KU LEUVEN

FACULTEIT INDUSTRIËLE
INGENIEURSWETENSCHAPPEN

CAMPUS DE NAYER SINT-KATELIJNE-WAVER

FPGA Benchmarking

Dries KENNES

Promotor: dr. ing. Kristof Van Beeck
Co-promotoren: ir. Nicolas Huot

Masterproef ingediend tot het behalen van
de graad van master of Science in de
industriële wetenschappen: Elektronica-ICT
Afstudeerrichting ICT

Academiejaar 2018-2019

Acknowledgments

On the eve of my graduation as Master in Industrial Engineering in the wonderful field of electronics and IT, I would like to thank some people for their support over the past few years.

First and foremost I'd like to thank my thesis supervisor Nicolas Huot and Kristof Van Beeck for their help and guidance on this last leg of my journey.

I would also like to thank the friendly colleagues at Antwerp Space, in particular Jo Van Langendonck and Dave Geerardyn, for their help and warm welcoming environment.

Thanks to my sister Lina for trying to get this document as free from spelling mistakes as possible. Lastly I thank my parents for their support throughout my entire school carrier, in particular the very intense preparatory program last year.

~ Dries Kennes

English Abstract

In the world of small batch hardware production, FPGAs are used to implement functionality that would normally require very expensive custom chips to be produced. Users traditionally rely on the limited and one-sided information provided by manufacturers and their own design experience to chose the right FPGA for the job.

We believe that using custom benchmarks to generate relevant data will empower the user to make better estimates of both the FPGA's performance and the project's required resources. We developed a methodology and implemented software to automate the generation of statistics about FPGAs across different product families and even vendors.

Alongside a number of existing benchmarks, we created a custom benchmark to test our methodology and implementation. Based on the results of these tests we conclude that it provides valuable insight into the performance of the FPGA, at the cost of additional effort and time to select and compute the benchmarks.

Keywords: FPGA, Benchmarks, Performance analysis

Nederlandstalig Abstract

In de wereld van hardware in kleine oplagen worden FPGAs gebruikt om functionaliteit te implementeren die normaal onbetaalbare, op maat gemaakte chips, zou vereisen. Makers van deze producten zijn traditioneel afhankelijk van de fabrikanten en hun eigen ervaring voor het kiezen van een geschikte FPGA.

We zijn er van overtuigd dat het testen van een selectie op maat gemaakte benchmarks de gebruiker in staat zal stellen gerichte objectieve cijfers te produceren. Zowel over de performantie van de FPGA als over wat het project vereist. Hiervoor hebben we een methodologie ontworpen en software geschreven die het produceren van deze cijfers vergaand automatiseert. De software is geschikt om verschillende productfamilies en zelfs fabrikanten met elkaar te vergelijken.

Naast een selectie bestaande benchmarks hebben we ook onze eigen benchmark geïmplementeerd om de methodologie en software te testen. Gebaseerd op de resultaten van die tests concluderen we dat ons werk een waardevol inzicht bied. Dit inzicht komt ten koste van extra tijd en moeite die nodig is om geschikte benchmarks te selecteren en uit te voeren.

Sleutelwoorden: FPGA, Benchmarks, Performantieanalyse

Korte Samenvatting

Inleiding

In de wereld van hardware worden FPGAs gebruikt om functionaliteit te implementeren die normaal onbetaalbare, op maat gemaakte chips, zou vereisen. De ruimtevaartindustrie is hiervan het schoolvoorbeeld door de zeer kleine oplagen van de meeste projecten. Maar in deze industrie is er nog een toegevoegde complexiteit: de gebruikte componenten moeten geschikt zijn voor de extreme omgeving. Dit vereist specifieke versies die lang op voorhand besteld moeten worden. Hierdoor is de karakterisatie van de FPGA en de specificaties van het project extra belangrijk.

Ontwerpers van projecten die FPGAs gebruiken zijn traditioneel afhankelijk van de fabrikanten en hun eigen ervaring voor het kiezen van een geschikt model. We zijn er van overtuigd dat objectieve, relevante cijfers kunnen helpen bij deze selectie.

Literatuurstudie

De FPGA industrie is een niche, traditioneel zeer gesloten industrie. Dit maakt dat er weinig gepubliceerde werken zijn over dit onderwerp. Veel studies die we vonden zijn oud of toetsen andere onderwerpen af.

Wel relevant is de zogenoemde “stamping methodology” ofwel postzegelmethode. Die bestaat er uit een FPGA maximaal te vullen met identieke kopieën van hetzelfde ontwerp. Als de FPGA volledig vol zit wordt dan de kloksnelheid gemeten. Deze methode is vandaag de industriestandaard. Het ontwerp dat gekozen wordt is daarbij zeer belangrijk. Spijtig genoeg kan de fabrikant hier gemakkelijk een ontwerp kiezen dat alleen de eigen sterktes benadrukt en dat de zwakke punten negeert.

Nadat we onze werkingsmethode hadden uitgeschreven hebben we een project uit 2017 van de European Space Agency (ESA) ontdekt. De methodologie van dit project, de “Open ESA FPGA Benchmark Suite”, lijkt op de onze maar is beperkter in implementatie.

Methodologie en Implementatie

Het doel van deze thesis is het ontwikkelen van een platform dat een projectontwikkelaar in staat stelt objectieve, relevante cijfers te genereren over de FPGA en het project. Deze cijfers kunnen dan gebruikt worden tijdens de selectieprocedure.

Hiervoor hebben we een softwarepakket ontwikkeld in Python dat het proces bijna volledig automatiseert. De gebruiker kan een lijst met code bestanden doorgeven samen met een lijst van parameters. De software stuurt dan volautomatisch de software van de FPGA fabrikant aan. Deze zet de code om naar een hardware ontwerp voor de FPGA en bepaalt daarbij eigenschappen zoals de maximale klokfrequentie en de hoeveelheid middelen van de FPGA het ontwerp gebruikt.

Het ontwerpen van een benchmark wordt ook besproken. Vooral belangrijk is dat code geen IO pinnen aanspreekt maar interne registers gebruikt. IO pinnen zijn namelijk in snelheid gelimiteerd en bevinden zich enkel langs de buitenkant van de chip.

De verkregen data is vooral interessant als ze zou kunnen worden voorgesteld als een eenzijdig getal dat eenvoudig vergelijkbaar is met andere resultaten. Spijtig genoeg bestaat dit niet, omdat een FPGA te toepassingsspecifiek is. Daarom stellen we de resultaten voor als een grafiek die de gebruiker zelf kan interpreteren.

Per softwarepakket van de fabrikant is er een andere aanpak nodig. We kozen voor Vivado (van Xilinx) en nxPython (van NanoXplore) omdat Xilinx de grootste fabrikant is en omdat Antwerp Space (onze industriepartner) een specifieke interesse heeft voor NanoXplore. De integratie van deze softwarepakketten in onze software wordt uitgebreid besproken. Ten slotte kiezen we ook een FPGA model per fabrikant om onze benchmarks mee te testen.

Benchmarks en Resultaten

Onze belangrijkste testgevallen zijn de ISCAS'89 benchmark set en een zelfgemaakte combinatie tussen een van deze ISCAS'89 benchmarks en een FIR filter.

De ISCAS'89 set is een lijst van benchmarks die is ontworpen om digitale logica mee te testen. De benchmarks bevatten alleen maar simpele gates waardoor het een ideale basis vormt waar bovenop een ander, complexer, ontwerp kan worden getest.

Om de combinatie van achtergrond- en voorgrond-ontwerp te kunnen testen hebben we een bovenlaag ontworpen die ervoor zorgt dat de verschillende ontwerpen niet van elkaar gescheiden kunnen worden. Dit levert realistischere resultaten op.

De combinatietest resulteerde in bruikbare informatie voor ons testgeval van Xilinx. De software van

NanoXplore heeft nog enkele problemen die door tijdgebrek niet zijn opgelost of omzeild, waardoor we hiervoor geen resultaten hebben van de combinatietest.

Conclusie

Onze software maakt het overbodig om manueel vele iteraties van een ontwerp te testen in de zoektocht naar een optimum in performantie. Door de vergaande automatisering kunnen er veel meer datapunten worden berekend. Dit maakt het gemakkelijker om zowel de performantie van de FPGA te bestuderen als de performantie van (een onderdeel van) een ontwerp. Dit was het beoogde resultaat van deze thesis.

Contents

Acknowledgments

English Abstract i

Nederlandstalig Abstract iii

Korte Samenvatting v

Contents ix

List of Tables xiii

List of Figures xv

List of Symbols and Abbreviations xvii

1 Introduction 1

1.1 Industrial Background 1

1.2 Context 2

1.3 Goals 3

1.4 Thesis Outline 3

2 Literature Study 5

2.1 FPGA Fundamentals 5

2.1.1 Architecture 5

2.1.2 Design and Programming 6

2.2 Related work 7

2.3 Stamping Methodology 8

2.4 Open ESA FPGA Benchmark Suite 9

2.5 Conclusion 10

3 Methodology **11**

3.1 Overview . . . 11
3.2 Automation . . . 11
3.3 Software Architecture Overview . . . 12
3.4 Benchmark Design . . . 13
3.4.1 Benchmark IO . . . 13
3.4.2 Routing . . . 13
3.5 Output Data . . . 14
3.6 Conclusion . . . 15

4 Implementation **17**

4.1 Overview . . . 17
4.2 FPGAPerformanceSuite Python Package . . . 18
4.2.1 Task Definitions . . . 18
4.2.2 Parallelism . . . 19
4.3 Vivado . . . 19
4.3.1 Finding the Maximal Frequency in Vivado . . . 20
4.3.2 Report Output . . . 21
4.3.3 Xilinx Kintex-7 . . . 21
4.4 NanoXplore . . . 22
4.4.1 NxPython Run Script . . . 22
4.4.2 NX-MEDIUM . . . 23
4.5 Conclusion . . . 23

5 Benchmarks **25**

5.1 Naive Mathematical Operations . . . 25
5.2 FIR Filter . . . 26
5.3 ISCAS'89 . . . 26
5.4 FIR Filter With Filler . . . 28
5.5 Conclusion . . . 29

6 Results **31**

6.1 Naive Mathematical Operations . . . 31
6.1.1 Mathematical Operator Results With Kintex-7 . . . 31
6.1.2 Adder Results With NG-MEDIUM . . . 31
6.2 ISCAS'89 . . . 33
6.3 FIR Filter With Filler . . . 36

7 Conclusion **39**

References **41**

Digital Content **45**

Appendix A Code **47**

A.1 FPGAPerformanceSuite Python Package 47

A.2 Runner examples 47

A.3 Converting .bench to VHDL 48

A.4 Vivado TCL Script 50

A.5 NanoXplore Python Script 54

A.6 VHDL Top Module 55

Appendix B Miscellaneous **59**

B.1 Hierarchical Utilization Report 59

B.2 FIR Filter With Filler Graphs 61

List of Tables

4.1 Example of an oscillating result. 20

5.1 The internal structure of the ISCAS benchmarks. Every column is the count of that pin or gate type. A DFF is a D-flip-flop 27

6.1 The results for the simple adder benchmarks. 32

6.2 The results of the ISCAS benchmarks. 34

List of Figures

1.1 Margin and contingency visualized. 2

2.1 A simple CLB design. 6
2.2 A symbolic representation of FPGA fabric. 6
2.3 Stamping methodology as visualized by Xilinx (2017). 8

3.1 The program's structure. 12

4.1 The completed program structure with example tasks. 17

5.1 The top module in diagram form. 28

6.1 The results of a number of basic mathematical operations on the Kintex-7. 32
6.2 The results for the simple adder benchmarks. 32
6.3 The results for the ISCAS benchmarks. 35
6.4 The results for the FIR With Filler benchmarks as a scatter plot. 37

List of Symbols and Abbreviations

ASCI	Application-Specific Integrated Circuit
CLB	Configurable Logic Block
CPU	Central Processing Unit
DSP	Digital Signal Processor
ESA	European Space Agency
FLOPS	Floating Point Operations Per Second
FPGA	Field Programmable Gate Array
GPU	Graphics Processing Unit
GUI	Graphical User Interface
HDL	Hardware Description Language
HKMG	High-K Metal Gate
IO	Input and Output
IP	Intellectual Property
ISE	Xilinx Integrated Synthesis Environment
JSON	JavaScript Object Notation
LUT	LookUp Table
MOSFET	Metal-Oxide-Semiconductor Field-Effect Transistor
PAR	Place and Route
PCIe	Peripheral Component Interconnect Express
RAM	Random-Access Memory
TCL	Tool Command Language
TSMC	Taiwan Semiconductor Manufacturing Company
USA	United States of America
VHDL	Very High Speed Integrated Circuit HDL
Xilinx ISE	Xilinx Integrated Synthesis Environment
Yosys	Yosys Open SYnthesis Suite

Chapter 1

Introduction

In this chapter we first introduce our industry partner. Then we provide context on our research topic. Finally we state our goals and the structure of this thesis.

1.1 Industrial Background

This thesis is made in cooperation with Antwerp Space. Antwerp Space is a Belgian satellite communications company with over 70 engineers and PhDs. It was founded in 1962 as part of Bell Telephone. In 2010 it was bought by OHB SE, a European space and technology group. Since then the company goes by its current name. Their activities include commercial ground and onboard modem and RF converters, test systems and integration of onboard communication subsystems.[1]

As a company active in the aerospace industry, Antwerp Space is interested in FPGAs to use in small batch space grade equipment. Thanks to its flexibility, high achievable clock speeds, and parallel nature of its internal structure, an FPGA can process more data with lower latency than a CPU. It is the equivalent of creating dedicated hardware, but at a fraction of the cost of a custom chip. The initial cost estimate for an ASCI design is in the tens of millions of dollars.

Aerospace is a complex industry. Components like FPGAs and CPUs need to be able to handle not only factors such as extreme temperature swings but also much more radiation than is normal on the surface of Earth. Lead times for space grade components in the order of 40 weeks are no exception. This means that concurrent design of all the aspects of a system is a necessity. It requires early prediction of engineering budgets including the FPGA resources.

[1] www.antwerpspace.be

1.2 Context

The space industry requires extensive analysis of every component. Critical components -like FPGAs- are chosen with large safety margins and contingencies. The margins and contingencies are well-studied, managed and controlled. Two relevant examples from the ESA standard SRE-PA/2011.097: *"R-SW-1: Any on-board memory (Random Access Memory RAM used for code and/or data) shall include a memory margin of at least 50%."* and *"R-SW-2: Any on-board processor peak usage shall not exceed 50% of its maximum processing capability."* (ESTEC, 2012). This margin is 50% extra on top of the contingency, which is the maximal expected required amount needed. This value is in turn based on the current best estimate of the relevant resource requirement. This margin can drive up component cost considerably. It can also cascade into additional power consumption, causing those estimates, contingencies, and margins to go up as well. Figure 1.1 illustrates the definitions of margin and contingency as defined by NASA (Space Systems Engineering).

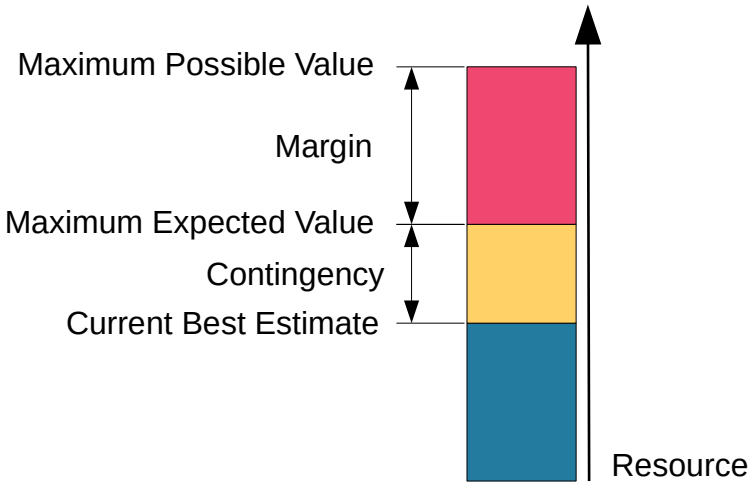


Figure 1.1: Margin and contingency visualized.

This thesis proposes that it is possible to create a better resource and performance estimation for FPGA components based on project specifications and benchmarking. The project specifications are used to select circuits to benchmark that have a similar footprint to those that will be used in the final design. The specifications also help inform the boundaries the benchmarking takes place in.

Eliminating uncertainty entirely is not possible without having a complete design. By automating the process of running many different designs, potentially on multiple target devices, a comprehensive dataset can be produced. This will help determine the best estimate, which hopefully snowballs into a larger overall margin. For reference: The current estimates made by Antwerp Space are up to 100% too high.[2]

[2]Based on internal, confidential numbers.

There are two approaches to improving these estimates: On the one hand the resource usage of the intended design can be improved. On the other hand, the performance of the target device can be better characterized. Both of these points could be addressed by customized benchmarking.

Today the primary source of information about FPGA performance are the manufacturers themselves. This information should be used with care as a conflict of interest is at play here. The published statistics will highlight the superior areas over the competition. These areas may not line up with the intended use case at all. On top of that, different manufactures do not use the same standards when it comes to publishing statistics, so figures like “maximal frequency” and “LUT count and size” are not really comparable without additional context.

1.3 Goals

This thesis focuses on researching and implementing a benchmarking based methodology that enables more accurate early estimation of required FPGA resources. This implementation should be able to compare devices from different manufacturers. Our intention is to make it easy to add to, or change the benchmarked algorithms to fit the needs of the end user to allow them to perform a detailed and targeted suite of tests.

To summarize, our main research question is: *Does a benchmarking based methodology enable an accurate and precise early estimation of required FPGA resources?*

With:

- *Accurate*: The final resource budget has to be within the boundaries of the estimate.
- *Precise*: The boundaries have to be as small as possible.
- *Early*: With only limited knowledge of the algorithm that will be implemented on the FPGA and its dimensioning characteristics.

1.4 Thesis Outline

The rest of this thesis is split into six chapters. In chapter 2 we start with FPGA fundamentals, describe related work and industry standard methods of benchmarking. In chapter 3 we describe our methodology, the software we have created for this project, what makes a good benchmark, and what output data we expect. Next, in chapter 4 we go in depth on how our software works, how it interacts with the FPGA vendor’s software, and which devices we used as test cases. Then in chapter 5 we show example benchmarks, and our solution for occupying the routing fabric. In chapter 6 we present the results of those benchmarks. Finally in chapter 7 we summarize our conclusion of this thesis.

Chapter 2

Literature Study

In this chapter we explain FPGA fundamentals, study the state-of-the-art, examine current industry methodologies for benchmarking FPGAs and the most used provider of IP cores used by those benchmarks. We also address the Open ESA FPGA Benchmark Suite.

2.1 FPGA Fundamentals

Field Programmable Gate Arrays (FPGAs) are integrated circuits based around configurable digital logic. Their configuration is loaded every time the device starts up, and can thus be reprogrammed. This is where the “Field Programmable” part of the name comes from. Historically the digital logic was an array of single purpose logic gates (hence “Gate Array”) which could be interconnected. The individual gates have been replaced by Lookup Tables (LUTs) which are more flexible. (Maxfield, 2004)

The main advantages of FPGAs are their flexibility and scalability. Their unit cost is low compared to the cost of developing even a single iteration of a custom ASCI design. This makes using them attractive in virtually every scenario where the expected sale volume is less than the order of tens of millions of devices. They an also be reconfigured later, which means that some types of bugs can be fixed without having to create a new (expensive) hardware revision.

2.1.1 Architecture

An FPGA is a matrix of Configurable Logic Blocks (CLBs). Every CLB contains at least a LUT, a flip-flop and multiplexers to configure the inputs and outputs as illustrated by figure 2.1. The LUT is used to emulate any combinatorial logic, the flip-flop is used to combine these combinatorial elements to create sequential circuits. (Keim, 2018)

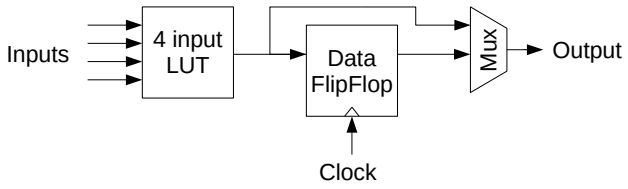


Figure 2.1: A simple CLB design.

In between the CLBs is a routing fabric that allows the different CLBs to interconnect as illustrated by figure 2.2. There are usually other blocks in the matrix as well, such as dedicated RAM blocks, DSP blocks (hardware multipliers with extra features like accumulators), IO blocks, and high speed interfaces like PCIe.

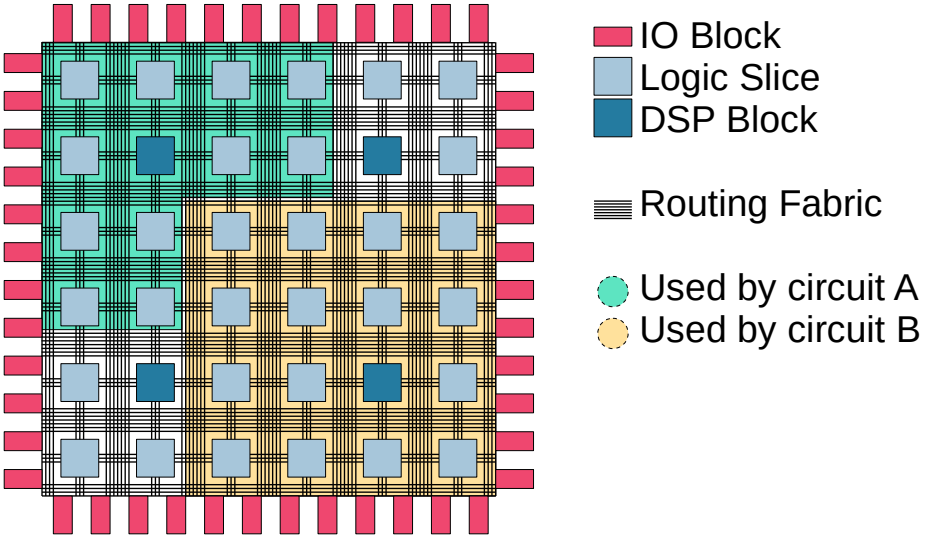


Figure 2.2: A symbolic representation of FPGA fabric.

2.1.2 Design and Programming

Programming an FPGA, or rather, creating the configuration that is loaded into the device on startup, usually involves a computer aided design tool. The user creates a design in a Hardware Description Language (HDL) such as VHDL or Verilog. From this design a netlist is generated. The netlist describes the connectivity of the fundamental building blocks of the FPGA. This process is called “synthesis”.

After synthesis the design must be mapped to the actual FPGA fabric in a process called Place And Route (PAR), eventually resulting in a configuration that can be programmed into the FPGA. Before this is done, a design is normally verified with timing analysis and simulations.

Most commercially usable (that can generate netlists for actually existing hardware, not only for academical models) synthesis and PAR software is proprietary and specific to the FPGA

manufacturer. Examples include Xilinx' Vivado[1] and Intel Quartus Prime[2]. An honorable mention goes out to the Yosys[3] project which created a completely open source toolchain that supports the Lattice iCE40 and most of the Xilinx 7-Series FPGAs.

2.2 Related work

Njuguna (2008) surveys a number of benchmarks and related studies. They focus on general purpose computing or on the performance of the FPGA toolchains. For example: the RAW Benchmark Suite (Babb et al., 1997) focuses on "comparing reconfigurable computing systems". Many of the benchmarks also focus on a single test case and are not extendable to include others. For example: the LINPAC Benchmark (Dongarra et al., 2003) tests (floating point) performance by solving systems of linear equations. The study remarks that *"Evidently, some of the benchmarks are very old relative to modern FPGA technology."*, and by now this study itself is over 10 years old. Most of them, are not relevant for this thesis. A notable exception would seem to be OpenFPGA.org. Unfortunately OpenFPGA.org no longer exists and their website is no longer available. Not even archive.org has a copy unfortunately.

Selvaraj et al. (2013) evaluates more recent tools, but only for using FPGAs to accelerate general computation, augmenting or replacing GPUs. The discussed tools for generation of HDL from normal source code are interesting in the context of this thesis. This approach would allow a multitude of tests to be written in more high level languages, or have them conditionally generate the required HDL. Spector (Gautier et al., 2016) is a high level synthesis benchmark based on OpenCL. They reuse/adapt part of the OpenDwarfs (Krommydas et al., 2016) project's benchmarks, which are also focused on parallel computing.

Vansteenkiste et al. (2015) points out that there is also a significant difference between academic and commercial results. Their conclusion is that the academic state-of-the-art is (far) behind the commercial reality, especially when comparing algorithm optimizations. This is important to consider when comparing the results from this thesis to other sources, as this thesis makes use of commercial toolchains.

Comparing devices across different technology nodes (the manufacturing process) is difficult. For MOSFET technology, there is a scaling law known as "Dennard scaling". It states that as the transistor technology get 30% smaller, their speed increases by about 40%. (Dennard et al., 1974) Scaling laws allow a rough estimation of what a speed increase could be from one device to another, but care must be taken that the architectures remain comparable. Another note is that the law was written in 1975. In the more recent past, the increase in speed has slowed down. (Bohr, 2007)

[1] `www.xilinx.com/products/design-tools/vivado.html`
[2] `www.intel.com/content/www/us/en/software/programmable/quartus-prime/overview.html`
[3] `www.clifford.at/yosys/`

Titan (Murray et al., 2013) utilize hybrid toolchains (commercial and academic) to reach more realistic results, but they only support one vendor. Industry players such as Xilinx and Intel (previously Altera) release white papers and technical reports on their benchmark methodology and results, such as Xilinx (2017), Altera (2007). Their focus is on showing a competitive edge, in both hardware and software. This is an obvious conflict of interest. We do however think their methodology is worth examining, which we do in the next section.

2.3 Stamping Methodology

Both Intel and Xilinx use the stamping methodology as described in Altera (2008) as illustrated in figure 2.3. It consists of selecting a HDL design (called cores or stamps) and instantiating it multiple times in the FPGA. This can be used to test maximal speed and maximal utilization of FPGA resources, but in reality it is more of a test of the capabilities of the design software. Since in almost all cases the use of this vendor specific design software is mandatory, it is considered part of this methodology.

Shift registers are added between the IO signals of the cores and the IO pins of the FPGA. This prevents the design from being optimized away by the synthesis tools. Caution must be exercised to make sure that the IO wrapping is not the critical path and does not add too much overhead. This would defeat the purpose of testing the different designs. To avoid this, the wrapping logic and cores run asynchronously on separate clocks.

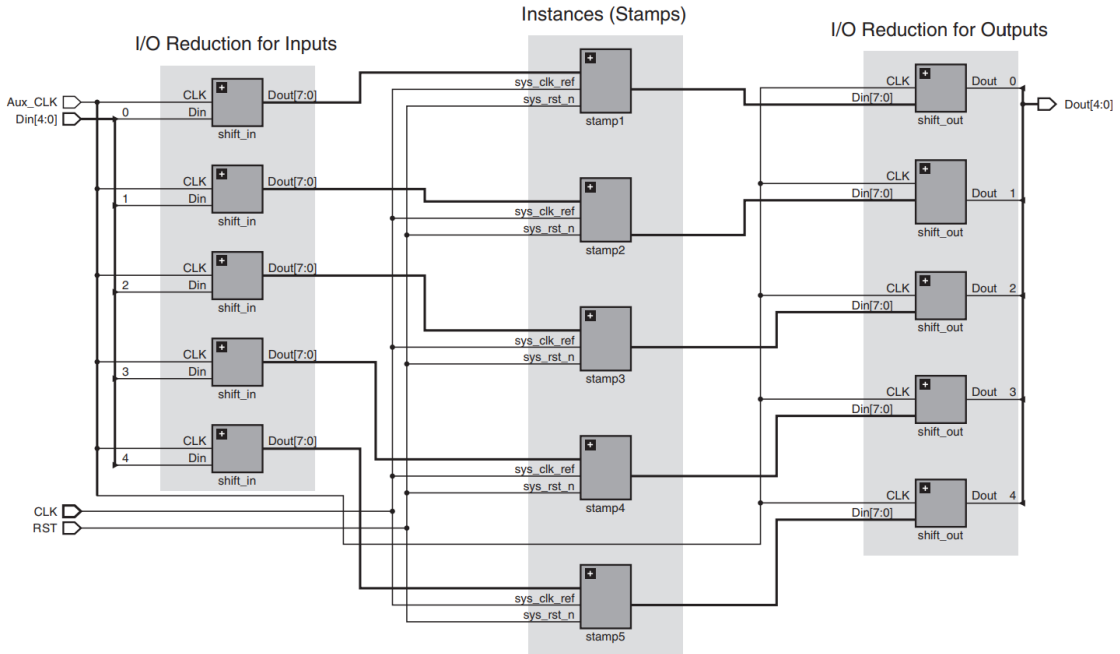


Figure 2.3: Stamping methodology as visualized by Xilinx (2017).

In most published results, the FPGA is filled to the point where the software can no longer place and route all the signals. The utilization is then measured, along with the maximal frequency. Although we think this approach may be useful to compare the efficiency of the design software, we don't think it is a good way to compare different architectures or devices. It does not provide much insight into the actual limiting factors of a device.

The design chosen as stamp is very important. A design can, due to programmer choice or pure coincidence, be more suited for one architecture or optimization algorithm over another. This is why most published benchmark results will include a set of designs. Those designs must be sourced from somewhere of which the most popular freely available source is OpenCores.

OpenCores is a website and organization which "is a community portal for professionals, amateurs, and enthusiasts interested in the field of digital design engineering. The site gives users open access to view, download, reuse, and share gateware designs. OpenCores specializes on bundles of structured files forming self-confined units, most commonly known as IP cores, coded in HDL".[4]

Both Intel and Xilinx use designs from OpenCores in their benchmarks, as well as private designs. Those private designs are often supplied by their customers and cannot be shared publicly. This makes their results of limited value since they cannot be reproduced independently. (Altera, 2008)

The downside of stamping is that it does not often match realistic use an FPGA. A real design will never attempt to make use of all available resources, since the design software will most likely not be able to compute the placement and routing. Stamping does give a good rough indication for maximal usable capacity that can be compared with other devices or vendors, if the same parameters are used. It cannot be used to find the optimal use case for one device since it offers no detailed information. Another danger is that the intended use case might not align well with chosen designs.

2.4 Open ESA FPGA Benchmark Suite

While we were implementing our already designed methodology, we found that the European Space Agency (ESA) had made an effort to create a standardized benchmark suite, not only for ESA projects but for the FPGA industry as a whole. The project is called "Open ESA FPGA Benchmark Suite" (Lange, 2017)[5]. The suite is designed to enable the public sharing of configuration and benchmark data by letting anyone create additional benchmarks and publishing the results.

The currently provided benchmark circuits are a set of small circuits which address only one particular architectural feature (such as LUTs, DSP blocks, memory blocks, . . .) and more complex circuits which represent arithmetic functions (such as a moving average filter). Design software support is limited to Xilinx ISE (deprecated since 2012) and Microsemi Libero, with upcoming support for

[4] opencores.org/about/mission

[5] Source available online: gitrepos.estec.esa.int/FPGA/open-ESA-FPGA-benchmark-suite

NanoXplore's synthesis tool NanoXmap being mentioned as in progress (August 2017). Operating system support is limited to Linux. The tool is not capable of parsing the output of the design software to produce a coherent report.

While researching this software, we found that it is not user friendly. In our opinion it is too complex to use and configure. It requires 5 languages (JSON, TCL, Python, VHDL, and Bash or C-shell) and quite a bit of work to add any new device, vendor or benchmark. Some parts of the software are hardcoded to the environment of the original author, complicating the initial setup on our computer. The software flow also seems to be designed around ISE with little regard given to other vendor's software. This is particularly noticeable in the use of TCL scripts to create projects and control various bits of the synthesis process.

2.5 Conclusion

Only the Open ESA FPGA Benchmark Suite is similar to what we have in mind with this thesis. It is unfortunate that we only discovered it after starting work on the implementation of our methodology, as it would have provided a nice starting point. It is unfinished and appears to be abandoned, no commits have been made to the code since May 2017. The experienced difficulty in setting up the software, combined with the status of our own implementation at the time we learned of its existence made us decide against using it any further.

A lot of time was spent trying to find more prior research, but this was the most relative selection of works. Various other studies do use benchmarking but they have other goals. The results of these studies cannot be used to compare various FPGAs or multiple vendors, or are only targeted to academic toolchains.

It is our belief that this lack of relevant published prior works is due to the industrial nature of this thesis. It is a niche, closed industry. We speculate that if this kind of work is done in companies, their methodology is kept secret. Another possibility is that many choose a product family based on experience or with help from a manufacturer's data sheet. Then development can happen on the largest device in that family and when the design is (almost) finished, a more appropriately sized FPGA is chosen. In the space industry this is rarely done given the inherent cost of swapping components. This leads to a situation where the biggest possible FPGA is used, which benefits manufacturers, making them unlikely to try and improve this situation.

Chapter 3

Methodology

This chapter explains why we created custom software with top-down overview of what it does and how it works. We will explain what makes a good benchmark circuit. Lastly, we state our expectations of the output data, our representation of it and how to analyze it.

3.1 Overview

The ultimate goal of this thesis is to develop a framework which enables a hardware design engineer to easily determine the most optimal FPGA for a specific project. Currently, the selection of the FPGA device is often determined based on the experience of the design engineer and the manufacturer's datasheets -which are often biased- rather then quantitative measures. Therefore, in this thesis we aim to develop a methodology which is able to automatically indicate which FPGA device is most suited for a specific project.

To achieve this, we developed an FPGA performance suite which is able to test the performance of devices from multiple vendors given different input parameters and specific VHDL input designs. The primary output statistic our system generates is the maximum clock speed a design can run at, although the end user can use another statistic. These output statistics allow the user to select the most suited FPGA based on quantitative figures. The design in question is determined by the end user, so that it can be similar to the final design that will be implemented in the project. It makes little sense to use a memory intensive design for a project that will mostly use DSPs for example.

3.2 Automation

It is our belief that the key to accurate resource estimations is to have sufficient access to relevant data. The data currently presented by manufactures is one-sided, it highlights their own advantage

and omits potential downsides. With in-house generated data, all aspects of the data are visible and specific tests to find weaknesses are possible.

To make it easy to generate this data we are creating a program that can automatically run a suite of benchmarks that are highly customizable. The program will be able to generate statistics such as the maximal frequency or the resource utilization in number of FPGA primitives (fundamental building blocks).

To facilitate variance of components, such as the bit width of a signal input to a filter, parameters (also called generics) are possible in the top level of a VHDL design file. The user must be able to specify these with great flexibility. This limits the amount of work needed to generate the required input files. For example: With a fully generic FIR filter implementation in VHDL, it would be possible to test everything from a small 7 tap filter with 8 bit data and 8 bit coefficients, to a large 129 tap filter with 32 bit data and 24 bit coefficients, and everything in between.

3.3 Software Architecture Overview

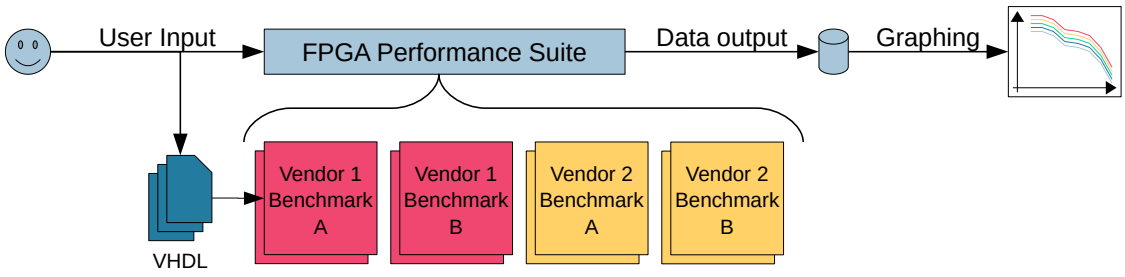


Figure 3.1: The program's structure.

Diagram figure 3.1 illustrates the basic working principle of our software. We'll be building on this diagram to explain elements of the software throughout this thesis. A complete version of the diagram with more details about the implementation is used in section 4.1.

The main program ("FPGA Performance Suite" in the diagram) is tasked with running the benchmarks as specified by the user input and collecting the output data. Every one of those benchmark tasks is the combination of a specific vendor, a set of input parameters, and a circuit design described by a set of input VHDL files.

To compute the results of a benchmark, our software must be capable of running the FPGA's synthesis software with a set of input parameters and extract all useful information from its output. Because running large benchmarks can take a long time, it is best if they can be run in parallel. Sections 4.3 and 4.4 go into detail on the specifics of individual vendor software.

3.4 Benchmark Design

Every benchmark is a combination of VHDL code and a set of input parameters. The code defines the circuit design under test. It can be a file chosen from a preexisting set or it can be created by the engineer specifically for a project. By using the input parameters we can create a great variety of designs with a small set of files.

The intended project of the end user determines what benchmarks will yield relevant data. This means it is not possible to create a generic set of VHDL design files to use. Such generalization would make the data less relevant.

We did not focus on creating a set of benchmarks, since they can be created or sourced elsewhere. Instead we focused on making it easy for the end user to integrate those files. We will however go into more detail on what makes a good benchmark and what needs to be avoided.

3.4.1 Benchmark IO

When creating or selecting the file to be benchmarked, it is important to make sure that the design is not accidentally constrained by its IO. If the data path includes IO pins, the maximum speed can be limited by the speed of the IO blocks instead of the intended benchmark.

Even if the speed of the IO blocks is not considered an issue, they should be avoided because they are placed on the edge of the routing fabric. This creates a situation where the routing is forced to place the design on the edge of the routing area or use long signal paths to connect the design to the IO blocks.

This situation can be prevented by using an internal register as input and output. The synthesis software must then be instructed to preserve those registers, since it will attempt to remove any part of the design it thinks is useless. The implementation of this is vendor specific, since there is no standardization for this in VHDL.

3.4.2 Routing

Figure 2.2 also illustrates that the routing fabric is an important resource to consider when benchmarking an FPGA. Experience shows that routing is often a limiting factor that is hard to predict. Comparing countable quantities like the number of LUTs or DSP blocks used by a particular design is easier than comparing how much of the routing infrastructure it uses. The routing fabric also varies greatly by FPGA architecture, some have better routing infrastructure than others.

With small designs the routing infrastructure is not stressed. There is enough space on the routing fabric for the PAR algorithm to optimize for speed. A larger design -one that takes a significant percentage of the chip- cannot be treated in the same way. Larger designs will therefore have longer

propagation delays. We propose to use a known background filler pattern to take up space on the chip i.e. loading the routing resources, then add the benchmark design. This forces the synthesis software to compete with a known quantity of used resources.

To estimate how much background filler is needed, the possible filler patterns and the actual benchmark design are run trough the same benchmark process separately. This results in a dataset with the LUT utilization and maximal clock frequency. A pattern can be chosen so that it does not interfere with the expected maximal clock frequency of the actual benchmark design. If the filler's maximal clock speed is lower than the actual design, it would drag the total clock down along with it.

The quantity of LUTs used is known per instance of a design. This allows the user to estimate how many times a pattern must be placed to reach the desired level of global utilization. This filler quantity can be subtracted from the output data to get an estimate of the utilization by the actual design. Some software keeps detailed records of the resources used per hierarchical part of a design, which can be used to verify those estimates.

The background filler should be architecture neutral i.e. be made of only the lowest level gates and flip-flops. It should be flexible enough to allow its use in many different benchmarks, so that the output data can be compared more directly. We opted for the ISCAS'89 (Brglez et al., 1989) set. It contains 31 circuits with varying size and complexity. The circuits are mostly combinatorial logic with a few D flip-flops. This makes it great background filler because it takes up LUTs but no special blocks such as DSPs. More details in section 5.3.

One potential issue is that the PAR algorithms used by vendors are generally capable of determining data dependencies. The software will attempt to segregate the benchmark into multiple sections if there is no link between them, as illustrated by the background colors in figure 2.2. This negates the benefit of using the background filler pattern. To combat this, section 5.4 implements a top level circuit design that links the different parts of the benchmark together.

3.5 Output Data

Generating a readable output format is important, and may appear simple at first but it must not be overlooked. A graph is great for one or two-dimensional inputs. After that the best visualization depends largely on what those input dimensions represent. Because of the input flexibility, that is hard to predetermine. Our program will output the raw data it gathers, in a format that can be easily read by the end user or by other programs to assist in visualizing the results.

One potential issue is the complexity in comparing the output data. While maximal speed (expressed as a clock frequency) is a number comparable across all devices, the primitives typically differ per FPGA family or architecture. Although they generally at least contain the number of LUTs, Block

RAMs, and DSP blocks. Care must be taken when comparing the raw data. The architecture of the primitives may not be the same, leading to incomparable data.

The number of used primitives is more useful as output when used as a fraction of the number of primitives available, instead of as an absolute number. This is called the “utilization rate”. This metric is used by manufacturers in papers and published statistics, which makes it a good measure to keep in mind when comparing results from our benchmarks to those found in other sources.

Since the LUT is the most basic and fundamental building block, we will use the LUT utilization rate this as *the* utilization rate for the entire device. This is a generalization out of necessity. It is also easier to manually calculate (or estimate) a boundary on number of DSPs or RAM blocks required, at least if the project specifications are known.

Ideally we could calculate a number that quantifies the computational power an FPGA has, much like “floating point operations per second” (FLOPS) is often used for CPUs. Such a number would have to be specialized to fit the project the FPGA will be used in. For example if our design is an FIR filter, the data throughput would be a good single metric. Though an issue immediately arises: FPGAs allow almost limitless pipelining increasing throughput, but this also increases latency, which might not be acceptable in the telecommunication industry.

Generalizing this is not easy but if we would have to define a set of basic metrics a few examples could be:

- number of flip flop operations per second
- number of N-bit wide multiplications per second
- usable memory at a minimum access speed

All these metrics require knowing the maximum usable frequency at a given fill rate. So these two will be used as primary output results for our visualizations by creating a “clock speed” versus “device fill rate” graph for a number of variations on the input parameters.

3.6 Conclusion

Our software must be automated to allow the easy generation of enough data to be useful, including automatically running the synthesis software and parsing its output data and creating a flexible and easy to use representation of that data.

The benchmarked circuits must be selected with an eye on the eventual project where the FPGA will be used, since otherwise the data will not be more relevant than data provided by the manufacturer. The circuits should not use IO pins, and ideally stress the PAR algorithms in the synthesis software by occupying enough of the routing fabric of the FPGA.

Chapter 4

Implementation

In this chapter, we elaborate our custom software, and how it interacts with the vendor's software. Then, we clarify our test component choices for each of the vendors

4.1 Overview

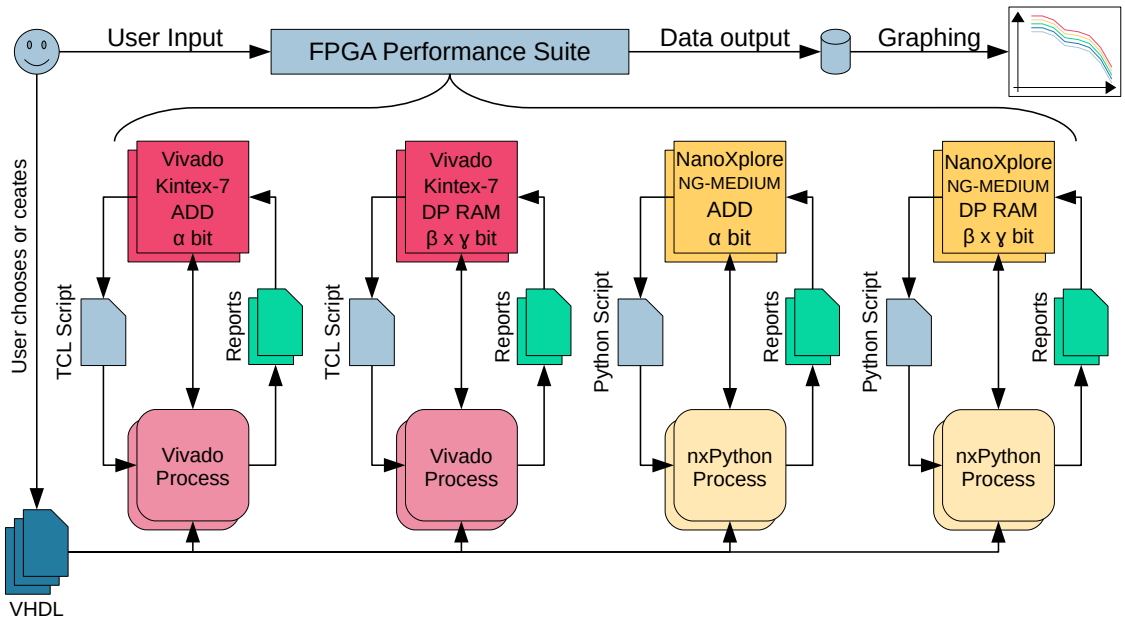


Figure 4.1: The completed program structure with example tasks.

Figure 4.1 illustrates our custom program's architecture and describes an example case where an adder ("ADD") and Dual Port RAM ("DP-RAM") are benchmarked on a device from Xilinx (Kintex-7) with the Vivado software and a device from NanoXpore (NX-MEDIUM) with it's software nxPython. The adder has a variable bit width α. The Dual Port RAM has variable depth β and word width γ.

4.2 FPGAPerformanceSuite Python Package

Our main custom program is the `FPGAPerformanceSuite` Python package. We chose Python 3 for its ease of use and flexibility. Our programming environment of choice is PyCharm[1] but the end user is free to chose a different programming environment. We follow recommended Python programming practices such as PEP 8[2]. We advice using a virtual environment, in accordance with PEP 405[3]. The module can be run like any other Python module, by running `python -m FPGAPerformanceSuite` from the command line. The source code for this package is included in appendix A.1

The package defines the vendor classes for Vivado and NanoXplore and several helper functions that are used to automate the process of defining the set of benchmarks to run. To keep everything flexible and customizable, all the configuration is done in Python. There are two options for defining a benchmark set: editing the `FPGAPerformanceSuite` package entry point (`__main__.py`) and using its infrastructure; or importing `FPGAPerformanceSuite` in a custom script, defining benchmarks and calling the `FPGAPerformanceSuite.run` function. The first option is the easiest because several helper functions are already defined, including a main function with argument parsing that supports setting the various options.

4.2.1 Task Definitions

Every benchmark task is the combination of a specific vendor and a set of input parameters. The input parameters needed can differ per vendor. Every task will start a new instance of the vendor software, it will instruct the software to run with the set of input parameters and collect the results when it is finished.

The code snippet below is an example of what a task set definition looks like. This definition will result in a set of 6 tasks, all using the same file but with a different generic parameter N. This is equivalent to the fist benchmark in figure 4.1, where α is filled in with the values 8, 16, 32, 64, 128, and 256.

```
1 # Define a runner for Vivado, that uses the shared thread pool
2 @runner(Vivado, single_thread=False)
3 def vivado_add_runner():
4     part = 'xc7k410tfbg900-1'  # The Vivado part number
5     files = ['../vhdl/add.vhd']  # The VHDL files required
6     for n in (8, 16, 32, 64, 128, 256):  # Input parameter N
7         name = 'add%d' % n  # Create a unique name
8         yield dict(name=name, part=part, files=files, generics={'N': n})
```

Every function with the `@runner` decorator is called a runner. Runners are automatically detected by the function `FPGAPerformanceSuite.run`. The decorator requires a reference to the vendor class. A single runner cannot define tasks for multiple vendors.

[1] `www.jetbrains.com/pycharm/`
[2] Style Guide for Python Code: `www.python.org/dev/peps/pep-0008`
[3] Python Virtual Environments: `www.python.org/dev/peps/pep-0405`

Runners are normal Python generators, this means that the `yield` keyword is used to return multiple sets of input parameters from a single function. It also makes them very powerful, since they can contain any arbitrary Python code. More complex examples of runner definitions are included in appendix A.2.

The generator must be finite due to a limitation of our current implementation. All of the inputs are computed before the worker threads start. So if the generator were infinite, the program would eventually run out of memory and (hopefully) get killed by the operating system.

4.2.2 Parallelism

To optimize for runtime the program will be multi-threaded[4] so more than one benchmark can be run at the same time. This is important because they can take quite a long time (more than an hour) per set of input parameters.

A runner can optionally set the parameter `single_thread` to indicate that the vendor software cannot be run in multi-threaded mode. This option was included because of license limitations in NanoXplore's software. Only one instance of the program can be run at the same time. Vivado only allows many parallel instances on the same computer with a single license.

The main thread ("FPGA Performance Suite" in figure 4.1) will set up a batch of all of the benchmark tasks that need to run based on the user's input. It will then dispense them to worker threads when they become available. This would in principle allow the benchmarks to be run on other host machines or computation clusters, but this is not implemented here. After a worker is finished with a task, it will pass back the results. The main thread then saves them and can create or update a visualization.

4.3 Vivado

Our first vendor software is Xilinx's Vivado. We chose to start with Xilinx because it is one of the biggest FPGA vendors, with a market share of ~50%[5]. We opted for Vivado instead of Xilinx ISE because the latter has been deprecated since 2013[6]. We use the HLx edition version 2017.1.

Vivado can be used as a GUI program or via TCL[7] in scripted or interpreted mode. The TCL batch mode makes it easier to setup automation, so that is what we used. We will be using the non-project design flow. It is easier to use with the TCL automation, but since the project files are not stored to disk, the project cannot be opened in the GUI version of the program.

[4] Technically we don't use multi-threading but multi-processing. This is an important distinction in Python, but it is outside the scope of this thesis. The end result is the same: multiple tasks can be run at the same time.

[5] According to `hardwarebee.com/list-fpga-companies/`

[6] `www.xilinx.com/support/download.html`

[7] Scripting language. Pronounced "tickle", more information on the website: `www.tcl-lang.org/`

Vivado's way to prevent signals from being optimized away is setting the `dont_touch` attribute on them. This is done in all benchmarks that don't use IO pins on their input and output registers. To prevent warnings about undriven inputs, the input can be looped back to itself with an inverter. This also prevents possible "always zero" optimizations further in the design.

4.3.1 Finding the Maximal Frequency in Vivado

The Vivado PAR process is constraint driven, it cannot operate without setting a target clock frequency. Because of this, benchmarking a design is not as simple as running synthesis and PAR once. After synthesis, a clock constraint must be added that is strict enough so the PAR process will attempt to optimize the layout, like it would in a real design. The goal is to find the maximum frequency after all.

If the clock frequency is too strict, the PAR process will fail and produce a path with negative slack. The negative slack is a good indication of how much extra time is required on the clock period to get PAR to succeed. However it is not the definitive answer because given a different set of constraints, PAR will make different decisions and produce a (usually) better result.

This trade-off is inherent to the job of the PAR algorithm. If a lower clock speed is required, it must pack the logic close together, but heavily packed logic runs the risk of running out of routing resources. This is a complex optimization problem with a different, proprietary, solution from every manufacturer.

The obvious downside of this approach is that it requires multiple passes and thus is slow. We feel this approach is justified against the potentially high error that would go unnoticed by running once and then taking the slack as truth.

To get the best result, the slack is subtracted from the previously used period. This new value is used as the period for a new PAR run. This iterative loop can be repeated until the slack is zero or within an acceptable margin of error. Sometimes the algorithms will not reach this margin, but start oscillating instead. An example in table 4.1.

Table 4.1 Example of an oscillating result.

Run	Period	Slack
1	1 ns	-10 ns
2	11 ns	2 ns
3	9 ns	-2 ns
4	11 ns	2 ns
5	9 ns	-2 ns
6	11 ns	2 ns

Vivado's algorithms are deterministic. This means that any oscillating run can be terminated, since it will not produce productive output. The best course of action in this case seems to mark the results as an oscillation and either dismiss the result or use the worst case maximum frequency. We chose to use the worse case frequency in our implementation.

This entire process is done inside a single Vivado batch run, by using a complex custom TCL script that does the iterative loop until it either reaches a result within a given margin of error, produces an error, runs out of iterations or detects an oscillation.

The source code of the script is included in appendix A.4. The default values for the parameters have been chosen based on experimentation. A starting period of 1ns, an error tolerance of 5% and an iteration limit of 10 were chosen because they produce good results. The iteration limit is almost never reached, thanks to the error tolerance and oscillation detection.

4.3.2 Report Output

Our TCL script instructs Vivado to output reports once it is finished. Those reports contain all of the output data we need. The reports are text files with ASCII tables such as these:

Site Type	Used	Fixed	Available	Util%
Slice LUTs	14254	0	254200	5.61
LUT as Logic	14189	0	254200	5.58
LUT as Memory	65	0	90600	0.07
LUT as Distributed RAM	0	0		
LUT as Shift Register	65	0		
Slice Registers	20750	0	508400	4.08
Register as Flip Flop	20750	0	508400	4.08
Register as Latch	0	0	508400	0.00
F7 Muxes	0	0	127100	0.00
F8 Muxes	0	0	63550	0.00

Our Python module parses the tables and processes the data into a uniform format. The reports also contain information that is not used by the program, but that can be viewed later if the benchmark results do not conform with expectations. The hierarchical utilization report is particularly useful because it shows the resource utilization for every part of a synthesized design. An example and source for the table above is included in appendix B.1.

4.3.3 Xilinx Kintex-7

Although no actual hardware is required to run these benchmarks, we must choose a part number for Vivado to target. We chose the Kintex-7 `XC7K410T` at the request of Antwerp Space. This is considered our primary target, and was used for most of the development. This is a short summary of the specifications of this device: (Xilinx, 2018)

- 28nm HKMG process technology by TSMC
- Xilinx part number: `XC7K410T`
- Vivado part number: `xc7k410tfbg900-1`
- Speed grade -1
- 1 540 DSP blocks
- 254 200 LUTs (6 input)
- 508 400 registers
- 28 620 KBit Block RAM

4.4 NanoXplore

NanoXplore is a relatively new French manufacturer (formed in 2010). It is also a fabless[8] company. Their effort is supported through European funding with the express purpose of being independent from the USA. They develop embedded FPGA cores and “radiation hardened by design” FPGAs for the space industry. Their focus on space grade hardware is why they are of particular interest to Antwerp Space, and why we chose them as our second manufacturer.

Unfortunately their software is still in a pre-release state and is sometimes unstable. This came to light very late in the process of creating the test results for this thesis. The early tests ran fine, and we output data for them, but the fully integrated test (FIR Filter With Filler) could not be completed. We think this is because of limitations with their VHDL synthesizer, but the software does not output any useful error messages, so it is hard to tell.

Another unfortunate observation is that the licensing model for the NanoXplore software does not allow multiple instances of the software to be run at the same on the same host. Vivado takes a single license for an entire host, meaning that it can be started multiple times in parallel. This is not the case of NanoXplore, limiting its benchmarking speed significantly.

Most datasheets of NanoXplore parts and documentation of the software are confidential. All information in this thesis is from a publicly available presentation (NanoXplore, 2018).

4.4.1 NxPython Run Script

The NanoXplore software for synthesis, placing, and routing is based on a special Python environment called “nxpython”. Unfortunately this is based on Python 2.5 and requires specially compiled modules that cannot be loaded in a newer version of Python. This meant it was easier to just start up a new process with the correct environment and load a small custom script with the input parameters passed via command arguments. This ended up being very similar to how Vivado is launched.

[8] Fabless means that the design is done in-house, but the fabrication of the devices is outsourced.

One big difference from Vivado is that the PAR algorithm is not driven by timing constraints. The software can be run without a clock speed given and will output the maximal frequency the design can be run at reliably. This makes the run script a lot less complex. The source code is included in appendix A.5.

The output format is similar to that of Vivado, it is also parsed and transformed into the same uniform format. This software provides less output data and does not have a similar hierarchical utilization output.

4.4.2 NX-MEDIUM

NanoXplore only has one part that is currently fully supported, called “NX-MEDIUM”. Compared to the Kintex-7 it has far fewer resources. Here is a short summary: (NanoXplore, 2018)

- 65nm C65-SPACE process technology by STMicroelectronics
- 112 DSP blocks
- 34 272 LUTs (4 input)
- 32 256 registers
- 2 688 KBit Block RAM

4.5 Conclusion

In this chapter we discussed the implementation details of our benchmark methodology. Our system nearly automates the entire process of generating and running the different benchmarks. Due to the flexibility of our system it should be easy for anyone with knowledge of Python to extend our module and add any vendor toolchain. For the end user very limited knowledge of Python is required to add custom benchmark sets.

Vivado was a good choice for the initial implementation. NanoXplore proved to be a difficult vendor to work with because of the pre-release state of their software, limited documentation, and almost non-existent error feedback. Hopefully this improves in the future.

Chapter 5

Benchmarks

In this chapter we present example benchmarks and our solution for occupying the routing fabric. Most of these benchmarks where used to generate test results presented in the next chapter.

5.1 Naive Mathematical Operations

For our initial testing phase, we used simple mathematical operations with a variable bit width input. To limit complexity, we used the same width input N for both operands for most operations.

We tried all these mathematical operations, natively supported by VHDL, with both signed and unsigned numbers:

- Add and subtract ($N \times N \mapsto N$)
- Multiply ($N \times N \mapsto 2N$)
- Divide and remainder (and modulus) ($N \times N \mapsto N$)
- Greater or less than with “or equal” variants, equal and not equal ($N \times N \mapsto 1$)
- Shift and rotate, left and right. ($N \times log_2 N \mapsto N$)

From our initial test runs we learned that only the following operations make sense to run, because the output data from the others is identical[1]:

- Add and subtract
- Multiply (implemented with DSP slices automatically)
- Divide
- Greater than and equal
- Shift left and right and one rotate (we chose left).
- Signed vs unsigned makes only a very small difference.

[1] This is logical given how the operations are implemented in hardware, but this is beyond the scope of this thesis.

No extra directives where added. We ran this test with a very large range for N for the Kintex-7 to test our initial implementation. The actual test results are not of much use due to their limited relevance to real world applications. Out of curiosity we did run simple adder for both the Kintex-7 and NX-MEDIUM with a more limited range for N.

5.2 FIR Filter

For a more realistic example, we chose an FIR filter provided by Antwerp Space. It is a pipelined symmetrical design created by Rondelez (2017). The design is entirely parametrized via generics, and can thus be used with a wide range of input parameters. The exact implementation is not important for our use, we regard it as a black box.

For our benchmarks of the Kintex-7 part, we chose to fix the parameters at 129 taps and 16 bits of data and coefficients. This is an arbitrary choice, but we wanted to make the design large enough to get more varied data without having to instantiate the design many times. These parameters result in 2.3% LUT utilization, and uses 65 DSP blocks. Even without knowing the inner workings of the FIR filter, 65 DSPs would make sense for a symmetrical filter.

5.3 ISCAS'89

The ISCAS'89 benchmark set, as described in Brglez et al. (1989), contains 31 circuits. The source code of the benchmarks can be found online[2]. Table 5.1 gives an overview of the files we used. The complexity increases the further down the table. The number in the name represents the number of interconnect lines among the gates. Some benchmarks have multiple versions (a letter is added to the name). For more information we refer to the source material.

The ISCAS'89 files are provided in a `.bench` file format. We did not find specifications on this file format and it cannot be synthesized by Vivado or NanoXplore's software. We analyzed the format and wrote a conversion script to create VHDL file from them. It can handle almost all of the cases. The source for this script is provided in appendix A.3, along with an example input and output. The entire set of files, including VHDL version, is included in the digital appendix.

This set will be primarily used as background filler for the FIR Filter With Filler test from the next section. It is also used on it's own as a simple testing case.

[2]Originally available at `www.cbl.ncsu.edu/benchmarks` but this site now links to an empty directory. We used this archived version: `www.pld.ttu.ee/~maksim/benchmarks/`.

Table 5.1 The internal structure of the ISCAS benchmarks. Every column is the count of that pin or gate type. A DFF is a D-flip-flop

Name	Inputs	Outputs	Gates	DFFs	NOTs	ANDs	NANDs	ORs	NORs
s27	5	1	13	3	2	1	1	2	4
s27a	5	1	13	3	2	1	1	2	4
s208	12	2	104	8	35	17	19	4	21
s298	4	6	133	14	44	31	9	16	19
s344	10	11	175	15	59	44	18	9	30
s349	10	11	176	15	57	44	19	10	31
s382	4	6	179	21	59	11	30	24	34
s386	8	7	165	6	41	83	0	35	0
s386a	8	7	165	6	41	83	0	35	0
s400	4	6	183	21	56	11	36	25	34
s420	20	2	212	16	74	28	46	9	39
s444	4	6	202	21	62	13	58	14	34
s510	20	7	217	6	32	34	61	29	55
s526	4	6	214	21	52	56	22	28	35
s526a	4	6	215	21	54	55	22	28	35
s641	36	24	398	19	272	90	4	13	0
s713	36	23	412	19	254	94	28	17	0
s820	19	19	294	5	33	76	54	60	66
s832	19	19	292	5	25	78	54	64	66
s838	36	2	422	32	149	58	89	16	78
s953	17	23	424	29	84	49	114	36	112
s1196	15	14	547	18	141	118	119	101	50
s1196a	15	14	547	18	141	118	119	101	50
s1196b	15	14	547	18	141	118	119	101	50
s1238	15	14	526	18	80	134	125	112	57
s1238a	15	14	526	18	80	134	125	112	57
s1423	18	5	731	74	167	197	64	137	92
s1488	9	19	659	6	103	350	0	200	0
s1494	9	19	653	6	89	354	0	204	0
s5378	36	49	2958	179	1775	0	0	239	765
s9234	37	39	5808	211	3570	955	528	431	113
s13207	63	152	8589	638	5378	1114	849	512	98
s15850	78	150	10306	534	6324	1619	968	710	151
s35932	36	320	17793	1728	3861	4032	7020	1152	0
s38417	29	106	23815	1636	13470	4154	2050	226	2279
s38584	39	304	20679	1426	7805	5516	2126	2621	1185

5.4 FIR Filter With Filler

As described in section 3.4.2 we need a way to make sure the synthesis software cannot segregate the design into multiple sections without any interconnecting signals if we want to use a filler pattern to occupy some of the routing network. To do this we created a custom VHDL top module. The full code is included in appendix A.6.

The different sub-sections of the design (called circuit A and B) are combined by linking their inputs and outputs to a large data chain. By convention circuit A is the actual design we want to benchmark, circuit B is the filler pattern. The clock drives the data in the chain forwards. In between every segment, the data is shifted over by 1 signal (bit) to avoid segregation. At the end of the chain all of the signals are inverted. This avoids the possibility that the synthesis software will conclude that parts of the circuit are always zero. The chain sections of the top file are also marked to prevent removal by the software.

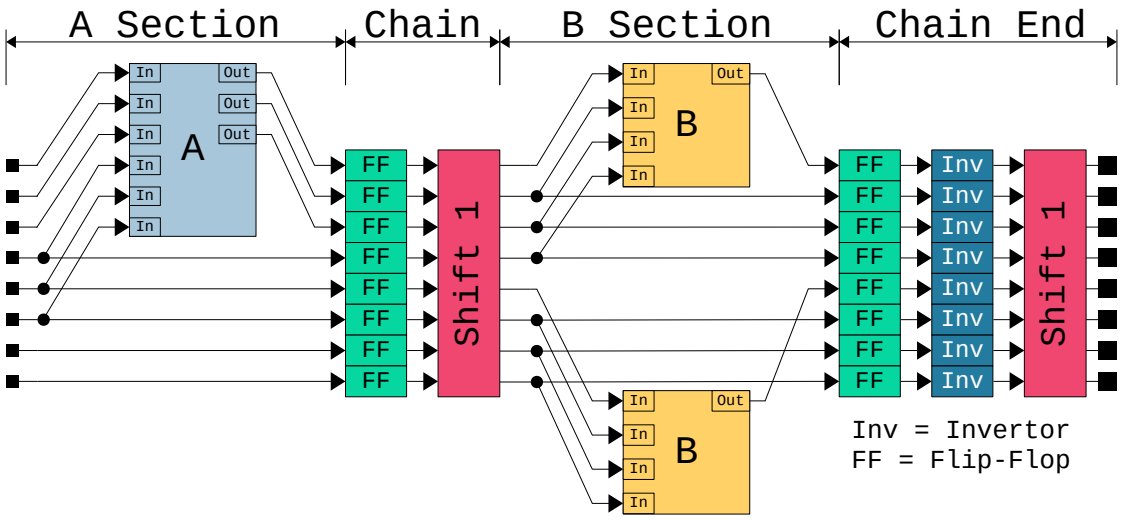


Figure 5.1: The top module in diagram form.

The way this module works is illustrated by figure 5.1. In this diagram, circuit A has 6 inputs and 3 outputs, B has 4 inputs and 1 output. This means that circuit B can be placed twice in a single segment if the bus width is set to 8. Circuit A then needs 2 bypass signals. Since in both circuits the number of inputs does not match the number of outputs, some inputs are also wired to the output, effectively bypassing the circuits. The number of instances for circuit A and B are passed into the module via a generic, meaning this can be set in the Python code.

This code is not fully generic because of the limitations of the language. It is made to be as generic as possible with only 3 sections requiring manual edits before it can be used with a custom benchmark and background filler. The component definitions for circuit A and B have to be filled in and the instantiation (linking of the chain data signals to the A and B circuit) have to be defined by hand.

Everything else (including the bypass signals) is fully automatic.

The overhead from this top module is limited. We know this thanks to the hierarchical utilization report from Vivado. It mainly consists of flip flops and a few LUTs. For the smallest version we tested (with a single the FIR filter and 10 instances of the ISCAS `s832` filler) it uses only 22 LUTs and 1037 flip-flops. For one of the largest (24 $\times$ FIR and 360 $\times$ `s832`) it uses 91 LUTs and 8808 flip-flops.

5.5 Conclusion

This set of benchmarks is a good starting point, but as mentioned before, it is not meant to be an authoritative set. The end user is expected to create their own benchmarks that are based on whatever project the FPGA is expected to used in.

Our custom solution for adding load on the routing fabric is unfortunately not completely generic, but it should be easy to use for anyone with knowledge of VHDL. We will be using it for our largest benchmark set.

Chapter 6

Results

In this chapter we present all results, remarks, and conclusions from the benchmarks we ran.

6.1 Naive Mathematical Operations

This data set is the first one we created. There is no LUT utilization data. It was computationally expensive to create (it took 2 weeks of CPU power on a 10 core server) and the results are of limited value. Because of this we did not re-run it with our more up-to-date software or for the NG-MEDIUM. The entire dataset is too large to include here, it is included digitally.

6.1.1 Mathematical Operator Results With Kintex-7

As stated above this data was only generated for the Kintex-7. Some data series have almost the same output, for example all signed and unsigned versions of the same operation. To make the graph readable they have been removed from figure 6.1.

6.1.2 Adder Results With NG-MEDIUM

While developing our implementation for NanoXplore, we did run a much shorter dataset with only an 8, 16, 32, 64, 128, and 256 bit adder. The data only includes the maximum clock speed because the LUT utilization is so low it does not provide any useful information. Results in figure 6.2 and table 6.1.

We did not expect the NG-MEDIUM to overtake the Kintex-7, and are also surprised that the NG-MEDIUM has a relatively flat curve for the first four results. The Kintex-7's curve does conform to the shape we where expecting: gradually sloping downwards, with the maximal clock roughly halving per doubling of the input size. This is what could be expected from an optimized, but not pipelined, adder design in hardware.

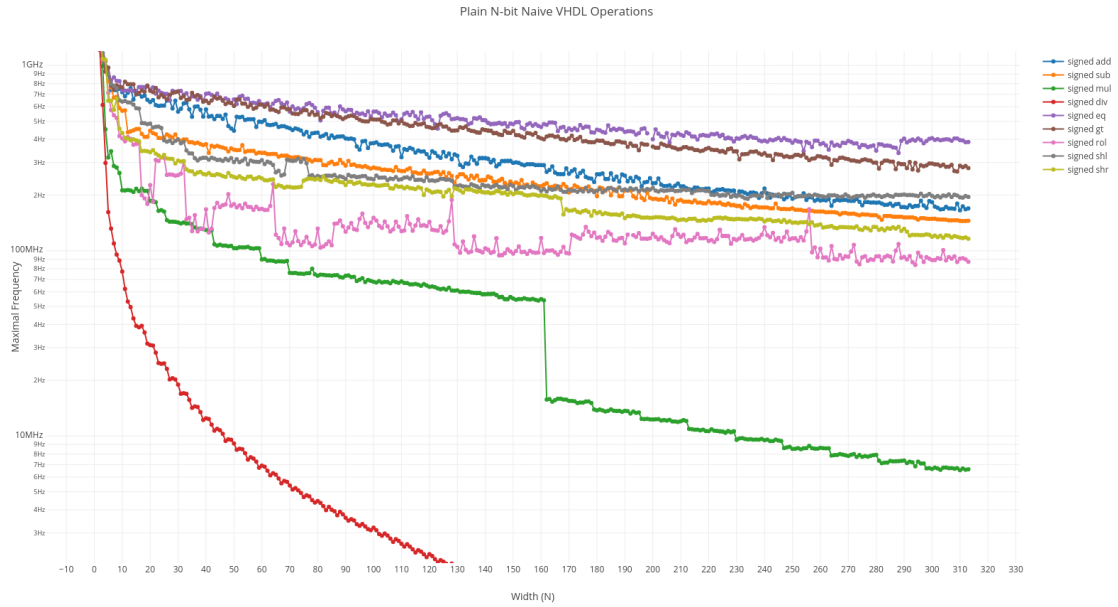


Figure 6.1: The results of a number of basic mathematical operations on the Kintex-7.

Table 6.1 The results for the simple adder benchmarks.

Name	Kintex-7 F_{Max} [MHz]	NG-MEDIUM F_{Max} [MHz]
add8	737.4631	590.6670
add16	738.0074	585.4800
add32	626.9592	554.6310
add64	480.7692	542.8880
add128	326.0515	304.7850
add256	194.9318	313.2830

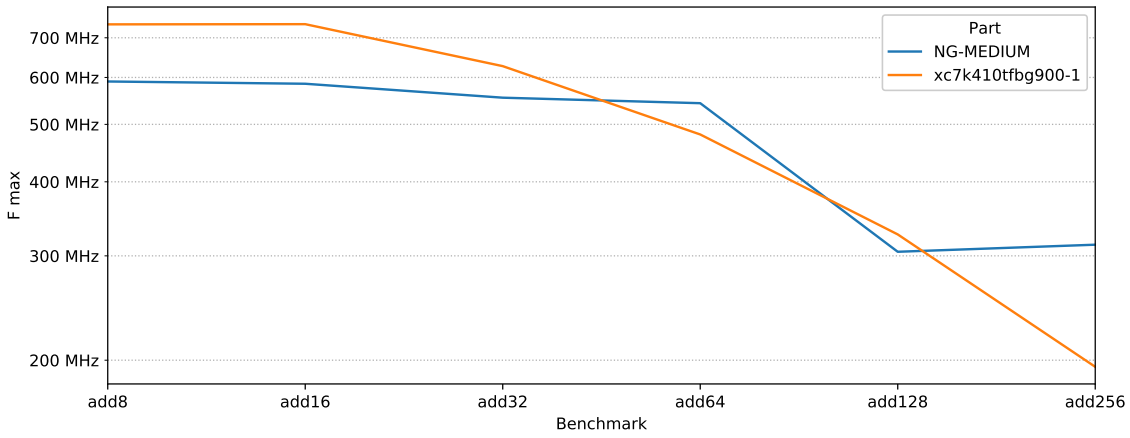


Figure 6.2: The results for the simple adder benchmarks.

6.2 ISCAS'89

Table 6.2 and figure 6.3 show the output data of the ISCAS benchmarks. The order of the table and graphs is the same. With this data we can select a pattern to use as background filler for the "FIR Filter With Filler" benchmark set. We chose `s832` (row highlighted in orange) because it still performs above 240MHz (the maximum speed of the FIR filter) and because its LUT utilization rate is 0.1%. This makes it easy to use multiples of 10 and get a good approximations of what the total LUT utilization rate would be.

The two lines highlighted in yellow `s641` and `s713` are twice as fast on the NG-MEDIUM as the Kintex-7. This is very surprising given the NG-MEDIUM's inherent disadvantage due to the difference in the technology node. Dennard's scaling law roughly predicts a 2x advantage for the Kintex-7.

We speculate that, given that the ISCAS benchmarks all consist of single or dual input gates, it is possible that the design can be placed more efficiently on the NG-MEDIUM with it's 4 inputs LUTs versue the Kintex-7 with it's 6 input LUTs. We think that this means that, when optimizing for speed, the synthesis tool will place more gates in a single LUT. This would mean that the "output" in between some of those packed gates cannot be used to feed another LUT, so those gates will end up in more than one LUT.

The results for the NG-MEDIUM from `s5378` onwards (except `s35932`) are bizarre, because they seem to be missing some flip-flops if we compare the numbers found in the output data with the ISCAS'89 specifications from table 5.1. We don't know where this discrepancy comes from. Analyzing the logged output information from our run script offered no explanation.

Generating this entire dataset takes about 20 minutes on a modern 20 core server.

Table 6.2 The results of the ISCAS benchmarks.

	Kintex-7				NG-MEDIUM			
Name	F Max [MHz]	LUTs	FFs	LUT %	F Max [MHz]	LUTs	FFs	LUT %
s27	564.6527	9	3	0.0035	362.8450	5	3	0.0146
s27a	564.6527	9	3	0.0035	362.8450	5	3	0.0146
s208	320.5128	81	8	0.0319	255.4280	23	8	0.0671
s298	325.6268	109	14	0.0429	254.7120	28	14	0.0817
s344	193.3488	146	15	0.0574	197.5110	43	15	0.1255
s349	196.7342	144	15	0.0566	197.5110	43	15	0.1255
s382	314.6633	146	21	0.0574	194.8180	45	21	0.1313
s386	281.9284	152	6	0.0598	254.1940	39	6	0.1138
s386a	281.9284	152	6	0.0598	254.1940	39	6	0.1138
s400	342.1143	154	21	0.0606	204.4990	45	21	0.1313
s420	289.1845	180	16	0.0708	248.8800	44	16	0.1284
s444	289.6032	169	21	0.0665	201.8980	46	21	0.1342
s510	279.4077	187	6	0.0736	195.1600	81	6	0.2363
s526	329.3808	178	21	0.0700	201.1670	40	21	0.1167
s526a	345.5425	178	21	0.0700	201.1670	40	21	0.1167
s641	52.9577	334	19	0.1314	111.0250	67	19	0.1955
s713	54.1126	355	19	0.1397	111.7190	67	19	0.1955
s820	300.0300	272	5	0.1070	165.6730	87	5	0.2539
s832	287.5216	269	5	0.1058	163.6930	87	5	0.2539
s838	282.7255	322	32	0.1267	253.7430	86	32	0.2509
s953	178.4758	298	29	0.1172	166.6110	134	29	0.3910
s1196	236.6864	440	18	0.1731	126.3260	149	18	0.4348
s1196a	236.6864	440	18	0.1731	126.3260	149	18	0.4348
s1196b	236.6864	440	18	0.1731	126.3260	149	18	0.4348
s1238	237.3042	379	18	0.1491	141.4030	157	18	0.4581
s1238a	237.3042	379	18	0.1491	141.4030	157	18	0.4581
s1423	59.5203	538	74	0.2116	54.0570	169	74	0.4931
s1488	178.6033	543	6	0.2136	162.8660	191	6	0.5573
s1494	184.3998	526	6	0.2069	158.7050	194	6	0.5661
s5378	126.1511	2199	179	0.8651	101.0100	405	161	1.1817
s9234	50.1655	4256	211	1.6743	78.4810	276	133	0.8053
s13207	50.4414	5807	638	2.2844	67.0380	586	475	1.7099
s15850	46.8121	6915	534	2.7203	47.5870	866	457	2.5268
s35932	70.1066	12265	1728	4.8249	141.2430	2626	1728	7.6622
s38417	65.6642	14698	1636	5.7821	54.5200	2726	1538	7.9540
s38584	55.8534	13086	1426	5.1479	68.5260	3362	1384	9.8098

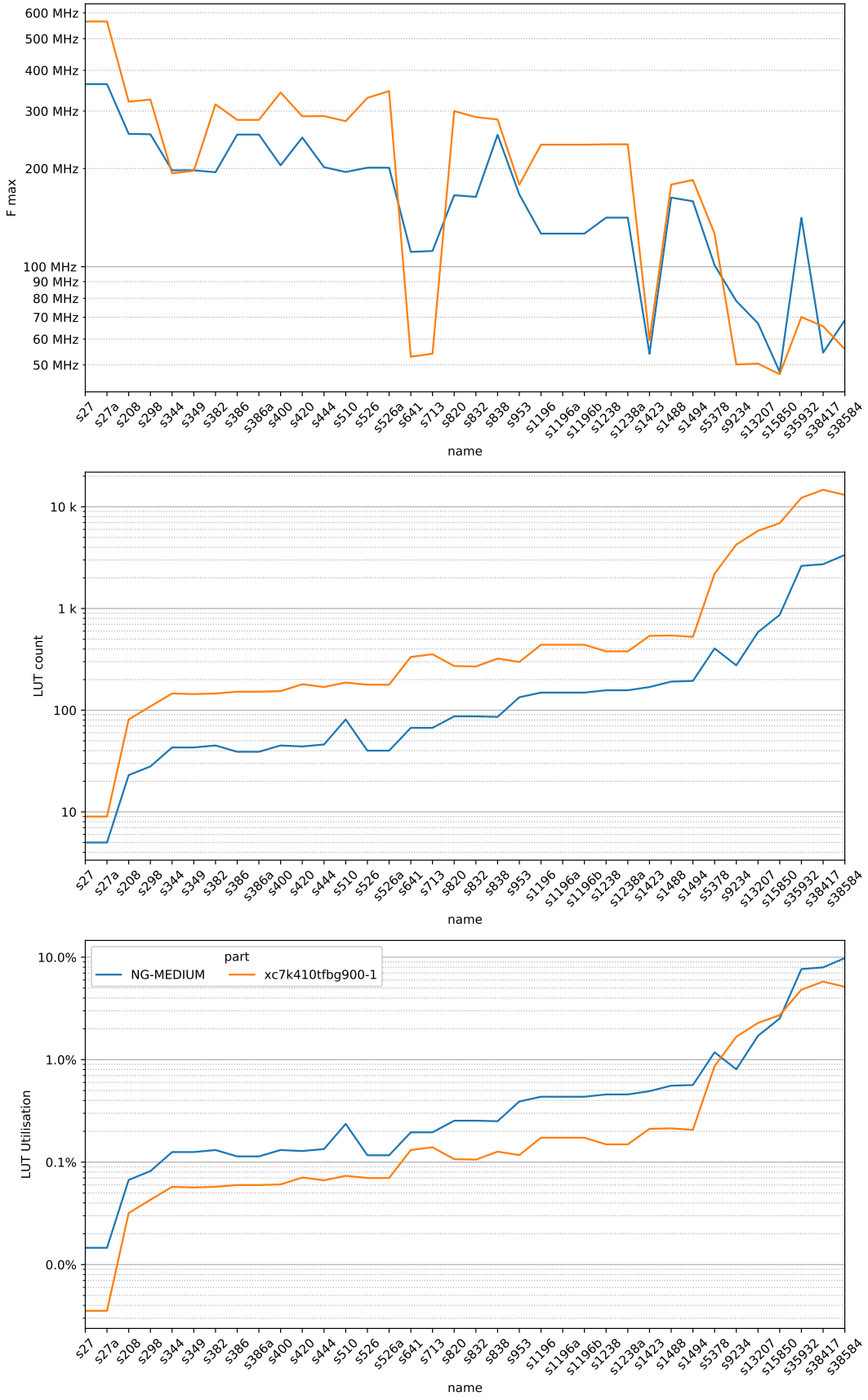


Figure 6.3: The results for the ISCAS benchmarks.

6.3 FIR Filter With Filler

The results below are all for the Kintex-7. The entire dataset is too large to include here, it is included digitally. Unfortunately we could not complete this benchmark set for the NG-MEDIUM due to the instability of the synthesis software. Due to the timing of when we ran the tests we did not have time to try more configurations.

Figure 6.4 shows a “clock speed” versus “device fill rate” graph. This one graph could be considered the summary of the benchmark data. These are our observations from this graph:

1. There is a region where the routing resources are not limiting the performances (below 50% of utilization). The independent FIR filters consistently hit the 220 MHz region. This performance is reachable for limited parts of a design, with the right design attention.

2. As the primary design has to make room for the filler, a second performance plateau is gradually reached at around 170 MHz and 80% utilization rate. These two plateaus, if proven sufficiently invariant, could make their way into a component database as assumptions for early performance estimations. The second point would be the default performance statistic for this device.

The usage limit was set to 90%, so no attempts were made to synthesize a combination of parameters if its utilization exceeded that. Generating this entire dataset takes a few days on a modern 20 core server. This is mostly due to that usage limit. The time it takes for one data point increases dramatically with fill rate. Appendix B.2 has some graphs on that show in more detail which data points were generated.

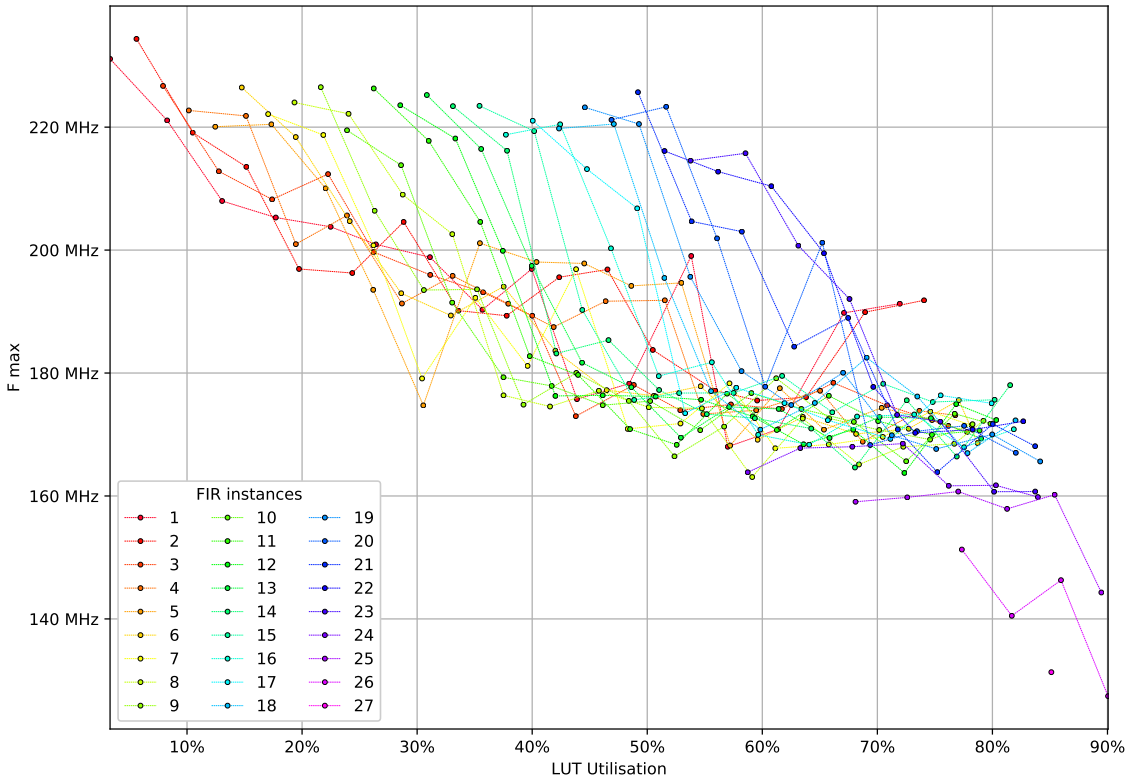


Figure 6.4: The results for the FIR With Filler benchmarks as a scatter plot.

Chapter 7

Conclusion

The FPGA industry is a niche, closed industry where most data comes from manufacturers with no incentive to provide objective data. Having a neutral source of data that can help evaluate a platform or device is useful. Our software enables anyone to generate data tailored to their own needs.

Based on our benchmarks and their results we can conclude that our software provides relevant insights into the performance of a target device. It also determines resource usage for the design being benchmarked. Both of these help improve the resource estimation, which was the intention of this thesis.

Our methodology does require more work and time from the user than simply looking at a manufacturers statistics. Combined with the requirement for at least a basic knowledge of what the final project design will look like, in the form of selecting an appropriate benchmark design, we think it is unlikely that our program will become an industry standard. We consider our contribution valuable none the less.

Our software makes it no longer required to manually generate many iterations of a design with different parameters to find out where the performance optimum lies. Initially we hoped to find a more universal measure to indicate performance, but as we progressed, we realized that the end user will always have to interpret the output data themselves. There simply is no way to represent this in a single statistic.

We implemented our final benchmark and tested it successfully on Xilinx' Kintex-7 on Vivado. We then realized that NanoXplore was not a great choice as our secondary vendor due to its software issues. If we had noticed this sooner or if this was a longer running project, we could have implemented a third vendor. This would have allowed us to generate data for a second target on this benchmark to compare the Kintex-7 results with.

The end user is still responsible for choosing and interpreting a good source of information. Whether that is the manufacturer, a third party, or custom generated data depends on the desired accuracy and the effort the user is willing to put into getting that accuracy.

This project has shown us that benchmarking and performance estimation is a complex matter. The work done for this thesis provides a chance to explore more options than previously available. The addition of more vendor and software support, for starters the other big vendor Intel, would be welcomed. Another addition that can be explored is the a more comprehensive set of “ready to use” benchmarks designs and possibly even automatically generating benchmarks based on higher level designs.

References

Altera. Fpga performance benchmarking methodology. Technical report, 08 2007. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/wp/wpfpgapbm.pdf.

Altera. Opencore stamping and benchmarking methodology. Technical report, 05 2008. URL https://www.intel.com/content/dam/www/programmable/us/en/pdfs/literature/tb/tb-098-opencore-stamping-and-benchmarking-methodology.pdf.

J. Babb, M. Frank, V. Lee, E. Waingold, R. Barua, M. Taylor, J. Kim, S. Devabhaktuni, and A. Agarwal. The raw benchmark suite: computation structures for general purpose computing. In *Proceedings. The 5th Annual IEEE Symposium on Field-Programmable Custom Computing Machines Cat. No.97TB100186)*, pages 134–143, April 1997. doi: 10.1109/FPGA.1997.624613.

M. Bohr. A 30 year retrospective on dennard's mosfet scaling paper. *IEEE Solid-State Circuits Society Newsletter*, 12(1):11–13, Winter 2007. ISSN 1098-4232. doi: 10.1109/N-SSC.2007.4785534.

F. Brglez, D. Bryan, and K. Kozminski. Combinational profiles of sequential benchmark circuits. In *IEEE International Symposium on Circuits and Systems,*, pages 1929–1934 vol.3, May 1989. doi: 10.1109/ISCAS.1989.100747.

R. H. Dennard, F. H. Gaensslen, V. L. Rideout, E. Bassous, and A. R. LeBlanc. Design of ion-implanted mosfet's with very small physical dimensions. *IEEE Journal of Solid-State Circuits*, 9(5):256–268, Oct 1974. ISSN 0018-9200. doi: 10.1109/JSSC.1974.1050511.

Jack J Dongarra, Piotr Luszczek, and Antoine Petitet. The linpack benchmark: past, present and future. *Concurrency and Computation: practice and experience*, 15(9):803–820, 2003.

ESTEC. Margin philosophy for science assessment studies. Technical report, European Space Agency - European Space Research and Technology Centre, 2012. URL http://sci.esa.int/science-e/www/object/doc.cfm?fobjectid=55027.

Q. Gautier, A. Althoff, Pingfan Meng, and R. Kastner. Spector: An opencl fpga benchmark suite.

In *2016 International Conference on Field-Programmable Technology (FPT)*, pages 141–148, Dec 2016. doi: 10.1109/FPT.2016.7929519. URL http://kastner.ucsd.edu/wp-content/uploads/2013/08/admin/fpt16-spector.pdf.

Robert Keim. What is an fpga? an introduction to programmable logic, Aug 2018. URL https://www.allaboutcircuits.com/technical-articles/what-is-an-fpga-introduction-to-programmable-logic-fpga-vs-microcontroller/.

Konstantinos Krommydas, Wu-chun Feng, Christos D. Antonopoulos, and Nikolaos Bellas. Opendwarfs: Characterization of dwarf-based benchmarks on fixed and reconfigurable architectures. *Journal of Signal Processing Systems*, 85(3):373–392, Dec 2016. ISSN 1939-8115. doi: 10.1007/s11265-015-1051-z. URL https://doi.org/10.1007/s11265-015-1051-z.

Thomas Lange. My brave journey, 2017. URL https://amstel.estec.esa.int/tecedm/website/stag_ygt/My%20BRAVE%20Journey.pdf.

C. Maxfield. *The Design Warrior's Guide to FPGAs: Devices, Tools and Flows*. Elsevier Science, 2004. ISBN 9780080477138. URL https://books.google.be/books?id=dnuwr2xOFpUC.

Kevin E Murray, Scott Whitty, Suya Liu, Jason Luu, and Vaughn Betz. Titan: Enabling large and complex benchmarks in academic cad. In *23rd International Conference on Field programmable Logic and Applications*, Sept 2013. doi: 10.1109/FPL.2013.6645503. URL http://www.eecg.utoronto.ca/~kmurray/titan.html.

NanoXplore. From efpga cores to rhbd system-on-chip fpga, 2018. URL https://indico.esa.int/event/232/contributions/2137/attachments/1820/2121/2018-04_NX-From_eFPGA_cores_to_RHBH_SoC_FPGAs-JLM-v2.pdf.

Raphael Njuguna. A survey of fpga benchmarks, Nov 2008. URL https://www.cse.wustl.edu/~jain/cse567-08/ftp/fpga/.

Lukas Rondelez. Vhdl-ontwerp van een generieke, platformonafhankelijke digital downconverter. Master's thesis, KULeuven, 2017.

Henry Selvaraj, Luka Daoud, and Dawid Zydek. A survey of high level synthesis languages, tools, and compilers for reconfigurable high performance computing. volume 240, 09 2013. doi: 10.1007/978-3-319-01857-7_47. URL https://www.researchgate.net/publication/257430335_A_Survey_of_High_Level_Synthesis_Languages_Tools_and_Compilers_for_Reconfigurable_High_Performance_Computing/download.

NASA Space Systems Engineering. Margins and contingency module. Technical report, National Aeronautics and Space Administration - Space Systems Engineering. URL https://spacese.spacegrant.org/uploads/Margins%20Module/16.%20Margins_Module_V1.0.ppt.

Elias Vansteenkiste, Alireza Kaviani, and Henri Fraisse. Analyzing the divide between fpga academic and commercial results. 12 2015. doi: 10.1109/FPT.2015.7393137. URL https://people.eecs.berkeley.edu/~alanmi/publications/other/fpt15_xilinx.pdf.

Xilinx. Measuring device performance and utilization: A competitive overview. Technical report, 08 2017. URL https://www.xilinx.com/support/documentation/white_papers/wp496-comp-perf-util.pdf.

Xilinx. 7 series fpgas data sheet: Overview, 2018. URL https://www.xilinx.com/support/documentation/data_sheets/ds180_7Series_Overview.pdf.

Digital Content

Contents of the included disk and/or USB key:

- This document in PDF form.
- All figures, tables and code snippets in this document.
- The compete source code of all programs written for this thesis.
- All results in CSV format.

An online version is hosted on `thesis.dries007.net`.
The USB key has been made read-only to prevent accidental overwrites.

MIT License

All custom source code is released under the terms of the MIT license. The digital version of the source code includes a license header where it applies. It is omitted in the print version for brevity.

Appendix A

Code

A.1 FPGAPerformanceSuite Python Package

The source code is too big to included in print. The full version is included digitally.

A.2 Runner examples

```
PART_V = 'xc7k410tfbg900-1'
PART_N = 'NG-MEDIUM'


@runner(NanoXplore, single_thread=True)
def nx_math_runner():
    for n in (8, 16, 32, 64, 128, 256):
        yield dict(name='add%d' % n, part=PART_N, files=('../vhdl/add_nx.vhd',),
        ↪  generics={'N': n}, path=NX_PATH)


@runner(Vivado)
def vivado_math_runner():
    for n in (8, 16, 32, 64, 128, 256):
        yield dict(name='add%d' % n, part=PART_V, files=('../vhdl/add.vhd',),
        ↪  generics={'N': n})


@runner(Vivado)
def vivado_iscas89_runner():
    """
    Set for all .vhd files in ../iscas89/vhdl/
    """
    files = list(helpers.natural_sort(glob.iglob('../iscas89/vhdl/*.vhd')))
    logging.debug('ISCAS89 files: %r', files)
    for file in files:
        name = os.path.splitext(os.path.basename(file))[0]
        yield dict(name=name, top=name, part=PART_V, files=[file])


@runner(NanoXplore, single_thread=True)
def nx_iscas89_runner():
    """
    Set for all .vhd files in ../iscas89/vhdl/
    """
    files = list(helpers.natural_sort(glob.iglob('../iscas89/vhdl/*.vhd')))
    logging.debug('ISCAS89 files: %r', files)
    for file in files:
        name = os.path.splitext(os.path.basename(file))[0]
        yield dict(name=name, part=PART_N, top=name, files=(file,), path=NX_PATH)
```

```

@runner(Vivado)
def vivado_fir_s832_runner():
    """
    Complex example with multiple control structures.
    This runner was used to generate the results in the thesis.
    The reason for the complexity is that we didn't know how long the tasks
    would take. With limited compute time available this prioritizes some
    data points over others. Priority high -> low:

    1. 1 -> 25 FIR filters with 1% -> 51% background filler in steps of 10%
    2. 1 -> 25 FIR filters with 1% -> 51% background filler in steps of 5%
    3. 1 -> 25 FIR filters with 1% -> 71% background filler in steps of 10%
    4. 1 -> 25 FIR filters with 1% -> 76% background filler in steps of 5%
    5. 1 -> 50 FIR filters with 1% -> 51% background filler in steps of 10%
    6. 1 -> 50 FIR filters with 1% -> 51% background filler in steps of 5%

    All combinations with an expected fill rate over 90% are skipped.
    The FIR filter takes 2.3%, the filler 0.1% per instance.
    """
    # List of files we need.
    files = ['../iscas89/vhdl/s832.vhd'] + FIR_FILES + TOP_FILES
    logging.debug('vivado_fir_s298_runner files: %r', files)

    pairs = []

    # NA = number of design A = FIR filter
    # NB = number of design B = Filler
        for na in range(1, 26):
        for nb in range(10, 511, 100):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    for na in range(1, 26):
        for nb in range(10, 511, 50):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    for na in range(1, 26):
        for nb in range(10, 711, 100):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    for na in range(1, 26):
        for nb in range(10, 761, 50):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    for na in range(1, 51):
        for nb in range(10, 511, 100):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    for na in range(1, 51):
        for nb in range(10, 511, 50):
            if (na, nb) not in pairs:
                pairs.append((na, nb))
    logging.info('Que length without duplicates: %d', len(pairs))

    for na, nb in pairs:
        if 2.3 * na + 0.1 * nb > 90:
            continue
        name = '%dxFIR_%dxS832' % (na, nb)
        yield dict(name=name, top='top', part=PART_V, files=files, libraries=FIR_LIBS,
                   generics={'TAPS':129, 'COEF_BW':16, 'DATA_BW':16, 'N_A':na, 'N_B':nb})
```

A.3 Converting .bench to VHDL

The source code is too big to included in print. The full version is included digitally.

Example input:

s27.bench

```
INPUT(G0)
INPUT(G1)
INPUT(G2)
```

```
INPUT(G3)

OUTPUT(G17)

G5 = DFF(G10)
G6 = DFF(G11)
G7 = DFF(G13)

G14 = NOT(G0)
G17 = NOT(G11)

G8 = AND(G14, G6)

G15 = OR(G12, G8)
G16 = OR(G3, G8)

G9 = NAND(G16, G15)

G10 = NOR(G14, G11)
G11 = NOR(G5, G9)
G12 = NOR(G1, G7)
G13 = NOR(G2, G12)
```

Example output:

s27.vhd

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

entity s27 is
    port (
        CLK: in std_logic;
        G0: in std_logic;
        G1: in std_logic;
        G2: in std_logic;
        G3: in std_logic;
        G17: out std_logic
    );
end entity;

architecture RTL of s27 is
    signal G5: std_logic;
    signal G6: std_logic;
    signal G7: std_logic;
    signal G8: std_logic;
    signal G9: std_logic;
    signal G10: std_logic;
    signal G11: std_logic;
    signal G12: std_logic;
    signal G13: std_logic;
    signal G14: std_logic;
    signal G15: std_logic;
    signal G16: std_logic;
begin
    process(CLK)
    begin
        if(rising_edge(CLK)) then
            G5<=G10;
            G6<=G11;
            G7<=G13;
        end if;
    end process;
    G14<= not G0;
    G17<= not G11;
    G8<=G14 and G6;
    G9<= not (G16 and G15);
    G15<=G12 or G8;
    G16<=G3 or G8;
    G10<= not (G14 or G11);
    G11<= not (G5 or G9);
    G12<= not (G1 or G7);
    G13<= not (G2 or G12);
end RTL;
```

A.4 Vivado TCL Script

This is a shortened version for printing. The full version is included digitally.

```
# Function color
# ==============
# Return ASCII colored text.
proc color {color text} {
    if {$::colored} {
        return "\x1b\[3${color}m${text}\x1b\[0m"
    } else {
        return $text
    }
}

# Function report_all
# ===================
# Generate all relevant reports
proc report_all {} {
    puts [color 5 "Generating all reports..."]
    exec mkdir report -p
    report_clock_utilization         -quiet -verbose -file report/clock_utilization.txt
    report_utilization               -quiet -verbose -file report/utilization.txt
    return 0
}

# Argument handling
# -----------------

# Defaults (flags)
set verbose 0
set colored 1
# Defaults (k=v)
set period 1
set top "top"
set max_error_percent 5
set max_iterations 10
lappend generics
# Required
lappend files
lappend library_files
set part ""

# Show help/usage
if {$argc < 3} {
    puts [color 1 "Missing arguments, showing usage."]
    puts ""
    puts "Usage: $argv0 <file ...> -p <part> \[optional arguments]"
    puts ""
    puts "Required Arguments:"
    puts "    <file ...>            The VHDL file(s) to load."
    puts "    -p|--part <part>      The name of part/device being benchmarked."
    puts "Optional Arguments:"
    puts "    -v|--verbose          Disable message suppression."
    puts "       --no-color         Disable color in message output."
    puts "    -c|--clock <period>   The initial period used in nanoseconds. Change not
    ↪  recommended. Default: $period"
    puts "    -t|--top <name>       The top module. Default: $period"
    puts "    -e|--error <percent>  The maximal allowed final frequency error in percent.
    ↪  This is the target the script tries to hit. Default: $max_error_percent"
    puts "    -i|--iterations <int> The maximal amount of design runs to do to try and hit
    ↪  the target. Default: $max_iterations"
    puts "    -g|--generic <K=V>    Set VHDL generics. Repeat as required. Must adhere to
    ↪  Vivado's standards (see ug835), example: N=100."
    puts "    -l|--library <K=V>    Add a file to a library. Repeat as required."
    return -1
}

# (Simple) Argument parser.
# Treats single dash and double dash arguments the same.
# There must be a space separating the argument and the value (if any).

# i = Argument index
for {set i 0} {$i < $argc} {incr i} {
    # arg = Current argument string
    set arg [lindex $argv $i]
```

```
69
70      # Starts with _not_ "-", must be a file.
71      if {[string index $arg 0] != "-"} {
72          lappend files $arg
73          continue
74      }
75      # If we are here: Must be a "-" or "--" argument.
76
77      # Strip first "-"
78      set arg [string range $arg 1 end]
79
80      # If starts with "--", strip second "-"
81      if {[string index $arg 0] == "-"} {
82          set arg [string range $arg 1 end]
83      }
84
85      # Flags (no value, just a key)
86      switch -exact -- $arg {
87          "v" -
88          "verbose" {
89              set verbose 1
90              continue
91          }
92          "no-color" {
93              set colored 0
94              continue
95          }
96          default {}
97      }
98      # If we are here: Must be a key-value argument.
99
100     # Find first "=", if not present: -1
101     set eqpos [string first "=" $arg]
102
103     # If there is no "=", the next argument is the value. If there is a "=", cut the string.
104     if {$eqpos == -1} {
105         set key $arg
106         incr i
107         if {$i >= $argc} {
108             puts [color 1 "Argument ($key) specified but no value given."]
109             return -1
110         }
111         set value [lindex $argv $i]
112     } else {
113         set key [string range $arg 0 $eqpos-1]
114         set value [string range $arg $eqpos+1 end]
115     }
116
117     unset eqpos
118
119     # Now we have $key = $value
120
121     switch -exact -- $key {
122         "p" -
123         "part" {
124             set part $value
125         }
126         "c" -
127         "clock" {
128             set period $value
129         }
130         "t" -
131         "top" {
132             set top $value
133         }
134         "e" -
135         "error" {
136             set max_error_percent $value
137         }
138         "i" -
139         "iterations" {
140             set max_iterations $value
141         }
142         "g" -
143         "generic" {
144             lappend generics $value
145         }
146         "l" -
147         "library" {
```

```
148             lappend library_files $value
149         }
150         default {
151             puts [color 1 "Unknown argument: $key = $value"]
152             return -1
153         }
154     }
155 }
156
157 # Required parameter check
158 if {[string length $part] == 0} {
159     puts [color 1 "No part set. Run without arguments to show help."]
160     return -1
161 }
162 if {[llength $files] == 0} {
163     puts [color 1 "No file(s) set. Run without arguments to show help."]
164     return -1
165 }
166
167 # Sanity checks
168 if {$period < 0.01} {
169     puts [color 1 "Initial clock period must be at least 0.01ns"]
170     return -1
171 }
172 if {$max_error_percent < 0.01} {
173     puts [color 1 "Maximal error must be at least 0.01%."]
174     return -1
175 }
176 if {$max_iterations < 1} {
177     puts [color 1 "You must allow at least 1 iteration."]
178     return -1
179 }
180
181 puts [color 2 "Using Parameters:"]
182 puts [color 2 "  files: $files"]
183 puts [color 2 "  part: $part"]
184 puts [color 2 "  period: $period"]
185 puts [color 2 "  top module: $top"]
186 puts [color 2 "  max_error_percent: $max_error_percent"]
187 puts [color 2 "  max_iterations: $max_iterations"]
188 puts [color 2 "  generics: $generics"]
189 puts [color 2 "  libraries: $library_files"]
190
191
192 if {[llength $generics] != 0} {
193     set generics "-generic [join $generics " -generic "]"
194 } else {
195     set generics ""
196 }
197
198 # Start of the actual script
199 # --------------------------
200 # Single shot operations
201 # ======================
202 # Read library files first
203 foreach lf $library_files {
204     set lf_split [split $lf "="]
205     set lib_name [lindex $lf_split 0]
206     set lf_split [lreplace $lf_split 0 0]
207     set f [join $lf_split "="]
208
209     puts [color 2 "  Library ${lib_name}: ${f}"]
210
211     # Allow guess of file extension if not provided and does existence check.
212     if {[string match *.vhdl $f] || [string match *.vhd $f]} {
213         if {[file exists ${f}]} {
214             read_vhdl -library $lib_name -vhdl2008 ${f}
215         } else {
216             puts [color 1 "Error: File ${f} does not exists."]
217             return -1
218         }
219     } else {
220         if {[file exists ${f}.vhdl]} {
221             read_vhdl -library $lib_name -vhdl2008 ${f}.vhdl
222         } elseif {[file exists ${f}.vhd]} {
223             read_vhdl -library $lib_name -vhdl2008 ${f}.vhd
224         } else {
225             puts [color 1 "Error: No vhdl or vhd file exists with name ${f}."]
```

```
226             return -1
227         }
228     }
229 }
230
231 # Read the VHDL files, this only needs to happen once.
232 foreach f $files {
233     # Allow guess of file extension if not provided and does existence check.
234     if {[string match *.vhdl $f] || [string match *.vhd $f]} {
235         if {[file exists ${f}]} {
236             read_vhdl -vhdl2008 ${f}
237         } else {
238             puts [color 1 "Error: File ${f} does not exists."]
239             return -1
240         }
241     } else {
242         if {[file exists ${f}.vhdl]} {
243             read_vhdl -vhdl2008 ${f}.vhdl
244         } elseif {[file exists ${f}.vhd]} {
245             read_vhdl -vhdl2008 ${f}.vhd
246         } else {
247             puts [color 1 "Error: No vhdl or vhd file exists with name ${f}."]
248             return -1
249         }
250     }
251 }
252
253 # Function single_run
254 # ===================
255 # Runs the entire synthesis and implementation chain once.
256 # Returns the period the next run should use.
257 proc single_run {period} {
258     puts [color 5 "Run with period ${period}"]
259     puts [color 5 "Synthesis"]
260
261     synth_design -part ${::part} {*}${::generics} -top ${::top} -assert
262
263     # We need a clock, otherwise none of the timings have meaning.
264     create_clock -period ${period} -name clk [get_ports CLK]
265
266     puts [color 5 "Place"]
267     opt_design
268     place_design
269     phys_opt_design
270
271     puts [color 5 "Route"]
272     route_design
273
274     puts [color 5 "Timing checks"]
275     check_timing
276     report_timing
277 }
278
279 # Main program loop
280 # =================
281 # While we are more than ${max_error_percent}% out, adjust expectations & run again!
282 set start_time_loop [clock clicks -milliseconds]
283 puts [color 5 "Start loop."]
284 set i 0
285 lappend periods ${period}
286 while {${i} < ${max_iterations}} {
287     incr i
288
289     # Actually run
290     set start_time [clock clicks -milliseconds]
291     single_run ${period}
292     set end_time [clock clicks -milliseconds]
293
294     set slack [get_property SLACK [get_timing_paths]]
295
296     # Sometimes slack is "inf", this returns a null/empty string.
297     # This causes syntax errors below.
298     if {$slack eq ""} {
299         puts [color 1 "SLACK is empty string. This is a problem."]
300         report_all
301         puts [color 3 "Stopping! Infinite slack!"]
302         return 1
303     }
304
```

```
    # Vivado is no more accurate than 3 decimal places anyway.
    set period [expr round(1000.0*(${period} - ${slack}))/1000.0];
    set error_precent [expr abs(100*${slack}/${period})]
    set fmax [expr 1000/${period}]

    puts [color 5 "Run $i done with slack: ${slack}ns period: ${period}ns freq: ${fmax}MHz
    ↪  Error: ${error_precent}%"]

    if {${error_precent} < ${max_error_percent}} {
        report_all
        puts [color 2 "Finished!"]
        return 0
    }

    if {[lsearch -exact $periods $period] >= 0} {
        report_all
        puts [color 3 "Stopping! Duplicate period!"]
        return 0
    }
    lappend periods ${period}
}

report_all
puts [color 1 "Stopping! Max iterations!"]
return 1
```

A.5 NanoXplore Python Script

```
#!/bin/env nxpython

from __future__ import print_function
from __future__ import division

import sys
import nanoxmap


def main(name, part, top, *args):
    """
    args must contain IN THIS ORDER:
        1 or more file args
        '--libraries'
        0 or more key-value name<>file pairs (space separated)
        '--generics'
        0 or more key-value key<>value pairs (space separated)

    Expected arguments:

         <name> <part/variant> <top module> [file ...]
            --libraries [<library name> <library file> ...]
            --generics [<generic key> <generic value> ...]
    """
    i_libraries = args.index('--libraries')
    i_generics = args.index('--generics')

    files = list(args[:i_libraries])
    libraries = args[i_libraries+1:i_generics]
    generics = args[i_generics+1:]

    generics = {generics[i]: generics[i+1] for i in range(0, len(generics), 2)}
    libraries_multimap = {}
    for i in range(0, len(libraries), 2):
        lib_name = libraries[i]
        lib_file = libraries[i + 1]
        if lib_name not in libraries_multimap:
            libraries_multimap[lib_name] = []
        libraries_multimap[lib_name].append(lib_file)

    print('----- PARSED ARGS -----')
    print('files:', files)
    print('libraries:', libraries_multimap)
    print('generics:', generics)
    print('----- STARTING... -----')
    # Creates a folder called <name> in which the project is stored.
    p = nanoxmap.createProject(name)
    p.setVariantName(part)
```

```
    p.setTopCellName(top)
    p.addFiles(files)
    for lib_name, lib_file in libraries_multimap.items():
        p.addFiles(lib_name, lib_file)
    p.addParameters(generics)
    # Should be enough to fill up the device.
    p.setOption('MaxRegisterCount', str(30000))
    # Actually do work
    p.synthesize()
    p.place()
    p.reportPorts()
    p.route()
    p.reportInstances()
    a = p.createAnalyzer()
    a.launch()

    print('Errors: ', nanoxmap.getErrorCount())
    print('Warnings: ', nanoxmap.getWarningCount())


if __name__ == '__main__':
    main(*sys.argv[1:])
```

A.6 VHDL Top Module

This is a shortened version for printing. The implementation specific A and B sections have been removed. The full version is included digitally.

```
--           A SEGMENT        | CHAIN |          B SECTION           | CHAIN END
--
--           A---------A                      B---------B
--         /-I         O-\                  /----I         O-\
--        //-I         O-\\                //----I    B    |  \
--       ///-I    A    O-\\\       /-\     ///----I CIRCUIT |   \         /-\
--      ////-I CIRCUIT |  \\\      v |    ////----I         |    \        v |
--     /////-I         |   \\\       |   ////     B---------B     \         |
-- 0>-//////-I         |    \\\-[FF]-//-////                        \--![FF]-//->
-- 1>-// ||| A---------A     \\-[FF]-//-+||-----------------------------![FF]-//->
-- 2>-/  |||                 \-[FF]-//--+|-----------------------------![FF]-//->
-- 3>----+||-------------------[FF]-//---+-----------------------------![FF]-//->
-- 4>-----+|-------------------[FF]-//----\                      /-----![FF]-//->
-- 5>------+-------------------[FF]-//---\ \     B---------B  /     /-![FF]-//->
-- 6>----------------------------[FF]-//---\\ \----I         O-/    //-![FF]-//->
-- 7>----------------------------[FF]-//---\\\---+-I    B    |     ///-![FF]-//->
--                                  |    \\--+--I CIRCUIT |    ///         |
--  This example diagram has:      v  |     \-+---I         |   ///        v  |
--  - A: 6 input to 3 output       \--/       |||B---------B  ///         \--/
--  - B: 4 input to 1 output                  |||            ///
--  - 1x A and 2x B.                          \\\-----------///
--    This isn't realistic.                    \\-----------//
--                                              \-----------/
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use IEEE.MATH_REAL.ALL;

entity top is
    generic (
        -- <<A circuit parameters>>
        -- <<B circuit parameters>>
        -- Global parameters
        N_A: integer := 2;
        N_B: integer := 4;
        MIN_BUS_WIDTH: integer := 1
    );
    port (
        CLK: in std_logic;
        RST: in std_logic
    );
end top;

architecture Behavioral of top is
```

```
45      ------------------------------------------------------------------------
46      -- User editable section: Configurable values
47      ------------------------------------------------------------------------
48      -- Must be updated with changes to A circuit.
49      constant A_IN : integer := 5; -- Number of inputs for A
50      constant A_OUT : integer := 4; -- Number of outputs for A
51      -- Must be updated with changes to B circuit.
52      constant B_IN : integer := 18; -- Number of inputs for B
53      constant B_OUT : integer := 19; -- Number of outputs for B
54      -- << component definition for A >>
55      -- << component definition for B >>
56      ------------------------------------------------------------------------
57      -- End of user editable section.
58      ------------------------------------------------------------------------
59      function max(a, b: integer) return integer is
60      begin
61          if a > b then return a;
62          else return b;
63          end if;
64      end;
65      function min(a, b: integer) return integer is
66      begin
67          if a < b then return a;
68          else return b;
69          end if;
70      end;
71      constant A_IO : integer := max(A_IN, A_OUT); -- Max IO Per A segment
72      constant B_IO : integer := max(B_IN, B_OUT); -- Max IO Per B segment
73      -- Bus width
74      constant BW : integer := max(MIN_BUS_WIDTH, max(A_IO, B_IO));
75      -- Maximum circuits per segment
76      constant A_PER_SEG : integer := BW / A_IO;
77      constant B_PER_SEG : integer := BW / B_IO;
78      -- Number of segments needed = ceil(N/PER_SEG)
79      constant A_SEGMENTS : integer := integer(ceil(real(N_A)/real(A_PER_SEG)));
80      constant B_SEGMENTS : integer := integer(ceil(real(N_B)/real(B_PER_SEG)));
81      constant BUS_SEGMENTS : integer := A_SEGMENTS + B_SEGMENTS;
82      -- 2D std_logic, for segment IO
83      type vector_array IS array(integer range<>) of std_logic_vector;
84      -- Every segment has an in and output. In = input to a segment.
85      -- They are linked by a register and a shift. The loop-around is special.
86      signal data_in: vector_array(BUS_SEGMENTS-1 downto 0)(BW-1 downto 0);
87      signal data_out: vector_array(BUS_SEGMENTS-1 downto 0)(BW-1 downto 0);
88      -- Important! Cannot be optimized away.
89      attribute dont_touch: boolean;
90      attribute dont_touch of data_in: signal is true;
91      attribute dont_touch of data_out: signal is true;
92  begin
93      -- Generate segments.
94      ------------------------------------------------------------------------
95      a_segment_generator: for i in 0 to A_SEGMENTS-1 generate
96          constant used_io : integer := A_PER_SEG * A_IO;
97          signal sin : std_logic_vector(BW-1 downto 0) := data_in(i);
98          signal sout : std_logic_vector(BW-1 downto 0);
99      begin
100         -- 1 segment = x_PER_SEG times circuit + bypass wires
101         a_circuit_generator: for j in 0 to A_PER_SEG-1 generate
102             constant n : integer := i * A_PER_SEG + j;
103             constant s : integer := A_IO * j;
104             -- Circuit in & out. (so indexing math becomes less complex)
105             signal cin : std_logic_vector(A_IO-1 downto 0) := sin(s+A_IO-1 downto s);
106             signal cout : std_logic_vector(A_IO-1 downto 0);
107         begin
108             -- Generate circuit.
109             a_circuit: if n < N_A generate
110                     -- Reduce circuit IO bit width to final dimentions. Use this
111                     signal inp : std_logic_vector := cin(A_IN-1 downto 0);
112                     signal outp : std_logic_vector(A_OUT-1 downto 0);
113                 begin
114                     ------------------------------------------------------------
115                     -- User editable section: A circuit instantiation
116                     ------------------------------------------------------------
117                     -- << A instantiation >>
118                     ------------------------------------------------------------
119                     -- End of user editable section.
120                     ------------------------------------------------------------
121                     cout(A_OUT-1 downto 0) <= outp;
122                 -- Bypass some signals if there are more inputs than outputs
123                 a_circuit_bypass: if A_IN > A_OUT generate
124                     cout(A_IO-1 downto A_OUT) <= cin(A_IO-1 downto A_OUT);
```

```
125                 end generate;
126             end generate;
127             -- Reached target circuit count, bypass all.
128             a_circuit_done: if n >= N_A generate
129                 cout <= cin;
130             end generate;
131         sout(s + A_IO - 1 downto s) <= cout;
132         end generate;
133         -- Bypass any wires not taken by any circuit
134         a_segment_bypass: if used_io < BW generate
135             sout(BW-1 downto used_io) <= sin(BW-1 downto used_io);
136         end generate;
137         data_out(i) <= sout;
138     end generate;
139     ------------------------------------------------------------------------
140     b_segment_generator: for i in 0 to B_SEGMENTS-1 generate
141         constant used_io : integer := B_PER_SEG * B_IO;
142         signal sin : std_logic_vector(BW-1 downto 0) := data_in(i+A_SEGMENTS);
143         signal sout : std_logic_vector(BW-1 downto 0);
144     begin
145         -- 1 segment = x_PER_SEG times circuit + bypass wires
146         b_circuit_generator: for j in 0 to B_PER_SEG-1 generate
147             constant n : integer := i * B_PER_SEG + j;
148             constant s : integer := B_IO * j;
149             -- Circuit in & out. (so indexing math becomes less complex)
150             signal cin : std_logic_vector(B_IO-1 downto 0) := sin(s + B_IO - 1 downto s);
151             signal cout : std_logic_vector(B_IO-1 downto 0);
152         begin
153             -- Generate circuit.
154             b_circuit: if n < N_B generate
155                     -- Reduce circuit IO bit width to final dimentions. Use this
156                     signal inp : std_logic_vector := cin(B_IN-1 downto 0);
157                     signal outp : std_logic_vector(B_OUT-1 downto 0);
158                 begin
159                     ------------------------------------------------------------
160                     -- User editable section: B Circuit instantiation
161                     ------------------------------------------------------------
162                     -- << B instantiation >>
163                     ------------------------------------------------------------
164                     -- End of user editable section.
165                     ------------------------------------------------------------
166                     cout(B_OUT-1 downto 0) <= outp;
167                 -- Bypass some signals if there are more inputs than outputs
168                 b_circuit_bypass: if B_IO > B_OUT generate
169                     cout(B_IO-1 downto B_OUT) <= cin(B_IO-1 downto B_OUT);
170                 end generate;
171             end generate;
172             -- Reached target circuit count, bypass all.
173             b_circuit_done: if n >= N_B generate
174                 cout <= cin;
175             end generate;
176             sout(s + B_IO - 1 downto s) <= cout;
177         end generate;
178         -- Bypass any wires not taken by any circuit
179         b_segment_bypass: if used_io < BW generate
180             sout(BW-1 downto used_io) <= sin(BW-1 downto used_io);
181         end generate;
182         data_out(i+A_SEGMENTS) <= sout;
183     end generate;
184     ------------------------------------------------------------------------
185     -- Chain link bits. The final/first link is special.
186     -- WARNING: Here data_out is the input and vice versa!
187     link_generator: for i in 0 to BUS_SEGMENTS-2 generate
188     begin
189         process(RST, CLK)
190         begin
191             if RST = '1' then
192                 data_in(i+1) <= (others => '0');
193             elsif rising_edge(CLK) then
194                 data_in(i+1) <= data_out(i) rol 1;
195             end if;
196         end process;
197     end generate;
198     -- Last link: Invert & rotate.
199     process(RST, CLK)
200     begin
201         if RST = '1' then
202             data_in(0) <= (others => '0');
203         elsif rising_edge(CLK) then
```

```
204             data_in(0) <= not data_out(BUS_SEGMENTS-1) rol 1;
205         end if;
206     end process;
207 end architecture Behavioral;
```

Appendix B

Miscellaneous

B.1 Hierarchical Utilization Report

To make the table fit on one page, the LUTRAMs, RAMB36, and RAMB18 columns were removed. The Instance and Module columns where shortened.

These particular tables are from the `Vivado_2xFIR_10xS832` benchmark.

This is the `Slice Logic` table in the `utilization.txt` report.

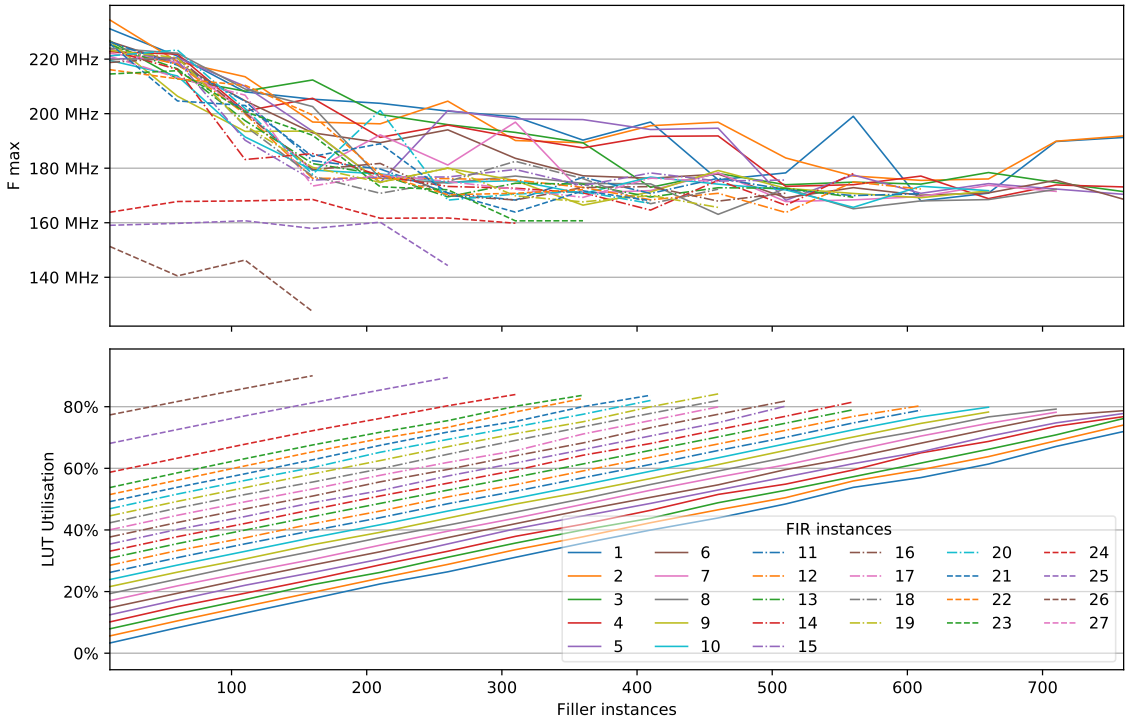
Site Type	Used	Fixed	Available	Util%
Slice LUTs	14254	0	254200	5.61
LUT as Logic	14189	0	254200	5.58
LUT as Memory	65	0	90600	0.07
LUT as Distributed RAM	0	0		
LUT as Shift Register	65	0		
Slice Registers	20750	0	508400	4.08
Register as Flip Flop	20750	0	508400	4.08
Register as Latch	0	0	508400	0.00
F7 Muxes	0	0	127100	0.00
F8 Muxes	0	0	63550	0.00

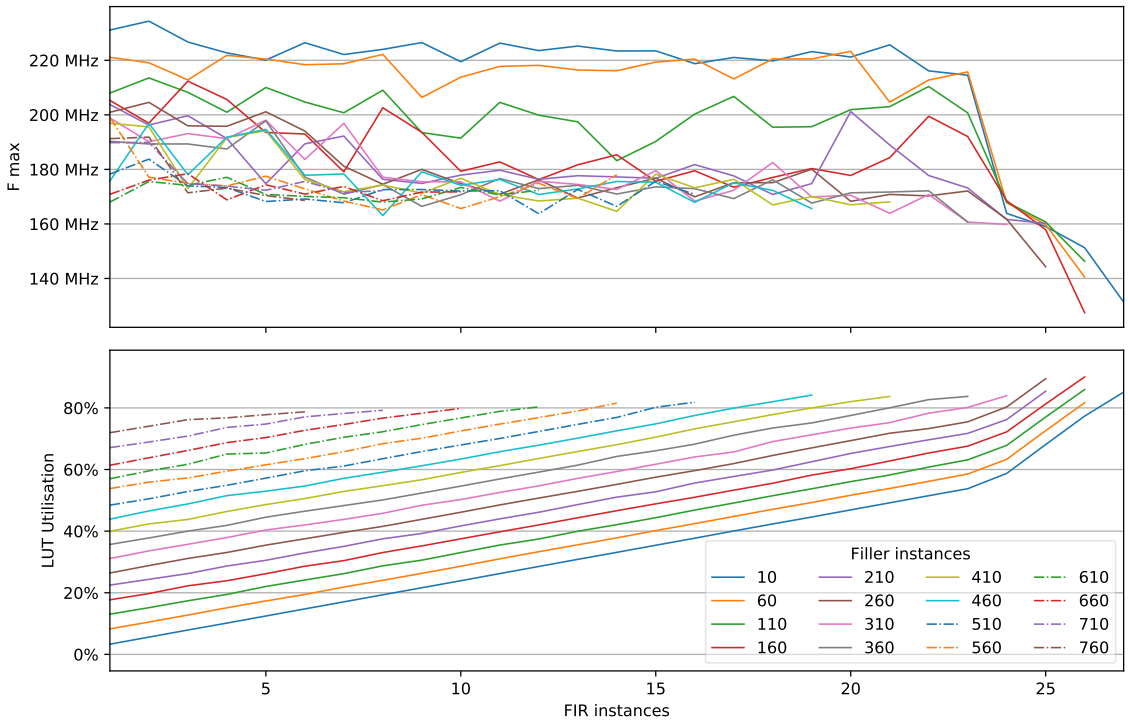
The next table is the `Utilization by Hierarchy` table in the `utilization_h.txt` report. This table is not parsed by our software due to its complexity (both in terms of parsing and representing it). It can be examined by the end user when needed.

Instance (Shortened)	Module (Shortened)	Total LUTs	Logic LUTs	SRLs	FFs	DSP48 Blocks
top	(top)	14254	14189	65	20750	130
(top)	(top)	25	25	0	364	0
a_segment_generator[0].a_circuit_generator[0].a_circui	fir	5826	5793	33	10160	65
(a_segment_generator[0].a_circuit_generator[0].a_cir	fir	1058	1058	0	1062	0
FIR_INSTANCE	symmFIR_top_10	4768	4735	33	9098	65
(FIR_INSTANCE)	symmFIR_top_10	1024	1024	0	4224	65
u_sop_tree	array_sum_tree_11	3744	3711	33	4874	0
(u_sop_tree)	array_sum_tree_11	1088	1088	0	2128	0
gen_array_tree[1].gen_first_layer.e_sum_layer0	array_sum_layer_12	1330	1297	33	1280	0
gen_array_tree[2].gen_layers.e_sum_layer	array_sum_layer_13	657	657	0	673	0
gen_array_tree[3].gen_layers.e_sum_layer	array_sum_layer_14	320	320	0	353	0
gen_array_tree[4].gen_layers.e_sum_layer	array_sum_layer_15	160	160	0	200	0
gen_array_tree[5].gen_layers.e_sum_layer	array_sum_layer_16	80	80	0	120	0
gen_array_tree[6].gen_layers.e_sum_layer	array_sum_layer_17	40	40	0	80	0
gen_array_tree[7].gen_layers.e_sum_layer	array_sum_layer_18	69	69	0	40	0
a_segment_generator[1].a_circuit_generator[0].a_circui	fir_0	5824	5792	32	10176	65
(a_segment_generator[1].a_circuit_generator[0].a_cir	fir_0	1058	1058	0	1062	0
FIR_INSTANCE	symmFIR_top	4766	4734	32	9114	65
(FIR_INSTANCE)	symmFIR_top	1024	1024	0	4224	65
u_sop_tree	array_sum_tree	3742	3710	32	4890	0
(u_sop_tree)	array_sum_tree	1089	1089	0	2129	0
gen_array_tree[1].gen_first_layer.e_sum_layer0	array_sum_layer	1328	1296	32	1281	0
gen_array_tree[2].gen_layers.e_sum_layer	array_sum_layer_0	656	656	0	680	0
gen_array_tree[3].gen_layers.e_sum_layer	array_sum_layer_1	320	320	0	360	0
gen_array_tree[4].gen_layers.e_sum_layer	array_sum_layer_2	160	160	0	200	0
gen_array_tree[5].gen_layers.e_sum_layer	array_sum_layer_3	80	80	0	120	0
gen_array_tree[6].gen_layers.e_sum_layer	array_sum_layer_4	40	40	0	80	0
gen_array_tree[7].gen_layers.e_sum_layer	array_sum_layer_5	69	69	0	40	0
b_segment_generator[0].b_circuit_generator[0].b_circui	s832	270	270	0	5	0
b_segment_generator[1].b_circuit_generator[0].b_circui	s832_1	254	254	0	5	0
b_segment_generator[2].b_circuit_generator[0].b_circui	s832_2	260	260	0	5	0
b_segment_generator[3].b_circuit_generator[0].b_circui	s832_3	256	256	0	5	0
b_segment_generator[4].b_circuit_generator[0].b_circui	s832_4	264	264	0	5	0
b_segment_generator[5].b_circuit_generator[0].b_circui	s832_5	262	262	0	5	0
b_segment_generator[6].b_circuit_generator[0].b_circui	s832_6	265	265	0	5	0
b_segment_generator[7].b_circuit_generator[0].b_circui	s832_7	269	269	0	5	0
b_segment_generator[8].b_circuit_generator[0].b_circui	s832_8	250	250	0	5	0
b_segment_generator[9].b_circuit_generator[0].b_circui	s832_9	257	257	0	5	0

B.2 FIR Filter With Filler Graphs

The graphs below show the same output data but grouped by filler instance count and FIR instance count respectively.





FACULTEIT INDUSTRIËLE INGENIEURSWETENSCHAPPEN
CAMPUS DE NAYER SINT-KATELIJNE-WAVER
J. De Nayerlaan 5
2860 SINT-KATELIJNE-WAVER, België
tel. + 32 15 31 69 44
iiw.denayer@kuleuven.be
www.iiw.kuleuven.be