μSystems Research Group

School of Electrical and Electronic Engineering

Design of Reconfigurable Dataflow Processors

Alessandro de Gennaro

Technical Report Series

NCL-EEE-MICRO-TR-2014-193

October 2014

Contact: a.de-gennaro@newcastle.ac.uk

Supported by EPSRC Impact Acceleration Award EP/K503885/1 (project “Dataflow Computation a la Carte”).

NCL-EEE-MICRO-TR-2014-193

μSystems Research Group
School of Electrical and Electronic Engineering
Merz Court
Newcastle University
Newcastle upon Tyne, NE1 7RU, UK

http://async.org.uk/

POLITECNICO DI TORINO

Facoltà di Ingegneria

Master's Degree in Computer Engineering

Graduation Thesis

Design of Reconfigurable Dataflow Processors

Advisor:
Prof. Luciano Lavagno

Co-advisor:
Dr. Andrey Mokhov

Candidate:
Alessandro de Gennaro

October 24, 2014

A mia nonna Pina, alla quale avevo promesso che mi sarei laureato.

Contents

Abstract xi

Acknowledgements xii

1 Introduction 1

2 Background 6
2.1 Conditional partial order graph model 6
2.1.1 A factorial issue in the number of solutions 8
2.1.2 Examples considered 12
2.1.3 Boolean equations extraction for each CPOG element 14
2.1.4 Boolean equations extraction for the model 15
2.2 Dataflow computing 19
2.2.1 Dataflow theory 20

3 Asynchronous dataflow pipelines design 24
3.1 Pipelining introduction 25
3.2 Asynchronous pipelining 28
3.2.1 4-phase dual-rail protocol 29
3.2.2 C-element 31
3.2.3 Pipeline implementation 33
3.3 Combinatorial circuitry 36

4 Dataflow graphs composition 40
4.1 Decoded circuit area evaluation 41
4.1.1 Gate library used 41
4.1.2 ABC tool usage 41
4.2 Cost function 43
4.2.1 Correlation between Partial Orders and op-codes 43
4.2.2 Cost function results and statistics 44
4.3 Graph Isomorphism Similarity 48

4.3.1 Graph isomorphism recognition . . . 49
4.3.2 Similarities between graph isomorphism and encoding process 50
4.3.3 Conclusions . . . 51
4.4 Encoding Generation . . . 51
4.4.1 Recursive encoding generation . . . 52
4.4.2 Random encoding generation . . . 54
4.4.3 Optimal encoding generation . . . 56
4.4.4 Tuning encodings by means of simulated annealing . . . 57
4.4.5 Generation outcome . . . 63

5 Workcraft integration and custom op-codes 64
5.1 Custom op-codes . . . 65
5.2 Workcraft integration . . . 67
5.2.1 Front-end and back-end connection . . . 67
5.2.2 Graphical User Interface . . . 69

6 CPOG applied to ISA development 72
6.1 Arm Cortex M0+ . . . 73
6.2 ISA designing technique . . . 73
6.3 Armv6-M representation via CPOG . . . 75
6.4 Composition results . . . 77
6.4.1 Final considerations . . . 84

7 Self-timed reconfigurable dataflow processors design 88
7.1 Ordinal pattern encoding for ordinal analysis . . . 88
7.2 Control-Unit design . . . 92
7.3 Datapath design . . . 97

8 Conclusions 109
8.1 Main contributions . . . 110
8.2 Future research directions . . . 111

A Partial orders modelling Armv6-M 114

Bibliography 126

List of Figures

1.1 Mobile devices growth along years, research by *IDC* [2]. 2
1.2 CPU-Batteries improvements throughout years, Figure taken from [11] p.7. 2
1.3 Cost of chip package Vs Power consumption, Figure from [11] p.12. . 3

2.1 Graphical representation of a *CPOG*. 7
2.2 *CPOG* projections: $\mathcal{H}|_{\mathcal{X}=1}$ on the right, $\mathcal{H}|_{\mathcal{X}=0}$ on the left side. 8
2.3 Op-code ensemble of length 3. 9
2.4 Two possible encodings for CPOG composed by 5 *Partial Orders* with 3-bits opcodes. 9
2.5 Encoding for CPOG composed by 5 *Partial Orders* with 3-bits op-codes exploiting don't care. 10
2.6 Simple *CPOG* models four basic operations of a ALU,([3], Figure 7.9). 12
2.7 8 instruction classes model. 13
2.8 Partial Order Graphs example ([3], Figure 2.5). 14
2.9 *Conditional Partial Order Graph* with 4 instruction classes. 17
2.10 CPOG-based microcontroller with acknowledgements/requests interface, taken from [3], Figure 7.4. 18
2.11 Particular of Maxeler Technologies transparencies, Slide N. 5 "Introduction of Dataflow computing" [30] 20
2.12 Static Dataflow Network example 22
2.13 Split and merge structures. ([31], Figures 6.12 - 6.13) 22

3.1 General structure of a RISC-based processor. ([32], P. 13) 25
3.2 Pipeline version of a RISC-based processor. ([32], P. 16) 26
3.3 Pipelined execution example. ([32], P. 26 modified) 27
3.4 Dataflow asynchronous pipeline example. 28
3.5 Typical handshaking protocol structure. 29
3.6 4-phase asynchronous protocol behaviour. 30
3.7 Bundled-data protocol model. 31
3.8 4-phase asynchronous dual-rail protocol model. 32
3.9 Schematic of 2 inputs c-element. 32

3.10 1 bit asycnhronous register 34
3.11 3 bits completion detection module 35
3.12 3 bits C-element implementation 35
3.13 2 bits - 3 stages asynchronous pipeline example. 36
3.14 Dual-rail AND gate implementations. ([38], Figure 2 modified) 37

4.1 *CPOG* with four instruction classes analysed. 44
4.2 Area-$\mathcal{F}$ plot of *CPOG* on Figures 2.9 (Blue) and 4.1 (Red) 45
4.3 Point dispersion of *CPOG* on Figure 2.7. 46
4.4 Line interpolation of *CPOG* on Figure 2.7. 47
4.5 Example of two isomorphic graphs. 49
4.6 On the left side an instance of Difference Graph, on the right side an Op-codes graph is depicted. 51
4.7 Number of solutions increments into *Recursive Generation* technique. 53
4.8 Recursive encoding generation applied to *CPOG* on Figure 2.7. . . . 55
4.9 Random encoding generation applied to *CPOG* on Figure 2.7. 57
4.10 Optimal encoding generation applied to *CPOG* on Figure 2.7. 58
4.11 Function fluctuations using *Simulated annealing* optimisation technique on Arm Cortex M0+ model. 60
4.12 Optimised encodings, applied to *CPOG* in Figure 2.7. 61
4.13 Generation encoding algorithms comparison. 62

5.1 Op-codes assignment example. 66
5.2 Graphical user interface of *SCENCO*. 70

6.1 ARMv6-M Instruction Set Architecture modelled via *CPOG* 76
6.2 Sequential encoding CPOG composition 79
6.3 Random encoding CPOG composition 80
6.4 Simulated annealing encoding CPOG composition 82
6.5 Exhaustive search encoding CPOG composition 83
6.6 Comparison between Simulated Annealing and Random search techniques 85
6.7 Comparison between Simulated Annealing and Random search technique (encodings weight) 86

7.1 Ordinal pattern encoding algorithm for subsequences of length 5, hardware structure inspired by [41]. 92
7.2 Ordinal pattern model for subsequences of length 5. 93
7.3 Ordinal pattern model for subsequences of length 7. 94
7.4 Compositional graph with Single-literal search of ordinal pattern operation structure. 95

7.5 Compositional graph with Simulated annealing heuristic search of ordinal pattern operation structure. 96
7.6 From fault to external damage([45], p.79). 98
7.7 N bit dual rail comparator from an external point of view. 100
7.8 Schematic of a comparator.([46], Figure 4.16) 101
7.9 1-bit dual-rail carry adder implemented with *NCL-X* gates 102
7.10 1-bit dual-rail carry adder implemented with *NCL-D* and *NCL-X* gates 103
7.11 Schematic. 103
7.12 *NCL-X* dual-rail XOR gate. 104
7.13 1-bit reliable dual-rail semi-adder implementation. 105
7.14 Legend of blocks present on Figure 7.15. 106
7.15 *Ordinal pattern encoding* hardware structure design under Workcraft. 108

8.1 Losses due to delayed market entry([47], p.13). 110
8.2 Error showed due to cost function used. 112

A.1 Memory instructions (Immediate offset addressing mode) 115
A.2 Memory instructions (Register offset addressing mode) 116
A.3 Memory instructions (Bunch registers transferring) 117
A.4 Load instruction Immediate addressing on PC 118
A.5 Load instruction register addressing on PC 119
A.6 POP instruction 120
A.7 Arithmetical, logic and data copy instructions (register addressing) . 121
A.8 Nop instruction 122
A.9 Unconditional branch instruction 123
A.10 Instructions with immediate addressing mode 124
A.11 Branch instructions 125

List of Tables

2.1 Example of Boolean conditions assignment of graph on Figure 2.8 . . 15
2.2 Boolean equations for each element for *CPOG* on Figure 2.9 16

3.1 Communication on dual rail protocol. 31
3.2 2 inputs c-element truth table. 33
3.3 NCL-D and NCL-X gates area comparison 38

4.1 Comparison among different solutions of two *CPOGs*. 45
4.2 Comparison between encodings generations applied to *CPOG* on Figure 2.7. 63

6.1 Controller size comparison for Armv6-M ISA model. 84

7.1 Truth table. 103

Abstract

This work presents a new method for designing reconfigurable dataflow hardware structures efficiently. The design-flow my research is based on is reliant on a model developed at Newcastle University named *Conditional Partial Order Graph*, also called *CPOG* [50], [3]. It is a representation which is finding many applications on the overall *VLSI* industry, because of its capability to capture different behaviours of a system in an extremely compact way.

Additionally, this dissertation presents a new heuristic algorithm, able to fill in the gap in this model at the controller synthesis phase. It is needed either for managing the reconfigurability of the system under design, and for handling request/acknowledgement signals for the sequentiality of the operations to be executed, in particular for self-timed structures. The new heuristic is able to seek an optimal op-code association (in terms of area consumption) for each different graph over a custom number of bits. The module comes up with the synthesis phase is used to control the events which may happen, likewise a control unit based structure.

The whole encoding tool has been integrated in `Workcraft` [54], which is a software that supports different models, both for synchronous and asynchronous devices design. One of the aim of this project indeed, was expanding such tool under *CPOG* plugin, and developing a Graphical User Interface to be used by designers during the former phases of a project. Furthermore, the whole design-flow has been applied to a real application in order to demonstrate the easiness through which a completely reconfigurable structure might be designed easily from scratch.

The number of applications this technique might be potentially applied for is huge. Let us think to how many systems could be splitted into events: the decoding part of *Instruction Set Architecture* of a processor for instance which tailors well to this technique, or an Application Specific hardware structure which must be able to perform different operations in parallel or sequentially. The limit of this design-flow could be found just into the imagination of the designer who has to be able to split the system to plan into smaller pieces, following the *Divide and Conquer* paradigm.

Acknowledgements

First of all, I want to thank my supervisors who have helped me during this research by giving me a great support both professionally and from a human perspective. In particular, I want to thank Prof. Luciano Lavagno who introduced me to world of hardware and software modelling, and guided me to the path who led me to Newcastle University where I have conducted my Masters thesis. Then, I would like to thank Dr. Andrey Mokhov who helped me to get settled in Newcastle upon Tyne, to carry out my dissertation and who has fostered my education throughout this research answering to my countless and stressful questions, he has been more than a simple supervisor to me. They have also helped me with invaluable advices which guided me throughout the decisions for my future career. I am also grateful to Prof. Alex Yakovlev, without his support I would have not been able to study in United Kingdom.

Additionally, I would like to send my acknowledgements to all the collegues of the School of Electrical and Electronic Engineering of Newcastle University I have had the pleasure to work with: Danil Sokolov, Maxim Rykunov, Paulius Stankaitis. They have shared their knowledge with me supporting my work and this research.

Moreover, I want to send my best acknowledgements to my whole family who have supported and tolerated me since I was young, thoroughly encouraging me to study hard even during the most difficult situations. I will be always thankful to them because, even though the countless sad situations have been through, they have never ever stopped to believe in me. Furthermore, I am grateful to Silvia, who has sustained me throughout the years as a student, tolerating me during the exams sessions and cheering me up when I was upset, tired or about to give in, even regarding the choices which led us far each other.

Lastly, I want to thank all my relatives starting from my aunt Maria Luisa and my cousin Federica who have helped me with an unbelievably great moral and professional support (first exams). My best friends who have been extremely important in my life and that have contributed to shape my character. And also my grandparents, with a particular emphasis on Pina who would have been extremely glad to see me achieving this goal.

Chapter 1

Introduction

The constant growth in the number of transistors it is possible to integrate in the same amount of area of an integrated circuit (as Moore estimated [1]), makes optimisation area problem one of the main concern designers should care about during the design of whichever electronic device. Additionally, the increasing amount of operations a component should be able to perform leads the need of an easy and flexible model to represent a system, simplifying not only the former phases of design-flow but even the latter ones. In fact, most of the mistakes that may affect a device stem by wrong specifications, or by the lack of a sound model to represent the whole system. On the light of above, a representation which cares about area/power consumption is needed, in order to reduce time to market of the products without loosing the quality of the device.

IDC (International Data Corporation), which is a corporation that helps IT professionals, business executives, and the investment community taking decisions via world wide statistics, forecasts an exponential growth on the smartphone shipment over the next few years, as depicted on Figure 1.1. It represents another valid reason for designers to address the high requirements in terms of battery life of mobile devices.

As illustrated in [11], four drivers lead the need of developing low power devices: *technological*, *market*, *economical* and *environmental* one. Poncino describes the former one as "the difficult to pack devices with high power consumption". In addition, another element which brought designers to care about power consumption was the big difference between battery and silicon technology improvement. As represented on Figure 1.2 in fact, during the past few years, while the silicon has been seeing an exponential enhancement, battery capacity has been remaining slightly unmodified, reducing battery-life of products.

Additionally, while *Market* is always more represented by higher number of portable devices, *Economical* reason expresses the correlation between the power consumed by a device and the cost of manufacturing it, due to chip packaging cost,

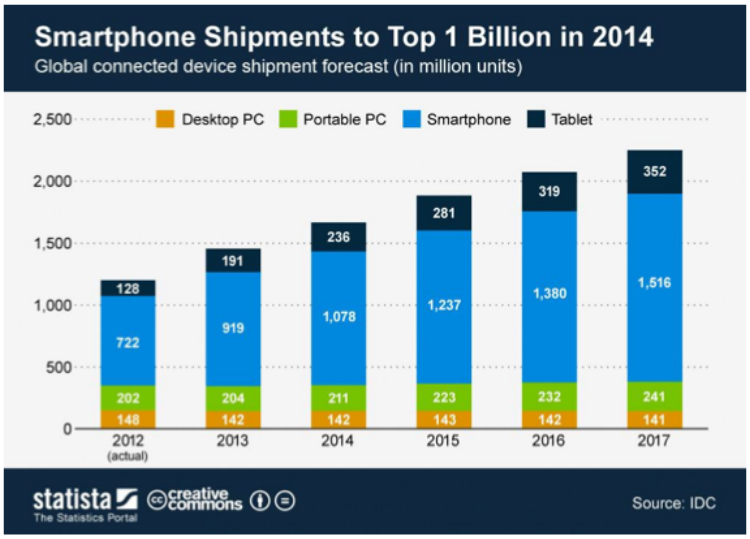


Figure 1.1: Mobile devices growth along years, research by *IDC* [2].

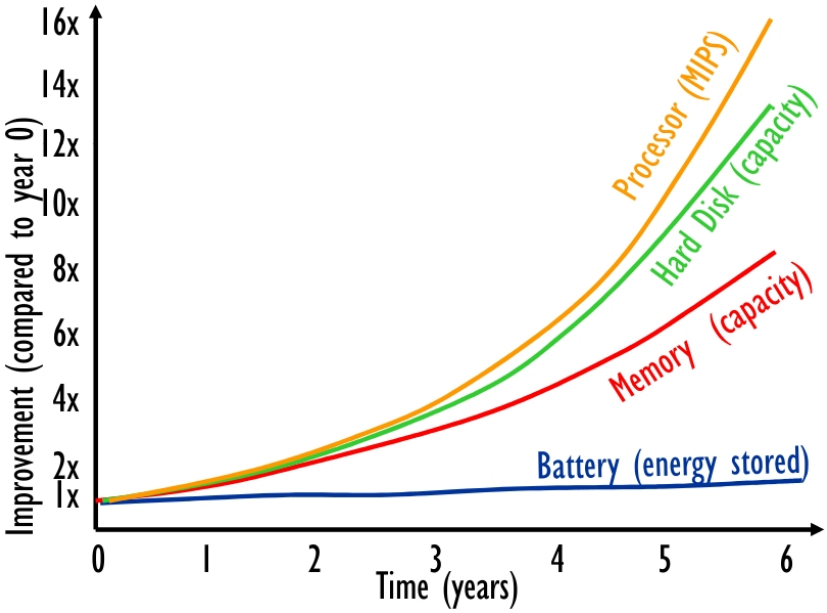


Figure 1.2: CPU-Batteries improvements throughout years, Figure taken from [11] p.7.

as depicted on Figure 1.3). Finally, it's important to develop low power components in order to build a sustainable market, as represented by *Environmental* reason.

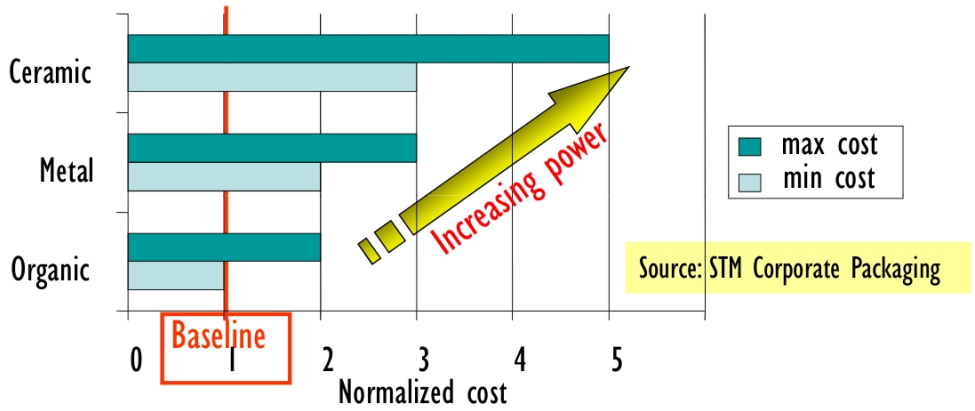


Figure 1.3: Cost of chip package Vs Power consumption, Figure from [11] p.12.

In this research, I am going to tackle the problems illustrated above, by introducing a new design-flow which fits well to *asynchronous dataflow reconfigurable hardware architectures*. As the former word explains having a self-timed structure may save power and area at the same time. Maxim Rykunov, who developed a totally self-timed 8051-based microprocessor on [4], states: "with a current multi-billion transistor design distribution of a global clock in the entire system could be costly in terms of area and power (up to 40% of the total chip power consumed by the clock distribution network)" ([4] p.6). Furthermore even a more flexible structure could be implemented, as proved in [4], able to keep working in very different operating conditions. It might tailor well to different kind of applications, spacing from medical to consumer market.

Additionally, designing a reconfigurable dataflow architecture means avoiding the whole the control-flow that takes place inside a single device. For instance, a RISC-based microprocessor contains several modules: from the fetching side to decoding part, which are in charge simply to figure out what an instruction should do. This is completely avoided with a dataflow structure which can execute the operation straight away. The drawback in the usage of such structure is the poor flexibility: as one might observe indeed, this application specific circuit can only perform one operation. Therefore the need of making such structure reconfigurable, in order to obtain an extremely good result tradeoff between flexibility and area/power consumption, as well as speed.

Hence, this work takes place. A recent model, able to describe such architectures very efficiently was born in order to support event-based hardware development. It is named *Conditional Partial Order Graphs* and described on [3] and [49] by Andrey Mokhov. First attempt to use such model can be found on [51], where phase-regeneration circuitry was the main concern. Afterwards, the theory of *Partial*

order has been elaborated and first presented officially on [50], where it was used in the context of asynchronous circuit design.

The model is composed by various number of graphs describing the different behaviours a system should be able to execute, and each of them may be seen also as a dataflow graph. The main goal of this thesis is to fill in the gap in this model by developing an algorithm for automating the composition of the separated graphs synthesising and mapping them on a real circuit, with a gate library set by the designer, trying to reduce area consumption of the controller for managing the whole structure. And to present the design-flow over a real application, with the future purpose to print it into an *ASIC*.

So far, *Conditional Partial Order Graphs* representation (hereinafter also called as *CPOG*) was used to build models for event-based hardware design, likewise dataflow structures, as well as asynchronous circuits. First step for synthesising such model into Boolean equations, is to assign an op-code to each scenario that composes the whole representation. The choice of a different encoding for each graph, affects the area of the final decoding circuit, and as will be pointed out over the next Chapters, the higher the numbers of *Partial orders* which compose the representation, the bigger the the area of the final circuit.

First aim of this research thus, is to automate the composition of several dataflow structures on the basis of *CPOG* theory, trying to reduce the area of the final decoder as much as possible. Along the research various examples will be analysed, and the design-flow will be applied to real applications, in order to demonstrate the results of the algorithm, and the applicability of this CPOG-driven flow in a more concrete case of study. Plenty of graphs have been analysed during time spent in the research. Here, for sake of readability, just few of them will be showed and discussed, the most meaningful ones.

The dissertation is composed by following Chapters: the **Introduction** presents the problem from a general point of view and illustrates the aim of this research, **Chapter 2** shows the background knowledge it is needed to reader for deeply inspecting the topic. Over the **Chapter 3** I am going to present and discuss about all the elements a designer needs to develop an asynchronous architecture, pinpointing on how to design a self-timed pipeline. Afterwards I am going to analyse the general procedure to design a reconfigurable hardware structure via the *CPOG* representation. The Conditional Partial Order Graph automation encoding problem is presented and discussed over the **Chapter 4**, where a rigorous analysis will be done, pinpointing on area optimisation. **Chapter 5** outlines the results obtained regarding high level software part (Graphical User Interface for supporting automation of graphs composition), illustrating how I modified the plugin for *CPOG* on `Workcraft`. Over **Chapter 6** is described an interesting possibility where such representation might be applied for: Instruction set architecture development, focusing on an example of a real processor. Finally, on **Chapter 7**, the designing

phase of a reconfigurable dataflow asynchronous processor will be showed, in order to make reader understand the power of such design-flow applied to a real application. Finally the **Conclusion Chapter** where the key contributions of this thesis are presented, and the area of future research are outlined.

Chapter 2

Background

This work is mainly based on previous research elaborated at Newcastle University, in particular by Dr. Andrey Mokhov and team of μSystem Research Group at School of Electrical and Electronic Engineering. With their work [3], they introduced a new light and flexible formalism able to model different event-based systems, spacing from self-timed to synchronous architectures.

Potentially, this representation might model each kind of structure that could be splitted into multiple events. For instance, as demonstrated in [4] by applying this formalism to `Intel 8051`, even the ISA of a processor could be modelled and synthesised with *Conditional Partial Order Graph* representation. Additional examples of the application of such method could be found in chapter 8 of [3].

In this Chapter I am going to revise the essential features of *CPOG*, in order to help the reader to understand this research and to introduce him to such formalism. Afterwards I am going to revise the properties of dataflow-computing, analysing the definitions and the different typologies it might be represented as.

2.1 Conditional partial order graph model

Definition.[1] Conditional Partial Order Graph is a quintuple $\mathcal{H}(\mathcal{V}, \mathcal{E}, \mathcal{X}, \rho, \varphi)$ where:

- $\mathcal{V}$ is a finite set of vertices which correspond to the events in the modelled system. V defines the system's event domain.

- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ is a set of edges representing dependencies between the events.

[1] This definition was elaborated entirely by Andrey Mokhov on his PhD dissertation [3], p.38. I just report it to introduce briefly this model for the reader.

- Operational vector $\mathcal{X}$ is a finite set of Boolean variables. An opcode is an assignment $(x1, x2, ..., x^{|\mathcal{X}|}) \in 0, 1^{|\mathcal{X}|}$ of these variables. An opcode selects a particular partial order from those contained in the graph.

- $\rho \in \mathcal{F}(\mathcal{X})$ is a restriction function, where $\mathcal{F}(\mathcal{X})$ is the set of all Boolean functions over variables in $\mathcal{X}$. ρ defines the operational domain of the graph: $\mathcal{X}$ can be assigned only those opcodes $(x1, x2, ..., x^{|\mathcal{X}|})$ which satisfy the restriction function, i.e. $\rho(x1, x2, ..., x^{|\mathcal{X}|}) = 1$. A graph is called singular iff its operational domain is empty, i.e. function ρ is a contradiction: $\rho = 0$.

- Function $\varphi : (\mathcal{V} \cup \mathcal{E}) \to \mathcal{F}(\mathcal{X})$ assigns a Boolean condition $\varphi(z) \in \mathcal{F}(\mathcal{X})$ to every vertex and arc $z \in \mathcal{V} \cup \mathcal{E}$ in the graph. Let us also define $\varphi(z) := 0$ for $z \notin \mathcal{V} \cup \mathcal{E}$ for convenience.

This model is based on strict graphical representation, where each event is represented by a circle ◯, and each connection between vertices is named arc, depicted as an arrow ⟶. Both the previous elements are labelled with a predefined pattern composed by vertex/arc name, followed by condition $\varphi(v/e)$. Next to each graph, a further condition is present called "restriction function" (ρ), composed by operational variables $\mathcal{X}$.

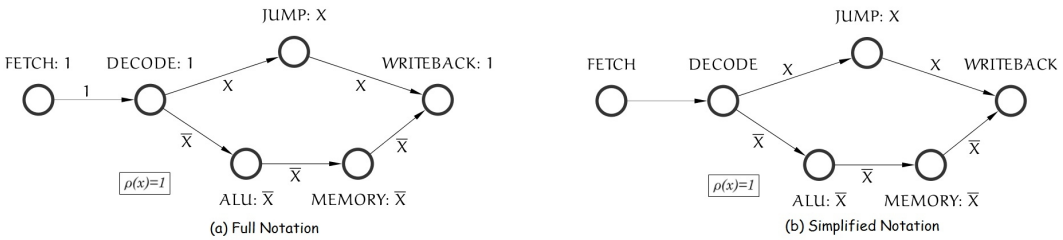


Figure 2.1: Graphical representation of a *CPOG*.

On figure 2.1, an example of a *Conditional Partial Order Graph* model is depicted. On the left of it all the Boolean conditions over each vertex and edge are listed, while on the right side of the Figure, a simplified notation is used. That is, where the element is always present in the model, a logic 1 is present, also indicated as no literals.

The purpose of the condition φ is to switch on/off vertices and edges, when the conditions on it are satisfied or not respectively. A clearer example could be observed on Figure 2.2, where the left side of the Figure shows the projection of the model if $\mathcal{X} = 1$, on the right side the projection on the other hand.

In this representation, dash edges and circles represent nodes and arrows switched off, in such a way not to affect the behaviour of that particular event class. Notice

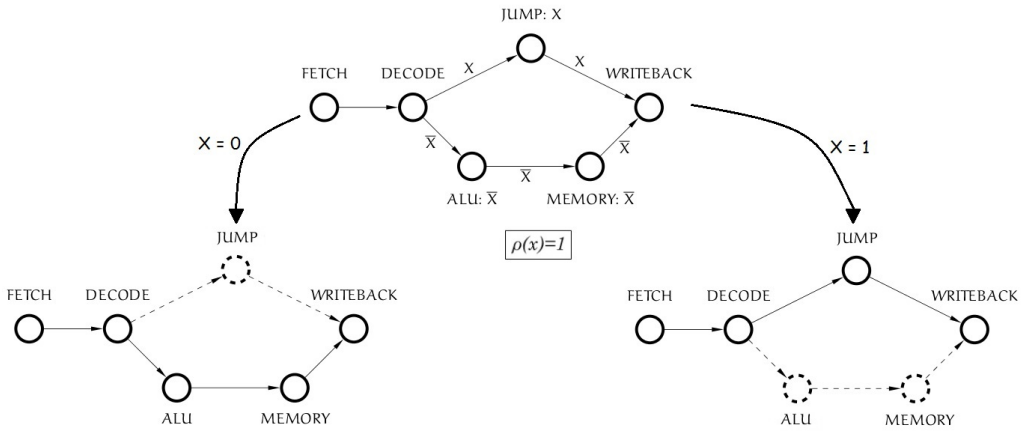


Figure 2.2: *CPOG* projections: $\mathcal{H}|_{\mathcal{X}=1}$ on the right, $\mathcal{H}|_{\mathcal{X}=0}$ on the left side.

that, the edges followed by a logic 1 are not present if one of the two vertices associated on it is not present too. It is a quite important feature, that allows representing the introduction of the Don't Care Boolean conditions.

The purpose of this research is expanding the next step, that is the synthesis phase when a *CPOG* is already optimised, ready to be translated into logic circuit. For this purpose, later on behavioural semantics of the model will be presented.

2.1.1 A factorial issue in the number of solutions

Once the *Partial Orders* have been prepared and optimised by the designer to model a system, each graph must be associated to an op-code in order to be distinguished by the remained ones. This is the first step of the synthesis stage, where the controller for managing the whole structure should be synthesised. Before discussing about the main issue I am going to handle during this research, let us define some useful words that will be used along the pages of the dissertation.

Op-code points out a variable length array of bits where each element can be a logic `0` or `1`, the following ones are examples of feasible op-codes $\{0010, 000000, 1110001, 0\}$. **Op-code ensemble** is defined as the the group of *op-codes* it is possible to use with a particular number of bits, for instance the *Op-code ensemble* of length 2 is composed by *op-codes*: {00,01,10,11} while the one of length 3 {000,001,010,011,100,101,110,111}. An **encoding**, also referred to as **solution**, is a subset of an *op-code ensemble* where each *op-code* is associated to one and only one *Partial Order* graph. Finally, the **solution space** is represented by the group of all the possible *encodings* for a particular CPOG. On Figure 2.3 is depicted the op-codes available to encode a CPOG that contains up to 8 *Partial Order*, composed by 3-bits op-codes.

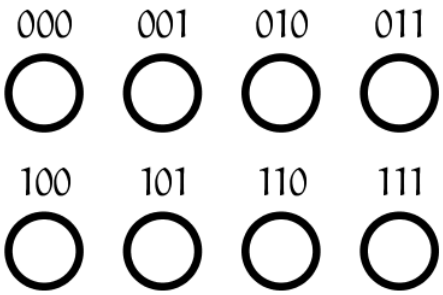


Figure 2.3: Op-code ensemble of length 3.

While on Figure 2.4, two possible solutions for what concerns a 5 graphs CPOG are depicted taking into account the op-code ensemble on Figure 2.3.

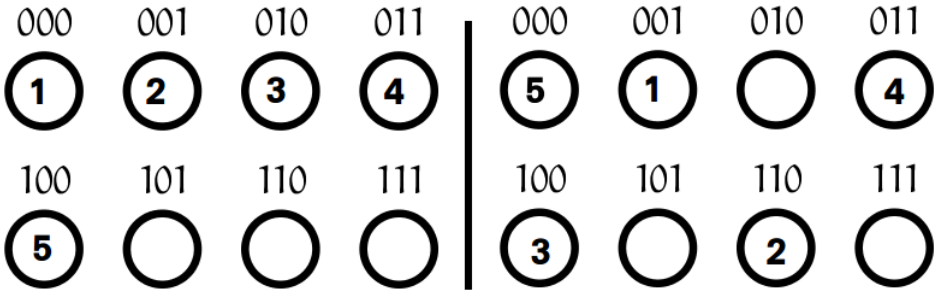


Figure 2.4: Two possible encodings for CPOG composed by 5 *Partial Orders* with 3-bits opcodes.

Moreover, another possibility designers have to exploit the higher size of the op-code ensemble with respect to number of graphs to encode, is represented by the *Don't care* conditions. For instance, on Figure 2.5 is represented the situation where the second *Partial Order* is encoded by the op-code 0XX.

On the light of above, as one might observe, there are plenty of possibilities to encode all the graph in a representation, therefore the size of the *solution space* can be really high depending on number of the graphs to encode and length of the op-code ensemble. Even though solutions allowing *don't care* conditions will be taken into account in this research, now I am going to neglect the *encodings* stem by them in the computation of the size of *solution space* due to the high complexity that might be followed. However this issue will be addressed on Chapter 5.

Referring to starting aim of this Section, the main goal of a designer, is the find inside the *solution space* related to a model, the best possible *encoding* in terms of area of the final controller. Due to the lack of a heuristic cost function able

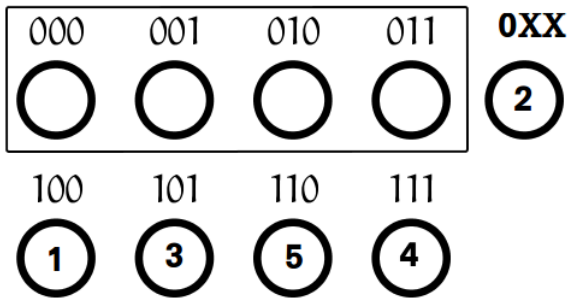


Figure 2.5: Encoding for CPOG composed by 5 *Partial Orders* with 3-bits opcodes exploiting don't care.

to inspect cleverly each *solution*, designers needed to apply each different *encoding* to the representation and to synthesise it on a real circuit in order to be able to compare the various *solutions*. Nonetheless, as one might observe, it is extremely time consuming, in particular for the CPOGs contain several graphs internally. Indeed, as this Section aims to explain, the size of the *solution space* is directly correlated with the number of *Partial Orders* the model is composed by and by the length of the *op-code* designer wants the graphs the be encoded with. I

Hence, the size of the *solution space* is of primarily importance because potentially, all the *solutions* might become a good *encoding* in terms of area and should be inspected. As a consequence, an increase in the size of the possible *solutions* may mean a higher complexity during the search phase and might worsen the final solution. This is why the main aim at this step is trying to reduce the size of the *solution space* as much as possible.

Hence, the need of a function able to target a subset of *encodings*, without going through the entire *solution space*. Before concentrating on it, let us discuss about the size of the *solution space*, it is useful to understand how we can approach the problem.

Let us consider a very small *Conditional Partial Order Graph* representation, composed by 4 different graphs only. As analysed in [3] and briefly above, there could be several approaches to encode various graphs in order to minimise the Boolean function in each vertex and edge, they could be encoded on minimum number of bits, or by using an one hot encoding fashion for instance. What we are going to take into account below will be *op-codes ensemble* with minimum length related to number of *Partial Orders*.

It means that in order to encode 4 different graphs, we need 2 bits (*Op-code ensemble* of length 2). Therefore, in order to get the minimum number of bits needed to encode an entire model we need to use formula 2.1, where m stands for the size of current *op-code ensemble*, assuming the graphs to be encoded with

op-codes on minimum number of bits.

$$\#\{\mathcal{B}\} = \lceil \log_2(k) \rceil \tag{2.1}$$

In this case, number of *encodings* fit perfectly graph to encode since with two bits it is possible to encode exactly four elements ($2^2 = 4$). Afterwards, in order to compute the entire solution space of such instance we should refer to permutation problem. It states that the total number of solutions we can select by sorting elements differently each time in a group of m different elements would be:

$$\#\{\mathcal{S}\} = \frac{m!}{(m-k)!} \tag{2.2}$$

where k is the number of graphs to encode. Since in such case $m = k$, formula comes up from example just mentioned depicted below:

$$\frac{4!}{(4-4)!} = \frac{4!}{1} = 4! = 24$$

Thus, total number of solution in this case is 24.

Moreover, there is another issue we have to take into account for reducing the size of *solution space*. The *encodings* I just considered contain all the possible combinations come up by permuting all the elements inside the *op-code ensemble*. Nonetheless one could reduce the *solution space* even more by fixing the first element of the *encoding*, without loosing any good results. It is because for each *encoding* present in the *solution space* can be found a complementary *solution* which comes up with the same final controller as for the complementary *encoding* with some inverter gates at the starting point. Thus with no area gain at all.

Hence, by taking into account statement before, we could modify formula 2.2 into:

$$\#\{\mathcal{S}\}' = \frac{(m-1)!}{(m-k)!} \tag{2.3}$$

It is extremely beneficial for the size of the *solution space*. By analysing problem before the ensemble reduces as following:

$$\frac{(4-1)!}{(4-4)!} = \frac{3!}{1} = 3! = 6$$

Since the gap in the *Partial Order* composition did not allow to automatically compose the graphs obtaining the area of the final controller. In order to first analyse the size the final controller comes up with various *encodings* and try to understand whether a correlation exists. I have focused on very small *Conditional Partial Order Graph* models, as it will be discussed on next section.

2.1.2 Examples considered

In order to be able to inspect all the solutions for a particular Partial Order Graph, at the beginning just smaller graphs have been considered. It is because, the number of solutions actually available depends on the number of Graphs in a single representation in a factorial fashion, thus the more the number of *CPOGs*, the more the solutions available for the entire model as argued in chapter 2.1.1.

Hence, models analysed in order to see whether a correlation is present are the following ones, taken from [3]. As depicted in Figure 2.6, it is one of the most basic model for four arithmetic instructions of an *ALU* unit.

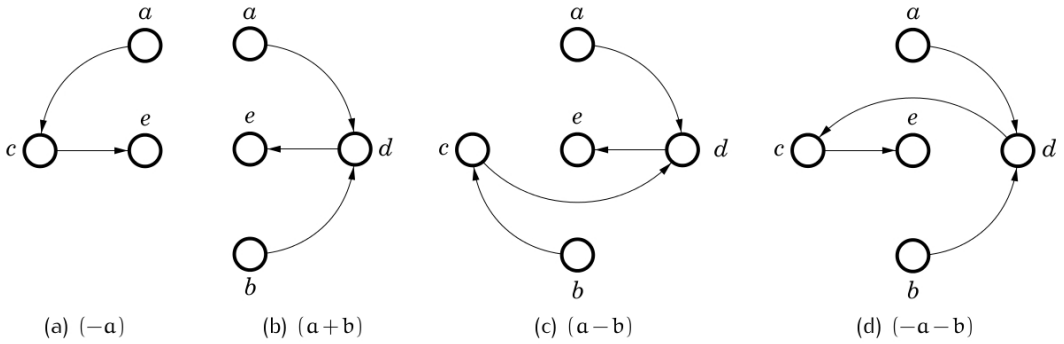


Figure 2.6: Simple *CPOG* models four basic operations of a *ALU*,([3], Figure 7.9).

A and *b* nodes represent load of a and b register respectively.*C* represents inverting operation, *d* the addiction and *e* the store instruction. As one could see this model is compact, and powerful at the same time, and may be able also used to model quite complex instructions, belonging to different kind of sectors inside the CPU, into very comprehensible and manageable graphs. This first example, is quite basic and simple, and allowed me to understand how I could deal with this representation.

Then I handled other two representations, always composed by 4 *CPOG*, in such a way to keep the number of solutions reduced to 6 (by fixing first element), so that to inspect well the entire solution space. The models used were taken from a more bigger *CPOG* composed by eight instructions classes who will be analysed separately later on.

At this point, I analysed as two separate models the graphs $\{a, b, c, d\}$ and $\{a, c, g, h\}$ which may be seen as the representation of the path an instruction should go through inside a general processor: each node indeed, corresponds to a different stage of CPU part. For instance *IFU* stands for **I**nstruction **F**etch **U**nit, the component in charge of fetching next instruction from Instruction memory. *MAU* stands for **M**emory **A**ccess **U**nit, *PCIU* for **P**rogram **C**ounter **I**ncrement **U**nit, the

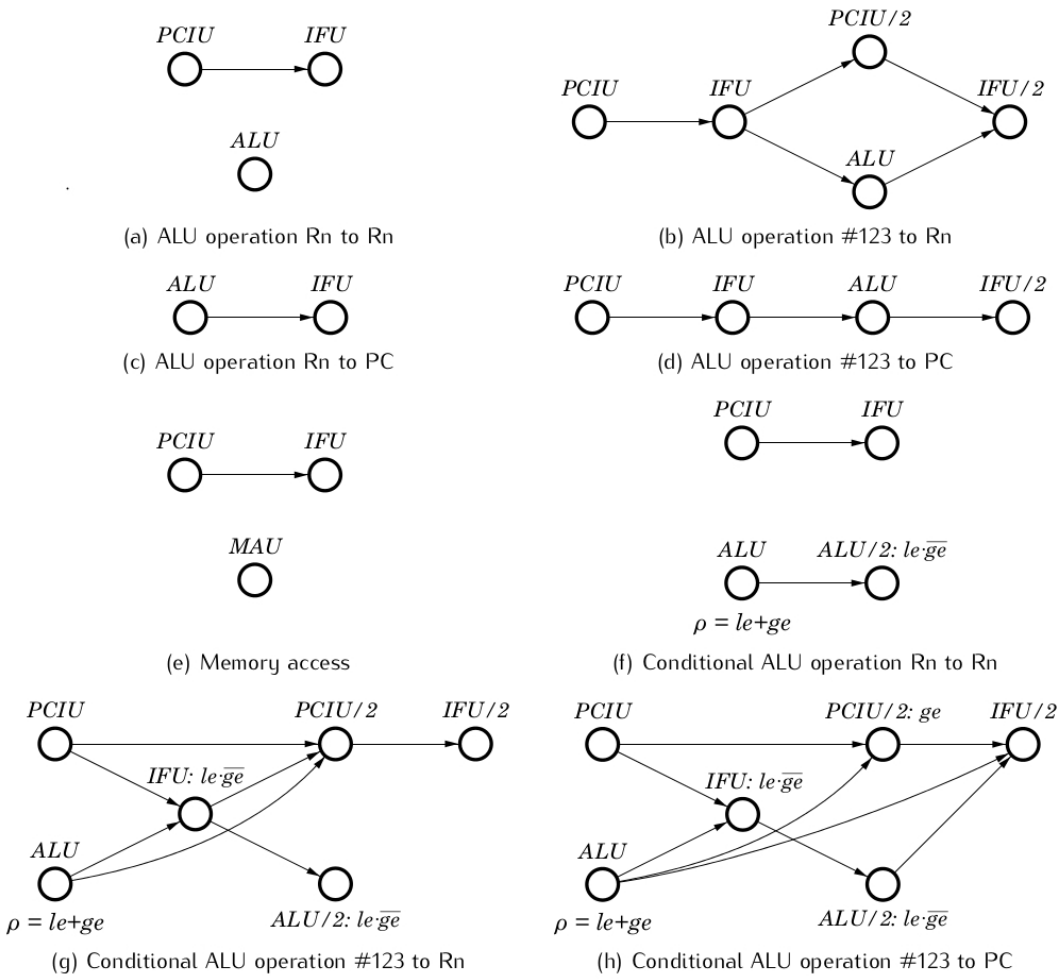


Figure 2.7: 8 instruction classes model.

one able to increment Program Counter register in order to select next instruction, and so on and so forth.

Since *Conditional Partial Order Graphs* is an *acyclic directed graph model*, when the flow should pass twice through a single node, another node, referred to as "/2" upfront is needed, just for distinguishing among two different stages of the same vertex.

Another issue we should address before working with *CPOG* representation, is conditions on vertices. As in the node *ALU/2* on f graph on Figure 2.7, nodes might have a condition on it; it means that the vertex, and the associated edges would be present inside the graph, if and only if the conditions on that vertex are satisfied. The theory of how to synthesise this kind of particular nodes is reported in [3].

2.1.3 Boolean equations extraction for each CPOG element

First of all, what I had to do was extracting the Boolean conditions for each element of the graphs. Very basic operations are performed to reach this goal out, which may be done manually on such small graphs. Over this Section these kind of operations are illustrated and explained, with the help of short examples.

In order to get the Boolean equations of an element (either node or edge), one needs to represent it by means of Boolean conditions on each particular *CPOG* projection. Concerning a node: it could be present or not, depending whether it would affect or not the behaviour of a projection respectively. If it does one should represent the node with a `1`, otherwise with `0`. On the other hand, an edge could be more difficult to manage, because we have to take into account some properties of the model in order to optimise it as much as we can, such as the transitive property depicted on the Definition below.

Definition[2]. A dependency $a < b$ (where $<$ denotes edge from a to b $a \rightarrow b$) between events $a, b \in \mathcal{S}$ in a partial order $\mathcal{P}(\mathcal{S}, <)$ is called transitive (denoted as $a << b$) iff there exists an event $x \in \mathcal{S}$ such that both conditions $a < x$ and $x < b$ hold:

$$(a << b) = \exists x \in \mathcal{S}, (a < x) \wedge (x < b) \tag{2.4}$$

Therefore, if in a particular graph two vertices are not directly connected, but they are in a transitive manner, one has to represent the edge in the projection with a Don't Care `X`. Let us consider for instance the Partial Order Graph on the Figure 2.8.

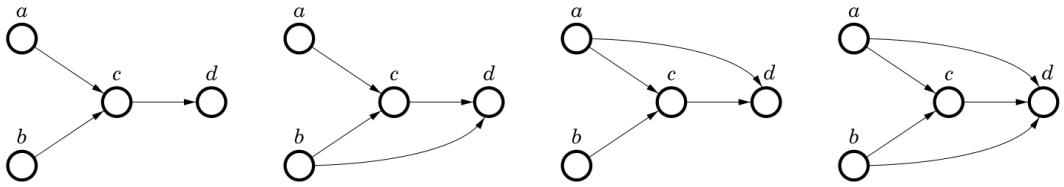


Figure 2.8: Partial Order Graphs example ([3], Figure 2.5).

The left-most image represents the reduced graph and on the Table 2.1 is depicted an example of the right way to assign a Boolean condition for that particular graph. As we can notice, $a < d$ edge for instance is represented by a Don't Care condition; it is because a is connected to d by a transitive edge, as in the third graph from left of Figure 2.8.

[2]**Definition 4.2 from [3]**

Edge	Bool. Condition
$a < b$	0
$a < c$	1
$a < d$	X
$b < a$	0
$b < c$	1
$b < d$	X

Table 2.1: Example of Boolean conditions assignment of graph on Figure 2.8

Another rule we have to follow to assign conditions to each arc is the following: if at least one out of two vertices of the edge one is considering is not present, a Don't Care condition (`X`) might be assigner to that edge. It is because, no matter the presence or not of the edge, it would not influence the projection at all since the nodes are not present.

Once an encoding for each of the *CPOG* class has been chosen, one has to minimise the condition for each element via a logic minimiser. I used *Karnaugh Map* method when I faced with simple 4 instruction classes *CPOG*, while *Espresso* logic minimiser [13] while dealing with more complex representation.

On the light of above, I would like to show to reader the results of Boolean conditions extraction starting from a couple of encodings. They have been applied to *CPOG* showed on Figure 2.9, Boolean conditions are reported on Table 2.2. As explained on Section 2.1.1, there are 6 solutions per model (the small ones), but for simplicity's sake, just two of them will be reported, the most meaningful.

As one may notice, the first encoding looks way much better than the second one in terms of number of literals present in the Boolean equations. On the light of above, over the next section these conditions would be exploited in order to compute equations for the final controller, area measurements will be performed on it.

2.1.4 Boolean equations extraction for the model

In order to extract all the Boolean equations from the *Conditional Partial Order Graph* model for developing the final circuit implementations, various steps are needed. Even tough they are described in [3], I want to outline some behavioural features of the representation.

As discussed on Section 2.1, when a *CPOG* is ready to be synthesised into Boolean equations, each vertex and edge have a Boolean condition associated to itself. And according to these conditions, a part of the graph would be present or not dynamically in the actual representation if and only if that condition is satisfied.

	Encoding Solutions	
Elements	**00 01 11 10**	**00 11 01 10**
IFU	$!X0+!GE$	$((!X0*!X1)+(X0*X1))+!GE$
$IFU < PCIU/2$	1	1
$IFU < ALU/2$	1	1
$PCIU/2$	$(GE*X0)+(X0*X1)$	$(!X0*X1)+GE*(X0*!X1)$
$PCIU/2 < IFU/2$	1	1
$ALU/2$	$X0*!GE$	$!GE*((X0*!X1)+(!X0*X1))$
$ALU/2 < IFU/2$	$!X1$	$!X1$
$IFU/2$	$X0$	$(X0*!X1)+(!X0*X1)$
$PCIU < IFU$	1	1
$PCIU < PCIU/2$	1	1
$PCIU$	$X0+!X1$	$!X0+!X1$
$ALU < IFU$	$X0+X1$	$X0+X1$
$ALU < PCIU/2$	1	1
ALU	1	1

Table 2.2: Boolean equations for each element for *CPOG* on Figure 2.9

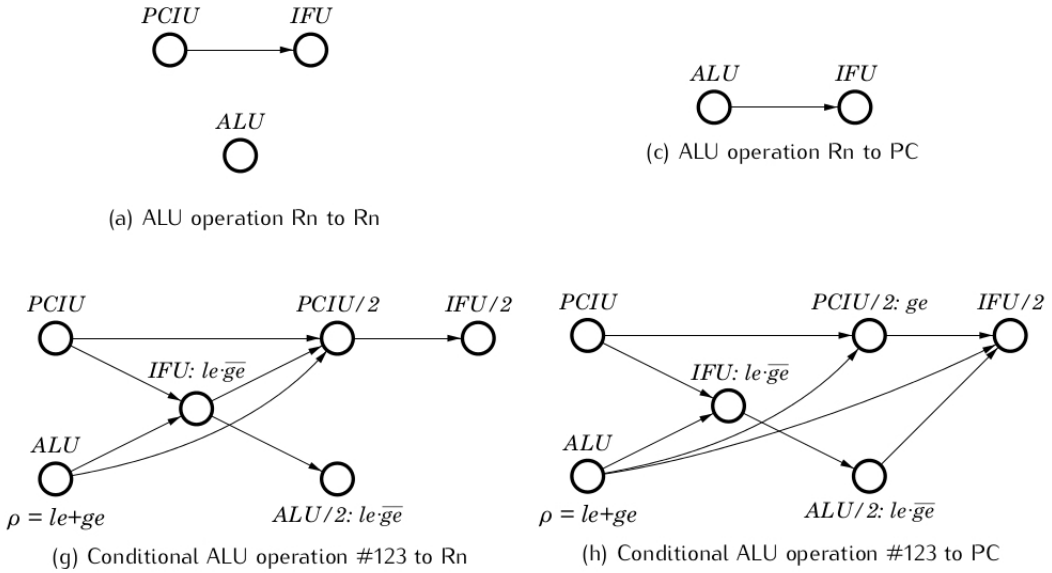


Figure 2.9: *Conditional Partial Order Graph* with 4 instruction classes.

Before discussing about synthesis process, it is worth enunciating following definition:

Definition[3]. Opcode $\psi : \mathcal{X} \cup \mathcal{Y} \to \{0, 1\}$ assigns Boolean values to all the static and dynamic operational variables in the graph. The preset $\bullet_\psi v$ of a vertex $v \in \mathcal{V}$ with respect to opcode ψ is:

$$\bullet_\psi v := \{u \in \mathcal{V},\ \varphi(u)|_\psi \cdot \varphi(u, v)|_\psi = 1\} \tag{2.5}$$

It contains all the vertices $u \in \mathcal{V}$ which precede vertex v in the partial order defined by complete projection $\mathcal{H}|_\psi$.

By considering definition just listed, I can analyse the mapping procedure for building the controller. During this phase a set of equations are produced. The overall size of them is thoroughly correlated to the size of the starting *CPOG* in terms of number of graphs and conditions on their elements, as already pointed out before. Therefore, applying optimisation techniques, as described in [3], before synthesising the final controller is extremely important in order to reduce the whole size of the circuit. Aim which must be also pursued by selecting the best op-code assignment, as this dissertation want to demonstrate.

[3]**Definition 5.17 from [3]**

Andrey Mokhov, the author of *Conditional Partial Order Graph*, describes mapping procedure in the following way:

Mapping procedure.([3], p.93) "a vertex $v \in \mathcal{V}$ is enabled to fire if and only if:

1. It belongs to the current complete projection, i.e. its condition is satisfied: $\varphi(v) = 1$;

2. All its preceding vertices have already fired, i.e. $\forall u \in \mathcal{V},\ (u \in \bullet v) \Rightarrow fired(u)$.

This can be captured in terms of Boolean equations as follows:

$$enabled(v) = \varphi(v) \cdot \prod_{u \in \mathcal{V}} \{\varphi(u) \cdot \varphi(u, v) \Rightarrow fired(u)\}$$

Now predicates $enabled(v)$ and $fired(v)$ should be replaced with real signals."

For instance, one might use an asynchronous-based interface where the transactions are managed by a central control-unit (as depicted on Figure 2.10) by a request-acknowledgement handshake interface. The $enabled(v)$ might be represented by signal req_v, and $fired(v)$ by signal $ack(v)$.

$$req(v) = \varphi(v) \cdot \prod_{u \in \mathcal{V}} \{\varphi(u) \cdot \varphi(u, v) \Rightarrow ack(u)\} \tag{2.6}$$

In simple terms: an event is allowed to send a request signal as soon as preceding event has been completed (signalled by an acknowledgement).

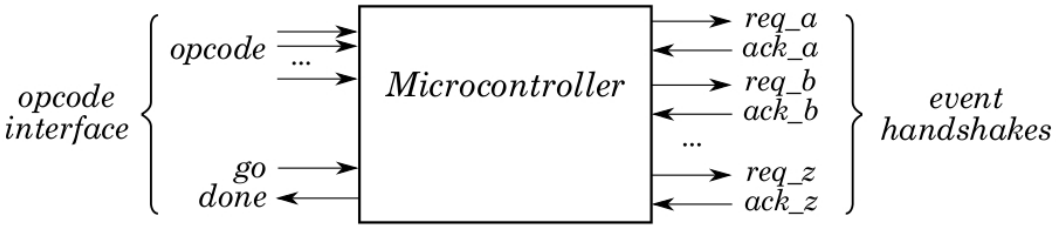


Figure 2.10: CPOG-based microcontroller with acknowledgements/requests interface, taken from [3], Figure 7.4.

Once discussed about rules on how to extract Boolean equations from *CPOG* model, the only thing one needs to care about is grouping together each Boolean function for every vertex, in order to feed *ABC* tool and minimise Boolean equations. In such a way to perform a fair comparison for what concerns the encodings applied.

2.2 Dataflow computing

Dataflow structure is a special kind of architecture which totally differs from control-flow like structure, because data can freely flow through the execution part of the computation, avoiding the decoding part of the instructions typical of a standard CPU of a CISC/RISC-based processor. In a general purpose processor indeed, before an instruction is executed, it must go through various stages of the pipeline which are useful to the CPU to fetch such instruction from the memory, and then to decode it. Hence, the typical stage an instruction must follow for being executed are the fetching phase and the decoding one; afterwards (depending on the instruction) even the operands the computation is based on must be fetched from the data-memory (in case of *Harward architecture*[4]) in order to have the operands of the computation ready for it.

Since all these operations are executed in parallel, the throughput of the CPU, which represents the rate of instructions to be executed in a specific amount of time, is not affected by such a long pipeline. It is implemented in this way in order to minimise the number of times CPU must wait for the result of any previous instructions of the program. Nonetheless, although this paradigm tailors well to really general purpose applications, that is where the user may carry out very different operations. It does not fit well to some particular kind of applications where, even tough the variability of the behaviour of the system can range from 1 to n predictable operations, such a big waste in terms of area can be avoided.

Even though the throughput of the processor is not affected by the control flow architecture, area is. As one might observe on Figure 2.11 indeed, the computational part which is the only really useful area dedicated to application occupies the minority of the whole chip. The one on the Figure is just a general example, but it is enough to make reader understand how much area one might save by avoiding the whole memory and control flow architecture, even if in this case the hardware would lose flexibility. Over the past few years dataflow computing has been one of the main research area of many companies, such as for Maxeler Technologies as one might read on [56] for example.

A good tradeoff could be obtained by making the dataflow structure reconfigurable. It means having various dataflow systems that can be also very different each other, but that can be interchanged as soon as user needs it. What one gains is having as many structures as wants where data can directly flow through, the area of the controller for managing the reconfigurability part may be negligible with respect to the area dedicated to computation and there is not an increment in the

[4]Harvard architecture differs with respect to Von Neumann structure because data and memory are stored in two different memory elements.

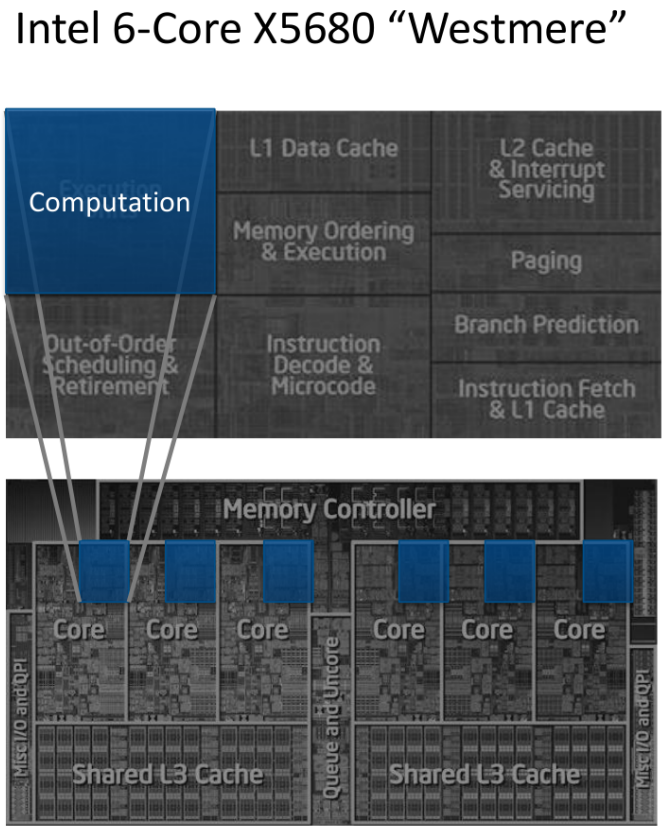


Figure 2.11: Particular of Maxeler Technologies transparencies, Slide N. 5 “Introduction of Dataflow computing” [30]

delay since all the structures are already synthesised onto the ASIC/FPGA chip under usage.

The really advantage of the representation I am going to use is the possibility to share different internal components between the dataflow structures available. Indeed *CPOG* allows designers to synthesise a controller able to activate the part actually needed for the computation without synthesising it multiple times onto the chip, getting a higher save of the area too.

In this Section I am going to introduce the dataflow paradigm, focusing on the typologies that such structures may have.

2.2.1 Dataflow theory

Systems can be divided in two categories: *control-dominated systems* and *data-dominated systems*. The former ones are strongly dominated by taking decision

and by the latency, the computation time is not a main concern provided that it is lower than the clock period superimposed by the external environment. The latter one whereas are structures where the main concern is on data computation and the throughput. Here each part composes such systems is autonomous and often asynchronous.

In this paragraph I am going to cover three dataflow networks:

- Kahn process neworks
- Static dataflow neworks
- Boolean dataflow neworks

Maurizio Tranchero et al. describe in [31] the Kahn process as networks of processes made by sequential deterministic code, where each state communicates with the next one via a point to point FIFO[5] connection. Reading operations are blocking, it means that a process does not go on until data is present on input FIFO. It guarantees the determinism, but may cause deadlock if such process does not receive any more data in input. Moreover another problem that affect such structures is the infinite memory requirements, due to various speed of the processes. A practical example may be found on [31] on pages 102-103.

Static dataflow networks are a subset of the *Kahn process networks*, where the deadlock and the problem of infinite memory requirement is addressed. Each process indeed can be executed repeatedly reading and producing a fixed number of data, when it is present in its input. Moreover the position of the initial values are fixed at the beginning and they are called *tokens*. On Figure 2.12 an example is depicted.

The red dots represent the token, the numbers close to each state are the reading/production rates depending whether the arrow goes in or out respectively from the process, for instance process `C` takes 2 token and produces 3 tokens on the next FIFO each time. If the reader follows the sequence of events may see that this process may be repeated infinitely. In this case, the way the processes are scheduled is `BBCBBCDA`, or better `2(2(B)C)DA`. It means that, if the processes follow this pattern can be executed without occurring into a deadlock and with the maximum size of a FIFO set by the connection between the process `C` and `D` which must able to contain 6 tokens.

The *Boolean dataflow networks* add two nodes to *Static dataflow networks*: split and merge. They are needed to express Boolean conditions for certain kind of applications like compression/decompression (as stated in [31]). Such structures are showed on Figure 2.13 and can be used to develop *loop* and *if-then-else* patterns.

[5]FIFO = First IN First OUT.

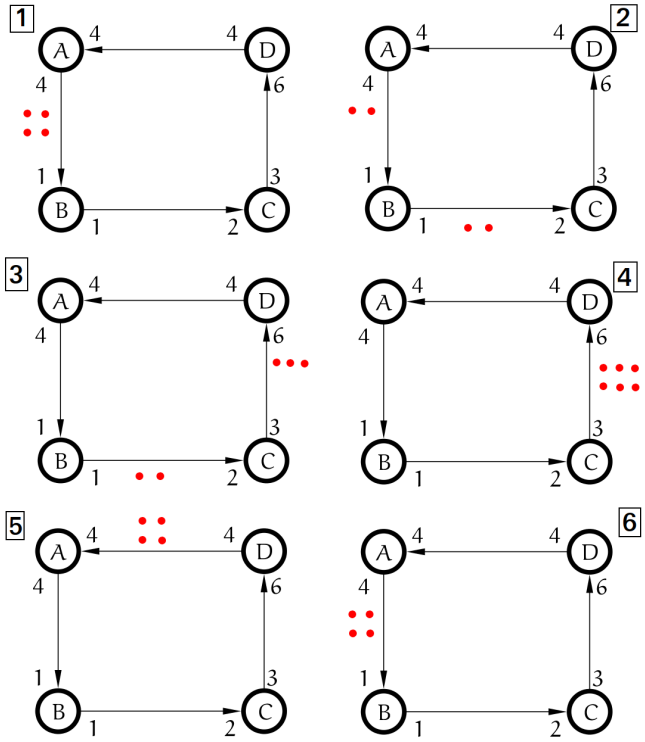


Figure 2.12: Static Dataflow Network example

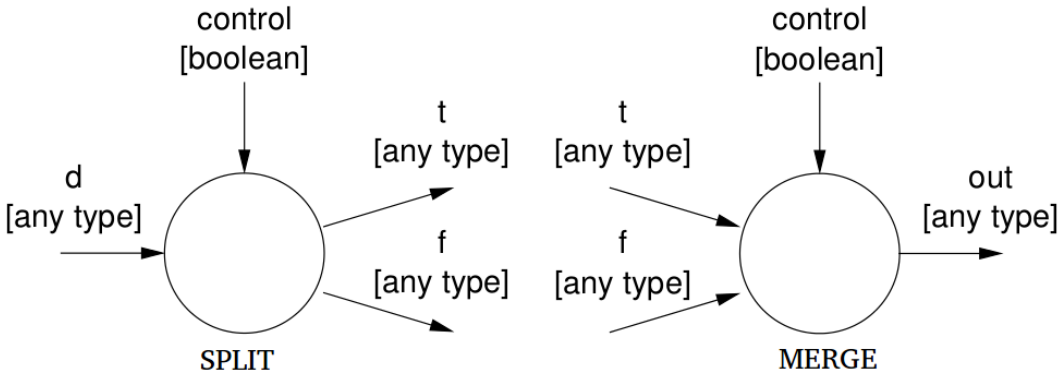


Figure 2.13: Split and merge structures. ([31], Figures 6.12 - 6.13)

In summary, there can be different typologies of dataflow structures. The ones I am going to deal with are *static dataflow network* where the rate of reading/production is 1, and where the processes start executing itself when data is present at the input. As already mentioned, this representation I am going to use fits well to

asynchronous structures which do not need any memory between two processes due to protocol implemented.

Chapter 3

Asynchronous dataflow pipelines design

In this Chapter I am going to discuss about the design of dataflow pipelines, pinpointing on self-timed structures. As already mentioned before, several devices may benefit from asynchronous pattern usage for different reasons: first of all the lack of the clock-net extremely reduces the size of the design which can be really affected by such component. It affects also the power consumption, since even when the device is not working the clock is always active consuming power constantly. Designers tried during the last few years to lower such a source of power consumption by means of techniques as *clock gating*, or by acting at the RTL Level pre-computing the result for what concerns common case computation [11]. But it turns out that such modifications contribute to an higher size of the design as well increasing the static power consumption which is getting bigger than dynamic one nowadays [12].

Asynchronous designs may contribute solving such problem due to the capability to work when the data is ready only. Self-timed hardware structures indeed, consume an extremely small amount of power consumption when the data is not ready at the inputs. Even though such structures may bring great advantages to VLSI market, they have been neglected for many years also due to the complexity of the development phase, in particular from the protocol point-of-view. One of the aim of this research is to simplify such design-flow, in order to lead designers to use such structures more easily.

On the light of above, I am going to present all the tools a designer may need for developing self-timed hardware. The reconfigurability of the dataflow graphs will be discussed over the next Chapter.

3.1 Pipelining introduction

Before going through the core of this Chapter, it is worth giving a short introduction to readers about the general concept behind the *Pipeline*. In the context of Computer Science, a pipeline is defined as a set of processing parts, completely separated each other, which are connected in succession to form a system.

The singularity of this structure is the capability of each single module the pipeline is composed by to work without the influence of the preceding parts. Let us think for example about a general pipeline inside a RISC-based *CPU* for executing instructions, it is composed by five computational elements: **Instruction Fetch** which is in charge of getting the instructions from the memory (typically instruction memory), hereinafter called **F**. **Instruction Decode** that is the hardware part that decodes the instruction to figure out which operation needs to be performed and to fetch the operands for the computation, called **D**. **Execute/E** which is in charge of actually executing the instructions, **Memory access/M** that load/store instructions into memory and finally **Write Back/WB** which writes the results into a processor registers or forwards them into the ALU whether some particular conditions occur. Such structure is depicted on Figure 3.1

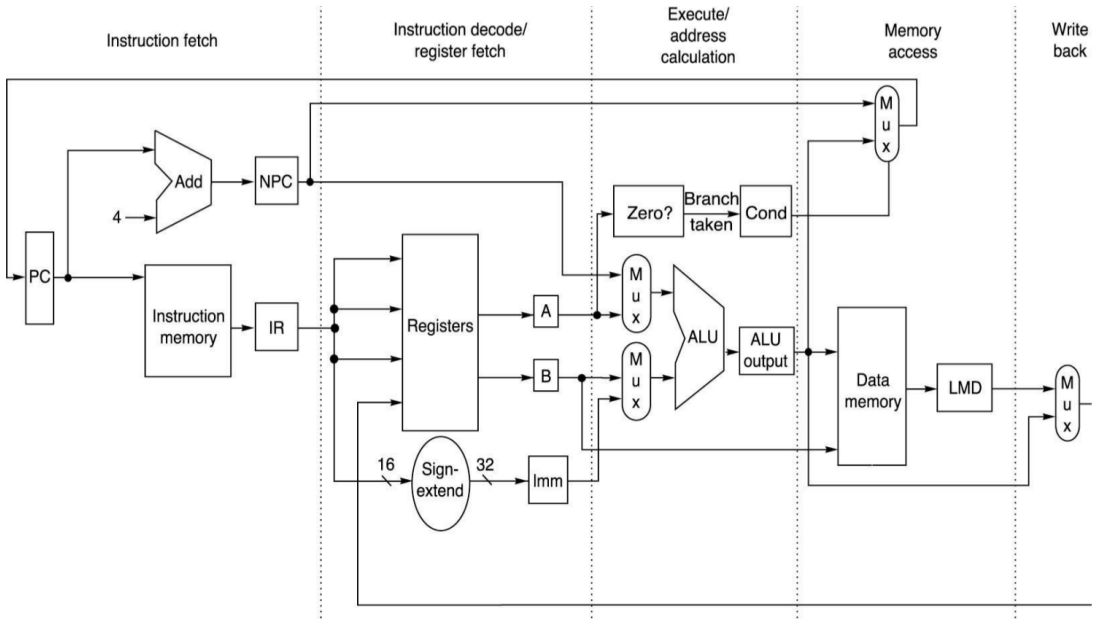


Figure 3.1: General structure of a RISC-based processor. ([32], P. 13)

As one may notice, this structure is not pipelined because it lacks of the structures needed to store the results in the middle of the computation, that are the

registers. A more complete design is showed on Figure 3.2 where the registers are present between every two consecutive stages of the pipeline.

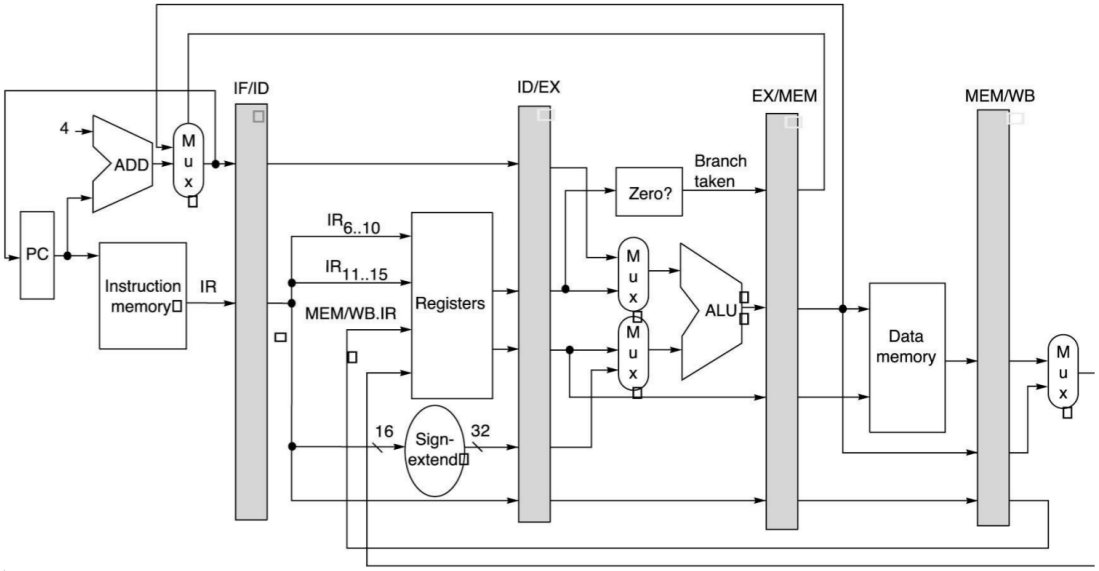


Figure 3.2: Pipeline version of a RISC-based processor. ([32], P. 16)

The hardware structures illustrated before are synchronous. Synchronous design benefits from Pipelining-based structures because of the reduction of the critical path and as a consequence the increasing frequency the microprocessor is able to run on. One of the parameter which contributes to *maximum operating frequency* of the circuit under design is the *critical path*. It is a time measure and is defined as the slowest path in the circuit present between two memory elements (registers) and it is connected to the frequency by following mathematical relationship:

$$\mathcal{F} = \frac{1}{\mathcal{CP}} \tag{3.1}$$

Therefore, the longer the critical path, the lower the frequency the CPU can run. On the light of above, one might observe that the advantage of inserting the registers in the middle of the computational units is reducing drastically the critical path and as a consequence thoroughly increasing the working frequency of the circuit.

Nonetheless, pipelining does not lead to advantages only, because first of all it increases the area of the circuit either due to the high number of registers needed to store all the results in the middle of the pipeline, and in particular for all the management units in charge of controlling the good behaviour of the structures and avoiding problems as hazards. In this brief description, I am not going to cover all

the problems stem from the pipeline-based architecture design, but if the readers is interested on it, they can find several information in [32].

A fundamental characteristic of a pipeline is the *throughput*. It is defined as the rate of production of a structure over a time unit and it is crucial in the concept of digital circuit nowadays. In order to better understand this concept I am going to present a short example taking into account the structures presented before on Figure 3.1 and 3.2. Let us assume that the time needed to execute each computational part (**F-D-E-M-WB**) is 1 μs both for pipelined and not pipelined-based structures; and let us compare how much time would elapse to execute five generic instructions consecutively assuming no hazards.

In the case of Figure 3.1, that is the not pipelined version, the μprocessor would execute each instruction one after the other, therefore in order to obtain the total time of execution one has to multiply the number of instructions times the time needed to process each instruction, as computed below:

$$ExecutionTime = N_{inst} \times INST_{time} = 5 \times 5\mu s = 25\mu s$$

For what concerns the pipelined version, in order to compute the overall execution time we have to consider that the instructions are internally executed in parallel, as represented on Figure 3.3

Clock cycle number									
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
IF	ID	EX	MEM	WB					
	IF	ID	EX	MEM	WB				
		IF	ID	EX	MEM	WB			
			IF	ID	EX	MEM	WB		
				IF	ID	EX	MEM	WB	

Figure 3.3: Pipelined execution example. ([32], P. 26 modified)

For instance, if one considers the operations under execution on fourth clock cycles, the ones inside the red circle, he might be able to observe that the first instruction is at the **Memory** stage, the second one at the **Execution phase**, the third one at the **Decode stage** and so on. In this case the overall time of execution is 9 μs.

Hence the total speedup one can get is:

$$Speedup = \frac{ExecTime_{nonPipelined}}{ExecTime_{Pipelined}} = \frac{25\mu s}{9\mu s} = 2.77$$

The considerations discussed above refer to synchronous-based structures and do not take into account all the hazards that may occur which might stall the pipeline slowing down the whole architecture. Nonetheless, they are enough to readers to understand the general concept in order to better follow next Sections, where the composition of an asynchronous pipeline is the main concern.

3.2 Asynchronous pipelining

Asynchronous circuits may have several advantages as introduced at the beginning of this Chapter. As Jens Sparso, from University of Denmark, points out in [33], self-timed-base structures may lead to great advantages such as running at high operating speed, since it is determined not by worst-case latency, but by the local one. [35], less emission of electro-magnetic noise [34], soundness towards variations in temperature, supply voltage and fabrication process parameters since the communication is based on handshaking protocols [36] and finally the lack of the clock skew problem due to the absence of the clock.

It is worth starting from a higher level of abstraction for analysing the functionality of an asynchronous circuit. As RTL (*Register Transfer Level*) represents a relatively good level of abstraction to understand what is going on inside the device regarding synchronous logic; *handshake-channel* and *data-token view* fit well to asynchronous circuits. The former models the signals connecting one register to next one, hence it represents the main link useful for communication purposes, while the latter one represents the data which is stored into the registers and that flows through the pipeline, where combinational logic between two registers should be as transparent as possible. Reader should think about a combinatorial block as a piece of logic which absorbs a token, performs its computation and outputs the results without looking at handshaking mechanism.

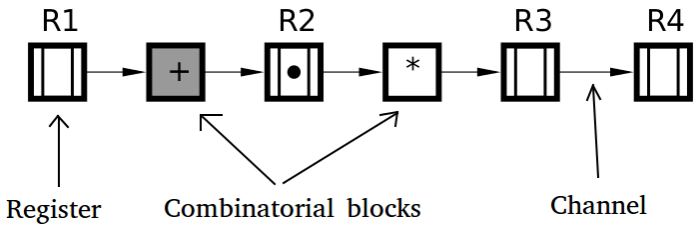


Figure 3.4: Dataflow asynchronous pipeline example.

As one might observe, dataflow representation tailors well to this kind of logic, and can be used for simplify the design-flow of such circuits. On Figure 3.4 are

depicted the main components I described above. The tool I am going to use for representing such structures is the *Dataflow Structure Plugin* under `Workcraft`, in this dissertation I am not going to cover the ideas behind it but if the reader is interested might find a good starting point about the topic on [37].

This section is structured as following: in the next part I am going to introduce the protocol I used for the asynchronous transactions and then the so called *Muller pipeline* by illustrating some circuits examples for understandability's sake.

3.2.1 4-phase dual-rail protocol

The advantage of having a synchronous architecture is the capability by all the elements to ideally share a predefined timing set by the clock signal. It should ideally come to each memory element at the same time, giving the possibility to every combinatorial part to finish its execution. It extremely simplifies the design process since designers may neglect the hazards that may happen between two clock ticks.

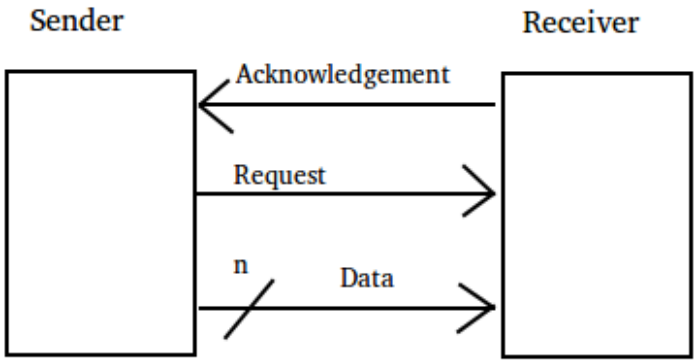


Figure 3.5: Typical handshaking protocol structure.

Asynchronous architectures, even tough bring many advantages, do not have the capability to share a single notion of time, that is the reason why a handshaking protocol must be used to fill in the gap of communication between two memory elements. A handshaking protocols allow two elements to communicate each other in a really flexible way, since communication may happen in whichever time period, no matter delay variations. Typically, three elements are present as showed on Figure 3.5: *Data bus* (composed by 1 to n wires) which propagates the token, that is the data useful to application; the *Request* which is the link the sender raises up to signal that the communication can start, and the *Acknowledgement*, that is the response of the receiver signalling that communication terminated correctly.

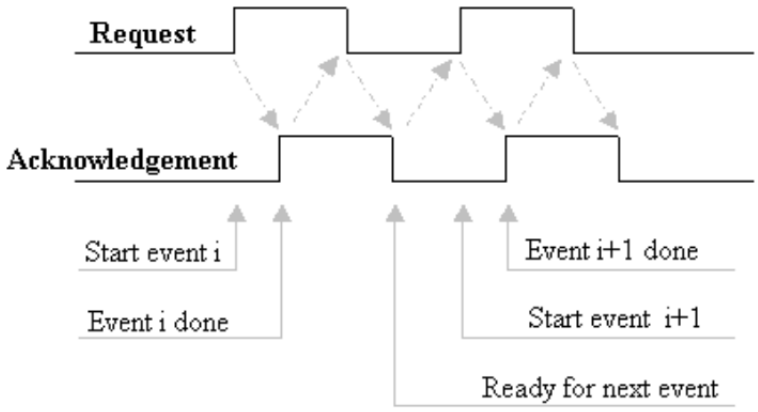


Figure 3.6: 4-phase asynchronous protocol behaviour.

As may be seen on Figure 3.6, such protocol also called *Return to Zero* because of its characteristic to zero the request signal after every transactions, works as follows: after the sender has set data available on the *Data bus* it raises the *Request* signal up; afterwards *Acknowledgement* is activated, once that the receiver has read the data correctly. Finally *Request* signal is deactivated followed by the *Acknowledgement*.

Although this protocol is simple, it is not the best possibility for what concerns power consumption. In fact, the transition to return to zero state could be avoided and power consumed for such transition saved (as implemented under the 2-phase protocol). Nonetheless, since the main purpose is to simplify the design phase I have used the 4-phase version of the protocol, even if the 2-phase version may be analysed in further research direction.

Nevertheless, the protocol I have used throughout this research is a bit different to the one described above. It embeds the *Request* connection into the *Data bus*, by avoiding any problems for what concerns the timing assumption one should take into account for combinatorial parts of the circuit. In order to simplify the concept of my previous statement for the readers, it is worth introducing one of the main problem occurring when designers deal with asynchronous circuits. That is, how could one figure out when a general combinatorial part has finished its execution, taking into account problems which might occur?

A first solution, nowadays still used ([4]), is the *bundled-data* protocol. It consists on setting some delays matching the ones of combinatorial modules for propagating the request signal delaying it for as much time as needed by the glue logic to put the result over the output bus. As one may figure out, this technique is extremely unreliable if the delays are not matched correctly to corresponding modules. It is not an easy task, due to the high number of parameters the speed of a module is affected of, in particular for the process variations, the temperature or the power

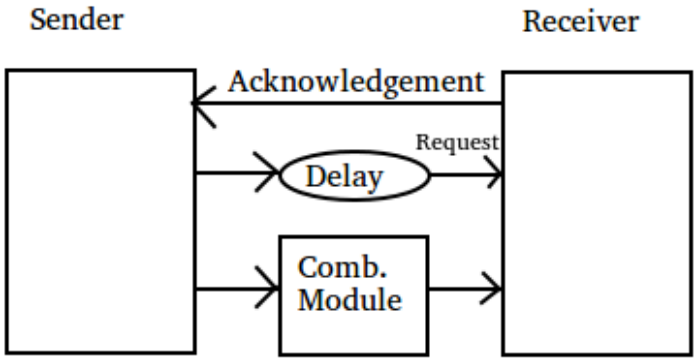


Figure 3.7: Bundled-data protocol model.

supply voltage swings. On Figure 3.7 a Figure representing such architectural choice.

To overcome this problem, I used the *dual-rail* extension of 4-phase protocol. It consists on double each wire in order to have one single wire in the case `1` need to be transmitted and another one in the case of a `0`. On Table 3.1 is showed the small truth table which might help to figure how the functionality of such structure.

Wire.T	**Wire.F**	**Sig. Transmitted**
0	0	Empty
0	1	False
1	0	True
1	1	Invalid pattern

Table 3.1: Communication on dual rail protocol.

The peculiarity of this protocol is the possibility to have one more logic state regardless the typical *True* and *False*, the *Empty* state. It signals the non presence of the data token between the sender and the receiver, therefore it is useful for the handshaking because it can be used to capture the presence of the result after a logic block, embedding the request signal into the data-bus and as a consequence avoiding every delay dependency with the timing path. On Figure 3.8 the abstract representation of such architecture.

3.2.2 C-element

Finally, to complete an asynchronous pipeline one needs another element which is commonly used when dealing with self-timed structures: *Muller C-element*. Jens

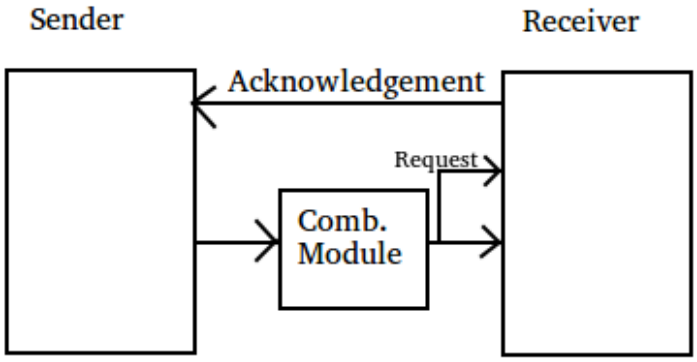


Figure 3.8: 4-phase asynchronous dual-rail protocol model.

Sparso defines the C-element as "a state-holding element much like an asynchronous set-reset latch. When both inputs are 0 the output is set to 0, and when both inputs are 1 the output is set to 0. For other input combinations the output does not change. Consequently, an observer seeing the output change from 0 to 1 may conclude that both inputs are now at 1; and similarly, an observer seeing the output change from 1 to 0 may conclude that both inputs are now 0."([33], Pag. 15). As stated by the author of "*Asynchronous circuit design - a tutorial*", this element might help designers to understand whether both inputs are present or not, observing the gate by the output side only.

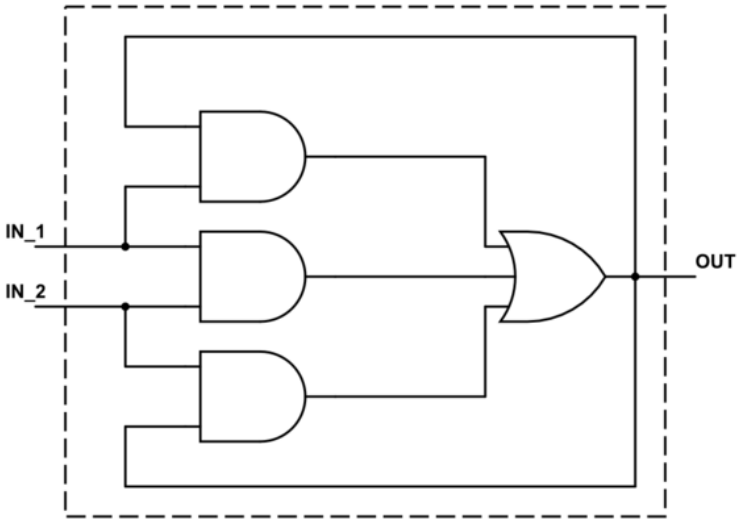


Figure 3.9: Schematic of 2 inputs c-element.

IN_1	IN_2	OUT
0	0	0
0	1	No change
1	0	No change
1	1	1

Table 3.2: 2 inputs c-element truth table.

On Figure 3.9, the schematic of a 2 inputs C-element is depicted, it may be also implemented with as many number of inputs as needed by designers. This component is needed to implement the self-timed register, that is a module which captures the data on its input when following conditions are both true:

- Next register contains an empty state.
- Data is present at the inputs.

When both these conditions are satisfied at the same time, data is captured and propagated through the pipeline. On Table 3.2 the truth table of this element is represented for understandability's sake. Afterwards, I am going to present the *Muller pipeline*, or better a 4-phase dual-rail asynchronous pipeline and the way it works.

3.2.3 Pipeline implementation

In order to develop the pipeline, let us first build a basic structure, which will help us to separate our hardware implementation via a higher level of abstraction. The element I am going to present is a *register*, that is a memory element which is able to store a token and then to propagate it. The particularities of this element with respect to the synchronous version are the absence of the clock signal, and the presence of two more ports which I am going to call *Ack_in* and *Ack_out*.

This small module, hereinafter named *asynchronous register*, embeds a completion detector mechanism which can establish whether the token has been captured and acquired. On Figure 3.10 is showed the schematic of the internal structure of such element.
The role of *Ack_in* is to deactivate the register if first condition mentioned before is not satisfied. Indeed, the next register, where the signal comes from, put the signal on `1` if a token is present. Afterwards the inverter set the signal to a logic `0` which feeds all the C-elements not allowing them to capture the signal. Whereas, if *Ack_in* is set to `0`, all C-elements can capture the input signals. *Ack_out* instead is the signal generated internally, which feeds the *Ack_in* of the previous register in

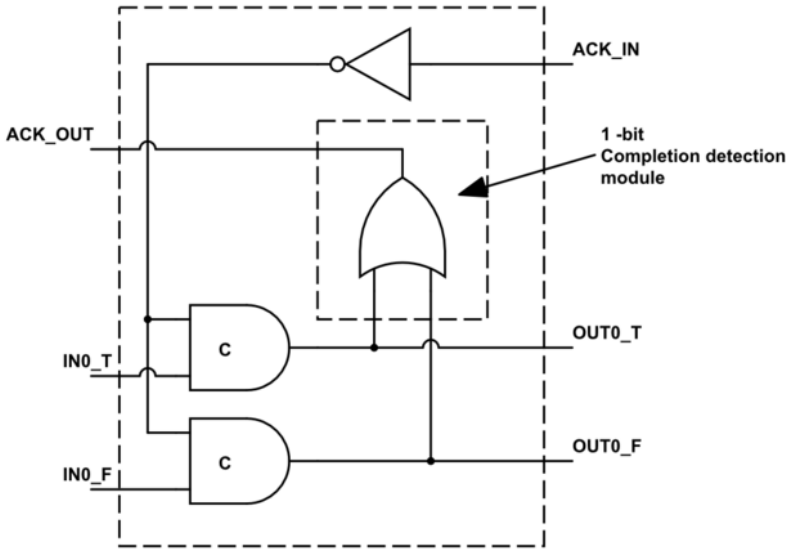


Figure 3.10: 1 bit asycnhronous register

the pipeline; it is set by the completion detection module which simply checks if at least one of the two signals per input coming from external environment and going through the C-elements are at logic `1`. If so, it means that a token is present. It is worth reminding to readers which *IN0_T* represents the logic true side of a signal in dual-rail fashion, while *IN1_F* the logic `0` side.

As depicted on Figure just described, the completion detection module may seem to reader a really trivial component since it is composed by one gate only. Nevertheless, things get more complicated when one has to deal with a register higher than one bit, since the completion detection module must check that the *empty* signal is not present for the whole bus. For clarity's sake, a 3 bits completion detection module is represented on Figure 3.11. As one might observe, the number of inputs of such structure is doubled with respect to number of bits to represent, and each *OR* gate must check whether the signal is present on the bus. Finally, C-element is in charge of checking that all inputs are present. As already mentioned, C-element tailors really well to the asynchronous circuits due to its capability to change the output when all the inputs change their logic state. Reader must pay attention even to the internal structure of the C-element that would change as the number of inputs change. On Figure 3.12 one of the possible implementation I used throughout this research.

On the light of the components presented above, I can finally introduce an asynchronous pipeline as final instance. It is showed on Figure 3.13 and it is composed

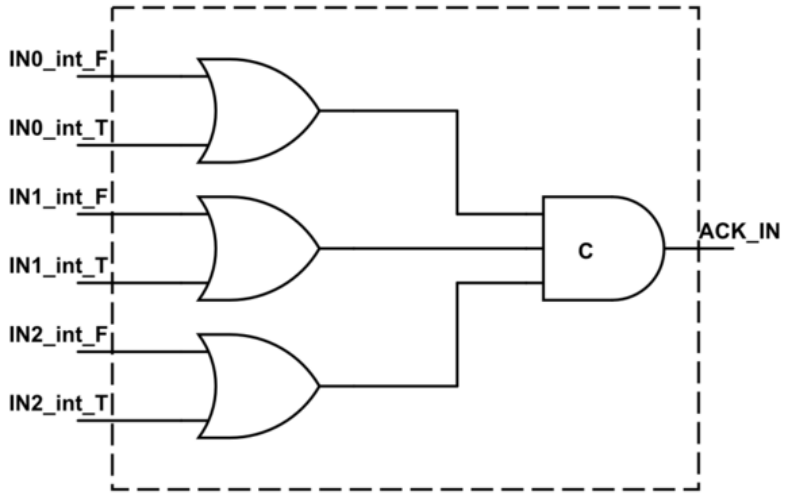


Figure 3.11: 3 bits completion detection module

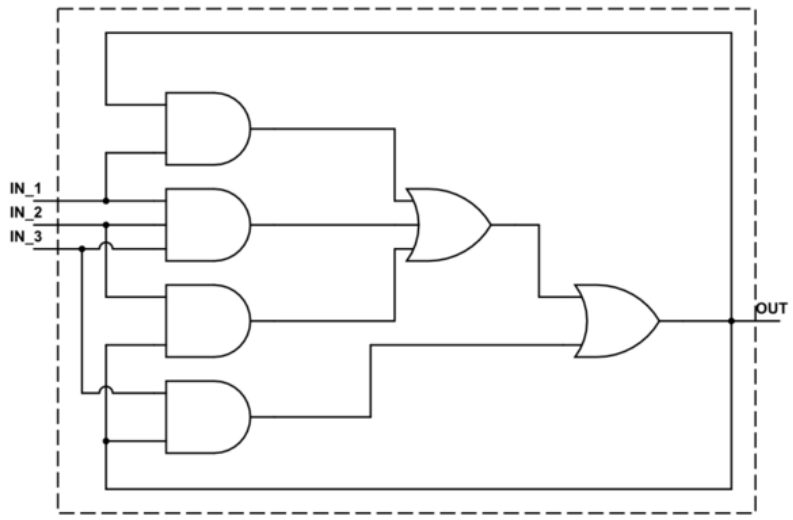


Figure 3.12: 3 bits C-element implementation

by asynchronous registers only without any combinatorial parts internally, they will be covered afterwards on Section 3.3. Since the registers do not share any notion of time, the token here is propagated like a wave, as Jens Sparso described on [33].

As discussed before, due to the protocol used for transactions, an empty state between two consecutive registers is needed in such a way to allow tokens to be propagated, therefore consecutive data tokens cannot be present consecutively inside

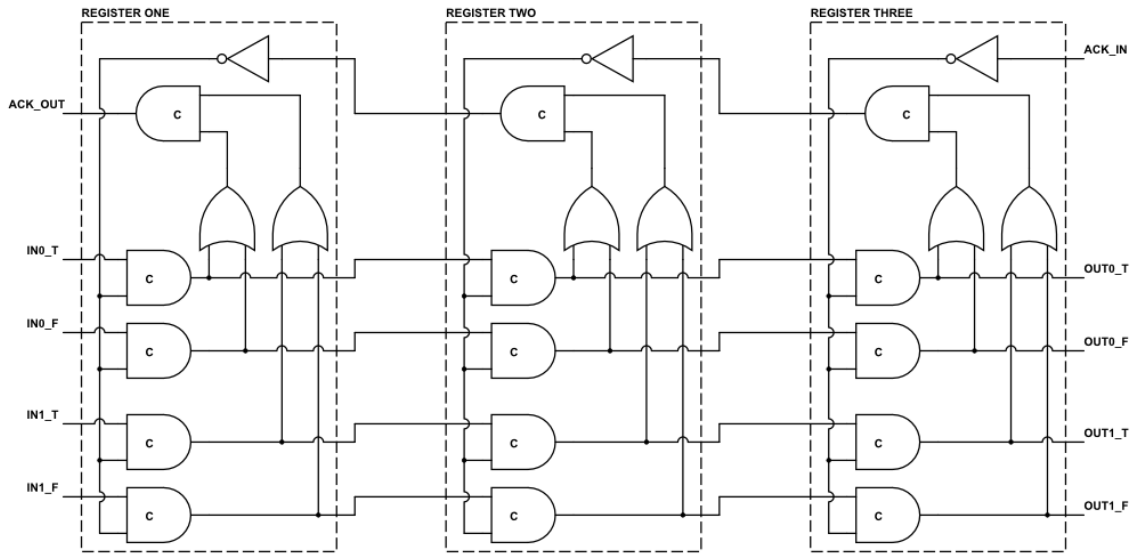


Figure 3.13: 2 bits - 3 stages asynchronous pipeline example.

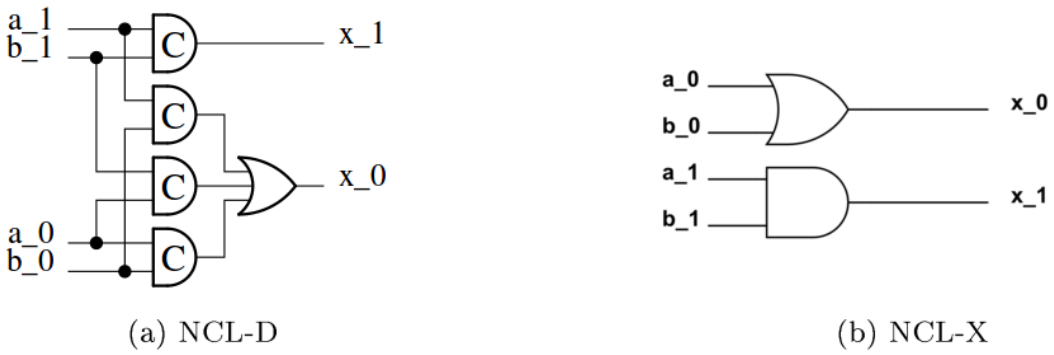
the chain. This is a quite negative problem since one would waste fundamentally half of the possible throughput. This is why it is totally worth pursuing the research toward the 2-phase asynchronous protocol, in order to increase the performance of the device under development.

3.3 Combinatorial circuitry

Now that the whole structure of the memory elements has been introduced, I can focus on presenting the combinatorial structure. It differs with respect to the normal single-rail structure, both because each signal is represented on two wires and because of avoiding hazards. For explaining the former statement, let us concentrate on the protocol which allow a single information to be transmitted on two rails. In order to be able by the designer to implement the same logic functions as the single-rail counterpart, one should use a totally different gate library-based structure, which assumes an information to have two inputs, not one only.

Let us take as instance a basic `AND` gate. It allows the output to be activated when both two inputs are stuck at one. This functionality can be reproduced for dual-rail protocol as on Figure 3.14, and according to [38] there could be two different implementations for such gate.

The one on the left, also called NCL-D implementation, is the most reliable but also the most expensive in terms of area. Indeed as one might observe it is composed by four c-elements and one `OR` gate. The implementation on the right instead is



(a) NCL-D (b) NCL-X

Figure 3.14: Dual-rail AND gate implementations. ([38], Figure 2 modified)

named NCL-X and is the cheapest and less reliable one. For discussing about the reliability of such gates and introducing to readers the reasons why designers may choose one implementation with respect to the other one, it is worth discussing about the the hazards issue.

Hazards are errors that might affect the pipeline due to various reasons. In the synchronous methodology, according to [32] they may be distinguished between *Data*, *Control* and *Structural* hazards, the former ones are related to instructions depending on results of previous instructions, the *Control* typology are instead related to branch instructions while the latter ones to conflict of resources. All the problems may be overcome by stalling the pipeline and introducing bubbles, that even though may slow down the CPU reducing the actual throughput, are needed for a correct behaviour.

In the asynchronous context, hazards may arise when a combinatorial module is not completely transparent. Therefore, according to the definition given by Jens Sparso, it may happen when the outputs of a combinational module arise before the computation is actually finished inside the module.

In order to explain this phenomenon let us refer to a standard `OR` gate used for single-rail transmission. Due to logic function it embeds, the output would arise if at least one of the two inputs is stuck at one. Even though in the synchronous context it is not a problem because registers capture the results of the glue logic at a frequency that assumes the computation to be finished and the results stable, it is for asynchronous combinatorial circuitry since the registers do not share a single notion of time. In the worst case, it means that potentially a wrong result may be propagated or that one of the next computation might be influenced by it.

In order to solve this problem, one needs to develop some structures allowing the output to be present and at the same time checking that the computation is entirely finished. Hence, different solutions are present on the design market. On [39] a

technique for optimising the number of completion detection blocks to be inserted inside the design is formulated, the main idea behind it is to *strategically* set a variable number of completion detection modules checking that signals are present at some circuit point beyond the output and the inputs, in order to be sure that the module has terminated its computation by keeping the area as low as possible. In this research I have not used such theory to reduce the area consumption of the design.

The strategy I have used during my work at *Newcastle University* was mixing reliable (NCL-D) and unreliable (NCL-X) gates in order to build hazards free components. Before analysing this approach, let us first discuss about the reliability of the gates presented on Figure 3.14. In the context of asynchronous design methodology, *reliability* is defined as the capability of displaying the results when all the inputs are present. Considering such definition, the *a* typology showed on Figure contains four C-elements, and this is the key that makes such gate totally reliable. Indeed, both *x_1* or *x_0* cannot be activated if both inputs are not activated. The disadvantage of this component is the big area it takes to be synthesised.

	NCL-D $[nm]$	**NCL-X** $[nm]$
AND	33,712	7,840
OR	33,712	7,840
XOR	36,064	20,384

Table 3.3: NCL-D and NCL-X gates area comparison

The *NCL-X* instead is really good from the point of view of the area, but worse than the NCL-D for what concerns reliability. It is because the output signalling a logic `0` may be activated as soon as one single input representing false is activated. It does not allow designers to use *NCL-X* gates only to implement total reliable components. A comparison between these two kinds of gates is depicted on Table 3.3, where the area was obtained by synthesising the gates via a 90 *nm Faraday* library, in particular the low VDD version.

As reader might notice by the Table, both `AND` and `OR` gates can be synthesised on the same amount of area, this is why in order to implement the logic `OR` under NCL-D fashion one needs just to change the connection of the final *OR* gate. Concerning the `XOR` gate, the two implementations are similar because of the implementation of the logic function requires the usage of more gates even for what regards the NCL-X implementation.

After having discussed about all the parts compose an asynchronous pipeline, I

am going to discuss about the composition of various dataflow structures (Chapters 4) for supporting the reconfigurability.

Chapter 4

Dataflow graphs composition

One of the aim of this research is to automate the composition of several dataflow graphs, represented with *Condition Partial Order Graph* representation. First step to synthesise the controller which will be in charge of managing the reconfigurability of the structure, is to seek an optimal *op-code* for each graph. It is an extremely important task because most of the area and as a consequence the power consumption of the whole control-unit stem by encoding achieved. Even though the best solution would be reachable by inspecting all the solutions only, it is infeasible due to the wide solution space, as described on Section 2.1.1, and the high time needed to synthesise the model into logic gates make the task of seeking the minimum area solution really difficult. Hence, the need of a clever and faster cost function, able to point out directly a good encoding both from the execution time and area perspectives.

Afterwards, I am going to discuss about the various techniques used in order to generate the encoding for the representation, it is the key point for developing a fast and efficient tool to support the composition phase of the model. Indeed, as the Section 4.3 points out, the encoding generation process cannot be executed quickly and excellently at the same time. Some approximation should be done in order to find a good trade-off between speed and quality of solution.

Therefore, over this Chapter I am going to present all the instruments, tools and procedures used for evaluation of the various encodings. The cost function will be also presented and described. On Section 4.3 the *Graph Isomorphism* issue is presented and compared to optimal search task for the purpose of the work. It might help to see this research from a bigger perspective in order to comprehend better final results comparing them with a similar problem already handled by countless researchers. Finally three encoding generation possibilities will be analysed: *Recursive*, *Random* and *Optimal* generation. Last one is the best possible trade-off between speed and time. Nonetheless all the algorithms will be explained, discussed and analysed in order to show the differences between them, the advantages of each

solution as well as the drawbacks.

4.1 Decoded circuit area evaluation

In order to evaluate the area of the circuit cleverly, I used a tool developed by Berkeley University named **ABC**. Among the other things this software is able to do, it can read a library in *Genlib* format, and map the circuit read by file into logic gates, computing the area it would take after synthesis process. A more complete description of this tool is provided on its dedicated web-site, [8].

Included in this Section, all the parameters and settings I used to compute the final area of the circuit. Following part would be structured mainly into three subsections. First one describes the library used to perform a comparison between various circuits, second one the equations needed to develop the final circuit from *CPOG* models, and how to extract them. Finally the settings set into **ABC** tool in order to minimise the area as much as possible, and map the boolean equations into a circuit.

4.1.1 Gate library used

The library I used to map each circuit with real logic gates was a $90nm$ library, in particular the one in [5]. Since *ABC* accepts just library in *Genlib* format, I had to convert it first. *Genlib* format is a fairly simple text arrangement that allows to specify different kinds of parameters for each gate considered. Below the arrangement of the parameters via such format:

$$GATE \quad Name \quad Area[\mu m^2] \quad Function \quad Propagation\_Delay$$

The Propagation delay is represented by four numbers as described by Robert Moniot in [6]: "The first is the maximum delay, the second is the maximum additional delay per fanout. The second pair of numbers are the minimum values of those quantities.". A more accurate description is showed on [7]. Below an example of a simple *AND* gate with two inputs.

$$GATE \quad and2 \quad 5.6448 \quad O = a * b; \quad 1.0 \;\; 0.2 \;\; 1.0 \;\; 0.2$$

For our analysis, just the area parameter will be taken into account, but the delay could also be inspected by means of this library.

4.1.2 ABC tool usage

As reported in [8], "ABC is a growing software system for synthesis and verification of binary sequential logic circuits appearing in synchronous hardware designs.

ABC combines scalable logic optimization based on And-Inverter Graphs (AIGs), optimal-delay DAG-based technology mapping for look-up tables and standard cells, and innovative algorithms for sequential synthesis and verification." More information can be found on corresponding website.

What I want to discuss over this Section is how I used *ABC* to achieve the aim of optimising the entire decoding circuit, and mapping it with a gate library in such a way to compare various solutions in terms of area consumption properly. Before going through the list of commands used in the software, I just want to highlight that there are several ways to reach the goal of optimising circuit in the tool, since plenty of commands and algorithms are present, and they are potentially exploitable to optimise whichever logic circuit targeting different parameters. Now, let us discuss about the particular procedure I used targeting area optimisation. Since plenty of solutions should be analysed and compared, an automatic evaluation of the records is needed. Hence, I used a script to perform all the operations sequentially on the current circuit under analysis.

First operation one has to perform is clearly reading the circuit (`read_eqn file_name` command) under design. *file_name* is the path of a file with a predefined layout as reported below, it is important in order to specify all the specifications of the logic circuit for the tool.

```
INORDER = A B;           % list of the inputs
OUTORDER = AND OR NOT;   % list of the outputs
AND = A * B;             % boolean equations
OR = A + B;
NOT = !A;
```

`INORDER` must be followed by the list of inputs of the circuit, and `OUTORDER` by the outputs. A brief example of three different gates is depicted indeed, where `*` represents logic *and*, `+` logic *or* and `!` the *not* operator.

Afterwards, library must be read to know the exact area of logic gates I am going to deal with. `read_library library_name` is the right command in such case, where I set as library the one I presented on Section 4.1.1.

After selecting those two parameters of the tool, I can focus on optimising circuit targeting area optimisation, obtaining results for each encoding considered. Therefore, three commands were used sequentially: `choice; map; ps`. The tool uses *Standard Cell Mapping*-based algorithms which allow to optimise area as well as delay of the circuits, both sequential or combinational ones.

The commands listed above help optimising area of the final circuit, provided they would be executed several times. As the handbook of this tool states indeed: "Typically it takes more than 10 iterations by circuit to converge and area keeps

improving. This is achieved by the ability of *AIG*[1] rewriting to find good circuit structures, by the ability of choices to capture structural flexibilities, and by the mapper to do a good area recovery."[8].

Finally `print_gates` command returns the area parameter of the circuit, it lists all the percentages of gates used belonging to library chosen, and the final result where all the logic gates are included.

4.2 Cost function

As mentioned before, at the beginning of this research small *CPOG* models were used (as discussed in Section 2.1.2). It allowed me to inspect the overall solution space by hand, without the need of automated algorithms, trying to find out a cost function useful to see the problem from another perspective. Over this Section, cost function will be presented, afterwards an analysis of the *CPOG* showed on Section 2.1.2 will be performed, and finally the results will be experienced on a bigger representation (the one on Figure 2.7), so that to demonstrate the usefulness of the function achieved.

4.2.1 Correlation between Partial Orders and op-codes

Before comparing different encodings, I want to introduce the cost function I used in order to find a heuristic able to minimise area the circuit would take to be synthesised on. Inside every *Conditional Partial Order Graph* representation, various number of single graphs are included, one different to each other. As I am going to demonstrate, one can count how much a graph is different from another one, and set this result inside a matrix (defined as *Difference Matrix*).

In particular, it is defined as a strict upper triangular matrix $[\mathcal{N} \times \mathcal{N}]$, where $\mathcal{N}$ is the number of graphs in a model, with all the entries on the main diagonal fixed to 0. Every row r is associated to a *CPOG*, as well as every column c. Each entry represents how much *CPOG* r, is different from *CPOG* c. An example is depicted below.

$$\mathcal{DM} = \begin{pmatrix} 0 & 2 & 1 & 4 \\ 0 & 0 & 1 & 2 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 \end{pmatrix}$$

[1] And-Inverter Graphs: is a directed, acyclic graph that represents a structural implementation of the logical functionality of a circuit or network.

Key point to find the cost function is to encode the graphs in such a way that: the more two graphs are similar, the more should be encoded with two similar op-codes, that is, with a minimum $\mathcal{H}$amming $\mathcal{D}$istance[2].

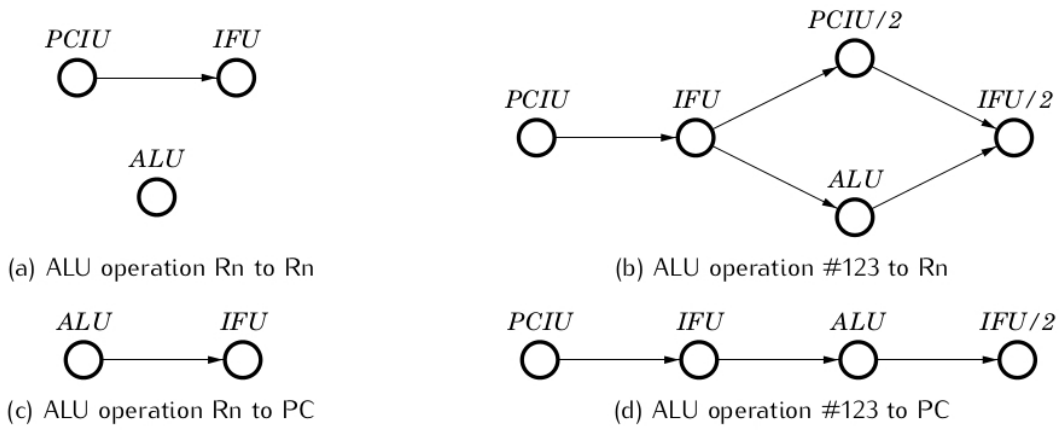
On the light of above, cost function I used to find minimum area is on Formula 4.1. $\mathcal{DM}_{i,j}$ represents entry of the matrix $\mathcal{DM}$ with $row = i$ and $column = j$, while $\mathcal{HD}_{i,j}$ represents $\mathcal{H}$amming $\mathcal{D}$istance between op-codes used to encode i and j *CPOGs*.

$$\mathcal{F} = \sum_{i,j \in \mathcal{N}}^{i \neq j} (\mathcal{DM}_{ij} - \mathcal{HD}_{ij})^2 \tag{4.1}$$

Minimising $\mathcal{F}$ means encoding the *Partial Orders* with more differences with a couple of op-codes with higher $\mathcal{HD}$, and the ones with less differences with op-codes with smaller $\mathcal{HD}$. On the next section, I am going to analyse the area consumption of each encoding with respect to $\mathcal{F}$ function.

4.2.2 Cost function results and statistics

Let us now compare different solutions for *Conditional Partial Order Graph* on Figures 2.9 and 4.1. All graphs on Figures before were analysed, but for readability's sake, the ones just mentioned will be showed only.



(a) ALU operation Rn to Rn

(b) ALU operation #123 to Rn

(c) ALU operation Rn to PC

(d) ALU operation #123 to PC

Figure 4.1: *CPOG* with four instruction classes analysed.

[2]In information theory, the Hamming distance between two strings of equal length is the number of positions at which the corresponding symbols are different.

On the Table 4.1 the correlation present between the encodings set for the models is depicted, the area of circuits synthesised and cost function I introduced on Section 4.2.1.

Encodings	**Area** $[\mu m^2]$		**Cost Function** $\mathcal{F}$	
	CPOG Fig.2.9	*CPOG* Fig.4.1	*CPOG* Fig.2.9	*CPOG* Fig.4.1
`00 01 10 11`	146,630	93,010	28	11
`00 01 11 10`	141,030	117,130	28	15
`00 10 01 11`	146,630	93,010	28	11
`00 10 11 01`	141,030	117,130	28	15
`00 11 01 10`	154,530	98,780	36	11
`00 11 10 01`	154,530	98,780	36	11

Table 4.1: Comparison among different solutions of two *CPOGs*.

First observation one might perform on Table 4.1 is that, as expected, area changes as encoding changes. In such a small models differences in terms of area is quite limited, but probably area swing should get wider as soon as representation gets bigger. Additionally, as one could notice: where $\mathcal{F}$ is higher, the area becomes bigger as well. Results are clearer if we plot the area function with respect to $\mathcal{F}$, as depicted on Figure 4.2, where the function plotted is an interpolation of second degree both for two graphs analysed.

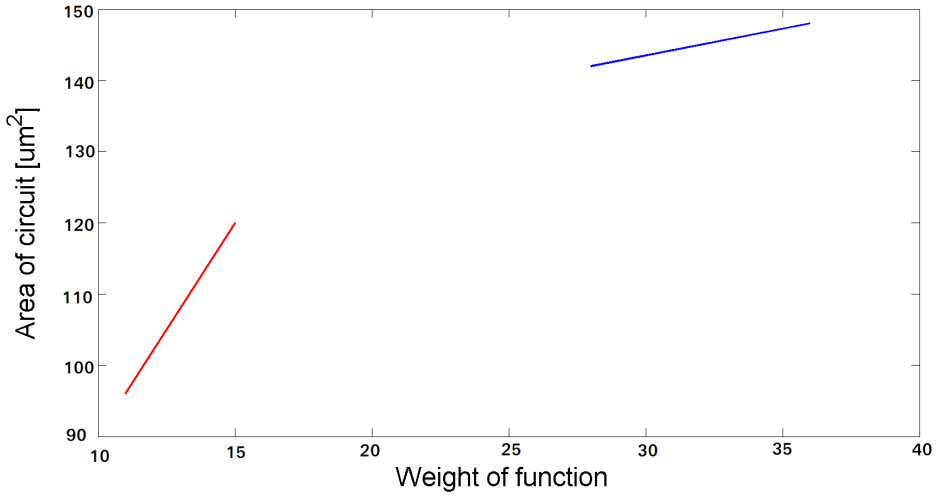


Figure 4.2: Area-$\mathcal{F}$ plot of *CPOG* on Figures 2.9 (Blue) and 4.1 (Red)

In particular regarding *CPOG* on Fig. 4.1, one can observe the strong correlation between these two parameters. Let us combine now the two representations just analysed in order to figure out whether this correlation states even for bigger models. As considered before indeed, a peculiarity of this representation consists on combining two or more systems without affecting the behaviour of each class of graphs. It is an easy and quick way to combine separate behaviours of a system.

The *CPOG* obtained is depicted on Figure 2.7. It contains 8 separate classes of events, and by exploiting Formula 2.3 I estimated potentially 5040 solutions that we should analyse. On Figure 4.3 each different solution is plotted, as done for graphs before, while on Figure 4.4 an interpolation of third degree is showed, just to have a clearer view of the correlation present between Area and $\mathcal{F}$.

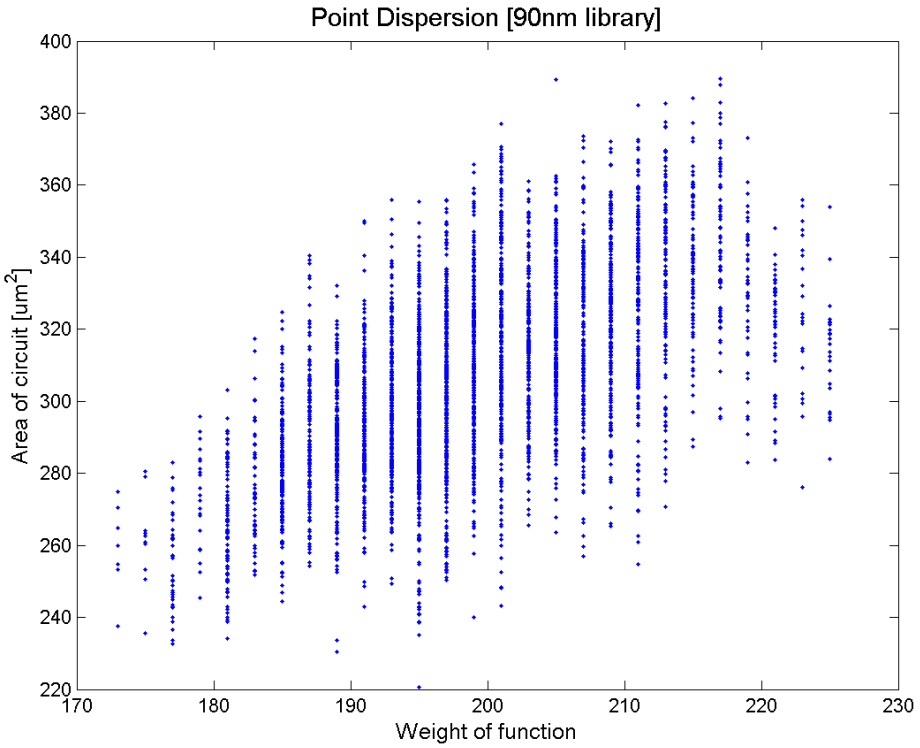


Figure 4.3: Point dispersion of *CPOG* on Figure 2.7.

As might be observed from Figure 4.4, the correlation holds for the two parameters, indeed the more $\mathcal{F}$ the more the area of the circuit is reduced. Nonetheless, it is worth mentioning that, as depicted on Figure 4.3 there is a certain margin of error. In fact, although the area keeps reducing as $\mathcal{F}$ gets lower as average, it is not verified for every points of the plot, but it is a statistical measure only.

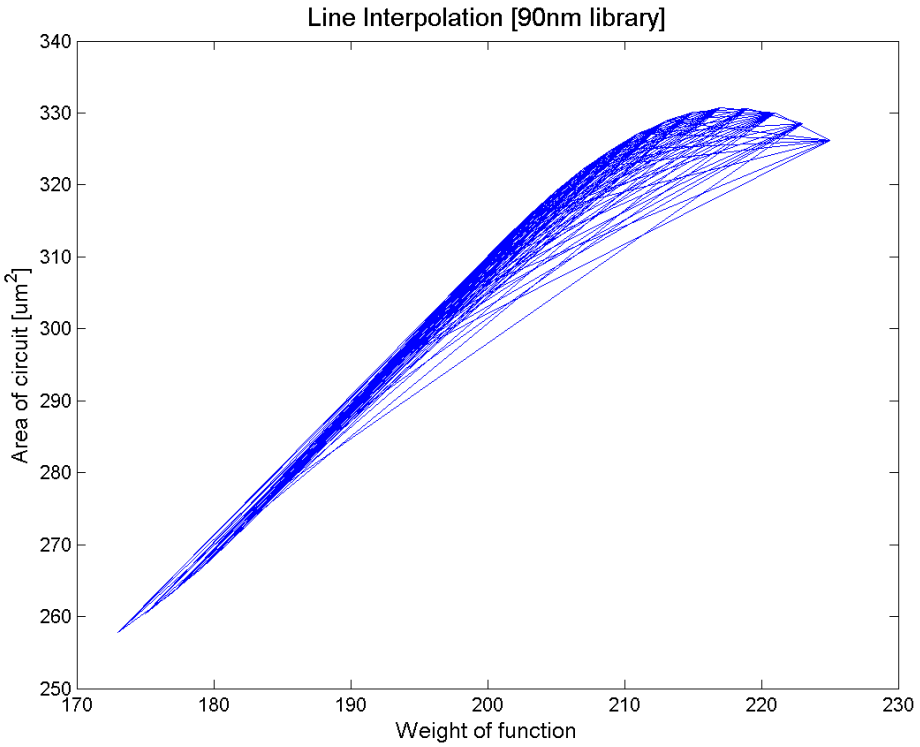


Figure 4.4: Line interpolation of *CPOG* on Figure 2.7.

By analysing better the statistics indeed, it is possible to find out that minimum area point is in correspondence of $\mathcal{F} = 195$, and it is 220,63 [μm^2]. Additionally, if one observes the x-axis on the minimum $\mathcal{F}$ point, that is 173. He might observe the presence of various solutions returning the same cost function but a different area, spacing from 237,52 to 274,79 [μm^2].

Latter consideration should make the reader understand that this research belongs to heuristic class of algorithms. Therefore, defining an error parameter might help the analysis.

$$\mathcal{E}_{avg} = (\frac{\mathcal{A}_{avg}}{\mathcal{A}_{min}} - 1) \times 100, \quad \mathcal{A}_{avg} \subseteq \mathcal{F}_{min} \tag{4.2}$$

It is showed on Formula 4.2 and is the average error of the cost function. it represents the error one might get neglecting the entire solution space. And as a consequence the percentage of area one might waste if one would have not considered the area points with higher $\mathcal{F}$. $\mathcal{A}_{avg}$ indeed is computed by considering the points with minimum weight of cost function.

On the light of above, it is better to look for solutions where $\mathcal{F}$ is low, without neglecting the encodings around the minimum cost function point, in order to improve the quality of the solution. On the other hand, it is a waste of time looking for solutions where the $\mathcal{F}$ is very high, because it is very unlikely to have a good encoding in terms of area centred on high values of $\mathcal{F}$. Formula 4.2 applied to *CPOG* on Figure 2.7, returns a value of:

$$\mathcal{E}_{avg} = (\frac{258{,}72}{220{,}63} - 1) \times 100 = 17{,}26\%$$

It represents the percentage of area one might waste whether a band of $\mathcal{F}$ values is not considered. It will be useful for comparing various kind of algorithm versions.

4.3 Graph Isomorphism Similarity

Nowadays, graphs are used to model and solve different kind of problems. In the last few years, plenty of research were conducted on different possibilities through which graphs could be involved to, likewise pattern recognition problems, or for describing images splitting them in different parts, a couple of examples could be found on [14], [15].

One of the main concern involved on algorithms exploiting graphs is the matching graphs issue. It is likely to compare graphs or sub-graphs in order to establish whether two or more graphs match to solve problems described before, and as discussed in [16], there could be two different types of matching sought: exact or inexact matching. In the former one, "a strict correspondence between the two graphs is sought"([16], Introduction section), while in the latter the purpose of the search is finding similarities between graphs, achieved by applying some editing operation to them.

As described by Luigi P.Cordella et al. in their research just cited, it is not yet known if the graph matching problems is solvable in NP-computational time or in polynomial time. It is a problem if one has to deal with very big graphs composed by several nodes, and although several algorithms have been developed in order to reduce execution time and memory usage of such issue ([17]), exploiting different representation likewise trees or planar graphs, it is still difficult to handle this representation.

In this Section, I aim at showing to reader, the similarities between the encoding sought problem, with graph isomorphism issue. In such a way to demonstrate that it is not possible yet to solve issue of this research with *Conditional Partial Order Graph* both in an optimal and exact way at the same time.

4.3.1 Graph isomorphism recognition

Let us start the discussion by presenting the notion of *isomorphism* between two graphs.

Definition. If $\mathcal{A}$ and $\mathcal{B}$ are two graphs of the same type (unordered, weighted, directional, ecc.), isomorphism between these graphs is the function bijection between the vertex set of $\mathcal{A}$ and $\mathcal{B}$, as depicted on Formula 4.3.

$$\mathcal{F}_B : \mathcal{V}(\mathcal{A}) \rightarrow \mathcal{V}(\mathcal{B}) \tag{4.3}$$

In such a way that if two vertices belonging to $\mathcal{A}$ are adjacent, they must be adjacent in $\mathcal{B}$ as well. If an *isomorphism* exists between $\mathcal{A}$ and $\mathcal{B}$, they are called isomorphic and it is possible to state $\mathcal{A} \simeq \mathcal{B}$.

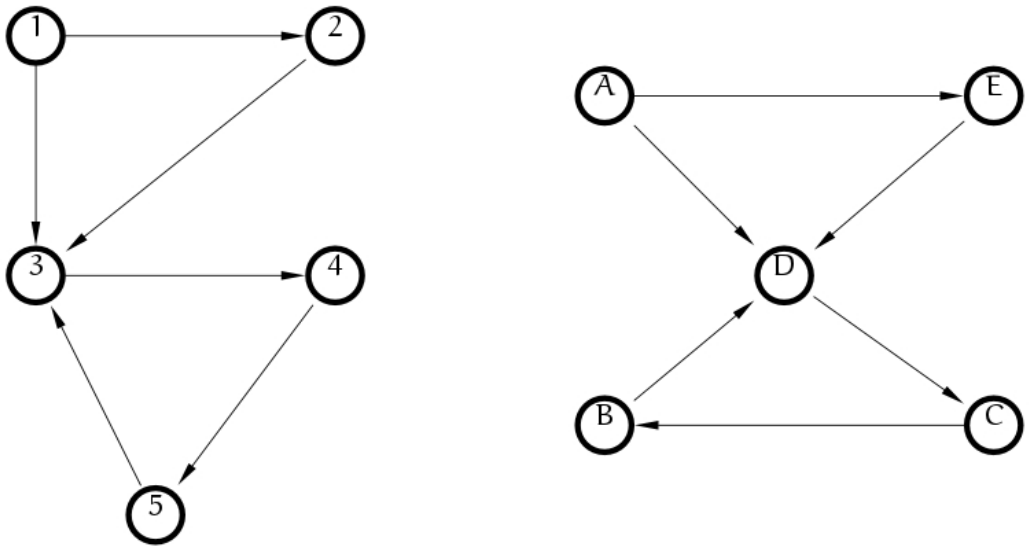


Figure 4.5: Example of two isomorphic graphs.

On Figure 4.5 is depicted an instance of two isomorphic graphs. Even though the graphs might seem different, either because they are labelled with different names, and because the vertices are placed in a different order, they represent the same system or behaviour. In fact, a possible bijection among vertices of two graphs could be the one below:

$$\mathcal{F}_B(A) = 1, \quad \mathcal{F}_B(B) = 5, \quad \mathcal{F}_B(C) = 4, \quad \mathcal{F}_B(D) = 3, \quad \mathcal{F}_B(E) = 2$$

Over the last few decades, algorithms for speeding up the matching graphs process have been thoroughly studied, among which it is worth mentioning the one developed in [18], concerned exact graphs matching. As well as the one on [19] which is based on transforming the graph in a canonical form, before starting the matching process. As Cordella et al. explained in their research, although the latter algorithm is one of the fastest available, "it has been shown that there are categories of graphs for which it employs exponential time."([16], Introduction). To complete my analysis, there is even a non deterministic technique [20], where even though a solution concerns this problem might be found in a polynomial time, it might not be optimal.

4.3.2 Similarities between graph isomorphism and encoding process

The process of seeking an optimal encoding for *Conditional Partial Order Graph* model is mainly based on Formula 4.1. Here, the basic idea, as explained on Section 4.2.1, is to assign op-codes to each pair of graphs in such a way to assign the codes with minimum number of differences (less $\mathcal{HD}$) to the most similar graphs.

On the light of above, one could think the problem of CPOG-encoding, like the problem of seeking an adequate $\mathcal{F}_B$ between two graphs; where the first graph represents the *Conditional Partial Order Graph* model:

- $\mathcal{V}$ represents the set of the instruction classes;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ the differences that each class has with another one.

And the second graph represent the set of op-codes, where:

- $\mathcal{V}$ represents the set of our codes one could potentially assign to an instruction class;
- $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ models the $\mathcal{HD}$ between two different op-codes.

In order to clarify my statement, an instance is represented on Figure 4.6, where on the right side of it is depicted the graph of op-codes, where as mentioned before each labelled circle ◯ represent a possible op-code. In this case, since the number of *Partial Order Graphs* is 4, we need 2 two bits to codify all the possible scenarios. While each undirected arch represents the $\mathcal{HD}$ among vertices.

On the left side of the Image instead, $\mathcal{DM}$ is depicted. As one could notice, the problem may become an instance of sub-graph matching isomorphism, since the number of vertices that may compose the op-codes graphs could be higher than number of vertices of *Difference Matrix* graph. For instance, in a model that contains five different instruction classes, one should need at least 3 bits for encoding all of them, nonetheless three vertices would remain spare.

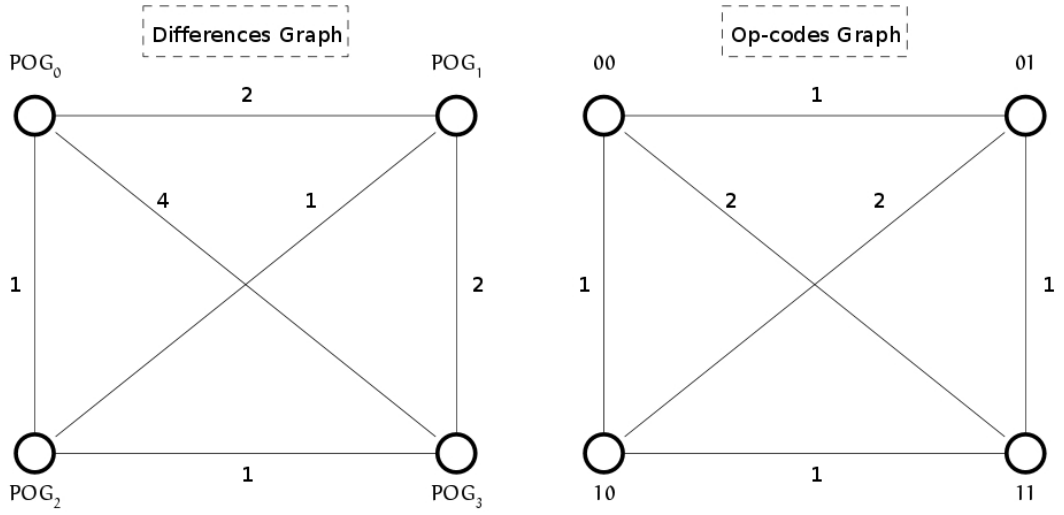


Figure 4.6: On the left side an instance of Difference Graph, on the right side an Op-codes graph is depicted.

4.3.3 Conclusions

The aim of this Section was showing to reader the similarities that this research has with a very well know problem belonging to graph theory field. As several researchers have shown along past years through a high number of analysis and works (Section 4.3), although plenty of algorithms were developed to speed up the graphs isomorphism matching process, it still remains a very difficult task to find an optimal solution in a very short amount of time.

Hence, as demonstrated on the Section before, due to several similarities the encoding process presented on this research shows up with graph isomorphism one, I can state that it is not possible yet to find an optimal solution in a short amount of time.

Overt the next Section, I am going to discuss this problem with more details, pinpointing on the issue of encoding generation with a particular attention to the time of the generation as well as the goodness of the solutions.

4.4 Encoding Generation

As mentioned in the introduction of this Chapter, this Section deals with all the various techniques one might use to generate encodings. Before going on discussing about the various techniques and software tools I developed for supporting the encoding generation problem, it is worth mentioning the first attempt to fill in

this gap on the article [53]. The most suitable technique depends on the structure under development, and the advantages and the drawbacks of each method will be presented and discussed over this Section.

4.4.1 Recursive encoding generation

First method I used to encode the *Conditional Partial Order Graph* model is to try every possible combination of op-codes by permuting each element inside the op-code ensemble. It is the simplest and the most straightforward way to inspect the overall solution space and to analyse which is the best area point over all the set of solutions.

First step one needs to perform is seeking the number of possible encodings available for the representation. It is achieved by means of Formula 2.3 which returns $\#\{\mathcal{S}\}'$, the number of encodings inside the solutions space fixing first element to `0...0`. Next step is recurring on the op-codes available, as I have implemented on Listing below:

Recursive generation algorithm.

```
void permutation(int *solution, // Current permutation
      int k, // Op-codes index
      int *opcode_chosen, // Support data structure
      int POG_number, // Number of Partial Order
      int opcodes_number){ // Number of op-codes
  long int i;
  // Limit on number of permutations to find
  if(index >= permutations_number)
    return;

  // Set solution when completed
  if(k == POG_number-1){
    for(i = 0; i < POG_number; i++)
      permutations[counter][i] = solution[i];
    index++;
  }
  else{
    // Recursive code
    for(i = 0; i < opcodes_number; i++)
      if(!enc[i]){
        solution[k+1] = i;
        opcode_chosen[i] = 1;
        permutation(solution,
              k+1,
```

```
                opcode_chosen,
                POG_number,
                opcodes_number);
        opcode_chosen[i] = 0;
    }
}
```

As one may already notice, this method is very good for small models, as the ones I am going to take into consideration in this analysis, but as the number of *Partial Orders* gets higher, the number of solutions increases in an exponential fashion, making the inspection of all the encodings not possible any more due to time constraint.

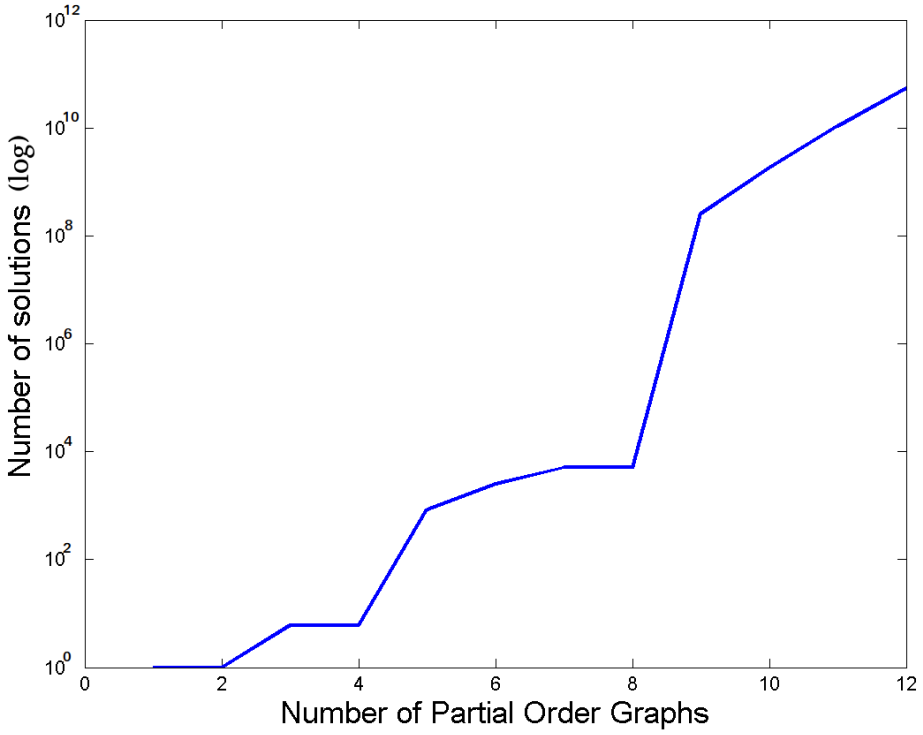


Figure 4.7: Number of solutions increments into *Recursive Generation* technique.

On the Figure 4.7 such correlation between number of *POG*s to encode and number of possible solutions is depicted. This plot is obtained from Formula 2.3, varying the number of graphs k and fixing the size of the op-code ensemble m, that is the number of op-codes available for encoding all the graphs assuming a code on minimum number of bits; while on y-axis the $\#\{\mathcal{S}\}'$ is depicted in logarithmic scale.

As one might notice the function does not grow up always, in fact when the number of *Partial orders* is 3 and 4 the number of solutions remains the same, as well as when it is 7 and 8. This is the consequence of having a slightly higher size of the op-codes ensemble with respect to number of *PO*s.

For sake of understandability let us try to list all the possibilities one has to encode 3 or 4 graphs assuming minimum number of bits op-codes. Applying Formula 2.1 it is possible to find out that 2 bits are needed to encode both 3 and 4 different systems, hence while m is fixed to 2 both in the two cases, k represents number of the graphs of *CPOG* model. Inserting these parameters inside Formula 2.3, number of solutions comes up with both the two cases is fixed to 6. In fact, by listing all the solutions (below) one may recognise the reason why the number of solution remains stable, $\#\{\mathcal{S}\}'$ in fact is always equal to 6.

$$PO = 3 \ : \ \{(00,01,10), (00,10,01), (00,01,11), (00,11,01), (00,10,11), (00,11,10)\}$$

$$PO = 4 \ : \ \{(00,01,10,11), (00,10,01,11), (00,01,11,10), (00,11,01,10),$$
$$(00,10,11,01), (00,11,10,01)\}$$

As soon as number of graphs becomes higher than 8, $\#\{\mathcal{S}\}'$ becomes too high to be inspected completely, making the *recursive approach* technique not very useful to designers. Nevertheless, it is really good for small graphs and allows to find the global minimum area point in a reasonable amount of time. On Figure 4.8, this technique is applied to the model analysed before.

Recursive generation should make reader understand the size of the issue. This technique can be used for models smaller than 9 *Partial Order Graphs* only, where the number of them is higher another algorithm should be used to generate encodings since it cannot be exploited any more. Indeed, by using *Recursive* generation on a big models one will be able to inspect only a small part of the overall solution space due to the huge time the heuristic would take for inspecting the overall solution space. Hence, it leads the need of an another technique aims at targeting few good encoding solutions.

4.4.2 Random encoding generation

Another simple generation technique I followed during this research was generating the encodings in a *Random* trend. Though this algorithm might seem useless, it guarantees generation of encodings uniformly overall the space of solutions, being a better choice with respect to technique discussed on Section 4.4.1, when analysing the entire solution space is not possible. It might be applied to whichever *Conditional Partial Order Graph* representation, no matter how many *POGs* it is composed by.

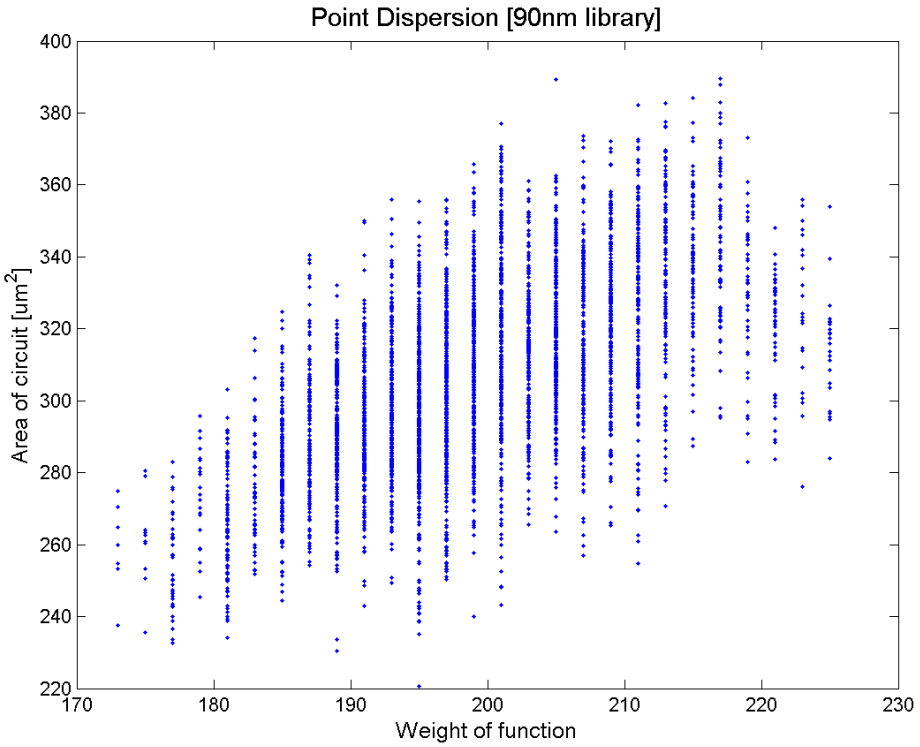


Figure 4.8: Recursive encoding generation applied to *CPOG* on Figure 2.7.

Even in this technique the the first graph is associated to `0...0` op-code, in order not to waste any useless encodings, and for the actual generation the algorithm on listing below has been used:

Random generation algorithm.

```
/* Insert all the op-codes available for encoding
 */
for(j=1; j < total_opcodes; j++)
  opcodes.push_back(j);

while( index < permutations_number)
{
  // Shuffling them in order to get random positions
  std::random_shuffle (opcodes.begin(), opcodes.end());

  // Fix first element of each permutation to op-code 0
```

```
  perm[index][0] = 0;

  // Couple each POG to a random op−code
  for (j=1; j< pog_number; j++)
    permutations[index][j] = opcodes[j−1];

  /* Keep generating till reaching number of permutations
     pre−set by user */
  index++;
}
```

Random-shuffle function refers to `algorithm` library [21], and guarantees the result of a shuffled vector in linear complexity O(n). As one could notice, first element of each permutation should be fixed at 0, which is the first absolute op-code available, the reason was already described in Section 2.1.1.

One might want to check, every time a new solution is produced by the algorithm, whether it was already present inside the encodings already produced or not. Nonetheless, due to the high size of the solution space for *CPOG* composed by at least 9 graphs, it is probably not worth performing this operation since it is very unlikely to pick two equal encodings in the whole solution space. Since doing this check is worth it only for very small model, I chose not to implement this feature.

On Figure 4.9 a plot of 100 permutations applied to representation of Figure 2.7 is depicted. As one might notice, solutions distribute all over the space uniformly. That is extremely good, because beside being able to seek totally uncorrelated solutions, the other algorithms may be compared with a very uniform technique such as the one just presented.

4.4.3 Optimal encoding generation

Finally, another algorithm was developed based on the cost function represented on Formula 4.1. Here, the encodings are generated by trying to minimise the function at run-time in order to point out the exact purpose of this heuristic, that is the cost function minimisation. Due to complexity and length of the code, it will not be reported on the dissertation as the ones before. Nonetheless the main idea will be described.

Since the main purpose of the cost function is minimising the product between the $\mathcal{H}$amming $\mathcal{D}$istance of a couple of op-codes times the difference of two *Partial Order Grahs*. Generation looks for the most similar couple of graphs inside the $\mathcal{DM}$ matrix, storing all of them into an array. Next, the couple of graphs to encode will be chosen randomly, and the op-codes minimising current cost function will be assigned by selecting among the ones still available. Even on the latter case, if

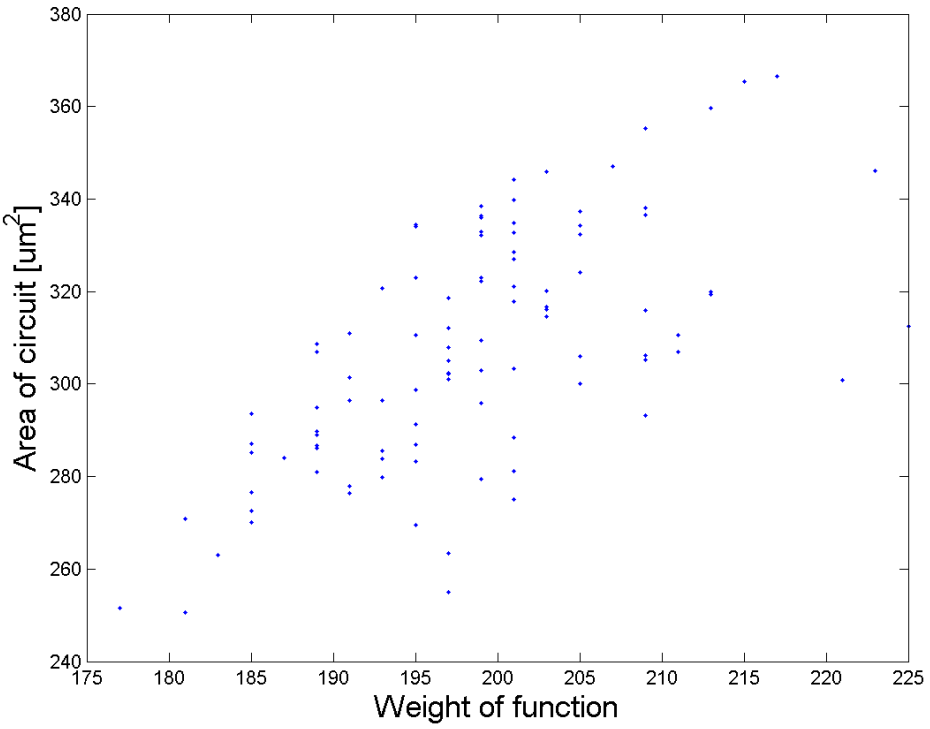


Figure 4.9: Random encoding generation applied to *CPOG* on Figure 2.7.

different op-codes return the same result, they will be chosen randomly.

As one could notice, the rationale is half random. It would limit the capabilities of the encoding generation in term of cost function, but it would reduce a lot the time that an optimal algorithm should take in order to iterate on all the possibilities.

On Figure 4.10 100 permutations generated with the just described technique are plotted. As one could observe on the Figure, points are concentrated a bit more where $\mathcal{F}$ is lower, cutting off points where the cost function is higher. But it is still not enough since plenty of solutions are concentrated in the middle of function. It is important to try as much as possible to target few and precise solutions, in particular when the representation gets higher.

4.4.4 Tuning encodings by means of simulated annealing

Simulated annealing optimisation technique (hereinafter named also *SA*), was introduced by Kirkpatrick et al. in 1983. As they stated in their work: "The subject of combinatorial optimization [23] consists of a set of problems that are

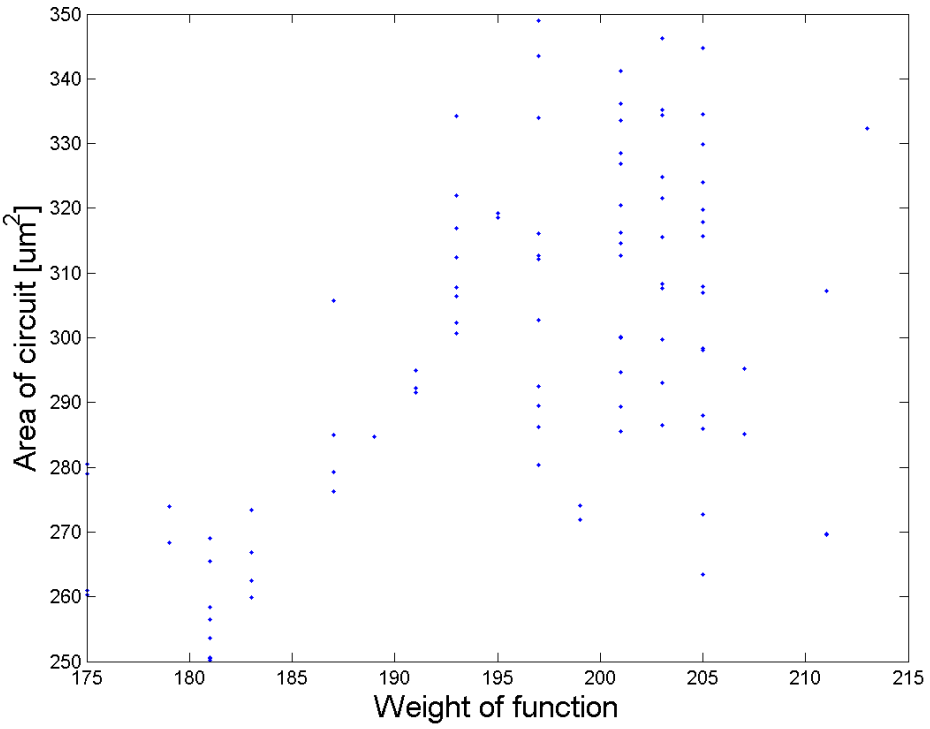


Figure 4.10: Optimal encoding generation applied to *CPOG* on Figure 2.7.

central to the disciplines of computer science and engineering. Research in this area aims at developing efficient techniques for finding minimum or maximum values of a function of very many independent variables [24]."([22]).

Commonly, it is named *cost function*, and in our case is the one on Formula 4.1. Several research were conducted exploiting such optimisation method, either in computer engineering field or in other subjects (i.e. [25], [26]).

Introduction to SA

As described in [22], *Simulated annealing* allows to locate global minimum/maximum of a whichever *cost function* allowing to escape from a local minimum by accepting temporary solutions worse than previous ones.

Concerning CPOG-encoding optimisation issue, an additional step is added to encoding generation process. That is, tuning the current solution swapping the elements belonging to it for reaching another encoding with a reduced $\mathcal{F}$ weight.

The pseudo-code of the algorithm is listed below:

Simulated annealing tuned algorithm.

```
likelihood; // Probability to accept next solution, if worse
alpha = 0.996; // Cooling factor
temperature = 10.0; // Starting temperature
epsilon = 1; // Ending temperature
delta; // Difference among solutions

solution = compute_solution();
weight = compute_weight(solution);
while(temperature > epsilon){

  // Compute next solution and weight
  next_solution = compute_next_solution();
  next_weight = compute_weight(next_solution);

  // Comparing cost functions
  delta = nextweight −weight;

  if(delta < 0){
    // Keep next solution
    solution = next_solution;
    weight = next_weight;
  }else{
    likelihood = (rand() ∗ 1.00) / RAND_MAX;

    /∗ If worse, keep solution with a likelihood
       always lower ∗/
    if(likelihood < exp(−delta/temperature)){
      // Keep next solution
      solution = next_solution;
      weight = next_weight;
    }
  }
  // Cooling temperature
  temperature ∗=alpha;
}
```

As one may notice from the pseudo-code, the parameters at the beginning could be tuned by the designer. The *temperature* represents the starting point of the algorithm, the bigger it is the more the number of iterations will be done inside the while-loop. Same thing regarding *epsilon* and *alpha* parameters, the former one for

example should not be too low in order not to waste iterations looping on the same solution.

The key point of the algorithm is the if statement contains condition below:

$$likelihood < e^{-\frac{delta}{temperature}}$$

Indeed, the more the *temperature* will be high, the more the algorithm would accept worse solutions since the entire exponential factor would be higher. But as soon as *temperature* cools down, right most parameter becomes always lower, making very unlikely the acceptance of a worse solution. Notice that both *likelihood* and the exponential are included between 0 and 1.

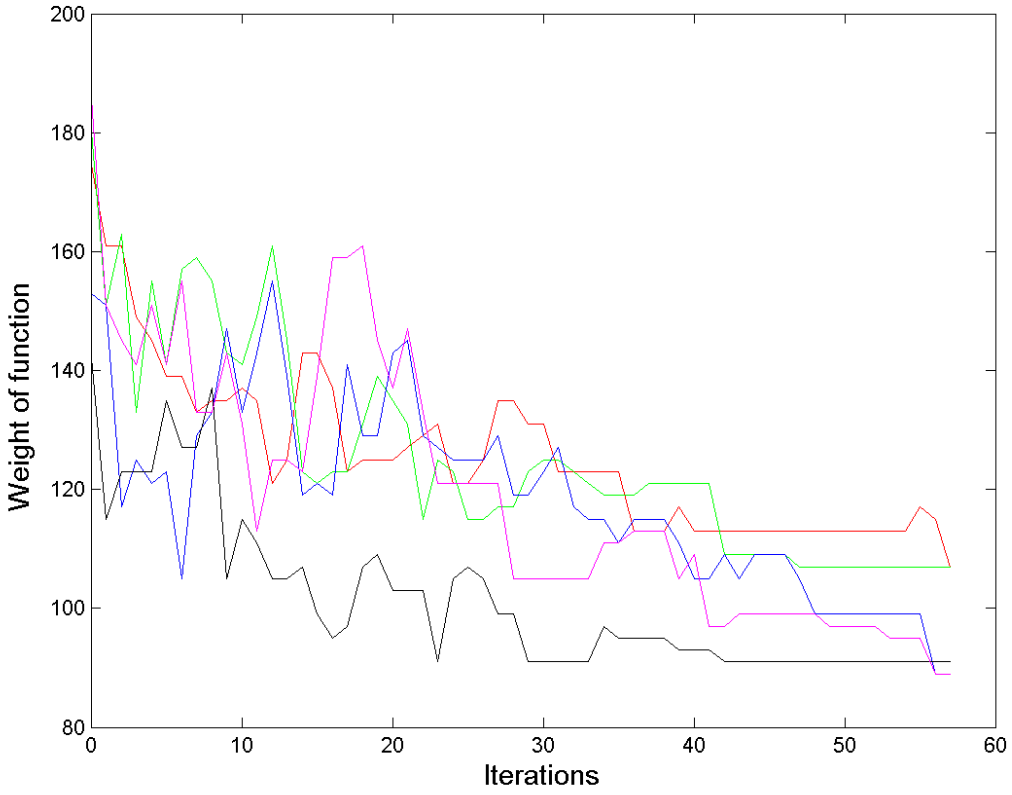


Figure 4.11: Function fluctuations using *Simulated annealing* optimisation technique on Arm Cortex M0+ model.

On Figure 4.11 the results of this technique, applied to five different encodings regarding `Arm Cortex M0+` Instruction Set Architecture, are depicted; the whole representation of the *ISA* will be discussed on Chapter 6. Each color inside the plot represents a different solution, which iteration by iteration gets lower in terms of

cost function. The increments represent the capability of such algorithm to accept worse solutions, in order to escape from a local minimum, achieving a global one.

Simulated annealing applied to encoding generation

SA (Simulated Annealing) therefore is able to minimise the cost function of a solution, starting from a whichever encoding. Consistently with the encoding generation paradigm, it was inserted to optimise encodings obtained with *Optimal* encoding generation (described on Section 4.4.3), in such a way to improve the cost of each solution, from a sub-optimal one.

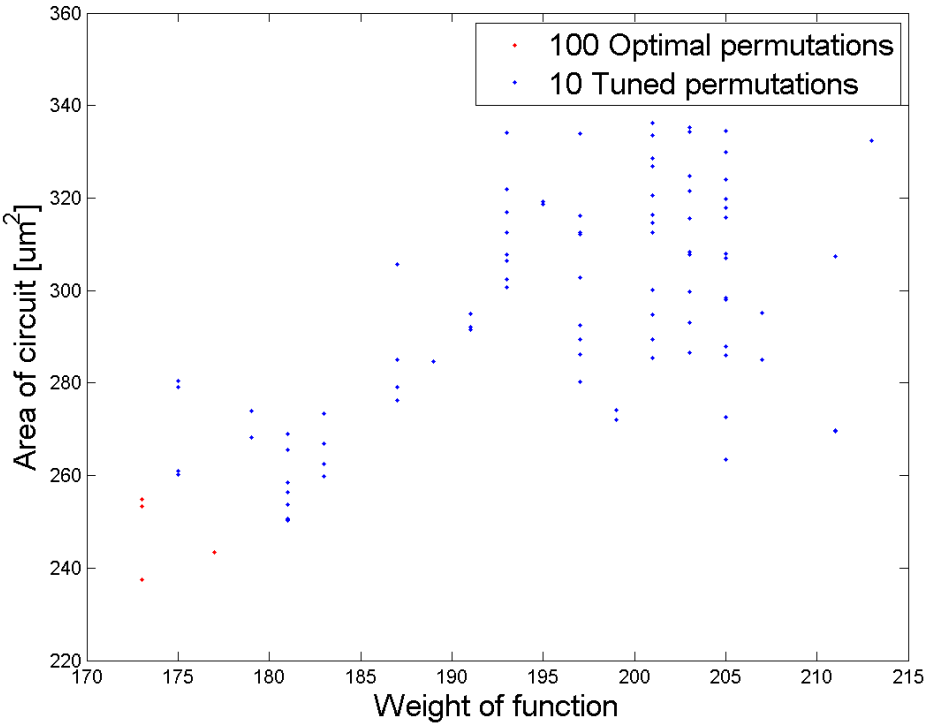


Figure 4.12: Optimised encodings, applied to *CPOG* in Figure 2.7.

The results could be observed on Figure 4.12, where the red solutions in the plot were obtained starting from the blue ones. As one may notice, all the points are concentrated on the left-side of the graph, where cost function values are extremely lower. It is good due to the strict correlation between function $\mathcal{F}$ and area of the circuit. It means that, the tool would be able to focus on few solutions, disposed where it is more likely to find encodings with a reduced area.

One might even want to compare the results of all the encoding generation algorithms described so far. It could be done by plotting the probability density function of the solutions generated through each method, exploiting the Gaussian function. The Gaussian's mean represents the spot, in terms of cost function, where it is likely to encodings to be produced by each algorithm.

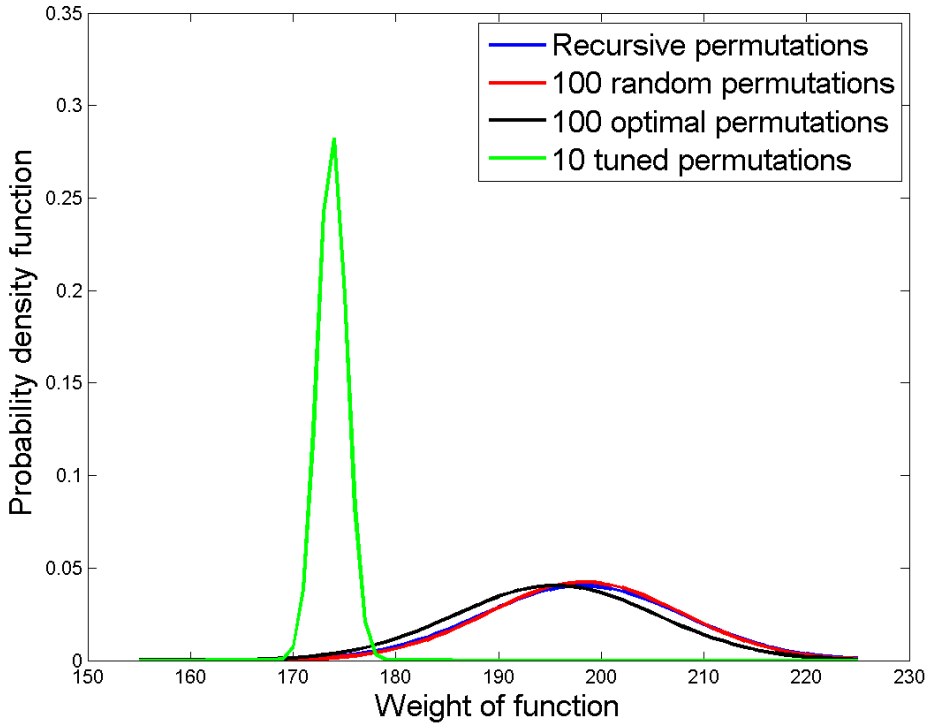


Figure 4.13: Generation encoding algorithms comparison.

On Figure 4.13 all the probability density functions regarding the techniques analysed so far are plotted. As one might notice the last method, exploiting *Simulated Annealing* optimisation technique, is the best one. It guarantees the generation of encodings always where $\mathcal{F}$ is the lowest possible. The black line represents the *Optimal* encoding generation technique, it is a slightly better than the first two methods where the mean is centred in the middle of the cost function, between the minimum and maximum values, that are 173 and 225 respectively.

Hence, by means of *Simulated annealing* technique it is possible to generate relatively few and good solutions that might be synthesised and compared reaching the best possible area compromise out. For instance, a very good solution is reached in terms of area even by trying 10 encodings only with a reasonable low error, results

on Table 4.2.

Generation	#Encoding	Area $[\mu m^2]$	Time $[s]$	Error from exact
Recursive	5040	220,630	2105	0%
Random	100	247,970	17	12,39%
Optimal + SA	10	237,520	2	7,65%

Table 4.2: Comparison between encodings generations applied to *CPOG* on Figure 2.7.

As one may notice from the statistics on Table 4.2, results confirm what has been said on the Section before. The only generation technique allows to find the exact solution is the first one, which indeed inspect the overall solution space. This is even the reason which leads the execution time to be very high with respect to other methods.

Random generation algorithm instead is able to to find a good solution with a greatly reduced amount of time, even if it wastes lots of time inspecting useless solutions. It is worth mentioning that this technique worsens as soon as representation becomes wider, probably because number of good solutions do not increase exponentially as the number of possible solution does (according to Figure 4.7).

Finally, solutions generated via *Optimal-generation* technique tuned by *Simulated annealing* optimisation method allows achieving an extremely good solution in an greatly reduced amount of time. It might be a really good result and I think it might be also exploited on bigger representations, since it is very fast.

4.4.5 Generation outcome

Overt this Section, all the generation techniques developed in this work has been analysed. According to statistics, showed on Table 4.2, the heuristic illustrated in Section 4.4.3 and 4.4.4 seems to be the best since it is an extremely good compromise in terms of time and area trade-off. It pareto-dominates the *Random* algorithm since the results obtained are better, and obtained in a shorter amount of time. Furthermore, it allows achieving a good result with 7,65% of error with respect to exact solution, which is a fairly good result considering the extremely low amount of time it takes for reaching such solution out.

On the light of above, I could state that either cost function described in Section 4.2.1, and generation algorithm illustrated over this Section work well and should allow designers reducing area consumption of circuit represented.

Certainly, these results should be experienced on bigger representations to be tested properly; in Chapter 6 findings will be experienced on a *Conditional Partial Order Graph* modelling an Instruction Set Architecture present in a real processor, the `Arm Cortex M0+`.

Chapter 5

Workcraft integration and custom op-codes

`Workcraft` is a recent software developed by the School of Electrical and Electronic Engineering at Newcastle University meant to support different graphs-based models. It allows designers working with different representations for concurrent system design using a standard and easy framework where whichever behaviour of a system could be modelled through graph representations composed by events and relationships between them. It was presented for the first time by Ivan Poliakov et al. on [54].

Some of the models supported by `Workcraft` are *Dataflow Structure*, *Petri Net*, *Signal Transition Graph* and many others. More information could be found on its dedicated web-site [27] constantly updated.*Conditional Partial Order Graph* representation is one of the models supported. So far, op-codes for synthesising the final controller were not carefully chosen in `Workcraft`, but given in the easiest way in a sequential fashion: 0 - 1 - 2 - 3 and so on, or managed by the old tool version if the size of the CPOG was small enough.

As my work aims to demonstrate, it was extremely inefficient from the area point of view. Hence, in this Chapter I want to fill in this gap and show to reader, without going through code development, how I integrated the optimal encoder (analysed on the previous Chapter) for inspecting an optimal area solution under this software in such a way to give an understandable interface for an easier usage, even for less expert designers. Different snapshots of the Graphical User Interface will be showed, as well as a description about how the interface between `Workcraft` and the encoder works will be presented.

Furthermore, I will be discussing about another feature I added to tool that may be extremely useful for a practical encoding purpose, that is assigning to any instruction classes an op-code on different number of bits. Notice that, hereinafter *CPOG Programmer* will be called *SCENCO*, this name stems from the two words

Scenario E**nco**der and was given by Dr. Andrey Mokhov to first Encoder tool incorporated into `Workcraft`, the one my tool is going to substitute. The source code will not be reported on this thesis, but if the reader is interested might find the source code freely available on [55].

5.1 Custom op-codes

Designers tend to reduce the op-code ensemble length for several reasons. As *RISC* (**R**educed **I**nstruction **S**et **C**omputer) demonstrated along the past years, having linear and simpler architectures makes processors faster and more energy efficient either because of a reduced instructions decoder is needed, and because of a regular and deeper pipeline could be exploited.

Nonetheless, it may not be easy to develop an *Instruction Set Architecture* both on reduced and regular number of bits for all the instructions. For instance, let us think about two totally different instructions as could be a *Shift* and an *Unconditional Jump*. The former one might need different attributes, such as the operand to be shifted and the one where to put the result on, the number of bits for shift operation and the direction, the type of shift (logical/arithmetical); while an *Unconditional Jump* may just need the address where to jump regardless the op-code of the instruction. This might lead designer to assign an op-code on reduced number of bits to *Shift* instruction in order to reserve other bits for other purposes.

This is achieved in my tool by letting designer selecting number of bits one wants an instruction class to be encoded on. It is simply done internally by means of Don't Care conditions. For instance, by using a set of op-codes composed by 4 bits each, the set will be composed by $2^4 = 16$ solutions. Assigning to a graph an op-code of 4 bits means deleting that particular solution from the set, since it must not be used anymore for other instruction classes. Instead, giving to same graph an op-code on less number of bits (i.e. 2) means deleting four possible solutions from the entire set.

Therefore, assigning to a Partial Order the op-code `01--` means setting the op-codes `0100`, `0101`, `0110` and `0111` as already assigned and not exploitable any more. As one could notice, two least significant bits are considered as Don't care conditions, in such a way to remove the other op-codes in the op-code ensemble which match with the shorter encoding.

On Figure 5.1 an example of such algorithm is depicted, where the set of op-codes is represented after and before assignments. On the left-most part the complete set where all op-codes are available is represented, in the center side after that op-code `1010` was assigned and deprived from the set (black circle models an encoding no more available). Finally on the right-most part of Figure the set after a 2 bits encoding assignment is showed. `01--` are the most significant bits.

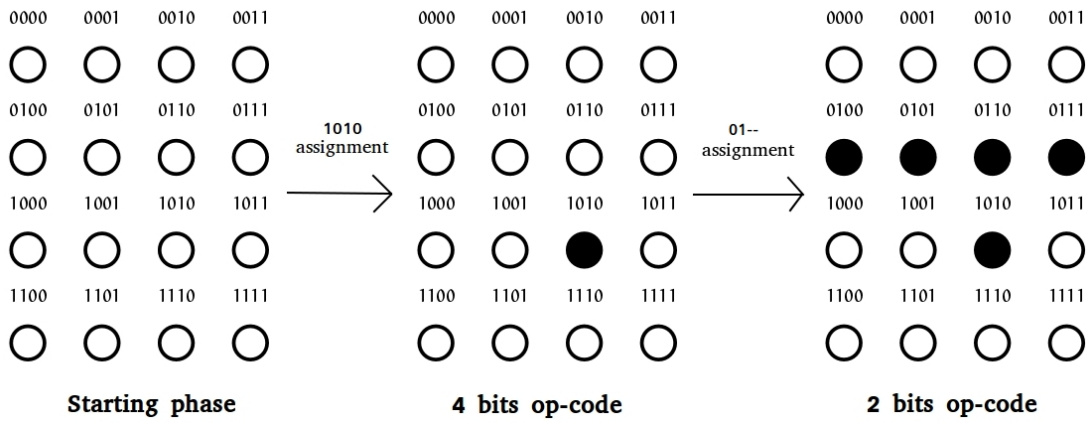


Figure 5.1: Op-codes assignment example.

As one might observe, assigning a reduced op-code to a *Partial Order*, may mean saving area for the final decoder as well. It is not always true and it may impact negatively the final controller. It is because, as showed by Figure just described, assigning a reduced encoding means also depriving from solution set more elements, and therefore having a reduced set of op-codes available for encoding graphs left. Hence, it might impact cost function negatively and making solutions worse than ones encoded on total number of bits.

Nonetheless, *Optimal* generation technique described on Section 4.4.3, adequately modified to support such encodings, is meant to optimise the solution even by setting instruction classes on reduced number of bits, therefore loosing a small amount of area.

An issue designers may face with is trying to set a very short op-code to a graph, making the entire set not enough wide to encode all the *Partial Orders*. For instance, in a set composed by 16 op-code elements, if one chooses to encode one graph with 1 bit only, 8 op-codes will be removed from the ensemble giving the possibility to encode 8 more graphs only on total number of bits. Designer should take into account this issue, trying to increase maximum number of bits of op-codes if a really short op-code is strictly needed, thus increasing number of elements of set from 16 to 32. Tool addresses this requirement letting designer the possibility to choose number of bits of each op-code separately.

The user interface I developed in order to let designers selecting custom op-codes for the *Partial Orders* is composed mainly by three different elements.

0/1: to set a custom bit to a whichever op-code;

X: it sets a Don't Care bit to custom op-code, tool will be in charge of finding best possibility for such bit;

-: it removes the bit from the op-code, in this case the encoding associated to this *Partial Order* will be shorter.

User can freely assign digits described on top on the op-codes as long as they are expressed correctly, avoiding repeated op-codes for instance. If an error into custom assignment would occur, a window will pop up signalling error's reason.

5.2 Workcraft integration

As already described in the introduction of this Chapter, `Workcraft` is a software that supports different models and provides a friendly user interface for designers who want to model event-based systems. It is written in `Java` following object programming paradigm and it contains different internal plug-in, one for each representation supported.

On this Section I am going to present the methods and the results I obtained on *CPOG* plug-in modification, mainly dividing the description into the connection part: where the link between `Workcraft` and *SCENCO* is presented and the Graphical User Interface part: which models the front-end, that is the graphical part of the tool.

5.2.1 Front-end and back-end connection

Scenco was written and compiled in `c++`. In order to connect it to `Workcraft`, the tool is executed from `Java` code providing all the needed parameters and afterwards reading the output as it was written into a file. Several modifications were done to starting tool in order to adapt the low level software to the front-end part, following the most important ones will be showed.

File paths

One of the first needed modification I had to perform was setting all the paths of the files needed to connect Workcraft environment to *SCENCO*. For pursuing this purpose, new input parameters were added to tool, one for each file passed by parameter:

- file description of the model considered;
- library file in genlib format, for selecting the library to consider during the synthesis process;
- `espresso` and `abc` tool executable, needed to *SCENCO* to work properly;

- custom encodings file, prepared by `Workcraft` by front-end plug-in and passed to *SCENCO* in order to set custom op-codes to the tool.

The paths are always given to the Encoder by giving the absolute path of the files, in order to avoid any incoherency problems due to directory where `Workcraft` is currently installed on and in order to have a good flexibility.

Statistics and error detection

Another issue I had to deal with was the errors and statistics detection, or better how to read the output of *SCENCO* in order to detect any possible errors, as well as statistics and useful information coming from the low-level tool.

I tackled this issue by assigning different *tags* to output of the program, bounding useful text lines with them, in order to be able to figure out from a high-level perspective which parts should be considered, and which not. On the bottom all the *tags* used into the output are listed and described.

- **.error / .end_error:** text lines bounded by these two tags signal an error happened into the processing flow;
- **.area / end_area:** they signal the minimum area found by *SCENCO*;
- **.statistics / end_statistics:** various statistics concerning intermediate execution of the tool;
- **.formulae / end_formulae:** all the information about encoding results found (formulae, controller, truth tables).

When high level software reads the output of the tool and one of the tag listed above occurs, `Workcraft` behaves in a different way depending on the *tag* present. When *error* tag is reached, `Workcraft` displays on screen the error message so that user could quickly understand what is wrong in the current execution. When *statistics* tag is reached instead, the program simply prints the text lines into the part related to the output window space, since statistics might contain information useful to designer. *Area* and *formulae* tags whereas contain information useful to core of `Workcraft` and are not displayed on screen, they will be used inside the program to manage all the back-end features of the model.

In this simple way, all the information are taken into account by the software, wasting nothing.

Continuous mode

As mentioned into first chapters of the dissertation, where the heuristic algorithm is explained, the higher the number of instruction classes present into the model considered, the wider the number of possible solutions potentially available for the analysis. Therefore is implicit that, the more the number of solution analysed by *Programmer*, the better will be the result obtained.

Hence one of the most effective feature I implemented into plug-in connected to *Encoder* is the **Continuous mode**. Through this option, the designer may run the encoder through in an interactive fashion where continuous solutions will be analysed displaying step by step the best result obtained so far. The user might let the tool run until is satisfied by result, at that point one just need to press *Stop* button to store the best result and keep working with it.

The designer may want to use this feature when is constrained by very strict area requirement.

5.2.2 Graphical User Interface

In this part, the graphical user interface of the tool will be showed and briefly described. Since *SCENCO* is an autonomous tool regardless few parts as the custom encodings option, it does not require an extensive interface for the user who is limited to setting few configuration parameters.

It is an extremely advantage from the point of view of time to market characteristic of the design. Designer indeed, may be free to think about the core part of the model, letting tool works for optimising as much as possible the design under work.

As one may notice from Figure 5.2, the interface is divided in two parts, on the top part various configuration parameters are present, starting from *Mode* of the tool to *Custom encodings* flag, while on the bottom part a table dedicate to custom encodings is present.

Respectively, starting from the top of Figure 5.2 let us discuss each *SCENCO* setting: through the **Mode** box, designer can choose between four different generation types through which the Encoder will generate the encoding solutions, there are the techniques described on Section 4.4, plus three further options: *Single-literal search* which generates one solution in such a way to have no more than one literal per vertex/edge (for further information refer to dedicated website on [27]), *Exhaustive search* which performs a deep search and *Sequential search* where the solution is the most basic one (0 1 2 3 and so on). *Strategy* box instead, is present only if *Heuristic-guided search* is selected and it can distinguish between the techniques discussed on Section 4.4.

Second box (**Generate equations for**) aims at optimising the structure either

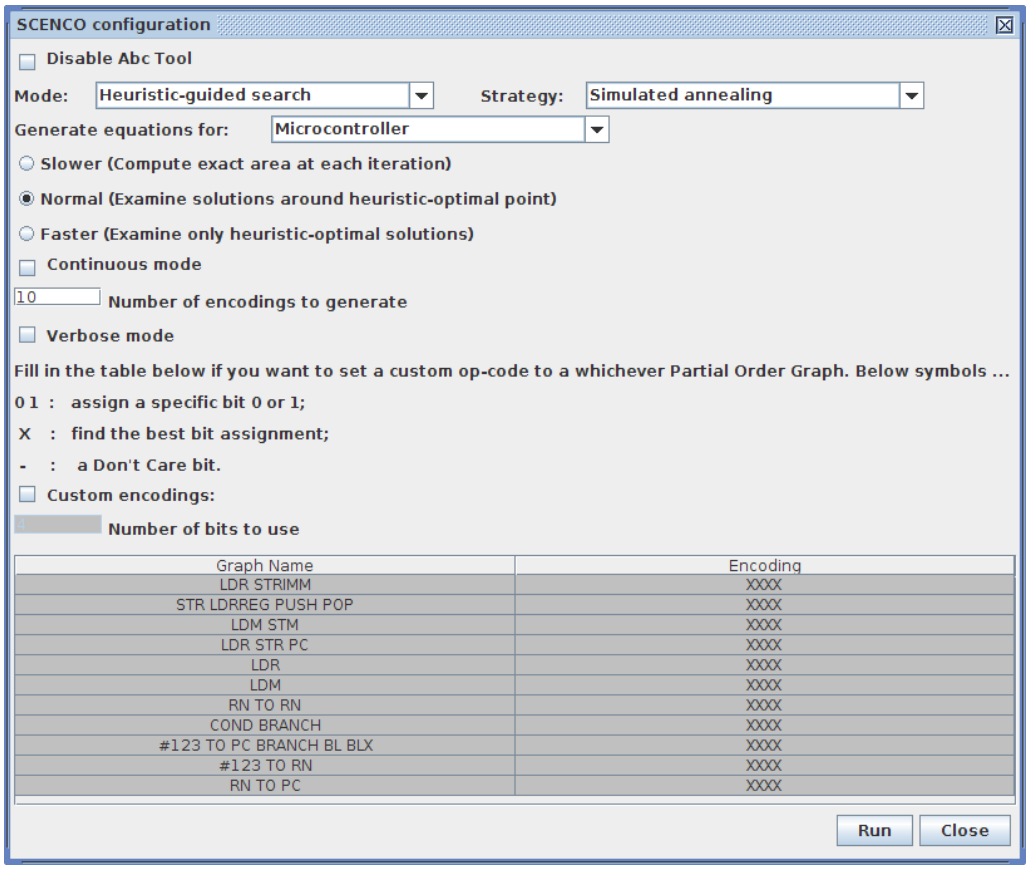


Figure 5.2: Graphical user interface of *SCENCO*.

for *Microcontroller* or for the *CPOG*. The first option respectively builds the controller, able to send to each element *ACK* and *REQ* signals in order to control the control-flow of the entire structure. Second option instead does not care about the control-flow structure, but just takes into account conditions on vertices, trying to reduce them as much as possible.

Afterwards, three different options might be selected by user. **Slower** disables the cost function approximation. It is meant to show to designer how much is the time gained by using the cost function I sought. By disabling it, *SCENCO* will compute the solutions trying to optimise them computing step by step the area of the entire circuit spending a bigger amount of time compared to one spent to optimise the cost function.

Normal -faster flags instead, changes the number of solutions tool would finally synthesise during the last phase of the optimisation. As already mentioned into previous Chapter, after that tool generates the solutions, each one with a different

$\mathcal{F}$, it will be in charged of synthesising it and getting the final area of the controller. With this option the designer, may choose to let the tool synthesise all the final encoding solutions, or just the one with absolutely minimum weight of function $\mathcal{F}$. For optimisation's sake, it is extremely better let the tool analyse all the solutions generated due to a certain intrinsic degree of uncertainty of the cost function, but *SCENCO* would loose much more time to scan all the final ensemble.

Number of encodings to generate, **Continuous mode**, and **Verbose mode** don't need further explanations since their purposes are straightforward, regardless second option respectively which has been explained on Section 5.2.1. Finally, **Custom encodings** flag enables the designers to set custom op-codes, already discussed on Section 5.1.

This friendly user interface allows designer to experience with this software in a fairly easy way, letting the freedom to think about the core part of the representation, that is the model of the system and the connection between any events.

Chapter 6

CPOG applied to ISA development

Conditional Partial Order Graph representation can be used to model different types of systems beside dataflow structures. As already mentioned in fact, it can be used to represent and support the design of *Instructions set architecture*, because of its capability to capture different behaviour of a system in a really compact and specific way. As pointed out on [52] indeed, CPOG-driven design flow might be efficiently applied to such application speeding up the whole ISA design starting from high-level specifications. Let us think for instance to different instructions running on a microprocessor, they might exploit different modules as well as go through different pipeline paths making the whole process of datapath-design extremely complicated, leading to the need of a compact and efficient model to capture the various behaviours instructions may assume.

For demonstrating the versatility of such representation and design-flow, I want to apply it on an *ISA* already present on the market, the one supported by *ARM* `Cortex M0+`. It fits well to this aim because it was already partially modelled on [29], and because of its simplicity given by the applications the processor was meant for (briefly introduced on next section).

Along the next Sections, the processor will be briefly described according to technical manual downloadable from the corresponding web-site [28], afterwards the model will be introduced partially and finally the heuristic algorithm will be applied showing to reader the advantages a real processor may benefit from this approach, pinpointing on area saved for the controller of the datapath. It is a really good test for the cost function presented before, where it could be applied on.

6.1 Arm Cortex M0+

Arm `Cortex-M0+` is a RISC-based 32-bits (**R**educed **I**nstructions **S**et **C**omputer) processor. Architectures belonging to `Cortex-M` family are meant to be used on deep embedded applications, or more in general on applications where power and area are hardly constrained: for instance bio-medical or mobile applications, where the lifetime of the devices or the electro-magnetic emissions are the main concern.

The **I**nstruction **S**et **A**rchitecture used on `M0+` belongs to RISC family. It is an *ISA* which aims at reducing the complexity of the instructions in order to be decoded more efficiently, and as a consequence having several advantages with respect to *CISC* (**C**omplex **I**nstruction **S**et **C**omputing) architecture, such as the possibility by the microprocessor to run at higher frequencies. Having the instructions on a fixed number of bits allows the structure of the pipeline to be more flexible and longer (composed by higher number of stages). The only disadvantage is the higher length of the code which needs a bigger instruction memory to be stored. Nonetheless, although years ago it was a really big problem due to the low budgets available and the higher cost of the single transistor, nowadays it is not and *RISC* architectures are present everywhere.

According to technical reference manual of *Arm* `Cortex M0+` ([28]), it embeds two pipeline stages in order to lower the power consumption as possible and support *Thumb/Thumb 2* instruction set architectures. *Thumb* is an ISA introduced by Arm that was aimed at reducing the code density of software running on the processor. It was achieved by supporting instructions on 32 and 16 bits as well, getting an extremely good improvement on power consumption. Drawbacks of such structure are a lost in terms of speed and higher delay. *Thumb 2* is an extension of first ISA version, where beyond the introduction of other instructions, designers achieved almost the same performance of 32 bits *ARM instruction set*, making the whole μprocessor a leading component in a big slice of the embedded system's market.

As one might notice, although this processor contains several complex and useful features, is fairly small for meeting the tight constraints of embedded applications, hence it is a perfect example for the new algorithms. Over the next Section the model of some instructions will be showed and described.

6.2 ISA designing technique

As usual on event-based models, the key point to represent an *instruction set architecture* of a general processor is grouping the instructions into multiple functional areas. In each area different instructions might be present, according to their characteristics and the steps they need to go through in order to be executed. For instance, let us discuss about the arithmetic instructions: even though each single

operation generates a different result, the step the two operands must go through, depending on where they are fetched, are the same. Hence, it is convenient to group all the instructions which fetch the operands by same sources inside one single group.

Moreover, another step a designer need to follow for designing a consistent ISA with *Conditional partial order graph* is to keep a certain degree of coherency for the modules modelled as vertices of the graph. Let us take as an instance again the arithmetic instructions since they fit well to explanation purposes due to their reduced complexity. Since the goal here is to share as many modules as possible increasing the complexity of the control unit, reducing the one of the datapath, designer needs to be consistent during the design phase of whichever module, taking into account all the characteristics a component should embed in order to be shared by different instructions of the system.

For instance, if one wants the ALU the be shared by the whole arithmetic instructions set, it is worth thinking about the ALU as a single component, seen as a black box from the external environment, accepting two inputs and generating one output, no matter where such inputs come from. On the other hand, a designer may want to be more precise and splitting the ALU into internal components such as the multiplier, the adder/subtractor or the divisor. By designing the graph as latter, one would have the advantage of keeping the level of abstraction quite low, so that to allow the decoder to distinguish between a MUL or an ADD instruction, even if it would increase the number of *Partial order* graphs since these two instructions just mentioned could not be grouped into a single group. While if one would set the ALU as the arithmetic module, designer should take into account the need of an internal decoder able to distinguish between different ALU instructions.

As this analysis wants to demonstrate, *CPOG* are really flexible and can capture the behaviours of a system form different abstraction-levels. It is up to the designer thinking about the degree of abstraction one wants to use in his model.

Once I have introduced the key points of designing an *ISA*, let us concentrate on discussing about a real example on the *Arm* `Cortex M0+`. Let us represent the arithmetic instructions considering the commands where the operands are present in the internal registers. The processor, either for addition, subtraction, division and multiplication need to perform exactly the same steps for executing the operations correctly, below the general pattern of an ADD, SUB and MUL instructions is depicted, according to manual [40].

```
ADD <Rdn>,<Rm> # Addition Rdn = Rdn + Rm
MUL <Rdn>, <Rm> # Multiplication Rdn = Rdn * Rm
SUB <Rdn>,<Rm> # Subtraction Rdn = Rdn - Rm
```

After instructions have been fetched and decoded, operands need to be fetched from corresponding registers, then the actual result should be computed internally into the *Arithmetic Logic Unit* and finally it must be written into the selected register

(addressed by *Rdn* for instructions just listed). All the instructions referring to this structure of events could be grouped together in such a way to be encoded with the same op-code, the *ALU* will be in charge of switching between particular instructions via another decoder internally.

In the next Section, the model used to represent the *ISA* of *Arm* `Cortex M0+` will be described.

6.3 Armv6-M representation via CPOG

Nearly all the instructions supported by `Cortex-M0+` were grouped into 11 *Partial Order Graphs* used by the heuristic presented in this dissertation in order to be encoded minimising controller size. Descriptions of graphs were inspired by the descriptions given on [29], where 9 instructions classes are described out of 11 analysed in this dissertation.

The instruction set architecture I am going to deal with is the `Armv6-M`. A complete description of such architecture may be found on [40], where the whole structure of the *ISA* is described by `Arm`. Nonetheless, I want to start my description of the model beginning from the system to represent: "`ARMv6-M` supports the Thumb instruction set including a small number of 32-bit instructions introduced with Thumb-2 technology".([40], Section A4.1). Although the *ISA* does not support ARM instruction set, the two *ISA*s can work together via a technique that allows the developer to switch to the Instructions he wants to use. In the Appendix A each class is showed and described, but if the reader is interested on inspecting all the functionality such architecture embeds, can refer to "`Armv6-M` architecture reference manual".

Reader should notice that the aim of this Section, is not modelling the Thumb instruction set in the best possible way, but having a quite big system composed by different *Partial Order* graphs in such a way to test the tools and the algorithm for automatic composition. Each designer may come up with a different model for the same architecture since, as already mentioned, such representation allows hardware developers to represent their systems flexibly. Therefore readers should focus the attention on the real goal of this Section neglecting any incoherency that may happen during the design phase.

`ARMv6-M` instruction set is a very complex architecture, which requires an extremely big effort to be modelled as well as more controllers for different level of abstraction, hence it would not be fair comparing the size of the real control unit *Arm* `Cortex-M0+` embeds internally with respect to this one I am going to generate. The former one in fact, is way more complicated than the one generated for the aim of this work and includes more instructions than the ones represented by *CPOG*.

In summary, this Section aims to demonstrate that such representation may be

cleverly used to model such a complicated system, and in particular the usefulness of the heuristic algorithm introduced by this research. All the eleven graphs designed are presented and described on Appendix A, over the next few pages I am going to show the whole *Partial order* representations as well as a couple of example of the graphs designed. On Figure 6.1 the entire *Conditional Partial Order Graph* is depicted.

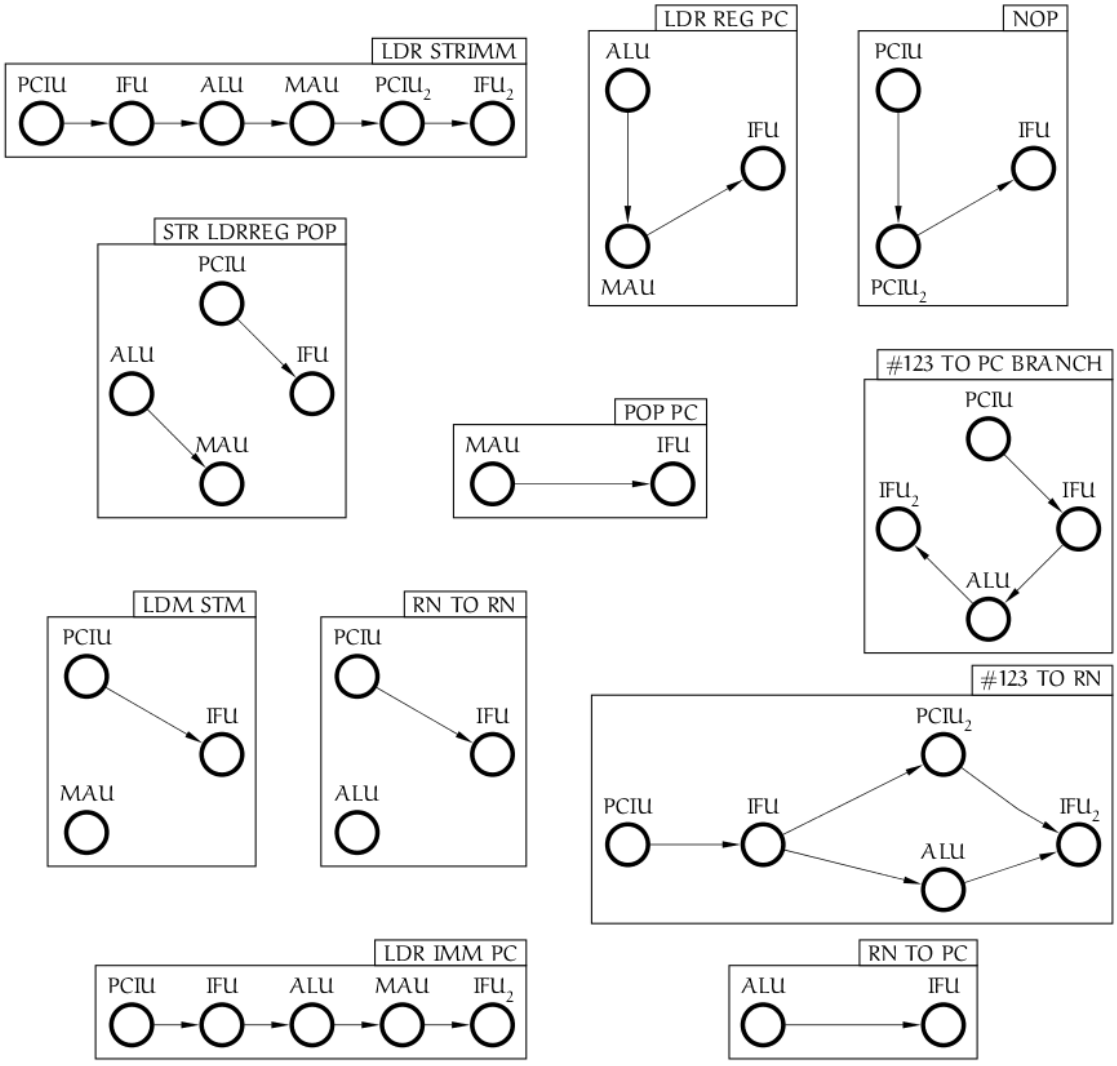


Figure 6.1: ARMv6-M Instruction Set Architecture modelled via *CPOG*

As might be observed, all *Partial orders* are very different each other. Nonetheless, all of them embed the same nodes, which represent the modules present inside the processor for executing an instruction. **PCIU** (**P**rogram **C**ounter **I**ncrement

Unit) is the module in charge of updating the program counter register of the processor in order to move on a new instruction to execute. **IFU** (**I**nstruction **F**etch **U**nit) is the unit for updating the instruction register fetching the new instruction, ready to be executed on the next cycle. **ALU** (**A**rithmetic **L**ogic **U**nit) is the block that performs all the mathematical operations, reader should notice that, in order to distinguish between all the operations that this unit can execute a further low-level model is needed. **MAU** (**M**emory **A**ccess **U**nit) is in charge of updating the handling the memory elements via the *Load/Store* operations and finally **PCIU_2** and **IFU_2** meant for the instructions which need a new instruction to be fetched twice, as a *NOP*.

All these elements are cleverly connected to model the behaviour of several instructions. For instance, let us consider the instruction class *NOP* represented on the top-right corner of Figure 6.1. It clearly models the *no operation* instruction of the processor by means of two consecutive program counter register incrementing operations, in order to skip the instruction which does not require any result to be computed, then fetch unit is in charge of loading next operation inside the instruction register. Additionally, another example could be the *Partial order* named *POP PC*: it represents the pop operation where the program counter register is used as a destination, the **MAU** block indeed is in charge of managing the load operation from the stack, while **IFU** of fetching new instruction addressed by the new *PC*.

Following this pattern, nearly all the instructions have been modelled. Reader is thoroughly fostered to read Appendix A where the whole model is listed.

6.4 Composition results

After that all the *Partial Orders* have been presented on previous Section, the graphs obtained by the composition of the single smaller systems will be presented, focusing on the area consumption coming out. As already mentioned there are plenty of algorithms a designer may use to compose the CPOG, as described on Sections 4.4 and 5, nonetheless in this Section I want to focus the attention of the reader on analysing the solutions come up with the usage of techniques described previously on Section 4.4.2 regarding the Random encoding generation and in Section 4.4.4 about the *Simulated annealing* method applied to such application. Those two algorithms indeed fit well to a *CPOG* with such a high number of graphs as the one described, and additionally designers may find these two techniques suitable to have a rough idea of the area this technique allows to save due to the wide difference of the two methods.

Before discussing about the direct composition of graphs, let us discuss about the solution space. As analysed in Section 2.1.1, the more the number of partial orders, the bigger the size of the solution space according to Formula 2.3. It means that

the number of possibilities present for encoding eleven graphs via sixteen different elements i:

$$\frac{(16-1)!}{(16-11)!} = 10897286400$$

As one might understand, it would be too time consuming inspecting all the solutions. That is why a designer needs to choose the technique the program should use to find a good solution. For sake of correctness, it is worth presenting the sequential solution first of all. In order to be able to compare the other techniques with the most basic one. It is based on assigning op-codes sequentially, hence going from 0 to 10 in order to assign an op-code to each of the eleven *Partial orders*. Below the op-codes assignment:

Sequential op-codes assignment

```
LDR STRIMM: 0000
STR LDRREG POP: 0001
LDM STM: 0010
LDR IMM PC: 0011
LDR REG PC: 0100
POP PC: 0101
RN TO RN: 0110
NOP: 0111
#123 TO PC BRANCH: 1000
#123 TO RN: 1001
RN TO PC: 1010
```

Though it is the simplest encoding one may come up with, it is not the best solution from the area point of view at all. Indeed, the area of the controller generated using the library described on Section 4.1.1 occupies 335,47 μm^2 and the total number of gates needed to synthesise the structure are 50. Afterwards, the reader will be able to compare the results with the areas coming up with other techniques. To complete the sequential solution profile: on Figure 6.2 is represented the compositional graph and below the controller Boolean equations.

Sequential encoding controller

```
REQ_PCIU_2 = ((x_1*x_2*x_3)+(!x_0*!x_1*!x_2*!x_3)+(x_0*x_3)) * (ACK_PCIU +
    !((!x_1*!x_2)+(!x_0*x_2))) * (ACK_MAU + !((x_1*!x_2)+(!x_0*!x_1))) * (ACK_IFU
    + !((!x_2)));
REQ_IFU_2 = (ACK_PCIU_2 + !((x_1*x_2*x_3)+(!x_0*!x_1*!x_2*!x_3)+(x_0*x_3))) *
    ((x_0*x_3)+(!x_1*x_2*x_3)+(!x_1*!x_2*!x_3)) * (ACK_ALU +
    !((!x_2*!x_3)+(x_1*!x_3)+(x_0)+(!x_1*x_3))) * (ACK_MAU +
    !((x_1*!x_2)+(!x_0*!x_1)));
```

```
REQ_ALU = ((!x_2*!x_3)+(x_1*!x_3)+(x_0)+(!x_1*x_3)) * (ACK_IFU +
    !((x_2*x_3)+(x_0*x_3)+(!x_1*!x_2*!x_3)));
REQ_PCIU = ((!x_1*!x_2)+(!x_0*x_2)) ;
REQ_MAU = (ACK_ALU + !((!x_2*!x_3)+(x_1*!x_3)+(x_0)+(!x_1*x_3))) *
    ((x_1*!x_2)+(!x_0*!x_1)) ;
REQ_IFU = (ACK_PCIU_2 + !(((x_2)) *
    ((x_1*x_2*x_3)+(!x_0*!x_1*!x_2*!x_3)+(x_0*x_3)))) * (ACK_ALU + !(((x_0*x_2)) *
    ((!x_2*!x_3)+(x_1*!x_3)+(x_0)+(!x_1*x_3)))) * (ACK_PCIU +
    !((!x_1*!x_2)+(!x_0*x_2))) * (ACK_MAU + !(((x_1)) * ((x_1*!x_2)+(!x_0*!x_1))));
```

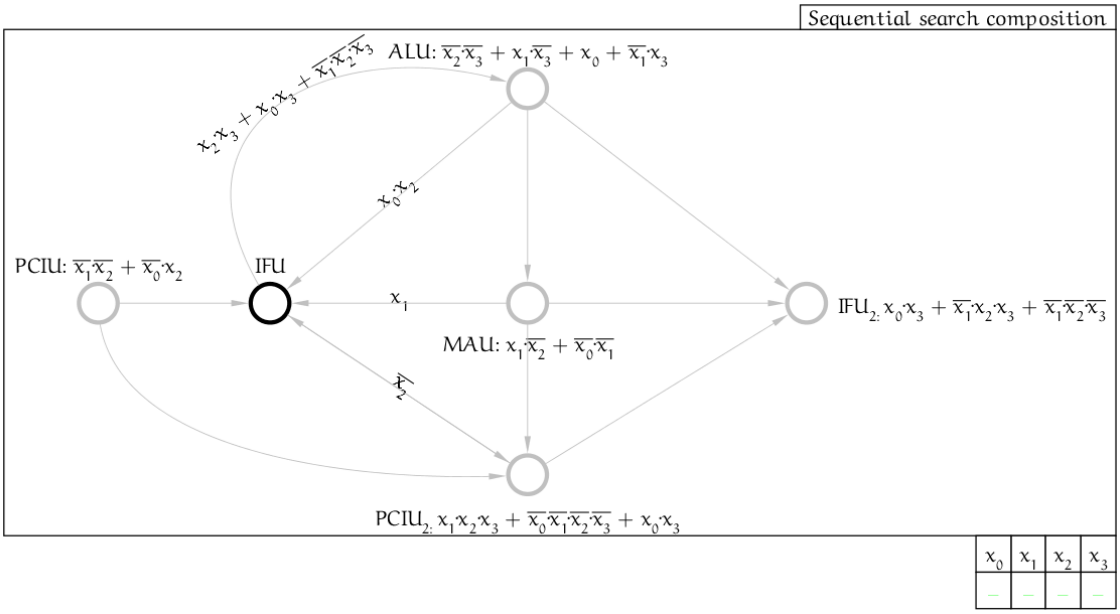


Figure 6.2: Sequential encoding CPOG composition

Once I have introduced the basic encoding, let us discuss about the random solutions. Here the op-codes are generated randomly until all *Partial Orders* are identified by an op-code. Designer can choose how many solutions the program should evaluate, for having a fair comparison in this Section I am going to generate one hundred solution both for random encoding and for the Simulated annealing technique. On Figure 6.3 the CPOG composition is depicted and below the op-codes assignment is showed.

Random encoding op-codes assignment

```
LDR STRIMM: 0000
STR LDRREG POP: 1101
```

LDM STM: 0110
LDR IMM PC: 1111
LDR REG PC: 1001
POP PC: 1011
RN TO RN: 1100
NOP: 0011
#123 TO PC BRANCH: 0101
#123 TO RN: 0100
RN TO PC: 1000

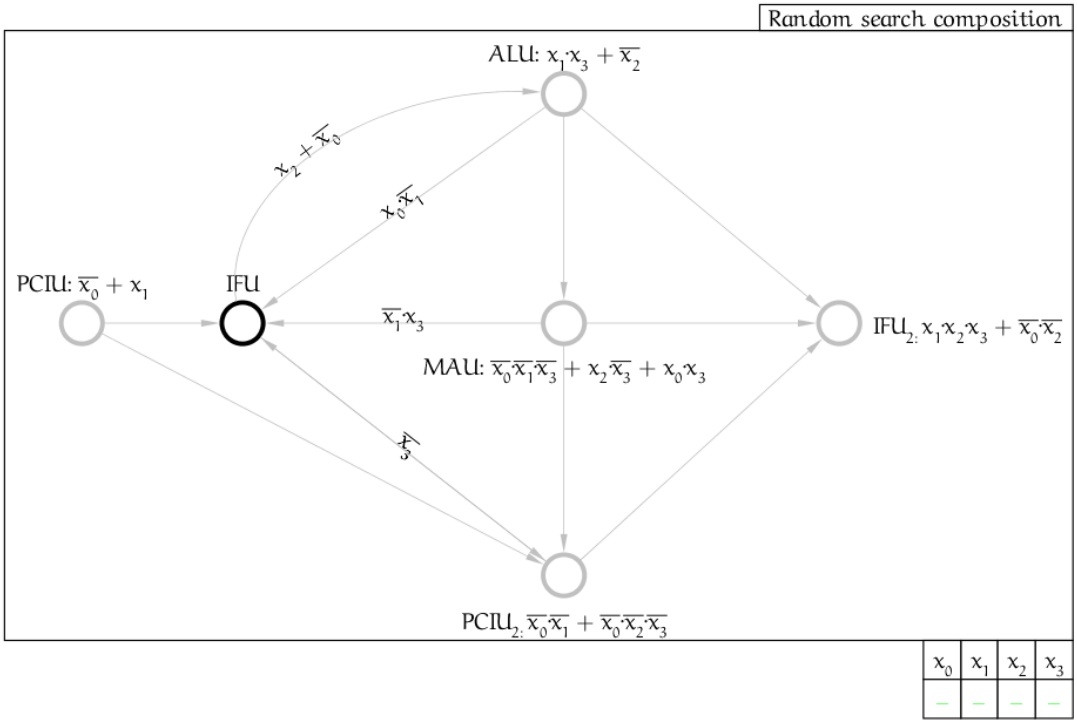


Figure 6.3: Random encoding CPOG composition

As one might notice, the number of literals which compose the conditions on Random CPOG is fairly reduced with respect to the previous one. It is the reason why the area is also reduced, in fact the size of the controller measures 284,53 μm^2 and 43 gates are needed for the synthesis process. Below the controller equations that are shorter in the number of literals as well.

Sequential encoding controller

```
REQ_PCIU_2 = ((!x_0*!x_1)+(!x_0*!x_2*!x_3)) * (ACK_PCIU + !((!x_0)+(x_1))) *
    (ACK_MAU + !((!x_0*!x_1*!x_3)+(x_2*!x_3)+(x_0*x_3))) * (ACK_IFU + !((!x_3)));
```

```
REQ_IFU_2 = (ACK_PCIU_2 + !((!x_0*!x_1)+(!x_0*!x_2*!x_3))) *
    ((x_1*x_2*x_3)+(!x_0*!x_2)) * (ACK_ALU + !((x_1*x_3)+(!x_2))) * (ACK_MAU +
    !((!x_0*!x_1*!x_3)+(x_2*!x_3)+(x_0*x_3)));
REQ_ALU = ((x_1*x_3)+(!x_2)) * (ACK_IFU + !((x_2)+(!x_0)));
REQ_PCIU = ((!x_0)+(x_1)) ;
REQ_MAU = (ACK_ALU + !((x_1*x_3)+(!x_2))) *
    ((!x_0*!x_1*!x_3)+(x_2*!x_3)+(x_0*x_3)) ;
REQ_IFU = (ACK_PCIU_2 + !(((x_3)) * ((!x_0*!x_1)+(!x_0*!x_2*!x_3)))) * (ACK_ALU
    + !(((x_0*!x_1)) * ((x_1*x_3)+(!x_2)))) * (ACK_PCIU + !((!x_0)+(x_1))) *
    (ACK_MAU + !(((!x_1*x_3)) * ((!x_0*!x_1*!x_3)+(x_2*!x_3)+(x_0*x_3))));
```

Last solution a designer should absolutely take into account is the *Simulated Annealing* search. It has been described in Section 4.4.4 and, as I aim to demonstrate following, it generates an extremely good solution in terms of area. Maybe the time for analysing the hundred solution is a bit higher, but it definitely pays off in term of size of the final controller. On Figure 4.11 an example of the cost function optimisation for five different solutions is depicted.

Simulated annealing op-codes assignment

```
LDR STRIMM: 0000
STR LDRREG POP: 1011
LDM STM: 0011
LDR IMM PC: 1010
LDR REG PC: 1001
POP PC: 0001
RN TO RN: 1111
NOP: 0111
#123 TO PC BRANCH: 1110
#123 TO RN: 0110
RN TO PC: 1101
```

As before, above is represented the op-codes assignment, below the controller Boolean equations and on Figure 6.4 the CPOG composition. The size of this last controller is 208,02 μm^2 and 31 gates are needed for synthesising it. Over the next part, these results will be compared and analysed.

Simulated annealing encoding controller

```
REQ_PCIU_2 = ((!x_2*!x_3)+(!x_0*x_1)) * (ACK_PCIU + !((!x_3)+(x_2))) *
    (ACK_MAU + !((!x_1))) * (ACK_IFU + !((!x_3)));
REQ_IFU_2 = (ACK_PCIU_2 + !((!x_2*!x_3)+(!x_0*x_1))) * ((!x_3)) * (ACK_ALU +
    !((!x_3)+(x_0))) * (ACK_MAU + !((!x_1)));
REQ_ALU = ((!x_3)+(x_0)) * (ACK_IFU + !((!x_3)));
REQ_PCIU = ((!x_3)+(x_2)) ;
```

```
REQ_MAU = (ACK_ALU + !((!x_3)+(x_0))) * ((!x_1)) ;
REQ_IFU = (ACK_PCIU_2 + !(((x_3)) * ((!x_2*!x_3)+(!x_0*x_1)))) * (ACK_ALU +
    !(((!x_2*x_3)) * ((!x_3)+(x_0)))) * (ACK_PCIU + !((!x_3)+(x_2))) * (ACK_MAU +
    !(((!x_2*x_3)) * ((!x_1))));
```

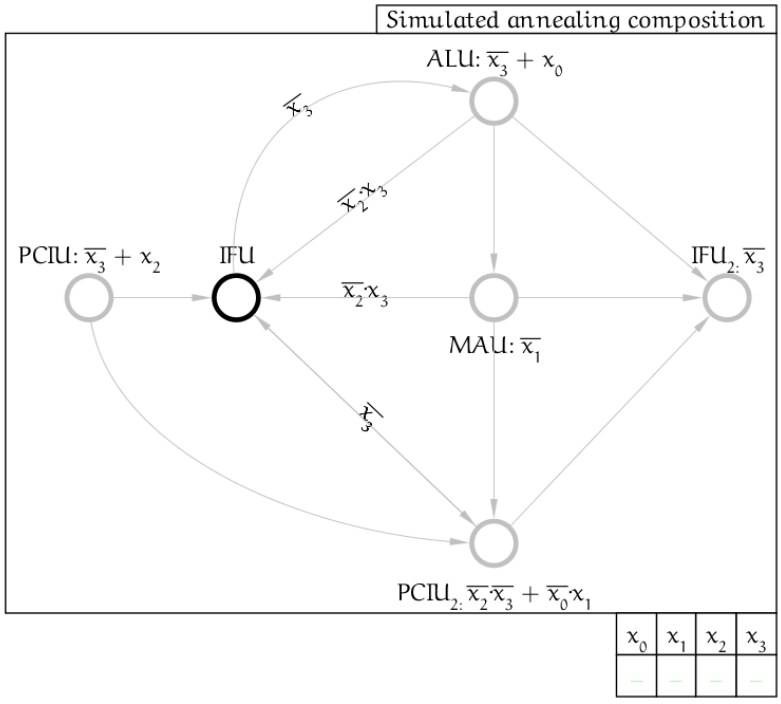


Figure 6.4: Simulated annealing encoding CPOG composition

Before concluding this Section, it is worth mentioning about another technique which show up the best result, even better than the one obtained with *Simulated annealing* optimisation algorithm concerning the execution that has been taken into account for the controller comparison. I am talking about the *Old Scenco* algorithm ([53]) which is able to find a solution which comes up with a controller on an area of 204,80 μm^2 with 29 gates. Below the op-codes assignment is depicted which has been named inside this plugin as *Exhaustive search*, while the composition of graphs is depicted on Figure 6.5:

Exhaustive search op-codes assignment

```
LDR STRIMM: 0000
STR LDRREG POP: 0001
LDM STM: 0101
LDR IMM PC: 0010
```

LDR REG PC: 0011
POP PC: 0111
RN TO RN: 1001
NOP: 1100
#123 TO PC BRANCH: 1010
#123 TO RN: 1000
RN TO PC: 1011

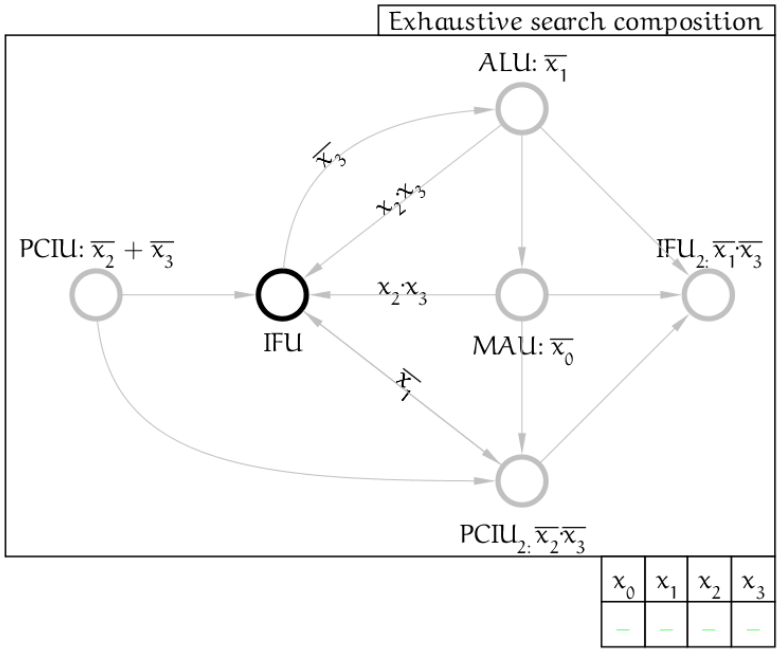


Figure 6.5: Exhaustive search encoding CPOG composition

Below the controller that comes up with this op-code association.
Exhaustive search encoding controller

```
REQ_PCIU_2 = ((!x_2*!x_3)) * (ACK_PCIU + !((!x_2)+(!x_3))) * (ACK_MAU +
    !((!x_0))) * (ACK_IFU + !((!x_1)));
REQ_IFU_2 = (ACK_PCIU_2 + !((!x_2*!x_3))) * ((!x_1*!x_3)) * (ACK_ALU + !((!x_1)))
    * (ACK_MAU + !((!x_0)));
REQ_ALU = ((!x_1)) * (ACK_IFU + !((!x_3)));
REQ_PCIU = ((!x_2)+(!x_3)) ;
REQ_MAU = (ACK_ALU + !((!x_1))) * ((!x_0)) ;
REQ_IFU = (ACK_PCIU_2 + !(((x_1)) * ((!x_2*!x_3)))) * (ACK_ALU + !(((x_2*x_3)) *
    ((!x_1)))) * (ACK_PCIU + !((!x_2)+(!x_3))) * (ACK_MAU + !(((x_2*x_3)) *
    ((!x_0))));
```

Even though this technique exhibits a really good result in terms of area, the exhaustive approach is computationally expensive and it is unlikely to handle bigger examples. For more details about such approach reader should refer to the article where it was described: [53].

6.4.1 Final considerations

Once the four techniques have been executed on the same *Conditional Partial Order Graph*, it is possible to perform some considerations about the results obtained. First of all let us insert all the previous results, in terms of area and number of gates needed for synthesising the structure on a Table (6.1). The worst solution is the Sequential search, it generates the biggest controller, in this case 17,9% bigger with respect to *Random* search and 61,2% bigger than the one with *Simulated Annealing* heuristic. Hence, it is totally inconvenient, but it is useful just for giving a general idea about the area of the controller without using any of the techniques described. Whereas, the *exhaustive search* approach gives a very good result in terms of area, since the controller comes up with it might be synthesised saving even more area than the simulated annealing technique. Sequential search in this case is 63,8% bigger than the exhaustive approach solution.

	Sequential	**Random**	**Simulated annealing**	**Exhaustive search** [53]
Area [μm^2]	335,47	284,53	208,02	204,80
N. Gates	50	43	31	29

Table 6.1: Controller size comparison for Armv6-M ISA model.

In order to compare the two heuristic techniques (random and simulated annealing approaches) and show the soundness of the solutions that the *Simulated annealing* algorithm is able to find, I tried to generate an extremely high number of solutions with Random encoding, and a small one with *Simulated annealing* search. The result is surprisingly good, and showed on Figure 6.6.
As one might observe on the Figure, I generated 100000 solutions with Random method and 1000 with SA technique. My aim indeed was seeking few and sound encodings via the latter technique respectively. On the x-axis of the graph is present the cost function presented in Section 4.2, while on the y-axis the bare area [μm^2] is present. As I demonstrated, the lower the cost function of the solution, the higher is the probability to seek a better encoding in terms of area, so the main goal is to have a heuristic returning encodings minimising the cost function.

Moreover, it is worth mentioning that the best result achieved via the *Simulated annealing* technique is even better than whichever results showed on previous Section. Controller indeed might be synthesised on an area of 191,090 μm^2, with an

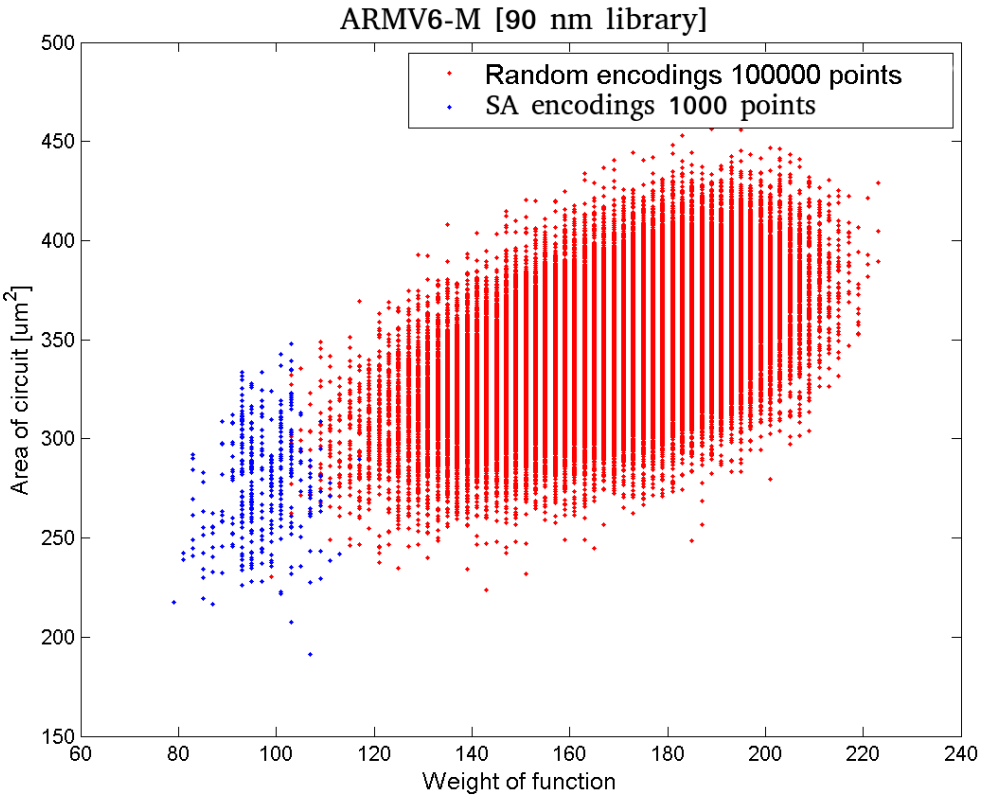


Figure 6.6: Comparison between Simulated Annealing and Random search techniques

encoding centred on a weight of the cost function of 107, according to Chapter 4 where it is stated that a higher band of values in terms of the cost function must be analysed in order not to loose any good results.

As one might observe, even though the extremely high number of solutions generated randomly, the weight of the cost function related to each encoding touches 110, and rarely achieve 100. The reason of the high weight of the cost function resides on the massive number of possibilities the solution space is composed by. Indeed, even though 100000 may seem a high number of solutions, it represents the 0,0009% of solution space only, so it is extremely unlikely to find a really good point. The results get worse if one normalises the points via a Gaussian considering the cost function. As depicted on Figure 6.7 in fact, the average of the function is centred on around 175 that represents the type of solutions is more likely to occur. It represents a negative disadvantage since the aim is not seeking as many solution as possible, but pointing out few and sound solutions.

Observing on the other hand the points and the density function of encodings

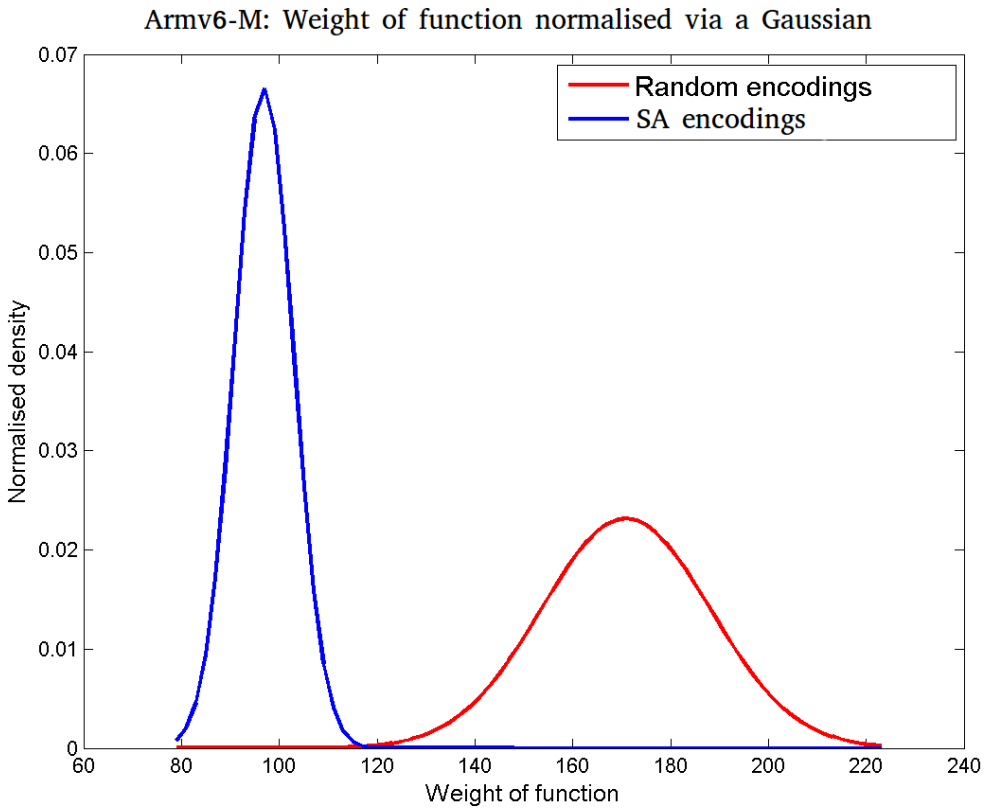


Figure 6.7: Comparison between Simulated Annealing and Random search technique (encodings weight)

generated by algorithm discussed on Section 4.4.4, one might realise the goodness of such heuristic. In fact, even though the number of solutions generated is two orders of magnitude less than the other number of encodings, the average of the weight of cost function is centred on 100 with several solutions that reach 80 out. Figure 6.6 does not need any explanations: nearly all the blue points generated are way better than the red points and moreover, due to the reduced number of encodings sought the time needed to compute them is fairly reduced too.

Hence, I can state that the *Simulated annealing* technique addresses the constraints a designer may need. The blue normal function on Figure 6.7, representing the normalised density of the number of solutions generated with a particular weight, is centred on 100 which represents a really good result allowing to find good area points in a short amount of time since few encodings are requested for achieving an optimal result. It allows to quickly find good encodings which help synthesising the hardware structure in a reduced amount of area, and as a consequence saving also

power.

Lastly, on the light of above I can affirm that one of the main goals of this research has been addressed. The composition of *Partial Order* graphs is now automated exploiting an algorithm which is also able to build the whole system in an optimal way, designers that use this model can now freely compose their own system quickly and easily through the *SCENCO* (SCEnario ENCOder) tool present in the *Conditional Partial Order Graph* plugin under `Workcraft`.

In the next Chapter, I am going to exploit the knowledge I introduced in the Chapter 3 to build from scratch a complete reconfigurable asynchronous dataflow processor, in order to demonstrate that such design-flow tailors extremely well to asynchronous design development.

Chapter 7

Self-timed reconfigurable dataflow processors design

In this final Chapter I want to introduce the reader to a real design of a self-timed reconfigurable dataflow processor. My aim is to make reader understand the easiness of such design-flow exploiting *Conditional Partial Order Graph* model and to present how it fits well to asynchronous hardware design. The methods I described on Chapter 3 will be exploited over the next Sections in order to build an asynchronous pipeline applied to a real application.

The important goal I aim to achieve is highlighting the flexibility of the design-flow, combined with self-timed structures to deal with a real application. This Chapter is structured as following: in Section 7.1 the application I am going to deal with will be presented, then I am going to handle the control-unit design part with the help of *CPOG* model. Afterwards the datapath development is presented switching from an higher abstraction level view (dataflow structure) to a lower level of abstraction (RTL level design).

Reader should notice that the final hardware structure is not finished yet because it needs to be revised and tested. So I am not going to insert in this Chapter final synthesis results such as area and power consumption of the whole structure. Nonetheless, it would not affect the understandability of dissertation since the main purpose of it is focusing mainly on the design-flow.

7.1 Ordinal pattern encoding for ordinal analysis

In this Section I will be presenting the application I want to apply the design-flow to. It is an application-specific processor suitable for Ordinal analysis. Since the whole idea was taken by [41], I am going to refer the description to this paper. Ordinal analysis issue is a statistical method to analyse the complexity of time series.

Over the last few years, the topic of time series analysis has been acquiring a lot of importance as a research area because of the applications it can be applied to, such as financial topics and signal processing in the context of biomedical applications. Since the time series that must be analysed are huge, the topic has been acquiring always more relevance, researchers aim to find new algorithms and hardware structures for better analysing time series, more rapidly and more efficiently.

Ordinal pattern encoding is a particular operation that can be applied for analysing time series in order to evaluate the stability of a consecutive series of numbers. The main concern one should have when applying this operation is to be able to compute how many subsequences are ordered for determining the regularity and the predictability of a time series. As Ce Guo et al. state: "This approach does not depend on a particular time series model, and it is robust against various types of noise."([41], Introduction), this is an advantage because one may reproduce an algorithm, or a hardware structure over different types of applications with the same module.

Inside the paper cited above, the authors present the first reconfigurable accelerator for ordinal pattern encoding operation. During this research I have exploited the main idea of this publication, modifying it accordingly for dissertation purpose, indeed I aim to build an asynchronous dataflow reconfigurable hardware structure running the *Ordianl pattern encoding* operation for different lengths of the subsequences.

Before going on discussing about the hardware architecture the research presents, let us give a definition to *Ordinal Pattern encoding*:

Definition.[1] Given a sequence of n distinct values $b = (b_1, ..., b_n)$, the ordinal pattern of b is mathematically described by a permutation $\pi = (k_1, ..., k_n)$ such that $b' = (b_{k_1}, ..., b_{k_n})$ is in ascending order.

For instance, if one has to apply the *Ordinal pattern* operation to the time series $b = (58, 20, 56, 10, 22)$ for the subsequences of length 3, one needs to consider the three subsequences $b_1 = (58, 20, 56)$, $b_2 = (20, 56, 10)$ and $b_3 = (56, 10, 22)$. Therefore, the *ordinal pattern* of these three sequences are three arrays: $\mathcal{OP}_1$ = (2,3,1), $\mathcal{OP}_2$ = (3,1,2) and $\mathcal{OP}_3$ = (2,3,1). These results are really useful to *Ordinal analysis* because they contain information about the regularity of the whole time series b, and can be analysed deeply looking at the distribution of these results.

First consideration one might perform is that the number of subsequences present in a time series composed by n uncorrelated numbers, is a number of sequences of length s equal to $n - s + 1$. Hence, the number of *Ordinal pattern* operations that

[1]Definition present on [41], Section II, Paragraph A

must be applied to a time series strongly depend on the length of the whole time series n, usually one does not need to inspect subsequences longer than 12.

As might be observed by the mathematical relationship above, this operation it is quite demanding from the time and resources point of view. Indeed, implementing such operation in hardware would require to sort the numbers of the sequence each time; and as a consequence since sorting operation requires an high number of comparison depending on how many numbers should be sorted, it does not fit well to hardware perspective. It is in fact difficult to have a good trade-off between speed of computation and area. Many algorithms were described for sorting numbers via hardware structures, but all of them are really area consuming ([43], [44]). Few research have inspected the topic of acceleration for time series, one of them refers to a software kind acceleration based on CPU platform and is described on [42].

The method I am going to review is based on an algorithm which converts the time series into a *Lehmer code*, that will be described below, and then compresses the results via a factorial number representation. It was developed on [41]. For the purpose of testing the design-flow of this research I am going to erase the compression part which, as will be described, is composed by various number of multipliers only. Before introducing the real algorithm and showing the structure of it, it is worth inserting the definition of a *Lehmer code*.

Definition.[2] Let $x = (x_1, ..., x_n)$ be a sequence of length n, the *Lehmer code* of x is also a sequence with length n in the form of $\mathcal{L}(x) = (l_1, ..., l_n)$ where:

$$l_i = \#\{x_j : j < i, x_j < x_i\}$$

On the light of the definition above, the *Lehmer code* of the sequence x = (25,35,12,89,2) will be computed as following:

$$\begin{aligned}
\mathcal{L}(x_1) = 0, \quad &\Rightarrow \quad \nexists\, x_j : j < i, x_j < x_i \\
\mathcal{L}(x_2) = 1, \quad &\Rightarrow \quad x_j : j < i, x_j < x_i = x_1 \\
\mathcal{L}(x_3) = 0, \quad &\Rightarrow \quad \nexists\, x_j : j < i, x_j < x_i \\
\mathcal{L}(x_4) = 3, \quad &\Rightarrow \quad x_j : j < i, x_j < x_i = x_1, x_2, x_3 \\
\mathcal{L}(x_5) = 0, \quad &\Rightarrow \quad \nexists\, x_j : j < i, x_j < x_i
\end{aligned}$$

Hence, *Lehmer code* of the sequence x described above is $\mathcal{L}(x)$ = (0,1,0,3,0). As one might notice, in order to apply this operation to a whichever sequence of consecutive

[2]Definition present on [41], Section III, Paragraph A

numbers, one should compare the most recent value just came into the pipe with the other values instantiating as many comparators as the length of the subsequence one wants to compute the code for. Another consideration I can perform is that at every step of the chain the most recent value will be always taking the 0 code because it has not any values preceding the sequence, while the last number of the chain, that is also the first came into it will be pushed out from the sequence losing last number of the code. Concerning all the other elements in the pipe, the *Lehmer code* associate to each element should be increased by one if and only if the new value is less than the considered one, otherwise it would remain the same. Formula below represents such relationship:

$$\mathcal{L}(l_i) = \begin{cases} l_i + 1 & if\ x_i > x_0 \\ l_i & otherwise \end{cases} \tag{7.1}$$

Lehmer code tailors extremely well to hardware due to its capability to compute each number separately by the other ones. Internally it embeds the information representing the order the numbers compose the sequence are disposed by, and it is enough for *Ordinal analysis* purpose. Moreover, all the operations leading to result can be executed in parallel relying on the Formula 7.1, so it is also convenient either in terms of resource consumption and of speed. As mentioned before, the complete algorithm described on [41] relies also on a layer to compact the results. Nonetheless, since for the aim of my work it was not needed at all, it will be completely neglected and the structure will be presented according to what I aim to build.

On Figure 7.1 is depicted the structure needed to compute the *Ordinal pattern* operation for sequences of length 5 following the *Lehmer code* technique. As one might observe, the whole architecture is structured mainly in five layers, each one is in charge of performing a particular operation which is useful to the entire structure to support *Ordinal analysis*. The top layer, also called *Sequence buffers* is composed by as many number of registers as the length of subsequence the designer needs to apply the *Ordinal pattern* for. Therefore in the case of a 5 length subsequence, one needs to instantiate 5 registers leftmost included, which must feed all the comparators below. Then $n-1$ comparators are present, where n represents the length of the subsequence. Their role is to compare each number of the subsequence with the one just entered in the chain in order to establish whether the condition is verified or not; if it is the output of the comparator signal a logic `1`, otherwise a `0`. Third layer is composed by $n-1$ adders. They are in charge of updating current *Lehmer code*; reader should notice that the first adder has fed by a logic `0` from one input, since as explained before, the first digit of the code would never be higher than 0. Fourth layer contains $n-1$ other buffers which are needed to store the current code at each iteration and for performing the shift feeding the adders of the layer above. Finally the compactor module which simply concatenate the numbers over a bus

sending all of them to output together. It is not a proper module actually since it could be obtained via a certain wires connection.

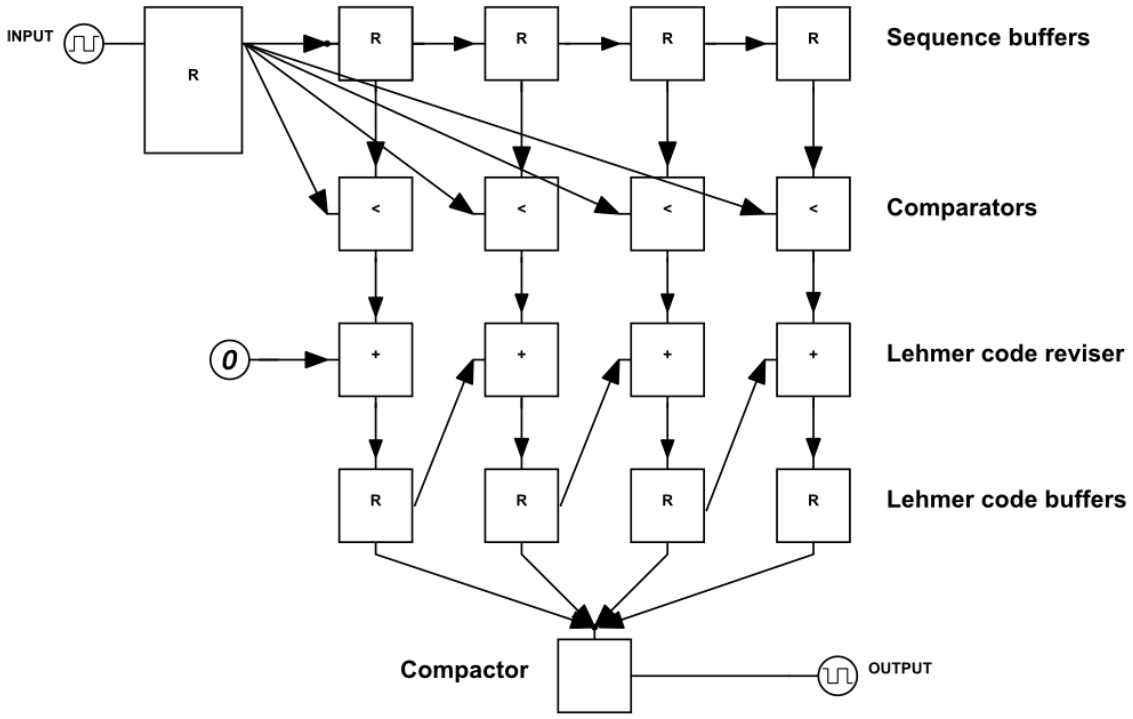


Figure 7.1: Ordinal pattern encoding algorithm for subsequences of length 5, hardware structure inspired by [41].

First consideration which could be done is that the structure is extremely flexible in the length of the subsequences to be applied for. Indeed, for supporting a longer or a shorter sequence, a designer may just need to modify the number of the modules compose the layers making the whole path longer or shorter. Here it comes my goal about reconfigurability. My aim is designing a reconfigurable hardware structure which is able to dynamically support different subsequence lengths. Hence, I am going to design the control unit part with the support of *CPOG* model, and the datapath part representing it as a dataflow structure building an asynchronous circuit with methods and protocol described in Chapter 3.

7.2 Control-Unit design

As discussed in Chapter 4, designing the control-unit that manages the reconfigurability of the hardware structure may be done via *Conditional Partial Order*

Graph representation. Concerning this architecture for supporting the *ordinal pattern* operation, it is fairly straightforward to model it via *CPOG* since one needs just to replicate all the components and connect them as depicted on Figure 7.1.

Since the final goal is supporting the *ordinal pattern* encoding for subsequences from length four to ten, I created seven graphs via `Workcraft` modelling each kind of sequence. For readability's sake, I am not going to report all of them but two only in order to show to reader the slight differences between them. They are showed on Figures 7.2 and 7.3. As one might notice, it is exactly equal to the structure presented on previous Section, since each module is replicated here, and it mimics the original behaviour of module it is related to.

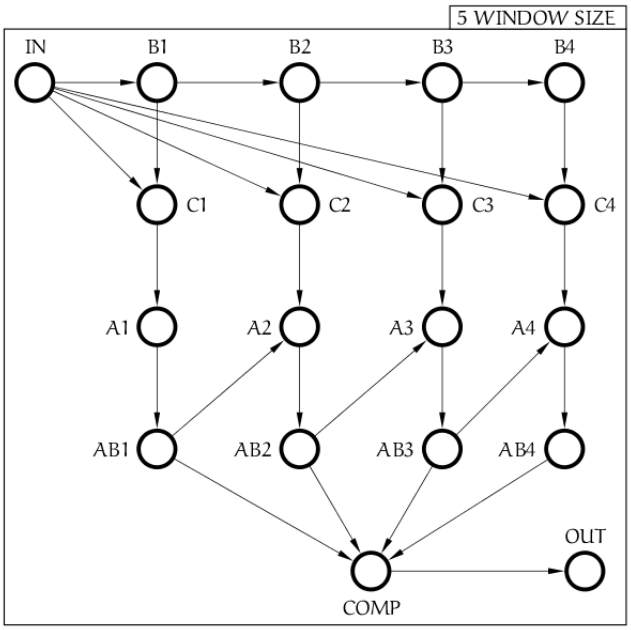


Figure 7.2: Ordinal pattern model for subsequences of length 5.

For instance, the comparators present at the second layer of the structure, on Figures 7.2 called Cx with x representing the index of the *Lehmer code* position, behave as described before being fed by buffers on top and feeding the adders at the bottom. Here reader should notice the high level of abstraction I am dealing with. In fact, none of the following parameters are defined such as the number of bits the comparators are composed by, or the length of the wires, the implementation the modules present are done. Designer is free from handling the modules as bare hardware, simplifying the design process of the whole control-unit. The model indeed does not depend on characteristics just mentioned, and might be designed no matter the low level implementation component used. For sake of understanding

let us define what the names assigned on the Figures stand for: IN corresponds to the first buffer which receives the time series in input. Bx are the buffers where the time series flow through, Cx correspond to comparators as already mentioned. Ax represent the adders , ABx the buffers which stores the current *Lehmer code* and finally the $COMP$ the compactor for concatenating all the numbers of the code into a single data stream. OUT models the symbolic output of the structure.

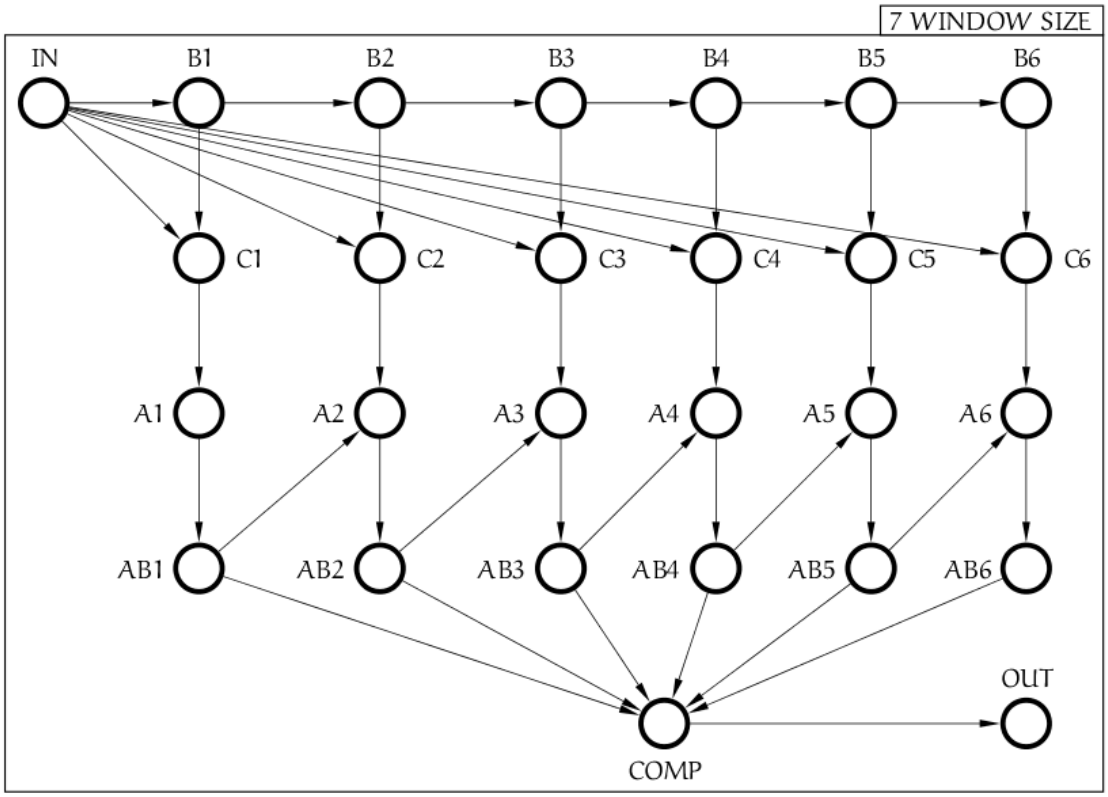


Figure 7.3: Ordinal pattern model for subsequences of length 7.

Additionally, I want to highlight the regularity of this architecture: observing Figure 7.3 and comparing it to the 5 length size version indeed, it is possible to see that the two structures are greatly similar. The only different part is that the 7 window size version contains a longer chain, composed so by more modules without any modifications on the links. It simplifies the whole design process.

The particularity of this system is the asynchronous pipeline I want it to be composed by. As described in Section 3.2, a self-timed pipeline does not need any control signals (REQ or ACK) to propagate each token because they should be propagated automatically by the pipeline itself. Therefore, the only control signals needed are the ones for switching the length of the pipeline and as a consequence, one

can use to build the control unit the option *CPOG* in the graphical user interface. It allows the program to synthesise the Boolean equations for the controller in such a way to minimise the area they will be synthesised on targeting the activation on each module belonging to *partial orders* rather than the request-acknowledgement signals, as described on Section 5.2.2.

Moreover, a designer needs to consider the search method the *GUI* has to use to find a good solution. Since the number of graphs is reasonably low, and due to the regularity of the structure, it may be convenient using the *Single-literal search*. It allows to increase the number of bits of the op-codes achieving a solution where all vertices of the graphs are associated to a single literal condition, the only drawback is the increased number of input bits for the controller.

Let us compare the two possible solutions: *Single-literal search* and the standard *Simulated annealing* technique. The compositional graph of the former one respectively is depicted on Figure 7.4.

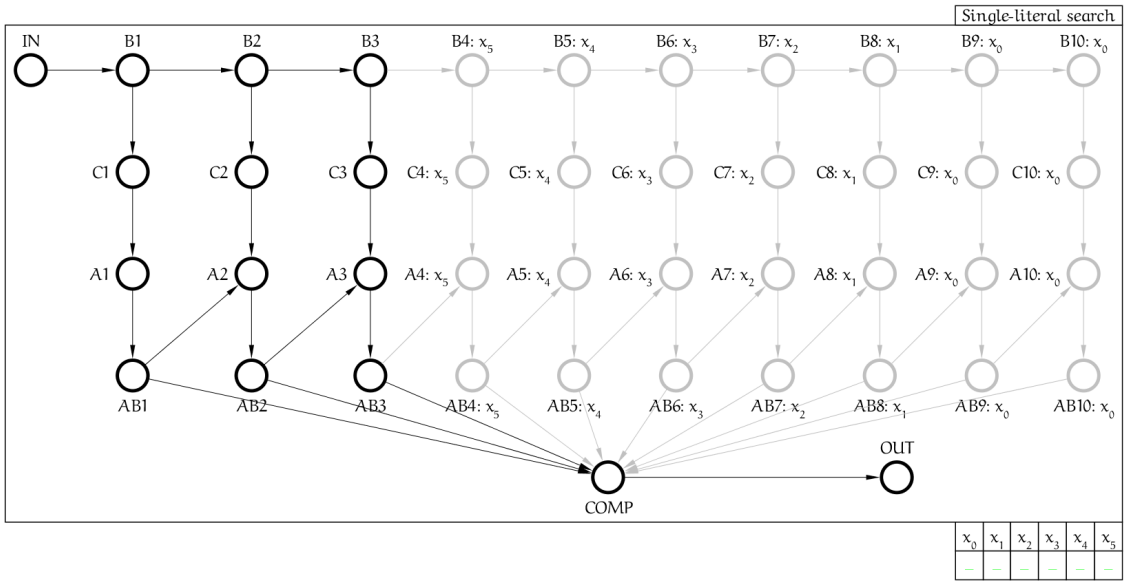


Figure 7.4: Compositional graph with Single-literal search of ordinal pattern operation structure.

The light grey nodes represent the conditional vertices that should be active if and only if the condition of that vertex is satisfied, while the black ones are always active since are present inside each *partial order*. The op-codes assignment is showed below.

Single-literal search op-codes assignment

4 WINDOW SIZE: 000000

```
5 WINDOW SIZE: 000001
6 WINDOW SIZE: 000011
7 WINDOW SIZE: 000111
8 WINDOW SIZE: 001111
9 WINDOW SIZE: 011111
10 WINDOW SIZE: 111111
```

As one might notice, the op-codes selected for the graphs are designed in such a way that each *partial order* can be associated to one single bit. For instance the *5 WINDOW* graph could be associated to fifth bit, if it is stuck at one it means that nodes $\{B4, C4, A4, AB4\}$ must be activated. The same for the *10 WINDOW SIZE* graph which has got the first bit associated to it x_0. The first graph only must not be associated to any bits since all the nodes present inside it are also present in the other graphs, therefore are always active and the actual condition associated to each vertex is and edge a logic `1`.

The most important characteristic is that the circuit here does not need any logic gates at all to be synthesised, unless some buffers to keep the signal strong for sustaining the fanout of the modules ahead. In order to compare this solution with the heuristic one, let us generate the compositional graph for what concerns the *Simulated annealing search*. It is depicted on Figure 7.5.

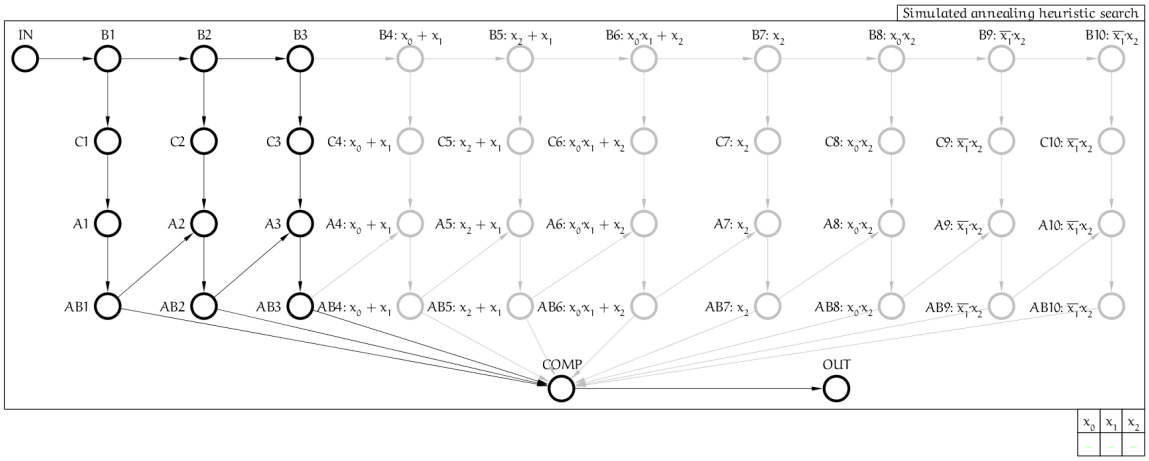


Figure 7.5: Compositional graph with Simulated annealing heuristic search of ordinal pattern operation structure.

As one might observe, each vertex of this compositional graph is associated to a more complex condition with respect to *Single-literal solution*. So, it is clearly inconvenient in terms of area since it would take a bigger area than before to be synthesised. The advantage is the reduced number of inputs it would need to switch

between these seven graphs, it requires half inputs than before. Below the op-codes assignment of this solution.

Simulated annealing heuristic search op-codes assignment

```
4 WINDOW SIZE: 000
5 WINDOW SIZE: 100
6 WINDOW SIZE: 010
7 WINDOW SIZE: 110
8 WINDOW SIZE: 011
9 WINDOW SIZE: 111
10 WINDOW SIZE: 101
```

Hence, after the considerations before the designer should choose which solution fits better to the design, one might need to consider the heuristic solution if number of pins are tightly constrained. Sometimes indeed number of pins constrained the area of the circuit, in particular if a small System on Chip is the target. On the other hand, if the number of input/output pins is not a problem the *Single-literal solution* is surely the best choice.

As I was aimed to show, designing control units for managing the reconfigurability of dataflow processor is extremely quick and easy with *Conditional Partial Order Graph*. The key feature which contributes to speed the whole development process up is the dynamical abstraction mechanism that can be applied via *CPOG*. As mentioned indeed, the model allows designer to tackle the problem through whichever level of abstraction one prefers: from hardware to Register Transfer Level.

So far, reconfigurable hardware structures have been often excluded from the VLSI (**V**ery **L**arge **S**cale **I**ntegration) industries panorama. Maybe also for the complexity which has always followed the design process. Even though a lot of work must still be done to simplify and automate the design process of such structures, this techniques might be exploited for creating working reconfigurable designs and it is certainly a good starting point.

7.3 Datapath design

Over this Section, the asynchronous hardware design of the architecture will be discussed, pinpointing on how to build a self-timed pipeline following the description given on Chapter 3 avoiding hazards. As pointed out before, one of the main problem afflicting self-timed circuitry is the presence of hazards, that are often caused by delay in the links or wrong specifications for what concerns asynchronous transactions.

They may end up on misbehaviours of the machine, or worse to external damages. In order to clarify better this terminology, on Figure 7.6 the whole process that might bring the machine to an external damage is showed.

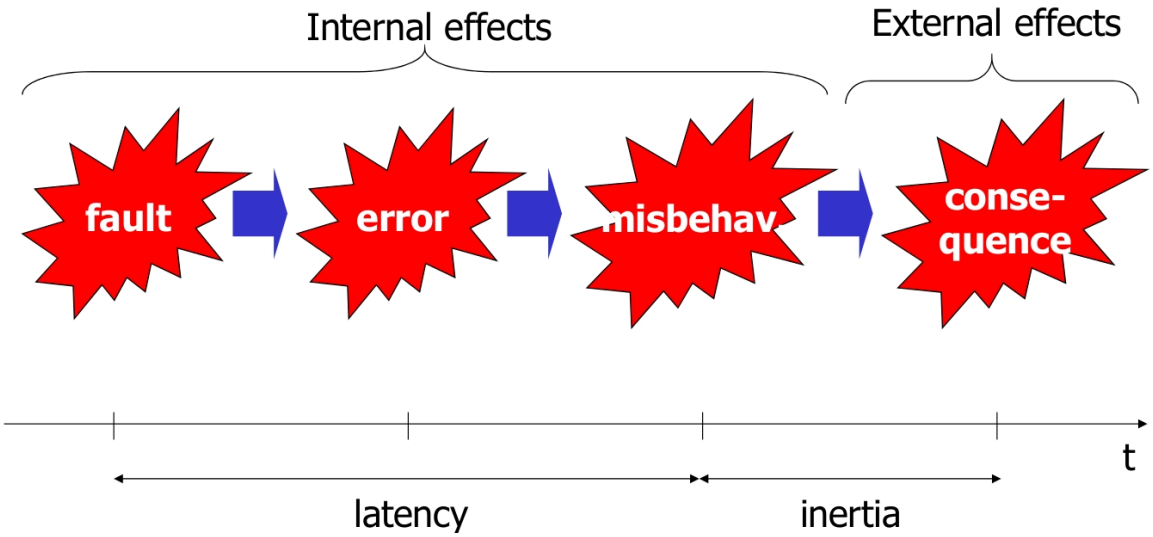


Figure 7.6: From fault to external damage([45], p.79).

As depicted on the Figure, *faults* are defined as the primary mistakes which might arise in a device. They are divided as defects or bugs, depending on whether the mistake stems by a hardware error or by a bug into the software, they cause the misbehaviour of the module which can be propagated to other components. They may be categorised into internal faults or external. Former ones are due to any defects into the hardware or software part, while second ones respectively stem by environmental strange behaviours, such as a strong increase of the external temperature, or the presence of a really noisy environment (like the space).

"A fault may manifest itself as a change in the state of the system; such a change is called *error*"([45], p.71). In turn, it may generate a *misbehaviour* which is defined as a behaviour of a product different than the expected one. It can be defined though different parameters, for instance it is categorised as static if the result is wrong, while dynamic whether the result is right but not the timing it comes up.

Finally, a misbehaviour may end up in an *external damage* (or *consequence*), which may be dangerous in the case the device is used for critical applications. Obviously, a designer should try to design a device avoiding the presence of any possible faults; nonetheless, due to the increase in the complexity of the design and due to variability that irremediably affects the manufacturing processes, bugs or defects may arise intrinsically affecting the dependability of the product.

Additionally, let us define the *latency* as the time a fault takes to manifest itself as a misbehaviour, while the *inertia* as the time between a misbehaviour and an external consequence. As pointed out by Matteo Sonza Reorda et al., one wants the latency to be as short as possible in order to detect the misbehaviours and remove them, on the other hand one wants the inertia to be as long as possible in order to have more time to remove the wrong behaviours of a product before it ends up with an external damage. Interested reader may find more information on [45].

In the context of synchronous design, sometimes faults are present between two consecutive clock cycles, but they are masked by the predefined timing set by the clock signal. It does not allow modules to capture the results in the middle of the computation, avoiding efficiently several faults and as a consequence misbehaviours which may arise. This technique does not work for asynchronous circuits due to the lack of the clock. In self-timed transactions indeed, in particular using the 4-phases dual-rail protocol as described in Section 3.2, the timing is set by the data-flow itself which automatically goes through all the modules without the need of a pre-set signal.

Even though from many points of view the clock lack can be considered an advantage, such as for saving of area/power. The hazards which may arise during the computational phases cannot be masked any more without a predefined clock signal, and therefore designers should seek other methods to avoid faults. Bundled-data asynchronous protocol attempts to solve this problem by matching the combinational modules with a delay depending on several parameters such as temperature, frequency, and so on. Nonetheless, somehow this technique may not fit well to all designs, in particular the ones subject to high variability.

4-phases dual-rail protocol attempts to overcome this problem by embedding the control-flow mechanism into the data-flow structure, obtaining fairly good results in terms of reliability. Nevertheless, sometimes it is not enough because signals may be affected by some delays which might cause the requests-acknowledgements to be arisen with wrong timing values, propagating the token either when it is not entirely ready, or when it is already been captured. As already mentioned in Chapter 3, in this research I am going to apply a recent method with combines *NCL-D* (reliable) and *NCL-X* (unreliable) gates to build sound hazards-free modules.

In order to build a reliable combinational component there is not an effective design-flow yet that, applied to a module, leads to a complete hazard-free component. Designers in fact need to customise it checking that none of the outputs raise up unless all the inputs are present, and that the whole token went already through the logic. The method I am going to use to design total reliable components involved on using *NCL-X* gates only, and then substituting some of them with *NCL-D* counterparts cleverly for meeting the main goal just mentioned.

Comparator design

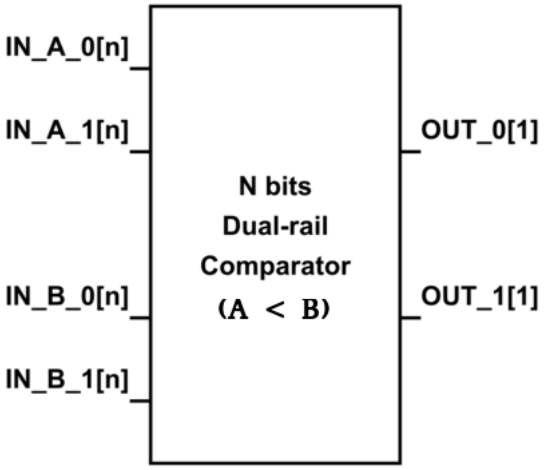


Figure 7.7: N bit dual rail comparator from an external point of view.

Let us take as case of study the only two combinational components present in the *ordinal pattern operation*: the comparator and the adder. As described on Section 7.1, comparator must be able to compare the newest element entering into the pipeline, with the current number of the time series stored by the buffers present on the top layer, and output a logic `1` if the condition is met, or a `0` on the other hand. On Figure 7.7 the schematic of the comparator is showed from a high level point of view. As one might observe, the inputs and the outputs are doubled since the module must work with the dual-rail protocol.

The implementation I chose to design the comparator is the one illustrated on Figure 7.8. For the purpose of *ordinal pattern* operation, just the $A < B$ output port is needed, the other ones might be cut off. The comparator has been designed as a subtractor where the output OUT_0 raises up if and only if the carry out signals a logic `1`, otherwise OUT_1 is activated. For more details of the theory of comparators refer to [46].

Thus, since the only part of the comparator one should care about for dealing with this implementation is the $CARRY_{OUT}$, I reproduced the logic function of the it starting from an adder/subtractor component and a version of the function with *NCL-X* gates only has been implemented. The Boolean function is listed below and it takes into account also the $CARRY\_IN$ which must be propagated through the chain of 1 bit carry adders:

$$CARRY_{OUT} = A \cdot B + A \cdot CARRY_{IN} + B \cdot CARRY_{IN}$$

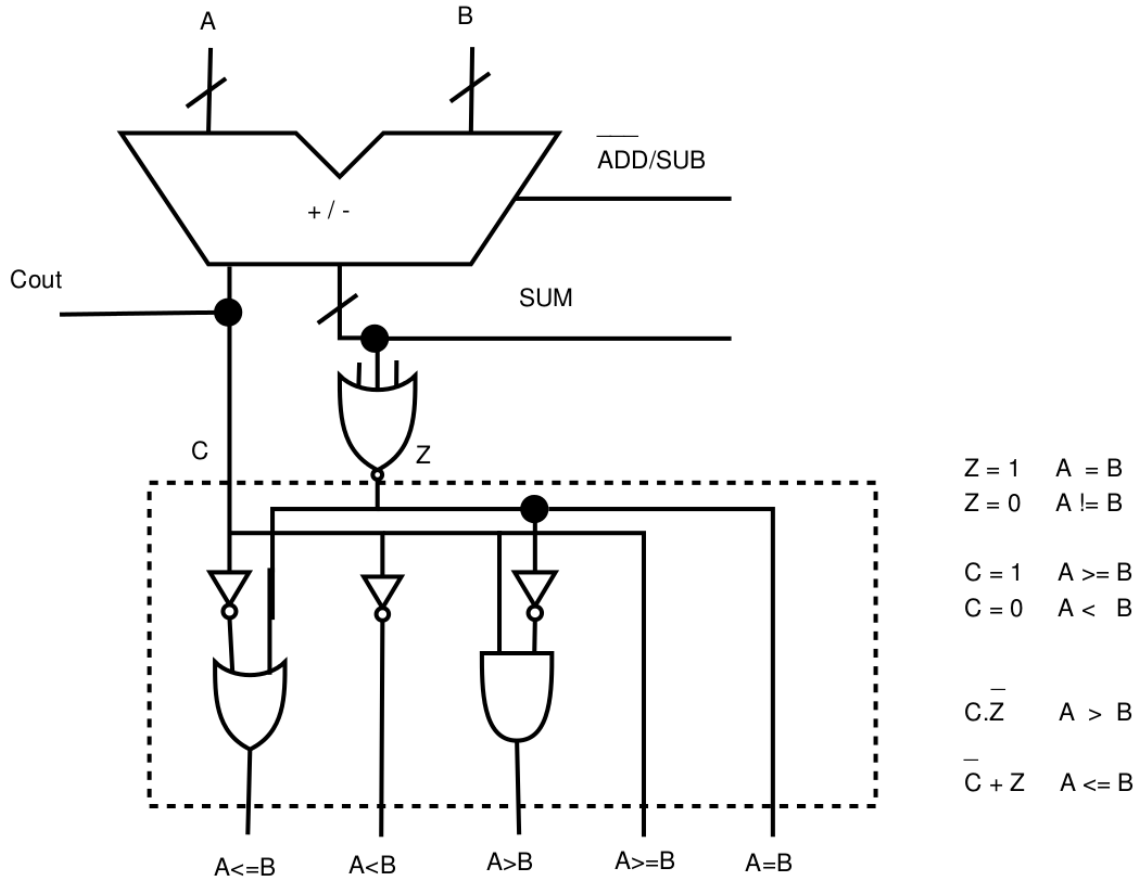


Figure 7.8: Schematic of a comparator.([46], Figure 4.16)

While the 1-bit Carry adder is represented on Figure 7.9. The gate-level representation showed on the Figure just mentioned was obtained via the *Digital circuit* plugin under `Workcraft`, which allows not only to design a digital circuit, but also to test it running a basic simulation of the signals propagation. It could be very useful when the propagations of the signals represent the main concern for avoiding hazards.

The structure represented on Figure 7.9 is extremely unreliable. In fact, just two input signals out of three are enough to propagate the result at the output, for instance, if the inputs $B1$ and $CIN1$ are active, the token will be propagated through the bottom `AND` gate, and as a consequence into the final `OR` gate reaching instantaneously the output. The same happens if $A1$ and $B1$ are active together, signal will be propagated through the top `AND` port and afterwards through the two `OR` gates. In order to stop the flow of the token if some inputs are delayed, I implemented the same structure under analysis substituting some *NCL-X* gates with *NCL-D* counterparts which, even tough the cost in terms of area, make the

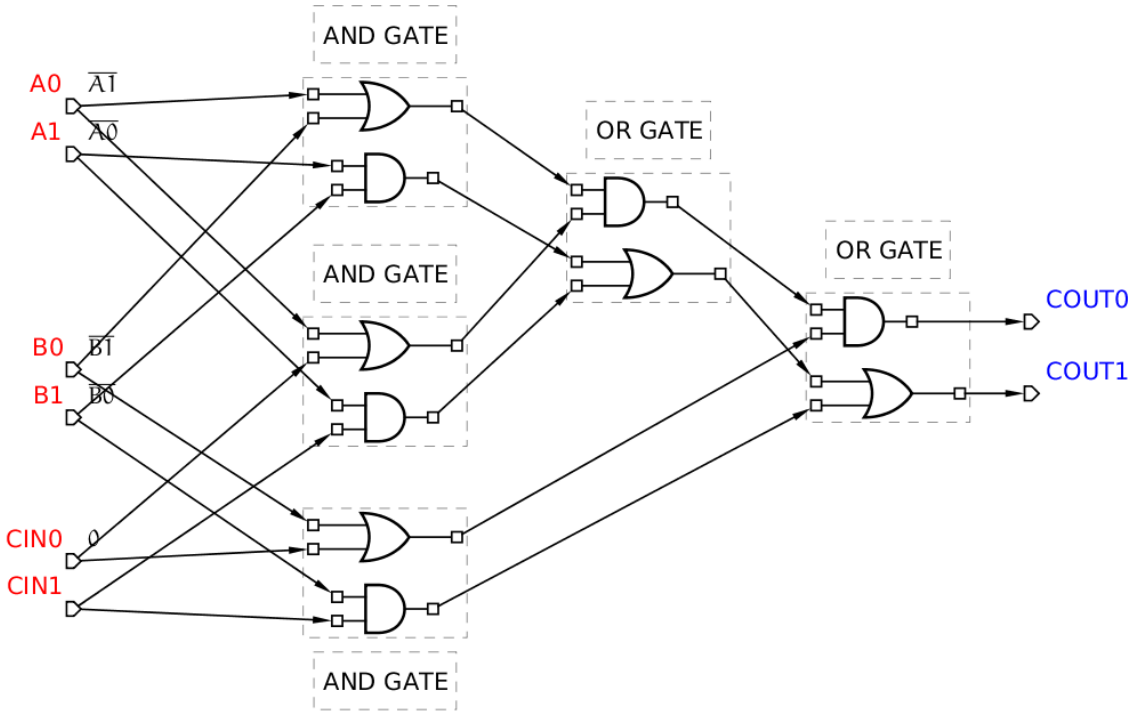


Figure 7.9: 1-bit dual-rail carry adder implemented with *NCL-X* gates

whole structure totally reliable. The structure is depicted on Figure 7.10.

As the reader might observe, four out of five gates have been replaced in this implementation to make the structure hazards-free. It is because of the high unreliability of the gates, none of them in fact are able to stop the output flow and most of them have been replaced in order not to propagate the output if at least three inputs are not present. With the implementation that combines *NCL-X* and *NCL-D* in fact, if two of the inputs are active, the output will never arise but it will be waiting for the other input to come up. This is the best one can do for what concerns this structure.

Adder design

For what concerns the adder, let us first think about the characteristics of the external hardware I can exploit to simplify the design. As showed before, the adder is in charge of adding the previous *Lehmer code* number to 1 if and only if the current number of the time series is greater than the new number just entered into the chain. It means that one could remove the second input of a standard adder exploiting the $CARRY_{IN}$ to perform such operation simplifying a lot the internal structure of the

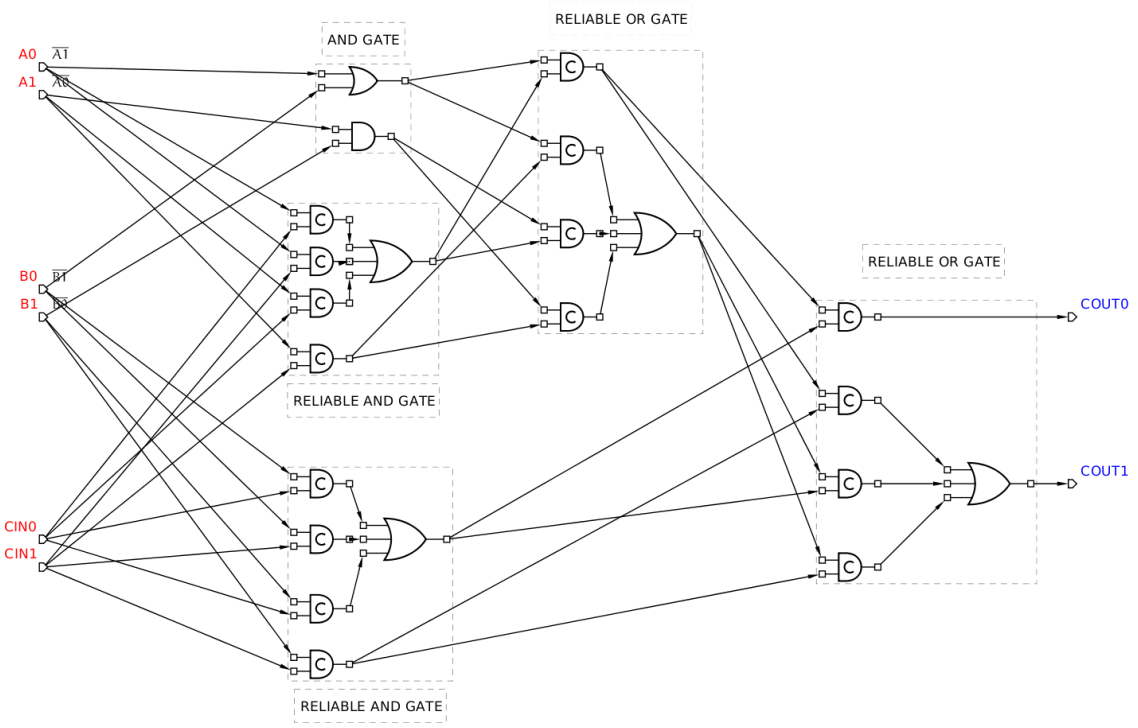


Figure 7.10: 1-bit dual-rail carry adder implemented with *NCL-D* and *NCL-X* gates

combinational component. Indeed the truth table comes up removing such operation is depicted on Table 7.1, and for readability's sake on the left the schematic view of the component is showed. It is not represented in dual-rail because the logic function is independent on the protocol used for transactions.

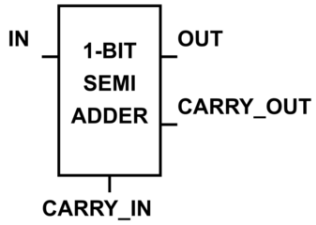


Figure 7.11: Schematic.

IN	CARRY_IN	OUT	CARRY_OUT
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

Table 7.1: Truth table.

The Boolean functions come out from the truth table are depicted below:

$$OUT = IN \oplus CARRY_{IN}$$

$$CARRY_{OUT} = IN \cdot CARRY_{IN}$$

They are really simple and require just two gates to be implemented. Moreover, by

analysing the structure of the *NCL-X* xor gate (showed on Figure 7.12), one might observe that it is reliable and does not need the substitute *NCL-D*.

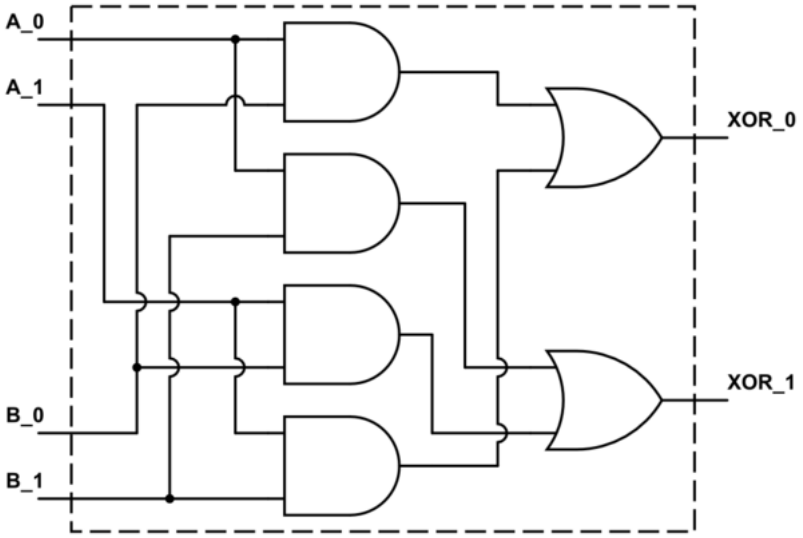


Figure 7.12: *NCL-X* dual-rail XOR gate.

Indeed, the first layer of AND gates, cut off all the signals unless both of the inputs are present at the same time. For example, if *A_0* is active but none of the inputs belonging to B input are active, the output does not make any transitions. The only way to trigger a transition at the outputs is that at least one of the following combination is present: $\{A_0B_0, A_0B_1, A_1B_0, A_1B_1\}$ which is right the conditions we want the circuit to address, conditions as $\{A_0A_1, B_0B_1\}$ are invalid and cannot be present inside the circuit. Hence, it is possible to implement a totally reliable 1-bit semi-adder as showed on Figure 7.13. In order to build a n-bit adder one just needs to connect n number of dual-rail semi-adders in cascade.

Ordinal pattern encoding hardware design

Once that I have discussed how to design and build all the components present in this structure: asynchronous registers on Chapter 3 and combinational reliable dual-rails modules in the previous Section. I am going to illustrate the implementation of the whole structure, focusing on the 4-phase protocol implementation, which contributes to modify the structure according to the protocol's needs.

Before showing the structure, it is worth mentioning that most of the dataflow project was designed and tested with the support of *Dataflow plugin* under `Workcraft`. It allows to build dataflow structures via predefined instruments typically used with

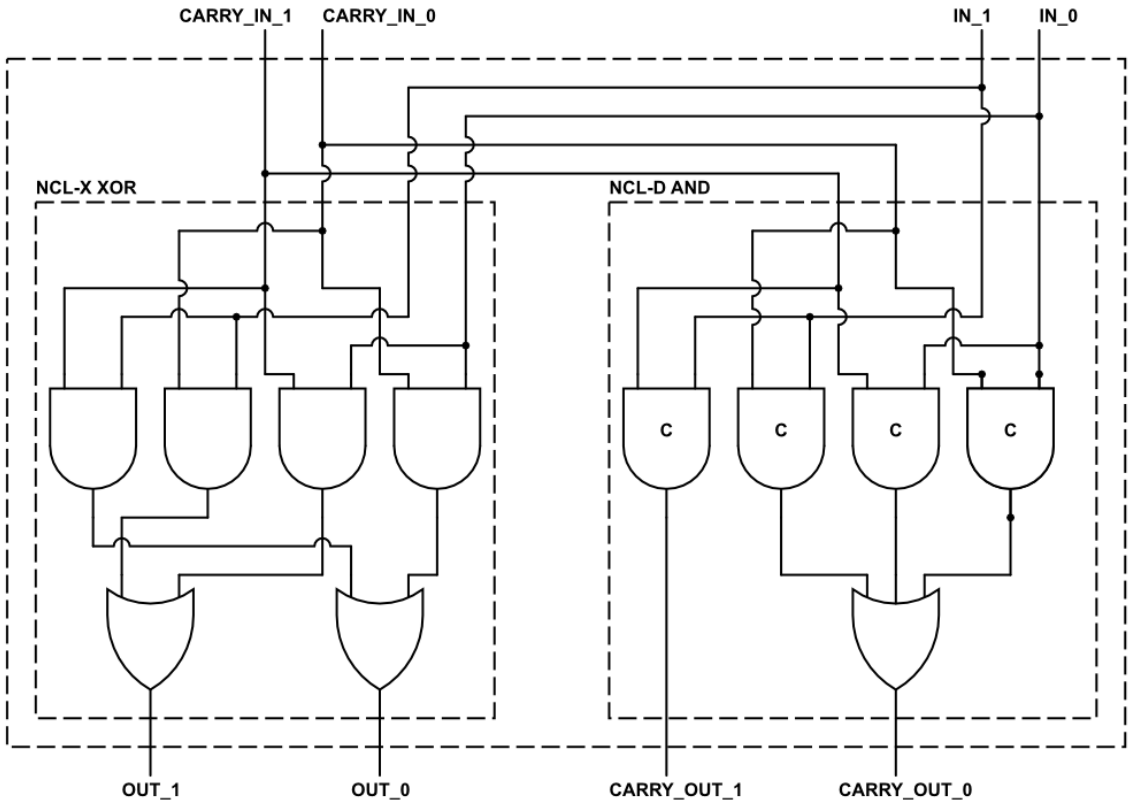


Figure 7.13: 1-bit reliable dual-rail semi-adder implementation.

this technique, such as registers, combinational component, wiring and push/pop blocks for managing the reconfigurability of the flow. Reader may find more information on [37]. Additionally, it supports the propagation of tokens according to 4-phase protocol tailoring well to this implementation.

Let us start presenting and describing the project designed under `Workcraft`, it is depicted on Figure 7.15. In such representation the blocks present internally mimic the behaviour of real modules as described in Section 7.1. On Figure 7.14 the legend is showed in order to simplify the readability of the scheme. The asynchronous register is implemented as described on Figure 3.10 for a variable number of bits, it contains a completion detection module next to it able to detect if the module has captured correctly the signal. The only difference with respect to the register with a dot inside it is that the latter one stores the data, so is not able to receive another token in input. That is the main rule of a 4-phase transaction indeed: *two data-token must always be separated by a register*, otherwise something wrong is happening into the circuit.

In order to address this constraint and to propagate the data through the pipeline

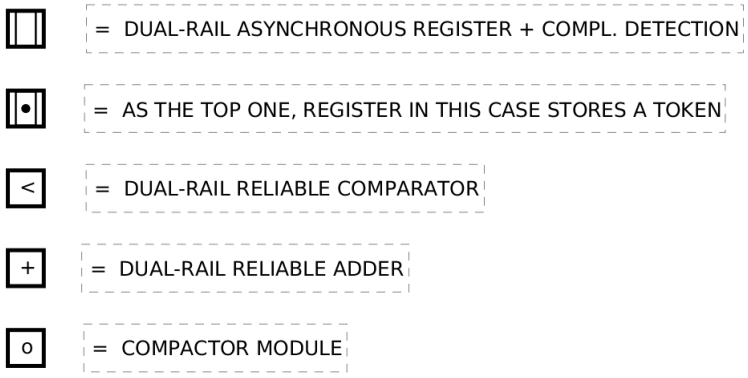


Figure 7.14: Legend of blocks present on Figure 7.15.

accordingly, I doubled each register implementing a *Master* who is in charge of capturing the data and propagate it to the *Slave* without any combinational logic between them. So that to give the possibility to the *Master* to be free to capture a new data as soon as it is available on the bus, and to the *Slave* to capture the data without delay and put token available on the bus for the computation. In this way all the combinational parts of the circuit can be executed at the same time as in a pipeline, avoiding the problem of having half modules which can be triggered and half that cannot. The other blocks have been already described in this Chapter over the previous Sections.

For what concerns the actual structure, first of all reader should notice that the total length of the chain involved corresponds to ten stages, as the maximum window size designed under *Conditional Partial Order Graph* model. This is why in order to support subsequences from length 4 to 10, all ten stages are needed and some of them must be disabled to support shorter subsequences via some reconfigurable management forms. All the registers which refer to a subsequence greater than the minimum one in fact, should have an input needed to switch itself off from the control unit that manages the reconfigurability; additionally another block should be inserted before the compactor in order to tell to the final register that the index which has been disabled from the *Lehmer code* is present, in such a way not to wait forever for a disabled token. It can be achieved by using *Push* and *Pop* registers which are described on [37]. They are omitted from this representation for readability's sake and because they will be inserted directly in the hardware description design.

Next step, after designed both the control unit and the datapath, is to connect them together according to the specification mentioned before. This part will not be covered as well as the hardware coding part done in *VHDL*(VHSIC (= Very High

Speed Integrated Circuits) Hardware Description Language). It is worth mentioning that due to the extremely good matching between this model designed with *Dataflow plugin* under `Workcraft` and the original circuit implemented in *VHDL*, it might be possible to automate the coding part of the hardware description language, shortening the design and as a consequence the *Time to market*, which is defined as the time the company takes to push in the market its product.

In next final Chapter some considerations will be performed to close this research.

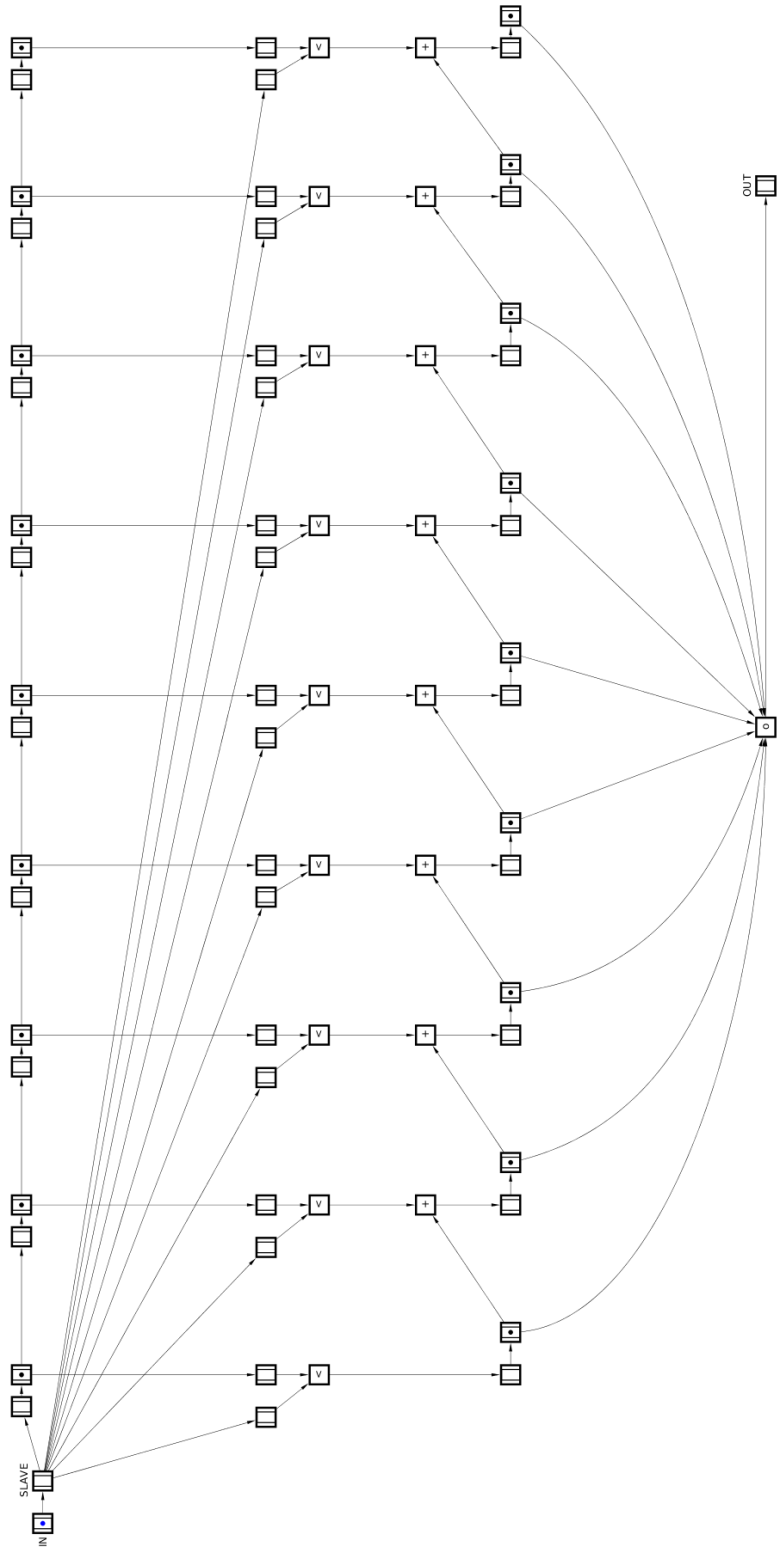


Figure 7.15: *Ordinal pattern encoding* hardware structure design under Workcraft.

Chapter 8

Conclusions

This thesis presented a new design-flow composed by different elements, which can be applied for designing reconfigurable data-flow processors. It is meant to be used with various models and might simplify and shorten the time to market of many products used in several applications.

In this Chapter I want to summarise all the contributions my dissertation proposed, as well as the future research directions which might follow this one, expanding not only the world of hardware modelling, but also the one of reconfigurable architectures and data-flow machines, which are not extensively used in the market even due to the complexity which drives the designing process. Nowadays indeed, due to the high competition present in the electronics branch of the market, companies try to push their own products towards the manufacturing phase quickly in order to increase their earnings as much as possible.

Even the smallest delay on time-to-market may means an extremely huge loss of money, as represented on Figure 8.1 indeed, where the areas of the triangles stand for the revenues a company could potentially earn. That is why a company privileges already working, tested and accomplished tools, neglecting the new ones present on research fields which might introduce new design techniques.

One of the aim of this dissertation is to push *Conditional Partial Order Graph* model forward to industries, in order to be used and applied to real concrete applications going into the marketplace. It is a really stable, powerful and compact model able to really help designing process of whichever module or processor. It would be an invaluable error neglecting it.

In summary, this Chapter is structured as following: Section 8.1 illustrates the main contributions such dissertation aims at bringing to research, while Section 8.2 describes some future research directions which might be inspected to extend not only the *Conditional partial order graph* model, but all the fields surrounding it.

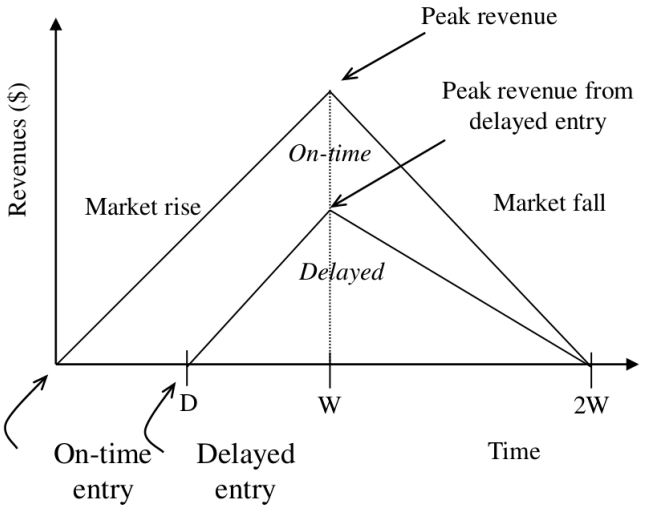


Figure 8.1: Losses due to delayed market entry([47], p.13).

8.1 Main contributions

Main concern of this dissertation is to develop a tool supporting the CPOG-driven design process of reconfigurable dataflow processors. As showed throughout the pages in fact, there was a gap of instruments able to support the automation in the composition, synthesis and mapping process of systems composed by greatly different behaviours.

The contributions this thesis aims to bring on the research fields span from automating the composition of *Partial Order* graphs to the creation of a *Graphical user interface*, allowing the designer to set constraints in the op-codes meeting the need of flexibility. As showed in Chapter 4 indeed, the previous plugin for the encoding process did not allow designers the possibility to choose neither the algorithm via the op-codes were associated to the graphs, nor the number of bits the op-codes should be composed by or the possibility to fix some bits for reserved function purposes. Following are showed the main contributions of such dissertation.

Algorithms for encoding process. The process of encoding *Conditional Partial Order Graphs* now can be chosen by the designer. Who has now the possibility to benefit from various algorithms for seeking the best solution which fits well to the design. As described in Chapter 4 indeed, designer might use either heuristic search algorithms exploiting different and customisable cost function, randomly-driven rationales or recursive ones. Moreover, it has been demonstrated that the cost function presented in Chapter 4 may be extremely effective for seeking a good solution in terms of search-time and area.

Op-codes flexibility. Additionally, designers could now benefit from the high flexibility in the op-codes association. In fact, one can fix how many bits each *Partial Order* should be composed by, set one or more bits for each op-codes in the case of reserved bits. The tool developed will automatically check the feasibility of such encoding informing the developer whether any wrong specifications would occur.

Graphical User Interface. A friendly *GUI* has been developed in the *CPOG plugin* under `Workcraft` in order to simplify the encoding process driving the designer via a highly flexible and simple interface. It was not present before, now it is possible to benefits from it.

Synthesis and Mapping process. The automation process is able not only to synthesise all the Boolean equations of the controller automatically, but also to map them by using a gate library set manually by the designer, showing final results as number of gates used and area occupied by the circuit in order to have concrete information to exploit for the whole project.

Real application examples. Since *Conditional Partial Order graphs* tailor well to asynchronous circuits as well as to *Instruction Set Architecture* designs, such dissertation demonstrated with two real examples how this *CPOG*-driven design-flow might be applied for a real *ISA* such as the `Armv6-M` in Chapter 6, or to a reconfigurable asynchronous data-flow processor on Chapter 7 for supporting *Ordinal analysis*.

Such design-flow has been demonstrated to be successfully applied on real projects, designers may now benefit from it throughout the development phases exploiting the tool generated during this research. Over the next Section the areas of future research will be presented and discussed.

8.2 Future research directions

Since the main purposes of the thesis have been presented, in this Section I am going to show the main research directions which may follow this work, pinpointing on the topics this thesis handled.

One of the main research direction this thesis may be followed by, is the seeking of a new cost function in order to improve the search of an optimal encoding even more. As indeed I have discussed in Chapter 4, and as one might observe on Figure 6.6 where two heuristic techniques are applied to a concrete instance, the cost function so far is affected by a certain degree of error, which may also affect the result in terms of area of the encoding found.
Indeed, as depicted on Figure 8.2, where the Δ of the error is represented, the cost function is not able to linearly separate the various encodings. Many of them end up with a same weight of the function though they can be synthesised with very different results in terms of area. Thus, the need of a better cost function, maybe

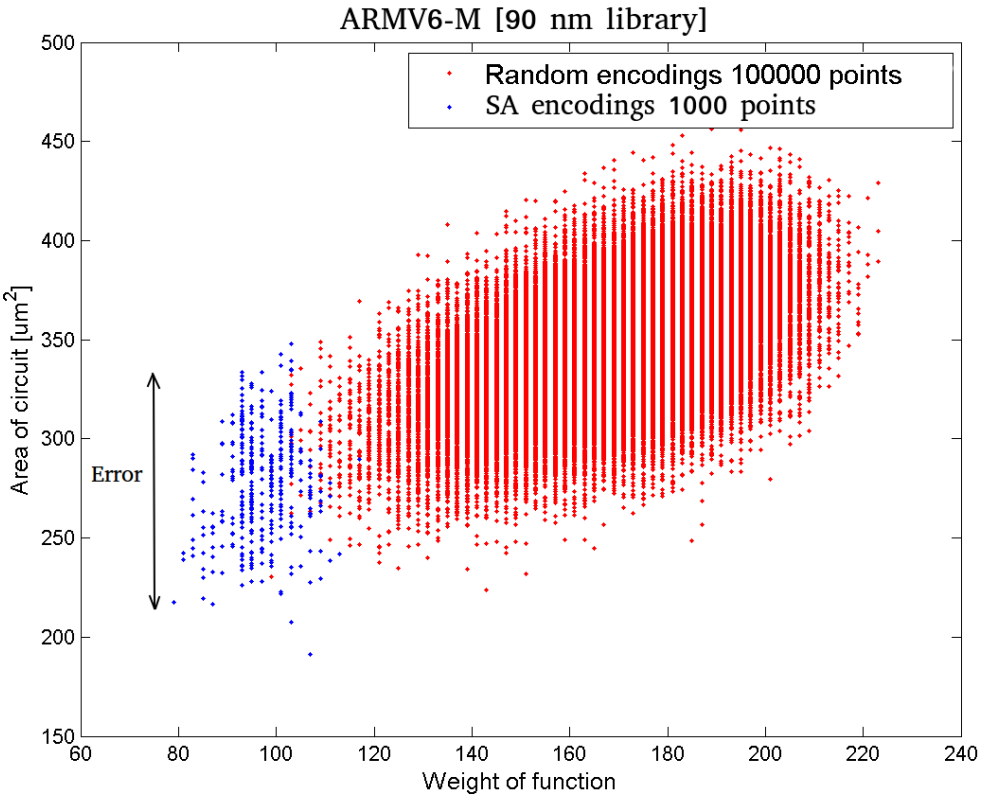


Figure 8.2: Error showed due to cost function used.

taking into account also the op-codes not used inside the solution, that are the ones which are not associated to any *Partial Orders* when the number of op-codes is higher than the graphs. So far in fact, cost function is computed by using the elements available for the solution which are actually used, neglecting the other ones. Maybe by taking them into account, a more linear solution might be obtained.

Another research branch that might be followed regards the automation of hardware description language starting from the *dataflow* representation under `Workcraft`. As discussed in Chapter 7 indeed, and in particular as can be observed on Figure 7.15: the *dataflow* representation gives an greatly good starting point for translating the model into hardware description language. Dataflow structure indeed, tailors perfectly well to an asynchronous structure exploiting 4-phase protocol for transactions. Hence, there are plenty of possibilities researchers have to simplify and to speed the design process up, such as the development of a low level hardware layer for each of the block present in the plugin in order to let designers choose how the predefined modules present in the structures could be implemented, maybe in single

and dual-rail fashion either. I guess that designers will absolutely benefit from it.

Furthermore, the techniques applied for the design in Chapter 7, can be experienced with the 2-phase protocol which has the capability not to return to zero at each cycle, doubling the throughput of the whole structure as well as saving a big amount of power due to the reduced swings of the signals for transactions. More information about this protocol may be found on [33], dataflow plugin under `Workcraft` may be thoroughly enhanced supporting the propagation of the tokens also via this protocol.

Additionally, synthesis of final controller starting from *Conditional Partial Order Graphs* might be driven by different cost parameters such as the delay, or the power consumption. It would not be too difficult to implement this feature since developers should modify the interface with `Abc` tool allowing the user to choose which parameters the circuit should be optimised for.

All the previous ideas might be potentially inspected and implemented with a deep research phase. Furthermore, this dissertation aims at pushing forward the reconfigurable development in the context of asynchronous design. It might be applied to particular types of applications that may benefit from the lack of the clock signal, the power consumption reduction and high flexibility and reliability of the transactions where the control-flow is embedded in the data-flow.

For instance, as Danil Sokolov et al. demonstrated on [48], asynchronous circuits fit well to analogue electronic power management due to its capability of capturing small swings of signals without requiring high clock frequencies in the input interface. A further research direction might be the creation of formal ways to formalise analogue specifications in order to represent an asynchronous circuit able to satisfy the analogue requirements. Hence, both specification and verification of the circuit compliance is subject of future work.

Appendix A

Partial orders modelling Armv6-M

In the Appendix, the whole *Conditional Partial Order Graph* representation modelling the *Armv6-m* Instruction Set Architecture is showed. As already briefly described over Chapter 6, the entire model is composed by eleven *Partial orders* which mimic the behaviour of several instructions embedded into *Arm* `Cortex-M0+` processor. For more information about the model itself, readers can refer to [29], where this representation is the main topic.

Over the next pages, each graph representing a particular instructions class will be showed. The descriptions will be structured over four parts: a *Figure* which shows the actual graph composed by the standard elements such as nodes and arcs; a list of all the instructions represented by the *Partial order*; a brief *description* of the current graph, pinpointing on how it is structured and an explanation of the way it works; and finally some examples of instructions as described by the official handbook [40].

Due to the complexity of this Instruction Set Architecture embedded on a real processor on the market, this model is not intended to be a perfect representation of the *ISA*. Therefore, a fair comparison between the decoding logic of the *Arm* `Cortex-M0+` and the module comes up with this model cannot be carry out because of the wide differences between the two structures. *CPOG*-driven model indeed needs further graphs either for representing the instructions left (not represented yet) and for representing the behaviour at a lower level of abstraction, for instance for all the operations a single component, as an ALU (Arithmetic Logic Unit) is able to perform. As already mentioned before, the goal of this representation is to have a good model to test the new algorithms and cost function.
Following the the model handled in Chapter 6.

Instructions class 1

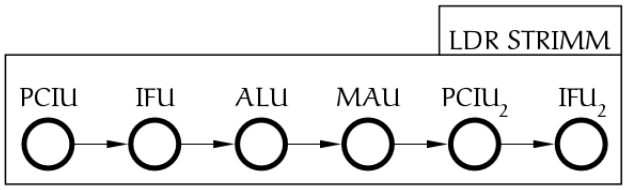


Figure A.1: Memory instructions (Immediate offset addressing mode)

Instructions modelled by the graph: LDR (imm.), LDR (literal), LDRB (imm.), LDRH (imm.), STR (imm.), STRB (imm.) STRH (imm.).

Description: Instructions related to this *Partial order graph* are related to LOAD-/STORE structure architecture. Through *Store* command programmer can save an operand into a position of the memory by using immediate offset addressing mode. While trough *Load* instruction, one can load into a register close to processor a value from the memory, addressing methods one could use are the same as for Store command.

Examples:

```
LDR <Rt>, [<Rn>{,#<imm5>}]
LDR <Rt>,[SP{,#<imm8>}]
LDRB <Rt>,[<Rn>{,#<imm5>}]
STRB <Rt>,[<Rn>,#<imm5>]
```

Instructions class 2

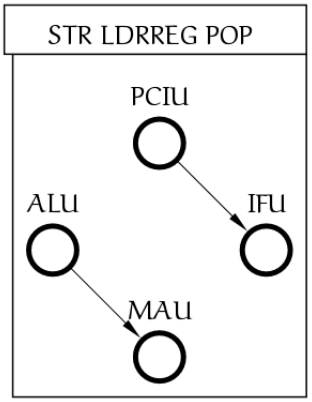


Figure A.2: Memory instructions (Register offset addressing mode)

Instructions modelled by the graph: LDR (reg.), LDRB (reg.), LDRH (reg.), LDRSB (reg.), LDRSH (reg.), STR (reg.), STRB (reg.), STRH (reg.), POP.

Description: This class models the same operations that could be done under the first class, but with a different addressing mode system. Here indeed, one could select a memory address for storing/loading a value by exploiting register offset addressing mode.

The flow of events this instruction class refers to is totally different, since the immediate operand should not be fetched and evaluated. The addition in order to evaluate the actual address and the extraction from memory could be done in parallel with the next instruction fetching. On the bottom two examples of such instructions. Pop instruction modelled by this *Partial order* cannot move data from the stack to program counter, this behaviour is modelled by the sixth Instructions class

Examples:

LDR <Rt>,[<Rn>,<Rm>] => Rd = Mem[Rn + Rm]
LDRH <Rt>,[<Rn>,<Rm>]
LDRB <Rt>,[<Rn>,<Rm>]
STR <Rt>,[<Rn>,<Rm>] => Mem[Rn + Rm] = Rd
STRH <Rt>,[<Rn>,<Rm>]
STRB <Rt>,[<Rn>,<Rm>]

Instructions class 3

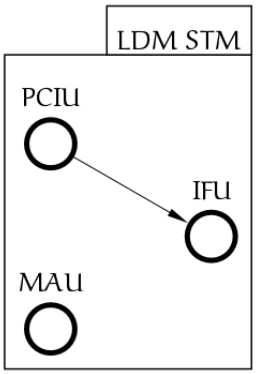


Figure A.3: Memory instructions (Bunch registers transferring)

Instructions modelled by the graph: LDM, LDMIA, LDMFD, PUSH, STM, STMIA, STMEA.

Description: This instructions class always refers to memory as the two *Partial orders* before. The difference here is that a bunch of registers are transferred, so since programmer can specify registers to transfer via a 8 bits special registers, alu event to compute the final address is not needed anymore. The address of the bunch of register to transfer is already present in the second operand.

According to description given into Technical Manual of the processor, push instruction is integrated inside this class of instructions, since the way through which the register to push on stack is addressed is highlighted by the same 8 bits word.

Examples:

```
PUSH <registers>
STM <Rn>!,<registers>
LDM <Rn>!,<registers> #<Rn> not included in <registers>
LDM <Rn>,<registers> #<Rn> included in <registers>
```

Instructions class 4

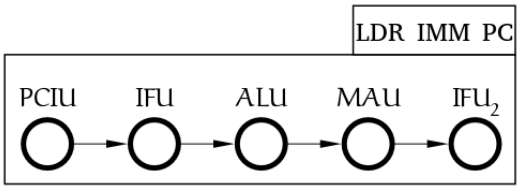


Figure A.4: Load instruction Immediate addressing on PC

Instructions modelled by the graph: LDR (imm.).

Description: This instruction allows to load an address from the memory to the *Program Counter* register. It could be useful when designer deals with code which may contain some predefined functions addresses stored in memory. The flow of the operations described by the *Partial Order* is pretty straight forward, since memory address where instruction takes the data from is an *immediate*, it must be fetched (*PCIU* followed by *IFU*). In the case the address represents an offset, ALU must be used to compute the actual address and afterwards the data may be taken from the memory via the *Memory Address Unit* and the result is placed on Program Counter. Finally, next instruction is fetched, addressed by current PC address.

Instructions class 5

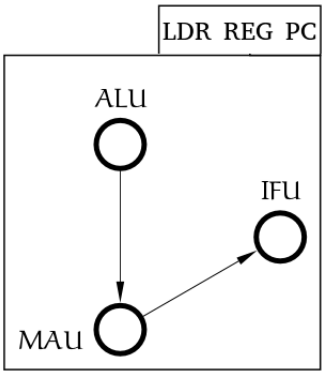


Figure A.5: Load instruction register addressing on PC

Instructions modelled by the graph: LDR (reg).

Description: This graph models the same behaviour described for *Partial Order* on Figure A.4. It differs from the previous one because of the addressing mechanism, in this case indeed the address of data to load is contained in a processor register, therefore PCIU and IFU beginning nodes are not needed any more. Following behaviour is the same as before.

Instructions class 6

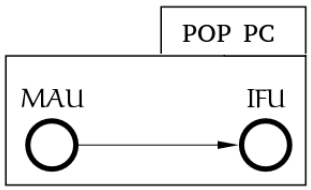


Figure A.6: POP instruction

Instructions modelled by the graph: POP

Description: The *Partial order* showed on Figure A.6 represents one single instruction only, that is the POP instruction. It is often used inside whichever program, and it is in charge of moving the data from the Stack into the *Program Counter*, in order to perform a branch. As described on [29], the instruction field of this operation uses an extra bit to specify whether the data should be moved in the program counter, otherwise the *Partial order* in charge of performing a regular pop instruction is the second one.

Since the base address of the register is used for addressing, alu module is not used at all and therefore the graph of this group results to be very simple. MAU is used to move data between memory and register and IFU node models the next instruction to be fetched.

Examples:

POP <registers>

Instructions class 7

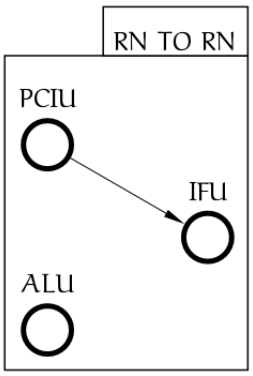


Figure A.7: Arithmetical, logic and data copy instructions (register addressing)

Instructions modelled by the graph: ADC (reg.), ADD (SP + reg.), AND (reg.), ADD (reg.) ASR (reg.), BIC (reg.), CMN (reg.), CMP (reg.), EOR (reg.), LSL (reg.), LSR (reg.), MOV (reg.), MUL, MVN (reg.), ORR (reg.), SXTB, SXTH, TST, UXTB, UXTH.

Description: As one might observe from the list of operations covered by such graph, it represents the execution of arithmetic, logic and data movement operations where the register addressing mode is used only. Indeed as one might understand by the events in the graph: *ALU* operation is executed in parallel with the fetching of the new instruction (*PCIU* - *IFU*), this is because the operands do not need to be fetched since they are already available in the registers of the processor. Below some examples are depicted related to the operations listed above.

Examples:

ADD <Rdn>,<Rm>
CMP <Rn>,<Rm> #<Rn> and <Rm> both from R0−R7
ANDS <Rdn>,<Rm>
MOV <Rd>,<Rm>

Instructions class 8

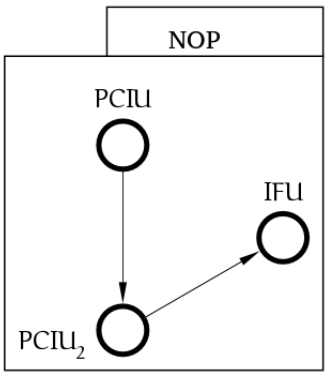


Figure A.8: Nop instruction

Instructions modelled by the graph: NOP.

Description: This *Partial Order* models the NOP instruction only. The behaviour of the graph may be interpreted as follows: since the current instruction requires to perform no operations at all, Program Counter is incremented again ($PCIU_2$) in order to fetch a new instruction into the Instruction Register (IFU).

Examples:

NOP

Instructions class 9

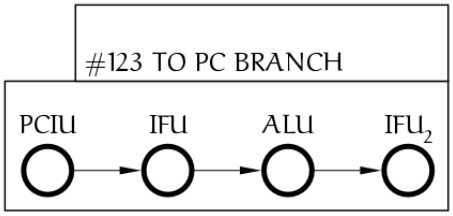


Figure A.9: Unconditional branch instruction

Instructions modelled by the graph: B.

Description: This *Partial Order* models the behaviour of the unconditional branch instruction. It jumps to the address labelled by the developer of the software. Once the Program counter is updated with the offset present inside the instruction computed by the compiler (*PCIU* followed by *IFU*), the *ALU* module is used to add/subtract to the *Program Counter* the offset the instruction must jump on and afterwards, the *Instruction Register* is updated with the new instruction to execute.

Examples:

```
B <label>
```

Instructions class 10

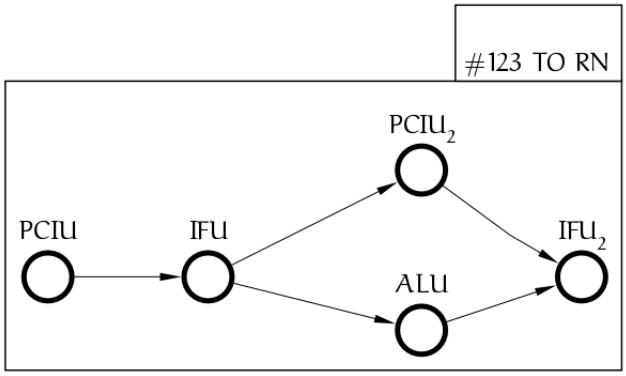


Figure A.10: Instructions with immediate addressing mode

Instructions modelled by the graph: ADD (imm.), ADD (SP + imm.), ADR, ASR (imm.), CMP (imm.), LSL (imm.), LSR (imm.), MOV (imm.), RSB (imm.), SUB (imm.), SUB (SP - imm.).

Description: This *Partial Order* embeds a high number of instructions internally, all of them share same addressing mode and go through the same modules. The instructions listed above requires to have an immediate operand, it requires to be fetched and written into the *Instruction Register*. Then the ALU operation can be performed in parallel with respect to the second update of the *Program Counter* ($PCIU_2$) in order to allow the next instruction to be ready for next fetch phase (IFU_2).

Examples:

```
ADDS <Rd>,<Rn>,#<imm3> # immediate on 3 bits
ADDS <Rdn>,#<imm8> # immediate on 8 bits
SUBS <Rd>,<Rn>,#<imm3> # S indicates that instruction updates the flag
CMP <Rn>,#<imm8>
LSRS <Rd>,<Rm>,#<imm5>
```

Instructions class 11

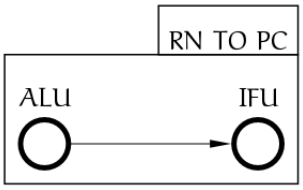


Figure A.11: Branch instructions

Instructions modelled by the graph: BLX (reg.), BX.

Description: This graph models two slightly different branch instructions: BLX (Branch with Link and Exchange) and BX (Branch and Exchange). Both of them use register addressing mode, so no constants need to be fetched before the execution of the operations. The former instruction differs by the latter one because it saved the current *Program Counter* address into the *Link Register* (LR) before branching. It could be useful in the case of a function.

ALU module provides all the functions just described, it is able to move the current address to LR or to move the address stored in the register pointed by the instruction into the Program counter in order to jump. Afterwards *IFU* provides the fetching mechanism for next instruction. Below a couple of example of the usage of these two instructions are showed, as provided by the *Arm* handbook.

Examples:

```
BLX <Rm> # RM stores the address to jump
BX <Rm>
```

Bibliography

[1] Moore Gordon E. *"Cramming more components onto integrated circuits"*, 1965.

[2] IDC, `www.statista.com/topics/840/smartphones/chart/1011/connected-device-shipment-forecast/`, March 2013.

[3] Andrey Mokhov. *"Conditional Partial Order Graphs"*. Newcastle University, PhD Thesis, Technical Report Series NCL-EECE-MSD-TR-2009-150, September 2009.

[4] Maxim Rykunov. *"Design of Asynchronous Microprocessor for Power Proportionality"*. Newcastle University, PhD Thesis, December 2013.

[5] TSMC, *"90nm Core Library"*, Ver. 150a, December 2005.

[6] Dr. Robert Moniot, *"SIS Tutorial Combinational Logic Circuit Design"*, February 7, 2011.

[7] Richard Rudell. *"Genlib: Combinational Gate Specification"*. Berkeley University, `http://www.eecs.berkeley.edu/~alanmi/publications/other/SIS_paper_genlib.pdf`.

[8] Berkeley University, `www.eecs.berkeley.edu/~alanmi/abc/`, Dept. of Computer and Electronic Engineering.

[9] M.Donno, E.Macii and L.Mazzoni, *"Power-aware clock tree planning"*, Proceedings of ISPD'04,2004.

[10] Cisco Systems Inc. Entering the Zettabyte Era, *"Visual Networking Index: Forecast and Methodology 2010-2015"*, 2011.

[11] Massimo Poncino, *"Power: Models and Metrics"*, *Energy Optimization for Embedded Systems* Slides, Politecnico di Torino.

[12] Massimo Poncino, *"Voltage & Frequency Scaling"*, *Energy Optimization for Embedded Systems* Slides, Politecnico di Torino.

[13] Berkeley University, `www.ecs.umass.edu/ece/labs/vlsicad/ece667/links/espresso.1.html`, Dept. of Computer and Electronic Engineering.

[14] I. Rocha and T. Pavlidis, *"A Shape Analysis Model with Application to a Character Recognition System"*, IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 16, 1994.

[15] L. Jianzhuang and L.Y. Tsui, *"Graph-Based Method for Face Identification from a Single 2D Line Drawing"*, IEEE Trans. Pattern Analysis and Machine

Intelligence, vol. 23, 2001.

[16] Luigi P. Cordella, Pasquale Foggia, Carlo Sansone, and Mario Vento, *A (Sub)Graph Isomorphism Algorithm for Matching Large Graphs*, IEEE Trans. Pattern Analysis and Machine Intelligence, Vol. 26, No. 10, October 2004.

[17] E.M. Luks, *"Isomorphism of Graphs of Bounded Valence can be Tested in Polynomial Time"*, J. Computer System Science, 1982.

[18] J.R. Ullmann, *"An Algorithm for Subgraph Isomorphism"*, J. Assoc. for Computing Machinery, vol. 23, 1976.

[19] B.D. McKay, *"Practical Graph Isomorphism"*, Congressus Numerantium, vol. 30, 1981.

[20] W.J. Christmas, J. Kittler, and M. Petrou, *"Structural Matching in Computer Vision Using Probabilistic Relaxation"*, IEEE Trans. Pattern Analysis and Machine Intelligence, vol. 17, no. 8, 1995.

[21] Random-shuffle description, `www.cplusplus.com/reference/algorithm/random_shuffle/`.

[22] S. Kirkpatrick, C. D. Gelatt Jr and M. P. Vecchi, *"Optimization by Simulated Annealing"*, vol. 220, number 4598, 13 May 1983.

[23] E. L. Lawlor, *"Combinatorial Optimization"*, (Holt, Rinehart & Winston, New York, 1976).

[24] A. V. Aho, J. E. Hopcroft, J. D. Ullman, *"The Design and Analysis of Computer Algorithms"*, (Addison-Wesley, Reading. Mass. 1974) .

[25] Francisco Javier Rodriguez-Diaz, Carlos Garcia-Martinez and Manuel Lozano, *"A GA-Based Multiple Simulated Annealing"*, IEEE, Evolutionary Computation (CEC), July 2010.

[26] QI Ji-Yang, *"Application of Improved Simulated Annealing Algorithm in Facility Layout Design"*, IEEE, Proceedings of the 29th Chinese Control Conference, July 2010.

[27] Workcraft web-site, `www.workcraft.org/wiki/`

[28] ARM Ltd., *"Cortex-M0+ Technical Reference Manual"*, Available on `http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ddi0432c/index.html`

[29] Paulius Stankaitis, *"Algebraic Specifications of ARM Cortex M0+ Instruction Set"*, Chapter 4. Technical report series NCL-EEE-MICRO-TR-2014-192, 2014.

[30] Maxeler Technologies. *"Introduction to Dataflow Computing"*, Slides course, Summer School 2013.

[31] Luciano Lavagno, Juan Chi Wang, Negin Ostadabbasi, Alessandro Moré, Giovanni Causapruno, Du Boyang, Emanuele Bellocchia and Maurizio Tranchero, *"Modeling and Optimization of Embedded Systems"*, Modeling and Optimization of Embedded Systems course, Politecnico di Torino, Department of Information Engineering, September 29, 2013.

[32] E. Sanchez, M. Sonza Reorda, *"Pipelining"*, *Computer architectures* Slides, Politecnico di Torino, Department of automation and information technology.

[33] J. Sparso, *"Asynchronous circuit design - a tutorial"*, In J.Sparso and S.Furber (eds.), *Principles of asynchronous circuit design - A systems perspective*, chapter 1-8, pages 1-152. Kluwer Academic Publishers, 2001.

[34] C.H. van Berkel, R. Burgess, J. Kessels, A. Peeters, M. Roncken, and F.Schalij. *"A fully asynchronous low-power error corrector for the DCC player"*, in ISSCC 1994 *Digest of Technical Papers*, volume 37, pages 88-89. IEEE, 1994. ISSN 0193-6530.

[35] T.E. Williams, M.A. Horrowitz. *"A zero-overhead self-timed 160 ns. 54 bit CMOS divider"*. IEEE Journal of Solid State Circuits, 26(11):1651-1661, 1991.

[36] L.S. Nielsen, C. Niessen, J. Sparso, and C.H. van Berkel. *"Low-power operation using self-timed circuits and adaptive scaling of the supply voltage"*. IEEE transactions on VLSI Systems, 2(4):391-397, 1994.

[37] Danil Sokolov , Ivan Poliakov, Alex Yakovlev. *"Analysis of Static Data Flow Structures"*. Fundamenta Informaticae 88 (2008) 1–30, IOS Press.

[38] Andrey Mokhov, Victor Khomenko, Danil Sokolov, Alex Yakovlev. *"On dual-rail control logic for enhanced circuit robustness"*. Application of Concurrency to System Design (ACSD), Pages 112-121, 27 June 2012.

[39] A. Mokhov, D. Sokolov and A. Yakovlev. *"Completion Detection Optimisation"*. Newcastle University, Technical Report Series NCL-EECE-MSD-TR-2005-109, October 2005.

[40] ARM Ltd. *"ARMv6-M Architecture Reference Manual"*. ARM DDI 0419C (ID092410), 2010.

[41] Ce Guo, Wayne Luk, Stephen Weston. *"Pipelined reconfigurable accelerator for ordinal pattern encoding"*.Application-specific Systems, Architectures and Processors (ASAP), 2014 IEEE 25th International Conference. Pages 194-201, 18-20 June 2014 .

[42] V, A, Unakafova, K.Keller. *"Efficiently measuring complexity on the basis of the real-world data"*. Entropy, vol. 15, no. 10, pp. 4392-4415, 2013.

[43] J.Martinez, R.Cumplido, C.Feregrino. *"An FPGA-based parallel sorting architecture for the burrows wheeler transform"*, Proceedings of International Conference on Field-Programmable Technology, 2005, pp. 295-296.

[44] D. Mihhailov, V. Sklyarov, I. Skliarova, A. Sudnitson. *"Parallel FPGA-based implementation of recursive sorting algorithms"*. International Conference of Reconfigurable Computing and FPGAs, 2010.

[45] Paolo Bernardi, Fulvio Corno, Maurizio Rebaudengo, Matteo Sonza Reorda. *"Dependable systems"*, *Testing* Slides, Politecnico di Torino, Department of automation and information technology.

[46] Mariagrazia Graziano. *"Microelectronic Systems Lecture Notes"*, *Microelectronic systems* Slides, Politecnico di Torino, III Facoltà di Ingegneria.

[47] Frank Vahid, Tony Givargis. *"RT-level embedded systems design"*, University of California. Paolo Enrico Camurati, *Specification and simulation of digital systems* Slides, Politecnico di Torino, III Facoltà di Ingegneria.

[48] Danil Sokolov, Andrey Mokhov, Alex Yakovlev, David Lloyd. *"Towards asynchronous power management"*. 2014 IEEE Faible Tension Faible Consommation (FTFC), Pages 1-4, May 2014.

[49] Andrey Mokhov, Alex Yakovlev. *"Conditional partial order graphs: Model, synthesis, and application"*. IEEE Transactions on Computers, Volume 59, Pages 1480-1493, November 2010.

[50] Andrey Mokhov, Alex Yakovlev. *"Conditional partial order graphs and dynamically reconfigurable control synthesis"*. Design, Automation and Test in Europe, 2008. DATE'08, Pages 1142-1147, 10/03/2008.

[51] Crescenzo D'Alessandro, Andrey Mokhov, Alex Bystrov, Alex Yakovlev. *"Delay/phase regeneration circuits"*. Asynchronous Circuits and Systems, 2007. ASYNC 2007. 13th IEEE International Symposium, Pages 105-116, 12/03/2007.

[52] Andrey Mokhov, Alexei Iliasov, Danil Sokolov, Maxim Rykunov, Alex Yakovlev, Alexander Romanovsky. *"Synthesis of Processor Instruction Sets from High-level ISA Specifications"*. IEEE Transactions on Computers, Volume 63, Issue 6, Pages 1552-1566. June 2014.

[53] A Mokhov, A Alekseyev, A Yakovlev. *"Encoding of processor instruction sets with explicit concurrency control"*. Computers & Digital Techniques, IET. Volume 5, Pages 427-439. November 2011.

[54] Ivan Poliakov, Danil Sokolov, Andrey Mokhov. *"Workcraft: A static data flow structure editing, visualisation and analysis tool"*. Petri Nets and Other Models of Concurrency–ICATPN 2007. Pages 505-514, 2007.

[55] Main contributors: Ivan Poliakov, Danil Sokolov, Stan, Bowen Li, Alessandro de Gennaro, Parviz Palangpour. Workcraft source code. `https://launchpad.net/workcraft`. Registered on 09-09-2009.

[56] Maxeler Technologies. O. Pell, V. Averbukh. *"Maximum Performance Computing with Dataflow Engines"*. Computing in Science & Engineering, doi: 10.1109/MCSE.2012.78, July-August 2012.