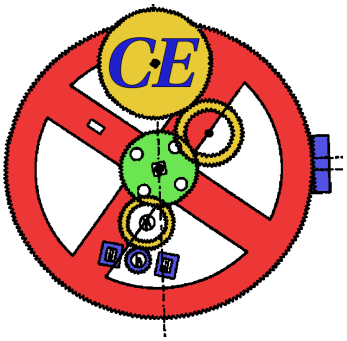


Computer Engineering
Mekelweg 4,
2628 CD Delft
The Netherlands
http://ce.et.tudelft.nl/

2012

MSc THESIS

Towards high-level synthesis of neural models on FPGAs

Zaid Al-Hilli

Abstract

Recent trends in studying the brain activity have attracted interest in the simulation of neurons to understand the brain functionality. However, these simulations are computationally intensive and time consuming, which limits this study. High-performance FPGA-based platforms are being utilized to accelerate these simulations. However, the transformation of sequential code to a data flow graph, the scheduling of operations for an efficient use of resources, the generation of hardware description code, the simulation and synthesis are tedious and time consuming tasks. Furthermore, the design space is huge and various scheduling algorithms need to be evaluated based upon variants of area-delay cost metrics to make decisions. An open source tool has been developed at TUDelft to automate this process, which considerably reduces the implementation time. This tool allows us to schedule, plot, explore the design space and generate VHDL code. We have utilized this tool in order to implement in hardware, a detailed neural network model of the inferior olive module in the brain. In order to support the model's mathematical requirements, we have extended the tool with three more operations and added the support for floating-point arithmetic. Based on various cost metrics, two Pareto points were chosen for synthesis. The neural network model was synthesized at the granularity level of 1, 2 and 4 neurons. Performance analysis of the synthesis results shows that, in the best case, we were able to achieve a speedup of 80% over real-time execution for 500 neurons. We have also showed that using smaller neuron cells results in higher speedup and better area utilization.

CE-MS-2012-18

Towards high-level synthesis of neural models on FPGAs

THESIS

submitted in partial fulfillment of the
requirements for the degree of

MASTER OF SCIENCE

in

COMPUTER ENGINEERING

by

Zaid Al-Hilli
born in Baghdad, Iraq

Computer Engineering
Department of Electrical Engineering
Faculty of Electrical Engineering, Mathematics and Computer Science
Delft University of Technology

Towards high-level synthesis of neural models on FPGAs

by Zaid Al-Hilli

Abstract

Recent trends in studying the brain activity have attracted interest in the simulation of neurons to understand the brain functionality. However, these simulations are computationally intensive and time consuming, which limits this study. High-performance FPGA-based platforms are being utilized to accelerate these simulations. However, the transformation of sequential code to a data flow graph, the scheduling of operations for an efficient use of resources, the generation of hardware description code, the simulation and synthesis are tedious and time consuming tasks. Furthermore, the design space is huge and various scheduling algorithms need to be evaluated based upon variants of area-delay cost metrics to make decisions. An open source tool has been developed at TUDelft to automate this process, which considerably reduces the implementation time. This tool allows us to schedule, plot, explore the design space and generate VHDL code. We have utilized this tool in order to implement in hardware, a detailed neural network model of the inferior olive module in the brain. In order to support the model's mathematical requirements, we have extended the tool with three more operations and added the support for floating-point arithmetic. Based on various cost metrics, two Pareto points were chosen for synthesis. The neural network model was synthesized at the granularity level of 1, 2 and 4 neurons. Performance analysis of the synthesis results shows that, in the best case, we were able to achieve a speedup of 80% over real-time execution for 500 neurons. We have also showed that using smaller neuron cells results in higher speedup and better area utilization.

Laboratory : Computer Engineering
Codenumber : CE-MS-2012-18

Committee Members :

Advisor:	Christos Strydis, Department of Neuroscience, Erasmus MC
Advisor:	Sebastian Isaza, Department of Neuroscience, Erasmus MC
Chairperson:	Koen Berterls, CE, TU Delft
Member:	Rene van Leuken, CAS, TU Delft
Member:	Carlo Galuzzi, CE, TU Delft

To my grandparents, parents, brother, family and friends. A special feeling of gratitude to my loving grandparents and parents, whose words of encouragement and push for tenacity ring in my ears. My brother has never left my side and is very special. Without their help and support in each step it simply never would have been.

Contents

List of Figures vii

List of Tables ix

Acknowledgements xi

1 Introduction 1
- 1.1 Problem Statement . . . 1
- 1.2 Thesis Objectives . . . 2
- 1.3 Document Structure . . . 3

2 Related Work 5
- 2.1 High-level Synthesis (HLS) Tools . . . 5
  - 2.1.1 The Catapult C Synthesis tool . . . 5
  - 2.1.2 The HercuLeS high-level synthesis tool . . . 6
  - 2.1.3 Vivado High-Level Synthesis . . . 6
  - 2.1.4 Impulse C . . . 6
  - 2.1.5 LegUp . . . 6
  - 2.1.6 The TUDelft tool . . . 6
- 2.2 Scheduling Techniques . . . 6
  - 2.2.1 Static Scheduling . . . 7
- 2.3 Neural Modeling in hardware . . . 9
- 2.4 Summary . . . 11

3 Background 13
- 3.1 Neurons and the Brain . . . 13
  - 3.1.1 Neuron Cells . . . 14
- 3.2 Neural Models . . . 16
  - 3.2.1 The Hodgkin-Huxley Model . . . 16
  - 3.2.2 Compartmental Models . . . 17
- 3.3 The Inferior Olive . . . 18
- 3.4 Summary . . . 19

4 Application Description and Tool-flow 21
- 4.1 The Inferior-Olive Network Model . . . 21
- 4.2 Scheduling and the TUDelft Tool . . . 24
  - 4.2.1 Scheduling Methods . . . 25
  - 4.2.2 Limitations . . . 30
  - 4.2.3 Modifications . . . 30
- 4.3 New Capabilities . . . 32
- 4.4 Summary . . . 35

5 Design Evaluation **37**
5.1 Tools and Methods . . . 37
5.2 Experimental Setup . . . 38
5.2.1 Validation Methods . . . 38
5.3 Results . . . 38
5.3.1 Design-Space . . . 39
5.3.2 Design Space Initialization . . . 40
5.3.3 Design-Space Exploration (DSE) . . . 40
5.3.4 Evaluation Metrics . . . 42
5.3.5 Area and Performance . . . 42
5.3.6 AD Metric . . . 43
5.3.7 AD2 Metric . . . 44
5.3.8 A2D Metric . . . 44
5.3.9 Conclusion . . . 46
5.4 VHDL Code Generation . . . 51
5.5 Design Simulation and Synthesis . . . 51
5.6 Scalability Analysis . . . 51
5.7 Discussion . . . 56
5.8 Summary . . . 56

6 Conclusions and Future Work **59**
6.1 Evaluation Conclusions . . . 59
6.2 Thesis Contributions . . . 61
6.3 Future Work . . . 61

A Appendix: TUDelft Tool Modifications **63**
A.1 Tool Modification . . . 63

Bibliography **71**

List of Figures

2.1 Scheduling Classes [32] 8
2.2 Motor Activity 11

3.1 Nervous System [1] 14
3.2 Basic illustration of a biological neuron [10] 15
3.3 Hodgkin-Huxley model [5] 17
3.4 Compartmental model [5] 18

4.1 Microcircuitry of the cerebellum [34] 22
4.2 The IO-cell with its three compartments 23
4.3 The 2-D mesh of the IO-Network 24
4.4 The TUDelft tool 27
4.5 Data-Flow Graph (DFG) based on ASAP 28
4.6 Data-flow graph obtained with view_DFG(). 1D represents one Delay Element, 5D means 5 consecutive delays 29
4.7 Resource mapping and Register binding as reported by gen_VHD() function 30
4.8 Functions dependency graph 31
4.9 .cir files before and after modification, in which $ is EXP, / is DIV, * is MUL, +/- is ALU and ? is MIN 32
4.10 The design space of the FIR5.cir that is plotted by the old Xplore() function 33
4.11 The design space of the FIR5.cir that is plotted by the new Xplore() function 33
4.12 Screenshot of the schedGUI() function before and after modification . . . 34
4.13 Resource mapping and Registry binding as reported by gen_VHD() function after modification 35

5.1 Strategy steps 39
5.2 Design-space (DS) fo (IOcellFunctionCa) 41
5.3 Area and delay States (performance) objectives 43
5.4 Compound metrics 45
5.5 Distribution windows for the design points (ALAP and LIST41) of Table 5.1 47
5.6 Distribution windows (LIST150 and LIST250) for the design points of Table 5.1 48
5.7 Data flow graph (DFG) of our neuron cell by using ALAP 49
5.8 Data flow graph (DFG) of our neuron cell by using LIST41 50
5.9 The architecture of the generated VHDL code 52
5.10 The synthesis result of a single cell using LIST41 and LIST150 52
5.11 A single IO-Cell 53
5.12 Two interconnected IO-Cells 54
5.13 Four interconnected IO-Cells 54

List of Tables

5.1 Conclusion Table 46
5.2 Scalability Table of the second scenario 55

Acknowledgements

First and foremost my sincerest gratitude to my supervisors, Dr Christos Strydis and Dr Sebastian Isaza Ramirez, who have supported me throughout my thesis with patience and knowledge, which giving me the space to allowing me the to work independently. I assign the level of my Masters degree to their encouragement, potential and constant attention to detail. Moreover, without them this thesis would not have been completed or even written. One simply could not wish for better or friendlier supervisors.

I would also like to thank Rob Seepers and Georgios Smaragdos for their invaluable help with the modification of the tool and evaluation of the result.

Special thanks to Professor Rene van Leuken and Mr. Huib Lincklaen Arriens for their support and the collaboration of the TUDelft tool.

Professor Chris de Zeeuw has offered much advice and insight throughout my work on the cerebellar modeling and the future work. Dr Bas Koekkoek provided unlimited encouragement and support throughout. Henk-Jan Boele has provided me with the neuron-scientific information covered in this thesis.

I am very grateful to Professor Arjan van Genderen for his support, without him I would not have been able to start on my master at the Delft University of Technology and also grateful to both Professor George Gaydadjiev and Professor Zaid Al-Ars for offering me the chance to meet my supervisour, Dr Christos Strydis.

I also extend sincere thanks to the following people, without whom the study would not have been possible: Drs Sonja van Boxtel and Imran Ashraf.

My thanks to Drs Diane Butterman-Dorey for proof-reading several pages and for providing helpful suggestions for improving this thesis.

Thank you my friends Pepijn Westen and Mottaqiallah Taouil who in one way or another were the best assistances and advisors.

Many thanks to the UAF organization and its old and new stuff for their assistance, concern and encouragement: Martin van der Randen, Friso Harms, Petra Veltman, Afsaneh Najjar Azari and Stannie Maessen.

The Computer Engineering Group and the students advisors (especially Mrs Esther Buizer-Janssen) at Delft University of Technology and the department of Neuroscience at the Erasmus Medical Center have provided the support and equipment I needed to produce and complete my study and my thesis.

Lastly, I offer my warmest regards and thanks to my grandparents, parents and family for their continued support. Without them I would not have been able to continue my study and I will always appreciate their love and patience.

Zaid Al-Hilli
Delft, The Netherlands
October 23, 2012

1 Introduction

"For, usually and fitly, the presence of an introduction is held to imply that there is something of consequence and importance to be introduced."
-Arthur Machen

Despite of huge efforts and significant discoveries in the field of neuroscience, scientists do not fully understand yet how the brain functions. Two of the main reasons are the vast amount of neurons present (nearly 100 billion) and the sheer complexity of the interconnection network. In trying to understand brain mechanisms, scientists are doing experiments in which they measure brain activity by using electrodes, which is very difficult because these need to be extremely small. Moreover, the electrical signals that pass through the neurons are also very difficult to measure because the voltage levels are low and therefore very sensitive to noise. Nevertheless, scientists have collected enough data to allow them to build mathematical neuron models. Running simulations of these models becomes a complementary research tool which, in many cases, could replace in vivo procedures.

Another scenario in which brain modeling is useful is for brain rescuing. As a result of an accident or a neurological disorder, part of the brain can be damaged. If we have accurate enough computational models then we would be able to replace that damaged part with a computational electronic device.

1.1 Problem Statement

As it was mentioned above, two ultimate goals of building a neural model are:

- Simulating the brain to understand its underlying mechanisms
- Rescueing a damaged part of the brain

Neuroscientists use neural models to simulate the brain. These neural models are mathematical representations of the biological principles that govern the behavior of neuron cells. They are therefore designed to accurately describe and predict biological processes. Simulating them can give us insight into brain mechanisms. Software programming languages are used to implement the neural mathematical representation. Different frameworks and languages (like Genesis, Neuron, Matlab, Java, C, etc) have been used extensively to simulate small parts of the brain. However, these models are very complex, especially given the fact that we would like to simulate networks containing many neurons. The platform type also affects the execution time. Desktop PCs are in most cases too slow. For instance, Matlab simulations of the neural model that we use in this thesis [13] exhibit a performance 2.5 times slower than the brain real-time for

a single neuron cell. When using C to simulate the same model, the simulation becomes 20 times faster than the brain real-time [20]. This also means that simulating a cluster of 100 cells for 10 minutes of real time (100x10minx2.5) would take (2500 minutes) 42 hours. Therefore, faster simulation platforms are very important. The most common way to achieve a high speed is by using supercomputers. These are systems made of many desktop-like computers interconnected and working simultaneously. One alternative to supercomputers is FPGAs. These are reconfigurable application-specific chips with high computational power. The first option needs a big place to accommodate a large amount of computers or racks, that also have huge power consumption. The latter are application-specific chips with high computational power density. They are very small and could perform as well as the former while being more portable and consuming less power. However, that speed through a parallel custom design comes at the expense of a large design effort.

Two questions we aim at answering in this thesis are:

- How many neuron cells can we simulate at brain real-time?
- How fast can we simulate a given amount of neuron cells?

Brain simulation is more concerned with highest performance whereas rescuing a defective part concerns the size of the design and its power consumption. However, they also have things in common as they should be high performance devices because we have to implement a large number of neurons in real-time or, in the case of simulation, as fast as possible. The challenge we address in this thesis is that of producing an efficient FPGA implementation of the target application.

1.2 Thesis Objectives

This thesis sets out to attain the following objectives:

- Tool selection: To select a High Level Synthesis (HLS) tool amoung different HLS tools that helps us to automate the process of code conversion from software to hardware.
- Scheduling: To explore different scheduling methods to optimize our design. The scheduling work should be done statically, thus, before the code conversion step. This provides us with a more efficient implementation.
- Hardware implementation of the targeted neural model: To produce a hardware implementation of the design point that we will choose based on some cost metrics. Several cost metrics involving area and delay will be used for choosing an appropriate design point.
- Scalability analysis: After the design point for one cell has been chosen, the scaling of performance and area should be analyzed when more cells are implemented and interconnected.

The selection of the tool is restricted by many factors like the availability and the functionality of these tools. The scheduling objective is related to the previous one because if we want to have an efficient implementation we have to assure that the HLS tool that we want to choose has this feature. Once we are done with the scheduling step, we convert our design into hardware, based on the scheduling information after applying some cost metrics. Same as the previous objective, the HW implementation of the targeted neural model is also related to the first objective. In the end, we perform a scalability analysis for the design that we will choose before mapping the design on the FPGA.

One of the goals of this thesis is to have an implementation that consumes as little area as possible. Therefore, tackling the scheduling problem is compulsory. Many HLS tools have been introduced to tackle this problem. Such tools also help us to convert the software code of an application into hardware. The first objective of this thesis is to select one of these tools. There are many scheduling methods that have recently been introduced to solve the scheduling problem, using this tool will help us to achieve such a scheduling process much better than doing it manually. Another important objective of this thesis work is the HW implementation of the targeted neural model. Such a tool supports that feature by generating VHDL code of our model. A thorough study should be performed to make a scalability analysis and check suitability for implementation.

1.3 Document Structure

This thesis is organized as follows: in Chapter 2 we make a brief presentation of the related work in the subject of HLS tools, scheduling and brain hardware modeling. Chapter 3 gives the basic concepts needed to follow the remainder of this thesis and also a few additional subjects in which the HW model is the core. In Chapter 4 we present the implementation of the design and the tool modifications. In Chapter 5 we describe the experimental setup and the evaluation of the results for our design. Lastly, in Chapter 6, we present the conclusions stemming from the evaluation of our work and provide suggestions for future research.

2 Related Work

"Consider the past and you shall know the future."
-Chinese Proverb

In order to have high performance and small hardware design mapped on an FPGA, we should have a thorough understanding of the data flow graph (DFG) of the target application and the scheduling techniques that are used. In this chapter, we present some High Level Synthesis (HLS) tools that are used to solve such problem along with other features like converting a software code into hardware. Then, we present a literature survey of scheduling and its classes. Finally, we present related works on the implementation of our targeted application on the FPGA.

2.1 High-level Synthesis (HLS) Tools

High-level synthesis (HLS) tools are very useful ways of helping us to convert the high level description design (software design) into register transfer level (hardware) very quickly. Some of these tools have built-in functions in which they tackle the scheduling problem. In general, by using an HLS tool, user constraints could be automatically specified and time to market reduced. However, most of these tools have some limitations like it does not show a design space graph for different scheduling method results. Moreover, because it is a commercial tool, its source code is not available to us to modify for our specific needs. In addition, we do not know exactly whether it is supporting scheduling issues or what the metrics used to perform the scheduling solutions are. Lastly, such a tool does not provide different cost functions that help us choose a point for our analysis.

2.1.1 The Catapult C Synthesis tool

Catapult C Synthesis [3] is a high-level synthesis tool that helps to implement an optimal hardware design. It takes high level code written in C++ and/or SystemC as input and generates an RTL for complex hierarchical systems comprised of control-logic and algorithmic units. This tool offers easy control over the synthesized hardware implementation such as interface protocols, memory architecture, throughput, latency and low-power transformations. It is also able to schedule hardware resources and structured control logic efficiently which is used to produce superior designs and predictable timing closure. It has a built-in analysis tool that is used to analyze, optimize, and converge on the optimal solution in terms of power, performance, and area. It could automatically verify the VHDL code against the original C++ or systemC code.

2.1.2 The HercuLeS high-level synthesis tool

HercuLeS [7] is a high-level synthesis tool that automatically generates a VHDL code for non-programmable hardware. HercuLeS can translate a C code to NAC which, in turn, is translated into VHDL code. NAC is a bit-accurate typed-assembly language. The VHDL code that the tool generates contains a Finite-State Machines with datapath. The VHDL code generated by HercuLeS can be simulated with GHDL and the industry-standard Modelsim. It is possible to generate VHDL using either the Synopsys packages (the "old" de-facto standard) or the official IEEE library packages.

2.1.3 Vivado High-Level Synthesis

Vivado High-Level Synthesis [8] is a HLS tool (for FPGAs and ASICs) which accelerates design implementation. It accepts a code written in C/C++ and SystemC and converts it directly to an RTL level to be downloaded on an FPGA.

2.1.4 Impulse C

Impulse C [6] is a HLS tool that is used to convert a software code to hardware. It anticipates C code as an input and generates VHDL code. It uses multiple parallel processes for increasing performance. It uses C-language development tools for the debugging and the verification issues.

2.1.5 LegUp

LegUp [2] is an open source high-level synthesis tool that is being developed at the University of Toronto. The LegUp framework allows us to convert C code to Verilog synthesis without building an infrastructure from scratch. They attempt to make FPGA programming easier for software developers. However, LegUp does not support scheduling methods for the shaving of hardware blocks.

2.1.6 The TUDelft tool

This tool is able to convert a .cir code into a VHDL code. The .cir code is a high level code that is implemented based on three state statements. The tool has different scheduling methods which are used to generate a design space of Pareto points. It can also show the DFG of any Pareto points shown in the design space. It does not support floating-point arithmetic and it works only with fixed-point arithmetic. However, this tool is available to us so that we can modify it to support all the arithmetic components involved in our neuron cell design such as the exponent, the divider and the min components.

2.2 Scheduling Techniques

Scheduling is a process of allocating operations (n) to resources (m) over a given period of time. The resources and operations can take many different forms. The resources may be machines, processors, computational units. The operations may be jobs, processes,

tasks, operations or, threads of computer programs. Each operation may have a certain priority level, an earliest possible starting time and a due time. The objectives can also take many different forms. One objective is more related to reducing the delay of the last operation (makespain), which is proven as an NP-hard [9] problem (unless P=NP) when there are more than two resources (m), whereas the other one cares about reducing the number of operations completed after their respective due times. In real-time scheduling (see Figure 2.1) the operations are divided according to their deadline into two kinds:

- operations that have hard-deadline
- operations that have soft-deadline

Scheduling of operations that have soft deadlines are frequently based on extensions to standard operating systems. For example, providing operation and operating system call priorities may be sufficient for systems with soft deadlines.

Scheduling operations that have hard-deadline is divided into:

- Scheduling periodic operations
- Scheduling aperiodic operations

Periodic operations are operations that are repeated periodically after certain time periods. Aperiodic operations are operations that are repeated at variable period of time. Each one, in turn, is classified into either preemptive or non-preemptive operations. Preemptive operations are operations that could be suspended during the execution time, due to another critical operation, and then re-executed later. The non-preemptive operations cannot be interrupted until they complete their execution. Both are classified into static and dynamic operations.

In the next section, we will talk about static, preemptive periodic operations that have a hard-deadline. We are not interested in dynamic scheduling because our application has a fixed amount of operations that never change during the execution time. Also, since our goal is to rescue a damaged part of the brain, then having an implementation that consumes as few resources as possible is mandatory. Having less area consumption means having a smaller design. However, the first implementation is on the FPGA which is a prototype design that will be an ASIC design in the future.

2.2.1 Static Scheduling

Real-time scheduling is a very important problem in architectural synthesis. In such cases, determining the start time of each operation is very important since the sequencing graph (which we want to schedule) shows only dependencies among the operations. So, the start time of the scheduling of a sequencing graph determines the precise start time of each operation. That start time must satisfy the original dependencies of the sequencing graph. That, in turn, could reduce the amount of parallelism of the operations. Many algorithms have been proposed [22] to solve such a problem:

- As Soon As Possible (ASAP)

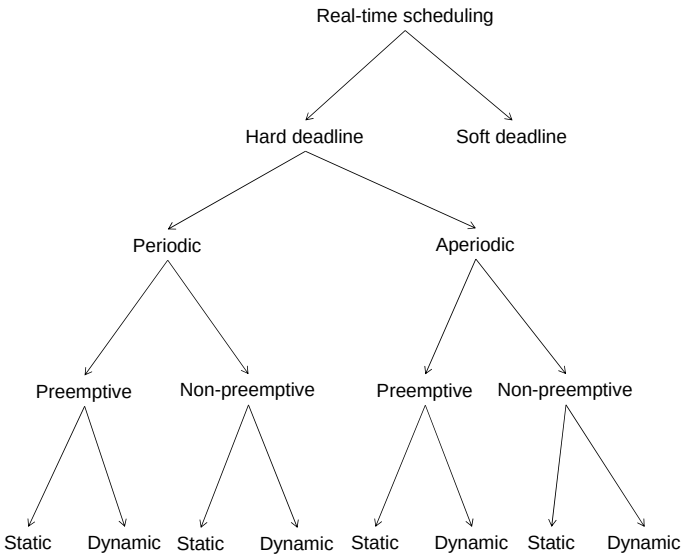


Figure 2.1: Scheduling Classes [32].

- As Late As Possible (ALAP)
- Force-Direct (ForceD)
- List scheduling

ASAP achieves the minimum latency at the expense of, in many cases, needing resources with low average utilization. It uses as many resources as possible and tries to finish them as soon as possible. ALAP uses the Mobility technique which delays some resources so that they are executed as late as possible. Mobility measures how free we are to move the operations into different time-slots. Operations with zero mobility are critical operations, and together form the critical path, which determines how fast our circuit will run. ForceD is a scheduling method that takes information from ASAP and ALAP and generates its result using a special formula known as force.

The basic idea behind List scheduling is to make an ordered list of operations. Such an order is made by assigning some priorities to these operations. After making that list of operations, the following two steps are executed repeatedly until a valid schedule is achieved:

- Select the operation that has the highest priority for scheduling
- Select a resource to accommodate this operation

The priorities are determined statically, even before operation scheduling begins. The first step involves choosing the operation with the highest priority, the second step selects the best possible resource. Some known modifications to list scheduling strategies are:

- Path-based Heuristic operation Scheduling (PHTS)
- Decisive Path Scheduling (DPS)
- Modified Critical Path (MCP)
- Fast Critical Path (FCP)
- Heavy Node First(HNF)
- Critical Path Method (CPM)

PHTS reduces the makespain [14]. DPS improves the priority of node selection criteria in List scheduling algorithms by assigning the priorities to the nodes using the decisive path [23]. FCP reduces the complexity and improves the performance [24]. HNF and CPM improves the performance of multiprocessor by distributing the operations efficiently (operation allocation) [31].

2.3 Neural Modeling in hardware

If we model any application with conventional means like standard CPUs, then that would lead us to an unrealistic model with respect to its real-time (in most cases). In this thesis we use FPGAs to implement a neural model and achieve high performance. The goal of this is:

- Accelerating brain simulation
- Emulating brain function

Software languages (like Matlab and C) or neuron simulator languages (such as NEURON and GENESIS) are used to simulate the brain. Because they are slower than the brain real-time or too slow to simulate large network of neurons, then we need faster hardware implementations.

Such hardware implementations range from the simplest to more complex modeling implementations based on the modeling accuracy demanded and the resource constraints.

The Leaky-Integrate-and-Fire (IaF) model is one of the most basic spiking neuron models in which the action potential is increased with every single input until it reaches a threshold thats allows it to fire and propagate the signal to the next neuron. Ghani et al. [16] have proposed a neuron model for area efficiency. The model was an IaF neuron with 10 synapses per neuron that is implemented on a Virtex II FPGA. Only one neuron model was simulated as an example, to assess the area costs of the model it used 120 LUTs of the FPGA.

Cassidy et al. [12] have reported a Spiking Neural Network (SNN) implementation of an IaF network. 32 neurons were interconnected with 675 weighted synapses. One neuron included STDP learning for its synapses and its running time is 3125 faster than real-time. The network was implemented on a Spartan 3 FPGA by which, the maximum possible network size was estimated at 64 neurons. The network could also be used for

speech recognition besides simulation purposes. Later on, some of the same authors, managed to simulate a million neuron IaF SNN without any synaptic learning, on a state-of-the-art Virtex 6 FPGA at x195 of real-time performance [11]. Most Neural Networks (NN) data were stored on external RAMs in this case.

In 2006, an architecture of a IaF NN using serial execution of arithmetic calculations is proposed by Schrauwen et al. [26]. The network simulated 56 interconnected neurons with exponential decay at 2930 faster than the real-time on a Spartan 3. On a more advanced FPGA (Virtex 4) the maximum network size was estimated at 1400 neurons at 5860 times faster than real-time performance.

A neuron model using a Quadratic IaF model has been proposed by Shayani et al. [29, 28]. That model included multiple compartments that represented the properties of a dendrite and an axon. Each neuron had 16 synapses and the maximum network size was 161 neurons. The system also supported on-line network topology adaptation and its simulation run is 4210 times faster than real-time on a Virtex 5 FPGA. The authors expected to simulate a little above 10000 neurons at real-time, provided that there are enough memory resources. They also proposed, in later publications, models with more advance synapse behaviors and the inclusion of learning algorithms for network topology adaptation [30].

Afterwards, scientists tries to emulate the brain by implementing its sections. Ruben Guerrero-Rivera et al. [18] implemented a Leaky-IaF network model that includes exponential decay and axonal delays. The SNN emulated the Olfactory bulp and had 100 somas interconnected with 675 synapses, implemented on a Virtex II FPGA. The acceleration was almost 2500 times faster than the PC simulation .

Brendan Glackin et al. [17] have proposed a larger network that emulates the function of the Medial Superior Olive in terms of sound localization. The brain uses the time difference of the same sound coming to different ears to establish the general direction of the sound source. The network was implemented on a Virtex 4 FPGA. It simulated 1029 neurons (49 physical models time-multiplexed) and 25221 STDP synapses. It performed in real-time and has very accurate results. The authors estimate that the model could simulate at most 7539 neurons with 1353261 synapses because only few of the FPGA resources were used. A population that is quite close to the biological one estimated around 10-11000 neurons.

More detailed artificial neural implementations can have much more refined behavior. That adds more complexity, which in turn, creates greater limitations in terms of network size and simulation speed. There were some attempts to create FPGA implementations for conductance models in recent years, but more limited in number than with simpler models.

C.I. De Zeeuw et al [34] proposed diagram of the brain circuitry controlling sensory-motor activity. The main role in the model is believed to be played by the inferior olive, the cerebellar nuclei and the cerebellar cortex. In order to have the complete olivo-cerebelar model, we have to implement the cerebellar nuclei and the cerebellar cortex. In this thesis, we have implemented the inferior olive (IO-cell). These three components are linked to each other through connections in the brain and they form a loop of three elements as what as shown in Figure 2.2. In this loop, we have reverberating loops, theory which the motor activity can be controlled. The Neuroscientists think that activity in

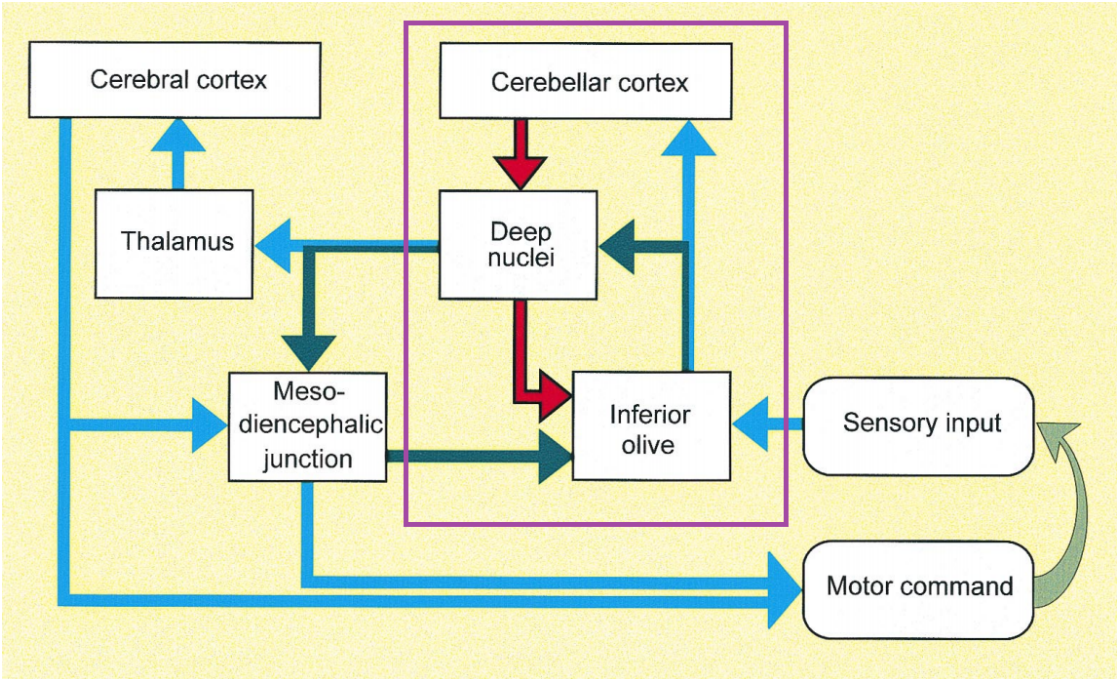


Figure 2.2: The three-element network (dark blue) composed of the mesodiencephalic junction, deep cerebellar nuclei, and inferior olive forms an important loop in sensorimotor control. Red arrows indicate inhibitory pathways and (light and dark) blue arrows mark excitatory systems [34].

this brain section will control sensory motor integration and will control particularly the right timing of movement. Since timing is essential, we want to have a model in which we can mimic the activity in the olivo- cerebellar loops with realistic timing components. This means that if a particular process takes X milliseconds in the brain to occur, it should take the same time period in our model. If we model the olivo- cerebellar system with conventional desktop computers, then we cannot model, at the same time, enough neurons accurately while keeping the timing at the temporal aspects realistic. Therefore, we will develop an FPGA-based model in which we will be able to keep the timing as realistic as possible.

2.4 Summary

In this chapter we described a number of HLS tools presented a recent of research works on the field of scheduling and neuron modeling in general. In Chapter 4 we will show how we involve the HLS tool to optimize our neural implementation.

3 Background

"A good short-story writer has an instinct for sketching in just enough background to ground the specific story."
-Lynn Abbey

In Chapter 2 we explained the nature of brain model research and the motivation behind the work presented in this thesis. Naturally, there are a number of concepts that one needs to be familiar with when tackling this problem. This chapter serves the purpose of presenting the concepts that are needed to understand and follow the content of the work described in this document.

This chapter is organized as follows: Section 3.1 gives a brief description of the neurons and the brain. In Section 3.2, the Neural Models are presented whereas the Inferior Olive is presented in Section 3.3. Finally, the chapter concludes with a small summary.

3.1 Neurons and the Brain

The nervous system, depicted in Figure 3.1, is an organ system that contains a network of specialized cells called neurons. Neurons work and communicate based on electrical and chemical signals. These neurons coordinate different actions that we do by transmitting signals between different parts of our body. It consists of two parts:

- Central
- Peripheral

The peripheral nervous system consists of sensory neurons that link sensory receptors on the body surface or deeper within it with relevant processing circuits in the central nervous system. Different activities like thinking, dreaming, remembering, feeling pain or pleasure, moving, etc, are achieved thanks to the central nervous system. The central nervous system consists of two parts:

- Spinal cord (Glia)
- Brain (neurons)

The brain is part of the central nervous system and its function is to control other organs of the body by generating patterns of muscle activity or driving secretion of chemicals called hormones.

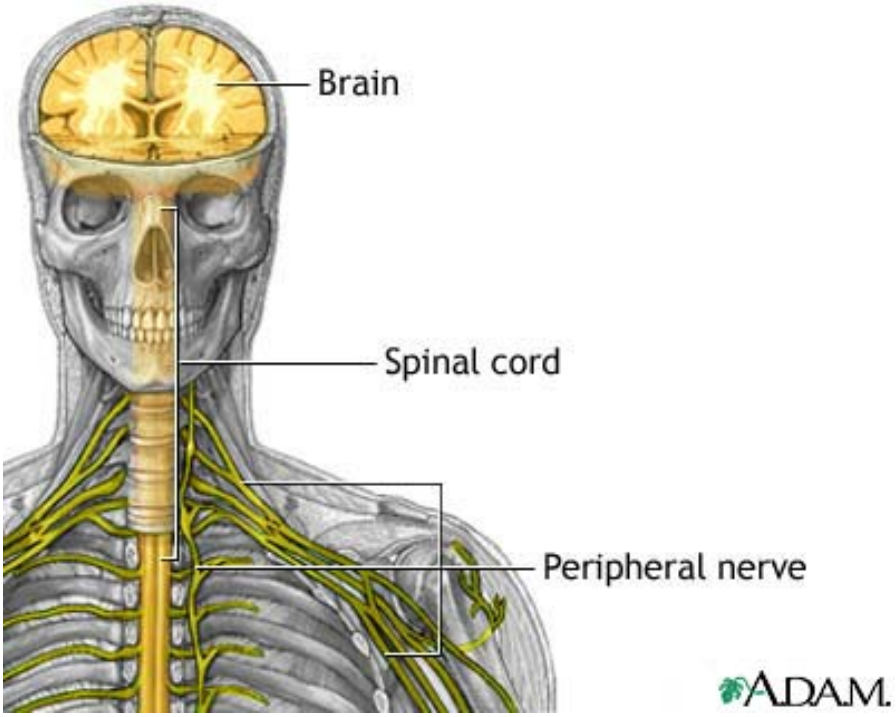


Figure 3.1: Nervous System [1].

3.1.1 Neuron Cells

Cells are the building blocks of life. They are the basic smallest functional living tissues that form life. The Human body contains a hundred distinct cell types and it contains around 50 up to 75 trillion cells in total. These cells are doing different tasks. There are many different cell sizes but a typical cell size is about 10 micrometers and its mass is around 1 nanogram. Each cell has its own membrane across which it can exhibit a voltage difference.

Neurons are a special type of cells that extend form the brain. A neuron is the basic information processing structure, also called nerve cell. It processes electric signals by receiving signals, impulses, and transmitting them as outgoing signals via its axon, which is a cable-like extension to other cells or, to a muscle cell. In general, a neuron cell is divided into three main compartments (see Figure 3.2) that are:

- Cell body (also called soma)
- Dendrites
- Axon

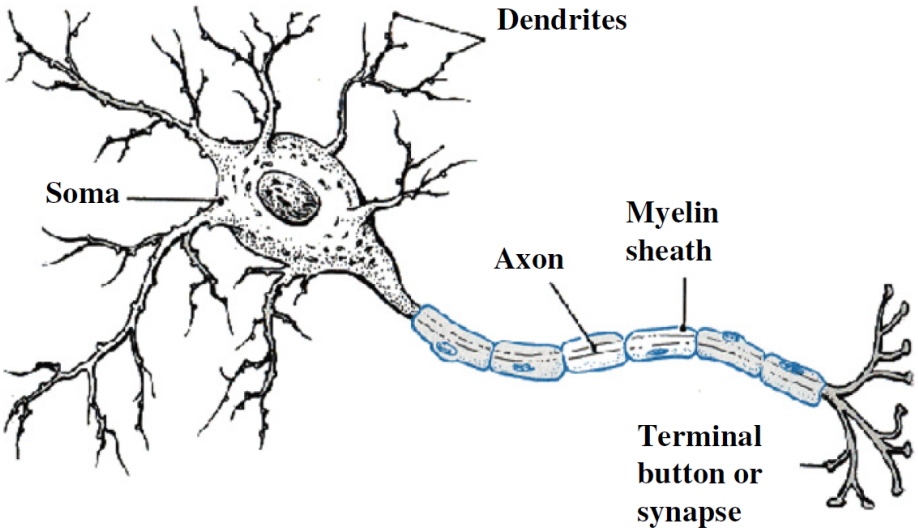


Figure 3.2: Basic illustration of a biological neuron [10].

Soma

The soma (Greek word for body) is the largest part of the cell that is often called the cell body. It holds the general parts of the cell as well as the nucleus which is the control center of the cell. The nucleus contains the DNA which is located in the chromosomes.

Dendrites

Dendrites are projections from the cell body. They receive electric signals from other cells and combine these signals into the soma. Dendrites have a unique structure. They are the largest components with respect to both surface area and volume of the brain. Their specific form helps to classify neurons into classes such as: Pyramidal, Purkinje, Amacrine and Stellate.

Axon

The axon is the extension part from the soma to the terminal endings of the nerve. Its function is to transmit impulses from the soma to another dendrite and/or to a muscle cell.

Synapses

A synapse is a small structure at the end of two cells. It is the linking part that passes the electrical or the chemical signals between two cells, connecting an axon terminal with a dendrite.

3.2 Neural Models

Since the 19th century, scientists and engineers started studying the brain and its mechanisms. The first mathematical model of a Neural network that simulate the brain behavior was presented [21] in the beginning of 1940s. That model opened the door for further study of such networks and to crate more advanced artificial models that mimic the biological behavior of the brain. Many solutions of current computational problems have been proposed such as the perceptron [25] and other works on connectionist networks.

Advances in engineering and neuroscience led to the creation of mathematical models of networks that emulate biological behavior of the brain. The *Spiking Neural Network* (SNN) [33] is an example where information is not just transfered by the firing rates of each neuron in the network, like in classical Artificial neural networks (ANNs), but by the transfer of spikes. That helps us model much more advanced models and encode richer information. The advances of technology in computer science and engineering has supported such research and opened up the possibility of implementing larger scale NNs that have the ability to emulate brain behavior.

3.2.1 The Hodgkin-Huxley Model

The electrical properties of the nerve membrane segment can be modeled as an electrical circuit according to Hodgkin-Huxleys (HH) [19] model. Figure 3.3 shows the HH circuit in which current flow that passes through the membrane has two main components, one associated with charging the membrane capacitance Cm and one associated with the ionic movement across the membrane. This ionic current is divided into three components and that are:

INa sodium current

IK potassium current

IL leakage current, carried by chloride ions

Furthermore, the behavior of the circuit of Figure 3.3 can be described with the following equation:

$$Cm\frac{dVm}{dt} + Iion = Iext \text{ [5]} \tag{3.1}$$

where:

- Cm is the membrane capacitance
- Vm is the intracellular potential (membrane potential)
- Iion is the net ionic current flowing across the membrane
- Iext is an externally applied current

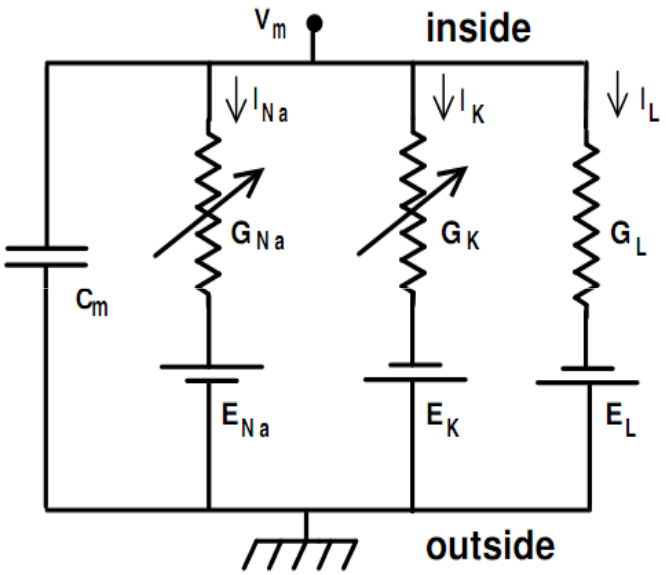


Figure 3.3: Hodgkin-Huxley model [5].

Iion on the left-hand side of Equation 3.1 has an opposite sign than Iext on the right-hand side. According to Equation 3.1, a positive Iext will tend to depolarize the cell (i.e., make Vm more positive) whereas a positive Iion will tend to hyperpolarize the cell (i.e., make Vm more negative). We use the term inward negative to refer to the inward flow of positive ions into the cell which is considered to be a negative current. This term comes from the fact that when we study an ionic current in a voltage clamp experiment we actually measure the clamp current that is necessary to balance it. Thus an inward flow of positive ions is observed as a negative-going clamp current, hence explaining the inward negative convention.

The membrane potential (Vm) is the difference in voltage between the inside and the outside environment. The potentials are always relative that means the potential differences can be measured directly. We may assume that the inside voltage is equal to zero or the outside is equal to zero. In either case, the difference of that potential should always be around -70 mV. Hence, the absolute value of the potential difference across the membrane is the same, thus the only matter is how zero is defined.

3.2.2 Compartmental Models

The compartmental model is a model that emulates the neuron cell. It has been divided into a finite number of connected compartments. Figure 3.4 (a) shows a pyramidal cell where we can see the tree-like structure of the dendrites, synaptic endings and soma. The cell receives impulses via its synaptic ion channels which creates postsynaptic potentials

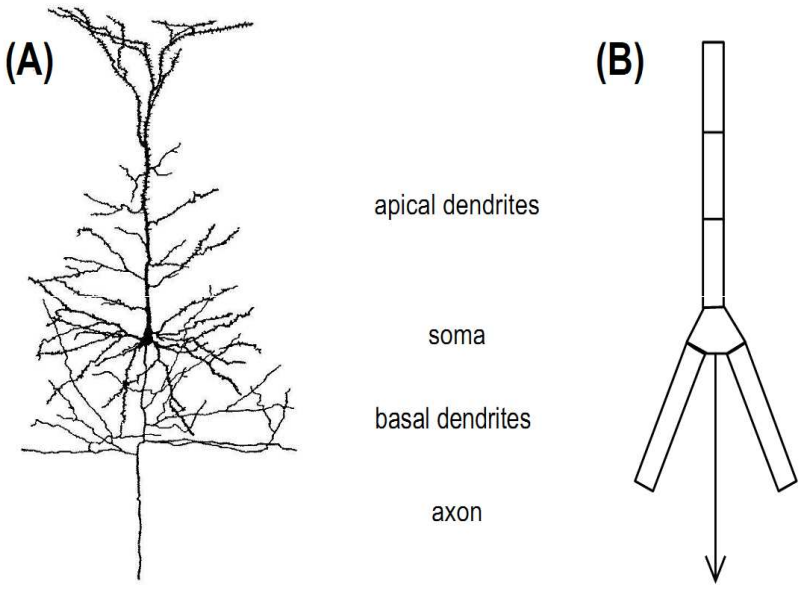


Figure 3.4: Compartmental model [5].

that are propagated into the soma where voltage-activated ion channels may create action potential. These ion channels are connected in a region called axon hillock near the base of the soma. Figure 3.4 (b) shows a simplified electrical model that emulates the pyramidal cell. Each compartment of that model is represented by equations describing an equivalent electrical circuit. Then we can model the behavior of each compartment by using these specific equations.

3.3 The Inferior Olive

It is the biggest part in the cerebellum that consists of a complex group of cells. The cerebellum is the part of the brain that is involved in the motor control and some other functions such as attention and pleasure responses. The cerebellum does not initiate movement (but it coordinates its precision and accurate timing). It receives input from sensory systems of the spinal cord and from other parts of the brain, and integrates these inputs to fine tune motor activity [15].

C.I. De Zeeuw et al [34] studied the cerebellum motor activity of the brain in which the essential part is timing. We want to have a model in which we can emulate the activity in the olivo-cerebellar loops with realistic timing components. That means if a particular process takes about 30 milliseconds or 100 Milliseconds in the brain to occur, it should take the same time periods in our model. If we model the olivo-cerebellar system

with convention means like normal CPUs, then we think that we cannot model enough models realistically and keep the timing at the temporal aspects realistic. Although, scientists do not have much information about the inferior olivo-cerebellar function, it is commonly associated with the cerebellum activity which let us conclude that it is involved in motor coordination and motor learning. Therefore, we will try to use an FPGA based model in which we will be able to keep the timing as realistic as possible. In this work, based on Hodgkin-Huxley ion channels, we are interested in modeling the Inferior Olive (IO) Network.

3.4 Summary

In this chapter, the basic background required for the presentation of our work was provided. This included a small presentation of the neurons and the brain framework in which our work can be linked with. Next, we presented the basic mathematical model that emulates the neurons biological function. Finally, the previous section showed the part of the brain that we are interested in to start with it in this thesis. With the necessary background provided, the next chapters are devoted to presenting the design challenges in modifying the HLS tool to achieve the ability of converting the sequential code of the IO-cell into a parallel implementation to be downloaded on the FPGA. We will also present different evaluation metrics that help us to do our analysis.

Application Description and Tool-flow

4

"Production is not the application of tools to materials, but logic to work."
-Peter F. Drucker

In this thesis, we are interested in implementing an Inferior Olive model on an FPGA. In the first stage, we implement only a single cell, after that we implement two IO cells and in the end four of them. As it was mentioned before, the goal of this thesis is to come up with two designs that either let us simulate as many IO cells as possible at brain real-time or, simulate a fixed number of IO cells as fast as possible. We have used the TUDelft tool, which is an automated means that helps us schedule the design and converts a suitable point from the design space into synthesizable VHDL. In order to do that we had to modify the TUDelft tool and create our own version of it capable of handling specific features of the model.

This chapter is organized as follows: In Section 4.1 we present the Inferior Olive (IO) network model. The Scheduling, the TUDelft tool and the limitations we found will be presented in Section 4.2. Finally, we explain the new capabilities of the tools after the modifications that we applied.

4.1 The Inferior-Olive Network Model

The Inferior Olive is the largest nucleus situated in the Olivary body, part of the medulla oblongata. It is closely associated with the cerebellum, meaning that it is involved in control and coordination of movements, sensory processing and cognitive operations, likely by encoding the timing of sensory input independently of attention or awareness. The model used in this thesis has been developed by Jornt R. de Gruijl, a PhD researcher at the Netherlands Institute for Neuroscience (NIN). His design [13] is originally based on the design of N. Schweighofer [27]. The cell that we want to build has all the three compartments that we have mentioned in Chapter 3. The IO cell is composed of three compartments:

- Dendrites
- Soma
- Axon

As explained earlier, the Dendrites represent the input connections to the cell body (Soma). Dendrites have an input current coming from other Deep-Cerebellar-Nuclei (DCN) cells. Furthermore, neighboring IO-cells interact through connections between their dendrites, called gap-junctions (see Figure 4.1). In the IO-cell model we are using, the numerous dendrites are modeled as a single compartment. As we explained in

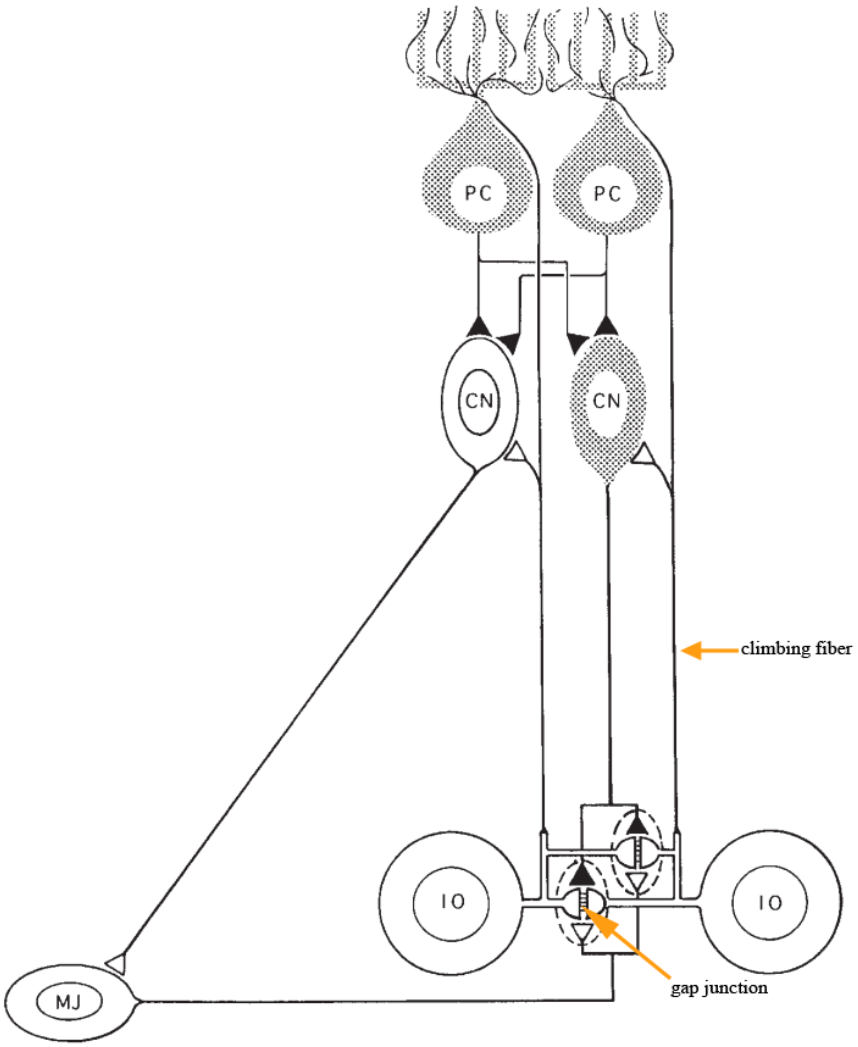


Figure 4.1: Microcircuitry of the cerebellum [34]. MJ: mesodiencephalic junction, CN: cerebellar nuclei, IO: Inferior olive, PC: Purkinje cell.

Chapter 3, the Soma is the neuron-cell body. Through electrochemical reactions, the soma transforms the input signal from the Dendrites into a response signal through the Axon. The Axon represents a single output connection of the cell that transmits the electrochemical-output signal to other cells. An Axon typically terminates in a connection (i.e. a synapse) with a Purkinje-Cell (PC) through the so-called Climbing Fibers (CFs). An IO-cell Axon does not form a synapse with another IO-cell but with PCs in the cerebellar cortex.

The communication in the IO-cell model can be categorized as:

- External communication with the DCN, the cerebellar cortex and neighboring IO-cells

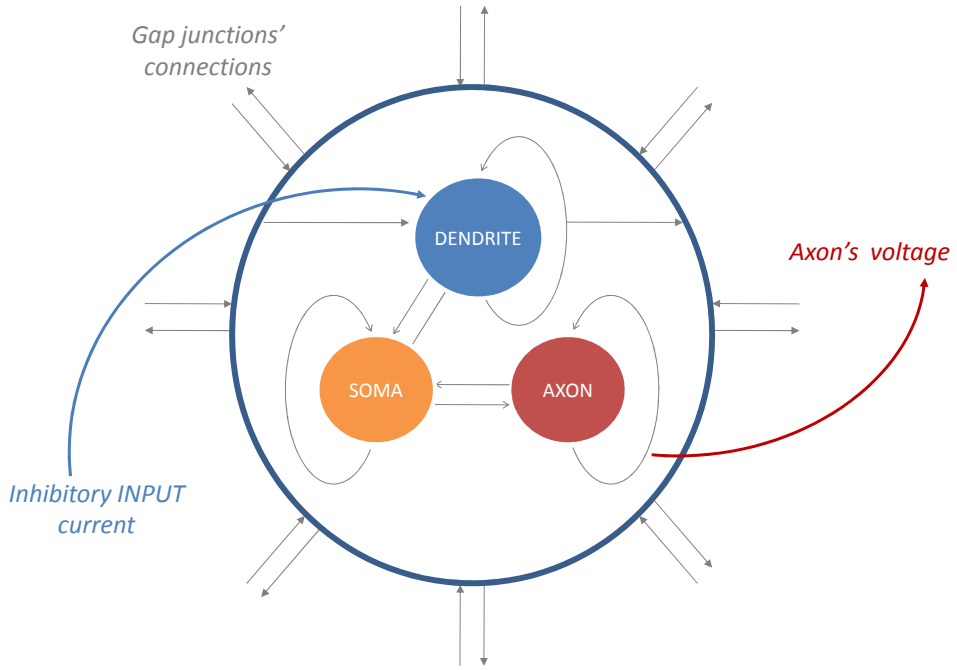


Figure 4.2: The IO-cell with its three compartments [Courtesy: S. Isaza, Erasmus MC].

- Internal communication between the dendrite, soma and axon compartments

Figure 4.2 shows these communication channels as well as the three cell compartments. Inside the IO-cell, the first compartment propagates the input to the second one which in it turns, processes and propagates that input to the last compartment. Each IO-cell compartment further breaks down into a number of mathematical functions which implement the functionality of that compartment. Each function contains different computational units (operations) which we are interested to implement. These operations need to be executed by Exponents (EXPs), Dividers (DIVs), Multipliers (MULs), Arithmetic Logic Units (ALUs) and the MINimum operation (MINs).

As we explained in Chapter 3, the cell state is defined as the set of variables that characterize the three compartments of a particular cell at a given time, including channel conductances, voltages and currents. These variables exist for each of the cell compartments and include, for example, the Potassium and Sodium channel conductances. The cell state is used to compute both the inter-compartmental currents and the next cell state.

Multiple IO-cells form a network of cells. The network of IO-cells could be seen as a two-dimensional (2-D) mesh, as depicted in Figure 4.3. Every mesh node receives an electric current from the DCN and, after computing the electrochemical reactions, outputs a voltage value. As it is depicted in Figure 4.1, IO voltage outputs go to the CFs that connect with the PCs in the cerebellar cortex. Furthermore, neighboring nodes

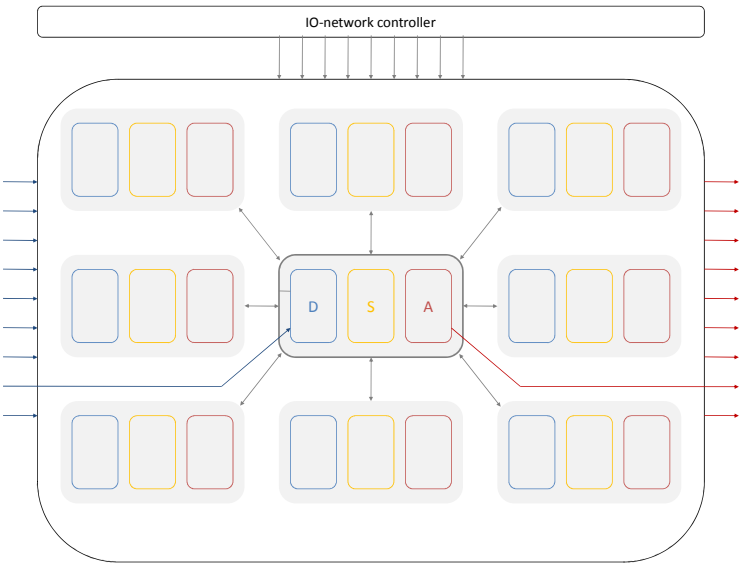


Figure 4.3: The 2-D mesh of the IO-Network [Courtesy: S. Isaza, , Erasmus MC].

affect each others electrical behavior. The computational model we are using works in a synchronous fashion: In every system cycle, all mesh nodes (i.e. IO-cells) compute their output and their new state based on the current inputs and their previous (internal) state. To ensure that all network updates are performed in a harmonious pattern, an IO-network controller is included to guarantee this. Such a model is very complicated and requires a lot of computations; so we need high performance computation. The simulation we are using is assuming a timing accuracy of 1 sample every 50 microseconds. This means the computation has to be so fast that it should be able to calculate 1 simulation unit every 50 microseconds which is the brain real-time deadline.

4.2 Scheduling and the TUDelft Tool

The main challenge that needs to be addressed in our design is the scheduling of the model operations. Scheduling means assigning each operation on a certain resource at a certain time. As we saw in Chapter 2, there are many scheduling algorithms available like ASAP, ALAP, ForceD and List. List is a very interesting algorithm and suits our specific application as we have shown there. This kind of algorithm makes a heuristic search in the design space. In other words, the resource allocation face is explored exhaustively and then it is fed to the list scheduling algorithm, which is a heuristic algorithm, to explore all these different permutation of resources.

Scheduling manually is an option but it is infeasible because of the huge number of resources needed. So, we need an automated tool that schedules these operations. Ideally, we also need a tool that generates VHDL code for this scheduled design that we chose. Moreover, this tool should support floating-point arithmetic because our

application was built based on that. Also, by supporting floating-point arithmetic, we will be able to validate our VHDL implementation by comparing its output with the original implementation output which is also written in floating-point math.

As we explained in Chapter 2, there are a number of tools that do scheduling and generate VHDL code like the Catapult C Synthesis tool [3], the HercuLeS high-level synthesis tool [7] and others. However, these tools do not support all the operations that form our application like the Exponent (EXP) operations. Moreover, the source code of these tools are not commercially available to us to be modified or expanded as we have explained in that chapter. While the option of building an HLS tool from scratch was beyond the time budget of a master thesis, we also learned about the TUDleft tool, which offered us most of the features we were looking for.

The TUDelft tool is a high level synthesis tool that is used to design HW circuits efficiently. This tool is implemented in Matlab and has multiple functions. It starts from a high level description of the design and expects a .cir file as input, whereas generates a VHDL code as an output. The .cir file is a special code that is written in a three-state-statement format which means each line has one output and two inputs where:

- Inputs start with an 'i' and can be at the left (internal input) or right (external input) hand side of an assignment;
- Outputs start with an 'o' and are either a result of an operation (external output) or directly connected to an input (internal output);
- Constants start with an 'a' or a 'c';
- Outputs of the sequencing graph (SG), which is called scheduled sequencing graph (SSG) after scheduling, that are registered and fed-back to its inputs are preceded by a 'T';

The tool takes this code and tries to schedule its operations using different scheduling methods. Different scheduling methods are evaluated in this tool like: ASAP, ALAP, forcD and LIST. Figure 4.4 shows the structure of the TUDelft tool. It expects a .cir file and generates VHDL code after performing an efficient implementation based on the built-in scheduling methods. We then simulate this VHDL code by using ModelSim. After that, we synthesize this implementation by using Xilinx. Once we are done, then it is the time to download the implementation on the FPGA.

4.2.1 Scheduling Methods

The scheduling methods (ASAP, ALAP, ForceD and List) that the TUDelft tool uses are presented in the following subsections.

ASAP

As-Soon-As-Possible is one of the scheduling methods that does not have resource constraints, and thus, tries to schedule operations as soon as possible, using as many resources as it needs.

ALAP

Since ASAP schedules operations as soon as possible then it is equally possible to schedule operations at the latest opportunity by knowing the overall latency constraints. This leads to the concept of As-Late-As-Possible (ALAP). ALAP is another scheduling method that also does not have resource constraints, but, tries to schedule operations as late as possible. ALAP calls ASAP and builds its work based on the information provided by ASAP. ALAP, sometimes, generates better results than ASAP because it uses the so-called Mobility technique. The mobility technique measures how free we are to move the operations into different time slots. Operations with zero mobility are critical operations that we cannot change their execution time. Such operations form together the critical path which determines how fast our design will be. When we have a positive mobility then we could balance the resource over time by letting them to be executed as late as possible. ALAP scheduling can be performed by seeking the longest path between each operation and the last node.

forcD

Another scheduling method is forceD that consists of three steps:

- Determine a time frame of each operation
- Create a distribution graph
- Calculate the *force*. The force is a metric used to optimize the utilization of the resources. A high positive force value indicates a poor utilization.

Note that forceD does not schedule resources that need more than one clock cycle in order to execute.

LIST

The List scheduling algorithm stores operations in a decreasing ordered list, based on its length. Then, it loads the first operation in that list on the available resource. The loading time of the operation is called the starting time of that operation and, once it completes its execution, we will know the completion time of that operation. In this method, we move cycle-by-cycle through the schedule by choosing operations from the list and schedule them on the available resources and updating the list for the next cycle.

Internal Tool Structure

All of the previous scheduling functions are scheduling statically and are located in the scheduling part of Figure 4.4. Resource constraints are applicable in this tool. These scheduling methods are implemented in such a way that they can generate a result with or without design constraints. In the case that we do not specify any design constraints, the tool automatically sets a constraint of 1 for each of the resource number or its latency.

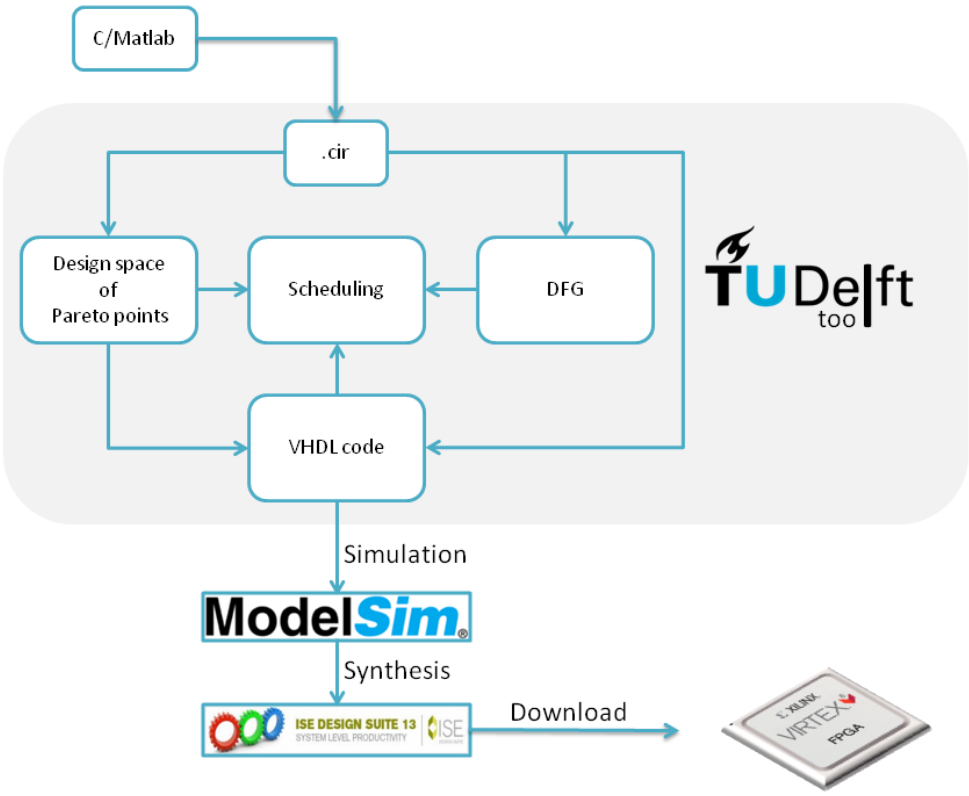


Figure 4.4: The TUDelft tool.

As is depicted in Figure 4.4, the scheduling part is located in the middle of the tool-flow and all the other parts of the tool call it, thus, performing their work based on it. The tool translates the design (.cir-file) into a Data-Flow Graph (DFG). Different functions are used to generate such a DFG. The DFG that is drawn by the tool consists of operations, delays and interconnect. Figure 4.5 shows the DFG of our application which is drown based on ASAP.

View_DFG() and ShcedGUI() are two main functions that are used to generate such DFG. By using the former, we are not able to choose the scheduling method that will be used to draw such a graph because ASAP is already hard-coded as its scheduling function, whereas by using the latter we are able to specify the scheduling method via its graphical interface.

The other important part of Figure 4.4 is the part that generates a design space of Pareto points. The Xplore() function is used for that and it checks every single point that is generated by the schedule part to generate that design space, containing all the possible Pareto points. This part is very important because all the analysis that we will do is based on it. This design space, that the function generates, is a two dimensional (2-D) plot that contains two objectives. These are the area and the performance objectives. It also shows the resources that are used to produce a specific Pareto point such as MULs,

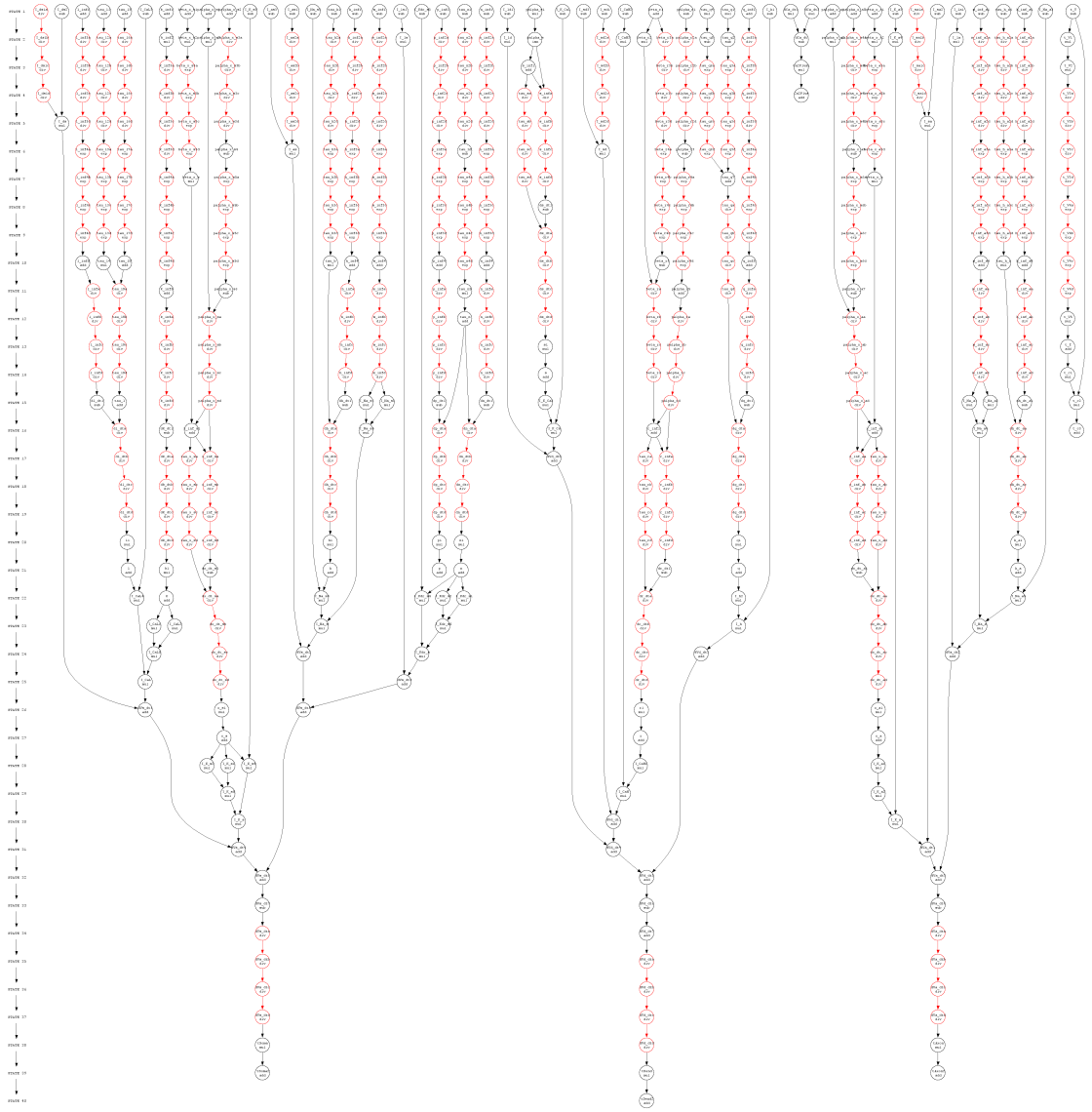


Figure 4.5: Data-Flow Graph (DFG) based on ASAP.

ALUs and REGs.

This tool has yet another feature which is checking whether retiming can generate a faster design. The function that is used for that is called retime_minCycles() and it uses registers as delay elements. Once the design is able to be retimed, the function forwards these delay elements and then combines them to get a faster design. Figure 4.6 shows the retiming process in which (a) shows the forwarding process of the delay elements whereas, (b) shows the combining process of them.

Still using the tool, we are also able to simulate the scheduled design and calculate its output values depending on user-definable input sequences (these values can be used later on to verify that your hardware simulation operates correctly) by using the gen_mT()

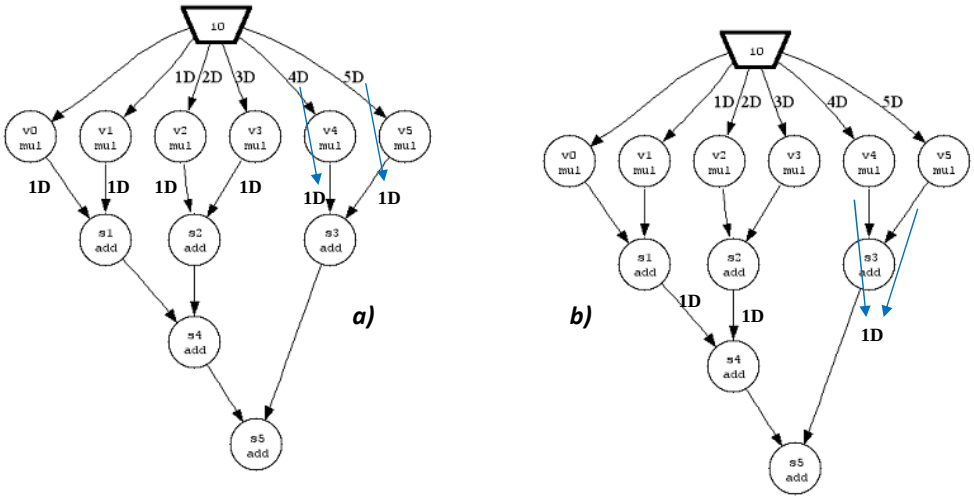


Figure 4.6: Data-flow graph obtained with view_DFG(). 1D represents one Delay Element, 5D means 5 consecutive delays.

function.

Having said that, the tool generates VHDL code for the Pareto point that we choose from the design space. The generated VHDL code contains three parts:

- Data path
- controller
- Testbench

Gen_VHD() is used for generating that code. This code will have to be simulated with the aid of a VHDL simulator like ModelSim. Evidently, the outputs of both Matlab and the VHDL testbench should be in accordance with each other. The tool has also a built-in fixed-point library that contains an Arithmetic logic unit (ALU), a Multiplier (MUL) and a Register (REG). During its running, gen_VHD() generates two of plots, in which it shows the mapping of the operations on the given number of MULs, ALUs and REG (see Figure 4.7).

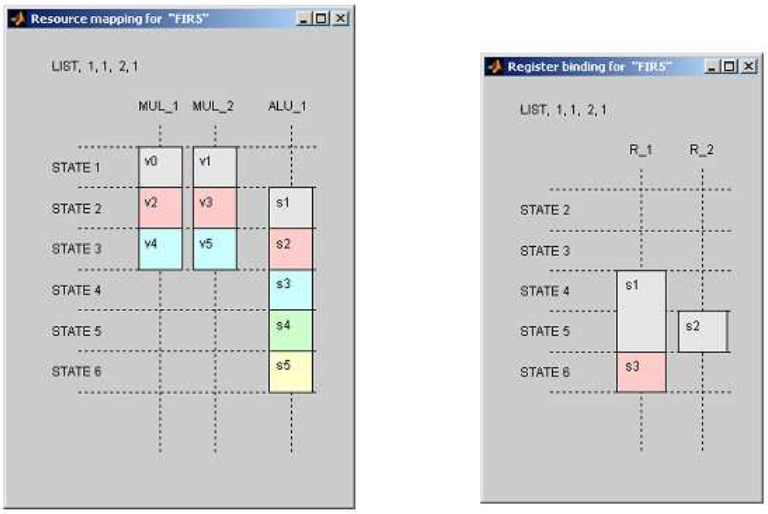


Figure 4.7: Resource mapping and Register binding as reported by gen_VHD() function.

4.2.2 Limitations

Our application has more types of operations than what the tool supports, such as the Exponent (EXP), the Divider (DIV) and the Minimum (MIN). This means that, unless we add the missing operators, our IO-model cannot be processed by the tool. Moreover, our application requires floating-point arithmetic, not supported by the tool either.

4.2.3 Modifications

In order to overcome the limitations explained, in this section we describe the work done in extending the tool capabilities. First of all, we divide the functions into two categories:

- Main functions
- Sub functions

Main functions are functions that we can directly run from the Matlab window of the TUDelft tool. These functions will call the sub functions whenever they need them. Thus, the main functions call the sub functions implicitly. Moreover, each function might have its own local function(s). Figure 4.8 shows the function dependency structure of the TUDelft tool.

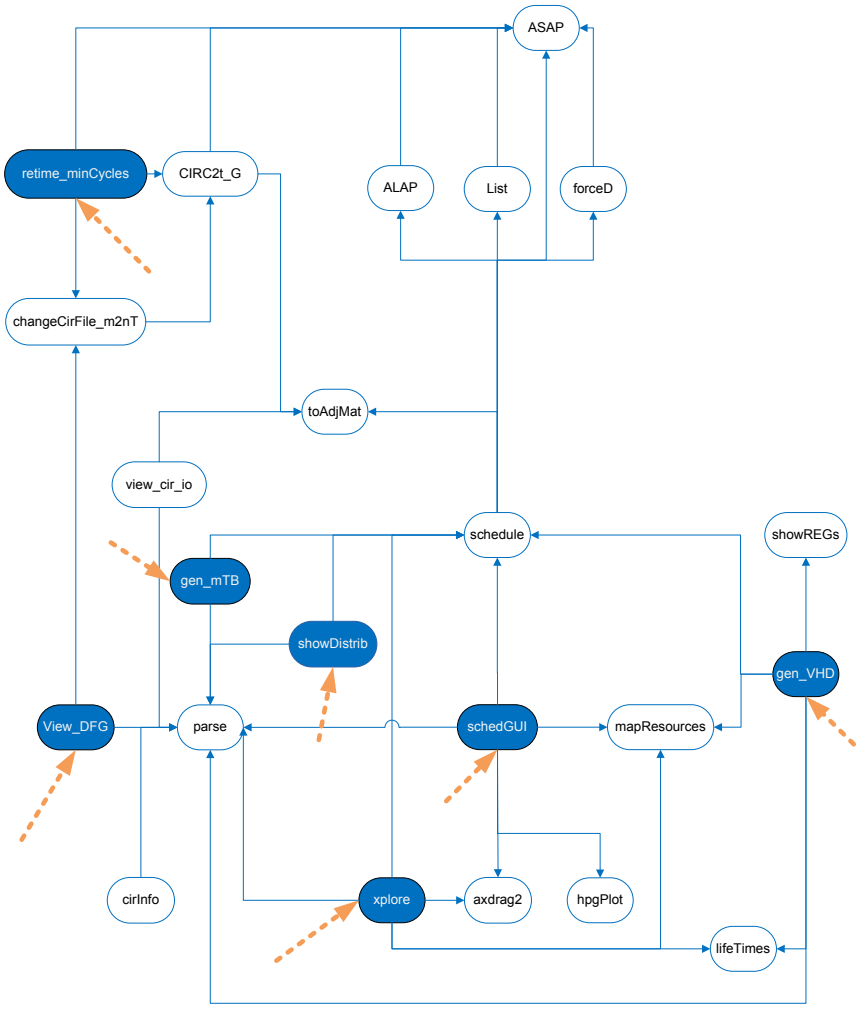


Figure 4.8: Functions dependency graph.

The tool is available for us, thus, we could expand it and add the new features that we need. Firstly, we had to expand the number of resources that are involved in the tool to make the tool suitable for all the operations that the application has. Starting from the .cir file, Figure 4.9a shows that file before the modification, whereas after the modification the new .cir file will look like as it is shown in Figure 4.9b. The new .cir file consists of all the new resource types as it is shown in the figure.

```
% operations
v0 = c0 * i0;
v1 = c1 * i1;
v2 = c2 * i2;
v3 = c3 * i3;
v4 = c4 * i4;
v5 = c5 * i5;

s1 = v0 + v1;
s2 = v2 + v3;
s3 = v4 + v5;
s4 = s1 + s2;
s5 = s4 + s3;

% output node
o5 = s5;
```

(a) .cir file of FIR5 before modification.

```
% operations
v0 = c0 - i0;
v1 = c1 / i1;
v2 = c2 ? i2;
v3 = c3 $ i3;
v4 = c4 * i4;
v5 = c5 * i5;

s1 = v0 + v1;
s2 = v2 + v3;
s3 = v4 + v5;
s4 = s1 + s2;
s5 = s4 + s3;

% output node
o5 = s5;
```

(b) .cir file of FIR7 after modification.

Figure 4.9: .cir files before and after modification, in which $ is EXP, / is DIV, * is MUL, +/- is ALU and ? is MIN.

We also reduce the scheduling time by letting the List scheduling run until a certain boundary that has been specified in advance. This boundary is based on the longest critical time of ASAP. Moreover, we improved the plotting by removing all the points that are not Pareto points (see Figure 4.10) and only keeping the Pareto points to make it more efficient and easier to read (see Figure 4.11). However, we reduced the font size in order to read all of the points.

Afterwards, we modified the schedGUI() function and redesigned its GUI graph interface in order accommodate the distribution window of the new operations that we added. Figure 4.12a shows the old gui whereas Figure 4.12b shows the scheduled version of the new .cir file (FIR7) after modification.

We have also modified the format and the number of the resources of the built-in library from fixed-point into floating-point format. The new tool can do Exponent (EXP), Division (DIV), Multiplier (MUL), Arithmetic Logic Unit (ALU) and Minimum (MIN) along with the Register (REG) compared to the old version that has only Multiplier (MUL), Arithmetic Logic Unit (ALU) along with the Register (REG). Figure 4.13 shows the resource mapping after the modification which shows more resources in comparison with the old version that is shown in Figure 4.7.

4.3 New Capabilities

The modifications are summarized as follows (see Appendix A for more information):

- Added new Functional units (FUs)

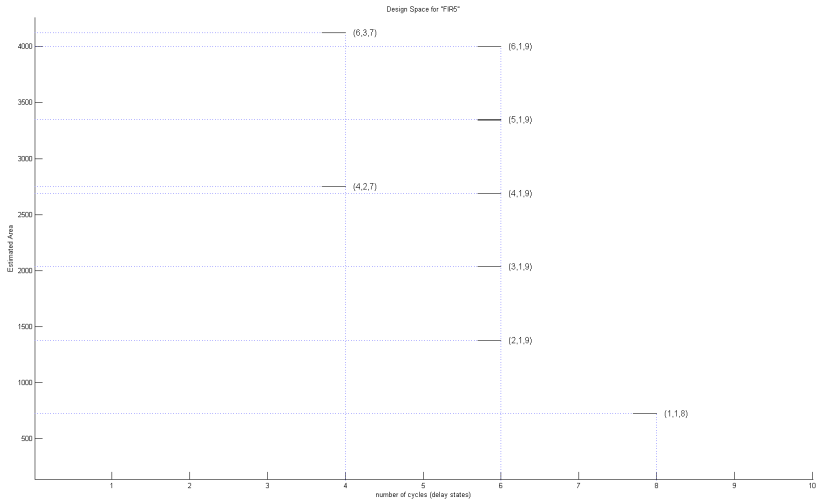


Figure 4.10: The design space of the FIR5.cir that is plotted by the old Xplore() function.

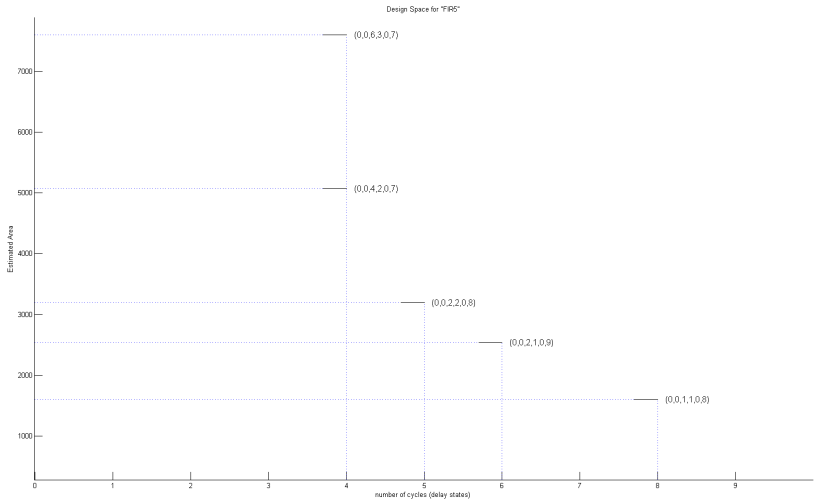
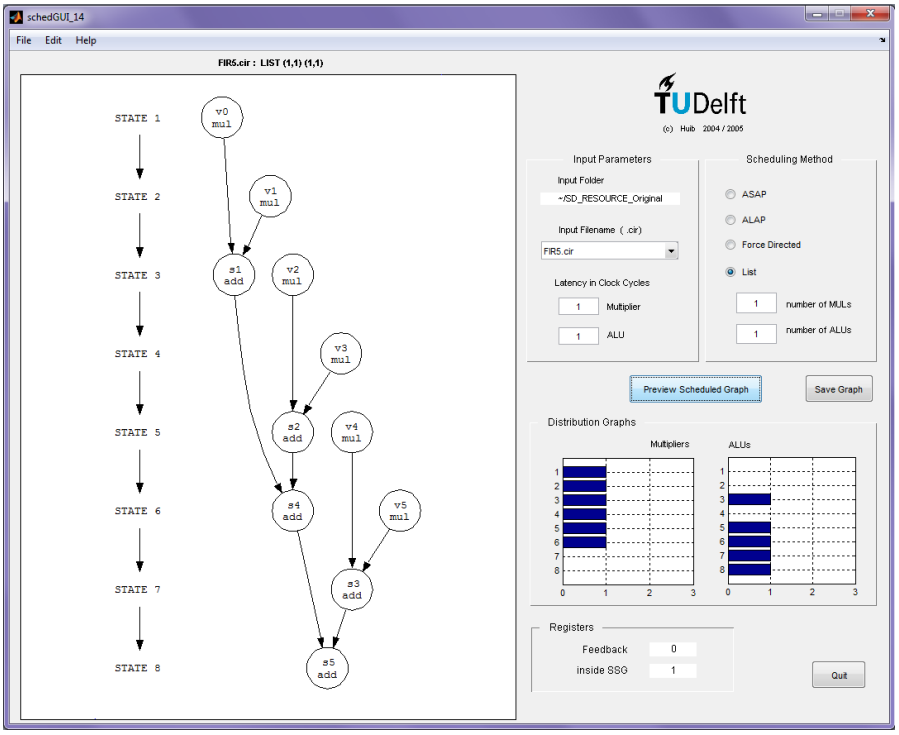
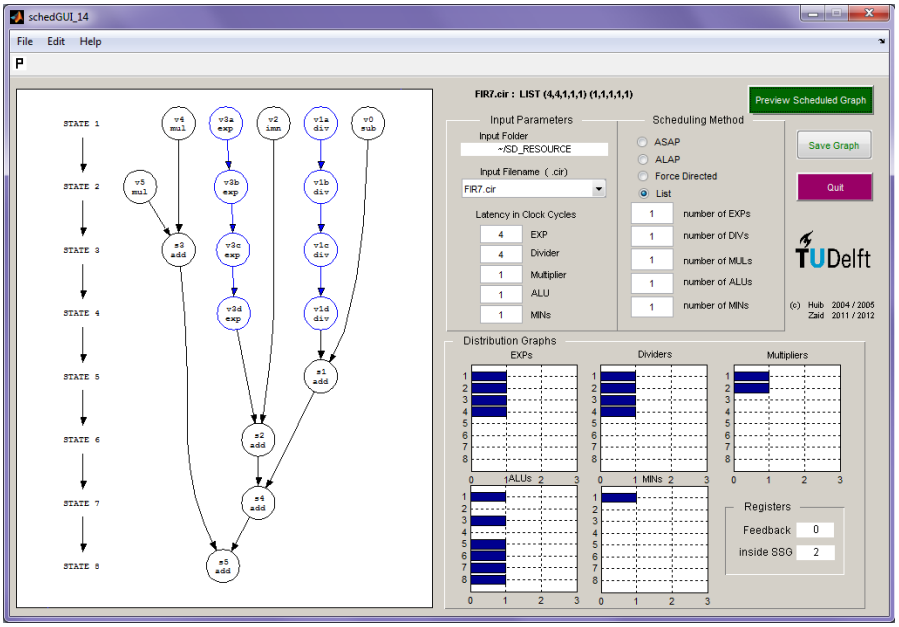


Figure 4.11: The design space of the FIR5.cir that is plotted by the new Xplore() function.

- Reduced scheduling time
- Improved plotting by removing all the points and leave only the Pareto points to make it easier to read
- Expanded tool GUI
- Added floating-point support



(a) Screenshot of the schedGUI() function showing the Scheduled Sequencing Graph (SSG) for FIR5 when a resource constraint List scheduling method has been applied before modification.



(b) Screenshot of the schedGUI() function showing the Scheduled Sequencing Graph (SSG) for FIR7 when a resource constraint List scheduling method has been applied after modification.

Figure 4.12: Screenshot of the schedGUI() function before and after modification.

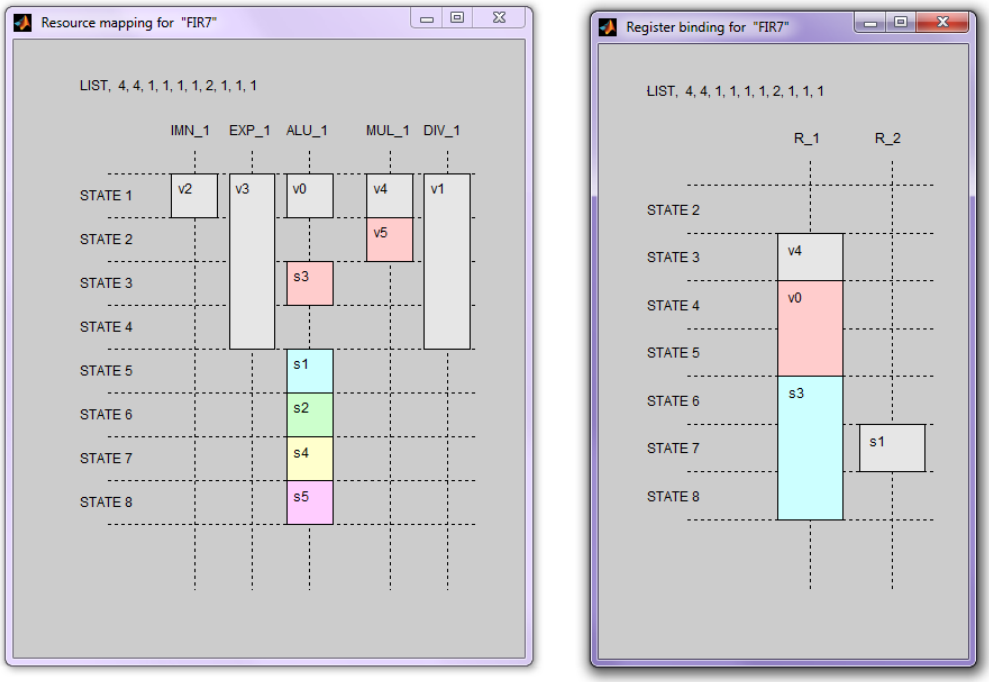


Figure 4.13: Resource mapping and Registry binding as reported by gen_VHD() function after modification.

Based on the previous discussion, we will see the implementation details of our application in Chapter 5 and how we are going to extend the work to multiple cells. We were able to synthesize a single cell, two cells and four cells. Since the tool will not wrap two or four cell together, we had to do some extra modifications in order to wrap two or more cells together. In that wrapper, when we wanted to instantiate two cells, we instantiate the cell component twice (which called IOCellFunctionCa_auto.vhd) and then interconnected their inputs and outputs together. The same with four cells. Such a network could give us more information about how the network of cells could work, what the area usage and so on. Moreover, the TUDelft tool become able to print the information and the computation results of the objectives and the cost functions that we will explain in the next chapter.

4.4 Summary

In this chapter we described in detail the modifications we performed in the TUDelft tool that we had to make in order to expand it. We analyzed the basic demands and challenges of the work. There are some issues that had to be dealt with: The correct floating-point values that both the constants and the variables should get. Finally, we described what happened when we interconnect a complete 4 cells. In the next chapter, we present the evaluation results obtained from the design space exploration and from the synthesis results of a single IO-cell, two interconnected single IO-cells and four interconnected

single IO-cells.

5 Design Evaluation

"There are no secrets to success. It is the result of preparation, hard work, and learning from failure."
-Colin Powell

In the previous section, we have presented our background, application description and tool-flow. In this section, we present the experimental setup and validation methods of the design process and the experimental setup used to evaluate the design space.

The chapter is organized as follows: In Section 5.1 we present the tools and methods. In Section 5.2 we provide the experimental setup and define the metrics of of implementation characteristics. After that, we show the validation methods. Finally, the chapter concludes with a presentation of the results in terms of performance and area.

5.1 Tools and Methods

It is crucial to have good tools to explore a wide range of design choices. Especially due to the complexity of today's embedded systems, especially during the early design stages, where the design space is at its largest. This forces designers to start with modeling, scheduling and simulating system components and their interactions at very early design stages. The TUDelft tool (a Matlab-based tool) is a useful means used to analyze an application, schedule resources, plot a design space, view a DFG and generate VHDL code. As we explained previously, there are different scheduling methods that are employed by this tool:

- As Soon As Possible (ASAP)
- As Late As Possible (ALAP)
- Force Directed (ForceD)
- List

Moreover, the TUDelft tool needs only a couple of seconds to execute the ASAP and ALAP algorithms. However, it takes much longer to run the List scheduling algorithm as it performs a heuristic search in the design space. List scheduling is a cyclic executive method. The basic scheme is to cycle through a repeating sequence of possibilities. The basic idea of list scheduling is to make an ordered list of operations by assigning to them different priorities, and then repeatedly execute the following two steps:

- Select from the list, the operation with the highest priority for scheduling

- Select the best possible resource to accommodate this operation until a valid schedule is obtained.

Note that forceD does not schedule resources that need more than one clock cycle in order to execute and since some of the operations need 4 clock cycles, forceD is dropped from the consideration, as we explained in Chapter 4.

Once we select a point from the design space, we generate its VHDL code using the TUDelft tool. Afterwards, we move to ModelSim SE 10.1b to simulate the generated code. Finally, the circuit is synthesized by Xilinx ISE Design 13.4b.

5.2 Experimental Setup

In this section, we discuss the experimental setup in which we evaluate the final design. Our setup is made up to evaluate design characteristics such as area, frequency and latency and try to find the optimal design point.

5.2.1 Validation Methods

To validate our work we simulate the VHDL code that is generated by the TUDelft tool and comparing the outputs of this simulation with the responses of the equivalent C code.

5.3 Results

We generate the VHDL design based on the flowchart shown in Figure 5.1. The flowchart consists of 5 steps:

- Design space definition. To define the design space
- Design space initialization. To initialize the design space
- Design space exploration. To explore the design space to choose a Pareto point that suits our design and its constrains, among other Pareto points in the Design Space
- VHDL code generation. Once we choose that point, we generate VHDL code based on this point
- Design simulation and synthesis. After that we move to the simulation and synthesize

Each of them is explained next in more detail.

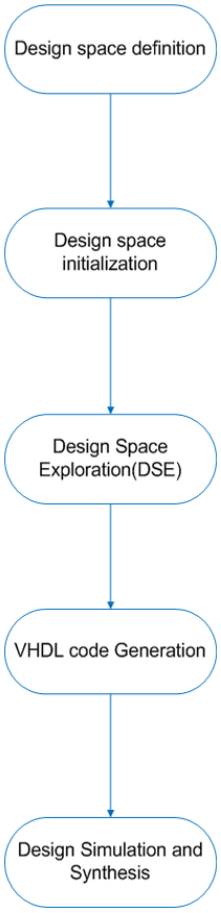


Figure 5.1: Strategy steps.

5.3.1 Design-Space

This step defines the design-space constraints and design parameters. This step comprises mainly three sub steps:

- design definition
- design constraints
- objectives

The design definition includes the parameters that allows the designer to define the design space. In this context Design Space Exploration (DSE) explores the design space by varying the parameter values without violating the design constraints, and determines the most suitable candidates that meet the objectives. In this work, we explore two design objectives: area and performance (which is the number of pipe stages). The design constraints are composed of the parameters that define the boundaries of the

design space, e.g. the target hardware (which is an FPGA in our case), area, latency, throughput, the number of general I/O pins. Finally, the objective step defines the design goals that will distinguish between optimal and suboptimal candidates. There are two main goals: Given a brain real-time (RT) behavior, how many cells can we simulate in a given FPGA array? And given a number of cells, how much speedup relative to brain real-time simulation cells can we achieve? This definition helps us to have a certain degree of control over the candidates.

5.3.2 Design Space Initialization

In this step, a set of Pareto points is generated. In our tools, the Xplore() function is used to generate the Pareto points that will be used in the next stage by employing three scheduling methods. These scheduling methods are ASAP, ALAP and List. List carries out in several iterations in which the best candidates of the previous iterations impose the choice of the next candidate during an exhaustive process. The number of List iterations is based on the number of resource types. We have five resource types as explained before. The range of each iteration is set by comparing the number of instances of each resource individually i.e.,(nEXP, nDIV, nMUL, nALU, nMIN), that is used in the previous two methods (ASAP and ALAP) and taking the maximum of this comparison. For instance, let the number of ASAP resources be (4, 7, 10, 16, 1), and the number of ALAP resources be (5, 10, 7, 10, 1) then the range of each List iteration becomes (nEXP= 5, nDIV= 10, nMUL= 10, nALU= 16, nMIN=1). Despite of the exhaustive search, results are obtained in an attainable time (3 to 4 days) for the given problem size after running the Xplore() function on an Intel Xeon E5620 (2.4GHz) 32.0 GB DDR3 64-bit OS windows server 2008. As a result, a number of globally optimal candidates is shown in Figure 5.2.

5.3.3 Design-Space Exploration (DSE)

This is the core step of the DS engine. As it is shown in Figure 5.2, the Xplore() function draws the Pareto points in the Design-Space (DS). Each Pareto point strikes an optimal trade-off between area and performance. As a result, we have a DSE with a number of Pareto points. One or more cost functions are required to spesify which point to select. Each cost function quantifies the merit of the solution with respect to a particular design goal. We have considered the following cost functions:

- AD
- A2D
- AD2

Where A refers to the consumed area units and D refers to the delay states. The DS graph is a very useful means for graphically depicting the relationship between the number of resources and delay states. By taking into consideration these two dimensions of the graph, i.e. the performance and area objectives, we can understand our design better. Figure 5.2 shows the DS of the IOcellFunctionCa generated with the TUDelft

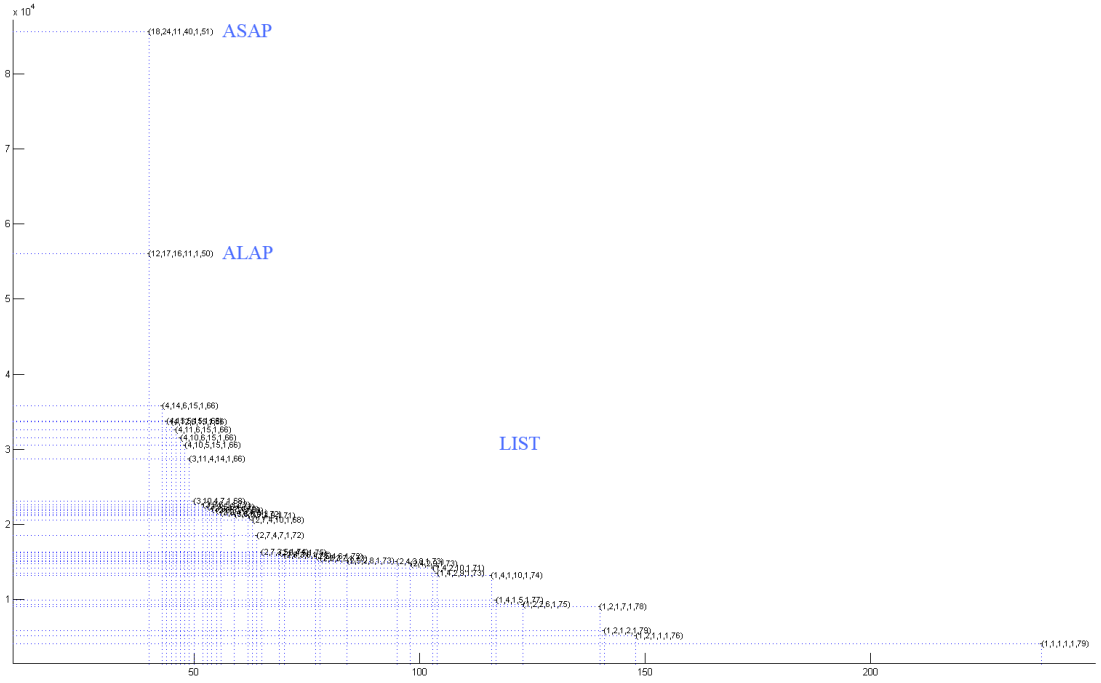


Figure 5.2: Design-space (DS) for IOcellFunctionCa.

tools for ASAP, ALAP and List scheduling. As it is shown in the figure, ASAP and ALAP have the best performance and highest resource consumption. With List scheduling, the execution time can be greatly increased when the number of resources is reduced. We can roughly classify the candidates into two groups:

- The ones that provide the lowest area
- The ones that provide the lowest latency

Note that both ASAP and ALAP require 40 delay states. However, ASAP needs 85639 area units while ALAP needs 56073 area units. This happens due to the fact that almost all of the resources in our application are concentrated in the early stages of execution and few at the end, but also ASAP tries to schedule as many operations as possible as soon as possible; then it needs more resources in the early stages than in the later stages, as it is shown in Figure 4.5 before. This leads to an inefficient use of the resources.

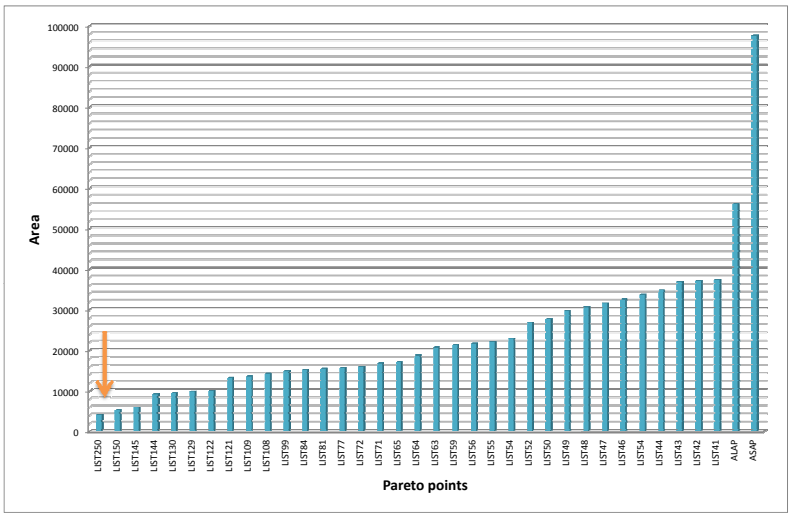
ALAP, on the other hand, tries to make each operation start at the latest opportunity without exceeding the 40 delay states. ALAP uses the Mobility technique as explained in the previous chapter. The remainder points in the DS reflect List scheduling candidates. Overall, area varies between 4098 and 39454 area units. Almost all these points are concentrated in the middle of the DS. In general, List candidates are distributed between 41 and 238 delay states (see Figure 5.2).

5.3.4 Evaluation Metrics

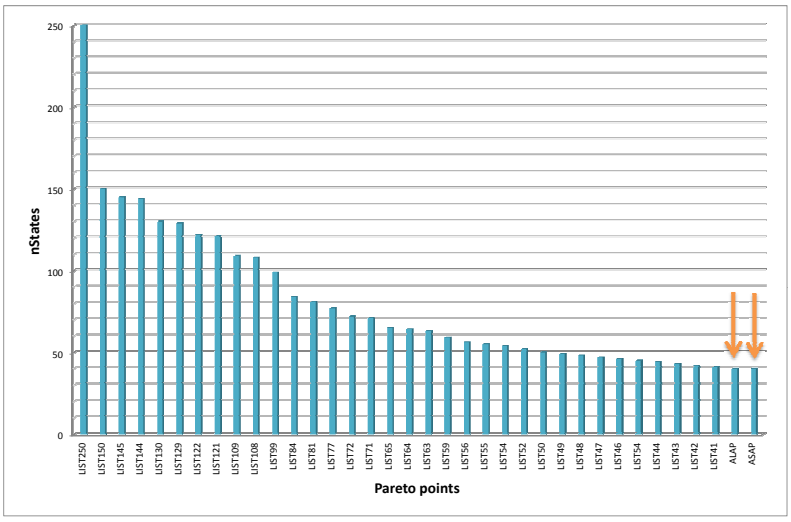
Without proper cost functions, the selection of a Pareto point in the Design-space is a very hard task since all Pareto points are optimal. In the beginning, we choose all of Pareto points and then we apply the design constrains to obtain a smaller set of Pareto points. Many real engineering problems involve more than two objectives. However, in this work which is a first attempt, we only study area and delay, which are two important objectives and are supported by the tool. That means we have a two dimensional (2-D) space as it is shown in the previous chapter. The objectives are often conflicting: (on one hand) we lose performance, (on the other hand) we gain a smaller design or vice verse. In such a problem, there is no single optimal solution. Instead, there exist a number of solutions that are all optimal. In the next section, we introduce area and performance objectives separately. Also, compound metrics are introduced to check which of the given Pareto points suits our implementation. Such objective metrics help us make our choice. Firstly, we introduce the Area-Delay (AD) product metric which emphasize equally for both area and delay. Secondly, we introduce the Area (square)-Delay (A2D) product metric which emphasize more for the area. Thirdly, we introduce the Area-Delay (square) (AD2) product metric which penalizes more for delay. In the end, we discuss our selection.

5.3.5 Area and Performance

For our target hardware, FPGA, area is a very important objective that we need to optimize in order to get a suitable implementation especially when having a big design that we should fit on it. Figure 5.3a shows the Pareto point candidates of the design-space of Figure 5.2. We arrange all candidates in an ascending order in terms of area. The highest resource consumption among all the scheduling methods is achieved by using the ASAP algorithm because it uses as many resources as it needs which, in turn, leads to a high performance. However, the ALAP column is nearly 42% shorter than ASAP and still performs as good as ASAP (40 delay states) as it is depicted in Figure 5.3b. This gain comes from the fact that in our application the operations that need to be executed are constrained in the early stages. That leads to highest resource consumption by ASAP/ALAP compared to list. On the other hand, and for this specific problem, ALAP has fewer tasks than ASAP and, thus, needs fewer resources than the former. Due to the fact that some tasks are flexible to be postponed to a later time slot to be executed, ALAP makes use of the Mobility technique. This technique takes into account the overall delay of the ASAP algorithm which is 40 delay states in our problem. In order to execute a single cell, we see that LIST250, which needs 250 delay states, has the shortest column i.e., lowest area, in Figure 5.3a. On the other hand, LIST250 has the highest column (highest latency) as it is shown in Figure 5.3b. In general, adding extra resources could reduce such a long delay at the cost of area. In conclusion, the candidate that has a low area (high area) consumption has also a long delay (short delay).



(a) Area objective.



(b) Delay States.

Figure 5.3: Area and delay states (performance) objectives.

5.3.6 AD Metric

The AD metric gives a balance between area and delay, which means, an equivalent combinational relationship between these two objectives.

The best candidate has the shortest AD value. Figure 5.4a shows these AD values for the Pareto points. We arrange all the candidates in an ascending order in terms of delay which is at the same time a descending order in terms of area as we explained in the previous subsection. As it is shown in Figure 5.4a, ASAP has the highest AD column.

That comes because the resources are too many with respect to the delay saved, as we discussed previously, and thus the highest performance. However, the ALAP column is nearly 42% shorter than the former according to the AD metric. The column of LIST41 is about 31% shorter than the ALAP candidate. It is shorter because it consumes less area than ASAP and ALAP.

The shortest column according to the AD metric is LIST150. It needs (from its name) 150 delay states in order to execute a single cell which is relatively a long delay. This means, on the other hand, low area consumption as it is proven before. Comparing the previous analysis information, LIST150 is the best candidate according to this metric. This means it represents the best compromise between area and delay.

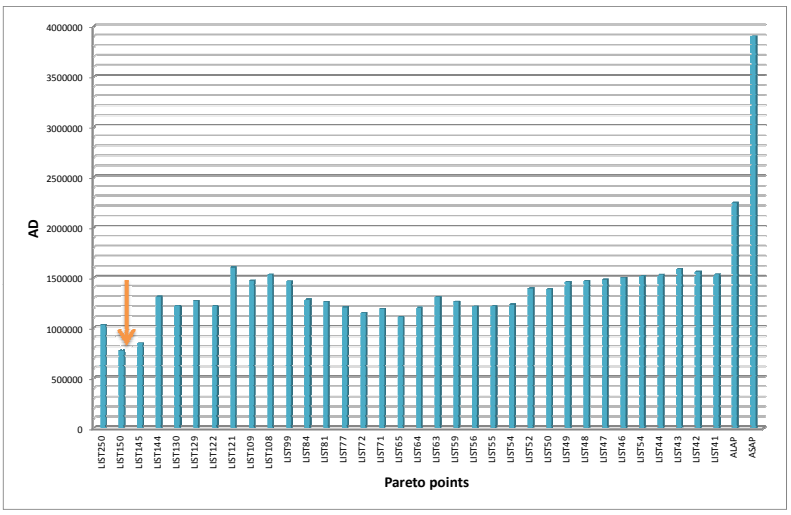
5.3.7 AD2 Metric

The second cost metric is the AD2 metric. This metric tries to stress the delay objective more than area. Figure 5.4b shows a relationship between Area and Delay squared. As is shown in the figure, this metric shows different optimal design points than the previous metric. We saw in Subsection 5.3.6 that ASAP has the highest column length and all List candidates have shorter columns than both ASAP and ALAP candidate. However, as it is shown in Figure 5.4b, the column of ASAP and ALAP are shorter than some List candidates. ALAP is also shorter than ASAP because of the same reason that we already explained before. Figure 5.3b shows that LIST250 candidate has the longest column of all. That happens because this candidate has a very long delay, which gets highly penalized when using the AD2 metric. LIST54 is an attractive candidate. It has better performance than the former and its column is about 74% shorter. LIST41 has the best merit for the AD2 metric. It is nearly 5% shorter and even has better performance than LIST54.

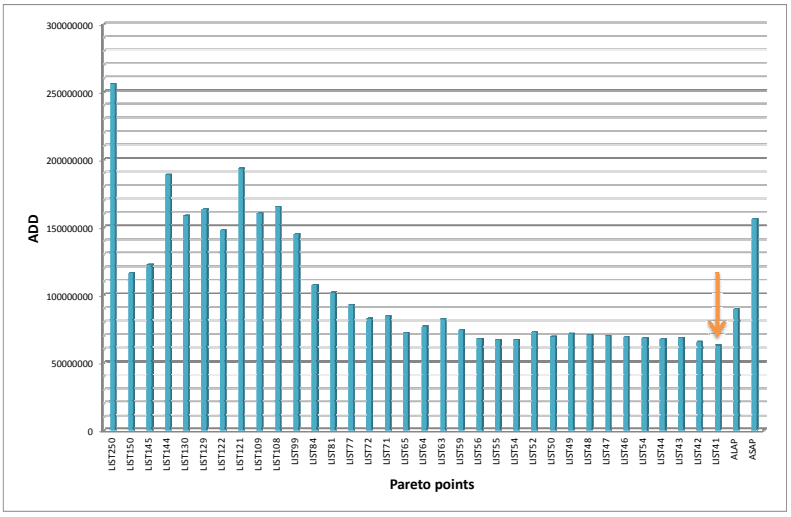
5.3.8 A2D Metric

The A2D metric, in which the area overhead is stressed is more relevant in cases of limited available resources.

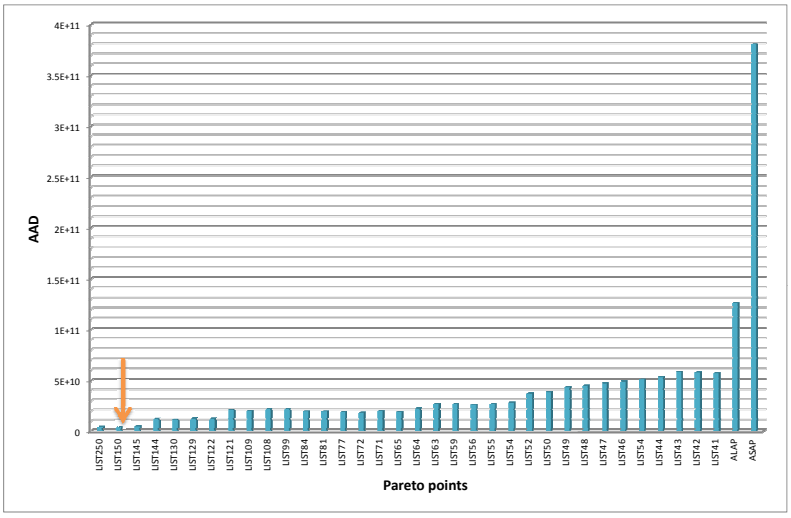
Figure 5.4c shows the relationship for the A2D metric. ASAP has a very high column, not only because it needs the highest amount of resources but also because this metric multiplies area squared by delay. Area is already high for ASAP and A2 makes it even more significant. On the other hand, ALAP is nearly 67% shorter than ASAP and still performs as well as it. This gain comes from the fact that ALAP uses fewer resources than ASAP, as discussed before. The List candidates show a little variety among each other. LIST41 has the highest performance among other List candidates but also its column is nearly 54% shorter than the column of the ALAP candidate which is, in turn, shorter than the ASAP candidate. LIST150 has the overall shortest column. Its A2D metric is approximately 93% lower than LIST41. The reason for that is the area consumption which is less in the last candidate. Comparing the previous analysis information, the LIST150 candidate is the best candidate according the A2D metric. This choice is based on the fact that this candidate has the shortest column.



(a) AD metric.



(b) AD2 metric.



(c) A2D metric.

Figure 5.4: Compound metrics.

Sched	Exp	Div	Mul	Alu	Min	Reg	nState	Area units
ALAP	12	17	16	11	1	43	**40**	56066
LIST41	8	14	7	8	1	60	41	37297
LIST150	1	2	1	1	1	73	150	5168
LIST250	1	1	1	1	1	76	250	**4095**

Sched.	AD	A2D	AD2
ALAP	2242640	125767253160	89705600
LIST41	1529177	57033714569	**62696257**
LIST150	**775200**	**4006233600**	116280000
LIST250	1023750	4192256250	255937500

Table 5.1: Conclusion Table, where *Sched* means scheduling method.

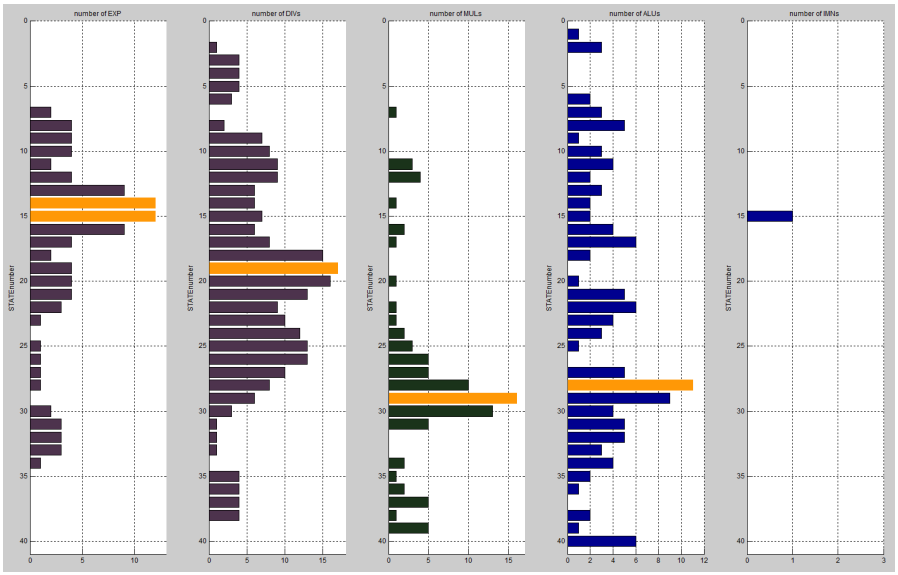
5.3.9 Conclusion

The goal in this section is to compare between the different nominated candidates. As it is shown in Table 5.1, we gather these candidates and put them in an ascending order based on their performance (number of states) from top to bottom. Starting from ALAP which has the lowest delay states, i.e., the highest performance, thus highest area consumption, down to LIST250 that has the lowest performance and thus least area consumption. In this table the lowest metrics and/or objective are displayed in bold. We find that the delay, area, AD-, A2D- and AD2-metrics are changed if the resource combination is changed. Using this table helps us find the most suitable trade-off in the Design-space for our problem.

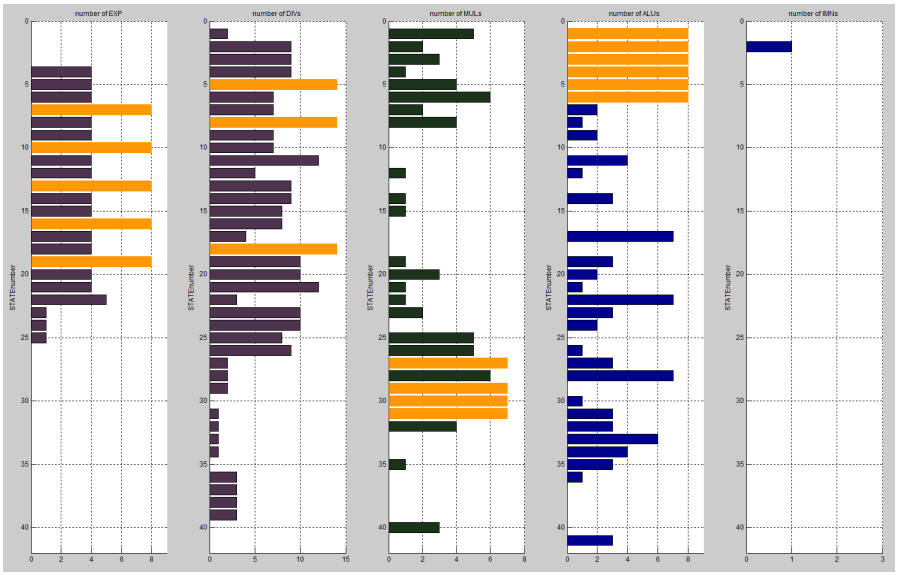
Table 5.1 shows that List41 candidate is the best solution on our design space according to the AD2 metric. It is also shown that List150 is advantageous according to both the AD and A2D metrics.

Figure 5.5a shows how the resources of the ALAP candidate are distributed by using the showDistrib() function. As we know, ALAP ranks first in terms of performance according to Table 5.1. However, this figure shows how inefficiently the resources are distributed. The orange column in the figure above shows the maximum usage of each resource. ALAP needs 12 EXPs and it consumes this number in only 2 states out of 40 states. It also needs 17 DIVs but it uses that number in only 1 state. It needs 16 MULs but it does not use that number of resources efficiently. The same with the ALU, it needs 11 ALUs and it uses that number only one time. However, it is the best candidate in terms of performance. Figure 5.7 depicts the DFG of that candidate in which the red circles are denoting a delay of 4 clock cycles elements, whereas the ones in black are denoting a delay of 1 clock cycle element. The reason that we have 4 clock cycles (drawn in red) for the EXP or DIV functional unit is that these components are consuming 4 clock cycles to execute. That means when there are red circles, we expect to see only a multiple of four; i.e. 4, 8, 16, etc. That is the case with every DFG depicted by using this tool.

Figure 5.5b shows the resource distributions of LIST41. It ranks second in terms of performance according to Table 5.1. This figure shows how inefficiently the resources are



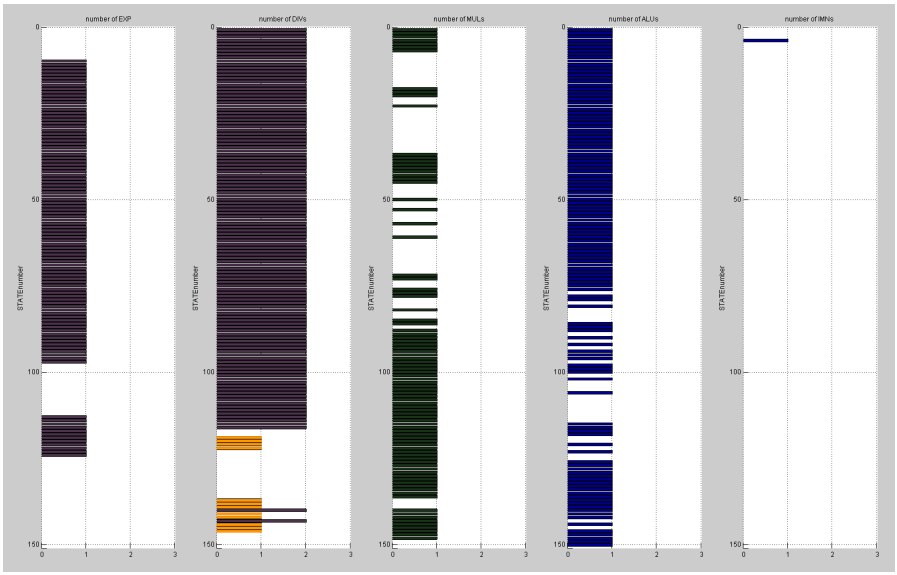
(a) The resource distribution of ALAP.



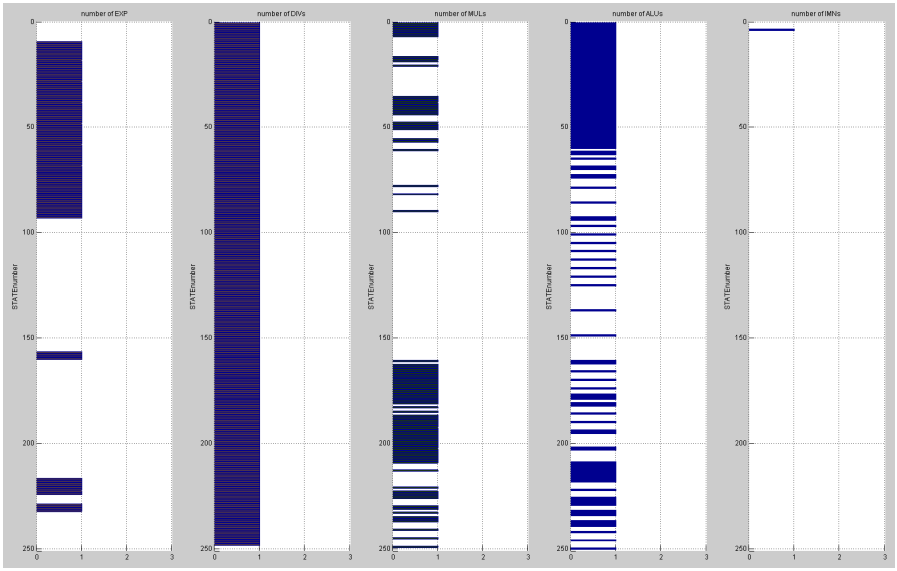
(b) The resource distribution of LIST41.

Figure 5.5: Distribution windows for the design points (ALAP and LIST41) of Table 5.1.

distributed. As we can see, the resources are distributed a bit better than the previous candidate, but they are still distributed inefficiently. EXPs, DIVs, MULs and ALUs are improved, however, they are still unbalanced. MIN is still unchanged because we need only one. LIST41 needs 8 EXPs but it does not use that number of resources very efficiently. It also needs 16 DIVs and it consumes this number in 2 states out of 36 states.



(a) The resource distribution of LIST150.



(b) The resource distribution of LIST250.

Figure 5.6: Distribution windows (LIST150 and LIST250) for the design points of Table 5.1.

It also needs 7 MULs but it uses all of them in only 2 states. It needs 8 ALUs but it does not use that number of resources very efficiently. Overall, it gains nearly 33% area units at the expense of 1 delay state with respect to ALAP. Figure 5.8 depicts the DFG of the LIST41 candidate.

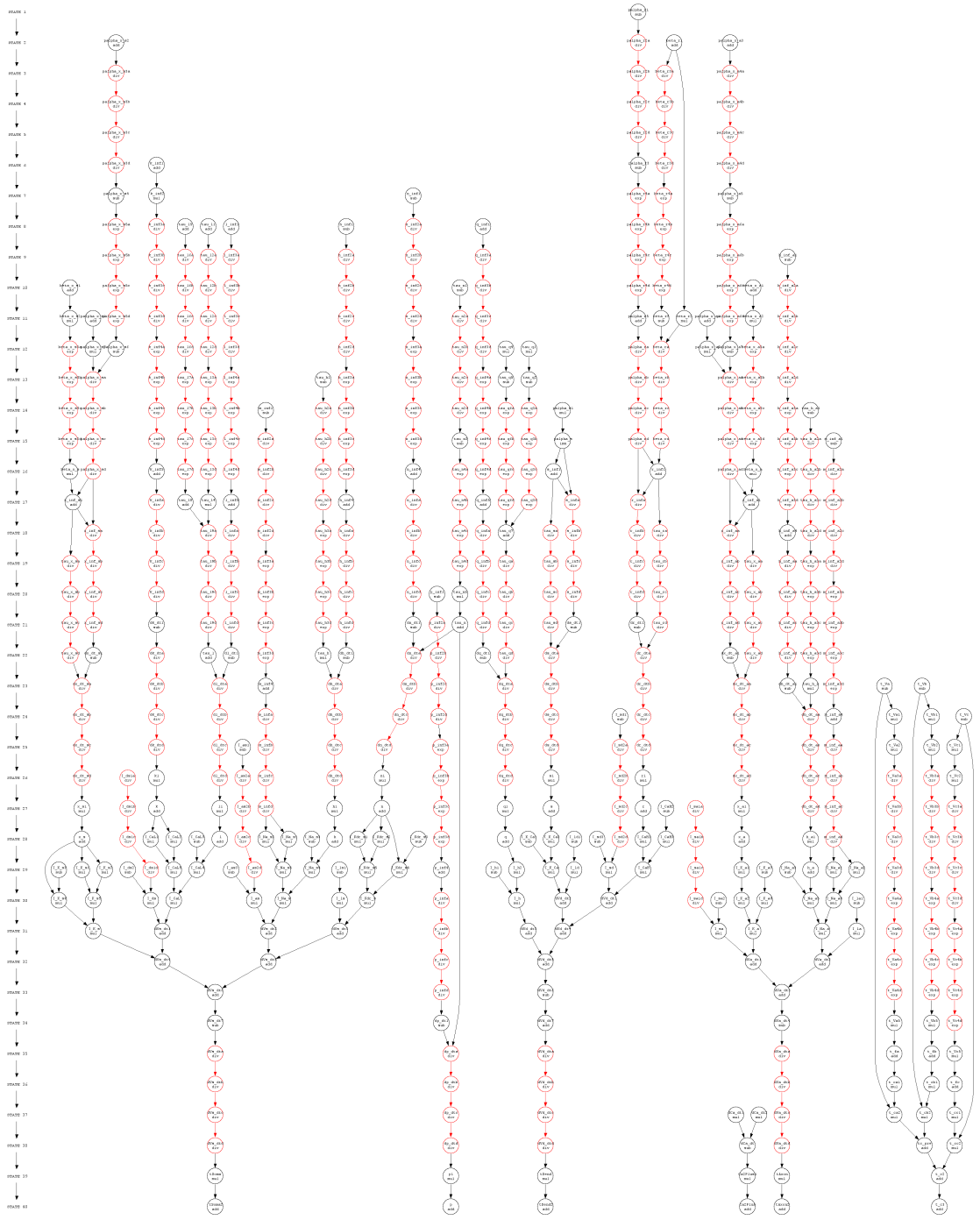


Figure 5.7: Data flow graph (DFG) of our neuron cell by using ALAP. The red circles refer to the resources that have a delay constraint of 4 clock cycles, whereas the black ones refer to those ones that take only 1 clock cycle.



Figure 5.8: Data flow graph (DFG) of our neuron cell by using LIST41. The red circles refer to the resources that have a delay constrain of 4 clock cycles, whereas the black ones refer to those ones that take only 1 clock cycle.

Figure 5.6a shows the resource distributions of LIST150. Its rank is third in terms of performance according to Table 5.1. In Figure 5.6a, the dominant colours are inverted, which shows that the resources are distributed a lot better than in the previous cases. EXPs, DIVs, MULs and ALUs are improved. As we can see from Figure 5.6a, the resources are distributed efficiently. However, LIST150 does not use the full number of DIVs in some states as we can see from the orange column in the figure. The figure shows that the design consumes almost the whole number of resources except in 12 states that

are shown in orange in that figure. Overall, it gains nearly 86% area units at the expense of 109 delay states. We did not show the DFG of the LIST150 candidate because it is a very big figure and if we include it we will lose its details.

Figure 5.6b shows how the resources are distributed according to the LIST250 candidate. It ranks forth in terms of performance according Table 5.1. LIST250 is not winning across the compound metrics but it shows full resource utilization. The figure shows how the resources are distributed better than the previous case in which DIV is improved. EXPs, MULs, ALUs and MIN is still unchanged. The figure shows the best resource distribution among the previous figures. Overall, it gains nearly 20% area units at the expense of 100 delay states with respect to the previous candidate. This means that better distribution leads to a long delay. We can see that compared to LIST150, LIST250 only saved one DIV, at the expense of 100 more delay states (cycles). If we look at the DIVs utilization in Figure 5.6a the high increase in delay of LIST250 is expected. Note that we have not included the DFG of LIST250 because it is very big.

5.4 VHDL Code Generation

Once we select a Pareto point from the design space (Figure 5.2), we move to the VHDL code generation. As mentioned in the previous chapter, the tool can generate the VHDL code by using the gen_VHD() function. After running this function three different VHDL codes will be generated that are:

- X_SSG_auto.VHD
- X_auto.VHD
- Testbench_X_auto.VHD

where X refers to the application name that we need to implement. This VHD code describes an architecture as it is shown in Figure 5.9. The generated code inside the X_SSG_auto.VHD is a data path that has FSMs which, in turn, contain various FUs. It works like a controller that implements the scheduling algorithm. The generated code of the X_auto.VHD works as a wrapper to the former. The testbench_X_auto.VHD code will be working on top of them, as is depicted in Figure 5.9.

5.5 Design Simulation and Synthesis

After making sure that the Pareto point(s) we chose suit(s) our application better than the others, we enter this stage. In this stage, we simulate our design by using ModelSim SE 10. Afterwards, we synthesize it by using Xilinx ISE suite 13.3. The synthesis result is shown in Figure 5.10:

5.6 Scalability Analysis

The purpose of this section is to analyze the result of different points after synthesizing them. We will first synthesize a single cell and study its synthesis result. Since our goal

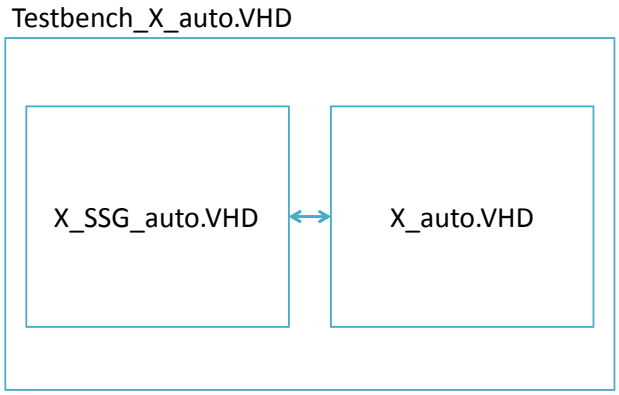


Figure 5.9: The architecture of the generated VHDL code.

Slice Logic Utilization (single cell, LIST41):
Number of Slice Registers: 6995 out of 687360 1%
Number of Slice LUTs: 34417 out of 343680 10%
Number used as Logic: 34291 out of 343680 9%
Number used as Memory: 126 out of 99200 0%
Number used as SRL: 126

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 39160
Number with an unused Flip Flop: 32165 out of 39160 82%
Number with an unused LUT: 4743 out of 39160 12%
Number of fully used LUT-FF pairs: 2252 out of 39160 5%
Number of unique control sets: 97

IO Utilization:
Number of IOs: 381
Number of bonded IOBs: 174 out of 840 20%

Specific Feature Utilization:
Number of BUFG/BUFGCTRLs: 2 out of 32 6%
Number of DSP48E1s: 70 out of 864 8%

Slice Logic Utilization (single cell, LIST150):
Number of Slice Registers: 3197 out of 687360 0%
Number of Slice LUTs: 9043 out of 343680 2%
Number used as Logic: 9025 out of 343680 2%
Number used as Memory: 18 out of 99200 0%
Number used as SRL: 18

Slice Logic Distribution:
Number of LUT Flip Flop pairs used: 11652
Number with an unused Flip Flop: 8455 out of 11652 72%
Number with an unused LUT: 2609 out of 11652 22%
Number of fully used LUT-FF pairs: 588 out of 11652 5%
Number of unique control sets: 58

IO Utilization:
Number of IOs: 381
Number of bonded IOBs: 174 out of 840 20%

Specific Feature Utilization:
Number of BUFG/BUFGCTRLs: 1 out of 32 3%
Number of DSP48E1s: 9 out of 864 1%

Figure 5.10: The synthesis result of a single cell using LIST41 and LIST150.

is having a network of cells, then we try to interconnect two single cells, four single cells, etc, until we have a network of n cells. In the first stage, we interconnect two cells and synthesize them. We compare the synthesis results of the single cell and the two cells and analyze the difference between them. In the end, we will deal with four cells and study their synthesis result in comparison with the single and the two interconnected cells.

Based on the information of the previous section we choose two points. These points are LIST41 and LIST150. The former is chosen based on the AD2 metric. The latter is chosen based on both AD and A2D metrics. In order to use these two candidates we introduce two scenarios:

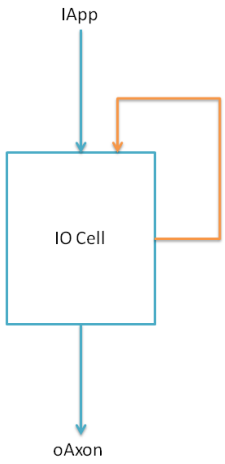


Figure 5.11: A single IO-Cell.

1. Respecting the brain Real-time constraint, how many simulated cells can we implement on a single FPGA?

2. Given a fixed number of simulated cells, what is the highest performance we can achieve?

We synthesized these two points separately by using Xilnix ISE design Suite 13.3. Firstly, we synthesize a single IO-cell (Figure 5.11) of each candidate. Secondly, we synthesize two interconnected single IO-cells (Figure 5.12). Finally, we synthesize four interconnected single IO-cells (Figure 5.13). We analyze the information obtained by the synthesis separately according to the above two strategies to know which candidate would be more suitable for our design.

Based on the synthesis result of the single, two and four cells we take the hardware-cell usage number (H) of the FPGA in order to calculate the total number of hardware cells (HWc) that we could put on the FPGA. We assume that we can use 100% of TA of the FPGA. Assume we want to simulate as many cells as possible on the FPGA given the brain real-time constraint of 50 microseconds. We take the number of the minimum frequency period (MFP) of this design from its synthesis report and multiply it by its delay states (D) to calculate the total execution time (ExT) of a single cell. Also, we assume the brain real-time (BRT) of a single IO cell is 0.05 Milliseconds. We use this information to know how many times-denoted as simulation steps (SSteps)-our hardware cell is faster than real-time, biological cell. In other words, SSteps is the number of simulation points (dots) in the output that a single hardware cell calculates per Milliseconds. For answering the first scenarion, we multiply HWc by SSteps in order to get the number of virtual cells that we could map on the FPGA, while respecting the model real-time constraint. We call these virtual cells at the brain real-time as VC-RT. Further, for the first scenario, we assume that we want to simulate 500 virtula cells

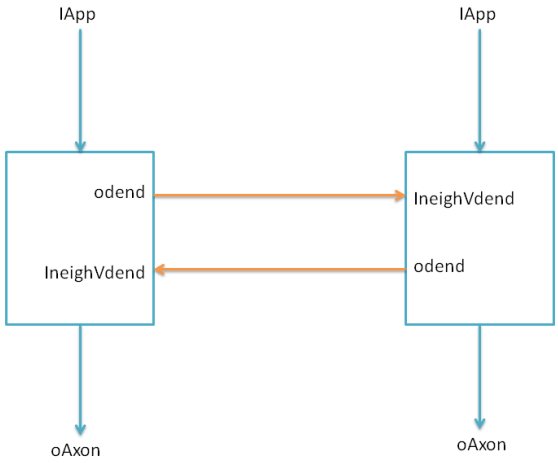


Figure 5.12: Two interconnected IO-Cells.

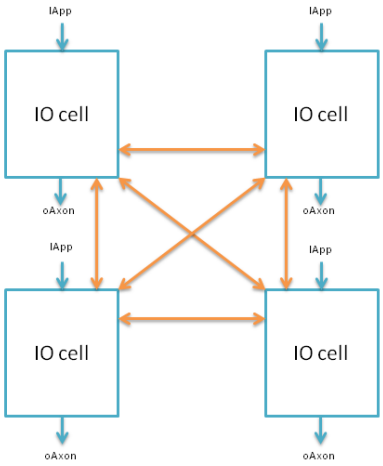


Figure 5.13: Four interconnected IO-Cells.

(VC). We divide VC-RT by VC to find how many times the network of 500 virtual cells is faster (SpeedUp) compared to the maximum network size (i.e. VC-RT). Below is the caculation that we applied on LIST150 based on these two scenarios Next, a sample caculation for point LIST150 (single cell) is given:

	1 IO-cell		2 IO-cells		4 IO-cells	
	List41	List150	List41	List150	List41	List150
HWc	10	50	10	48	8	44
VC-RT	795	**900**	759	**900**	560	**792**
SpeedUp	59%	**80%**	59%	**80%**	12%	**58.4%**

Table 5.2: Scalability-Table results of our two scenarios. HWc refers to the hardware cells, VC-RT refers to the virtual cells at the brain real-time and SpeedUp stands for the speed up.

$$H = 2\%$$

By knowing what the cell area is (H) and assuming we can consume the whole FPGA space, we can calculate the number of physical cells (the hardware cells) HWc.

$$HWc = \frac{TA}{H} = \frac{100}{2} = 50\ cells$$

By knowing the delay from the scheduling result and the MFP from synthesis, we can get the execution time of a single virtual cell (ExT).

$$ExT = MFP \times D = 18.24\ ns \times 151 = 2754.24\ ns$$

By dividing the brain real-time (BRT) constraint by the execution time (ExT) we can get how much faster one virtual cell is (SSteps).

$$SSteps = \frac{BRT}{ExT} = \frac{0.05\ Milliseconds}{2754.24\ ns} = \frac{50000\ ns}{2754.24\ ns} \geq 18\ times$$

By multiplying the number of hardware cells (HWc) by SSteps we got how many virtual cells we can implement at brain real-time speed.

$$VC - RT = HWc \times SSteps = 50 \times 18 = 900\ virtual\ cells$$

By dividing number of virtual cells at the brain real-time (VC-RT) by the network of 500 cells (VC) we know how many times our VC is faster compared to VC-RT.

$$SpeedUp = \frac{VC - RT}{VC} = \frac{900}{500} = 80\%\ faster\ than\ real\ time$$

Moreover, we did the same previous calculation with a single IO-cell (generated by LIST14) but also with 2 single IO-cell and 4 IO-cells (generated by LIST41 and List150). Table 5.2 shows the results of these two scenarios.

5.7 Discussion

We made a presentation of the evaluation results including performance and area. We also created some simple metrics for performance and area estimation that are AD, AD2 and A2D. Consequently, we made our scalability analysis based on two scenarios. Table 5.2 shows the summary of the previous individual design points. In this table, the best results are displayed in bold. In the first scenario, where we respected the brain real-time (RT) which is 50 microseconds, the missing variable was the number of virtual cells at the brain RT (VC-RT). Initially, when we implemented a single IO-cell (generated by LIST41) we found that we could map 795 VC-RT on 10 hardware cells (HWc) that we map on the FPGA as is shown in Table 5.2. The result also shows that when we implement one IO-cell (generated by LIST150) we could map 900 VC-RT on 50 HWc that we map on the FPGA. That means using a single IO-cell (generated by LIST150) provides us better area consumption because it allows us to map 105 extra VC-RT than using a single IO-cell that is generated by LIST41. That inspired us to interconnect more than one cell to understand the behavior of the IO-cell network. Secondly, when we interconnected two IO-cells generated by LIST41 we found that we could map 759 VC-RT on 10 HWc, whereas we could map 900 VC-RT on 48 HWc when we interconnected two IO-cells generated by LIST150. That means the two interconnected IO-cells produced area overhead. Lastly, we observed that interconnecting four IO-cells generated by LIST41 (LIST150) led to 560 (792) VC-RT on 8 (44) HWc. In conclusion, using a single IO-cell, two IO-cells and four IO-cells (generated by LIST150) provided us 105, 141 and 232 more IO-cells respectively than using a single IO-cell, two IO-cells and four IO-cells generated by LIST41.

We assumed that the number of virtual cells (VC) is equal to 500 cells in the second scenario and the the speed up (SpeedUp) was the missing variable that we had to calculate. Firstly, we found that when we have a single IO-cell (generated by LIST41) 99% of the total area (TA) of the FPGA is used and 59% SpeedUp with respect to BRT, whereas the IO-cell that is generated by LIST150 used 100% of TA of the FPGA with a speedUp of 80% faster than BRT, as is shown in Table 5.2. Afterwards, we found (when we interconnected two IO-cells) that LIST41 used 90% of TA of the FPGA and its SpeedUp is as the result of the single cell, whereas the two cell (generated by LIST150) did not show any difference than the single cell. Finally, we observed (when we interconnected four IO-cells) that LIST41 is used 74% of TA of the FPGA with SpeedUp of 12%, whereas LIST150 used 99% of TA of the FPGA with a SpeedUp of 58.4%. In conclusion, using a smaller design gives a better results in terms of simulation steps and area usage and also connected more cells gives more simulation steps in both small and big design point.

5.8 Summary

In this chapter, we discussed different subjects. First, the development and evaluation process, the validation methods of the TUDelft tool after our modification. Also, the experimental setup that was made in order to evaluate the outcome of the work and the goals behind it. Lastly, we made a presentation of the evaluation results including performance and area. We also create some simple metrics for performance and area

estimation that are AD, AD2 and A2D. We conclude due work with a preliminary scalability analysis of different cell schedules when optimizing either for cell-population size or simulation speed. In the next chapter, we will conclude this thesis by presenting our conclusions and suggestions for future work on the subject.

6 Conclusions and Future Work

"If you want a happy ending, that depends, of course, on where you stop your story."
-Orson Welles

In this thesis we tried to explore the challenges and characteristics of neural modeling application on FPGAs. The main idea was to implement the Inferior Olive (IO) cell towards a network of IO-cells by using an HLS tool. We have selected and described the HLS tool which is used to schedule, plot and generate the VHDL code. Then we described the tool abilities and its functions. Afterwards, we described the expansion and the modification that we have done to get the tool to work for the target application. Moreover, we presented some metrics that help us choose the most suitable Pareto points for the specification of our design by exploring the design space that the tool generates. We also presented the implementation, the evaluation process and provided the measurement results of one IO-cell, two interconnected IO-cells and four interconnected IO-cells. In this chapter we discuss some conclusions based on our evaluation and make some suggestions for future work.

6.1 Evaluation Conclusions

The results obtained through our experiments concentrated on two scenarios. The first scenario was having as many real-time simulated cells as possible on a single FPGA, whereas, the second scenario was implementing as fast a design as possible for a given amount of simulated cells. Our application is represented by a data flow graph (DFG) in which its operations have a high parallelism. That means it is better to move our application to hardware in which all of the parallel operations execute together at the same time. Of course, such a parallelism should respect the resource availabilities and resource constraints. There are many HLS tools that one could use in order to translate a software sequential code into a hardware parallel code. These tools reduce the implementation and the validation stage a lot and produce nearly optimal solutions. However, most of these tools were not available for us to modify, expand and use. The TUDelft tool is an HLS tool that schedules and converts the software code into a VHDL code. It tries to produce an efficient implementation by making use of the scheduling methods available in the tool. This tool uses ASAP, ALAP, forceD and List scheduling methods. In the first instance, the tool did not support all of the operations involved in our application. It also did not support the floating-point representation that our neural model application has been implemented with. However, the tool was available for us to modify, expand and use.

Firstly, we expanded the number of operations that the tool supports from 2 operations (MUL, ALU) to 5 operations (EXP, DIV, MUL, ALU and MIN). Furthermore,

we changed the library format from fixed-point arithmetic to single-floating-point arithmetic. Subsequently, we explored the design space that the Xplore() function of the tool generates. That design space consists of a number of Pareto points that have been plotted based on different scheduling methods. The design space is a 2-D space that traverses the area and performance objectives. We have explored it by introducing some compound metrics. These metrics are AD, AD2 and A2D. These new metrics along with the area and performance objectives stimulated us to choose two very promising Pareto points (LIST41 and LIST150) that suit our implementation best. From their name, these points have been generated by using the List scheduling method which fits our problem description and is known to generate near optimal schedules.

Afterwards, we created the first hardware design of the IO-cell by generating the VHDL code for both of the Pareto points that we chose from the design space, namely LIST41 and LIST150. After verifying their result with the original code that is written in C, we synthesized them. We found out that, in order to make the design synthesizable, we had to create yet a higher abstract wrapper to stay within the limit of the available input and output pins of the FPGA. We studied their behavior, such as the area that they use and how fast they perform, assuming that we can use all of the available area of our FPGA to implement the IO-cells.

We presented and evaluated our results based on some simple metrics for performance and area estimation that are AD, AD2 and A2D.

Consequently, in the scalability analysis that we made, two scenarios have been introduced. In the first scenario, the missing variable was the number of virtual cells at the brain RT (VC-RT) which is 50 microseconds. Initially, when we implemented a single IO-cell (generated by LIST41) we found that we could map 795 VC-RT on 10 HWc that we map on the FPGA. We also observed that when we implement a single IO-cells (generated by LIST150) we could map 105 extra VC-RT on 50 HWc compared with the result of the single cell. That inspired us to interconnect more than one cell to understand the behavior of the network of the IO-cells. Secondly, we found (when we interconnected two single IO-cells generated by LIST41) that we could map 36 less VC-RT than the single IO-cell on 10 HWc whereas, in the case of LIST150 we found that we could map the same amount of VC-RT as the single cell of the same kind on 48 HWc. That means the two interconnected IO-cells produced area overhead. Lastly, we observed that interconnected four IO-cells generated by LIST41 (LIST150) led to 199 (108) less VC-RT than the two interconnected IO-cells on 8 (44) HWc.

In the second scenario, we assumed that we have 500 virtual cells (VC) and the speedup (SpeedUp) was the missing variable. Firstly, we found that when we have a single IO-cell (generated by LIST41) 99% of TA is used with 59% SpeedUp whereas, the IO-cell that is generated by LIST150 used 100% of TA and has 80% speedUp, as is shown in Table 5.2. Afterwards, we found (when we interconnected two IO-cells) that LIST41 used 90% of TA its SpeedUp is as the result of the single cell whereas, the two cells (generated by LIST150) is not changed. Finally, we observed (when we interconnected four IO-cells) that LIST41 is used 74% of TA of the FPGA, which is 25% and 16% less than a single cells, two cells respectively, with 12% SpeedUp. We also found that LIST150 used 99% of TA of the FPGA with 58.4% SpeedUp. In general, using a smaller design gives a better results in terms of simulation steps and area usage and also

connected more cells gives more simulation steps in both small and big design point. That confirmed our previous observation that the smaller design generates better results than the bigger ones. Also, interconnect more than one cell produced area overhead.

6.2 Thesis Contributions

Through this thesis we accomplished:

- To explore the challenges and characteristics of implementing a single IO-cell as it is described in our problem statement. The challenges were focused on modifying the *HLS tool* and updating its *library*.
- To design and implement an IO-cell based on the tool and its library that we modified and expanded. We validate our work by comparing the result obtained from ModelSim and the result of the C code of the same implementation.
- To interconnect more than one IO-cell to form the IO-Network (two dimensional network of IO-cells).
- To evaluate our IO-cell and then the network of IO-cell. The difference between the IO-cells (generated using LIST150) is that four interconnected IO-cells led to 108 less IO-cells than the case both a single IO-cell and two interconnected IO-cells. That is because of the incorporation of the interconnect.

6.3 Future Work

Based on the experience gained while modifying and expanding the HLS tool of Delft University of Technology (TUDelft tool) and the evaluation results, we can now make some suggestions on future work that can be further explored based on the subject of this thesis:

- For completeness of the tool-flow, it is important to have a front end in the tool that converts the C or Matlab code to .cir format directly so that we do not need to hand write the code.
- It is even better to have a converter that converts the neuron languages (like Genesis) directly to .cir format.
- Modify the ForceD scheduling method to become able to schedule any operations that have more than one clock cycle.
- Validate the gen_mTB() function and the functions that it generates.
- Expand the HW generator (the gen_VHD() function) and make the wrapper directly generated along with the other files that it generates.
- Download the design on the FPGA which we could not do due to time limitations.

- Make an ASIC design to rescue the defected part of the brain.
- In this thesis, we have implemented the inferior olive (IO-cell). To have the complete olivo cerebellar circuit, we have to implement also the deep-cerebellar nuclei models and the Purkinje-cell models. These three components are linked to each other through connections in the brain and they form a three-element loop. With this loop, in which we have reverberating loops, motor activity can be controlled. We think that activity in these models will control sensory motor integration and will control particularly the right timing of movement. Thus, at least these two extra models need to be converted to VHDL as well.

Appendix: TUDelft Tool Modifications

A

This Appendix will attempt to give some more information on the modification of the TUDleft tool, used in the CE lab for Methods and Algorithmics for system design, and mostly concentrates on scheduling, design analysis and simulation. Its intent is to be used as a manual for future modifications of the tools.

A.1 Tool Modification

The tool is available for us, thus, we could expand it and add new features to it. Firstly, we had to expand the number of resources that are involved in the tool to make the tool suitable for all the operations that the application has. Also the format of these resources in the built-in library should be changed from fixed points into floating points format. In general, we divide the functions into two categories which are:

- Main functions
- Sub functions

Main functions are functions that we can directly run from the Matlab window of the TUDelft tool. These functions will lead to call the sub functions whenever they need them. Thus, the main functions call the sub functions implicitly. Each function might have its own local function (s). Figure 4.8 shows the function dependency structure of the TUDelft tool. That figure shows the dependencies between the functions.

After that, each function in the tool has been expanded by adding EXP, DIV and MIN to each piece of code that has MUL and ALU. Such expansion is done by replicating the piece of code that has MUL and ALU and change the resource name of the new code to the new resource name. The modification of some of these functions was more complex than only replicating a code. As we will see, We add to some of them extra conditions and/or loops. We start our modification with the main functions. These functions are shown in blue color as Figure 4.8 shows. We expanded the main functions by adding the new features to them. Our modifications start from the functions inputs. The new input parameters of these functions are based on EXPs, DIVs, MULs, ALUs, MINs - and consequently the temporary REGs needed - whereas, they based only on three parameters, that are MULs, ALUs and REGs, before the modification. These input parameters are used to specify the resource (operation) constraints. These operation constrains are divided into:

- Area constrains
- Delay constrains

This information is consumed by assigning each values to its parameter. Some times we have to assign a vector of constrains. This vector contains the same information that we mentioned before. Such a vector should contain six elements, that refer to these components, as follows: [expCost divCost mulCost aluCost minCost regCost].

That expansion generated errors because the tool could not recognize the new symbols. In that case we had to modify the tool's parser. The Parse function reads the .cir-file and converts it into an internal data format. It analyzes the text of the .cir-file, which is made of a sequence of words and symbols, to determine its grammatical structure with respect to a given formal grammar. In order modify the Parse function, we had to expand the Parse function by including the symbols of the new resources. We use $, / , *, +/ - and imn symbols to refer to the EXPs, the DIVs, the MULs, ALUs and MINs respectively. Note that we imn (and not min) as a symbol to refer to MIN operator to avoid the syntax errors because the MUL operator also starts with an m letter.

All these resource constrains are given in integer numbers of clock cycles. In order to consume these new input parameters we had to expand the internal code by replicating the part of code that contains MUL and ALU. Such expansion enforces us to modify the conditions and loops that these parameters involved with. The main functions may also have local functions that had to be modified.

Sub functions have been integrated in the tool to work as support functions. They have been called by the main functions to do a certain job. These functions have parameters that also had to be expanded. For example each main function (given in blue) in Figure 4.8 calls yet another function. For example, schedule function which provides the results of the scheduling functions to the Xplore function calls yet another function...etc (we refer the interested reader to Appendix A).

Some of the functions are able to draw figures, in that case, we had to do more modification to improve the information that these show. The figures that have been drawn by the tool before the modification were shown only three parameters that are nMULs, nALUs and nREGs respectively. We reduced the font number and add extra information to the plotting process to get the whole and clear view of the modification that we have done in such functions. Some of the tool functions contain loops that support the work of these functions. We expanded this kind of loops and add the new variables that we need.

Note that the expansion of some of the tool function was only in the input parameters further there was nothing need to be done like what we have done with the ASAP function whereas, other functions have not been modified att all like toAdjMat.

Yet another modification is done to reduce the execution time of the scheduling process. As we know that the complexity of List scheduling is NP-hard because it does an exhaustive search. We decide to reduce that time by reducing the number of the iterations that List scheduling does. We create new variable that gets the number of the delay states of the previous fast scheduling algorithms like ASAP and/or ALAP (any one of them since the have the same path length) and add 1 to that number. This stimulates us to get the result of the Xplore function faster than running the whole range of the design space. Figure 4.10 shows the the design space of the FIR5.cir that is generated by the unmodified version of the Xplore function, whereas Figure 4.11 shows the FIR5.cir

that is generated by the modified version of the Xplore function.

As we can see from these figures, the old function does not plot Pareto points and the points that are shown in it have three parameters (MUL, ALU, REG). That means that plot become a weakness specially when we have a huge design. There are two reasons motivated us to improve such a weakness that the tool has. The first reason is having a clear design space by reducing the number of the plotted points and the second reason is plotting only the Pareto points and make our work more optimal. As a result, Figure newXplore depicts the result of our modification. As it is shown in that figure, the points show the whole set of the resources (EXP, DIV, MUL, ALU, MIN, REG). Furthermore, We improve the work by reducing the number of the plotted points by only plotting the Pareto points as it is shown in Figure newXplore.

Another example is the result of the modification of the schedGUI function. SchedGUI is a graphical user interface that combines a number of the commonly used functions and eases the comparison of the scheduling methods. This GUI needs a .cir-file as input along with the resource constraints and resource latencies that we specify. Thus, this function has a gui interface that displays the DFG along with the resource distribution of that DFG. This function has many local functions that calls the gui interface. SchedGUI calls five external as it is shown in Figure 4.8 and also has many local functions that get their input parameters from the gui interface.

Figure 4.12a shows a screen shot of this schedGUI before modification, with the result when the FIR5 filter scheduled according to the List scheduling method, given the resource constraint that only 1 MULs and only 1 ALU (which can perform both the add and the subtract calculations) are available. Both MULs and ALU are supposed to have a latency of only one delay states. It can be seen that the result of the calculations will be valid after 6 delay states, while also the distribution of the resources over the delay states is shown. As we can see, the GUI expects that 1 additional REGisters will be needed to store intermediate results.

For the modification of this function we create new variables to be able to suit the new requirements along with the same modifications that should be involved within the main function. We create also new functions that are intended to receive the new inputs that we assign form the gui interface.

As a result, Figure 4.12b shows a screen shot of this schedGUI after the modification, with its DFG when the FIR5 filter scheduled according to the List scheduling method, given the resource constraint that is only 1 MULtipliers and only 1 ALU (which can perform both the add and the subtract calculations) are available. Both MUL and ALU are supposed to have a latency of only one delay states, but the figure shows that the gui interface also has a text windows in which we can specify the number of EXPs, DIVs and MINs along with their latency. It can also be seen that the result of the calculations will be valid after 6 delay states (because we use the same .cir-file) while the distribution windows of these five resources over the delay states is shown. As we can see, the GUI expects that 1 additional REGs will also be needed to store intermediate results. Figure 4.12b shows the screen shot of schedGUI after the modification with its DFG when FIR7 scheduled according List scheduling method, given the same resource constraints as the previous figure. FIR7 is the same as FIR5 but with same modification. We replace some multipliers with an exponent, divider and a min components. Further,

we give a latency of 4 clock cycles to the EXP and DIV and 1 clock cycle to the rest. Furthermore, the range of the resource number of the List scheduling of the TUDelft tool was between 1 and 15 before the modification, which means it could not accept zero or number more than 15. We expand that range and it accept the zero number and number more than 50. The previous figures show that the gui interface after the modification has three more distribution windows and more text windows to enter the constrains of the new resources. We change also the color and the position of some bottoms. From the gui we could get the DFG of different .cir-files.

The last modification was in the part that generates the VHDL code. First we modified the built-in library. There exist only two resources that are an ALU and a Multiplier along with a register. Thus, the TUDelft tool could only schedule operations that only have multipliers (MULs) or additions/subtraction (ALUs). This library supports only fixed point arithmetic which is not useful for our application. That adds extra challenges to the expansion process of the tool. We need not only modify the tool itself by letting it able to schedule more than one resources but, we need also to modify the library attached with it that is used in the simulation process after the generating of the VHDL code (see Figure 4.4).

Implementing a single floating point library is very complected but we did not have to do that because we were able to find that library we need. We ignore the attached file that contains the hardware library totally because these resources work with fixed points. Since our design is implemented with floating points, which means its results also in floating points, then our goal is having a design that supports floating points arithmetic to get such a result because in which we would be able to compare the results of our VHDL implementation with the original Matlab design.

The floating point library (FPLibrary) [4] that we use contains the core operators (+, *, /, square root) for floating point resources. To use the library, we first have to pre compile it. Its working library has to be fplib. Then, we just have to specify in the declaration of our entities that we need to include this library. This can be done by adding these two lines: library fplib; use fplib.pkg_fplib.all; After that, we can instantiate any of the components provided by FPLibrary in our designs. However the FPLibrary does not provide us all the resources that we need. That happens specifically with the ALU and the REG. Our design needs an ALU instead of having two separated components that are an addition and a subtraction. To do that, we had to modify the addition component and make it an ALU component. Such a modification is done by expanding the entity and adding an extra bit to be used as a sign bit (bit number 32) to control whether we need an addition component or a subtraction component. We called this bit as opcode and it used as follows. Whenever we need to used an addition component based on the .cir-file we assign a zero to this bit and it becomes one if we need to use a subtraction. That modification of adding the extra bit is enough to provide us an ALU because it uses a sign logic vectors. Such vectors becomes negative when the MSB is one and positive when the MSB is zero and practically the opcode bit dose that. Furthermore, the given library has a 34 bits instead of normal 32 bits. The extra 2 MSB are used to recognize whether the vector is a zero or a non-zero number.

Also in the VHDL generator part, the entities of the of the of fixed points library are hard-coded. We replaced them and include the entities of the new floating points

library.

We know that the application is written in Matland, whereas the tool could only understand a .cir-file. This adds yet another complexity to our work. We had to convert the Matlab design into a .cir-file. Unfortunately, the TUDelft tool is not able to convert the Matlab code into a .cir-file directly. That indicates a limitation in the tool.

Simulation

The TUDleft tool is ready now to schedule the .cir-file, display the DFG of that scheduling result, show the resource distribution of these resources, plot the design space to be explored later and generate the VHDL code to be simulated on ModelSim and synthesized on Xilinx. We have faced yet another challenge which is the input and the constant values that we should assign to the .INP file. ModelSim gets these inputs and the constant values from this .INP file which is created automatically by the tool via the gen_INP() function. This function generates the inputs that the ModelSim needs according the variables order of the .cir-file. It uses already made coefficient numbers that will be used as constant values. This function generates fixed point number which are not suited for our application and the modification that we did up to now. We modify this function as well to be able to generate floating point numbers. In the end, we conclude that these floating point number are in a double precision format. Since our library is based on single floating point formats, then these values should also be given in a single floating point format. Moreover, we prefer to make these floating point values in hexadecimal format because that would make the values validation procedure easier. In order to do that we make use of the C version of the IO-Cell function. From the C version, we could take the values that we need directly after compiling the code. Once we complete our input file, according to a given order, we would be able to run the experiments. As a result, we will get a file that has .OUT extension in the same directory of the VHDL project. That file will contain the output results. This file contains output variables and their values to be validated by using the original outputs of the C version of the IOCell design.

Synthesis

Afterwards, we moved to Xilinx to synthesis our design that we validated in ModelSim. The first synthesis attempt have not succeed. The design has a huge number of input and output variables which are more than the number of the already available IO pins. The use of a reasonably big FPGA like the Virtex-6 that we use did not prevent us such a problem. Nevertheless, our original IO-Cell has only one external input and only one external output. Therefore, the number of the input/output variables had to be reduced. That disparity between the demanded and the available amount of the inputs and outputs comes because of the constants that are given externally, whereas once could hard-code them. However, the tool does not have that feature. The rest input and output variables that are fed externally are actually used internally according to the original IO-Cell design. Based on that, we reduce the current number of that input and output variables by creating a higher abstract design that has only few inputs and output variables and the rest are made as internal variables, thus, we emulate the original IOcell design. We

called that high abstract design as an IOcellFunctionCa_wrapper_one_cell (wrapper). We defined vector signals inside that wrapper that have been hard-coded with the constants values that we put in the .INP file before. Once we give these signals their values we could use them in the port map of that wrapper. We should hard-code the values of the other internal variable. That is not enough because these variables get new values after a completion of one iteration; therefore, along with that we had to think in another solution. The TUDelft tool uses the feature of the linking variables. These linking variables are delay elements that are used to link two variables to consume the output of a variable as an input. We could get a feedback through these delay elements.

Bibliography

[1] *A.d.a.m. : http://www.adam.com/.*

[2] *http://legup.eecg.utoronto.ca/.*

[3] *http://www.calypto.com/.*

[4] *http://www.ens-lyon.fr/lip/arenaire/ware/fplibrary/.*

[5] *http://www.genesis-sim.org/genesis/.*

[6] *http://www.impulseaccelerated.com/products.htm.*

[7] *http://www.nkavvadias.com/hercules/.*

[8] *http://www.xilinx.com/products/design-tools/vivado/integration/esl-design/hls/index.htm.*

[9] J. Christopher Beck, T. K. Feng, and Jean-Paul Watson, *Combining Constraint Programming and Local Search for Job-Shop Scheduling*, INFORMS Journal on Computing **23(1)** (2011), 1–14.

[10] N. Carlson, *Foundations of Physiological Psychology*, Simon & Schuster, 1992.

[11] A. Cassidy, A. G. Andreou, and J. Georgiou, *Design of a One Million Neuron Single FPGA Neuromorphic System for Real-time Multimodal Scene Analysis*, 45th Annual Conference on Information Sciences and Systems (CISS) , March 2011, pp. 1–6.

[12] A. Cassidy, S. Denham, P. Kanold, and A. G. Andreou, *FPGA Based Silicon Spiking Neural Array*, IEEE Biomedical Circuits and Systems Conference, BIOCAS 2007, November 2007, pp. 75–78.

[13] De Jeu MTG De Zeeuw CI De Gruijl JR, Bazzigaluppi P, *"Climbing fiber burst size and sub-threshold oscillations in a network setting,* under review, (2012).

[14] R. Eswari and S. Nickolas, *Path-Based Heuristic Task Scheduling Algorithm for Heterogeneous Distributed Computing Systems*, International Conference onAdvances in Recent Technologies in Communication and Computing, 2009, pp. 30–34.

[15] Edward J. Fine, Catalina C. Ionita, and Linda Lohr, *"The History of the Development of the Cerebellar Examination*, Thieme Medical Publishers **22(4)** (2002), 375–384.

[16] A. Ghani, T. M. McGinnity, L. P. Maguire, and J. Harkin, *AREA EFFICIENT ARCHITECTURE FOR LARGE SCALE IMPLEMENTATION OF BIOLOGICALLY PLAUSIBLE SPIKING NEURAL NETWORKS ON RECONFIGURABLE HARDWARE*, International Conference on Field Programmable Logic and Applications (FPL '06), August 2006, pp. 1–2.

[17] B. Glackin, J. A. Wall, T. M. McGinnity, L. P. Maguire, and L. McDaid, *A spiking neural network model of the medial superior olive using spike timing dependent plasticity for sound localization*, Frontiers on Comput. Neurosci. **4** (2010), no. 18.

[18] Ruben Guerrero-Rivera, Abigail Morrison, Markus Diesmann, and Tim C. Pearce, *Programmable Logic Construction Kits for Hyper-Real-Time Neuronal Modeling*, Neural Computation **18** (2006), 2651–2679.

[19] A. Hodgkin and A. Huxley, *"A quantitative description of membrane current and its application to conduction and excitation in nerve.*, J. Physio **117** (1952), 500–544.

[20] C.S. Bouganis B. Falsafi G.N. Gaydadjiev R. Mariani D. Pnevmatikatos D.K. Pradhan G. Rauwerda I. Sourdis, C. Strydis, *Project periodic report for the first 6 months*, Tech. report, The DeSyRe project: on-Demand System Reliability, 2012, EC FP7 STREP project.

[21] W. McColloch and W. Pitts, *"A Logical Calculus of the Ideas Immanent in Nervous Activity*, Bulletin of Mathematical Biophysics **5** (1943), 115–133.

[22] Giovanni De Micheli, *Synthesis and optimization of digital circuits*, McGraw-Hill, 1994.

[23] Gyung-Leen Park, Shirazi B., J. Marquis, and Hyunseung Choo, *Decisive path scheduling: a new list scheduling method*, International Conference on Field Parallel Processing , June 1997, pp. 472–480.

[24] Andrei Radulescu and Arjan J.C. van Gemund, *On the Complexity of List Scheduling Algorithms for Distributed-Memory Systems*, International Conference on Field supercomputing, August 1999, pp. 1–8.

[25] F. Rosenblatt, *A Probabilistic Model for Information Storage and Organization in the Brain*, Psychological Review **65** (1958), no. 6, 368–408.

[26] B. Schrauwen and J. Van Campenhout, *Parallel hardware implementation of a broad class of spiking neurons using serial arithmetic*, IN PROCEEDINGS OF ESANN'06, 2006, pp. 623–628.

[27] Nicolas Schweighofer, Kenji Doya, and Mitsuo Kawato, *"Electrophysiological Properties of Inferior Olive Neurons: A Compartmental Model*, The American Physiological Society **99** (1998), 22–3077.

[28] H. Shayani, P. Bentley, and A. M. Tyrrell, *A Cellular Structure for Online Routing of Digital Spiking Neuron Axons and Dendrites on FPGAs*, ICES '08 Proceedings of the 8th international conference on Evolvable Systems: From Biology to Hardware, 2008, pp. 273–284.

[29] _____, *Hardware Implementation of a Bio-plausible Neuron Model for Evolution and Growth of Spiking Neural Networks on FPGA*, NASA/ESA Conference on Adaptive Hardware and Systems, June 2008, pp. 236–243.

[30] ———, *A Multi-cellular Developmental Representation for Evolution of Adaptive Spiking Neural Microcircuits in an FPGA*, NASA/ESA Conference on Adaptive Hardware and Systems , 2009, pp. 3–10.

[31] Behrooz Shirazi, Mingfang Wang, and Girish Pathak, *Analysis and evaluation of heuristic methods for static task scheduling*, Journal of Parallel and Distributed Computing **10(3)** (1990), 222–232.

[32] R. Walker and R. Camposano, *A survey of high-level synthesis systems*, Kluwer Academic Publishers, 1991.

[33] G. Wulfram and W. Werner, *Spiking Neuron Models*, Cambridge University Press, 2002.

[34] Chris I. De Zeeuw, John I. Simpson, Casper C. Hoogenraad, Niels Galjart, S.K.E. Koekkoek, and Tom J.H. Ruigrok, *"Microcircuitry and function of the inferior olive*, Trends Neurosci **21** (1998), 391–400.