Unfortunately, in the literature, there are no other specific papers on FIR filters design, with HLS. Performances of HLS are tested by many researchers and at the moment there isn't a unique shared

opinion about that, because performances of a design with HLS deeply depend on the type of architecture.

In a joint work between the University of Sfax and the University of Nice-Sophia Antipolis [7] the different performances between HLS and manual design of a H.264/AVC Deblocking Filter (part of a video coding system) are presented. In Fig.2.3 it is interesting to consider the different development times, noticed that Catapult HLS has been used in this work. What the authors demonstrate in this

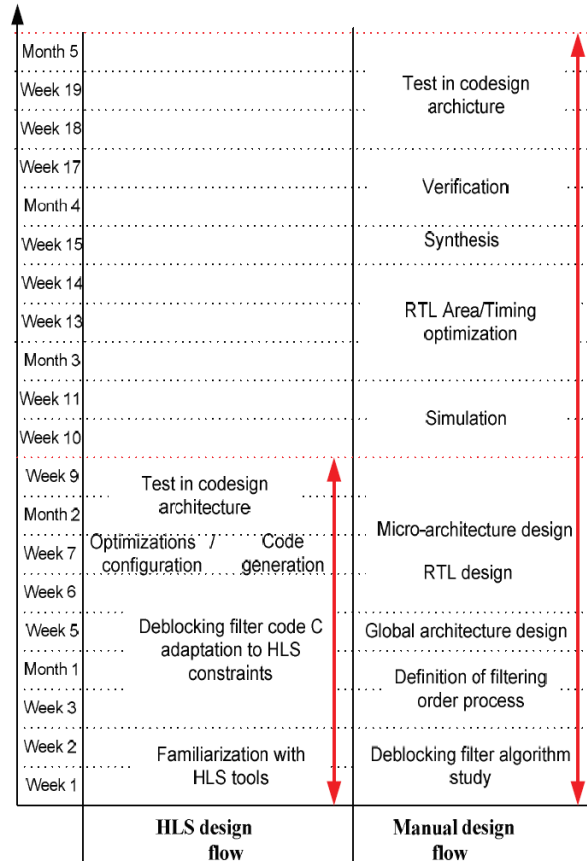


Figure 2.3: Table of development time [7]

work is that the HLS approach has an inferior development time but it also has 3.4 times less performances than the manual design. Also the throughput (in terms of filtered macroblocks) is worse: it is a quarter with respect to the manual design.

This is an example in which the HLS has not the desired effect, and this depends mainly on the application.

Another interesting work on the use of HLS for different applications is the Master thesis work of Tero Joentakanen of Tampere University of Technology [20] in which he summarizes the previous different works on HLS-design versus manual-design (Fig.2.4). Most of these works are Master theses of the University of Oulu.

| Author | Ref | Tool | Application | Target platform | Area (ASIC) / Resources (FPGA) | f_{max} | Effort estimation |
|----------------------|----------|-----------------|-------------------|--------------------|--------------------------------|-----------|-------------------|
| Ollikainen P. | | N/A | DSP + control | ASIC | +15% | – | –17% |
| Järviluoma J. | | HDL coder | IQ data scaling | ASIC | –29% | – | – |
| Zhu Q. & Tatsuoka M. | | Stratus | DMA controller | ASIC | –38% | – | –66% |
| Sun Z. et al. | | N/A | AES encryption | ASIC | +37% | +1.5% | 0% |
| Torppa E. | Catapult | Adder-tree FIR | ASIC | –30% | +1.6% | | |
| | | Systolic FIR | ASIC | –11% | $\pm 0\%$ | | |
| | | Basis functions | ASIC | –36% | –3.3% | | |
| | | Adder-tree FIR | FPGA | +36% LUT, –31% FF | –8.5% | | |
| | | Systolic FIR | FPGA | +23% LUT, +11% FF | –29% | | |
| | | Basis functions | FPGA | +35% LUT, +0.5% FF | +10% | | –80% |
| Kivimäki I. | | Vivado | Signal correction | FPGA | +173% LUT, +34% FF | +7.3% | –70% |
| Zwagerman M. D. | | Vivado | Image processing | FPGA | +61% LUT, –3% FF | –10% | –55% |
| Karras K. et al. | | Vivado | Memcached server | FPGA | –22% LUT, –35% FF | – | –50% |

Figure 2.4: Quality of results and design effort of HLS compared to hand-written RTL in several cases, work of [20], citation of [21], [22], [23], [24], [25], [26], [27], [28]

In the study of all these works one thing appears very clearly: there are huge variations of performances depending not only on the tool but also on the target technologies. In all of these works there are area variations between -38% and +173% and maximum frequency variations between -29% and +10%. All of these numbers must be interpreted as the differences between the HLS design with respect to the manual one. Moreover also the same application but for two different targets (i.e. the Systolic FIR designed in Catapult for both ASIC and FPGA), has completely different performances. For the ASIC target, the FIR has better performances if it is implemented with HLS (-11%). Instead if the target is the FPGA the manual design must be preferred (+23% LUT, +11% FF and -29% f_{max}).

CHAPTER 3

Design of filter

3.1 Upsampling FIR

The filter that is treated in this thesis is an upsampling filter that is part of a high-performance DAC designed to process digital stereo streaming. Target of this technology are the portable devices.

The upsampling grade of the filter is 2 and two of these filters are cascaded to obtain a final frequency that is 4 times the initial one. Sample data rate is typical of audio systems and was fixed to 384kHz.

The internal logic of the filter can work with different frequencies: 19.2MHz, 38.4MHz or 76.8MHz.

The highest frequency is chosen for this work, together with Silicon Mitus. This means that, in order to simulate the audio frequency the input samples need to arrive every 200 clock cycles.

Other specifications of this 1st upsampling stage are:

- Pass band Ripple: ± 0.0015
- Pass band: $0.432 \cdot f_{\text{syn}}$
- -3db band: $0.452 \cdot f_{\text{syn}}$
- Stop band: -100 dB at $0.5 \cdot f_{\text{syn}}$
- Order: 128

These specs define the frequency response of the filter (in Fig.3.1) and the coefficients. These last are usually computed by Matlab but in this case they were provided directly by Silicon Mitus. Coefficients are symmetric this is important because impact on the property of phase linearity of the filter.

The upsampling filter is an FIR (Finite Input Response) filter which architecture, in the direct form, can be schematize as in Fig.3.2. Finite input response means that the output of the filter will be settled to zero after a certain amount of time, so it has no internal feedback that cause an indefinitely

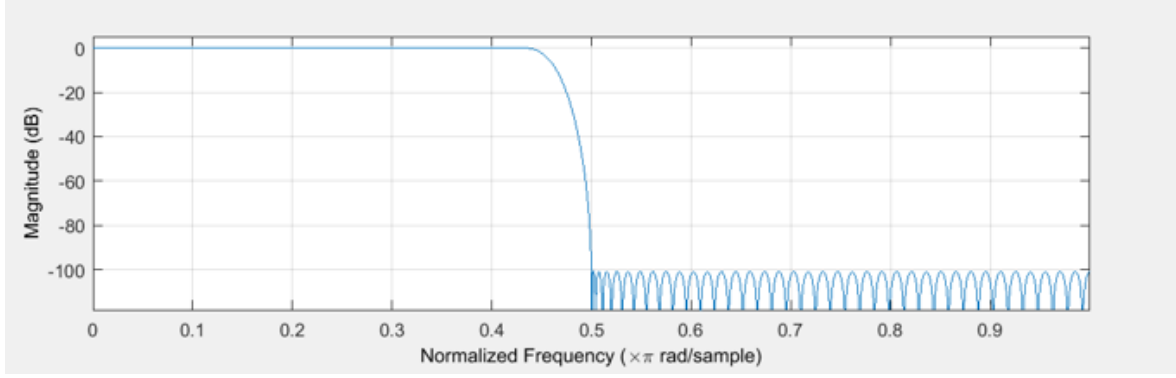


Figure 3.1: Frequency response of the filter

response. As in Fig.3.2 the output depends only on the input and not on the output itself, this makes it essentially a finite weighted sum:

$$y[n] = c_0 \cdot x[n] + c_1 \cdot x[n-1] + c_2 \cdot x[n-2] + \dots + c_N \cdot x[n-N] = \sum_{k=0}^{k=N} f[n-k] \cdot g[k] \quad (3.1)$$

The FIR filter has two main cores that are the delayed line and the MAC (multiply and accumulate) structure that is repeated. The multiplier and the adder form the so-called *TAP*. A filter of order N has $N+1$ TAPs. For this reason the entire filter has a structure that is commonly called *Tapped delay line*.

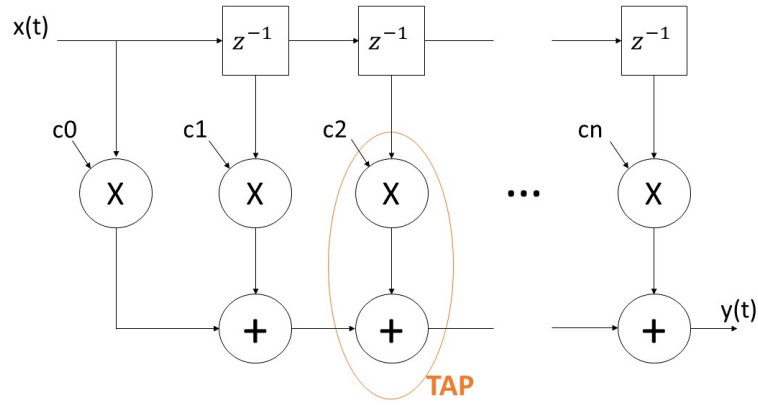


Figure 3.2: FIR

3.2 The original Matlab model

The algorithm of the filter given by Silicon Mitus is written in Matlab language and is comprehensive of one top module, *ovs_top_golden_gen* and other two functions, *gen_dig_stim* and *gen_rtl_stim_dec* that are presented in Fig3.3.

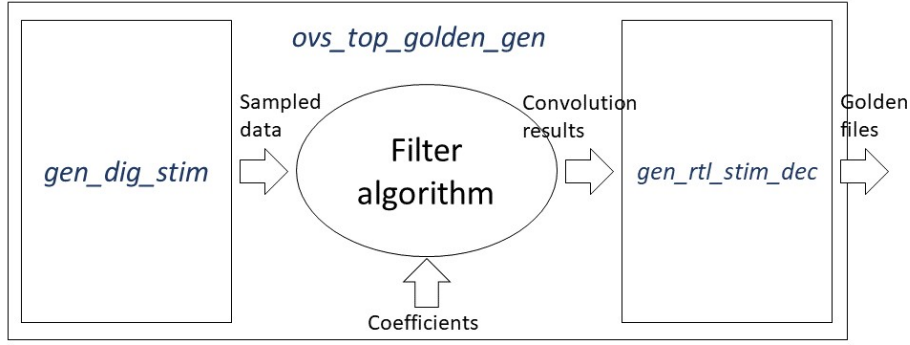


Figure 3.3: Matlab model of the filter

The top module describes a FIR filter with two channels that work on different data, every channel calls the function *gen_dig_stim* to generate input data starting from a sinusoidal wave with different frequencies. Data generated are then quantized on the chosen number of bits. Quantization means to assign every value at one and only one discretized value belonging to a smaller set, often with a finite number of elements. At this point it is necessary to introduce the *dithering*, that is a form of noise with a typical distribution. It is added at the sampled values to minimize the distortion introduced during the quantization step. Instead of round with a predictable cyclicity that would bring to a deterministic error, the round is random. The flow chart of this important function is reported in Fig.3.4.

The technique of zero filling is applied to the quantized data. This implies that two inputs are

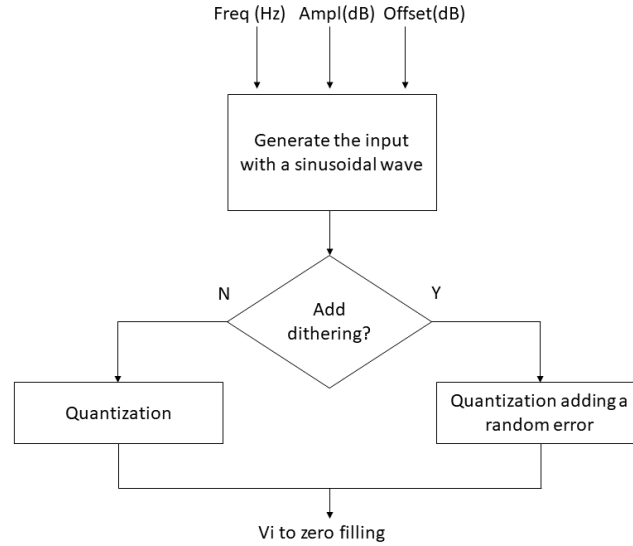


Figure 3.4: Gen_dig_stim function

interpolated by a zero (null-input) so the overall length of the input vector doubles.

Then convolution is applied between input vector and coefficients of the filter that are given. The convolution operation returns an output vector whose dimension will be, in the case of two generic

vectors A and B:

$$MAX(length(A) + length(B) - 1, length(A), length(B)) \quad (3.2)$$

The output vector is divided by NBC-2 instead of NBC to recover zero-padding attenuation and then is rounded.

3.3 Translation from Matlab to C++ of the algorithm

The first step to give the High Level description of the Filter to Catapult is to strictly translate the model from Matlab to C++ language. This step includes the creation of the corresponding function to generate the input vector. The function executes the creation of the inputs using the *math.h* element *sin()*. Also *dithering* is implemented, as shown in Listing 3.1.

```

1 vi[i]=pow(10, arg1/20)*sin(2*pi*arg2*t[i])+pow(10, arg3/20);
2
3 if(addith==1)
4     vi_int.elements[i]=round(vi[i]*(pow(2, nbit-1)-1)+((rand()%100)/100)-0.5);
5 else
6     vi_int.elements[i]=round(vi[i]*(pow(2, nbit-1)-1));

```

Listing 3.1: Dither insertion

The corresponding function of *gen_rtl_stim_dec* is not created because its functionality, which is only to create output files, is simply replaced by the *ofstream* in C++.

In this first implementation of the algorithm the input data are generated inside the FIR project while the coefficients are taken from a text file. Quantized data are then interpolated with zeroes. Zero-filling algorithm is reported in Listing 3.2.

```

1 ofstream out4 ("vi_pad_ch1.txt", ios::out);
2 for (int y=0; y<2*vi_ch1.length;y++){
3     out4 << vi_pad_ch1[y] << endl;
4 }
5 out4.close();

```

Listing 3.2: Zero Filling

The core of the filter is written in a function called *LinearConvolution* that is entirely reported in Listing 3.3.

```

1 void LinearConvolution(double X[], double Y[], double Z[], int lenx, int leny)
2 {
3     int lenz=lenx+leny-1;

```



```

4 for (int i = 0; i < lenz; i++)
5 {
6     Z[i] = 0; // set to zero before sum
7     for (int j = 0; j < leny; j++)
8     {
9         if (i-j >= 0 && i-j < lenx)
10             Z[i] += X[i-j] * Y[j]; // convolve: multiply and accumulate
11     }
12 }
13 }

```

Listing 3.3: LinearConvolution

The convolution definition is the following one:

$$(f * g)(t) = \int_{-\infty}^{\infty} f(\tau)g(t - \tau)d\tau = \int_{-\infty}^{\infty} f(t - \tau)g(\tau)d\tau \quad (3.3)$$

This can be simply replaced by a finite-time formula, used with discrete vectors:

$$y[n] = (f * g)[n] = \sum_{k=0}^{k=N} f[N - k] \cdot g[k] \quad (3.4)$$

This convolution is implemented in line 10 of Listing 3.3, not with an infinite integral but with a discrete sum. There are *lenz* output elements and every one is calculated with the convolution. In the end, convolved data, are firstly divided by *NBC-2* and then rounded adding 0.5 and then using the floor function that has the same meaning in Matlab and C++.

```

1 for (int i=0; i<len_max_conv; i++){
2     vo_firx2_ch1[i+1]=floor(vo_firx2_ch1_full[i]/(pow(2, nbc-2))+0.5);
3 }

```

Listing 3.4: Final Rounding

Every step (Input generation, Convolution and Rounding) is repeated for both channels. At every step, the outputs are tested to arrive at a final output that is the same of the golden one in Matlab.

3.4 The synthesizable filter

Catapult asks as input an algorithm that is synthesizable. It means that the inputs are no more generated internally but are taken from the outside one at a time, because they are the results of a sampling. A wave was sampled at a certain moment, the data was passed to the filter, going forward in time another data was sampled and given as an input to the filter that has memorized the previous one because is useful to generate the new output, and so on.

The other big difference is that now coefficients are stored in a internal memory, this lead to a great occupation of memory because we have 129 coefficients on 30bit. To be synthesizable the project must

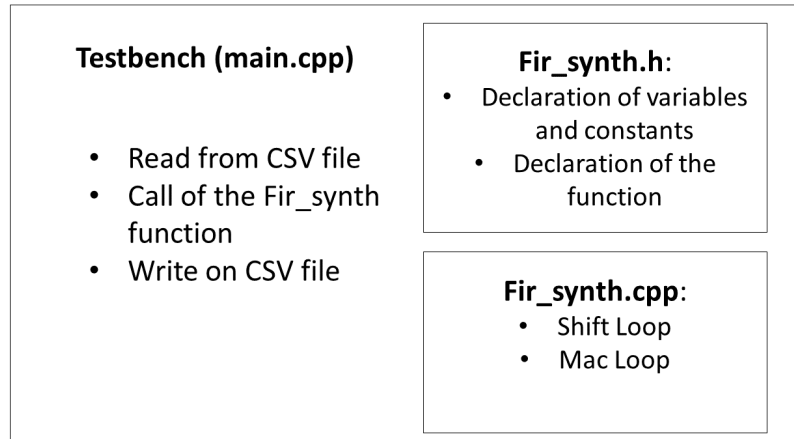


Figure 3.5: Architecture of the project

include three different files, that are the header file (*.h*), the description of the algorithm (*.cpp*) and the testbench that will be necessary to Catapult. A schematic figure of the architecture of the project is in Fig.3.5.

As was described before now the filter declaration has only one element as input and the reference where will be stored the final result as output. All of the variables and constants are defined as double, without considering any optimization in terms of bits. This because to firstly have a positive feedback from Catapult on the implemented algorithm, the optimizations of the space occupied by the variables is not taken into account. The declaration, belonging to *.h* is visible here:

```

1 const unsigned ORDER = 129;
2 const unsigned NBC = 30;
3 void fir_synth(const double i_sample, double &y);

```

Listing 3.5: fir_synth.h

This is the *fir_synth.cpp* script:

```

1 void fir_synth(const double i_sample, double &y)
2 {
3     double coefficients[ORDER] = {17015,...,17015}; //129 coefficients
4
5     static double samples[ORDER];
6
7     SHIFT_LOOP: for (int n=ORDER-1; n>0; n--) {
8         samples[n] = samples[n-1];
9     }
10    samples[0] = i_sample;
11

```

```

12  double sum = 0;
13
14  MACLOOP: for (unsigned n=0; n<ORDER; n++) {
15      sum += samples[n] * coefficients[n];
16  }
17  y=floor(sum/(pow(2, NBC-2))+0.5);
18 }

```

Listing 3.6: fir_synth.cpp

It is possible to clearly identify two main loops, the *SHIFT LOOP* and the *MAC LOOP*. The algorithm now is more oriented to the synthesis and it takes the direct form of the FIR filter described in hardware, like in Fig 3.2.

Shift loop implements a series of registers that introduce delays (z^{-1} in the Z domain). The first one at every clock accept a new data while all the others shift previous data by one position to the right. The *for* loop at line 7 has an index that decrements because otherwise data are overwritten. A total number of 129 registers have been instantiated because in the convolution there be 129 coefficients which in the script are not all represented.

The second loop instead, as its name suggest, perform the MAC operation between coefficients and samples, in this case the index increments because it's necessary to calculate all of the partial sums before arriving at the final output y .

In the end at line 17 is calculated the final rounded output as in Listing 3.4.

3.5 Bit-Accurate filter

In the previous synthesizable implementation of the filter all the variables were defined as *double*, this datatype is 64bit wide. In order to generate less expensive hardware it is possible to define, for every variable, which will be its parallelism in the architecture. Reduce accuracy leads to an increment of the performances and a saving of resources.

Catapult HLS tool supports a library that defines reduced-accuracy datatypes called *AC (Algorithmic C) library* that provides two different header files `< ac_int.h >` and `< ac_fixed.h >` for supporting signed/unsigned integer types and fixed point types respectively. In the table in Fig.3.6, provided by Catapult's guide [18], are shown the differences between the two definitions and which are the parameters to specify. In Fig.3.7 are instead defined which are the ranges of *int* and *fixed*.

From the project specification it is known that:

- Input data \rightarrow 32bit
- Coefficients \rightarrow 30bit
- Output data \rightarrow 32bit

| | Integers | Fixed-Point | Notes |
|-----------------|-------------|---------------------|---|
| Header File | ac_int.h | ac_fixed.h | +incdir+ in Catapult installation directory |
| Type definition | ac_int<W,S> | ac_fixed<W,I,S,Q,O> | W : The full bit width of the type S : Boolean flag, true=signed, false=unsigned I : the number of integer bits in W . Q : the rounding method used on each assignment to this type O : the saturation method used on each assignment to this type. |

Figure 3.6: Type definition [18]

| Type | Description | Numerical Range | Quantum |
|-----------------------|----------------------|--------------------------------------|-----------|
| ac_int<W, false> | unsigned integer | 0 to $2^W - 1$ | 1 |
| ac_int<W, true> | signed integer | -2^{W-1} to $2^{W-1} - 1$ | 1 |
| ac_fixed<W, I, false> | unsigned fixed-point | 0 to $(1 - 2^{-W}) 2^I$ | 2^{I-W} |
| ac_fixed<W, I, true> | signed fixed-point | $(-0.5) 2^I$ to $(0.5 - 2^{-W}) 2^I$ | 2^{I-W} |

Figure 3.7: Range of different types [18]

As consequence in *fir_synth.h* will be defined many different datatypes. The input type (*DATA_TYPE*) is defined *ac_fixed* signed on 32bit with the decimal point on the LSB (this means that is equivalent to the definition *ac_int< 32, true >*). Similar definition is used to define *COEFF_TYPE* and *OUT_TYPE*.

To understand the other definitions take a look at Fig.3.8 and in particular at the first two rows. It describes how many bits are necessary to have the correct precision at the end of an operation. The first two rows are interesting for the *MAC LOOP*. AC-library is useful to define the intermediate parallelism of an operation. At line 6 of the header file is noticeable the definition of the output of the product operation. In the definition are reported the two types of data, in this case *DATA_TYPE* and *COEFF_TYPE* and a little wording *rt_T* that is used when the new datatype is derived by two operands. In line 7 instead there is the definition of the operand type (*PROD_TYPE*) but is reported also the number of cyclic additions that must be done (*ORDER*). It defines a new datatype that is wide enough to store *ORDER* number of summations. In this case *rt_unary* defines a type that requires only one operand.

Last definition to understand is the *INTER_TYPE* that is an internal type useful to the correct execution of the rounding, but will be more clear in the *fir_synt.cpp*.

```

1 const unsigned ORDER = 129;
2 const unsigned NBC = 30;
3
4 typedef ac_fixed <32,32,true> DATA_TYPE;
5 typedef ac_fixed <30,30,true> COEFF_TYPE;
```

```

6 typedef DATA_TYPE::rt_T<COEFF_TYPE>::mult    PROD_TYPE;
7 typedef PROD_TYPE::rt_unary::set<ORDER>::sum    SUM_TYPE;
8 typedef ac_fixed<33,33,true>    INTER_TYPE;
9 typedef ac_fixed<32,32,true>    OUT_TYPE;
10
11 void fir_synth(const DATA_TYPE, OUT_TYPE &);

```

Listing 3.7: fir_synth_fix.h

| Operation | Operand 1 bit width | Operand 2 bit width | Result bit width |
|----------------------|------------------------|------------------------|------------------|
| Add/Subtract | n | m | max(m,n)+1 |
| Multiply | n | m | n+m |
| Negate | n | | n+1 |
| Sum of N operand 1's | n | | n+CLOG2(N) |

Figure 3.8: Exact bits for operations [18]

Now it will be analyzed the .cpp file in Listing 4.1 that unlike what is in the non-fixed version (Listing 3.6) has no more *double* definitions.

```

1 void fir_synth(const DATA_TYPE i_sample, OUT_TYPE &y)
2 {
3     COEFF_TYPE coefficients[ORDER] = {17015,...,17015};
4
5     static DATA_TYPE samples[ORDER];
6
7     SHIFT_LOOP: for (int n=ORDER-1; n>0; n--) {
8         samples[n]=samples[n-1];
9     }
10    samples[0]=i_sample;
11
12    SUM_TYPE sum = 0;
13
14    MAC_LOOP: for (unsigned n=0; n<ORDER; ++n) {
15        sum += samples[n] * coefficients[n];
16    }
17
18    INTER_TYPE temp_out;
19
20    temp_out=sum >> 27;
21    temp_out=temp_out+1;
22    y=temp_out >> 1;
23 }

```

Listing 3.8: fir_synth_fix.cpp

The *INTER_TYPE* pops up at line 18 and is useful because now the power function *pow()* cannot be used due to its definitions, in fact it does not support fixed-type operators. As a matter of fact now the division by a power of 2 is done with a binary shift to the right. After the division another operation must be done, before the *floor()*, that is to sum 0.5 to round the result.

The division was by 2^{28} so what is done is to perform a right shift of the *SUM_TYPE* number of 27bit, and save it in a new variable of width *SUM_TYPE*-27 that is 33 (*INTER_TYPE*). Then add a 1 on the LSB and perform another shift to the right, only by one position, it's like adding up a 1 on the first decimal digit to have an output rounded on 32bit. This simply trick is graphically explained in Fig.3.9.

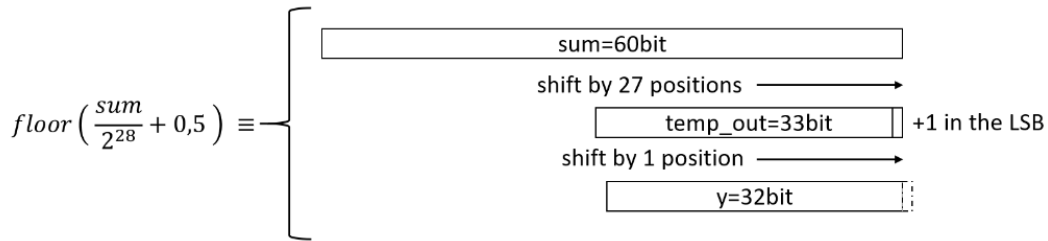


Figure 3.9: Rounding

3.6 The testbench

The synthesizable filter has no more the generation of the input data inside it neither the writing of the output files. These two functions are now inserted into the testbench. It is chosen to use csv files that are a particular type of calculus sheet where values are separated by commas. The definitions of new datatypes impact also the testbench.

The testbench is fundamental for a good synthesis because Catapult can automatically check, using Modelsim that the output after synthesis are equal to the ones of the testbench. This is effective thanks to ScVerify, an extension that compares the output of the testbench in C++ that is called *golden output* with the results of the simulation of the filter synthesise in VHDL or Verilog by Catapult, called *DUT output*.

We start to analyze the testbench from the Read function that is called first.

```

1 int ReadCSV_Samples(string filename , samplesVect_type &samples)
2 {
3     CsvParser *csvparser = CsvParser_new(filename.c_str() , " , " , 0);
4     CsvRow *row;
5     const CsvRow *header = CsvParser_getHeader(csvparser);
6
7     while ((row = CsvParser_getRow(csvparser)) )

```

```

8 {
9  const char **rowFields = CsvParser_getFields(row);
10 double double_stimulus_element;
11 stringstream(*rowFields) >> double_stimulus_element;
12
13 ac_fixed<DATA_TYPE::width, DATA_TYPE::i_width, DATA_TYPE::sign, AC.RND, AC.SAT.SYM>
    fixed_stimulus_element = double_stimulus_element;
14
15 samples.push_back(fixed_stimulus_element);
16 CsvParser_destroy_row(row);
17 }
18 cout << __FILE__ << ":" << __LINE__ << " - CSV file " << filename << " " << samples.
    size() << " samples were read in." << endl;
19 CsvParser_destroy(csvparser);
20 return samples.size();
21 }

```

Listing 3.9: Read function

For the Read function, the use of CSV files complicate a little bit the syntax. It is used the *csvparser.h* header to use function like *CsvParser_new*, *CsvParser_getHeader*, *CsvParser_getRow* and so on. Read function takes double elements from the file through the *stringstream* but then perform a cast because now input element of the filter must be *DATA_TYPE*, this cast is in line 13 of Listing 3.9. Samples read were stored in a *vector* element, a particularity of C++ language that differs from the classic *array* because is more flexible thanks to dynamic allocation. Elements can be accessed using offset on regular pointers to its element, like happens in Listing 3.10. A special iterator *it* that is initialize at the beginning of the vector and increments till the end of it, is needed. Then to access a single location we use the iterator as a pointer. To permit to ScVerify to understand which is the device under test, a particular instantiation of the filter must be done, reported in line 5 of Listing 3.10.

```

1 for (vector<DATA_TYPE>::iterator it = samples.begin(); it != samples.end(); ++it)
2 {
3     DATA_TYPE stimulus_element = *it;
4     OUT_TYPE exit_element;
5     CCS_DESIGN(fir_synth)(stimulus_element, exit_element);
6     samples_out.push_back(exit_element);
7 }

```

Listing 3.10: Testbench

The last function to analyze in the testbench is the Write function that simply take a vector of output data and write them into a CSV file. The script is not reported because is not enough interesting.

As it is shown in Fig.1.8 the HLS Verification is an important branch of Catapult tool. This is useful because permit to automatically verified the quality and the functioning of the automatically generated

code. The designer is not engaged into the verification process, it must only activate SCVerify option before the synthesis. SCVerify flow, within the Catapult HLS platform, automatically generates the infrastructure (a wrapper file) for verifying the functionality of the Catapult-generated RTL against the source code and reuses the original C++ Testbench. ScVerify starts a Modelsim session, and

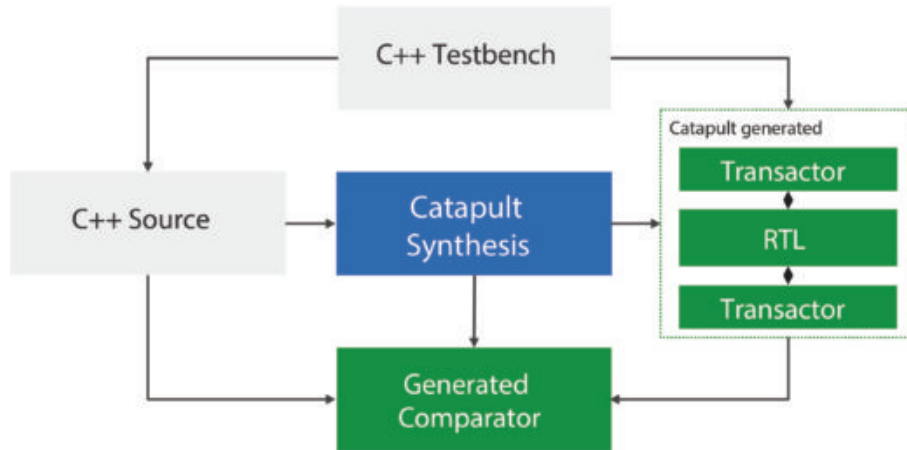


Figure 3.10: ScVerify

compare the output produces by the RTL code with the golden output from the testbench. In the end a report is generated where it is specified if the test is passed or not, how many input data are taken into account and how many data output we have and others informations. An example of this report is shown in Fig.3.11. This feature of Catapult HLS saves the designer a considerable amount of work and time as it don't have to manually write a RTL test bench each time the design specifications change.

```

# Info: Execution of user-supplied C++ testbench 'main()' has completed with exit code = 0
#
# Info: Collecting data completed
#   captured 770 values of i_sample
#   captured 770 values of y
#   captured 770 values of i_sample2
#   captured 770 values of y2
# Info: scverify_top/user_tb: Simulation completed
#
# Checking results
# 'y'
#   capture count      = 770
#   comparison count   = 770
#   ignore count       = 0
#   error count        = 0
#   stuck in dut fifo  = 0
#   stuck in golden fifo = 0
# 'y2'
#   capture count      = 770
#   comparison count   = 770
#   ignore count       = 0
#   error count        = 0
#   stuck in dut fifo  = 0
#   stuck in golden fifo = 0
#
# Info: scverify_top/user_tb: Simulation PASSED @ 5183152290 ps
# ** Note: (vsim-6574) SystemC simulation stopped by user.

```

Figure 3.11: Test result

3.7 Results

The results obtained in this case, with the filter described in this chapter are shown in Fig.3.12. These results are not significant because the filter take as input 770 sample per channel (half of them are 0) while in the Silicon Mitus architecture the zeroes elements are not passed as input, so the length of the input vector is a half. In their architecture the Zero-padding technique is applied internally at the filter with an optimization of the architecture that will be explain during the next chapter.


| Solution / | Latency... | Latency... | Throug... | Throug... | Total Area | Slack |
|---|------------|----------------|------------|----------------|------------------|-------------|
|  fir_synth.v1 (extract) | 516 | 6718.32 | 517 | 6731.34 | 242265.14 | 3.73 |

Figure 3.12: Results of the first synthesis

An important fact that can be learned from this architecture and that we will find also in the following ones, is the huge synthesized area for the registers. As it is shown in Fig.3.13, the percentage of area occupied by registers is the 73% of the all amount. It is important, with the optimizations, to reduce the number of registers to consequently decrease the total area.

| Area Scores | | | | |
|-------------------|-----------------|---------------|-----------------|----------------|
| | Post-Scheduling | | Post-DP & FSM | |
| | | | Post-Assignment | |
| Total Area Score: | 158479.6 | | 240157.0 | 242265.1 |
| Total Reg: | 107131.9 | (68%) | 175904.9 | 176052.0 (73%) |
| DataPath: | 158479.6 | (100%) | 240117.0 | (100%) |
| MUX: | 38626.8 | (24%) | 53645.6 | (22%) |
| FUNC: | 12720.8 | (8%) | 10102.3 | (4%) |
| LOGIC: | 0.0 | | 501.2 | (0%) |
| BUFFER: | 0.0 | | 0.0 | 0.0 |
| MEM: | 0.0 | | 0.0 | 0.0 |
| ROM: | 0.0 | | 0.0 | 0.0 |
| REG: | 107131.9 | (68%) | 175867.9 | (73%) |
| FSM: | 0.0 | | 40.0 | (0%) |
| FSM-REG: | 0.0 | | 37.0 | (93%) |
| FSM-COMB: | 0.0 | | 3.0 | (8%) |

Figure 3.13: Results of the first synthesis

CHAPTER 4

Architecture optimizations

4.1 Performances of SM filter

Silicon Mitus provided a filter described in Sistem Verilog that is manually optimized for occupying the least possible area. The RTL code was passed to Synopsys Design Compiler that synthesizes a gate-level architecture, using the library *Nangate45nm.dblm*. To synthesize in the correct way the filter, some directives were given to the tool. In particular, first of all, after reading the Verilog source files, a symbolic clock is created and it is set as *don't touch* because it is a special signal. Clock period is set to 13.02ns because the frequency of the circuit must be 76.8MHz from specifications. there are some directives on the clock properties:

```
create_clock -name MY_CLK -period 13.02 ckg_refclk
set_dont_touch_network MY_CLK
```

Now we want to specify that the clock could be affected by jitter and that every signal could have a delay with respect to the clock (delay of the input and output ports).

```
set_clock_uncertainty 0.07 [get_clocks MY_CLK]
set_input_delay 0.5 -max -clock MY_CLK [remove_from_collection [all_inputs] clk]
set_output_delay 0.5 -max -clock MY_CLK [all_outputs]
```

Finally it is possible to set the load of each output in our design, the chosen one is the input capacitance of a buffer (*BUF_X4*).

```
set OLOAD [load_of NangateOpenCellLibrary/BUF_X4/A]
set_load $OLOAD [all_outputs]
```

After having applied all these constraints it is possible to obtain an accurate estimation of area, power and timing (and consequently the maximum frequency) that are reported in Fig.4.1 and Fig.4.2.

| | |
|--------------------------------|---|
| Number of ports: | 138 |
| Number of nets: | 1640 |
| Number of cells: | 1280 |
| Number of combinational cells: | 1176 |
| Number of sequential cells: | 97 |
| Number of macros/black boxes: | 0 |
| Number of buf/inv: | 263 |
| Number of references: | 36 |
| | |
| Combinational area: | 23941.596185 |
| Buf/Inv area: | 2566.101947 |
| Noncombinational area: | 24062.360776 |
| Macro/Black Box area: | 0.000000 |
| Net Interconnect area: | undefined (Wire load has zero net area) |
| | |
| Total cell area: | 48003.956961 |

Figure 4.1: Area report for SM filter

As already said the most important design performance is the area that is estimated as $48000\mu m^2$. This will be the reference point for all the optimizations.

Power results are estimated with a switching activity of the clock and of all the others signals of 0.5. A more accurate analysis will be done using the back-annotation process on the architecture generated by Catapult but not on this one.

For what concern the maximum frequency the steps are several. The design should be synthesized with 0 ns clock period, at this point the timing shows a negative slack which means that obviously the clock period is not sufficient to compute the algorithm. This negative slack give us an estimation on which clock period can be necessary. Now a clock period equal to the slack has been used to synthesize again the design. Since Synopsys automatically changes the filter's structure, varying the clock frequency, for example is instantiated a ripple-carry adder rather than a carry-look-ahead, a process iteration is necessary until a null slack is obtained. In the end the period able to guarantee a null slack is 5.35ns that corresponds to a maximum frequency of about 187MHz.

4.2 Code changes

The purpose of this chapter is to explain which steps were done to arrive at the best solution of the filter in terms of area, starting from the synthesizable one described in Chapter 3.

Analysing the performances of this filter (Fig.3.13), pops up that the main percentage of the area is occupied by registers. The main difference with the architecture of SM is that the Zero-padding is done externally (so there are 770 input samples) or internally. That leads to the halve of input data (only non-zero elements are passed). Of course this implementation reduces the *Shift register* length

| Cell Internal Power = 2.5567 mW (94%) | | | | |
|--|--------------------|------------------|----------------|----------------------|
| Net Switching Power = 162.7111 uW (6%) | | | | |
| ----- | | | | |
| Total Dynamic Power = 2.7194 mW (100%) | | | | |
| Cell Leakage Power = 928.3768 uW | | | | |
| | | | | |
| Internal Power Group | Switching Power | Leakage Power | Total Power | Power (%) |
| ----- | | | | |
| io_pad | 0.0000 | 0.0000 | 0.0000 | 0.0000 (0.00%) |
| memory | 0.0000 | 0.0000 | 0.0000 | 0.0000 (0.00%) |
| black_box | 0.0000 | 0.0000 | 0.0000 | 0.0000 (0.00%) |
| clock_network | 0.0000 | 0.0000 | 0.0000 | 0.0000 (0.00%) |
| register | 2.4853e+03 | 2.0440 | 3.5956e+05 | 2.8468e+03 (78.04%) |
| sequential | 0.0000 | 0.0000 | 0.0000 | 0.0000 (0.00%) |
| combinational | 71.4248 | 160.6684 | 5.6881e+05 | 800.9014 (21.96%) |
| ----- | | | | |
| Total | 2.5567e+03 uW | 162.7123 uW | 9.2837e+05 nW | 3.6477e+03 uW |

Figure 4.2: Power report for SM filter

and also significantly the number of registers.

To better explain what is going on, we take as reference a structure very similar to the one under design but with a lower order. The order of the original filter is 128 while in the next figure we refer to a filter with an even order of 12. Now what is important to notice is that, half of the input are zeroes, but moreover, that every 2 clk cycles data are in the same positions and are multiplied for the same coefficients. So for example in the highlighted clocks (the odd ones) data are multiplied for the even coefficients (c_0 , c_2 , c_4 and so on) while in the white clocks data are multiplied by the odd coefficients (c_1 , c_3 and so on). This property is very useful because now we want to pass as input only non-zero data, but the output frequency (and of course values) must be unchanged. For example, in clock cycles XV and XVI the output are:

$$XV : D8c_0 + D7c_2 + D6c_4 + D5c_6 + D4c_8 + D3c_{10} + D2c_{12} \quad (4.1)$$

$$XVI : D8c_1 + D7c_3 + D6c_5 + D5c_7 + D4c_9 + D3c_{11} \quad (4.2)$$

In clock cycle XV data are multiplied by even coefficients, then they are shifted because a null data enters. In the new clock cycle (XVI) the same data are multiplied by all the odd coefficients.

This mechanism can be substituted, without zero-input by the multiplication of one data at the same time for an even coefficient and for the following odd coefficient. The two products are then distributed on two different clock cycles to maintain the same frequency. The reference architecture becomes the one in Fig.4.4 with an order of 6.

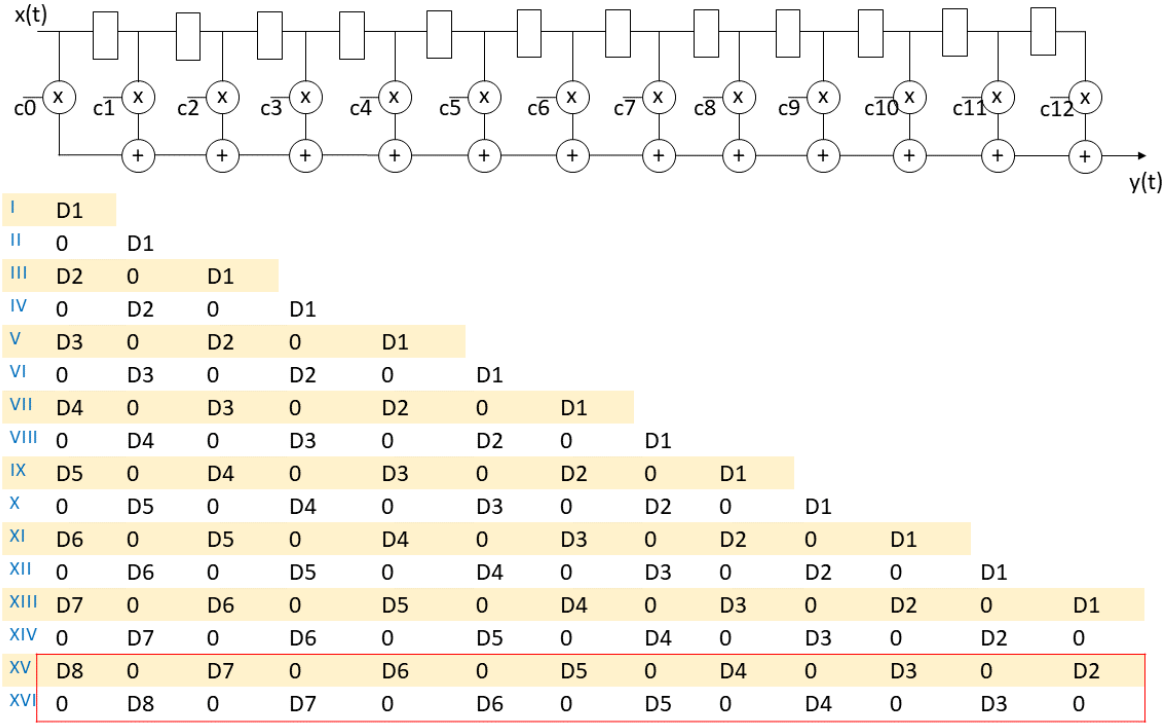


Figure 4.3: FIR with order 12

To describe this architecture, some changes must be done in the High-level code. The *MAC LOOP* must be separated in parts. One that describes what happens with even coefficients and one for the odd ones.

```

1 MACLOOP_E: for (i=0; i<ORDER; i++){
2     sum += sample[i]*coeff[2*i];
3 }
4 MACLOOP_O: for (i=0; i<ORDER-1; i++){
5     sum += sample[i]*coeff[(2*i)+1];
6 }

```

Listing 4.1: fir_synth_fix.cpp

This optimization is done to reduce the area occupied, but it is not the only one. Another solution is possible thanks to the coefficients' symmetry. Coefficients are 129 and are symmetric with respect to the central one (the 65th). Central coefficient, in the reference architecture of Fig.4.4, is *c6* (circled of orange). So the FIR becomes the one in Fig.4.5.

The ROM memory which stores the coefficients, thanks to this solution, can save only half of the coefficients, plus the central one. In the original filter it is possible to save now only 65 coefficients, or better 64+1 to have a ROM addressable with only 6 bits. Moreover, now it is possible to sum two samples that must be multiplied by the same coefficient to halve the loops cycles. This leads to an advantage because this addition can be performed in the same clock cycle of the multiplication.

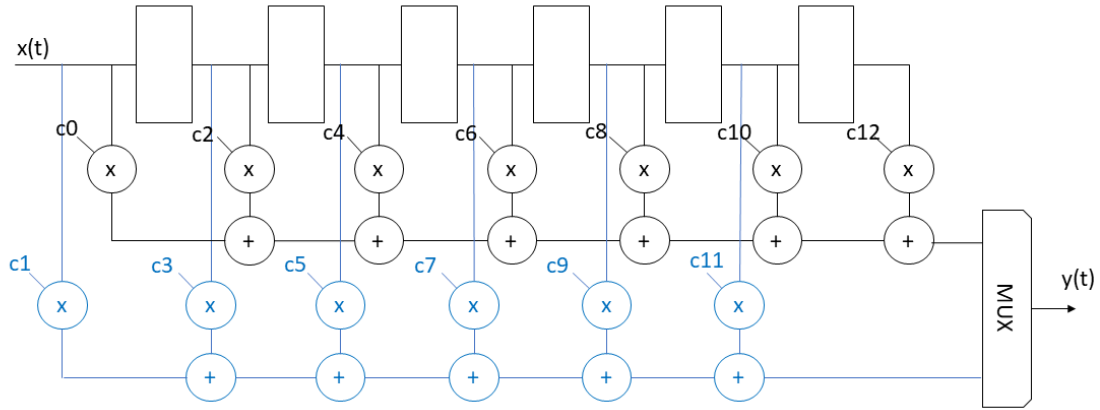


Figure 4.4: FIR with order 6

For our purpose this is not a great area optimization because in most of the later architectures the loops are rolled, then only one multiplier and only one adder will be instantiated. MAC loop for even coefficient in this case has one more element than the odd one. To have the same interval for the i counter variable, an addition must be added at the end of the *MAC_LOOP_E*. The new version of the code is reported in Listing 4.5

```

1  MAC_LOOP_E: for (i=0; i < (ORD/2) - 1; i++){
2      sum += (sample[i] + sample[ORD-i]) * coeff[2*i];
3  }
4      sum += sample[ORD/2] * coeff[ORD];
5
6  MAC_LOOP_O: for (i=0; i < (ORD/2) - 1; i++){
7      sum += (sample[i] + sample[ORD-1-i]) * coeff[(2*i)+1];
8  }

```

Listing 4.2: Loop for symmetric coefficients

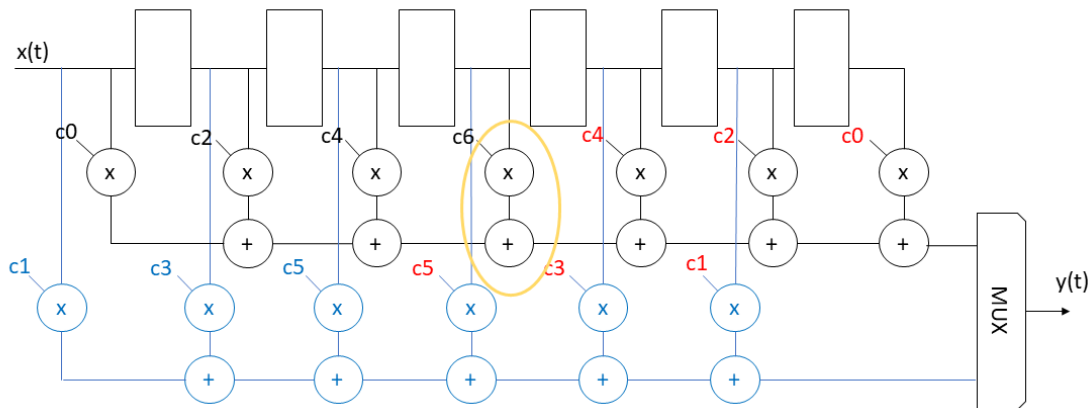


Figure 4.5: FIR with order 6 and symmetric coefficients

Starting from a filter with an even coefficient, dividing by two the order it is possible to have two different situations:

1. The new filter has an EVEN order
2. The new filter has an ODD order

The previous structures belongs to the first case because the initial order was 12 and divided by two results 6. This choice was because the SM filter has an original order of 128 so the new order is 64. But it is important to consider that the algorithm in Listing 4.5 is not valid for both situations illustrated above. In the following it is demonstrate why. Take as an example a starting filter with order 14. When it is divided the resulting order is 7 as in Fig.4.6. The same structure is reported

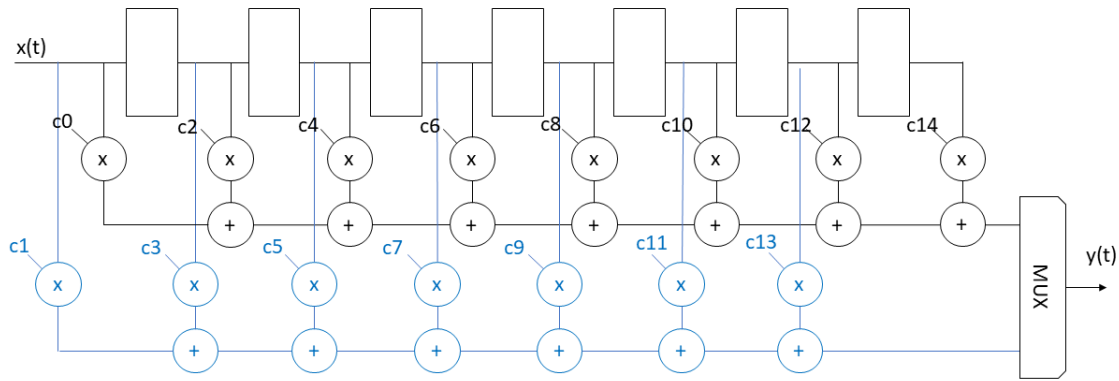


Figure 4.6: FIR with order 7

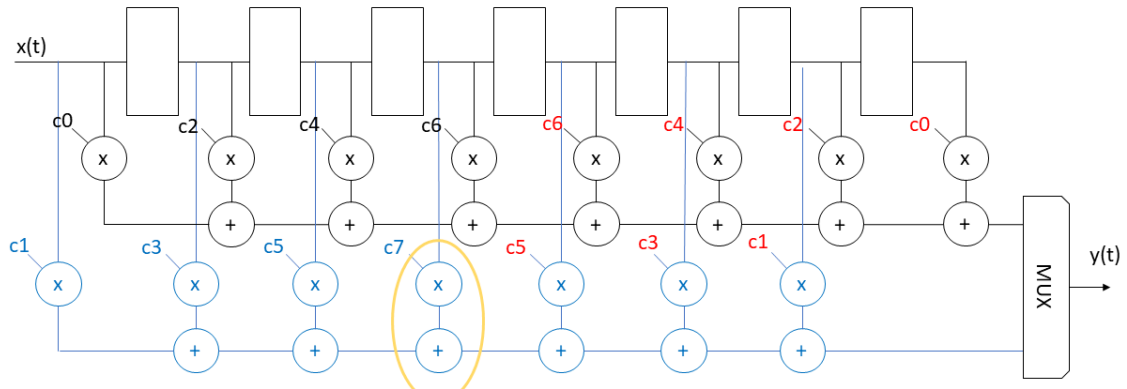


Figure 4.7: FIR with order 7 and symmetric coefficients

in Fig.4.7 where symmetric coefficients are underlined. In this case the central one is $c7$ (circled of orange) that is an ODD coefficient so the code must undergo a small variation in the position of the extra summation.

```

1  MACLOOP_E: for(i=0; i<(ORD/2)-1; i++){
2      sum += (sample[i]+sample[ORD-i])*coeff[2*i];
3  }
4
5  MACLOOP_O: for(i=0; i<(ORD/2)-1; i++){
6      sum += (sample[i]+sample[ORD-1-i])*coeff[(2*i)+1];
7  }
8  sum += sample[(ORD/2)-1]*coeff[ORD];

```

Listing 4.3: Loop for symmetric coefficients and ODD order

4.3 The ac_channel Class Definition

There is a last change that is necessary on the code. The problem is that now, for every input data, two outputs are required, because this is an upsampling filter.

Input data arrive, from specifics, with a frequency of 384 kHz that means every 200 clock cycles (remember that clk frequency is 76.8 MHz), while the output must be synchronized to change every 100 clk as in Fig.4.8.

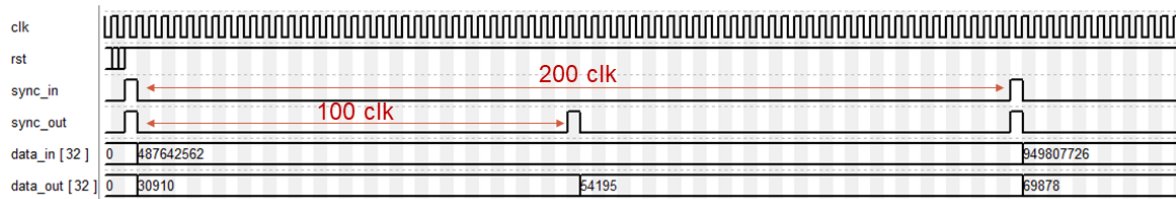


Figure 4.8: Data_in and data_out frequency

In the tesbench, for every input data, the filter is recalled only once but two outputs must be generated. The solution is to use a particular class for the HLS that is *ac_channel*. This class is useful when the frequency of the I/O changes within the design. That is the case of upsampling or downsampling filters.

As specified in the Mentor Graphics guide [18], the *ac_channel* class is a C++ template class that enforces a FIFO discipline (reads occur in the same order as writes). That is, for modeling purposes, an *ac_channel* is infinite in length (writes always succeed) while attempting to read from an empty channel generates an assertion failure (reads are blocking). If a channel appears in the top-level interface of a design and is only written by the design, then the external environment is assumed to be the unique reader. Likewise, the external environment is assumed to be the writer of a channel which is only read.

For the purpose of the thesis this is a good solution because, defining different *ac_channel* for the input/output of every channel of the filter, it is possible to write two different results in the output

channel and read them both within the same call in the testbench.

The definition of the function in the header file becomes:

```
1 void fir_synth(ac_channel<DATA_TYPE> &, ac_channel<DATA_TYPE> &,
2               ac_channel<DATA_TYPE> &, ac_channel<DATA_TYPE> &, bool);
```

Listing 4.4: Loop for symmetric coefficients and ODD order

While in the .cpp file four different channels are defined, two for every filter channel (ch and ch2), it is changed (with respect to Listing 3.5) like:

```
1 void fir_synth(ac_channel<DATA_TYPE> &in_ch, ac_channel<DATA_TYPE> &out_ch,
2               ac_channel<DATA_TYPE> &in_ch2, ac_channel<DATA_TYPE> &out_ch2, bool enable)
```

Listing 4.5: Loop for symmetric coefficients and ODD order

To access the *ac_channel* class specific functions are necessary, like *.read()* and *.write()*. Other specific functions not used in this work are listed in the guide for *Ac_data_types* [18].

4.4 Results given by the optimized code

To summarize, the optimizations on the code analyzed in previous section, are reported here:

1. Half of the input data are 0 \longrightarrow Reduction of the shift register by a factor of two.
2. In a clock cycle all data are multiplied by even coefficients, then by the odd coefficients in the next clock cycle.
 - XV: D8c0+D7c2+D6c4+D5c6+D4c8+D3c10+D2c12 \longrightarrow Only even coefficients!
 - XVI: D8c1+D7c3+D6c5+D5c7+D4c9+D3c11 \longrightarrow Only odd coefficients!
3. If the coefficients are symmetric:
 - XV: D8c0+D7c2+D6c4+D5c6+D4c4+D3c2+D2c0 \longrightarrow I can save only half of them
 - XVI: D8c1+D7c3+D6c5+D5c5+D4c3+D3c1

In the end the *ac_channel* class to perform the upsampling of the data was introduced.

The description is now ready to be synthesized by Catapult, and this time the performances can be compared to SM ones (Section 4.1) because the filter's structure is very similar. This first description of the filter will be from now on called *Rolled Loop* because by default Catapult synthesizes the solution with rolled Shift and Mac structures. Resulting architecture has the performances in Fig.4.9. Thanks to all the optimizations explained above, the area is decreasing with respect of the filter described at


| Solution / | Latency... | Latency... | Throug... | Throug... | Total Area | Slack |
|---|------------|----------------|------------|----------------|------------------|-------------|
|  fir_synth.v1 (extract) | 167 | 2174.34 | 168 | 2187.36 | 153941.76 | 9.17 |

Figure 4.9: Summary table of Rolled Loop solution

the end of Chapter 3, but it is still far away from the performances of SM filter. It is interesting for this first filter to report also some of the possible informations that Catapult gives us, so for example almost the complete *Bill of materials* is reported in the following images. Fig.4.10 is an overview of all the elemental blocks instantiated by Catapult, repeated more in detail (with also an area estimation) in Fig.4.13. Then it is also useful to show the table in Fig.4.12, a summary of the contributions to the total area.

| Operator Bitwidth Summary | | | | | |
|---|-------------|-------|------------|-------------|-------|
| Operation | Size (bits) | Count | Operation | Size (bits) | Count |
| ROM_1i5_1o26_4d055b8fedc1141da1ae4a8b418e4c4830 | 26 | 1 | nand | 1 | 3 |
| ROM_1i5_1o29_eb88306ebb957b52e2c40065b2f25cb430 | 29 | 1 | nor | 1 | 11 |
| add | 7 | 1 | not | 6 | 1 |
| - | 60 | 5 | - | 5 | 1 |
| - | 6 | 2 | - | 32 | 12 |
| - | 5 | 1 | - | 1 | 38 |
| - | 38 | 2 | or | 1 | 100 |
| - | 35 | 8 | - | - | - |
| - | 34 | 2 | read_port | 32 | 2 |
| - | 33 | 6 | - | 1 | 1 |
| - | 32 | 7 | read_sync | 0 | 1 |
| and | 60 | 4 | - | - | - |
| - | 6 | 1 | reg | 60 | 7 |
| - | 5 | 2 | - | 6 | 2 |
| - | 1 | 41 | - | 5 | 2 |
| mul | 60 | 1 | - | 38 | 1 |
| - | 59 | 1 | - | 34 | 1 |
| mux | 60 | 4 | - | 33 | 2 |
| - | 5 | 3 | - | 32 | 390 |
| - | 38 | 1 | - | 29 | 1 |
| - | 33 | 1 | - | 2 | 4 |
| - | 32 | 268 | write_port | 1 | 14 |
| mux1h | 60 | 2 | - | 0 | 2 |
| - | 29 | 1 | - | - | - |

Figure 4.10: Complete list of objects instantiated

There are some interesting things to discuss. The first one is regarding the multipliers instantiated. As it is possible to see in all the reports there are two multipliers instead of one unique for the SM structure. This can be modified, optimizing the architecture thanks to Catapult flexibility. The second one is about the ROM memory used to store the coefficients.

It is possible to notice that in Figures 4.13 and 4.10 two memories are listed instead of one unique. Looking only at the BoM it is not possible to understand clearly the reason of this choice, it is only reported that there is a ROM addressable with 5bit (so 32 elements) and a parallelism of 26bit while the other one has the same dimension but a parallelism of 29bit. In 4.13 is also specified the area of these elements.

Looking at the memories (coloured in violet) in the snippet of datapath reported in Fig.4.11, we can see how the memories are connected to the MAC structure. The ROM_1i5_1o26 is connected only to the MAC_LOOP_E thus must contains only the even coefficients while the ROM_1i5_1o29 contains only the odd coefficients and is connected to the other multiplier. Catapult optimizes the memory

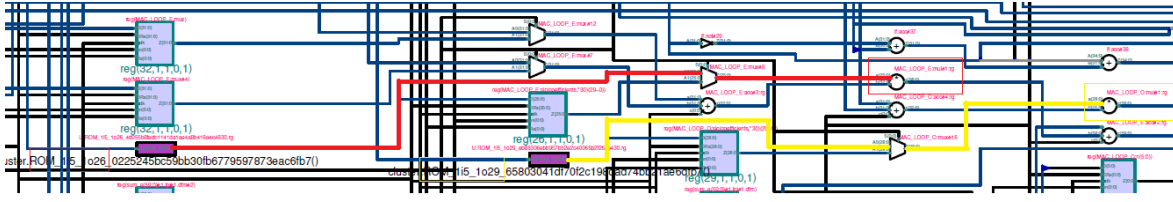


Figure 4.11: Two independent ROMs

dividing it into two memories, starting from an initial indication of the designer who defines a unique vector of `COEFF_TYPE` on 30bit (For the full code, refers to Appendix A).

Furthermore Catapult optimizes every single memory looking at the effective dimension of the even and odd coefficients. This type of optimization is automatically done also for multipliers and adders. Defining types *PROD_TYPE* and *SUM_TYPE* (Listing 3.8) gives to Catapult the possibility to have freedom for choosing the better operational block. As a matter of fact, two different multipliers are instantiated with different parallelism and, as consequence, different area.

In Fig.4.13 is shown that a multiplier that has two inputs of 26 bits and 33 bits has an output on 59 bits and consequently an area of $6676\mu m^2$ and it obviously will be used for the even coefficients. The other multiplier has higher parallelism, its output is on 60 bits and its area is around $7451\mu m^2$.

In the end, as already said in Chapter 3, in the *Rolled Loop* Architecture the main part of the area is occupied by registers. The filter has two channels with an order of 64. We would expect about 128 registers for the shift part. Unfortunately there are 390 synthesized registers (Fig.4.10). This because Catapult has no informations about which operations must be in series and which ones can be in parallel, so it instantiates a lot of registers to save temporary variables and by consequence a lot of muxes (268 muxes are instantiated).

| Area Scores | | | | |
|-------------------|------------------------|------------------------|------------------------|--|
| | Post-Scheduling | Post-DP & FSM | Post-Assignment | |
| Total Area Score: | 113748.8 | 177417.7 | 153941.8 | |
| Total Reg: | 82275.8 (72%) | 96670.9 (54%) | 96670.9 (63%) | |
| DataPath: | 113748.8 (100%) | 177346.7 (100%) | 153870.8 (100%) | |
| MUX: | 15350.2 (13%) | 54610.0 (31%) | 29397.4 (19%) | |
| FUNC: | 16122.9 (14%) | 25725.3 (15%) | 27183.5 (18%) | |
| LOGIC: | 0.0 | 404.4 (0%) | 683.0 (0%) | |
| BUFFER: | 0.0 | 0.0 | 0.0 | |
| MEM: | 0.0 | 0.0 | 0.0 | |
| ROM: | 0.0 | 0.0 | 0.0 | |
| REG: | 82275.8 (72%) | 96606.9 (54%) | 96606.9 (63%) | |
| FSM: | 0.0 | 71.0 (0%) | 71.0 (0%) | |
| FSM-REG: | 0.0 | 64.0 (90%) | 64.0 (90%) | |
| FSM-COMB: | 0.0 | 7.0 (10%) | 7.0 (10%) | |

Figure 4.12: Statistics on the area

With a comparison between this solution and the one proposed in Chapter 3 (Fig.3.13), results are impressive. The area occupied by registers and by muxes is almost halved, instead the functional logic

obviously increases as there is lower availability of resources like multiplier and registers. Registers occupy the 63% of the total area but also the muxes area is not negligible. The target is to reduce them to get as close as possible to the area of SM.

In the following is reported a comparison between the Silicon Mitus filter performances and the *Rolled Loop* filter ones. The filter optimized by SM has far better performances, so it is necessary another type of optimization, that works directly on the structure of the filter and uses technique like pipelining or unrolling. This will be available directly using the Catapult tool interface and will be explained in the following chapter.

| | FIR SM | Rolled Loop |
|--------------------------|---------------|--------------------|
| min Period | 5.35ns | 6.72ns |
| Combinational area: | 23941.59 | 52978.15 |
| Buf/Inv area: | 2566.10 | 5282.49 |
| Noncombinational area: | 24062.36 | 59306.02 |
| Total area (μm^2) | 48003.95 | 112284.18 |
| Internal P | 2.556 | 7.039 |
| Switching P | 0.162 | 0.313 |
| Leakage P | 0.928 | 2.543 |
| Total P (mW) | 3.646 | 9.896 |

Table 4.1: Performances comparison

To conclude this chapter in Fig.4.14 is reported also the complete RTL of this *Rolled Loop* from which was taken the frame in Fig.4.11. The gray part of the RTL represents the FSM and the control part while the datapath is highlighted.

| Bill Of Materials (Datapath) | | | | | |
|---|------------|-------|-------|------------|-------------|
| Component Name | Area | Score | Delay | Post Alloc | Post Assign |
| [Lib: ccs_ioport] | | | | | |
| ccs_in(5,1) | 0.000 | 0.000 | | 1 | 1 |
| ccs_in_wait(1,32) | 0.000 | 0.000 | | 1 | 1 |
| ccs_in_wait(3,32) | 0.000 | 0.000 | | 1 | 1 |
| ccs_out_wait(2,32) | 0.000 | 0.000 | | 1 | 1 |
| ccs_out_wait(4,32) | 0.000 | 0.000 | | 1 | 1 |
| [Lib: cluster] | | | | | |
| ROM_l15_lo26_0225245bc59bb30fb6779597873eac6fb7() | 816.315 | 0.250 | | 1 | 1 |
| ROM_l15_lo29_65803041df70f2c198dad74bb21ae6dfb7() | 910.505 | 0.250 | | 1 | 1 |
| [Lib: mgc_ioport] | | | | | |
| mgc_io_sync(0) | 0.000 | 0.000 | | 1 | 1 |
| [Lib: nangate-45nm_beh] | | | | | |
| mgc_add(32,0,1,0,32,1) | 170.778 | 0.225 | | 0 | 3 |
| mgc_add(32,0,1,0,32,3) | 103.722 | 0.402 | | 1 | 0 |
| mgc_add(32,0,31,1,32,1) | 329.069 | 0.312 | | 0 | 4 |
| mgc_add(32,1,26,1,33,1) | 330.892 | 0.302 | | 0 | 2 |
| mgc_add(32,1,30,1,33,1) | 332.710 | 0.300 | | 0 | 2 |
| mgc_add(32,1,32,1,33,1) | 333.620 | 0.300 | | 0 | 2 |
| mgc_add(32,1,32,1,33,3) | 217.385 | 0.713 | | 1 | 0 |
| mgc_add(34,0,33,1,34,1) | 350.293 | 0.318 | | 0 | 2 |
| mgc_add(34,1,32,1,35,1) | 354.324 | 0.306 | | 0 | 2 |
| mgc_add(34,1,33,1,35,1) | 354.779 | 0.305 | | 0 | 4 |
| mgc_add(35,0,32,1,35,1) | 360.838 | 0.320 | | 0 | 2 |
| mgc_add(37,1,33,1,38,1) | 385.835 | 0.313 | | 0 | 2 |
| mgc_add(42,1,40,1,43,3) | 287.565 | 0.736 | | 1 | 0 |
| mgc_add(44,1,43,1,45,3) | 301.044 | 0.726 | | 2 | 0 |
| mgc_add(46,1,45,1,47,3) | 314.921 | 0.724 | | 2 | 0 |
| mgc_add(5,0,1,0,5,3) | 17.702 | 0.053 | | 0 | 1 |
| mgc_add(5,0,1,0,6,3) | 17.702 | 0.053 | | 0 | 1 |
| mgc_add(5,0,2,1,6,4) | 23.663 | 0.414 | | 1 | 0 |
| mgc_add(51,1,47,1,52,3) | 350.807 | 0.744 | | 1 | 0 |
| mgc_add(55,1,53,1,56,2) | 423.410 | 0.603 | | 1 | 0 |
| mgc_add(55,1,53,1,56,3) | 377.764 | 0.716 | | 1 | 0 |
| mgc_add(57,0,56,1,57,3) | 391.765 | 0.802 | | 1 | 0 |
| mgc_add(6,0,1,1,6,2) | 52.086 | 0.124 | | 0 | 1 |
| mgc_add(6,0,1,1,6,4) | 27.962 | 0.482 | | 1 | 0 |
| mgc_add(60,0,59,1,60,1) | 626.203 | 0.378 | | 0 | 2 |
| mgc_add(60,0,59,1,60,3) | 412.569 | 0.800 | | 1 | 0 |
| mgc_add(60,0,59,1,60,4) | 257.583 | 4.396 | | 2 | 0 |
| mgc_add(60,0,60,0,60,1) | 603.934 | 0.371 | | 0 | 3 |
| mgc_add(60,0,60,0,60,3) | 402.056 | 0.864 | | 1 | 0 |
| mgc_add(60,0,60,0,60,4) | 255.039 | 4.392 | | 2 | 0 |
| mgc_add(7,0,1,0,7,2) | 32.527 | 0.104 | | 0 | 1 |
| mgc_add(7,0,2,1,7,4) | 32.215 | 0.555 | | 1 | 0 |
| mgc_add(7,0,7,0,7,4) | 29.886 | 0.536 | | 1 | 0 |
| mgc_and(1,2,1) | 1.110 | 0.024 | | 0 | 40 |
| mgc_and(1,4,1) | 1.660 | 0.051 | | 0 | 1 |
| mgc_and(5,2,1) | 5.548 | 0.024 | | 0 | 2 |
| mgc_and(6,2,1) | 6.658 | 0.024 | | 0 | 1 |
| mgc_and(60,2,1) | 66.575 | 0.024 | | 0 | 4 |
| mgc_mul(26,1,33,1,59,1) | 6676.446 | 0.978 | | 0 | 1 |
| mgc_mul(26,1,33,1,59,4) | 4280.237 | 4.245 | | 1 | 0 |
| mgc_mul(29,1,33,1,60,1) | 7451.217 | 0.987 | | 0 | 1 |
| mgc_mul(29,1,33,1,60,4) | 4777.873 | 4.336 | | 1 | 0 |
| mgc_mux(32,1,2,4) | 56.142 | 0.066 | | 130 | 260 |
| mgc_mux(32,5,32,3) | 1378.699 | 0.174 | | 0 | 6 |
| mgc_mux(32,5,32,4) | 1004.695 | 0.250 | | 4 | 0 |
| mgc_mux(32,6,64,3) | 2781.271 | 0.208 | | 0 | 2 |
| mgc_mux(32,6,64,4) | 2016.486 | 0.296 | | 2 | 0 |
| mgc_mux(33,1,2,4) | 57.896 | 0.066 | | 0 | 1 |
| mgc_mux(38,1,2,4) | 66.668 | 0.066 | | 0 | 1 |
| mgc_mux(5,1,2,4) | 8.772 | 0.066 | | 0 | 3 |
| mgc_mux(60,1,2,4) | 105.265 | 0.066 | | 0 | 4 |
| mgc_mux1hot(29,3,4) | 76.659 | 0.061 | | 0 | 1 |
| mgc_mux1hot(60,3,4) | 158.605 | 0.061 | | 0 | 2 |
| mgc_nand(1,2,1) | 0.771 | 0.012 | | 0 | 2 |
| mgc_nand(1,3,1) | 1.046 | 0.028 | | 0 | 1 |
| mgc_nor(1,2,1) | 0.771 | 0.025 | | 0 | 11 |
| mgc_not(1,1) | 0.529 | 0.006 | | 0 | 38 |
| mgc_not(32,1) | 16.941 | 0.006 | | 0 | 12 |
| mgc_not(5,1) | 2.647 | 0.006 | | 0 | 1 |
| mgc_not(6,1) | 3.176 | 0.006 | | 0 | 1 |
| mgc_or(1,2,1) | 1.110 | 0.031 | | 0 | 95 |
| mgc_or(1,3,4) | 1.385 | 0.066 | | 0 | 4 |
| mgc_or(1,4,4) | 1.660 | 0.078 | | 0 | 1 |
| mgc_reg_pos(1,0,0,1,1,0,0,4) | 6.375 | 0.087 | | 0 | 9 |
| mgc_reg_pos(1,0,0,1,1,1,1,4) | 7.374 | 0.092 | | 0 | 5 |
| mgc_reg_pos(2,0,0,1,1,1,1,4) | 14.748 | 0.092 | | 0 | 4 |
| mgc_reg_pos(29,0,0,1,1,1,1,4) | 213.845 | 0.092 | | 0 | 1 |
| mgc_reg_pos(32,0,0,1,1,0,0,4) | 204.000 | 0.087 | | 0 | 2 |
| mgc_reg_pos(32,0,0,1,1,1,1,4) | 235.966 | 0.092 | | 0 | 388 |
| mgc_reg_pos(33,0,0,1,1,1,1,4) | 243.340 | 0.092 | | 0 | 2 |
| mgc_reg_pos(34,0,0,1,1,1,1,4) | 250.714 | 0.092 | | 0 | 1 |
| mgc_reg_pos(38,0,0,1,1,1,1,4) | 280.210 | 0.092 | | 0 | 1 |
| mgc_reg_pos(5,0,0,1,1,1,1,4) | 36.870 | 0.092 | | 0 | 2 |
| mgc_reg_pos(6,0,0,1,1,1,1,4) | 44.244 | 0.092 | | 0 | 2 |
| mgc_reg_pos(60,0,0,1,1,1,1,4) | 442.437 | 0.092 | | 0 | 7 |
| TOTAL AREA (After Assignment): | 153870.761 | | | | |

Figure 4.13: Complete list of objects instantiated with area estimation

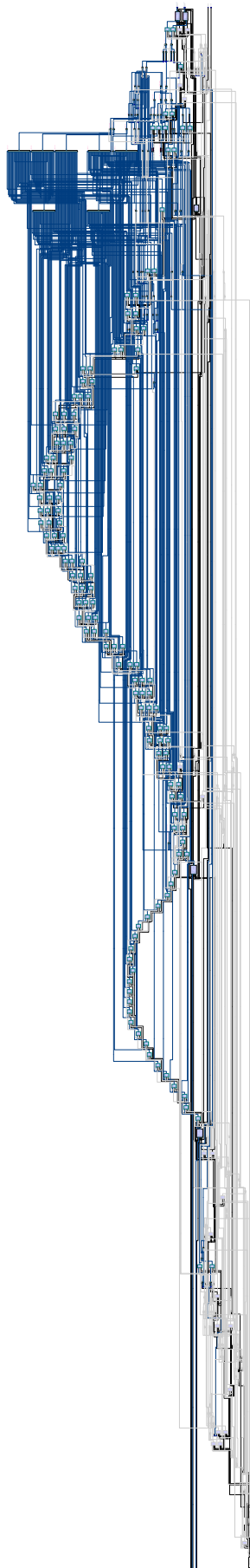


Figure 4.14: Overall RTL of the Rolled Loop

CHAPTER 5

Catapult optimizations

5.1 Recap of the GUI parameters

One of the most important benefits of the HLS is that, using the same code, without any changes, it is possible to explore elevate combination of solutions that with a traditional design would need a rewriting of the HDL code for every change the designer wants to make. This leads to the possibility of changing the structure of the filter without worrying about the code so in a very short time many solutions are tested.

During Chapter 1 most of the possibilities that Catapult tool gives to the designer have been discussed, in particular during the Mapping and the Architecture steps.

The first one is about the definition of clock, interface signals, resets and enable. Clock is fixed to 76.8MHz from specifications. The interfaces signals to synchronize inputs and outputs are chosen with the *wait handshake* (remember that *ac_channel* type has for definition this type of handshake). Enable signal is not set directly from the Catapult GUI but it is inserted in the code description of the filter as an *if* condition that comprehends both the SHIFT and the MACs loops. It has no handshake. It is only a single signal because the *transaction done* has been disabled (we have analysed this signal in Paragraph 1.4.1).

For what concerns the reset signals, it is possible to insert automatically a synchronous reset or an asynchronous one in the architecture. The Silicon Mitus architecture has both the resets however only the asynchronous one is an interface signal, it enters in the filter and then, with some logic, the other one is generated.

Starting from this assumption, among the architectures that were generated and tested with Catapult and that will be presented in the following chapter, there are some with one specific reset and others with both resets, to underline also which could be the differences in terms of area and power.

These are the possible optimizations that can be introduced in the Catapult Mapping step. All the

possible combinations between Architecture options are not reported here because they have already been largely described in Paragraph 1.4.1.

In Silicon Mitus structure there is only one MAC block and the latency of the overall architecture is 64 clock cycles that is entirely occupied by the delay group of the filter.

Before introducing the optimized architectures, it is important to underline a feature of Catapult HLS at which is dedicated the following subsection: the Cluster detection. This feature is used in some intermediate solutions but not in the final architecture.

5.1.1 Clusters

The Operator Clustering flow detects and optimizes groups of related data path operators such as adder trees, multiply-add, and squares. This methodology attempts to reduce many of the inefficiencies with fine-grained scheduling of operators. This targeted clustering approach can lead to an improved quality of results with better area and timing correlation with RTL synthesis results. Clustering offers the following key benefits:

- Better area and timing correlation with RTL synthesis
- Smaller area due to course-grained sharing
- Increased capacity due to reduced design complexity
- Faster runtimes on complex designs with many synthesis iterations
- Shorter design latency for lower power and smaller area

Catapult performs arithmetic decomposition and gate-level optimizations on the extracted clusters to create a highly optimized data path component that is annotated with very accurate area and timing information. Since clustering extracts a specific arithmetic operator, Catapult can generate a more accurate timing and area estimation because it is able to optimize at the technology gate level. By default, Catapult will implement the cluster using low level technology cells from the Catapult library (eg. Full/half adders). Alternatively, you can use your RTL Synthesis tool to optimize the cluster by setting the `CLUSTER_RTL_SYN` directive/pragma.

5.2 Unrolling of the SHIFT LOOP

The first possible optimization to decrease the area, is the unrolling of the shift loop, partially or totally. This is an optimization suggested in the Catapult guide, that is not easy to understand otherwise: the hardware used in the *Rolled Loop* architecture to implement the shift register is only a multiplexer, controlled by the loop counter [19]. Taking the filter described in Section 4.4, the first

optimization could be to unroll the shift loop in order to decrease the area. A big multiplexer, that is not area-optimized, will be replaced by a shift register that has the length equal to the unrolling factor (U) and that takes only a single clock cycle to shift all the values by one position when a new input sample is coming. Remember that with this architecture, a fully unrolled SHIFT LOOP is implemented with 65 registers.

In the following table performances in case of half unrolling ($U=32$) and fully unrolling are reported. All the other values of factor U lead to architecture with bad area optimization. In Fig.5.1 and Fig.5.2 the v2 is the partial unrolling, while the v3 is the total unrolling that means instantiates 64 registers.




| Solution / | Latency... | Latency... | Throug... | Throug... | Total Area | Slack |
|--|------------|------------|-----------|-----------|------------|-------|
|  fir_synth.v1 (extract) | 167 | 2174.34 | 168 | 2187.36 | 153941.76 | 9.17 |
|  fir_synth.v2 (extract) | 105 | 1367.10 | 106 | 1380.12 | 147878.32 | 9.16 |
|  fir_synth.v3 (extract) | 133 | 1731.66 | 134 | 1744.68 | 71494.00 | 9.16 |

Figure 5.1: Different shift-unrolling performances

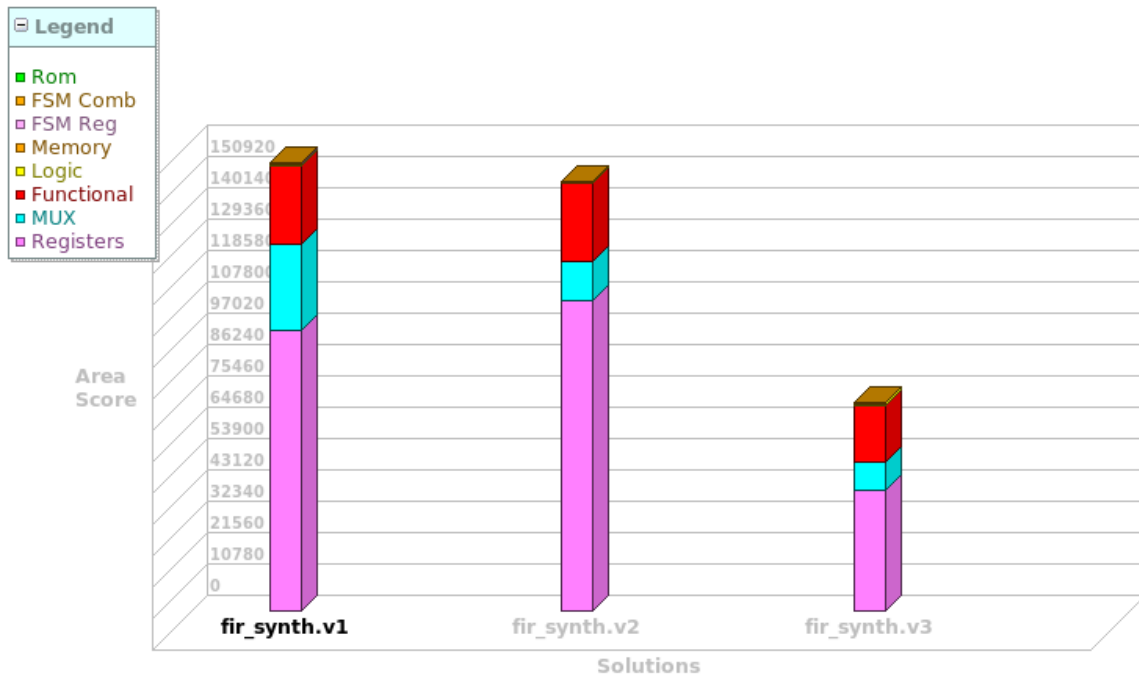


Figure 5.2: Shift unrolling area comparison

The total unrolling of the SHIFT LOOP leads to an extreme reduction of the number of registers, in particular the 32bit regs are reduced from 390 of Fig.4.12 to 142 in the BOM represented in Fig.5.6. Not only registers but also muxes are decreased, as a matter of fact, with respect to the *Rolled Loop*, the number of mux with 32bit of parallelism are only 14. The complete unrolling permits to instantiate only one multiplier, using four clocks cycles to perform EVEN and ODD MAC LOOP for both the

channels, this is explained in Fig.5.3.

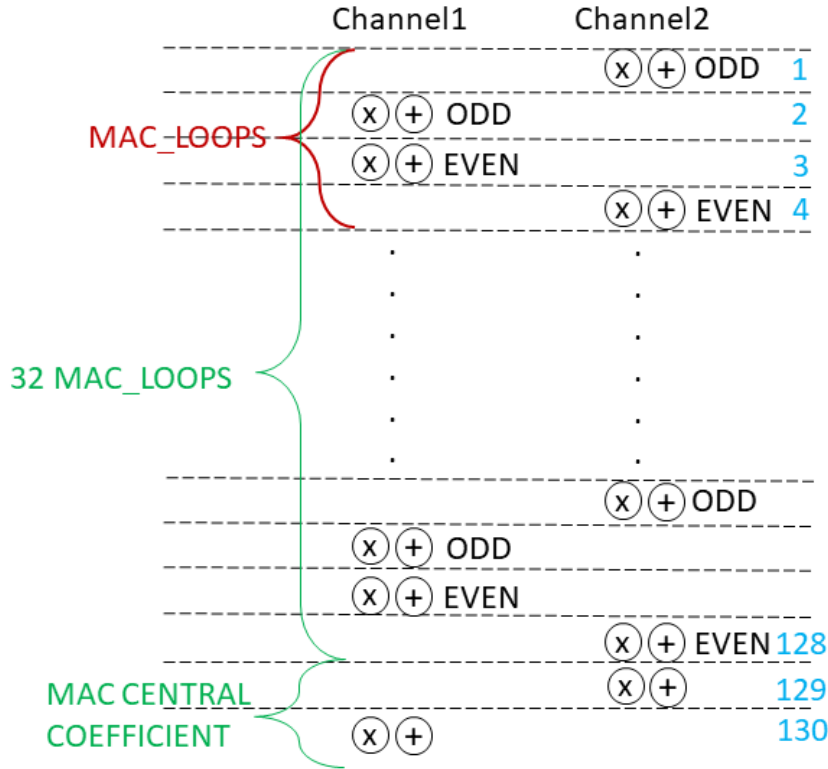


Figure 5.3: Scheduling of the Shift full unrolled filter

The multiplications are scheduled alternated for the two channels but there is no pipelining. This means that after 129 clock cycles the output of channel 2 is ready and after another clock cycle the result of channel 1 is ready too. In the end it takes 130 clock cycles to perform both the results because they are calculated in parallel. This scheduling is taken from the Gantt chart that is reported in Fig.5.4 where it is possible to see that there is an internal loop made of four clock cycles (that is repeated 32 times), then to calculate the throughput there must be added 6 clock cycles. These are subdivided taking the inputs, calculating the last operations for the central coefficient and rounding (C3 and C4) and finally generating the outputs (C5 and C6). In this way we arrive at the final throughput that is 134 clock cycles as reported in the table in Fig.5.1. While in the case of partial unrolling ($U=32$) the multipliers must be at least two, thanks to a different scheduling that is not analyzed, the overall latency and throughput will be lower.

This way to perform the output data is a bit different from the one of SM architecture in Fig.4.8, because now the two outputs of a single input are no more valid spaced by 100 clock cycles but outputs are valid one after the other in two clock cycles. Obviously, this is a change in the way the output data are expected and to accomplish this change the testbench should be modified. This new kind of handshake, explained better in Fig.5.5, can also be a valid alternative to the one proposed by SM.

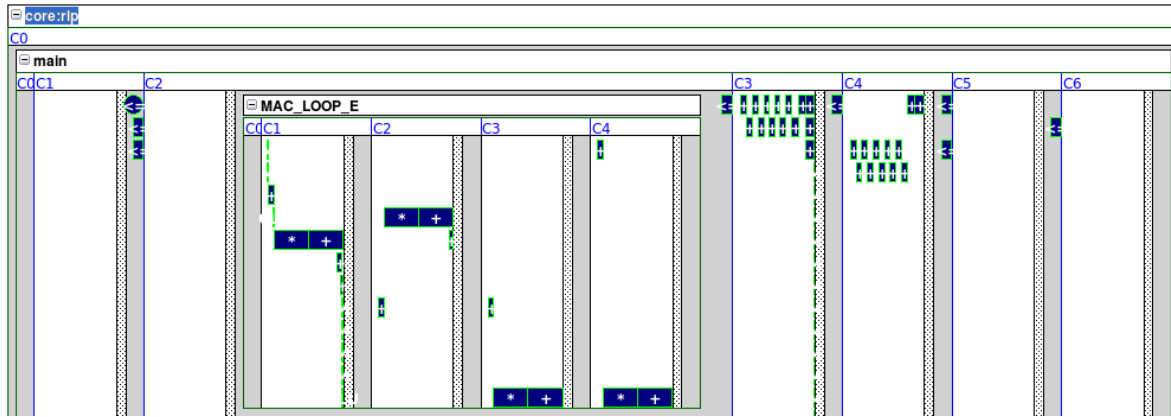


Figure 5.4: Gantt chart for the Shift full unrolled filter

This timing is only for a channel and puts in evidence that it is possible, maintaining the same input rate, to sample the output data two clock cycles in a row to have the correct ones.

For the scope of the thesis this type of architecture will be discarded to leave space to other architectures that fit perfectly within the handshake proposed by SM. Nevertheless it is possible to say

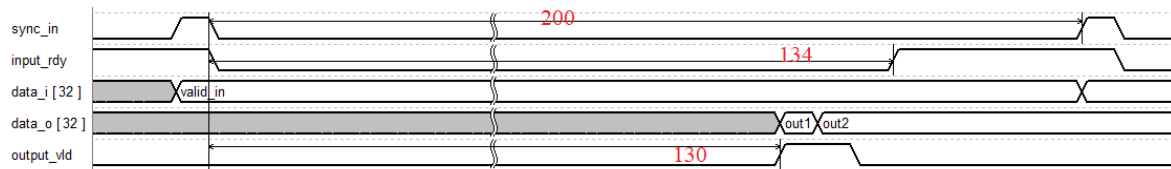


Figure 5.5: Timing of the Shift full unrolled filter

that the *Shift full unrolled* architecture is a good one in terms of area so it will be interesting to see which its performances are after the Synopsys synthesis. These are reported in Table 5.1 where

| | FIR SM | Rolled Loop | Shift full unrolled |
|--------------------------|----------|-------------|---------------------|
| Combinational area: | 23941.59 | 52978.15 | 24068.47 |
| Buf/Inv area: | 2566.10 | 5282.49 | 2113.90 |
| Noncombinational area: | 24062.36 | 59306.02 | 27314.21 |
| Total area (μm^2) | 48003.95 | 112284.18 | 51382.68 |
| Internal P | 2.556 | 7.039 | 2.858 |
| Switching P | 0.162 | 0.313 | 0.150 |
| Leakage P | 0.928 | 2.543 | 1.054 |
| Total P (mW) | 3.646 | 9.896 | 4.063 |

Table 5.1: Performances comparison

it is compared to the Silicon Mitus filter and to the Rolled one, analyzed in the previous chapter. With respect to this last one the performances are much better, the area is more than halved and the same happens for the power. Comparing it instead with the SM, architecture the performances

are comparable but slightly lower. The interface is similar and there are two resets as in the SM one, the synchronous and the asynchronous, automatically generated by Catapult. The enable signal is instead generated manually in the code.

| Operator Bitwidth Summary | | | | | |
|---|-------------|-------|------------|-------------|-------|
| Operation | Size (bits) | Count | Operation | Size (bits) | Count |
| ROM_1i5_1o26_4d055b8fedc1141da1ae4a8b418e4c4830 | 26 | 1 | nor | 1 | 12 |
| ROM_1i5_1o29_eb88306ebb957b52e2c40065b2f25cb430 | 29 | 1 | not | 5 | 1 |
| - | 60 | 4 | - | 32 | 12 |
| add | 6 | 1 | - | 1 | 21 |
| - | 5 | 1 | or | 1 | 17 |
| - | 38 | 2 | read_port | 32 | 2 |
| - | 35 | 8 | - | 1 | 1 |
| - | 34 | 2 | read_sync | 0 | 1 |
| - | 33 | 5 | reg | 60 | 8 |
| - | 32 | 8 | - | 6 | 2 |
| and | 60 | 4 | - | 5 | 2 |
| - | 5 | 2 | - | 32 | 142 |
| - | 1 | 47 | - | 29 | 1 |
| mul | 60 | 1 | - | 26 | 1 |
| - | 60 | 5 | - | 1 | 14 |
| mux | 5 | 1 | write_port | 0 | 2 |
| - | 32 | 14 | - | | |
| mux1h | 32 | 2 | | | |
| - | 29 | 1 | | | |

Figure 5.6: Overall RTL of the Rolled Loop

5.3 Pipelining of MAC Loops

Up to now only solutions that take into account the partial or total unrolling of the shift loop have been analyzed. Pipelining is another technique to modify the structure of the filter and was briefly explained in Section 1.4.1. It can be used, together with the unrolling.

First of all starting from the *Shift full unrolled* architecture, it is possible to generate with Catapult all the architectures with a growing *Initiation Interval (II)* up to grade 6. Results in Fig.5.7 are consequences of the pipelining of the MAC LOOPS.








| Solution / | Latency... | Latency... | Throug... | Throug... | Total Area | Slack |
|---|------------|------------|-----------|-----------|------------|-------|
|  fir_synth.v1 (extract) U=64 | 133 | 1731.66 | 134 | 1744.68 | 71479.75 | 9.16 |
|  fir_synth.v2 (extract) U=64, II=1 | 37 | 481.74 | 38 | 494.76 | 91227.75 | 9.16 |
|  fir_synth.v3 (extract) U=64, II=2 | 69 | 898.38 | 70 | 911.40 | 79743.97 | 9.16 |
|  fir_synth.v4 (extract) U=64, II=3 | 101 | 1315.02 | 102 | 1328.04 | 78048.01 | 9.16 |
|  fir_synth.v5 (extract) U=64, II=4 | 133 | 1731.66 | 134 | 1744.68 | 71333.30 | 9.16 |
|  fir_synth.v6 (extract) U=64, II=5 | 165 | 2148.30 | 166 | 2161.32 | 71343.95 | 9.16 |
|  fir_synth.v7 (extract) U=64, II=6 | 197 | 2564.94 | 198 | 2577.96 | 71349.19 | 9.16 |

Figure 5.7: Pipelining

Latency decreases if pipeline (II=1) is applied because both loops are involved and as consequence the latency in this case is divided by almost 4. Then if the *initiation interval* increases the latency increases but the area that is the key paramenter in this analysis is slightly decreasing up to solution with II=4. After this point, area increases. This means that those solutions are no more taken into

account. As we have seen in previous section, the complete unrolling of the SHIFT_LOOP leads to the necessity to have a latency lower than 100 clk to use the same handshake proposed by SM. With this constraint the eligible solutions are only the second and the third one, that means initiation interval lower than 3 ($II=1$ or $II=2$).

Solution v3, that has the minimum area between them, has a latency of 69 clock cycles but occupy almost $80000 \mu m^2$. This is not a good result compared to the one that will be proposed later. As a matter of fact, even if after the synthesis with Synopsys the area tends to decrease, this solution is discarded.

Starting instead from the other solution presented in Fig.5.1 that is the partial unrolling of the SHIFT_LOOP and applying pipelining as in the case presented above, the results are different and are presented below. Pipelining is applied up to grade 3 because others solutions with higher *initiation interval* exceed the limit of 200 clock cycles of throughput. This means that the inputs are not correctly sampled.





| Solution / | Latency... | Latency... | Throug... | Throug... | Total Area | Slack |
|---|------------|----------------|------------|----------------|-----------------|-------------|
|  fir_synth.v1 (extract) U=32 | 105 | 1367.10 | 106 | 1380.12 | 148393.97 | 9.16 |
|  fir_synth.v2 (extract) U=32, II=1 | 66 | 859.32 | 66 | 859.32 | 92411.64 | 9.16 |
|  fir_synth.v3 (extract) U=32, II=2 | 131 | 1705.62 | 132 | 1718.64 | 94704.69 | 9.16 |
|  fir_synth.v4 (extract) U=32, II=3 | 197 | 2564.94 | 198 | 2577.96 | 78198.57 | 9.16 |

Figure 5.8: Pipelining

This optimization leads to the best solution seen so far. Differently from the solutions presented above, these ones have pipelining applied not only to the MAC_LOOPS but also to the SHIFT_LOOP. This leads to better performances. With Catapult pipelining directly the main loop is possible and automatically all of the internal loops are pipelined of the same grade. These solutions comprehend always two different reset signals and a manual enable (because it is not generated automatically by Catapult). Moreover to achieve these results also the option "*Use old scheduling and allocation algorithm*" is enabled. This permits to use technique adopted by the versions of Catapult released before 2007. Below is reported the histogram of the area and all of its components for the four versions that has in common the partial unrolling (Fig.5.9). Using the partial unrolling ($U=32$), it is important to say that the solution has the double of the clock cycles to fit in the handshake of SM. This means that if before solutions with latency and throughput higher than 100 clock cycles were discarded, now solution up to 200 clock cycles must be accepted. In this way solutions like v4 of table in Fig.5.8 can be used.

From now on we will focus on this solution, that will be called *FIR_both_rst*, to better understand its performances.

After the Catapult generation of an RTL code in Verilog, this was passed to Synopsys together with the library that describes a technology of 45nm. The synthesis gate-level produced by Synopsys was

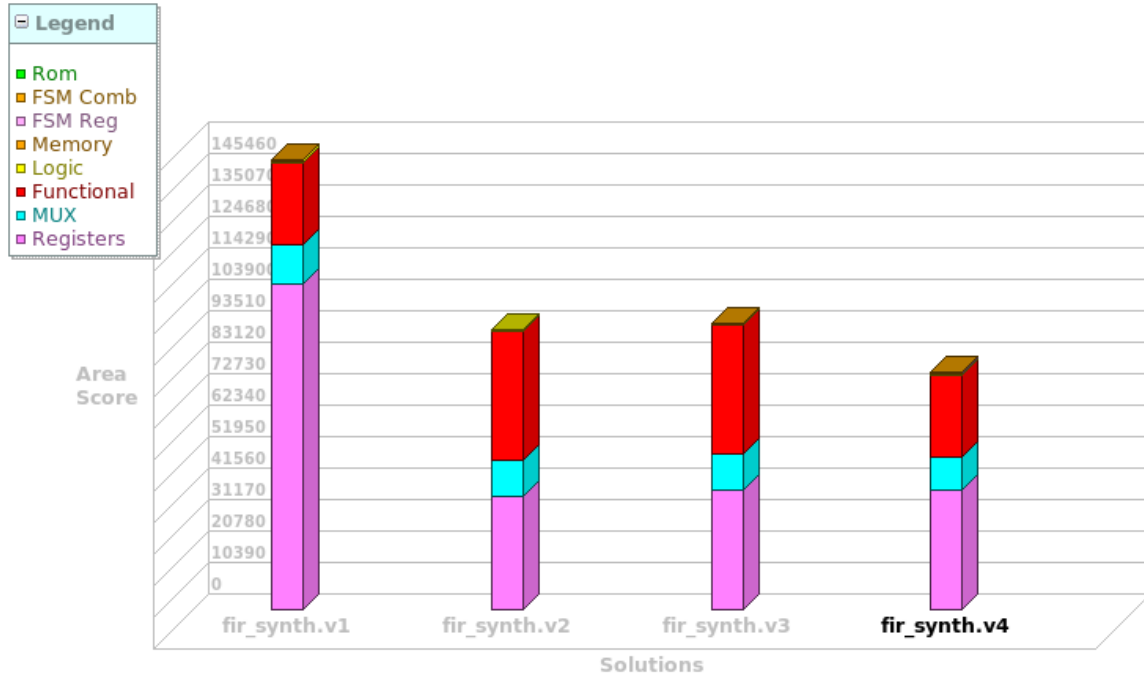


Figure 5.9: Pipelining

called *netlist* and was then passed to Modelsim to verify the correct functioning thanks to a testbench "ad hoc".

Before dealing with the verification part, the performances of this solution are shown, in a comparison with the one of SM. In the second column of Table 5.2 the performances of *FIR_both_rst* are reported. In the third column instead is reported the same architecture but the option "Use old scheduling and allocation algorithm" was disabled. The differences of performance in terms of area and power are not negligible. This option has a high impact on the generation of an efficient RTL. *FIR_both_rst* area has

| | FIR SM | FIR_both_rst | same w/o old_sched |
|--------------------------|----------|--------------|--------------------|
| Combinational area: | 23941.59 | 29452.05 | 39664.85 |
| Buf/Inv area: | 2566.10 | 2165.23 | 2884.23 |
| Noncombinational area: | 24062.36 | 21900.04 | 25665.01 |
| Total area (μm^2) | 48003.95 | 51352.09 | 65329.86 |
| Internal P | 2.556 | 2.710 | 2.695 |
| Switching P | 0.162 | 0.197 | 0.220 |
| Leakage P | 0.928 | 1.124 | 1.275 |
| Total P (mW) | 3.646 | 4.032 | 4.191 |

Table 5.2: Performances comparison

an increased of 7% with respect to the one of SM, while the power has a rise of 10%. Those numbers get worse if we look at the third column where both area and power increase for the filter without *old scheduling*. From now on *FIR_both_rst* will be the definitive solution for what concerns the filter

generated by Catapult. The verification stage will be made on this filter.

5.3.1 Testbench and verification

To verify the goodness of this solution after the gate-level synthesis, the netlist together with the library were passed to Modelsim. It is also necessary to have a testbench that must generate the input signal of the filter's interface. In Fig.5.10 are shown the differences at interface level between the SM architecture and the one of *FIR_both_rst*. One of the goals of the thesis was to generate a filter with the interface as close as possible to the reference, so that it could be replaced with a low effort in the complete system of the client. What emerges from the figure is that interfaces are quite similar but in the Catapult's one there is an extra reset signal. The synchronous one was generated internally by SM while is part of the external interface in the Catapult synthesized filter. Another difference is that SM interface has three different enables, two specific ones for the channels and a general one. In the Catapult's interface there is only the last one that has a clock gating function too. To have a complete vision also of the interface of SM, the output signals for saturation are not connected in the testbench. Now the synchronize signals must be explained.

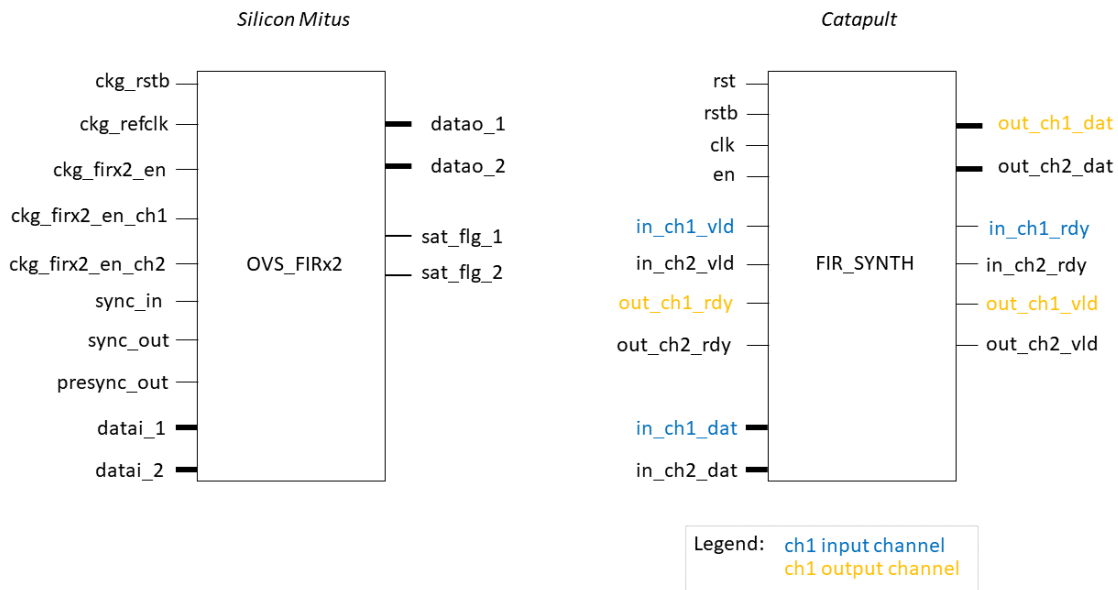


Figure 5.10: Interfaces differences

In Fig.4.8 was explained the two signals' behaviour, ideated by SM, to synchronize input and output stream. Catapult provides the possibility to specify the interface handshake as *wait*. This generates automatically three signals for every channel: valid, ready and data. For input channels (like *in_ch1* and *in_ch2*) the *valid* signal is an input of the system and the *ready* instead is generated as consequence and it is an output (look as example at the signals in blue in Fig.5.10 that belongs

to the *in_ch1*). For an output channel instead happens the opposite. As shown in Fig.5.10 with the signals in yellow, the *ready* of an output channel is an input of the filter while the signals *valid* and *data* are two outputs so they will be assigned to not-driven signal in the testbench. In the end, only the *valid* for the input channels and the *ready* for the output ones must be setted in the testbench.

```

1  fir_synth ifir_synth (
2      .clk(sys_clk),
3      .arst(rstb),
4      .rst(rst),
5      .in_ch_rsc_vld(sync_in),
6      .in_ch2_rsc_vld(sync_in),
7      .out_ch_rsc_rdy(1),
8      .out_ch2_rsc_rdy(1),
9      .in_ch_rsc_dat(datai_ch1),
10     .in_ch2_rsc_dat(datai_ch2),
11     .out_ch_rsc_dat(firx2_datao_ch1),
12     .out_ch2_rsc_dat(firx2_datao_ch2),
13     .in_ch_rsc_rdy(in_ch1_rdy),
14     .out_ch_rsc_vld(out_ch1_vld),
15     .in_ch2_rsc_rdy(in_ch2_rdy),
16     .out_ch2_rsc_vld(out_ch2_vld)
17     .enable(enable)
18 );

```

Listing 5.1: Port-map of the filter in the testbench

Input channels' *valid* is equivalent to the SM *sync_in* because it says when the input data are valid and so it is driven to '1' every 200 clock cycles, for a single clock period. For what concerns the output channels' *ready*, this is driven always to '1' to take the output data immediately when computed. The validation of the output data is subject to the *sync_in_x2* generated in the testbench every 100 clk synchronized with the *synch_in* as in Fig.4.8.

The first *sync_in* arrives after few clock cycles after the reset and the enable are both high (reset is active low). Those clock cycles are used by the architecture to initialize correctly the registers being a non-fixed number but depending on the structure of the filter. Obviously before the first *sync_in* the output must not be calculated and the counter must not start. To give the correct synchronization of all of these signals, a *valid_in* (that must not be confused with the *valid* of all the *ac_channels*) is activated only after the first *sync_in* and then stays high indefinitely. It can be generated by a SR-latch that will never be reset. This particular event happens only once after the start, as reported in Fig.5.11 In Fig.5.11 the 'adjustment' clock cycles before the first *rdy* are N=6. Then *valid_in* goes high, the first input data is sampled and from this moment the counter that synchronize input and output can start. The outputs are valid after 198 clk that is the throughput for this architecture. This event sets the *rdy* at '1' since another *sync_in* and another input data arrives. In this way the

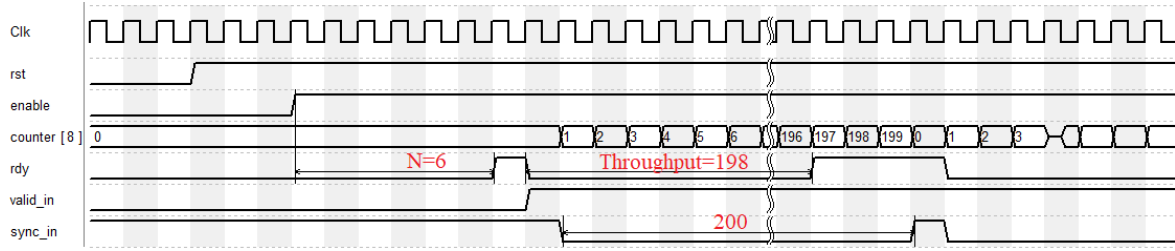


Figure 5.11: Valid_in behaviour and wait handshake

rdy stays high for 3 clk in this architecture specific case.

This behaviour is attested in the Modelsim's timing in Fig.5.12 in which both the channels of the filter are reported. The behaviour is the same of Fig.5.11. Arrows help to comprehend which the consequences of the events are.

It is interesting to report also the general functioning of the filter. For every *sync_in* there are two *sync_in_x2* that synchronize the outputs. In the timing the output are valid every 100 clk but the first output is shifted of 100 clk cycles with the respect of the point in which the input data is sampled because it takes the same period to be computed (Fig.5.13).

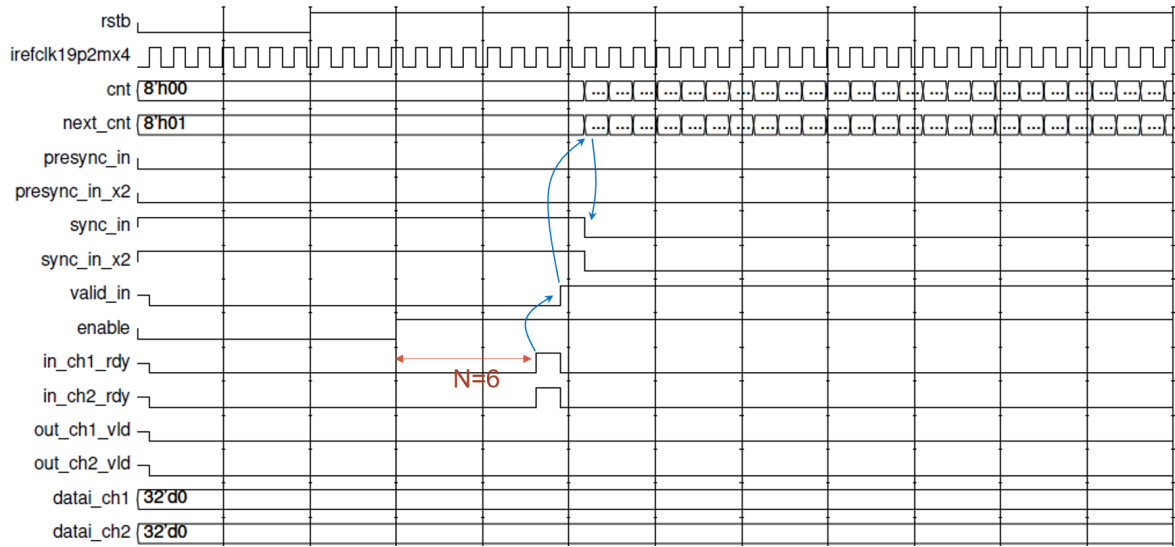


Figure 5.12: Start of the timing and valid_in behaviour

5.3.2 Back annotation and switching activity

After the simulation in Modelsim of the behaviour of the filter, another passage can be done to obtain a more accurate estimation of power with respect to the one obtained by Synopsys, reported in Table 5.2. If we take a look at the following report (Fig.5.15), that specify which the *Switching probability* of all the input nets of the filter are, it is possible to notice that most of all (like the enable and the reset signal) it has a default value equal to 0.5, that is not correct. Next to the switching probability,

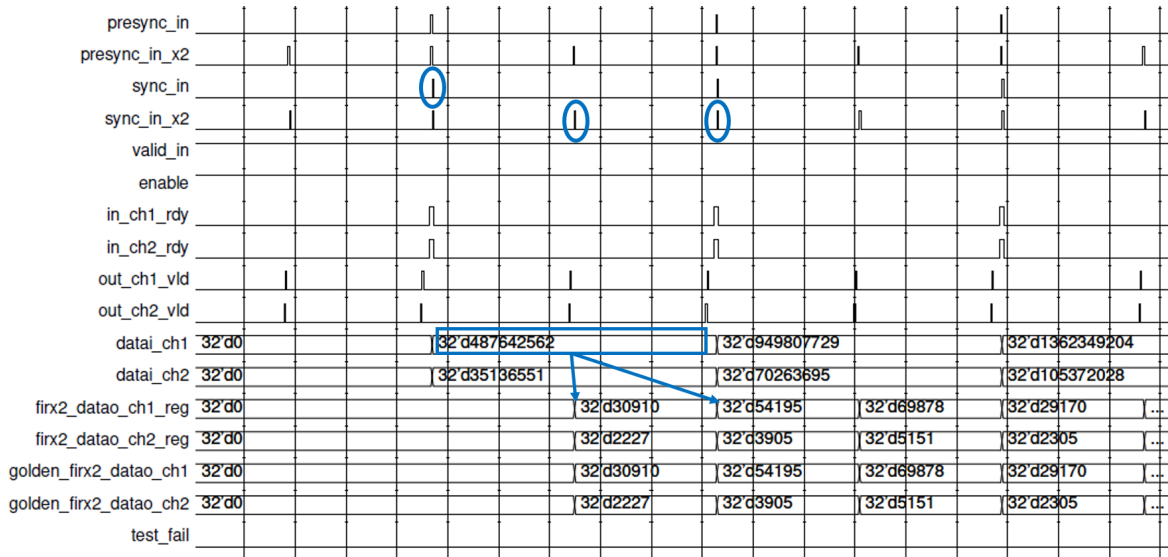


Figure 5.13: General functioning of the input/output sync

another value that is called *Toggle ratio* and represents the number of transitions of a net in a clock period (divided by the reference clock period) is reported. For example in the clock case, the Toggle rate reported is 0.1536 and it is correct because we know that in a clock period, the clock toggle 2 times so $TR = 2/13.02 = 0.1536$. The last column is the attribute of the net and it may be a letter *a*, that means that the net switching activity information was annotated by the designer (as the case of the *clk*) or a letter *d*, that means default switching activity information on net. In this case Synopsys assigns itself values at the nets, but they are not even realistic values.

To have a better estimation of the power, a designer should specify which are the probabilities and the toggle rate for every input net but this is very time consuming and not practical. The so called *Backannotation process* is used instead. The description of this technique is schematized in Fig.5.14. Starting from the RTL, generated by Catapult in our case, the synthesizer can generate the SAIF file (Switching Activity Interchange format) in which all the nets of the design associated to an "activity" are annotated. This field is completed by the simulator, Modelsim for this work, that after running the simulation, writes on the same SAIF file the switching activity for each node. In the end the SAIF with the annotations is passed back to Synopsys that, at this point, can generate a more accurate estimation for the power.

Unfortunately this annotation was available only for the design generated by Catapult and not even for the Silicon Mitus filter. For this reason the obtained results will be reported in the following but they cannot be compared to the reference power.

As reported in Table 5.3 the total power decreases in a relevant way (20%) but the Switching power increases, because now the switching activities of all the nets are no more setted to a default value that underestimate them.

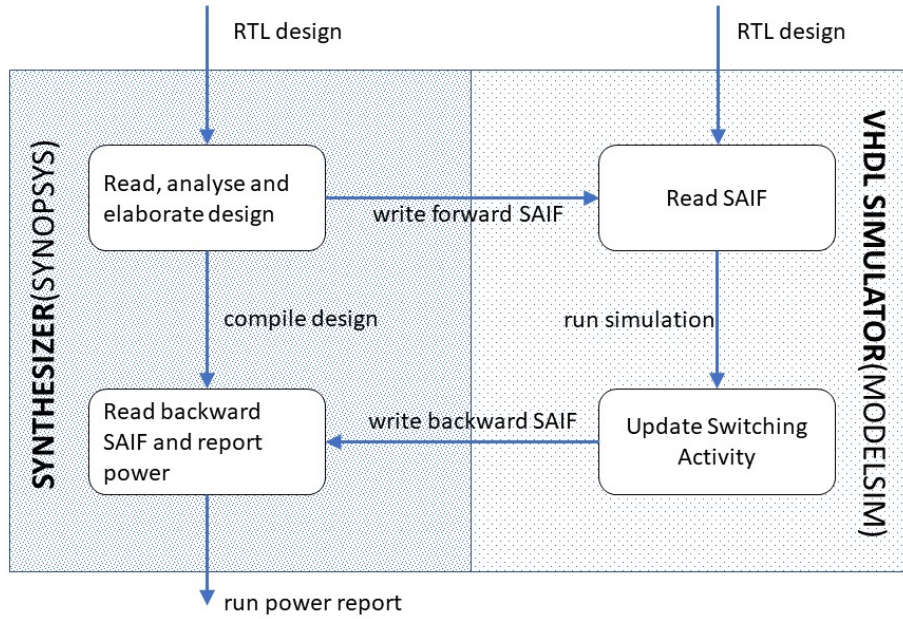


Figure 5.14: Backannotation process

| | FIR Optimized | FIR Optimized with SW_A |
|--------------|----------------------|--------------------------------|
| Internal P | 2.710 | 1.691 |
| Switching P | 0.197 | 0.413 |
| Leakage P | 1.124 | 1.107 |
| Total P (mW) | 4.032 | 3.213 |

Table 5.3: Power comparison with and without Switching activity

5.4 Solutions with low latency

It was interesting for Silicon Mitus to search also the optimized structure, taking as reference the minimum latency that is 64 clock cycles. This can be computed as the *time delay* or the *group delay* of a filter that is the interval between an input sinewave applied to the filter and the output sinewave generated by the filter. For an N-tap FIR filter, with symmetric coefficients, the time delay is computed as $(N-1)/2$. As consequence in our case will be of 64 clock cycles.

We have already seen in the previous section that in Fig.5.8 there is the second version of that filter with 66 clock cycles of latency, that is very close to the optimum one, considering a pipeline stage. This is the filter design that up to now has the lower latency and an area hypothesized by Catapult of $92200\mu m^2$.

In table 5.4 are reported other possible implementations of the filter with different optimizations that more or less are comparable with the one presented above (the first one in the table). The maximum latency considered is 69 clock cycles, that means 5 cycles over the minimum. The most interesting is the fourth because it has an area of about $76600\mu m^2$ with a latency of only 68 clk: -17% with respect

to the first alternative. This will be the better solution because it also perfectly fits in the Silicon Mitus handshake algorithm (was told about that in Section 5.3).

| | | Latency | Throughput | Area | Slack |
|------|---|---------|------------|-------|-------|
| I) | Pipeline Main (II=1) Part. Unroll Shift (U=32) with both rst, with or without old_sched | 66 | 66 | 92240 | 9.16 |
| II) | Pipeline Main (II=2) Fully Unroll Shift (U=64) with both rst, with old_sched | 65 | 64 | 91263 | 9.16 |
| III) | Pipeline Main (II=2) Fully Unroll Shift (U=64) with both rst, without old_sched | 67 | 64 | 86131 | 9.72 |
| IV) | Pipeline only MACs (II=2) Fully Unroll Shift (U=64) with both rst, with old_sched | 68 | 69 | 76629 | 9.16 |
| V) | Pipeline only MACs (II=2) Fully Unroll Shift (U=64) with both rst, without old_sched | 69 | 70 | 79758 | 9.16 |

Table 5.4: Performances comparison

The performances of this design are reported below. As it is possible to see, the area and power are not so far from the optimized structure seen in Table 5.2, but the latency in this case is 68 instead of 197. The comparison with the reference filter shows that there is an area increasing of 14% while 12% for the power. However we can consider this a good alternative implementation of the filter.

| | FIR SM | FIR_68_latency |
|--------------------------|----------|----------------|
| Combinational area: | 23941.59 | 28429.28 |
| Buf/Inv area: | 2566.10 | 2234.39 |
| Noncombinational area: | 24062.36 | 26115.88 |
| Total area (μm^2) | 48003.95 | 54545.16 |
| Internal P | 2.556 | 2.799 |
| Switching P | 0.162 | 0.202 |
| Leakage P | 0.928 | 1.095 |
| Total P (mW) | 3.646 | 4.097 |

Table 5.5: Performances comparison

| Net | Total Net Load | Static Prob. | Toggle Rate | Switching Power | Attrs |
|--------------------|-------------------|-----------------|----------------|--------------------|-------|
| clk | 1882.033 | 0.500 | 0.1536 | 174.9048 | a |
| rst | 27.924 | 0.500 | 0.0077 | 0.1298 | d |
| enable_rsc_dat | 3.983 | 0.500 | 0.0077 | 0.0185 | d |
| in_ch_rsc_rdy | 3.712 | 0.109 | 0.0049 | 0.0110 | |
| in_ch2_rsc_rdy | 3.712 | 0.113 | 0.0049 | 0.0110 | |
| in_ch2_rsc_vld | 2.010 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_vld | 2.010 | 0.500 | 0.0077 | 0.0093 | d |
| out_ch2_rsc_rdy | 2.010 | 0.500 | 0.0077 | 0.0093 | d |
| out_ch_rsc_rdy | 2.010 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[0] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[1] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[2] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[3] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[4] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[5] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[6] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[7] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[8] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[9] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[10] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[11] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[12] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[13] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[14] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[15] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[16] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[17] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[18] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[19] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[20] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[21] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[22] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[23] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[24] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[25] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[26] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[27] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[28] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[29] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[30] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch2_rsc_dat[31] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[0] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[1] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[2] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[3] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[4] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[5] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[6] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[7] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[8] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[9] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[10] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[11] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[12] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[13] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| ... | | | | | |
| in_ch_rsc_dat[29] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[30] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| in_ch_rsc_dat[31] | 1.997 | 0.500 | 0.0077 | 0.0093 | d |
| Total (139 nets) | | | | 175.7061 uW | |

Figure 5.15: Nets Switching activity and Toggle rate

CHAPTER 6

FPGA implementation

As a conclusion of this thesis work, the developed filter is implemented on an FPGA board. The chosen one is Zedboard of the Xilinx family because it is priced at a suitable level for students (even if it is used as development platform also for industry) and because Catapult HLS can generate a Verilog code that is compatible with this board.

Zed is the acronym of *Zynq Evaluation and Development* where Zynq is the name of a new kind of device which combine together two sections: the Processing System (PS) and the Programmable Logic (PL). These sections can be isolated or can be combined rising the number of possible applications. The basic architecture is the same for all the Zynq devices but it is quite complex. As an example there is the possibility to implement in the PL a 'soft' processor like the Xilinx MicroBlaze that can be used as alternative or combined with the 'hard' processor that is an ARM Cortex-A9 (part of the PS). The analysis of Zynq architecture is not the scope of this work, for our purpose is only necessary

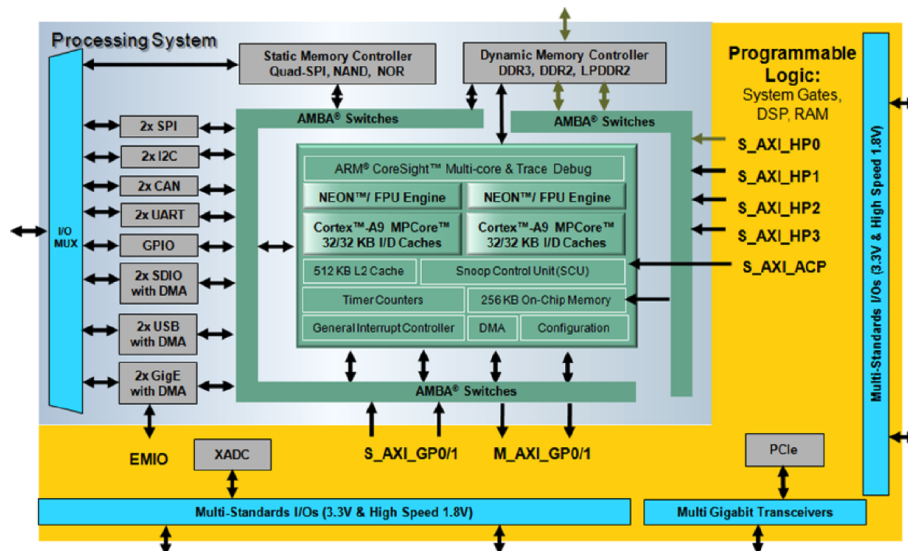


Figure 6.1: Zynq architecture[30]

to have a clear vision of the distinction of these two main parts of the Zynq. In Fig.6.1 we have a distinct separation between the PS (in grey) and the PL (in orange). We will use the Processing System to generate all the necessary signals (clk, rst and so on...) and to store the data that will be sent to the PL where the FIR filter will be implemented. The interface between these two separate sections is an AXI type (Advanced eXtensible Interface). The output data are read by the processor and then sent to the user on a UART line.

6.1 AXI bus

AXI is part of ARM AMBA (Advanced Microcontroller Bus Architecture), a family of microcontroller buses developed in 1996. At the moment the latest version of this bus (AXI4) was released in 2010 and it is used in this project. More precisely, there are three different types of AXI4 interface:

- **AXI4 Full:** This is the default interface, it is bidirectional and memory-mapped. This means that data need to be stored in a memory both in the master and the slave. As consequence, for every writing or reading operation, an address needs to be sent or received. It supports the burst communication up to 256 data words (or ‘data beats’) after sending only one address. The width of data can be parametrized up to 1024 bits. The protocol is composed by 5 signals, the ones in Fig.6.2
- **AXI4 Lite:** This is an easier version of the complete AXI4. Used for low-throughput memory-mapped communication. It allows only single data transfer.
- **AXI4 Stream:** This type is mono-directional, always Master to Slave. It is no more classified as memory-mapped because it removes the requirement of an address and the burst mode is the only possible. It is effectively an AXI4 single *Write Data channel*.

In Vivado there is the possibility to create new IP blocks with an AXI interface, totally customizable. It is possible to choose how many interfaces and which types of it are necessary. Moreover the tool allows the user to choose also how many interfaces registers must be instantiated. These registers are very important since they represent the link between the internal logic of the IP and the AXI interface that communicate with the PS.

6.2 From Catapult to Vivado

Now how to include the RLT description obtained with Catapult HLS in Vivado will be described. Using this new tool it is possible to create a specific project for the Zedboard and finally to see our filter working on a real FPGA.

First of all, in Catapult, it is necessary to change the target technology and re-synthesize the filter.

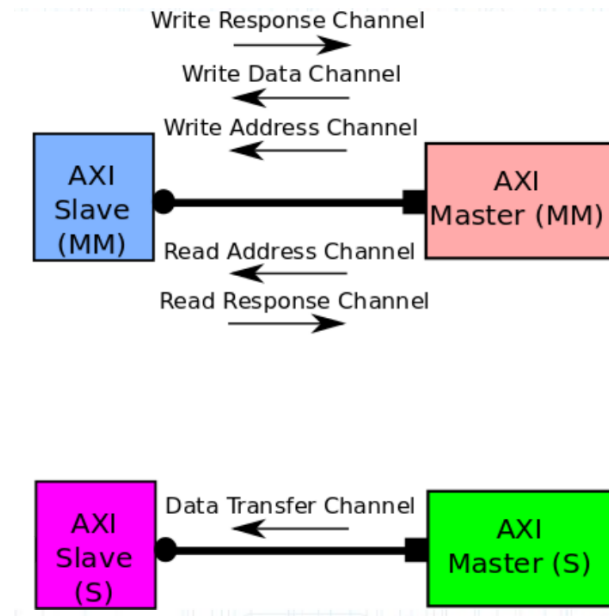


Figure 6.2: Differences between the Memory-mapped interface and the Streaming interface[31]

To select the correct one, in the Library section of the tool, choose the Xilinx family, in particular the Artix-7 and the identification code of a compatible board, as indicated in Fig.6.3. The compatibility

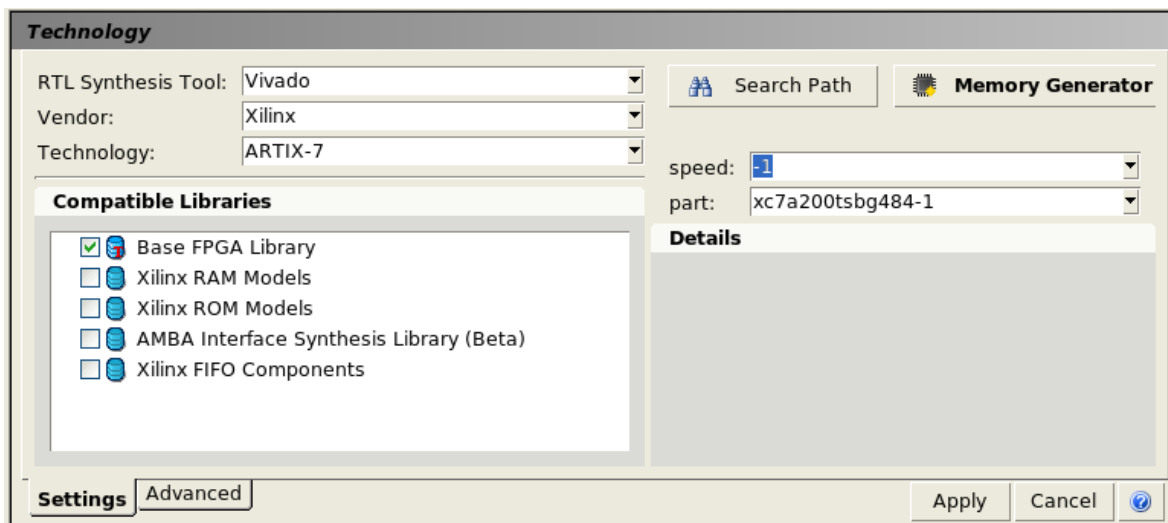


Figure 6.3: Setting for the compatible board

between the chosen identification code and the Zedboard code (XC7Z020-CLG484-1) was derived by the datasheet.

The steps to arrive at the RTL description are the same described in Paragraph 1.4.1 but the final area instantiated by Catapult is inferior to the one obtained at the end of Chapter 5, with the 45 nm technology. This result it is not unexpected because Catapult optimize the design with different basis structures that are present in the Zedboard.

After this new synthesis of the filter it's time to move on to Vivado. When creating a new project, the first thing to do is to specify the target part. Then Vivado gives the designer the possibility to insert into the project new sources in HDL or to create a new block diagram. In this diagram must be added all the components of the Zedboard (in a block form) that will be used in that project choosing from a long list that contains for example: Processor (Zynq processing system), GPIO, memories, AXI interconnects and so on. All the blocks added to the diagram must be connected together. Then after the design validation, a wrapper file must be created. In the end synthesis and the implementation will run and a bitstream will be created and loaded into the FPGA.

The processing system instead can be programmed in C language using an SDK application.

6.3 Block diagram generation

The filter is an application that usually is subject to an infinite stream of data but, for sake of simplicity, in this case the input data are stored in a memory inside the processor and are sent one by one to the filter.

The AXI interface that communicates with the filter is Memory-mapped and not Stream. In particular it is an AXI4-Lite because high performances are not requested.

Sending of the data cannot be left solely as a prerogative of the processor because a software application cannot exactly timing all the handshake signals (ready, valid and so on). For this purpose four FIFOs (two per channel) are interposed between the AXI interconnection and the filter as shown in Fig.6.4. The FIFO memories allow the processor to send data with a certain frequency, without problems, because they store all the data (they must be deep enough). Enabling the read from the FIFOs with the correct timing, will allow filter to receive and sample data, together with the correct synchronization signals. Quite the same happens to the output data, which are synchronized to go out every N clock cycles. They are stored in the output FIFOs and then the processor, when available, goes to get the data directly in the FIFO.

The steps to create the correct block design for our architecture are:

1. Filter IP creation
2. FIFO IP creation
3. Design a block diagram with Filter and four FIFO and package this design in a new IP
4. Create a new IP composed by an AXI4 Lite interface connected to the IP created in the step before
5. Insert this IP into the final block diagram with the processing system and other blocks

All of these steps are now analysed more in details.

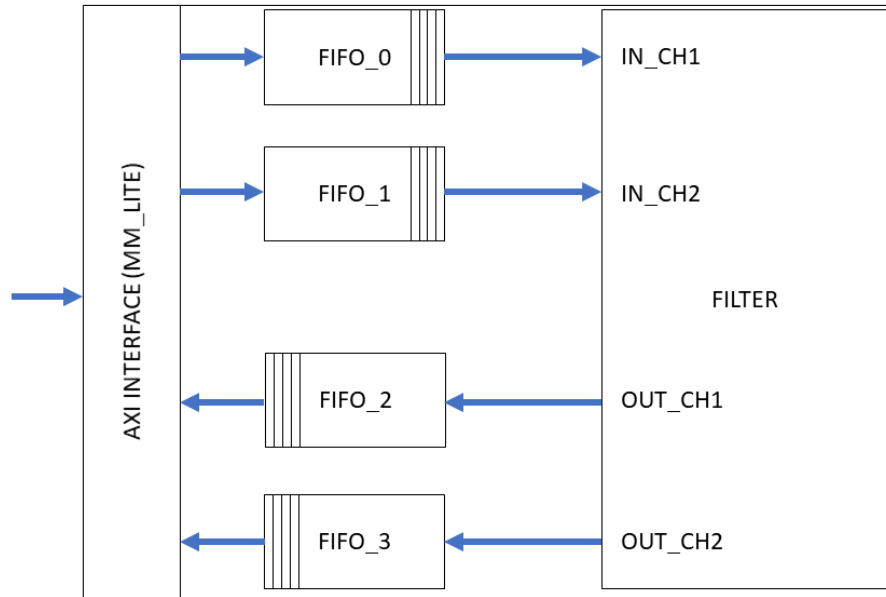


Figure 6.4: AXI Interface of the block with FIR and FIFOs

6.3.1 Filter IP creation

This step is very linear. The *concat.rtl.v* file generated by Catapult with the correct settings of the target technology is added as a source file in Vivado. To generate a new IP, Vivado opens the options window in Fig.6.5. In this case the chosen one is the first, because we don't have a block diagram and we don't want an AXI interface for this IP. After this step, the generation of a new IP is basically

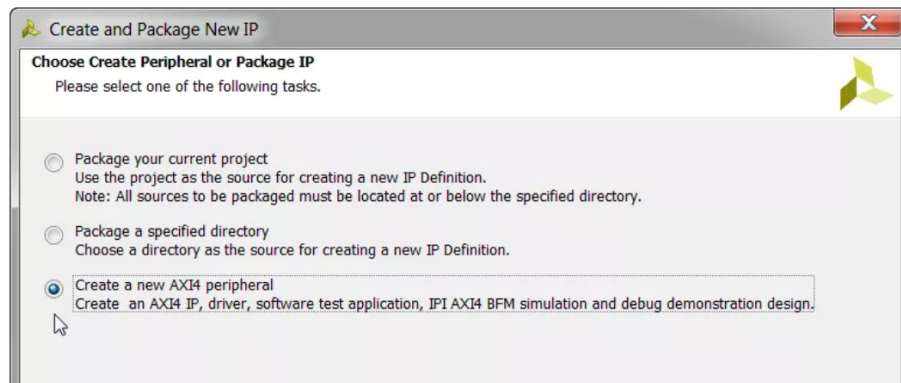


Figure 6.5: Window for creating a new IP

done. Another window is opened where changes can be made in the top level entity that will be generated for this IP or you can simply package the IP and terminate this process.

6.3.2 FIFO IP creation

The same procedure is done for the packaging of the FIFO component. This is a Fall Through FIFO because it has a combinatorial output rather than a registered output. The consequence is that timing usually results in a lower f_{max} . The code snippet of the definition of the output is reported below:

```

1 always @(posedge clk)
2     begin
3         if (fifo_we)
4             data_out2[wptr[pointers_width-2:0]] <= data_in ;
5         end
6
7 assign data_out = data_out2[rptr[pointers_width-2:0]];

```

Two pointers (*wptr* and *rptr*) are used because even if a FIFO is a sort of shift register, actually shifting data around in memory is costly to be done in hardware. A better way is to use like a circular buffer where every location has an address. Data are not moved but the shifting operation is done by manipulating the next address to write to and read from. These two addresses are saved in the cited above pointers.

When a new sample arrives, it is written in the location pointed by *wptr*, then the pointer is incremented. The same happens to *rptr* after a reading operation.

There are two critical conditions where the FIFO must be stopped: in the case it is full or empty. The memory is put in wait state until these conditions are no more verified.

```

1 assign pointer_equal = (wptr[pointers_width-2:0] - rptr[pointers_width-2:0]) ? 0:1;
2 assign fbit_comp = wptr[pointers_width-1] ^ rptr[pointers_width-1];
3
4 always @(*)
5     begin
6         fifo_full = fbit_comp & pointer_equal;
7         fifo_empty = (~fbit_comp) & pointer_equal;
8     end

```

Both the conditions of full and empty are verified when the two pointers reach the same value, without considering the MSB (condition *pointer_equal*==1), but they differ for the values of the XOR between the MSBs of the pointers. If for example we have 5 bits for the address, 4 are really used for the FIFO, while the MSB is used for the overflow. In the full condition, the read pointer reads from the lowest address while the write one has saturated the 4 LSB so the next address should have 1 as MSB:

- *rptr*="00000"
- *wptr*="10000"

In this condition *fifo_full* = 1 because the 4 LSB are equal, but the MSB is different.

Instead in the empty condition it happens that the FIFO has no data so the read pointer points the

lowest address but also the *wptr* must writes in that position. The situation is the following:

- *rp*tr = "00000"
- *wptr* = "00000"

In this case *fifo_empty* = 1 because the 4 LSB are equal and the XOR between MSBs is 0.

Another key point of the functionality of this FIFO is that the write and read operations are blocked by the full or empty conditions:

```
1 assign fifo_rd = (~ fifo_empty) & rd;
2 assign fifo_we = (~ fifo_full) & wr;
```

So *rd* and *wr* signals that arrive from the external are not the real signals that enable the read and write operations.

The full code of the FIFO is reported in Appendix B.

6.3.3 IP with Filter and FIFOs

In Vivado a new block diagram is created. It includes the Filter together with four FIFO memories. The connections of the full and empty signal are left unconnected, because it is not important to have them as external signals. The block diagram is shown in Fig.6.6. After the validation of the

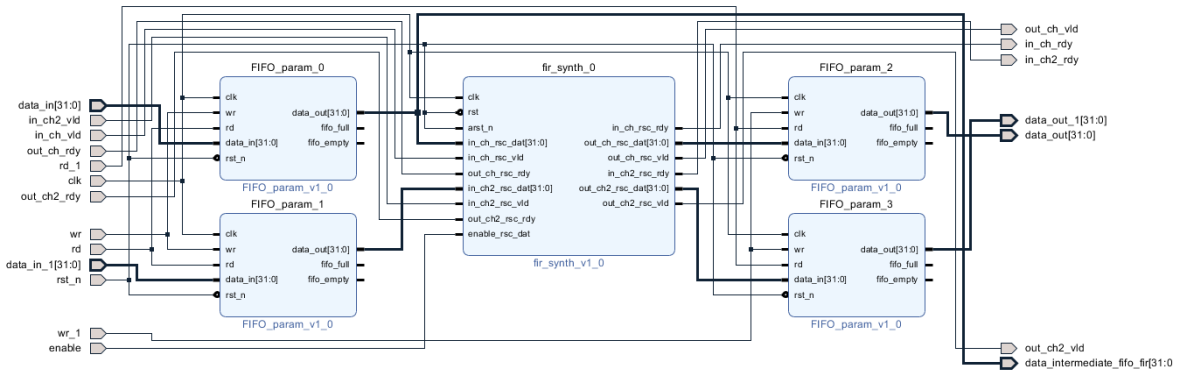


Figure 6.6: Filter with FIFOs IP

design, this new IP can be packed. A validation of the correct behaviour of this IP is done with Modelsim. Input data are sent in the same way described in Section 5.3.1. The only change is the clock frequency set to 100MHz, one of the basic frequencies of the Zedboard (the other oscillator produces a frequency of 33.3MHz). In order to send input samples with the same frequency of the original testbench (audio frequency = 384kHz) the counter module 200 is replaced by a module 261. This because $100\text{MHz}/384\text{kHz}=260.4$.

6.3.4 IP with AXI interface

At this point the IP with filter and FIFOs must be connected to an AXI interface and together generate a new IP called *AXI_FIR_FIFOs*. A scheme with the connection only for one input and one output channels is in Fig.6.9. At this point it is necessary to insert in this new IP also an FSM to generate all the correct signals for the handshake. The only synchronizer that arrives from the external is a signal called *Start*. The state machine is a replication of what happens in the original testbench, where a counter and other signals (ready of input channel, valid_in, enable) synchronize the reading of input data and the writing of the output ones. Now the processor is managed by a software application written in C language that makes impossible the correct sending of these triggers signals to the filter together with the input data.

It was chosen to write all the data into the input FIFOs (FIFO_0 and FIFO_1), then write into an interface register the value '1' for enabling the Start. Only after the Start is high, the filter requests the first data and all the others in sequence. The output results were saved into the output FIFOs

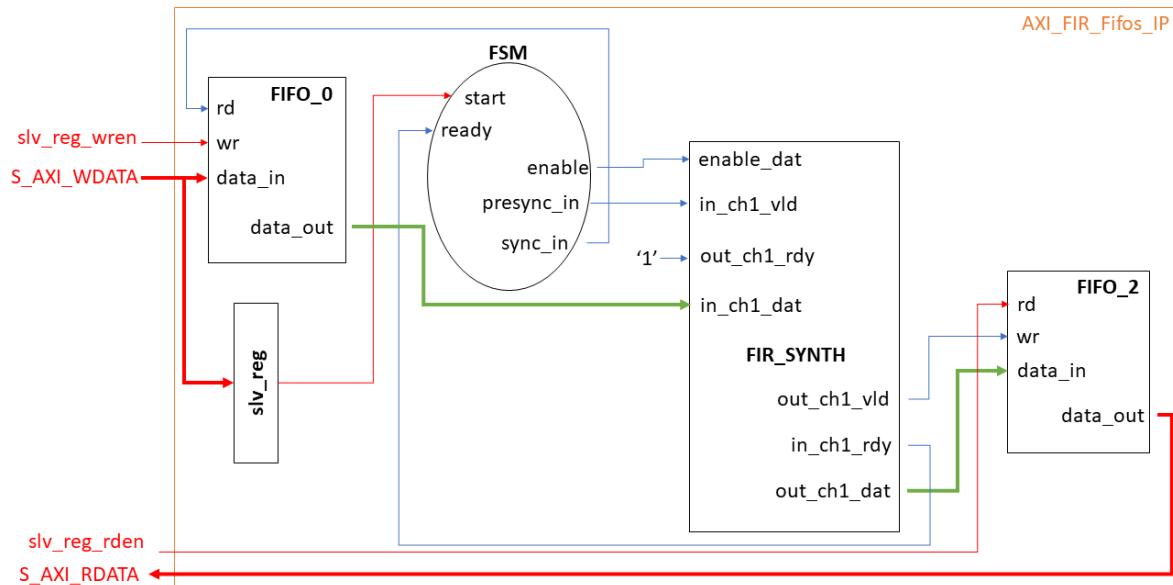


Figure 6.7: Interconnections between the AXI interface and the Filter+FIFOs IP

(FIFO_2 and FIFO_3) and, after a certain time, the AXI interface can start to read data and send them back to the processor.

The FSM description is reported below. It starts with the rising of the enable signal only a clock cycle after the Start. Then the signals *Valid_in*, *rd_en_fifos_in* and *wr_en_fifos_out* are set. The counter starts only one clock cycle after the *Valid_in* goes high.

```

1 always@(posedge clk or negedge rst_n_1)
2   if(~rst_n_1) begin

```

```

3     enable_1 = 0;
4 end else if(start == 1) begin
5     enable_1 = 1;
6 end
7 else begin
8     enable_1 = 0;
9 end
10
11 always@(posedge clk)
12     if(~rst_n_1 | ~enable_1) begin
13         valid_in <= 1'b0;
14     end
15     else if(in_ch1_rdy == 1) begin
16         valid_in <= 1'b1;
17     end
18
19 always@(posedge clk or negedge rst_n_1)
20     if(~rst_n_1)begin
21         cnt <= 0;
22         rd_en_fifos_in = 0;
23         wr_en_fifos_out = 0;
24     end
25     else if (valid_in == 1)begin
26         cnt <= next_cnt;
27         rd_en_fifos_in = sync_in;
28         wr_en_fifos_out = out_ch1_vld;
29     end

```

Below is shown the design_1 (FIR_FIFOs_IP) portmap into the AXI interface. As in the Fig.6.9 the data_in signal arrives to the FIFO directly from the AXI interface through the bus *S_AXI_WDATA*. This bus, as the read one, communicates with a set of interface registers, called *slv_reg*. To cut out these registers from the communication between the processor and the FIFOs some tricks are used. For what concerns the writing operation, the *S_AXI_WDATA* arriving from the external is connected both to all the slave registers and to the input of the FIFO. When a write function is called from the processor, it specifies also an address. If the slave register in which the data should be written is the *slv_reg0* (writing address is 0x43c00000) it is written instead in the data_in of the input FIFO. In other words, the data is intercepted and redirected.

This condition is expressed by the *write_data_en* enable signal. If the writing address is not zero, the write operation is considered traditional and the incoming data is written on the correspondent register, as in the case of the Start condition that is written on the *slv_reg2* (writing address is 0x43c00004).

```

1 wire write_data_en;
2 assign write_data_en = slv_reg_wren & (axi_awaddr[ADDR_LSB+

```

```

3      OPT_MEM_ADDR.BITS:ADDR.LSB]==4'h0);
4
5  design_1 design_1_i
6      (.clk(S_AXI_ACLK),
7       .rst_n(S_AXI_ARESETN),
8       .start(slv_reg2),
9       .data_in(S_AXI_WDATA),
10      .data_out(data_out),
11      .rd_1(slv_reg_rden),
12      .wr(write_data_en));

```

For what concerns the output data, it is associated in the portmap to a wire called `data_out`. In the code of the slave AXI interface (not shown) this wire is associated to the last interface register. If a reading operation is requested from that register, `data_out` is put on the output bus (`S_AXI_RDATA`), as in Fig.6.9.

Before moving to the last step, that will include this IP in a block diagram with other blocks of the Zedboard, this AXI IP was packaged and tested on Modelsim. The testbench now is a little bit different because it must simulate which commands will arrive from the Processing System. Moreover the FSM for the synchronization was "moved" into the IP so the Testbench must only generate the Start signal. The entire testbench code is reported in Appendix C.

The results of simulation are correct, input data are taken from a file and are stored in the input FIFOs after the reset signal goes high. The start arrives after 600ns immediately followed by all the synchronization signals. But we have to wait the first *sync_in* signal before the first data is taken from the FIFO and passed to the filter. It is important to remember that *sync_in* is assigned to the read enable of the first FIFO. In the timing in Fig.6.8 there are some key passages that are underlined. The first one (blue circle) is the input data writing in the first FIFO, after the reset signal goes high. An undetermined time flows until the Start arrives (in this testbench is chosen by the designer, but not in the real world), the FSM produces the synchronization signals, and after the *valid_in* goes to '1' and the first *in_ch1_rdy* arrives, the first input sample (0) is sent to the filter, then the second sample arrives after 261 clock cycles. The filter can compute all of its results that are stored in the output FIFO waiting to be read. After another undetermined time (that in this demonstration is $50\mu s$), the read enable of the output FIFOs goes high. The FIFO starts sending out the data but obviously not all have been calculated yet. When all the calculated results have already been sent out the FIFO will be empty. It will have to wait for a new input data to enter the filter and calculate the two outputs, so that the output FIFO has some data in it again.

This is what happens in the orange circle in the testbench. There is a period, immediately after the read enable, where stored data are sent out, one after the other, every clock cycle, then the output bus (`datao_ch1` and `datao_ch2`) will receive new output data with a certain periodicity.

After checking the system behaviour is correct, it is the time for the last step which will lead us

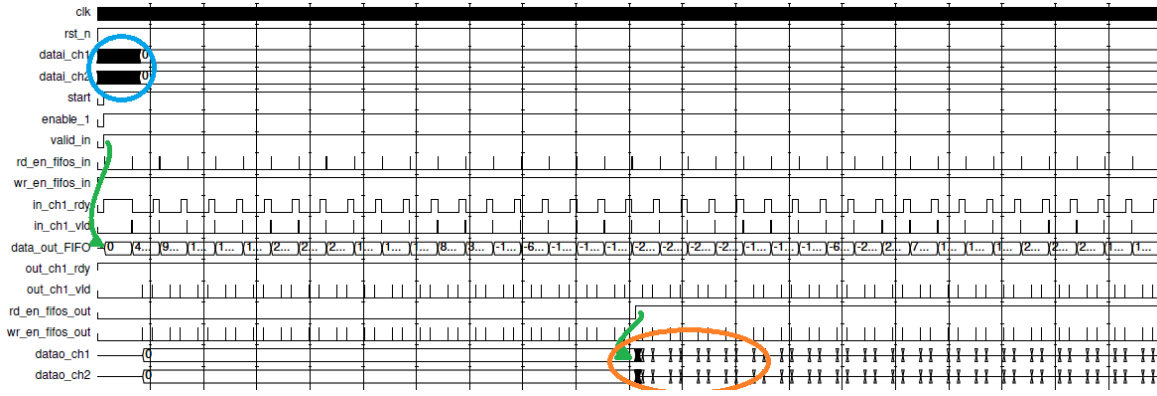


Figure 6.8: Timing for the verification of the AXI_FIR_FIFOs IP

directly to the FPGA.

6.3.5 Final block diagram

Now a new project is created in Vivado and the generated IP (AXI_FIR_Fifos) is added. There are also other blocks that must be added as the Processing system, the Reset generator and an AXI interconnect block. These last two are added automatically by Vivado once we try to connect our IP to the Zynq PS. There are no critical points during this step. The only thing that must be verified, is that the frequency generated in the PS is effectively 100MHz and the block *Processor System Reset* generates a synchronous reset. The block diagram is verified, then synthesized and implemented.

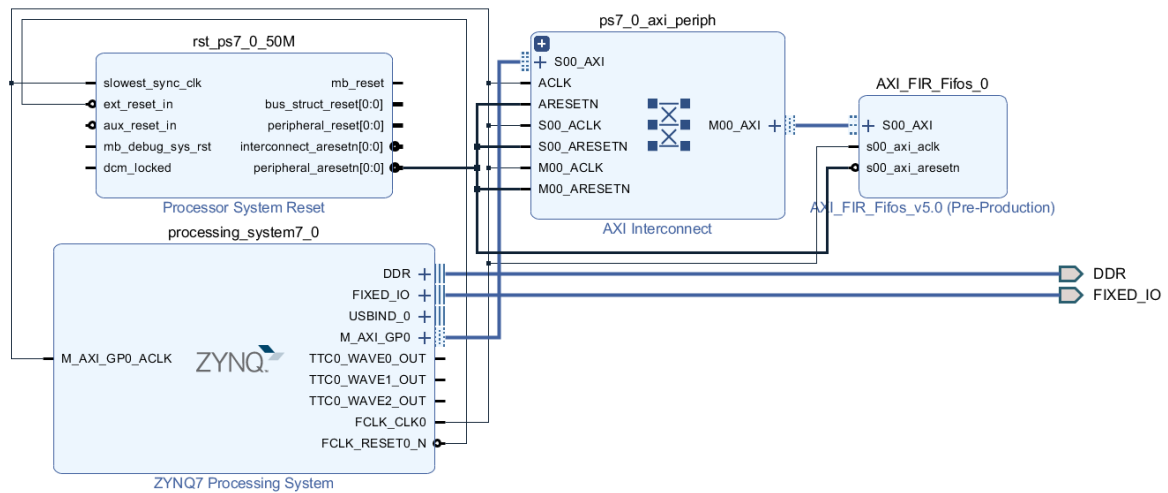


Figure 6.9: Interconnections between the Zynq PS and the AXI IP through other blocks

Finally a bitstream is generated and loaded on the FPGA. The C code that manage the processor activities, is partially presented in the Fig.6.10 but is also inserted in Appendix D. The operations done are summarized here:

1. Write the start value equal to 0
2. For cycle to write inputs, on the entry FIFO
3. Write the start value equal to 1
4. Software delay to be sure that the filter has at least computed one result
5. For cycle to read outputs, from the exit FIFO
6. Write the read data on a UART line

The results correctness is testified by Fig.6.10 where, in the Terminal on the right, are read all the values transmitted by the UART line.

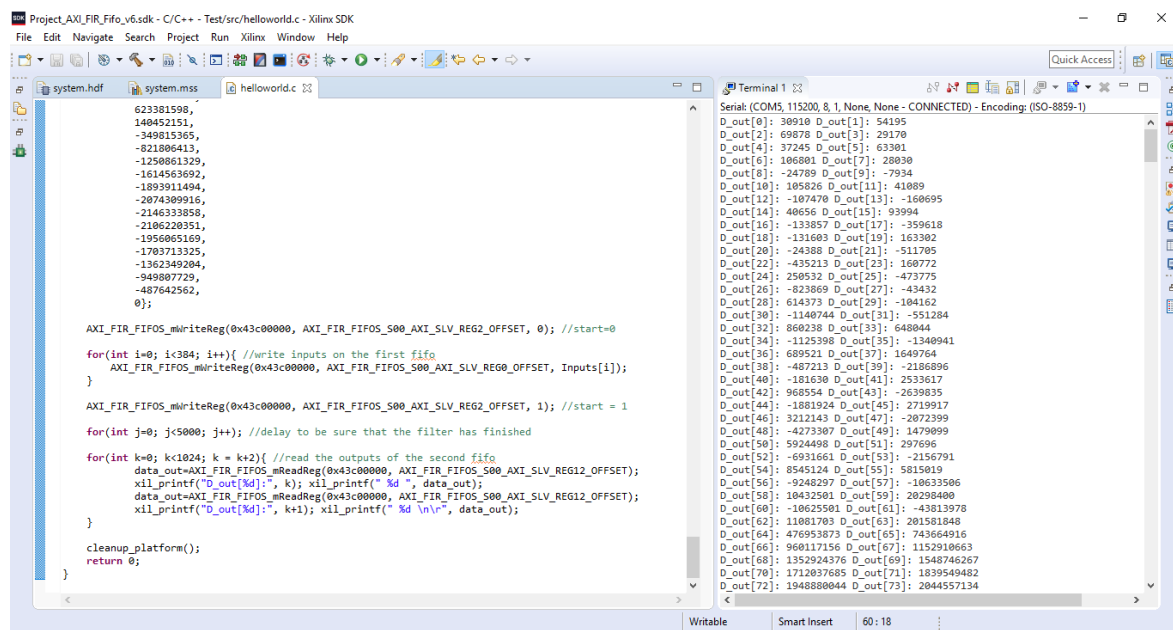


Figure 6.10: Screen of Vivado SDK with the UART Terminal

CHAPTER 7

Conclusions

This work was born with the willingness to further investigate the possibilities of an excellent design method such as High Level Synthesis. This method lays its foundation during the 80's years but only with the new millennium it reaches a maturity that allows it to be used in a business environment. It is still not optimized for specific applications as outlined in Chapter 2 of this work. But the results obtained in the design of circuits, in particular for FPGA application, are relevant.

This work aims to test the quality of results of a Finite Input Response filter's design with HLS. Starting from a C++ model of the filter an RTL description was automatically produced by Catapult HLS. The work made by the designer is to guide the tool through an elevate number of possible optimizations, to the best architecture.

The goodness of the solution was tested in terms of area and power, having as reference the RTL description of the same filter made by the company we have collaborated with.

In the end the filter is tested also on a Xilinx Zedboard, to verify it behaviour with real applications. The final statement is that it is possible to obtain, with HLS and Catapult, quality of results comparable to manually written RTL using traditional design (HDL).

Bibliography

- [1] Robert A. Walker and Donald E. Thomas, Electrical and Computer Engineering Dept. Carnegie-Mellon University Pittsburgh, *A Model of Design Representation and Synthesis*, IEEE 1985
- [2] Grant Martin, Gary Smith *High-Level Synthesis: Past, Present, and Future*, IEEE 2009
- [3] Michael C. McFarland, member IEEE, Alice C. Parker, senior member IEEE, Raul Camposano *The High-Level Synthesis of Digital Systems*, IEEE 1990
- [4] A. Parker, D. Thomas, D. Sicwiorek, M. Barbacci, L. Hafer, G. Lcive, J. Kim Carnegie-Mellon University Departments of Electrical Engineering and Computer Science Pittsburgh *The CMU Design Automation System, An Example of Automated Data Path Design*, Proc. Design Automation Conf. (DAC 79), ACM Press, pp. 73-80.
- [5] T. Ogunfunmi, S. Desai *Fast FIR Filter Implementation using High Level Synthesis Tools*, 37th Midwest Symposium on Circuits and Systems, August 3-5, 1994
- [6] Liu Hanbo, Wang Shaojun, Zhang Yigang *Design of FIR filter with high level synthesis*, IEEE 2015
- [7] Taheni Damak, LellaAycha Ayadi, Nouri Masmoudi, Laboratory of Electronics and Information Technology-LETI, University of Sfax, Sébastien Bilavarn, Laboratory of Electronics Antennas and Telecommunications, University of Nice-Sophia Antipolis Nice *HLS and manual Design methodology for H.264/AVC Deblocking Filter*, IEEE 2015
- [8] Rajesh Gupta, FForrest Brewer *High-Level Synthesis: a retrospective*, January 2008
- [9] Shawn McCloud, Calypto Design Systems *High-Level Synthesis Report 2011*
- [10] Mentor Graphics *RTL Power Reduction & High-Level Synthesis Report 2013*
- [11] Mentor Graphics *High-Level Synthesis Report 2014*
- [12] Laurent Hili *High Level Synthesis techniques*, ESA-ESTEC August 2011
- [13] Louise H. Crockett, Ross A. Elliot, Martin A. Enderwitz, Robert W. Stewart, Department of Electronic and Electrical Engineering University of Strathclyde, Glasgow, Scotland, *The Zynq Book*, Strathclyde Academic Media, July 2014
- [14] S McCloud, *Catapult-C, synthesis-based design flow: speeding implementation and increasing flexibility*, White Paper, Mentor Graphics, 2004
- [15] Mentor Graphics *Catapult® High-Level Synthesis Datasheet*
- [16] Calypto *Catapult C Synthesis Datasheet*,

-
- [17] Frans Sijstermans and Jc Li, NVIDIA company *Working smarter not harder: NVIDIA closes design complexity gap with High Level Synthesis*
 - [18] Mentor Graphics *Algorithmic C (AC) Datatypes*, Software Version v3.9, September 2018
 - [19] Mentor Graphics *Catapult® Online Training* Software Version v10.2 Beta, January 2018
 - [20] Tero Joentakanen, Tampere University of Technology *Evaluation of HLS modules for ASIC Back-end*, Master of Science thesis, 2016
 - [21] P. Ollikainen, University of Oulu, Department of Electrical Engineering *SoC Subsystem Design Using SystemC based High-Level Synthesis*, Master's thesis, 2016, 48 p.
 - [22] J. Järviluoma, University of Oulu, Department of Electrical Engineering *Rapid Prototyping from Algorithm to FPGA Prototype*, Master's thesis, 2015, 59 p.
 - [23] Q. Zhu and M. Tatsuoka, *High Quality IP Design using High-Level Synthesis Design Flow*, 21st Asia and South Pacific Design Automation Conference, 2016, pp. 212-217.
 - [24] Z. Sun, K. Campbell, W. Zuo, K. Rupnow, S. Gurumani, F. Doucet, and D. Chen, *Designing high-quality hardware on a development effort budget: A study of the current state of high-level synthesis* 21st Asia and South Pacific Design Automation Conference (ASP-DAC), 2016, pp. 218-225.
 - [25] E. Torppa, University of Oulu, Department of Electrical Engineering *High-Level Synthesis in IP based SoC Development*, Master's thesis, 2015, 69 p.
 - [26] I. Kivimäki, University of Oulu, Department of Electrical Engineering *High-Level Synthesis Design Flow in FPGA Design*, Master's thesis, 2016, 60 p.
 - [27] M. D. Zwagerman, Grand Valley State University *High Level Synthesis, a Use Case Comparison with Hardware Description Language*, Master's thesis, 36 p.
 - [28] K. Karras, M. Blott, and K. A. Vissers, *High-Level Synthesis Case Study: Implementation of a Memcached Server*, 1st International Workshop on FPGAs for Software Programmers, 2014, pp. 77-82.
 - [29] Xilinx *Zynq-7000 All Programmable SoC Family Product Tables and Product Selection Guide*
 - [30] Andreas Habegger, Bern University of Applied Sciences *Introduction Zynq*
 - [31] Mohammadsadegh Sadri, Post Doctoral Researcher, TU Kaiserslautern *AXI Stream Interface*, April 2014

APPENDIX A

C++ codes

Listing A.1: fir_synth.h

```
#ifndef FIR_SYNTH_H_
#define FIR_SYNTH_H_

#include <ac_fixed.h>

const unsigned ORDER = 129;
const unsigned NBC = 30;

typedef ac_fixed<32,32,true>   DATA_TYPE;
typedef ac_fixed<30,30,true>   COEFF_TYPE;
typedef DATA_TYPE::rt_T<COEFF_TYPE>::mult   PROD_TYPE;
typedef PROD_TYPE::rt_unary::set<ORDER>::sum   SUM_TYPE;
typedef ac_fixed<33,33,true>   INTER_TYPE;
typedef ac_fixed<32,32,true>   OUT_TYPE;

void fir_synth(const DATA_TYPE, OUT_TYPE &);

#endif
```

Listing A.2: fir_synth.cpp

```
#include <iostream>
#include <cmath>
#include <vector>

#include "fir_synth.h"

//#pragma hls_design top
```

```

using namespace std;

void fir_synth(const DATA_TYPE i_sample, OUT_TYPE &y, const DATA_TYPE i_sample2,
               OUT_TYPE &y2)
{
    COEFF_TYPE coefficients[ORDER] = {17015,
    29833,
    5325,
    -42050,
    -37405,
    33403,
    ...
    29833,
    17015};

    static DATA_TYPE samples[ORDER], samples2[ORDER];

    SHIFT_LOOP: for (int n=ORDER-1; n>0; n--) {
        samples[n]=samples[n-1];
        samples2[n]=samples2[n-1];
    }
    samples[0]=i_sample;
    samples2[0]=i_sample2;

    SUM_TYPE sum = 0, sum2=0;

    MAC_LOOP: for (unsigned n=0; n<ORDER; ++n) {
        sum += samples[n] * coefficients[n];
        sum2 += samples2[n] * coefficients[n];
    }

    INTER_TYPE temp_out, temp_out2;

    temp_out=sum >> 27;
    temp_out=temp_out+1;
    y=temp_out >> 1;

    temp_out2=sum2 >> 27;
    temp_out2=temp_out2+1;
    y2=temp_out2 >> 1;

}

```

Listing A.3: main.cpp

```

// Include the design function to be tested
#include "fir_synth.h"

// Include utility headers
#include <iostream>
#include "csvparser.h"
#include <vector>
#include <assert.h>
#include <string>
#include <sstream>
#include <fstream>
#include <iomanip>
using namespace std;

typedef vector<DATA_TYPE> samplesVect_type;

// Forward Declarations of utility functions

int ReadCSV_Samples(string filename, samplesVect_type &samples);
bool WriteCSV_Samples(string, samplesVect_type &samples);

int main()
{
    /// define data structure for holding input and output samples:
    samplesVect_type samples;
    samplesVect_type samples_out;

    // read in_samples from CSV file
    if (ReadCSV_Samples("samples.csv", samples) < 0)
    {
        cerr << __FILE__ << ":" << __LINE__ << "_Failed_to_read_input_samples" << endl;
        return -1;
    }

    // Loop through samples, applying them to the filter
    for (vector<DATA_TYPE>::iterator it = samples.begin(); it != samples.end(); ++it)
    {
        DATA_TYPE stimulus_element = *it;
        OUT_TYPE exit_element;
        fir_synth(stimulus_element, exit_element);
        samples_out.push_back(exit_element);
    }
}

```

```

cout << "Size_of_samples_out:" << samples_out.size() << endl;
WriteCSV_Samples("filter_output.csv", samples_out);

cout << __FILE__ << ":" << __LINE__ << " _End_of_testbench." << endl;
return 0;
}

int ReadCSV_Samples(string filename, samplesVect_type &samples)
{
CsvParser *csvparser = CsvParser_new(filename.c_str(), ",", 0); //1: first line is
                                                                    a header; 0: not

CsvRow *row;

while ((row = CsvParser_getRow(csvparser)) )
{
const char **rowFields = CsvParser_getFields(row);
double double_stimulus_element;
stringstream(*rowFields) >> double_stimulus_element;
//cout<<"Double_input:"<<double_stimulus_element<<endl;
ac_fixed<DATA_TYPE::width, DATA_TYPE::i-width, DATA_TYPE::sign,AC.RND,AC.SAT.SYM>
fixed_stimulus_element = double_stimulus_element;
//cout<<"Fixed:"<<fixed_stimulus_element<<endl;
samples.push_back(fixed_stimulus_element);
CsvParser_destroy_row(row);
}
cout << __FILE__ << ":" << __LINE__ << " _CSV_file_" << filename << " _" <<
samples.size() << " _samples_were_read_in." << endl;
CsvParser_destroy(csvparser);
return samples.size();
}

bool WriteCSV_Samples(string oFileName, samplesVect_type &samples)
{
ofstream oSampleFile;
cout << __FILE__ << ":" << __LINE__ << " _Writing_output_csv_file_to_" <<
oFileName << " _" << endl;
oSampleFile.open(oFileName.c_str());

if (!oSampleFile.is_open())
{
cerr << __FILE__ << ":" << __LINE__ << " _CSV_output_file_" << oFileName <<
"' _could_not_be_created." << endl;
return false;
}
}

```

```
for (samplesVect_type::iterator it = samples.begin(); it != samples.end(); ++it)
{
oSampleFile << fixed << setprecision(0) << (*it) << endl;
}
oSampleFile.close();
return true;
}
```

APPENDIX B

VHDL codes

Listing B.1: FIFO.v

```
module FIFO #(
    parameter data_width = 32,
    parameter fifo_depth = 1024,
    parameter pointers_width = 11
)
(data_out, fifo_full, fifo_empty, clk, wr, rd, data_in, rst_n) ;

input wr, rd, clk, rst_n;
input[data_width-1:0] data_in;
output[data_width-1:0] data_out;
output fifo_full, fifo_empty;

wire fifo_threshold, fifo_overflow, fifo_underflow;
wire[pointers_width-1:0] wptr, rptr;
wire fifo_we, fifo_rd;

write_pointer top1(wptr, fifo_we, wr, fifo_full, clk, rst_n);
read_pointer top2(rptr, fifo_rd, rd, fifo_empty, clk, rst_n);
memory_array top3(data_out, data_in, clk, fifo_we, wptr, rptr);
status_signal top4(fifo_full, fifo_empty, fifo_threshold, fifo_overflow,
    fifo_underflow, wr, rd, fifo_we, fifo_rd, wptr, rptr, clk, rst_n);
endmodule

module memory_array #(
    parameter data_width = 32,
    parameter fifo_depth = 1024,
    parameter pointers_width = 11
```

```

)
(data_out , data_in , clk , fifo_we , wptr , rptr );

input [data_width-1:0] data_in;
input clk , fifo_we;
input [pointers_width-1:0] wptr , rptr;
output [data_width-1:0] data_out;

reg [data_width-1:0] data_out2 [fifo_depth-1:0];
wire [data_width-1:0] data_out;

always @(posedge clk)
    begin
        if (fifo_we)
            data_out2[wptr[pointers_width-2:0]] <= data_in ;
        end
    assign data_out = data_out2[rptr[pointers_width-2:0]];
endmodule

module read_pointer #(
    parameter data_width = 32,
    parameter fifo_depth = 1024,
    parameter pointers_width = 11
)
(rptr , fifo_rd , rd , fifo_empty , clk , rst_n);

input rd , fifo_empty , clk , rst_n;
output [pointers_width-1:0] rptr;
output fifo_rd;

reg [pointers_width-1:0] rptr;
assign fifo_rd = (~fifo_empty)& rd;

always @(posedge clk or negedge rst_n)
    begin
        if (~rst_n) rptr <= 11'b00000000000;
        else if (fifo_rd)
            rptr <= rptr + 11'b00000000001;
        else
            rptr <= rptr;
        end
    endmodule

module status_signal #(
    parameter data_width = 32,
    parameter fifo_depth = 1024,

```

```

        parameter pointers_width = 11
    )
    (fifo_full , fifo_empty , fifo_threshold , fifo_overflow , fifo_underflow , wr , rd ,
    fifo_we , fifo_rd , wptr , rptr , clk , rst_n);

    input wr , rd , fifo_we , fifo_rd , clk , rst_n;
    input [pointers_width-1:0] wptr , rptr;
    output fifo_full , fifo_empty , fifo_threshold , fifo_overflow , fifo_underflow;

    wire fbit_comp , overflow_set , underflow_set;
    wire pointer_equal;
    wire [pointers_width-1:0] pointer_result;
    reg fifo_full , fifo_empty , fifo_threshold , fifo_overflow , fifo_underflow;

    assign fbit_comp = wptr[pointers_width-1] ^ rptr[pointers_width-1];
    assign pointer_equal = (wptr[pointers_width-2:0] - rptr[pointers_width-2:0]) ? 0:1;
    assign pointer_result = wptr[pointers_width-1:0] - rptr[pointers_width-1:0];
    assign overflow_set = fifo_full & wr;
    assign underflow_set = fifo_empty & rd;

    always @(*)
        begin
            fifo_full = fbit_comp & pointer_equal;
            fifo_empty = (~fbit_comp) & pointer_equal;
            fifo_threshold = (pointer_result[pointers_width-1]
                            || pointer_result[pointers_width-2]) ? 1:0;
        end
    always @(posedge clk or negedge rst_n)
        begin
            if(~rst_n) fifo_overflow <=0;
            else if((overflow_set==1)&&(fifo_rd==0))
                fifo_overflow <=1;
            else if(fifo_rd)
                fifo_overflow <=0;
            else
                fifo_overflow <= fifo_overflow;
        end
    always @(posedge clk or negedge rst_n)
        begin
            if(~rst_n) fifo_underflow <=0;
            else if((underflow_set==1)&&(fifo_we==0))
                fifo_underflow <=1;
            else if(fifo_we)
                fifo_underflow <=0;
            else
                fifo_underflow <= fifo_underflow;
        end

```

```
        end
endmodule

module write_pointer #(
    parameter data_width = 32,
    parameter fifo_depth = 1024,
    parameter pointers_width = 11
)
(wptr, fifo_we, wr, fifo_full, clk, rst_n);

input wr, fifo_full, clk, rst_n;
output [pointers_width-1:0] wptr;
output fifo_we;

reg [pointers_width-1:0] wptr;
assign fifo_we = (~fifo_full)&wr;

always @(posedge clk or negedge rst_n)
    begin
        if(~rst_n) wptr <= 11'b00000000000;
        else if(fifo_we)
            wptr <= wptr + 11'b00000000001;
        else
            wptr <= wptr;
        end
endmodule
```

APPENDIX C

Testbench

Listing C.1: Testbench for the AXI_FIR_FIFOs_IP

```
'timescale      1 ns/ 1 ps
'define         DELAY 5

module          Testbench();

// 4. Parameter definitions
localparam EOF = -1;
// 5. DUT Input regs
reg            start;
reg            clk;
reg            rst_n;
reg            wr;
reg            rd_l;

integer f_stim_ch2      , r_stim_ch2;
integer f_stim_ch1      , r_stim_ch1;

reg  signed [31:0]  datai_ch1;
reg  signed [31:0]  datai_ch2;
wire signed [31:0]  datao_ch1;
wire signed [31:0]  datao_ch2;
wire signed [31:0]  data_out_FIFO;

// 7. DUT Instantiation
design_1 design_1_i
(.clk(clk),
.rst_n(rst_n),
.start(start),
.data_in(datai_ch1),
```

```

.data_in_1(datai_ch2),
.data_intermediate_fifo_fir(data_out_FIFO),
.data_out(datao_ch1),
.data_out_1(datao_ch2),
.rd_1(rd_1),
.wr(wr));

//ENDSIM
reg test_fail = 0;

// 8. Initial Conditions
initial
    begin
        f_stim_ch1 = $fopen("ovs_in_ch1.txt", "r");
        if (f_stim_ch1 == 0) $stop;
        f_stim_ch2 = $fopen("ovs_in_ch2.txt", "r");
        if (f_stim_ch2 == 0) $stop;

        clk      = 1'b0;
        wr        = 1'b1;
        rd_1      = 1'b0;
        rst_n     = 0;
        start     = 0;

        #100;
        rst_n     = 1;

        #500;
        start     = 1;

        #50000;
        rd_1      = 1'b1;
    end

// 9. Generating Test Vectors

initial
    begin
        while (1) begin
            clk <= 0;
            #(5);
            clk <= 1;
            #(5);
        end
    end
end

```

```
always @ (posedge clk or negedge rst_n)
    begin
        if (rst_n)
            begin
                r_stim_ch1 <= $fscanf(f_stim_ch1, "%d", datai_ch1);
                r_stim_ch2 <= $fscanf(f_stim_ch2, "%d", datai_ch2);
                if (r_stim_ch1 == EOF)
                    begin
                        $display(">>>");
                        $display(">>>_Input_file_1_end");
                        //end_simulation;
                    end
                else if (r_stim_ch2 == EOF)
                    begin
                        $display(">>>");
                        $display(">>>_Input_file_2_end");
                        //end_simulation;
                    end
            end
    end

end

endmodule
```

APPENDIX D

C application for the FPGA

Listing D.1: Test.c

```
/*
 * *****
 *
 * Copyright (C) 2009 – 2014 Xilinx, Inc. All rights reserved.
 *
 * Permission is hereby granted, free of charge, to any person obtaining a copy
 * of this software and associated documentation files (the "Software"), to deal
 * in the Software without restriction, including without limitation the rights
 * to use, copy, modify, merge, publish, distribute, sublicense, and/or sell
 * copies of the Software, and to permit persons to whom the Software is
 * furnished to do so, subject to the following conditions:
 *
 * The above copyright notice and this permission notice shall be included in
 * all copies or substantial portions of the Software.
 *
 * Use of the Software is limited solely to applications:
 * (a) running on a Xilinx device, or
 * (b) that interact with a Xilinx device through a bus or interconnect.
 *
 * THE SOFTWARE IS PROVIDED "AS-IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
 * IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
 * FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL
 * XILINX BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY,
 * WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF
 * OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
 * SOFTWARE.
 *
 * Except as contained in this notice, the name of the Xilinx shall not be used
 * in advertising or otherwise to promote the sale, use or other dealings in
 * this Software without prior written authorization from Xilinx.
 */
```

```

*
*****/

/*
* Test.c: simple test application
*
* This application configures UART 16550 to baud rate 9600.
* PS7 UART (Zynq) is not initialized by this application, since
* bootrom/bsp configures it to baud rate 115200
*
* -----
* | UART TYPE   BAUD RATE                               |
* -----
*   uartns550    9600
*   uartlite     Configurable only in HW design
*   ps7_uart     115200 (configured by bootrom/bsp)
*/

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xil_io.h"
#include "xuartps_hw.h"
#include "xuartps.h"
#include "sleep.h"
#include "AXI_FIR_Fifos.h"

int main()
{

s32 data_out;

init_platform();

s32 Inputs[385] = {
0,
487642562,
949807729,
1362349204,
1703713325,
1956065168,
2106220351,
...
-1703713325,
-1362349204,
-949807729,

```

```
-487642562,
0};

AXI_FIR_FIFOS_mWriteReg(0x43c00000 , AXI_FIR_FIFOS_S00_AXI_SLV_REG2_OFFSET,0);// start=0

for(int i=0; i<384; i++){ //write inputs on the first fifo
AXI_FIR_FIFOS_mWriteReg(0x43c00000 , AXI_FIR_FIFOS_S00_AXI_SLV_REG0_OFFSET, Inputs[i]);
}

AXI_FIR_FIFOS_mWriteReg(0x43c00000 , AXI_FIR_FIFOS_S00_AXI_SLV_REG2_OFFSET,1);// start=1

for(int j=0; j<5000; j++); //delay to be sure that the filter has finished

for(int k=0; k<1024; k = k+2){ //read the outputs of the second fifo
data_out=AXI_FIR_FIFOS_mReadReg(0x43c00000 , AXI_FIR_FIFOS_S00_AXI_SLV_REG12_OFFSET);
xil_printf("D_out[%d]:", k); xil_printf("%d_", data_out);
data_out=AXI_FIR_FIFOS_mReadReg(0x43c00000 , AXI_FIR_FIFOS_S00_AXI_SLV_REG12_OFFSET);
xil_printf("D_out[%d]:", k+1); xil_printf("%d_\n\r", data_out);
}

cleanup_platform();
return 0;
}
```