# Design of Digital Filters in FPGA using High Level Synthesis

G Ravi Kumar Reddy, Assistant Professor.
CVR College of Engineering College, ECE, Hyderabad, India.
Email: ravigrkr@gmail.com

*Abstract*—**This work is aimed at the high level synthesis of FPGA based IIR digital filters using Vivado HLS produced by Xilinx and HDL coder produced by Math works. The higher layer model of the filter was designed in Vivado HLS, MATLAB and simulink. Simulations, verification and synthesis of the RTL code were done for both the tools. Further optimizations were done so that the final design could meet the area, timing and throughput requirements. The resulting designs were later evaluated to see which of them satisfies the design objectives specified. The present work has revealed that Vivado HLS is able to generate more efficient designs than the HDL coder. Vivado provides the designer with more granularity to control scheduling and binding, the two processes at the heart of HLS. In addition, both tools provide the designer with transparency from modeling up to verification of the RTL code. HDL coder did not meet timing. Vivado HLS on the other hand met the timing requirements.**

*Index Terms*— FPGA, HDL, HLS, Synthesis, RTL, MATLAB, IIR.

## I. Introduction

Electronic products currently are composed of highly complex designs in such areas as; communication, control, medical, defense and consumer electronics. They feature in applications such as digital signal processing (DSP), communication protocols, soft processors etc. Many DSP algorithms such as FFTs, FIR or IIR, which were previously built using application specific integrated circuits (ASICs) can be built on FPGAs with very high flexibility. In addition, these devices offer better economic prospects as compared to the ASICs. Consequently designs that were previously implemented on ASICs have experienced a move to the reconfigurable technology. These designs have become increasingly complex and are stretching the boundaries of device density, design performance and device power consumption. It is always the objective of designers to minimize costs by utilizing device resources appropriately to meet design objectives. In addition, given the shortened windows of design development time, it is very important to hit the target for the design objectives within the allocated time and schedule. Many downstream problems can be avoided with an appropriate methodology during the design flow. By taking appropriate steps early in the design phase, significant design productivity and minimized iterations can be achieved. It is therefore important to utilize tools that offer a good design methodology and provide proper estimates of project viability, cost and design closure early in the design phase [8].

In the applications of ABB HVDC (High Voltage Direct Current), voltage and current measuring IO-units in the Modular Advanced Control for HVDC system (MACH) [9] perform digital filtering of analog signals after analog to digital conversion. The filtering in the digital domain is done by Digital Signal Processors (DSP) and / or Field Programmable Gate Arrays (FPGA). An efficient way for filter designing is using VHDL (Very-high-speed integrated- circuits Hardware Description Language). When filters are implemented in FPGAs, the corresponding VHDL-code is usually written at Register Transfer Level (RTL) which is thereafter synthesized into logic gates. This means that the filter architecture and characteristics need to be determined before the implementation is done. Also, once the implementation is done, an architectural change on the filters may cause a large impact on the implementation, and may result in a change of most of the RTL-code.

There is plenty of High Level Synthesis (HLS) tools available for FPGA design on the market today e.g. HDL coder tool, Vivado HLS tool, Catapult e.tc. An HLS tool usually takes in a higher level language description, for example in C, C++, MATLAB /Simulink or System-C and then based on directives, translates the high level code into RTL-code which can then be synthesized into logic gates. With this methodology, one can easily make changes in an algorithm and/or directives, and have the tool automatically regenerate the RTL-code.

## II. FPGA design Overview

The Field-Programmable Gate Array (FPGA) is a Prototype device that can be programmed after manufacturing. Instead of being restricted to any predetermined hardware function like an application specific integrated circuit (ASIC), an FPGA allows a designer to program functions and product features, adapt it to new standards, and reconfigure the hardware technology for specific applications even after the product has been installed in the field-hence the name "field-programmable". The FPGA configuration is generally specified using a hardware description language (HDL). FPGAs can be used to implement any logical functions that ASICs perform. In addition, the ability to update the functionality after shipping offers advantages for many applications as compared to the ASICs. Specifically


DOI: 10.32377/cvrjst0906

FPGAs offer the following advantages as compared to the ASICs.

FPGAs contain programmable logic components called "logic blocks or Logic elements", and a hierarchy of reconfigurable interconnects that allow the blocks to be "wired together" - somewhat like many changeable logic gates that can be inter-wired in many different configurations. Logic blocks can be configured to perform complex combinational functions, or merely simple logic gates like AND and XOR. In most FPGAs, the logic blocks also include memory elements, which may be simple flip-flops or more complete blocks of memory as shown in figure 1. Unlike previous generation FPGAs using I/Os with programmable logic and interconnects, today's FPGAs consist of various mixes of configurable embedded SRAM, high-speed transceivers, high-speed I/Os, logic blocks, and routing [5] .

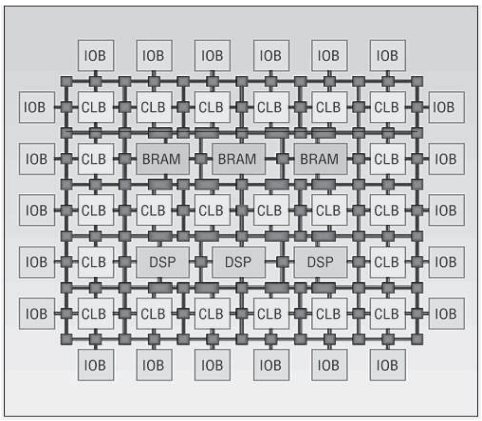


Figure 1: Modern FPGA Architecture

## III. Filter Realization

As a general rule, linear time-Invariant (LTI) systems can be classified into either finite impulse response (FIR) or infinite impulse response (IIR) depending on whether their operations have finite or infinite response duration. Additionally, depending on the application and hardware, the filtering operation can be organized to operate either as a single block or as a sample by sample process. With block processing, the input signal is considered to be a block of many samples. Essentially the block is filtered by convolving it with the filter input, and the output is also obtained as a block of samples. In cases where the input is very large, it can be broken down into multiple blocks, filtered and then the output blocks pieced together again. This can be implemented by ordinary convolution or fast convolution algorithms.

In the sample processing case, the input samples are processed one at a time as they arrive at the input. In this scenario the filter operates like a state machine by utilizing the current sample together with current internal state of the filter to compute the current output sample. It also updates the current internal state in preparation for processing of the next sample. This paper expounds on the concepts of the sample by sample processing technique to develop a high level language (HLL) for a filter used in an instrumentation application on the MACH 2 platform.

In general FIR filters have an impulse response $h(n)$ that extends over a finite duration interval say $0 \le n \le M$, and is identically equal to zero elsewhere i.e. {$h0$, $h1$, $h2$, … $hM$, 0, 0, 0, 0, … }. M is referred to as the order of the filter and the impulse response coefficients [$h0$, $h1$, $h2$] are referred to as filter coefficients. In general the filter equation for the FIR filters is given by

$$y(n) = \sum_{0}^{M} h(m)x(n-m)$$

or, explicitly as

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + \cdots + h_Mx(n-M)$$

Thus the I/O equation is obtained as a weighed sum of the present input sample and the past M samples. IIR filters on the other hand have the impulse response $h(n)$ that extends over an infinite duration defined over the infinite interval $0 \le n \le \infty$. In general the equation for IIR filters is given by

$$y(n) = \sum_{0}^{\infty} h(m)x(n-m)$$

Or, explicitly as

$$y(n) = h_0x(n) + h_1x(n-1) + h_2x(n-2) + \cdots\cdots$$

This I/O equation is not computationally feasible since practical systems cannot deal with an infinite number of terms. Therefore, practical implementations normally restrict their attention to a subclass of IIR filters in which the infinite number of filter coefficients {$h0$, $h1$, $h2$, … } are not chosen arbitrarily, but rather they are coupled to each other through constant coefficient linear difference equations. With this subclass of IIR filters, their I/O equation can be rearranged as a difference equation allowing the efficient recursive computation of the output $y(n)$. Practical implementations are normally concerned with filters that have impulse responses $h(n)$ that satisfy the constant-coefficient difference equations of general type:

$$h(n) = \sum_{i=1}^{M} a_i h(n-i) + \sum_{i}^{L} b_i \delta(n-i)$$

The convolution equation can be written as:

$$y(n) = \sum_{i=1}^{M} a_i\, y(n-i) + \sum_{i}^{L} b_i x(n-i)$$

Or, explicitly as

$$y_n = a_1 y_{n-1} + a_2 y_{n-2} + \cdots a_M y_{n-M} + b_0 x_n + b_1 x_{n-1} + \cdots + b_M x_{n-L}$$

And in general one can think of FIR filters as a special case of IIR filters whose recursive terms are absent i.e. $a_1, a_2, \cdots, a_M = 0$.

This paper is concerned with IIR filters since the current design implementation is an IIR filter. The equivalent transfer function of the IIR filter can be represented as;

$$H(z) = Y(z)/X(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2} \cdots + b_L z^{-L}}{1 + a_1 z^{-1} + a_2 z^{-2} \cdots + a_M z^{-M}}$$

For a second order filter, the general form of the transfer function is

$$H(z) = Y(z)/X(z) = \frac{b_0 + b_1 z^{-1} + b_2 z^{-2}}{1 + a_1 z^{-1} + a_2 z^{-2}}$$

In addition, filters can be realized in different ways such as:1) Direct form, 2) Canonical form and 3) Cascade form. The IIR filter for which this investigation is performed is a cascade of seven second order sections, realized in direct form II (canonical form). The design is targeted for the 45nm low power process technology FPGAs which are optimized for cost, power, performance and most efficient utilization of low-power copper process technology [1]. The filter is targeted for use in the voltage and current measuring IO-units in the MACH control system that performs digital filtering of signals after analog to digital conversion. The system of equations (difference) for the second order section of the filter are shown as

$$d_0(n) = gain * x(n) - a_1 d_1(n) - a_2 d_2(n)$$
$$y(n) = b_0 d_0(n) + b_1 d_1(n) + b_2 d_2(n)$$
$$d_2(n+1) = d_1(n)$$
$$d_1(n+1) = d_0(n)$$

The cascade realization form of a general second order function assumes that the transfer function is the product of such second order transfer functions as shown above. In general any transfer function that can be factored into second order filters with real valued coefficients.

$$H(z) = \prod_{i=0}^{k-1} H_i(z) = \prod_{i=0}^{k-1} \frac{b_{0i} + b_{1i} z^{-1} + b_{2i} z^{-2}}{1 + a_{1i} z^{-1} + a_{2i} z^{-2}}$$

The current design is of order 14 and therefore, since the cascade is of second order sections, then $k = 7$.

$$H(z) = \prod_{i=0}^{6} H_i(z) = \prod_{i=0}^{6} \frac{b_{0i} + b_{1i} z^{-1} + b_{2i} z^{-2}}{1 + a_{1i} z^{-1} + a_{2i} z^{-2}}$$

The design upon which the first evaluation of the tools is made is the second order section of the filter as shown in the diagram in figure 2. This filter is an anti-aliasing filter that is placed between the DSP and the ADC. The ADCs sample at approximately 500 kHz whereas the filter outputs to the DSP run at a configurable rate depending on the measuring board. Typical values could be 50, 100 or 125 kHz. This is the fundamental reason why the anti-aliasing filter is needed between the DSP and the ADCs. The design of the filter is such that all the coefficients and internal states (delay registers), are stored in RAM. The coefficients and decimation parameter of the filter (g0, a1, a2, b0, b1, b2, ds) are configurable externally by the DSP and it is therefore possible to change the characteristics of the filter depending on which measuring board the communication board is piggybacked on. The filter block performs data filtering (anti-aliasing) as well as storage of filtered values and internal states into the DRAM.

The current implementation of the design utilizes one multiplier, one adder and one subtractor in a pipelined fashion in order to save the FPGA resources. It is the goal of this investigation to be able to achieve these resource restrictions while satisfying the timing of 100MHZ.

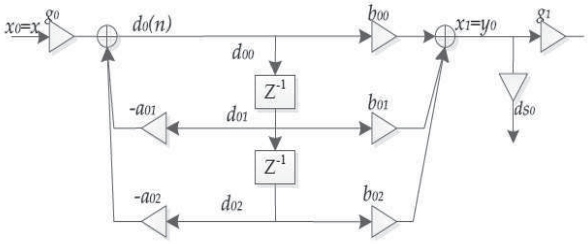


Figure 2. Second Order Section (SOS) of the IIR filter

## IV. Vivado High Level Synthesis

Vivado High Level Synthesis is Xilinx's HLS tool for transforming a C, C++ or SystemC specification into an RTL implementation which can then be packaged as an Intellectual Property (IP) core or exported as RTL source code for synthesis. This adds an extra layer of abstraction above the traditional RTL coding approach. The fundamental reason why the FPGA community has moved from one abstraction level to the next is to manage the complexity of the designs. The move is such that each added abstraction layer hides some complexity of the corresponding implementation step. The RTL description captures the desired functionality by defining data path and logic between boundaries of registers. RTL synthesis creates a netlist of Boolean functions to implement the design. The focus of the RTL abstraction layer is to define a functional model for the hardware [7]. A functional specification would therefore remove the need to define the register boundaries in order to implement a desired algorithm.

The designer's goal is now focused on only specifying the desired functionality. In Vivado HLS, moving up the design hierarchy to use the functional specification for creating RTL descriptions provides productivity in both verification and design optimization [2]. High-Level Synthesis shortens the manual RTL creation process and avoids translation errors by automating the creation of RTL from a functional specification. High-Level Synthesis automates RTL architecture optimization, allowing multiple architectures to quickly be evaluated before committing to an optimum solution. C based entry is the


DOI: 10.32377/cvrjst0906

most popular mechanism to create functional specifications and Vivado HLS currently supports to a synthesizable level all three C input standards(C, C++ and SystemC). This enables it to simulate C code with minimal modifications.

As described in [3], the Vivado HLS tool performs two distinct kinds of synthesis on the design; Algorithm synthesis takes the content of the functions and synthesizes the functional statements into RTL statements over several clock cycles. This type of synthesis typically builds the algorithm and is significantly affected by the interface level synthesis described below[4]. Interface synthesis transforms the function arguments (or parameters) into RTL ports with specific timing protocols, allowing the design to communicate with other designs in the system. The types of interfaces that can be synthesized are wire, Register one way and two way handshakes, Bus, FIFO and RAM. In addition, functional level protocols that dictate when a function can start or end can be synthesized.

*A) Area Optimization in Vivado HLS*

The design objective of this exercise in terms of area optimization is to be able to utilize one multiplier, one adder and one subtractor in a pipelined fashion in order to save the FPGA resources. This measure comes as a design specification set in the current solution to minimize resource usage on the FPGA. It suffices to say therefore that the tool chosen for use ultimately should be able to meet at least these basic requirements for this simple design. Vivado HLS comes with a set of directives that a designer can use to control the operators,resources, the binding process and the binding effort level [3]. The first activity is to limit the number of operators used in the design. In order to do this, Vivado uses the command set_directive_allocation [OPTIONS] <location> <instances>. This command specifies the instance restrictions for resource allocation. It defines, and can limit, the number of RTL instances that are used to implement specific functions or operations. The set_directive_allocation command can either be embedded in the source code as #pragma or placed in the directives tcl file. Both options are configurable using the HLS command line interface or the HLS GUI. In this exercise, the first option is preferred so that the directives are carried over across different solutions. Table 1 shows a comparison between the generated design and the hand coded design.

TABLE .1
RESOURCE USAGE STATICS FOR THE CURRENT DESIGN AND THE GENERATED DESIGN

| Resource | Current Design | Generated Design |
|---|---|---|
| Slices | 150 | 160 |
| LUTS | 277 | 398 |
| FF | 486 | 417 |
| DSP | 4 | 4 |
| BRAM | 0 | 0 |
| SRL | 18 | 19 |

Overall the Vivado HLS tool achieves almost the same resource utilization as the handwritten code. In the table above, there are a rise in the used slices by 10, a rise in the used LUTS by 121, a reduction in the number of FF by 69 and a rise of SRLs by 1. In percentages of available resources on the device, as calculated by ISE, the device utilization of the two designs is the same. In conclusion, it is reasonable to say that one can achieve very good area optimization with appropriate directives in the C/C++ design.

*B) Timing Optimizations*

Both designs exceed the timing requirement by a very good margin; however the current handwritten design achieves a higher operation frequency of 129.467MHZ as compared to 114.338MHZ in the generated design. It is however important to note that the timing requirements of the generated code can be altered as quickly as the design can be regenerated which is comparably difficult when it comes to adjusting timing requirements for handwritten RTL designs in general. For more details about the timing results see following timing reports shown in figure 3.

**Static Timing Report for the generated design:**

```
Timing summary:
---------------
Timing errors: 0 Score: 0 (Setup/Max:
0, Hold: 0)
Constraints cover 19793 paths, 0 nets,
and 2299 connections
Design statistics:
Minimum period: 8.746ns{1} (Maximum
frequency: 114.338MHz)
```

**Static Timing Report for the current design:**

```
Timing summary:
---------------
Timing errors: 0 Score: 0
(Setup/Max: 0, Hold: 0)
Constraints cover 17411 paths, 0
nets, and 1826 connections
Design statistics:
Minimum period: 7.724ns {1} (Maximum
frequency: 129.467MHz)
```

Figure 3: Static Timing Reports for the generated design and current design

*C) Latency and Throughput Optimizations*

Overall the latency and the iteration interval for the current design and the generated design are equal, however the value for the delay register zero of the filter gets ready after 14 clock cycles in the generated design as compared to 8 clock cycles in the handwritten RTL code.
This value cannot be improved since latency and throughput constraints cannot be specified for this single output value. The tables 2 and 3 illustrate the Latency and throughput values.


DOI: 10.32377/cvrjst0906

TABLE 2
LATENCY AND ITERATION INTERVAL VALUES FOR THE CURRENT DESIGN.

| latency | | Interval | |
|---|---|---|---|
| min | max | min | max |
| 19 | 19 | 6 | 6 |

TABLE 3
LATENCY AND ITERATION INTERVAL VALUES FOR THE GENERATED DESIGN.

| latency | | Interval | |
|---|---|---|---|
| min | max | min | max |
| 19 | 19 | 6 | 6 |

## V. HDL Coder

The HDL coder tool embedded in the MATLAB/Simulink environment lets a designer generate synthesizable HDL code for FPGA and ASIC implementations in the following steps:

- Build a model of the design using a combination of MATLAB code, Simulink and State flow charts.
- Optimize the design to meet area-speed objectives.
- Generate the design using the integrated HDL workflow advisor for MATLAB and Simulink
- Verify the generated code using HDL verifier.

HDL coder also features a Workflow advisor for automating the FPGA design process from MATLAB algorithms and Simulink models into Xilinx and Altera FPGAs. The HDL Workflow Advisor integrates all steps for traditional FPGA design process, and also includes the following features [6]:

- Checking the Simulink model for HDL code generation compatibility. Generating HDL code, an HDL test bench, and a co-simulation model
- Performing synthesis and timing analysis through integration with Xilinx ISE and Altera Quartus II
- Estimating resources used in the design
- Back annotating the Simulink model with critical path timing

The HDL tool in MATLAB leverages on the HDL Workflow to guide the designer during the process of generating HDL code. The HDL Workflow Advisor automatically converts MATLAB code from floating-point to fixed-point and generates synthesizable VHDL and Verilog code. Similarly the Workflow Advisor can generate VHDL and Verilog code from Simulink and State flow. The power of the tool lies in its ability to generate code from algorithms built using a library of more than 200 blocks, including State flow charts. In addition, the MATLAB language gives designers the capability to model their algorithm at a high level using abstract MATLAB constructs. This, together with the huge library provides complex functions, such as the Viterbi decoder, FFT, CIC filters, and FIR filters, for modeling signal processing and communications systems, and generating HDL code [6].

*A) Area Optimization in HDL coder*

HDL coder fundamentally uses the sharing optimization to ensure resource re-use in the generated RTL code. In addition, MathWorks advises designers to follow the following guidelines when implementing designs in MATLAB/Simulink [5]: Input and output data should be serialized since parallel data processing structures require more hardware resources and a higher pin count. Designers should use add and subtract algorithms instead of algorithms that use functions like sine, divide and modulo. This is because add and subtract operations use fewer hardware resources. Designers should avoid large arrays and matrices since they require more registers and RAM for storage. Code should be converted from floating-point to fixed-point since floating-point data types are inefficient for hardware realization.

HDL coder provides an automated workflow for floating-point to fixed-point conversion as discussed earlier. In addition unrolling loops increases speed at the cost of higher area; unrolling fewer loops and enabling the loop streaming optimization conserves area at the cost of throughput. By default, HDL implements hardware that is a 1-to-1 mapping of Simulink blocks to hardware module implementations. The resource sharing optimization enables users to share hardware resources by enabling an N-to-1 mapping of 'N' functionally-equivalent Simulink blocks to a single hardware module. The user specifies 'N' using the 'Sharing Factor' implementation parameter.

Without the sharing factor, the design is able to achieve the following hardware resource usage as viewed from the code generation report.

TABLE 4
RESOURCE USAGE BEFORE SHARING

| | |
|---|---|
| multipliers | 6 |
| Adders/Subtractors | 6 |
| Registers | 21 |
| RAMs | 0 |
| Multiplexers | 0 |

Table 4 above clearly shows that the number of multipliers (corresponding to DSP48As since the signal processing parameter is turned to on in the model) used is very high, approximately 41% of the entire available DSP48A resources available on chip, which is not acceptable in this design. One can deduce therefore from looking at the instances that the increased number of DSP48A usage is due to the number of multiplication blocks in the model. Each multiplier block corresponds to 4 DSP48As. There are approximately 6 multiplication blocks and these will result in 24 DSP48As. The objective at hand therefore is to reduce this number to just 4 DSP48As instances which correspond to only one multiplier implementation in the design. By setting the sharing factor to 6, the number of multiplier blocks in the design could be reduced to 1. The synthesis and mapping results clearly demonstrate the EDA tool results. Important to note at this point is that HDL coder does not give actual


DOI: 10.32377/cvrjst0906

resource usage estimates in a summarized form as Vivado HLS. The designer has therefore to invoke the required EDA tool and examine the reports generated by the tool for thorough conclusions on resource usage. Table 5 below shows resource utilization after resource sharing. The case for the multipliers is fairly trivial. Since the multipliers are 6, a sharing factor of 6 results in one multiplier being shared in the design. The design however uses more registers as can be seen by the increase in the number of the registers and flip-flops used by the design. However this is expected, and the end result is conserving more multipliers which are in less numbers as compared to flip-flops. Tthe tool is able to reduce the overall area usage.

TABLE 5

RESOURCE USAGE AFTER SHARING.

| | |
|---|---|
| Multipliers | 1 |
| Adders/Subtractors | 6 |
| Registers | 28 |
| RAMs | 0 |
| Multiplexers | 18 |

Table 6 shows the results after design synthesis using ISE by HDL coder.

TABLE 6

RESOURCE USAGE AFTER DESIGN SYNTHESIS

| Resource | Number |
|---|---|
| Slices | 340 |
| LUTS | 679 |
| FF | 1132 |
| DSP | 4 |
| BRAM | 0 |
| SRL | 28 |

In conclusion, synthesis and mapping results from ISE reveal considerable reduction in the area used by the design after sharing, which is a significant advantage of the tool. Significant to these reductions and increases in these designs however is but one important design achievement; the number of DSP48A1s used for the design has been reduced. Without a sharing factor, up to 41% of the DSP48A1s on the FPGA are used. With a sharing factor, this percentage reduces to just 6%, which is a considerable reduction in the resources. Correspondingly there is an increase in the number of LUTS, SRLs, FF and Slices in the design by a factor of approximately 2 overall. If ignored this could result in increasingly large designs.

*B) Comparison of Area Statistics with the Current Implementation*

Overall the HDL coder generates VHDL code with a high resource usage. In table 7, there is a rise in the used slices by a factor of 2.3, a rise in the used LUTS by a factor of 2.5, a rise in the number of used flip-flops by a factor of 2.3 and a rise of SRLs by a factor of 1.6. The DSP481As are however the same. In conclusion, it is reasonable to say that code generation with HDL coder results in increased resource usage.

TABLE 7

HDL CODER RESOURCE USAGE COMPARISON WITH THE CURRENT DESIGN

| Resource | Current Design | Generated Design |
|---|---|---|
| Slices | 150 | 340 |
| LUTS | 277 | 679 |
| FF | 486 | 1132 |
| DSP | 4 | 4 |
| BRAM | 0 | 0 |
| SRL | 18 | 28 |

*C) Timing Optimization in HDL Coder*

HDL coder utilizes the concept of distributed pipelining which is a subsystem-wide optimization to achieve high clock speed hardware. By turning on 'Distributed Pipelining', the coder redistributes the input pipeline registers, output pipeline registers of the subsystem and the registers in the subsystem to appropriate positions to minimize the combinatorial logic between registers and maximize the clock speed of the chip synthesized from the generated HDL code [5]. To increase the clock speed for any given design, the designer can set a number of pipeline stages for any subsystem. Without turning on distributed pipelining, the specified number of registers will be added to each of the output ports of the subsystem. Once distributed pipelining is turned on, the registers in the subsystem, including output pipeline registers and input pipeline registers, will be repositioned to achieve best clock speed. It is equivalent to retiming at subsystem level [5] as shown in figure 4.

```
Timing summary:
---------------
Timing errors: 0 Score: 0 (Setup/Max:
0, Hold: 0)
Constraints cover 31148965 paths, 0
nets, and 5500 connections
Design statistics:
Minimum period: 14.880ns{1} (Maximum
frequency: 67.204MHz)
Minimum input required time before
clock: 17.503ns
   Maximum output delay after clock:
              13.200ns
```

Figure 4: Timing report using HDL Coder

*D) Latency and Throughput in HDL Coder*

In order to increase throughput, HDL coder gives the option of pipelining. This option is handled together with improving timing in the section below. In addition the coder specifies a maximum computation latency parameter which enables designers to specify a time budget for the HDL coder when performing a single computation. Within this time budget, the coder does its best to optimize the design without exceeding the maximum oversampling ratio. When the designer sets a maximum computation



DOI: 10.32377/cvrjst0906

latency, N, each Simulink time step takes N time steps in the implemented design. In essence what this means is that the coder implements a design which captures the DUT inputs once every N clock cycles, starting with the first cycle after reset. The DUT outputs are held stable for N cycles. The requirement of this filter is that the design should be able to have at least an iteration interval of 6, so the maximum computational latency was set to 6.

## VI. Conclusion

This work has shown that using high level synthesis in FPGA, application development significantly achieves accelerated product development cycles. This work has established that the Vivado HLS tool provides the designer with a mechanism for influencing the HLS process (scheduling and binding) with significant granularity as compared to the HDL coder. A designer can explicitly specify the number of operations, specific cores, function instances, RAM cores, communication interfaces etc. quite easily in Vivado HLS as compared to the HDL coder. In addition the tool provides the designer with a detailed analysis of the design with clock level granularity i.e. the designer is able to establish quite easily which operations are performed in which clock cycle and which variables, either in the source code or the generated code, are affected.

This work has also shown that the designer may be able to achieve the design objectives of area, throughput, latency and timing in an easier way in Vivado HLS as compared to HDL coder. This may be fundamentally because the aspects of pipelining, resource usage etc., are handled in a much better way in Vivado HLS compared to HDL coder. The HDL coder workflow also supports both Altera and Xilinx FPGAs and seamlessly integrates into their respective synthesis tools. This gives developers a wider coverage of hardware technology. In addition, the tool supports addition of legacy code for final design synthesis.

## References

[1] R. Zoss et al, “Comparing Signal Processing Hardware-Synthesis Methods Based on the MATLAB Tool-Chain,” 2011, January IEEE Sixth IEEE International Symposium on Electronic Design, Test and Application (DELTA)

[2] W. Meeus et al , “An overview of today’s high-level synthesis tools” Journal for Design Automation for Embedded Systems; September 2012, Volume 16, Issue 3, pp 31-51

[3] Xilinx (2013, June 19), Vivado Design Suite User Guide on High Level Synthesis, UG902 (v2013.2).

[4] P. Coussy et al , “An Introduction to High-Level Synthesis,” Design & Test of Computers, IEEE (Volume:26 , Issue: 4 ), August 2009

[5] M. Haldar et al, “FPGA Hardware Synthesis from MATLAB,” IEEE, Fourteenth International Conference on VLSI Design, January 2001

[6] C .Tseng, “ Automated Synthesis of Data Paths in Digital Systems,” IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems CAD-5, 3, July 1986, pp. 379-395.

[7] H. Flamel et al , “A High-Level Hardware Compiler,” IEEE Transactions on CAD CAD-6, 2 , March 1987, pp. 259-269.

[8] G.E. Moore, “Cramming more components onto integrated circuits.” Proceedings of the ieee, vol. 86, no. 1, january 1998, pp. 144–144116

[9] L. Araujo et al, "MACH2-modular advanced control 2nd edition," Transmission and Distribution Conference and Exposition, Latin America, 2004 IEEE/PES, vol., no., pp.884,889, 8-11 Nov. 2004


DOI: 10.32377/cvrjst0906