VHDL Design Tool Flow for Portable FPGA Implementation

Vijaykumar Guddad, Alexandra Kourfali and Dirk Stroobandt
ELIS department, Computer Systems Lab, Ghent University
iGent, Technologiepark Zwijnaarde 126, B-9052 Ghent - Belgium
Email: {Vijaykumar.Guddad, Alexandra.Kourfali, Dirk.Stroobandt}@UGent.be

***Abstract*—In Field-Programmable Gate Array (FPGA) design, the coding style has a considerable impact on how an application is implemented and how it performs. Many popular Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) logic synthesis tools like Vivado by Xilinx, Quartus II by Altera, and IspLever by Lattice Semiconductor, have significantly improved the optimization algorithm for FPGA synthesis. However, the designer still has to generate synthesizable VHDL code that leads the synthesis tools and achieves the required result for a given hardware architecture. To meet the required performance, VHDL based hardware designers follow their own rules of thumb, and there are many research papers which suggest best practices for VHDL hardware designers. However, as many trade-offs have to be made and results depend on the combination of optimized implementations and optimized hardware architectures, final implementation decisions may have to change over time. In this paper, we present a VHDL design tool flow that makes portability of the design to new design requirements easier. It helps to generate automated portable VHDL design implementations and customized portable VHDL design implementations. This tool flow helps the VHDL hardware designers to generate a single VHDL design file, with multiple design parameters. It also helps the end-users of VHDL hardware designs in choosing the right parameter settings for a given hardware architecture and generating the right bit file corresponding to these parameter settings, according to their requirements.**

Keywords–FPGA; VHDL; Toolflow; VIVADO.

I. INTRODUCTION

Field-Programmable Gate Arrays (FPGAs) are attractive platforms for custom hardware implementation. They have been used in accelerating high-performance applications in which the complexity is significantly reduced by employing custom hardware to parts of the problem. They have been attractive for many new applications in which their flexibility and configurability are in high demand. The FPGA's strength comes from the fact that hardware developers can program it to deliver exactly what they need for their design, and massive spatial parallelism at low energy gives FPGAs the potential to be core components in large scale High-Performance Computing (HPC).

Very-High-Speed Integrated Circuits Hardware Description Language (VHDL) is a Hardware Description Language (HDL) that is used to program FPGAs. It describes the behaviour of an electronic circuit or system, from which the physical circuit or system can then be implemented. FPGAs work on configuration bits that define the functionality. To generate configuration bits from HDL, an FPGA tool flow is used. An FPGA tool flow aims to produce a configuration for the target FPGA that implements the functionality described in the HDL design. The current FPGA tool flow consists of synthesis, technology mapping, packing placement and routing. In the synthesis step, the HDL code is translated from a human-readable form to a gate-level logic circuit. The synthesis tool is also responsible for optimising the circuits depending on the needs of the designer. During technology mapping, the gate-level circuit generated by the synthesis step is mapped onto the resource primitives (ex. Lookup tables (LUTs), Flipflops (FFs), DSP blocks (DSPs), BlockRAMs) available in the target FPGA architecture. In packing, LUT primitives and FFs from the mapped netlist are clustered into Configuration Logic Blocks (CLBs) according to their interconnectivity. During placement and routing, CLBs are assigned to physical logic blocks on the FPGA, and these CLBs are connected using switch blocks and wires. Finally, the configuration bitstream is generated. Recently, increasing research has been performed in the field of placement and routing, and also commercial FPGA tools (Vivado by Xilinx, Quartus II by Altera, IspLever by Lattice Semiconductor, Encounter RTL compiler by Cadence Design Systems, LeonardoSpectrum, Precision by Mentor Graphics, and Synplify by Synopsys) optimised their algorithm for better results [1]–[3].

The tool flow proposed in this paper is mainly related to targetting the synthesis step. The coding style can have a severe impact on the resource utilisation of an FPGA architecture, and how it performs on the target board. The designer can write HDL code that forces the synthesis step to make use of available FPGA resources or not to use the specific resources. The designer can also write HDL code that has higher or lower throughput or a different design operating frequency. There are many research papers and books [4]–[6] which suggest best practices and techniques; also, each commercial FPGA vendor has their own coding guidelines [7] [8]. Apart from all these guidelines, each designer and company follows their own rules of thumb in order to achieve the required results according to the design requirements.These techniques and guidelines are specific to the particular FPGA architecture and model. As we can see in the current market, FPGA architectures and models keep changing to fulfill a new market need (e.g., currently, most FPGA vendors are designing their boards to target machine learning, deep neural networks, and data server requirements). From the above discussion, we can categorize these coding techniques and methods into three main categories: i) technology independent coding styles, ii) performance driven coding, iii) technolonology specific coding techniques.

In this paper, we present a method to combine all possible coding techniques, methods, and rules of thumb in a single VHDL design file. The tool flow processes the VHDL design file with all possible techniques, methods, and rules of thumb, which is independent of the FPGA vendor and the architecture. In Section II and Section III, we present different types of VHDL coding techniques and methods and describe a method

 ISBN: 978-1-61208-748-1

to combine all possible techniques in a single VHDL design file. In Section IV, we present the portable tool flow integrated with a synthesis tool to process input VHDL design files. In Section V, we give results and a conclusion.

II. VHDL CODING TECHNIQUES AND METHODS

In this section, we will discuss three main coding techniques and methods with an example. In the end, we describe a method to combine these techniques and methods in a single VHDL design file.

A. Technology independent coding styles

As the name suggests, technology independent coding techniques are independent of the FPGA architecture, vendor, and technology. Here, we will discuss a few techniques [9]–[12].

1) Sequential devices design techniques: In sequential devices, we have two main types of memory devices: a latch and a flip-flop. A latch is a level-sensitive memory device and a flip-flop is an edge triggered memory device [3] [13].

Data-Latches: Here we will see different ways, of using Data-latches (D-latches).

D-Latch with data and enable:

```
begin
process (enable, data) begin
   if (enable= '1') then
   y<=data;
```

D-Latch with gated asynchronous data:

```
process (enable, gate, data) begin
    if (enable = '1') then
       q <= data and gate;
```

D-Latch with gated enable:

```
process (enable, gate, d) begin
    if ((enable and gate) = '1') then
         q <= d;
```

D-Latch with asynchronous reset:

```
process (enable, data, reset) begin
    if (reset = '0') then
       q <= '0';
    elsif (enable = '1') then
       q <= data;
```

2) Datapath: Datapath logic is a structured repetitive function. These structures can be modelled in a different implementation depending upon timing and area constraints. The following synthesis tools generate optimal implementations for the target technology depending upon the datapath model used in the VHDL code [9] [13].

(i) Using if-then-else and case statement: An if-then-else statement is used to execute sequential statements based on a condition. Each of the if-then-else statements is checked until a true condition is found. Statements associated with a true condition are executed and the rest of the statement is ignored. Using if-then-else statements in VHDL code forces synthesis tools to realize the circuit in a way shown in Figure 1.

A case statement implies parallel encoding and a case statement is used to select one of several alternative statement

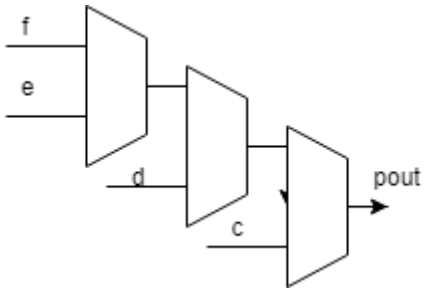

Figure 1. Effect on synthesis tool using an if-then-else statement

sequences based on the value of a condition. The condition is checked against each choice in the case statement until the match is found. Using case statements in VHDL code forces synthesis tools to realise the circuit in a way shown in Figure 2.

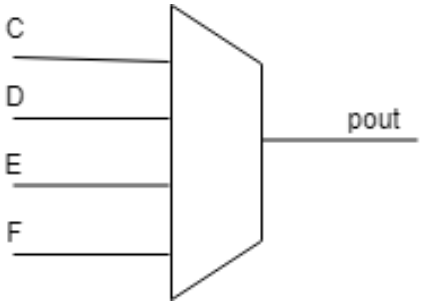

Figure 2. Effect on synthesis tool using a case statement

While writing VHDL code, it is difficult to predict how using an if-then-else statement or a case statement will effect the critical path of the final design or the design throughput or design requirements. The optimisation level of each statement varies from one synthesis tool to the other. In such cases, we can define both statements in a VHDL design file using the keyword - -# using_case and - -#using_if_else. Later, we can make choices at the synthesis step depending upon requirements using the tool flow presented in the next section.

(ii) Designing Multiplexers: While writing VHDL code, we can force the synthesis tool to make use of 4:1 or 6:1 or 12:1mux. The LUT inputs vary with the architecture, thus optimizing for different mux types can affect the synthesis to architecture step [10] [14].

(iii) Counters: Counters count the number of occurrences of an event that occurs either at regular intervals or randomly. Counters can be designed in one of the following ways: i) a counter with count enable and asynchronous reset, ii) a counter with load and asynchronous reset, and iii) a counter with load, count enable, and asynchronous reset. However, most synthesis tools cannot find the optimal implementation of counters higher than 8- bits. If the counter is in the critical path of a speed and area critical design, it is better to redesign using one of the ways mentioned above or to use a pre-instantiated counter provided by the vendor [9] [10].

3) Input-output buffers: We can infer or instantiate an Input/Output buffer in the VHDL design depending upon design requirements. The usage of inference and instantiation

 ISBN: 978-1-61208-748-1

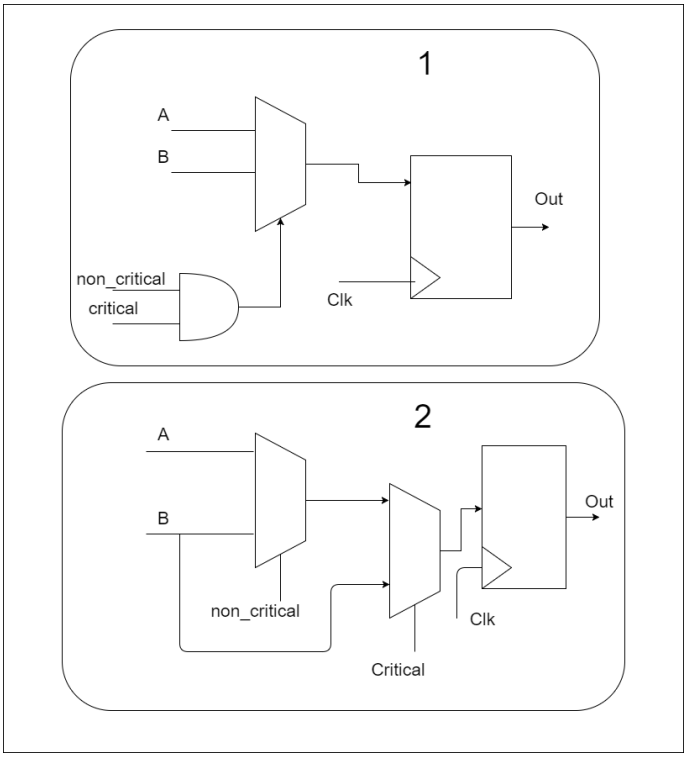

Figure 3. Example circuit designs

has its own advantages and disadvantages. For example, in the inference method, we define a tri-state buffer using an entity port in the design, whereas in the instantiation method, we make use of the tristate component design provided by synthesis tools [4] [9].

B. Performance driven coding

In the FPGA, each logic level used in the design path can add a delay. As a result, meeting timing and area constraints on a critical path with many logic levels becomes difficult. Using an efficient coding style is important because it dictates the synthesis logic implementation. In this section, we will discuss a few essential techniques [9] [15].

1) Reducing logic levels on critical paths: Consider a small circuit design, as shown in Figure 3. Here, we have two circuit designs with the same functionality but designed differently.

In circuit 1 of Figure 3, the signal ”critical” goes through two logic gates.

```
if (clk'event and clk ='1') then
     if (non_critical and critical) then
        out1 <= A
     else
        out1 <= B
     end if;
end if;
```

To reduce the logic gate usage on “critical” signals, multiplex inputs “A” and “B” based on “non_critical” and call this output “out_temp”. Then multiplex “out_temp” and “B” based on “critical”. As a result, the signal “critical” goes through one logic gate as shown in circuit 2 of Figure 3.

```
if (clk'event and clk ='1') then
     if (non_critical and critical) then
        out1 <= A
     else
        out1 <= B
     end if;
end if;
```

2) Resource sharing: The resource sharing technique is used to reduce the number of logic modules needed to implement VHDL operations. Here, we have two pieces of VHDL code: one makes use of four adders and another uses two adders.

```
--Example implementation with 4 Adders
if (...(siz == 1)...)
    count = count + 1;
else if (...((siz ==2)...)
    count = count + 2;
else if (...(siz == 3)...)
    count = count + 3;
else if (...(siz == 0)...)
     count = count + 4;
--Example implementation with 2 Adders
if (...(siz == 0)...)
   count = count + 4;
else
   count = count + siz
```

C. Technology specific coding techniques

These coding techniques are used to take advantage of the specific FPGA architecture, to improve speed and area utilization of the design. These techniques have their own coding guidelines to take advantage of their FPGA architectures [8] [16].

III. Method to combine all possible design techniques in a single VHDL design file

From the above discussions, we observe that the VHDL coding style has a considerable impact on how an FPGA design is implemented, and ultimately, how it performs. Here, we present a method to combine all possible VHDL design methods and techniques in a single design file. Let us consider a VHDL design file where we have two processes in a behavioral architecture and each process has to be described with different pipeline stages. Later, before the synthesis step, we plan to select the pipeline stages between each process, and a method to combine this would be as follows:

```
architecture behavioral of <identifier> is
begin
 process(<signal>)    -- Process 1
begin
  --#pipelined=0
  --  < code>
 --#pipelined=0
 --#pipelined=1
      < code >
 --#pipelined=1
 process(<signal>)    --Process 2
begin
  --#pipelined=0
     < code>
```

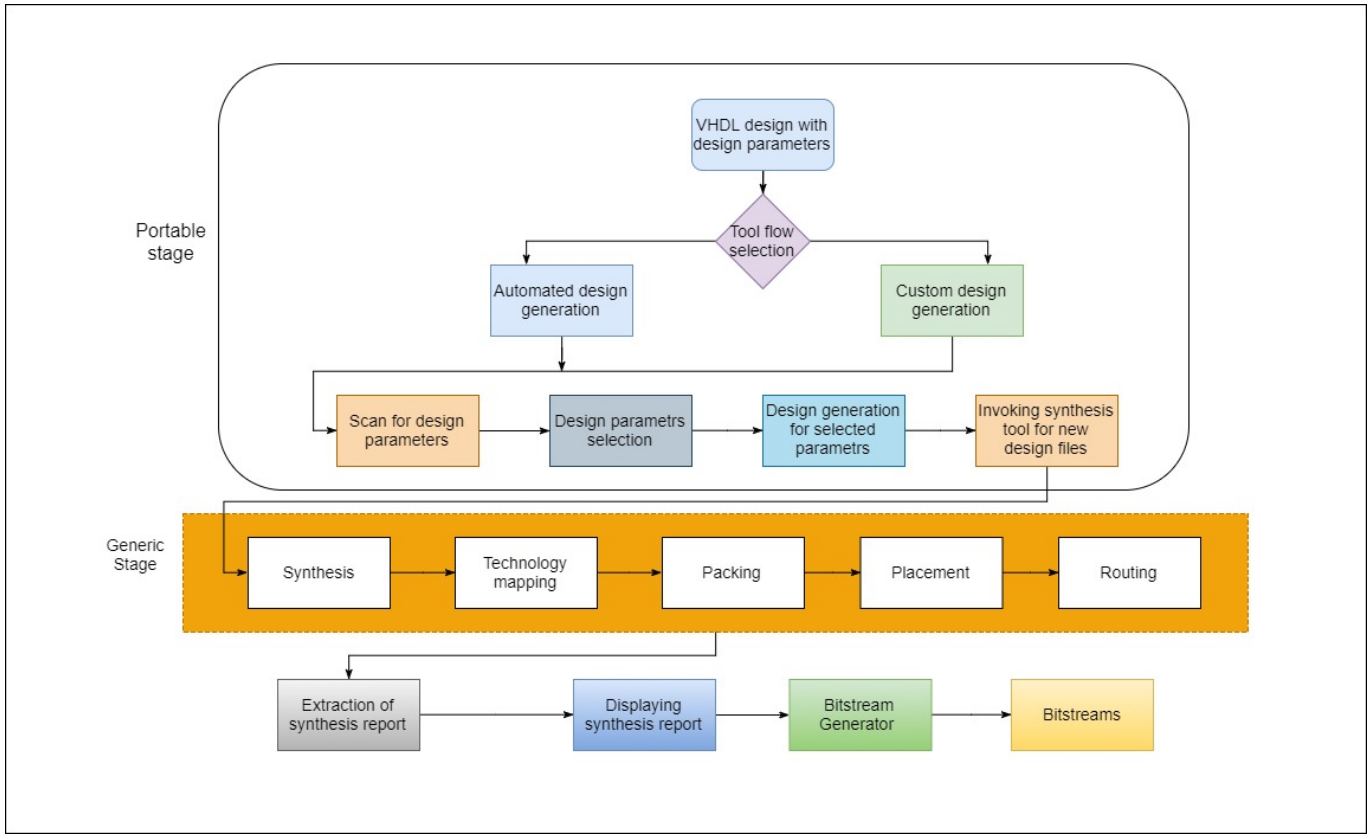

Figure 4. VHDL design tool flow for portable FPGA implementation

```
--#pipelined=0
--#pipelined=1
--     < code >
--#pipelined=1
```

Here, we make use of the keyword "- -#" followed by a design parameter instance. We can use this keyword to define any part of the code like input-output ports, signals, or the architecture part of the code. We can use any possible name to define the design parameter instance and there is no syntax rule for the names. The only condition we put forward for different parameter design parts is that they should start and end with the same design parameter name (from the above code design, a parameter means –#pipelined=1, –#pipelined=2). From the above code, we can observe in process 1 we commented the pipelined=0 and kept the pipelined=1 as default. In process 2, we kept pipelined=0 as default. It will help in processing the design files without our tool flow.

IV. VHDL Design Tool Flow for Portable Design Generation

As seen in Section III, now we have a way to combine multiple design techniques and methods in a single VHDL design file. However, this type of VHDL design file cannot be synthesised by regular synthesis tools. Therefore, we need a tool flow which can guide the synthesis tool in implementing combined VHDL design files, and which can allow the user to decide and generate portable VHDL designs.

In Figure 4, we present our proposed tool flow. As it can be observed, the tool flow is designed in two stages: the portable stage and the generic stage. The portable stage processes the input design files and allows the user to make the selection between different design parameters. The generic stage processes the new design files generated by the portable stage and generates a synthesis report and a bit-file.

In the portable stage, the tool flow receives the input design files. The tool flow scans for the design parameters available in the design files and prints them to the user. Then, the tool flow allows the user to select either automated design or customised design. In the automated design, the user can apply a selected set of design parameters or techniques to all available design files. In the customised design selection, the user can choose between different design files, for using selected design parameters. After successful selection of the design parameters available from the input design files, the tool flow searches for the user selected design parameters in each design file and extracts the design parameter content from each design file. If the user selected design parameter does not match with parameters in the design file, the tool flow will keep the default design parameter. After this the tool flow generates the new design files. The newly generated design files are processed further with the generic stage to generate a synthesis report. Next, the synthesis report is extracted and displayed to the user, and finally, the bit-file is generated. The portable tool flow proposed in this paper is independent of

the used FPGA architecture. This can be integrated with any FPGA architecture, just by changing the invoking statement in the code. For example, to invoke the Xilinx Vivado design suite, the following code is used:

```
vivado-mode batch -source design.tcl
```

One can think it is a lot of manual work to define all possible design parameters in a design file, but as we discussed in the introduction, our idea is to combine all possible coding techniques, methods, and rules of thumb in a single design file. It is easier for a designer to propose different smaller design options in sub-parts of the design than to provide different complete designs full of different choices. So the designer does not have to worry about how different choices are combined (as this is done automatically) but can focus on the individual different options. This is a huge difference.

V. Results and Discussion

In this section, we will evaluate the portable VHDL design tool flow and present the results [17]. In our experiments, we used an 8 core CPU system. We integrated our portable VHDL design tool flow with the Xilinx synthesis tool (Vivado 2018.3) [18], to generate the synthesis report and the bit-file.

A. Evaluating the portable VHDL design tool flow for a single design file (using technology independent coding styles)

Here, we considered the data flip-flop VHDL design file with seven design parameters (techniques) in a single VHDL design file. We compared our results with the standard VHDL design with a single design parameter. These results are tabulated in Table I.

TABLE I. RESULTS USING PORTABLE VHDL DESIGN TOOL FLOW AND GRAPHICAL USER INTERFACE OF SYNTHESIS TOOL

	Design parameters available	Time taken to write design file in Minutes (aprox)	Time required to edit for other design parameters in Minutes (aprox)	Space required to store design file	Time to run synthesis for all parameters
	Without uisng portable tool flow				
Data flip-flop design	1	3 Min	15 Min	4.00 KB	20 Min
	Using portable tool flow				
	7	7 Min	0 Min	7.02 KB	10 Min

From Table I, we can observe that we have two different implementation results: one using our portable VHDL design tool flow, and another one using the Graphical User Interface (GUI) of Vivado design suite. In this experiment, we considered the seven design techniques (parameters) to design a data flip-flop. These are i) rising edge flip-flop, ii) rising edge flip-flop with asynchronous reset, iii) rising edge flip-flop with asynchronous preset, iv) rising edge flip-flop with asynchronous reset and preset, v) rising edge flip-flop with synchronous reset, vi) rising edge flip-flop with synchronous preset, vii) rising edge flip-flop with asynchronous reset and clock enable. While using the GUI, we designed the data flip-flop considering one design technique at a time and subsequently edited the design for other parameters to get the corresponding synthesis report and the bit-file. In the second instance, we used the portable VHDL design tool flow, combining all techniques in a single design file and generating the synthesis report and the bit-file.

From the results in Table I, we can observe that writing a combination of required VHDL design techniques in a single file takes more time, but if we want to edit for other parameters or design techniques, it will take an extra 15 minutes at later stages without the use of our tool flow. Using our portable tool flow, we can generate the synthesis results and the bit-file for all seven design parameters at the same time, so we can easily compare the results and choose the right implementation. Otherwise, we need to edit the design file each time and run the synthesis tool.

B. Evaluating the portable VHDL design tool flow for multiple design files (H264 Video encoder design)

In this section, we evaluate our tool flow for the complete hardware H264 video encoder design, which consists of 15 different design blocks, as shown in Table II. Here, we added six design parameters in the design blocks. Using 6 different design parameters in the H264 encoder design, we are able to generate 8 different combinations, by selecting one design parameter each time or multiple design parameters in various combinations, which leads to different implementations and results. Using our tool flow here, we have the option of customisation by making parameter selections to a few design blocks.

Apart from portability options, we provide the automation option in running the complete synthesis steps. Using our tool flow, we can check the resource utilization for each design block. Our tool flow runs the synthesis step in parallel. A few more comparisons are tabulated in Table III.

TABLE II. EVALUATION OF HARDWARE VIDEO ENCODER H264 USING PORTABLE VHDL DESIGN TOOL FLOW

	Number of design blocks	Number of design parameters used	Design parameters used	Number of possible design implementations	Customisation option between different design blocks
Hardware H264 Video encoder design	15	6	1. Pipelined 2. Nonpipelined. 3. Less flip-flop design 4. More flip-flopdesign 5. Reduced logic 6. Normal logic	8	Yes

VI. Conclusion

In this paper, we proposed a way to combine the different possible VHDL design techniques and methods in a single VHDL design file. This can be evaluated into multiple (functionally equivalent) design files, that can be compared to allow the designer to choose between different implementation techniques, based on the achieved trade-off between FPGA resource utilisation and performance. In that way, by evaluating the same VHDL design file, the designer can estimate which design option is the better choice.

Additionally, we proposed a tool flow that allows the user to generate the design bit-files of all possible combinations automatically. This tool flow can be integrated into any other synthesis tool. Hence, it forms a pre-synthesis step, that provides the user with the flexibility of having multiple design

 ISBN: 978-1-61208-748-1

TABLE III. COMPARISON OF PORTABLE VHDL DESIGN TOOL FLOW OVER USUAL TOOL COMMAND LANGUAGE (TCL) AUTOMATION

	FPGA architecture independent	Parallel implementation of synthesis steps for multiple boards	Need to change in the design file for other parameters	Synthesis report on same screen	Synthesis report for each design file
Using Portable tool flow	Yes	Yes	No	Yes	Yes
Using normal automation TCL	No	No	Yes	No	No

options, but without having to redesign them, only to generate the new file, based on a set of parameters.

Using our tool flow, designers can easily redo the design exploration when the underlying FPGA architecture for the design changes. They do not have to delve into the VHDL source code for this. For the final design parameter choice, the tool can automatically generate the bitstream. Hence, this tool flow significantly enhances the portability of designs to new FPGA devices.

ACKNOWLEDGMENT

This work was supported by the Help Video! imec.icon research Project, funded by imec and the Flemish government (Agentschap innoveren and ondernemen).

REFERENCES

[1] J. de Fine Licht, M. Blott, and T. Hoefler, “Designing scalable FPGA architectures using high-level synthesis,” Proceedings of the 23rd ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming, 2018, pp. 403–404. [Online]. Available: http://doi.acm.org/10.1145/3178487.3178527

[2] K.Kuusilinna, Timo.Hmlinen, and Jukka.Saarinen, “Practical VHDL optimization for timing critical FPGA applications,” Microprocessors and Microsystems, vol. 23, no. 8, pp. 459–469.

[3] P. I. Necsulescu and V. Groza, “Automatic generation of VHDL hardware code from data flow graphs,” 2011 6th IEEE International Symposium on Applied Computational Intelligence and Informatics (SACI), May 2011, pp. 523–528.

[4] P. P. Chu, “Coding for efficiency, portability, and scalability,” hardware design using VHDL, 2006, pp 50-100, ISSN:13: 978-0-471-72092-8.

[5] Xilinx corporation ”Xilinx product guide”, 1.1, 2006, URL: https://bit.ly/2Gz2RO9/ [accessed: 2019-09-17].

[6] T. Davidson, K. Bruneel, and D. Stroobandt, “Identifying opportunities for dynamic circuit specialization,” 2012, workshop on Self-Awareness in Reconfigurable Computing Systems Proceedings, Oslo Norway, p-p 18-21. [Online]. Available: http://srcs12.doc.ic.ac.uk/docs/srcs_proceedings.pdf

[7] M. Arora, “The art of hardware architecture, design methods and techniques for digital circuits,” Design Methods and Techniques for Digital Circuits, springer-Verlag New York, 2011, DOI:10.1007/978-1-4614-0397-5.

[8] R. Jasinski, Effective Coding with VHDL: Principles and Best Practice. The MIT Press, 2016.

[9] J. C. Baraza, J. Gracia, D. Gil, and P. J. Gil, “Improvement of fault injection techniques based on VHDL code modification,” Tenth IEEE International High-Level Design Validation and Test Workshop, 2005., Nov 2005, pp. 19–26.

[10] R. P. P. Singh, P. Kumar, and B. Singh, “Performance analysis of fast adders using VHDL,” 2009 International Conference on Advances in Recent Technologies in Communication and Computing, Oct 2009, pp. 189–193.

[11] Z. Zhang, Q. Yu, L. Njilla, and C. Kamhoua, “FPGA-oriented moving target defense against security threats from malicious FPGA tools,” 2018 IEEE International Symposium on Hardware Oriented Security and Trust (HOST), April 2018, pp. 163–166.

[12] Z. Jia, B. Qi, L. Chen, H. Chen, and L. Ma, “Relative radiometric correction for remote sensing images based on VIVADO HLS,” IET International Radar Conference 2015, Oct 2015, pp. 1–4.

[13] V. S. Rosa, F. F. Daitx, E. Costa, and S. Bampi, “Design flow for the generation of optimized FIR filters,” Dec 2009, pp. 1000–1003.

[14] G. Donzellini and D. Ponta, “From gates to FPGA learning digital design with deeds,” March 2013, pp. 41–48.

[15] M. S. Sutaone and S. C. Badwaik, “Performance evaluation of VHDL coding techniques for optimized implementation of ieee 802.3 transmitter,” Jan 2008, pp. 287–293.

[16] Altera corporation, Altera product guide, 9.1, 2009, URL: http://bit.do/eJsnK/ [accessed: 2019-09-17].

[17] Github link, URL: https://bit.ly/2Sh1kgU/.

[18] Xilinx vivado URL: https://bit.ly/2AVvccx.

 ISBN: 978-1-61208-748-1