UNIVERSITY OF Southampton

University of Southampton Research Repository
ePrints Soton

Copyright © and Moral Rights for this thesis are retained by the author and/or other copyright owners. A copy can be downloaded for personal non-commercial research or study, without prior permission or charge. This thesis cannot be reproduced or quoted extensively from without first obtaining permission in writing from the copyright holder/s. The content must not be changed in any way or sold commercially in any format or medium without the formal permission of the copyright holders.

When referring to this work, full bibliographic details including the author, title, awarding institution and date of the thesis must be given e.g.

AUTHOR (year of submission) "Full thesis title", University of Southampton, name of the University School or Department, PhD Thesis, pagination

http://eprints.soton.ac.uk

UNIVERSITY OF SOUTHAMPTON

# Investigating Behavioural Synthesis into Bespoke Instruction Set Processors (BISPs)

by

Abdeldjalil Belouettar

A thesis submitted in partial fulfillment for the degree of Master of Philosophy

in the

Faculty of Engineering, Science and Mathematics
School of Electronics and Computer Science

November 2013

UNIVERSITY OF SOUTHAMPTON

ABSTRACT

FACULTY OF ENGINEERING, SCIENCE AND MATHEMATICS

SCHOOL OF ELECTRONICS AND COMPUTER SCIENCE

Master of Philosophy

by Abdeldjalil Belouettar

We propose a new approach to the behavioural synthesis of digital systems for which a synthesis tool has been implemented.

Traditionally, behavioural synthesis tools convert the behaviour of a system into a RTL description, modelling a data-path and a controller. Instead, in the proposed approach, a behavioural description is translated into a RTL description that models a set of *Bespoke Instruction Set Processors (BISPs)*. A BISP is a stripped-down microprocessor, which is composed of the minimal computational resources necessary to implement the part in the behavioural description from which it is derived. We refer to a BISP as a nano-processor throughout the thesis. This thesis looks at previous research on behavioural synthesis, describes the new approach and outlines the results of its evaluation and comparison to an existing behavioural synthesis tool. Results show that the new approach is less efficient than the existing technique when applied to small systems. However, the inability to support some VHDL constructs was the main obstacle against a full evaluation of the new approach.

# Contents

**Acknowledgements** **vi**

**1 Introduction** **1**

1.1 Overview . . . 1
1.2 Objectives . . . 2
1.3 Report Structure . . . 4

**2 Background** **6**

2.1 Timeline of Digital System Design . . . 6
2.2 Hardware Description Languages . . . 9
2.2.1 VHSIC Hardware Description Language (VHDL) . . . 9
2.2.2 Verilog . . . 10
2.3 Register Transfer Level synthesis . . . 10
2.4 Behavioural Synthesis . . . 10
2.4.1 Advantages of behavioural synthesis over RTL synthesis . . . 11
2.4.2 The need for an automatic high level synthesis system . . . 12
2.5 Previous Research in Behavioural Synthesis . . . 12
2.5.1 Intermediate Representation . . . 13
2.5.2 Scheduling Techniques . . . 15
2.5.3 Resource Allocation . . . 20
2.5.4 The MOODS Behavioural Synthesis System . . . 20
2.6 Multiple Objective Systems . . . 21
2.7 The Variety of Input Languages . . . 27
2.8 Harvard vs Von Neuman architectures . . . 27
2.9 CISC vs RISC . . . 28
2.10 Application Specific Instruction-set Processors . . . 29
2.11 Discussion . . . 30

**3 Behavioural Synthesis into Nano-Processors** **33**

3.1 The Nano-Processor Concept . . . 33
3.2 The Behavioural Synthesis Process . . . 35
3.3 VHDL as The Input Language . . . 36
3.4 Source Code Analysis . . . 36
3.5 The intermediate representation . . . 37
3.6 Generating Nano-processor structures . . . 39
3.7 Generating RTL description of the circuit of Nano-processors . . . 40
3.8 Example Transformation . . . 40

**4 Translating a Behavioural Description into a Parse Tree 43**
4.1 VHDL Compiler Environment . . . 43
4.2 Lexical Analyser . . . 44
4.3 Syntax Parser . . . 44
4.3.1 Parsing using a hypothesis tree . . . 47
Formal description of the algorithm . . . 49
4.4 VHDL Syntax: . . . 50
4.5 Semantic Analysis: . . . 50

**5 Synthesis into Fetch-Execute Structures 52**
5.1 Parse Tree to Control & Data Flow Graph . . . 52
5.2 CDFG to Processor Objects . . . 53
5.3 Processor Objects to RTL description . . . 54
5.3.1 Nano-Processor Template . . . 54
Instruction Memory: . . . 57
The Program Counter . . . 57
The Instruction Register . . . 57
The Register File . . . 57
The Execution Unit . . . 58
The Data Memory . . . 59
The Controller/Sequencer module . . . 59
5.3.2 Binding Processor Objects to Customisations of the Nano-Processor Template . . . 59

**6 Experiments and Results 61**
6.1 Evaluation Strategy . . . 61
6.2 Synthesis Results of the logical AND operator . . . 62
6.3 Synthesis Results of the Arithmetic Multiplier . . . 67
6.4 Iterative Fibonacci Calculator . . . 72
6.5 Further Experiments . . . 76
6.5.1 Cycle Count . . . 76
6.5.2 Longer sequential code . . . 76

**7 Final Remarks 78**

**A Details Of The VHDL Parser 79**

**B RTL Synthesis of Test Cases 82**

**C Long sequential code 84**

**Bibliography 87**

# List of Figures


1.1 Behavioural Synthesis within the electronic system design process . . . . . 2
1.2 An example controller and data path view of a system . . . . . . . . . . . 4
1.3 An example output of the proposed approach . . . . . . . . . . . . . . . . 5

2.1 Example mask layouts of logical gates . . . . . . . . . . . . . . . . . . . . 7
2.2 An example circuit layout, drawn using logic gate symbols . . . . . . . . . 7
2.3 General Flowchart of Behavioural Synthesis . . . . . . . . . . . . . . . . . 13
2.4 High-Level description & corresponding Data Flow Graph . . . . . . . . . 14
2.5 ASAP and corresponding ALAP schedules for a simple system . . . . . . 16
2.6 Force-Directed Scheduling . . . . . . . . . . . . . . . . . . . . . . . . . . 18

3.1 Example VHDL code and corresponding Nano-Processors . . . . . . . . . 34
3.2 Flow Chart of the proposed behavioural synthesis process . . . . . . . . . 35
3.3 VHDL code and corresponding parse tree . . . . . . . . . . . . . . . . . . 38
3.4 VHDL code with corresponding Control and data flow graph . . . . . . . 39
3.5 Block Diagram of the Example Nano-Processor Implementation . . . . . . 41

4.1 Example Structure and Use of the Hypothesis Tree . . . . . . . . . . . . . 48

5.1 Example CDFG extracted from a parse tree . . . . . . . . . . . . . . . . . 53
5.2 A sample object representation of a circuit containing two processor objects 55
5.3 Template of a Nano-Processor . . . . . . . . . . . . . . . . . . . . . . . . 56

6.1 The number of Slice Registers used to implement the logical AND . . . . 63
6.2 The number of LUTs used to implement the logical AND . . . . . . . . . 64
6.3 Delay in nano seconds through the critical path of the logical AND . . . . 65
6.4 Percentage difference with MOODS output optimised for area . . . . . . . 65
6.5 Percentage difference with MOODS output optimised for delay . . . . . . 66
6.6 The number of Slice Registers used to implement the Arithmetic Multiplier 68
6.7 The number of LUTs used to implement the Arithmetic Multiplier . . . . 68
6.8 Delay in nano seconds through the critical path of the Arithmetic Multiplier 69
6.9 Percentage difference with MOODS output optimised for area for the Arithmetic Multiplier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 70
6.10 Percentage difference with MOODS output optimised for delay for the Arithmetic Multiplier . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 71
6.11 The number of Slice Registers used to implement the iterative Fibonnaci Calculator . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 73
6.12 The number of LUTs used to implement the iterative Fibonnaci Calculator 73

6.13 Delay in nano seconds through the critical path of the iterative Fibonnaci Calculator . . . . . 74
6.14 Percentage difference with MOODS output optimised for area for the iterative Fibonnaci Calculator . . . . . 74
6.15 Percentage difference with MOODS output optimised for delay for the iterative Fibonnaci Calculator . . . . . 75

A.1 Graph Representation of the Syntax Rule for Entity_declaration . . . . . 81

# Acknowledgements

I would like to thank my supervisor, Professor Andrew Brown, for his support throughout the course of this research. His invaluable insight and years of experience have been pivotal to this contribution.

I would also like to thank the University of Southampton and the School of Electronics and Computer Science for providing the facilities and environment enabling me to investigate this research topic.

Many thanks go to my family and friends for their continuous encouragement and support.

I would also like to recognize the value of the support I received from the Algerian government during the initial phases of this project.

# Chapter 1

# Introduction

## 1.1 Overview

Ever since its introduction to the design flow, reconfigurable hardware has been widely used by engineers and became an integral part of electronic systems design.

Programmable integrated circuits, such as Field Programmable Gate Arrays (FPGA), are not restricted to prototyping and design verification, but can now be used as stand-alone fully fledged circuits or as co-processors to commercial and mainstream computer systems for the purpose of balancing the computational load.

Equipped with flexibility, speed and affordability, FPGAs will no doubt be an indispensible part of electronics and computer systems of the next generation.

At the time of writing, the most widely used input formats for programming an FPGA are Hardware Description Languages (HDL). HDLs give the designer the capability of describing electronic hardware in a number of different levels of abstraction.

**Gate Level** descriptions represent the hardware as a network of gates or cells from a technology specific library. **Register Transfer Level (RTL)** descriptions represent the hardware in terms of the transfer of data between registers and functional units.

Some hardware description languages, such as VHDL, allow designers to describe hardware at a higher level of abstraction. At such level, the designer can write a behavioural description of a system that represents the desired algorithm or behaviour using common high-level programming language constructs.

In electronic system design, **Behavioural Synthesis**, as commonly referred to as High-Level Synthesis, is the process of transforming a high-level behavioural description of an electronic system into a lower-level register transfer level representation. The latter is then transformed into a lower level transistor layout using third party RTL synthesis tools. Figure 1.1 shows the use of behavioural synthesis in the design process.

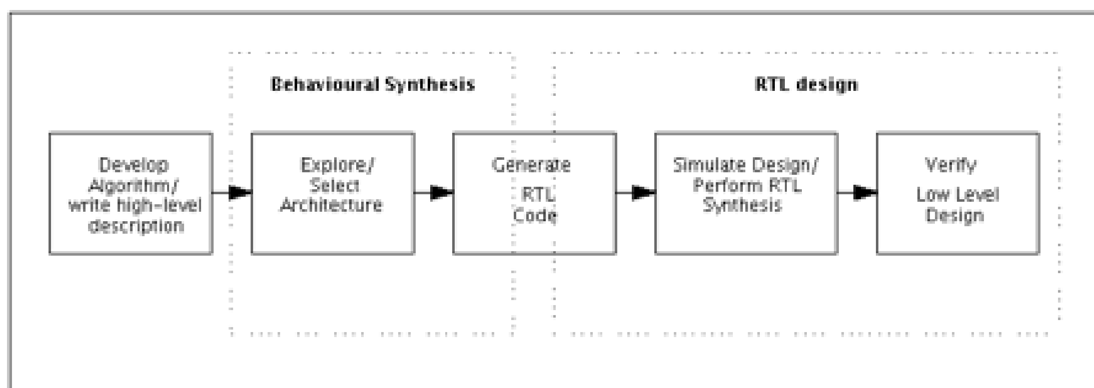


FIGURE 1.1: Behavioural Synthesis within the electronic system design process

## 1.2 Objectives

Algorithms require a varying amount of hardware resources and may have certain timing criteria. Circuits implementing those algorithms may be required to use power efficiently

and be easily tested and diagnosed for faults. The complex task of finding the optimum configuration triggers the need for an expert designer.

However, expert designers significantly increase costs and would not deliver a complex product within today's time to market requirements. The more complex the system is, the higher probability of error in human designs. On top of that, an expert designer can only visit a limited subset of the design space within a limited time frame. For these reasons, there was an inevitable shift towards using automated design exploration tools.

The goal of this project is to explore the viability of a new approach to behavioural synthesis, as most behavioural synthesis tools, that have been developed previously, use a common approach to generate a RTL description equivalent to the originating behavioural description.

Using the common approach, the output of the behavioural synthesis system is typically split into two architectural parts: a data path and a controller. This essentially reduces the representation of all systems to a simple state machine, which generates control signals that orchestrate the flow of data through the data path.

The data path is composed of a number of storage, interconnect and functional units. Figure 1.2 shows an example of such representation.

In the new approach explored in this project, the behavioural synthesis system generates an equivalent RTL description as a network of custom-fit instruction set processors. Each processor has a unique instruction set architecture, tailored to implement part of the behavioural description as a sequence of fetch-execute instructions.

The motivation behind investigating such an approach is to provide a new scope for design exploration, where factors such as instruction set architectures, register file and

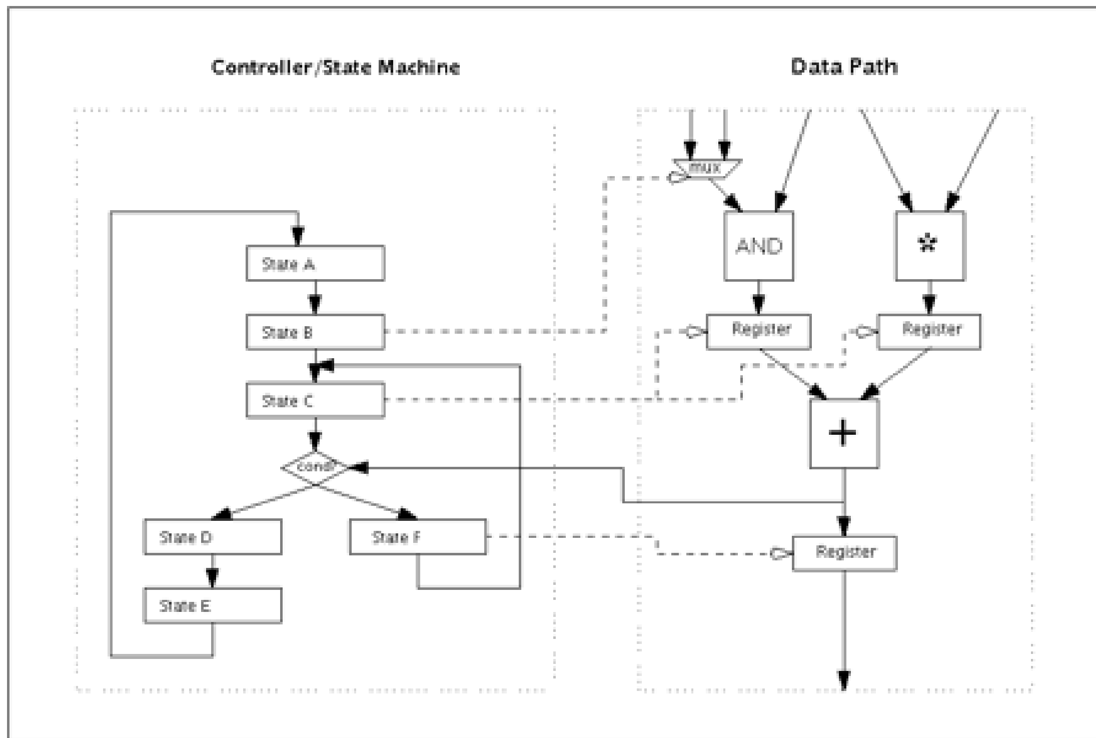


FIGURE 1.2: An example controller and data path view of a system

data bus dimensions are some of the variables to be manipulated.

A behavioural synthesis tool has been created that implements such an approach. The input to the synthesis tool is a behavioural subset of VHDL and the output is RTL VHDL code describing the network of fetch-execute structures. Figure 1.3 shows a diagram of an example of such output.

## 1.3 Report Structure

This report serves to document the outcome of investigating the newly described behavioural synthesis approach.

It is divided into four parts. **Chapter 2** delves into previous research in high-level synthesis and explains some of the concepts referenced in this report. **Chapter 3** further

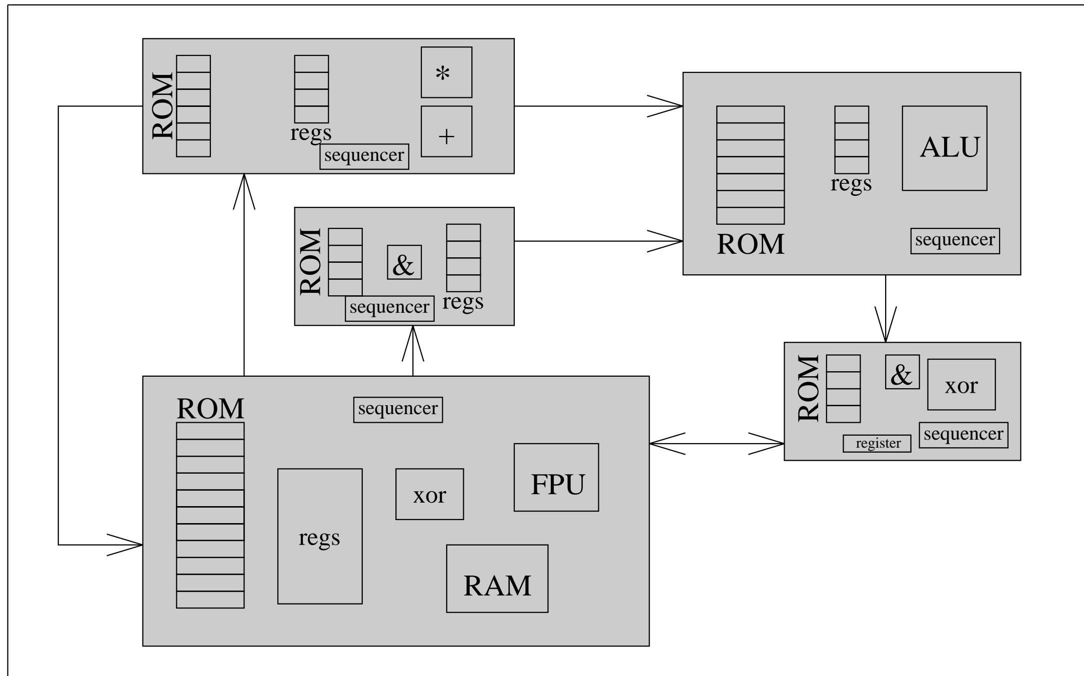


FIGURE 1.3: An example output of the proposed approach

explains the new approach to behavioural synthesis using fetch-execute structures as output. **Chapter 4** describes the implementation of the front-end of the system (the VHDL compiler). **Chapter 5** describes the implementation of the back-end of the system (the synthesis). **Chapter 6** showcases the results of the evaluation of the new approach. **Chapter 7** draws some final remarks.

# Chapter 2

# Background

## 2.1 Timeline of Digital System Design

Three decades ago, before the introduction of the Personal Computer (PC), digital circuit designs consisted of few hundreds of logic gates. Referred to as Large Scale Integration (LSI) designs, the engineers created these designs by manually drawing transistor level layouts (Mask Layouts). At that scale it was possible to generate fully functioning and reliable Integrated Circuits in time for market deadlines. Figure 2.1 shows an example circuit layout.

Further technological advances resulted in higher circuit resolutions, this time in the order of thousands of gates. At this scale, digital circuit designs were referred to as Very Large Scale Integration (VLSI) designs.

Increased complexity of the circuits, coupled with shorter time to market windows meant that there was a need to move the design process to a higher level of abstraction.

Backed by the availability of Personal Computers, designers had at their disposal a few design automation frameworks that allowed them to describe the circuits in terms of networks of logic gates instead of mask layouts. Figure 2.2 shows an example circuit represented at the gate level.

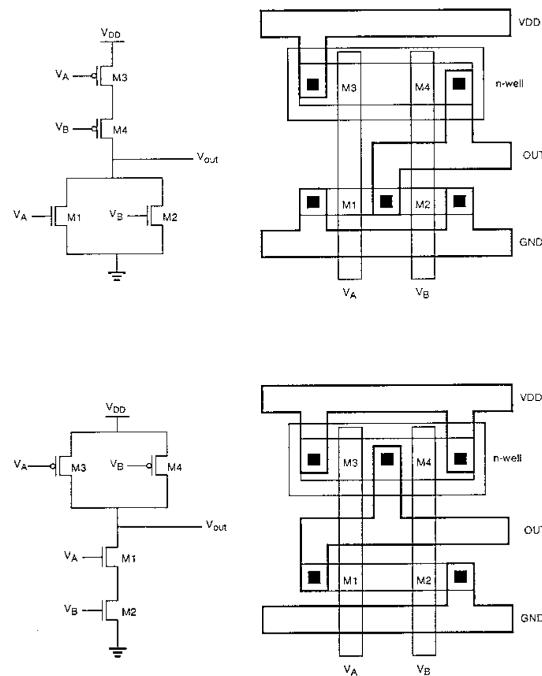


FIGURE 2.1: Example mask layouts of logical gates

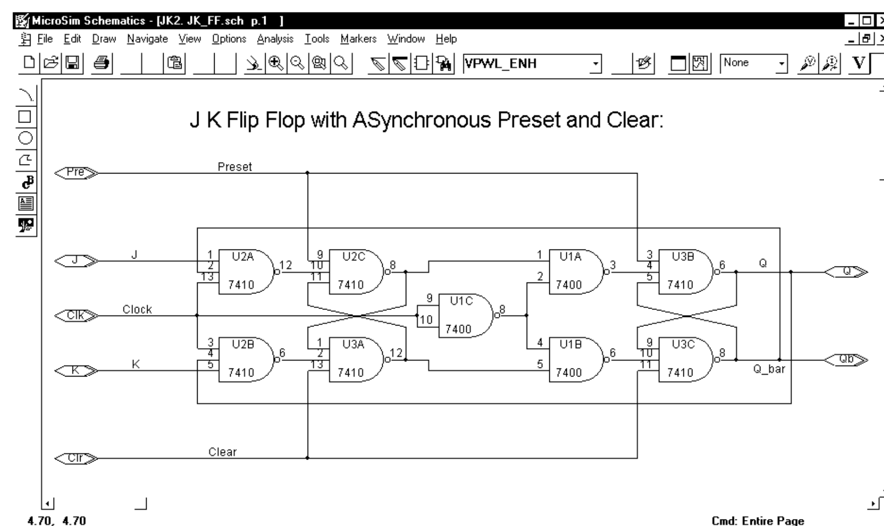


FIGURE 2.2: An example circuit layout, drawn using logic gate symbols

Continuous developments led to affordable higher density circuits being readily available, as a platform for more complex circuits. This resulted in shifting the circuit designer's priorities from producing the best possible design to producing a working design within the time to market window.

At this point, place-and-route algorithms that did a better job than the human designer, in a shorter time span, were implemented and shipped as commercial Electronic Design Automation (EDA) tools. Designers were able to describe circuits at an even higher level of abstraction (the Register Transfer Level).

Using Hardware Description Languages (HDL) such as Verilog and VHDL, the designer could describe the whole circuit in terms of data storage elements (Registers, Memories) and functional blocks (Adders, Logic gates), which form a data path part of the circuit controlled by a Finite State Machine that orchestrates the data flow through the data path. RTL synthesis tools took over and produced near optimal implementations of the user's RTL input.

The continuous development in the fabrication process resulted in even higher transistor density. This led to the availability of circuits that could accommodate much larger designs. Circuits were large enough to become increasingly time consuming to design using RTL code. At this point research has already started on the high-level synthesis process.

Moving the design to a higher level of abstraction reduces the design time significantly, compared to design using low level RTL code. Tens of lines of high level code were equivalent to thousands or more lines of RTL code. In addition to that, each behavioural description could be implemented using more than a single architecture.

Automating the process of high-level synthesis opened the possibility to explore many possible architectures within a shorter time frame. However, finding the best possible architecture for a behavioural description posed a challenge that is still being tackled today. In fact, behavioural synthesis has not yet won the industry's approval and RTL synthesis is, to date, the tool of choice.

## 2.2 Hardware Description Languages

A Hardware Description Language (HDL) is a programming language used by electronic systems designers to describe the structure and behaviour of a digital circuit. HDLs are a common input form to the majority of Computer Aided Design (CAD) tools that are capable of hardware synthesis.

### 2.2.1 VHSIC Hardware Description Language (VHDL)

VHDL stands for Very high speed integrated circuit Hardware Description Language. It is a widely used hardware description language mainly because of the flexibility in allowing user defined types and structures, and also its capability of describing circuits at different levels of abstractions.

VHDL originates from the Ada programming language; hence the two have a lot of common syntactic and semantic constructs. For this reason, VHDL can be considered as a good input format for behavioural descriptions.

### 2.2.2 Verilog

Verilog is another popular hardware description language. Although it can be used to describe the behaviour of a system, it is more suitable for hardware description at the RTL level.

Compared to VHDL, the user cannot define their own data types. Instead, only a limited set of data types is available, aimed mainly at representing actual hardware constructs (such as wire, reg and tri).

## 2.3 Register Transfer Level synthesis

An RTL representation describes a digital system in terms of functional units (such as logical gates, adders, multipliers and floating point units), storage units (registers, RAMs, ROMs), interconnect infrastructure (wires, busses, multiplexers and so on), and the flow of data between these components. Popular languages used to write RTL code are VHDL and Verilog.

## 2.4 Behavioural Synthesis

In electronic systems design, behavioural synthesis is the automatic process of transforming a behavioural (algorithmic) description of a system into a Register Transfer Level (RTL) description.

The input representation to such a process describes the behaviour of a digital system in terms of data structures, operations, and control structures (such as conditionals, loops, and procedure calls). The input may be written in any high level programming language

like C or C++, or any Hardware Description Language (HDL) like VHDL, Verilog or SystemC.

The output RTL description is in turn transformed, using well established third party RTL synthesis tools, into a physical silicon layout for prototyping or manufacturing hardware circuitry (on FPGAs or ASICs).

### 2.4.1 Advantages of behavioural synthesis over RTL synthesis

Behavioural descriptions for typically large circuits are orders of magnitude smaller than RTL descriptions and require much less designer effort to describe. Behavioural descriptions essentially have little or no implementation details, such as number and type of functional units as well as the underlying interconnect structure and timing information.

Behavioural synthesis allows the circuit designer to explore alternative architectures from the vast design space in a limited amount of time through manipulating a small number of design constraints such as area of resources and critical path delay or power consumption.

The alternative architectures are explored automatically as part of the behavioural synthesis process and the best architecture that fits the design constraints is fed back to the designer. The aim of all behavioural synthesis tools is to explore as many points of the design space as possible in a short amount of time. Therefore, some algorithms only explore parts of the design space that have a higher probability of falling within the constraints.

Although this achieves the desired goal, the level of quality of the chosen architecture cannot be guaranteed to be the optimal level for the given constraints.

### 2.4.2 The need for an automatic high level synthesis system

Technological advances allowed the continuous decrease in transistor sizes; hence, millions of transistors on a single integrated circuit became the norm. This allowed bigger and more complex digital systems to be implemented on a single chip.

Manually designing the netlist and writing the RTL description for such digital systems became a time consuming and error prone process. The move to automatic synthesis was inevitable.

Automating the synthesis process meant that design time could be spent on making important design decisions and error checking, rather than being wasted on monotonous transistor layout. Behavioural Synthesis allowed designers to explore multiple solutions in the design space within an affordable time frame.

## 2.5 Previous Research in Behavioural Synthesis

The idea of behavioural synthesis can be traced back to the *Bristle Blocks* system proposed by Johannsen [22]. At that time, such systems were known as *silicon compilers*.

The Bristle Blocks silicon compiler managed to automatically generate data-path and control blocks from a high-level input description of the chip in a short time, even though parts of the chip had to be manually laid out.

Better silicon compilers then emerged, which produced complete circuit layouts automatically.

Most of the high-level synthesis systems that were proposed partitioned the synthesis task into three very close and interdependent subtasks: Scheduling, Resource Allocation and Module Binding (see Figure 2.3).

The starting point however, was always an input description written using a high-level programming language or a Hardware Description Language. Since this description was tailored for human readability, no scheduling, allocation or binding could proceed before it was parsed into a machine friendly intermediate representation.

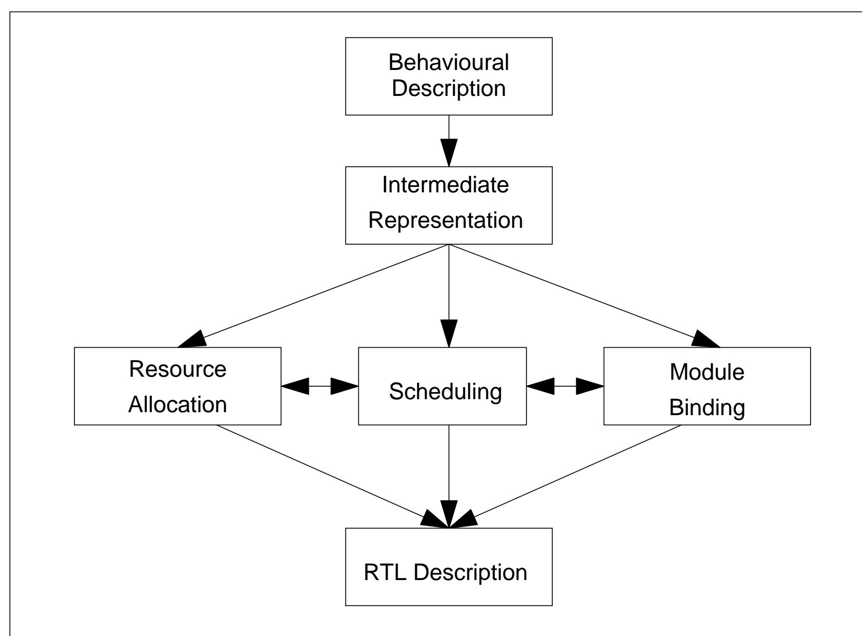


FIGURE 2.3: General Flowchart of Behavioural Synthesis

### 2.5.1 Intermediate Representation

The majority of synthesis systems use a graph-based internal representation derived from the flow of data in the description.

The Data Flow Graph (DFG) captures the movement of data derived from assignment statements where the vertices represent operations and the edges represent the data dependencies.

The Control Graph captures the control portions of the description such as conditional branches and loops. As an example, Figure 2.4 shows a high-level description of a simple system and the corresponding data flow graph.

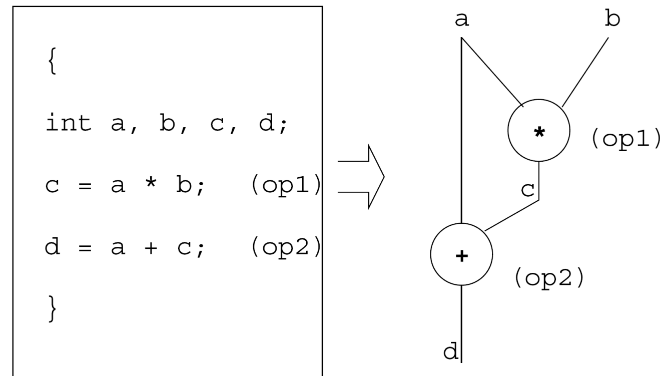


FIGURE 2.4: High-Level description & corresponding Data Flow Graph

Many systems have adopted a different name for the combination of the two graphs. The Control and Data Flow Graph (CDFG) was used in the HAL system [37], the Value Trace was found in the CMUDA system by Thomas et al. [45] while others called it the Data Dependency Graph (DDG) or the Directed Acyclic Graph (DAC).

IGR was a slightly different Internal Graph Representation used in the SCHOLAR silicon compiler [3], where nodes corresponded to time steps instead and each node contained the operations to be executed in the same time step.

Another intermediate representation was used in parallel controller synthesis called Petri Net [25]. Where the graph was a bipartite, weighted, directed graph which had two types of nodes called places and transitions.

Recent research by Sinha and Patel [44] looked at the use of Abstract State Machines (ASMs) as an intermediate representation, targeted specifically at high level languages which lack the parallelism semantics such as C-like languages. This representation would be a suitable model for supporting the numerous variants of the C language that provide

parallelism and timing semantics extensions, as well as an output of algorithms that automatically extract such semantics from the behavioural code. The authors base their synthesis tool on an extensible java based open-source framework for modelling and simulating ASM specifications called CoreASM [13]. Hardware description languages such as VHDL can be directly translated into such intermediate representation since parallelism is inherent to the language definition.

A recent study carried out by Kelley et al. [23] compared the use of microcode sequencers against flexible Finite State Machines as an implementation of runtime reconfigurable controllers. The study finds that microcode sequencers are often the more efficient implementation in this context.

### 2.5.2 Scheduling Techniques

Scheduling is the process where timing information, necessary for hardware implementation, is introduced into the data flow graph of the intermediate representation; giving an overall order of the execution of operations. This is done by mapping the operations onto time steps, or control steps as commonly known. These control steps can be viewed as clock cycles.

Many scheduling techniques have been proposed. The simplest one is, possibly, the As Soon As Possible (ASAP) scheduling technique, used in the early CMUDA system [20] (developed at Carnegie-Mellon University) and in the Flamel system [46].

ASAP scheduling assumes no limit on the resources available and proceeds by assigning operations to the earliest possible control step, as soon as the data dependencies allow. The issue when ASAP scheduling is used, assuming a limit on the available resources, is that operations which lie in the critical path may be delayed and miss their deadlines.

Another scheduling algorithm is the As Late As Possible (ALAP) scheduling, which assumes a maximum number of control steps, and proceeds by assigning operations to the latest possible control step that does not violate the delay constraint.

Figure 2.5 shows the difference between the ASAP and ALAP scheduling techniques. We can see in the ASAP schedule that operation 4 was scheduled in the first control step because all its inputs were available whereas in the ALAP schedule it was scheduled in the second control step because operation 5 which requires its output was scheduled in c-step 3.

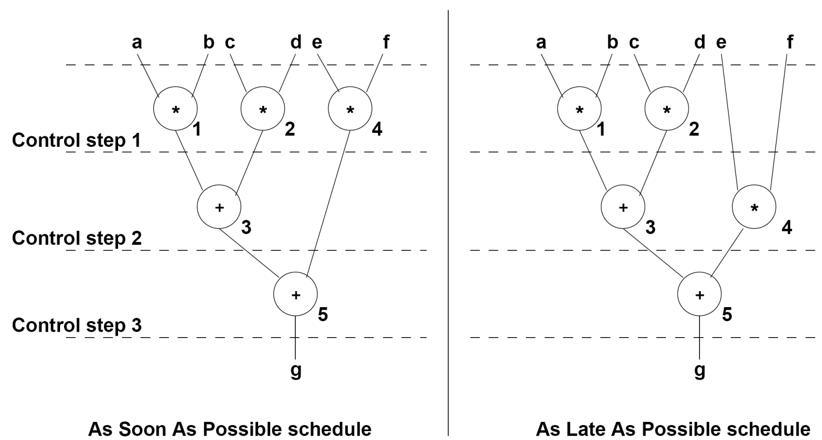


FIGURE 2.5: ASAP and corresponding ALAP schedules for a simple system

List Scheduling algorithms are adaptations of ASAP and ALAP techniques, which generate a list of operations that are ordered based on a priority function, as an attempt to prioritise the operations on the critical path.

A priority criterion, called operation mobility, was proposed in the SLICER system [33]. Operation mobility is calculated as the difference in the number of control steps between the ASAP and ALAP schedules. An urgency criterion is, instead, used in the ELF system [14], that is the length of the shortest path from an operation to the nearest deadline.

Force-Directed scheduling [36], used in the HAL silicon compiler [37], is another approach that attempts to uniformly distribute operations that use the same resources into the entire control steps.

Like list scheduling, the force-directed algorithm derives both the ASAP and ALAP schedules in order to determine the time frame of each operation. Figure 2.6 shows the workings of this algorithm, starting with the ASAP and ALAP schedules, then drawing the time frames for each operation.

At this point, it is possible to determine the probability that an operation will be assigned to a particular control step. The next step is to create a Distribution Graph which sums up the probabilities of each type of operation for each control step of the CDFG. This can be considered as a measure of parallelism (concurrency) of similar operations.

The scheduling techniques mentioned above can be grouped together under the class of *Constructive Algorithms*, because they proceed by scheduling one operation at a time and build up a schedule, until all operations are assigned a control step.

Another class of scheduling systems use an *Iterative Transformational* technique, where an initial schedule (either maximally parallel or maximally serial as in the CAMAD system [39]) is the starting point. This is followed by iterative modifications and transformations in order to improve the initial schedule and meet the constraints.

Another group of schedulers use the *Integer Linear Programming* formalism, such as the one proposed by Lee et al. [27]. This kind of schedulers reaches optimal solutions for small systems, but is inefficient when faced with large and complex behavioural descriptions.

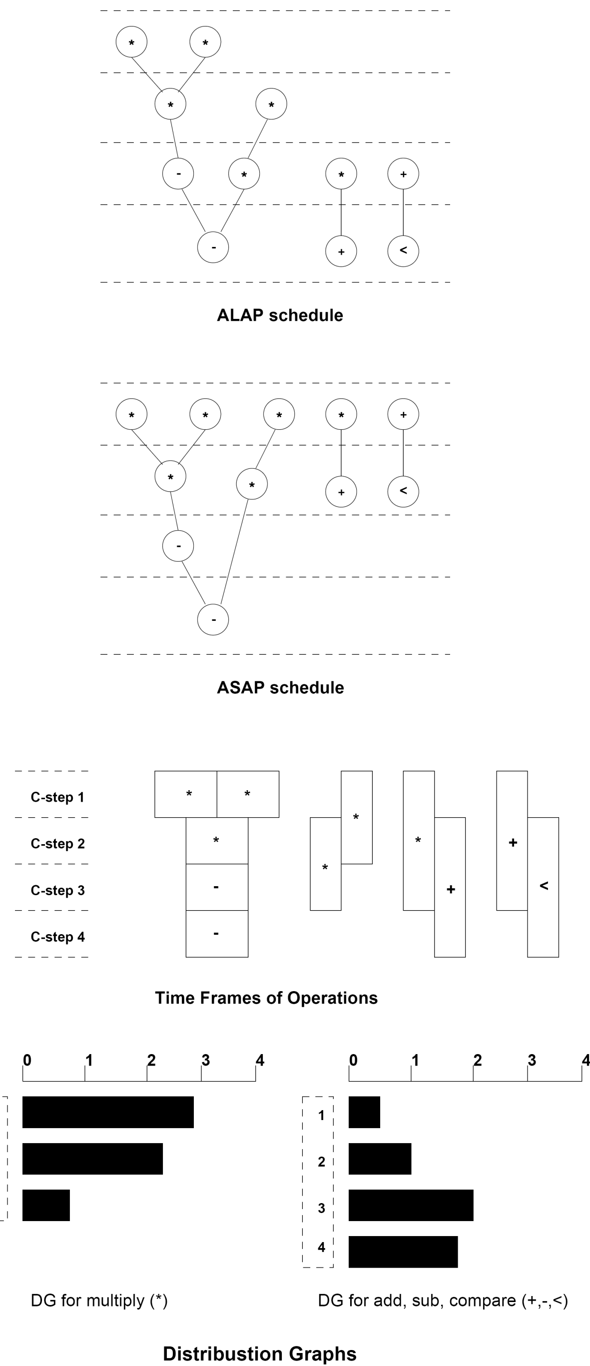


Figure 2.6: Force-Directed Scheduling

The other class of schedulers is the *probabilistic scheduling* techniques. These include the Simulated Annealing algorithm introduced by Devadas and Newton [10] (used in the SALSA system [32] and MOODS [7]) and the Genetic Algorithm technique such as the one proposed by Heijligers et al. [19], which is a resource constrained scheduler that improves on list scheduling techniques.

Some scheduling algorithms employ multi cycle functional units in order to reduce the length of the critical path (i.e. the cycle duration) of the resulting architecture [24] [21]. However, the reusability of such underlying hardware resources is fairly limited which is a gap that Molina et al. [31] tries to fill, using a post synthesis algorithm which can be described as two steps:

1) Decomposition of multi-cycle operators

2) Removal of some datapath functional units (mainly because the number of datapath FUs increases after the first step).

The algorithm mainly targets multiplier functional units.

Some researchers have been looking at reducing the cycle duration of common functional units such as adders and multipliers (where the length of the carry path is a major contributor to cycle delay) by using Speculative Functional Units [9]. These functional units employ prediction mechanisms for the carry signal and thus reduce the cycle delay by removing the carry path. Eventually the technique must employ a recovery mechanism as well for when the predictions are wrong. Even though the dynamic scheduling algorithms proposed are reported to improve performance by up to 33%, the area overhead for implementing the prediction and recovery mechanism is not to be ignored. Add to that, the increased complexity of the resulting architecture.

### 2.5.3 Resource Allocation

Resource Allocation is the process of determining the number and type of resources that will be assigned to the operations of the behaviour. This is not to be confused with the Module Binding task, which is the process of the actual mapping of each hardware component to one or more operations.

Data-path allocation techniques have been classified by researchers into three different approaches: *constructive methods* [20], *decomposition methods* (using clique-partitioning or left edge algorithms [26]), and *iterative methods.* Lin [29] published a discussion about most of these approaches.

A more recent iterative approach has been published by Cong et al. [6] where the authors propose using consistent optimisation goals throughout the high-level synthesis process. In the proposed approach, resource sharing intentions at the allocation step are fed to the scheduler in an iterative process where the scheduler would use the constant feedback to find schedules that satisfy the overall optimisation objectives.

### 2.5.4 The MOODS Behavioural Synthesis System

Multiple Objective Optimisation of Data and control path Synthesis (MOODS) [7] is a behavioural synthesis tool developed at the University of Southampton. It compiles a behavioural description of a digital system written in behavioural VHDL into a structural description written in structural VHDL.

The MOODS system performs the behavioural synthesis task by initially building a flow graph of the description with one operation in every control step and one functional unit for every operation. This is an inefficient but accurate representation of the input

behavioural description. The graph is then iteratively modified until the user objectives are met. At that point, the structural description of the behaviour is directly translated from the modified graph.

## 2.6 Multiple Objective Systems

### Area & Delay Objectives

One of the advantages of high-level synthesis is the ability to include some constraints within the behavioural description of a system.

The human designer can request that certain operations need to execute within a limited time frame, or that a whole algorithm must not exceed a certain delay. He can also request that the whole system has to be implemented within a constrained silicon area, or using a limited number of functional resources.

Silicon area and delay constraints represent the basic trade-offs that most scheduling, allocation, and binding techniques are concerned with. The different scheduling techniques, mentioned earlier, have managed to a certain extent to explore the effects of these constraints on the final output.

As a general observation, maximally parallel implementations of a system are faster, but use a huge silicon area. Whereas minimum use of silicon area results in slower implementations.

After developing a wealth of Behavioural Synthesis systems with acceptable output, researchers started to address a set of other issues.

### The Low Power Objective

The need for low power consumption in electronics applications especially the fast growing market of portable devices (e.g. mobile phones and tablets) and the requirements for saving energy have triggered research on a new aspect of high-level synthesis.

Design for low power became a very important part of the behavioural synthesis process. The existing scheduling and allocation algorithms were modified and new ones emerged in order to meet this new requirement.

The benefits of a power efficient system are priceless, not only allowing a longer battery life but also increasing the reliability of the circuit by reducing the rate of failures due to high temperatures, in addition to minimizing the impact on the global environment [38].

According to Pedram [38], there are four main sources of power dissipation in digital CMOS circuits: leakage current, standby current, short-circuit current and capacitance current (switching activity). And there are three dimensions of low-power design space: voltage, physical capacitance, and data activity.

At the behavioural level, significant power reductions can be achievable if power consumption estimates of RTL components are provided. These estimates can be introduced as properties and metrics for power aware scheduling and binding algorithms.

$$DynamicPower = Capacitance * Frequency * SupplyVoltage^2$$

The equation above shows that the best way of reducing dynamic power dissipation is by reducing the supply voltage. However, there is a limit to that which is the threshold voltage of the circuit.

A technique which makes use of this property was proposed in [4] where increased parallelism and pipelining are used to compensate for the increased delays when lowering the supply voltage. However there are limitations to this technique in finding an optimal solution that meets the constraints of speed, area and low power.

A large source of power dissipation in microprocessors and ASICs is due to the clock distribution. Because all registers need a clock input and a significant number of these registers are inactive for long periods of time, there is a possibility of reducing clock activity at the input of those registers.

This can be done by adding an AND gate with an input that acts as an enable signal to allow the clock to propagate when it is high. This technique is called Clock Gating and has the disadvantages of increasing the complexity of the controller and further delays which are not desired in a high speed system.

Other ways of reducing power include the use of multi-cycle functional units like the one used in the SCALP system [41]. Another technique used in [42] is called Variable Supply Voltage which gives the highest supply voltage to modules in the critical path in order to meet critical deadlines, and uses lower supply voltages for the non-critical path modules.

This last approach requires the use of level shifters which scale the voltage up or down between connected modules operating at different supply voltages. The advantage of this technique is the lower area overhead for meeting computation time constraints compared to the parallel transformations proposed in [4].

Shin et al. [43] proposed an allocation algorithm that targets dual voltage storage elements and minimises the number of high Vdd registers. The authors report that the algorithm reduces the switching and leakage power by 20% on average.

Power is consumed in large due to the switching activity of the functional units, however, the continuous advancements of transistor technology brings the power dissipation of the interconnect structure in line with that of functional units.

Because of the big size of interconnects relative to the components, the resistances of the wires are no longer to be ignored. An interconnect-aware binding technique was presented in [48].

There are many ways to reduce power in behavioural synthesis, but some solutions are more practical than others. Some techniques cannot be applied to certain applications and are limited by the target technology. For example, if the target technology does not allow voltage scaling or there are only two possible voltages to use, it may render the whole technique inefficient.

### Testability and Reliability

If an integrated circuit is designed as part of a digital system that is responsible of the proper functioning of a spacecraft, it is essential to provide proof that the circuit will work as required. Furthermore, it is also desirable in the event of a fault under unpredictable circumstances that the system will be able to detect that fault, revert back to the last successful operation and retry in order to recover from the fault. The same issue applies to life-critical systems.

Equally important is the ability for manufacturers to test their products before delivering it to the customer. Companies know that there is no benefit of managing to ship the orders in time if half of the products are faulty. In general, failing to satisfy the customer in this way might risk the future of a company. For a large and complex integrated circuit, a huge number of test vectors need to be generated and fed through.

Testability can be defined using two concepts: The first one is Controllability, which answers the question of whether we can control a variable to any desired value by appropriately controlling the primary input variables of the circuit in order to establish if there is a fault.

The second concept is Observability, which answers the question of whether it is possible to observe a certain variable at the primary outputs. The principle of the Built-In-Self-Test technique is to add extra logic to the circuit to improve the testability of the circuit and perform the testing by generating test vectors internally and applying them to the Circuit Under Test (CUT). This self testing can be performed in field as well as in the fabrication lab. Test vectors can be generated using a linear feedback shift register for example.

In high-level synthesis, providing an easy and efficient way of incorporating test circuitry in the final RTL description is of similar importance to meeting area, speed or low power constraints. However this task is not a trivial one, as difficulties arise for large and complex systems.

The inclusion of test circuitry undoubtedly results in area and delay overheads which bring up another dimension for trade-offs and optimizations. Other challenges arise when dealing with loops, conditionals, multi-cycling or pipelining. A lot of research has been carried out in this area during the past 20 years.

A possible way of achieving behavioural synthesis for testability is to optimize the scheduling algorithm in order to enhance the testability and increase the fault coverage of the internal registers.

The Mobility Path Scheduling (MPS) algorithm proposed by Lee et al. [28] does this following a set of scheduling rules in order to achieve two allocation objectives: To

allocate a register to at least one primary input or primary output variable whenever possible, and to reduce the path from a controllable register to an observable register.

Other techniques try to avoid the creation of loops in the data-path since they contribute to the difficulties of automatic test pattern generations. Loops can either be inherited from the behavioural description or be formed as a result of hardware sharing [40].

It is also possible to modify the behavioural description in order to achieve RTL implementations of better testability. This can be done by analysing the description to determine hard-to-test areas similar to the technique proposed by Chen et al. [5] which then classifies variables as controllable, partially controllable, observable and partially observable in order to identify where to insert test points or use partial scan.

The testability issue is not only concerned about the data path testability. In fact, it spans the impact of the controller synthesis, the power dissipation during a test and the time required to complete it. Researchers have been working on these aspects as well.

### Other Objectives

At high-levels of abstractions, there is virtually no limit to the kind of constraints a user can impose on the hardware implementation of a system. If a user designs a digital signal processing algorithm, they can decide the floating-point accuracy required at the hardware level, by feeding this as a constraint to the behavioural synthesis tool.

For improved security, one can also constrain the behavioural synthesis tool to generate hardware implementations that reduce or avoid side channel leakage. One can also request that the circuit consumes constant power, as to avoid security compromise.

## 2.7 The Variety of Input Languages

Currently, there are two widely used hardware description languages (VHDL and Verilog). However, in an attempt to close the gap between programming languages such as C/C++ which are commonly used and favoured by software designers, C-based languages such as SystemC and Handle-C were developed.

These new languages include hardware concepts such as concurrency and timing constructs, as well as conventional software concepts. Even though such initiatives are aiming to standardise the language used to represent system behaviours, creating new languages certainly comes with the extra burden of user training and familiarization and can result in loss of productivity.

Economakos et al. [12] presented a new design environment for high-level hardware synthesis, where the input behaviour is described using the SystemC language, and the output is described in either VHDL, Verilog or SystemC.

A behavioural synthesis system, which takes high-level descriptions written in the C language and generates the equivalent RTL description in VHDL, was developed by Gupta et al. [18]. It uses list scheduling and transformations such as speculative code motions. This system was mainly targeted towards multimedia and image processing applications.

## 2.8 Harvard vs Von Neuman architectures

A Harvard architecture is a computer architecture where the instructions of the program are stored separately from the data; whereas in a Von Neuman architecture, the program and the data share the same storage device.

The implication of using a Von Neuman architecture is that the processor can either read an instruction word or a data word at any given point of time. Since the program and the data share the same bus, there is a potential of saving resources at the expense of performance.

A Von Neuman architecture allows for self modifying programs, since the memory used to store the instructions and the data must be a read/write memory. Whereas in a Haravard architecture, the program could be stored in a read only memory in order to prevent the user from writing self modifying programs.

## 2.9 CISC vs RISC

A computer's architecture greatly depends on the Instruction Set Architecture (ISA) that defines its functionality. There are two contrasting ways of defining the functionality of a computer.

A Complex Instruction Set Computer (CISC) is a computer where a number of operations can be executed as part of a single instruction, and where instructions vary in format and in size. A CISC architecture usually contains a large number of instructions because it contains a lot of specialised instructions.

A Reduced Instruction Set Computer (RISC) is a computer where instructions perform very simple operations. In a RISC architecture, all instructions have the same width, and adhere to a common format, but most importantly execute in equal clock cycles.

The difference between a CISC computer and a RISC computer is that a RISC architecture requires more instructions to implement a certain program than what a CISC

architecture requires. The CISC philosophy shifts the complexity from software to hardware because of the assumption that hardware is always faster than software.

## 2.10 Application Specific Instruction-set Processors

Digital Signal Processing designers have been exploring the capabilities of ASIPs (Application Specific Instruction-Set Processors) as opposed to DSPs and ASICs/FPGAs.

ASIPs are processors of configurable instruction set, with a fixed basic instruction set architecture for minimum operation. This kind of technology couples the flexibility of a general purpose processor with the high performance of an ASIC, which reduces the cost of implementation for DSP systems that need frequent updating.

Reconfigurable processor methodology has had a better success rate at being accommodated into the design cycle than traditional high-level synthesis mainly because the underlying base processor architectures target specific, well defined application domains by providing base functional units optimised for the target application while allowing the end user to extend the base instruction set to enable the use of accelerated custom logic at the hardware level.

The ASIP design methodology tackles the communication overhead between the CPUs and coprocessors that implement the custom functionality by integrating the custom hardware and associated instruction set with the base implementation of the processor.

The Tensilica Instruction Extension (TIE) language [15], can be used to describe a restricted set of processor extensions for the Xtensa processor generator. The key advantage of using such an abstraction is that the RTL descriptions and the software tools extensions can be inferred from the definition. Goodwin and Petkov [16] describe a

framework that extends the base TIE ISA with VLIW instructions, vector operations and vector register files, which can be automatically recognised by the compiler, and therefore require no change in the application.

Research into automated design of ASIPs has primarily focused on the instruction set optimisation space, which covers both the structure of instructions (aiming to satisfy orthogonality, regularity and completeness characteristics) and finding the optimal encoding (which is crucial to minimising the instruction decode logic)

Further research has looked at applying more specific optimisation techniques, such as hardware pipelined execution of loops [49]. The technique was tested by extending the OpenRISC processor where a speedup of 3.1X over pure software execution was achieved.

Further optimisations can be achieved by overcoming the data bandwidth limitations between the base processor and the custom logic as shown in [2]. In this approach an ILP model is employed to integrate the data bandwidth information and the data transfer costs into the design space exploration of the instruction-set extension.

## 2.11 Discussion

Almost all research on behavioural synthesis has focussed on synthesis of behavioural descriptions that contain a single "process", while a multiprocess system was handled by synthesizing each process separately.

A new methodology, proposed by Wang et al. [47], adapts existing high-level synthesis tools to optimise multiprocess descriptions. This technique does not change the existing high-level synthesis tools. Instead, it carries out multiprocess performance analysis in order to identify critical and near-critical operations and to partition global resources

into constraints for each process; then feeds this information to the high-level synthesis tools along with the behavioural description.

Since most of the behavioural synthesis systems generate data-paths and controllers, the idea of generating a special set of processors, all on a single FPGA or an ASIC, is worth exploring and evaluating.

Converting behavioural descriptions into an equivalent set of Nano-Processors may reduce the complexity of generating the final RTL description. Of course, a small part of the behavioural description which runs a single operation such as an ADD or logic AND might be better synthesised using the traditional approach than generating a whole Nano-Processor for it. But a careful mix of the two approaches can hypothetically present the best of both approaches.

A study conducted by Gorjiara and Gajski [17] explored such a hypothesis by using a bottom-up datapath trimming technique. The approach starts from an initial general processor datapath and reduces the architecture down to a minimal datapath by removing redundant functional units according to the input behaviour.

As part of the study a comparison of using the trimming technique on two general purpose architectures (VLIW and DLX) was carried out. The authors report that the DLX type of architecture was more efficient in all fronts (area, performance, and power consumption) despite having a higher cycle count compared to the VLIW.

In addition to that, the authors compared the trimming methodology on three types of custom baseline datapaths:

1) A base datapath where resource sharing is enabled by connecting all inputs of functional units to all RF read ports and constants, and the outputs of all functional units

to all write ports of the RFs.

2) The base datapath in (1) as a pipelined architecture

3) the datapath in (2) using a bypassing architecture.

The results of the study show that the 3rd datapath type where bypassing (forwarding) is employed is the most effective of the three types of architecture in optimising performance and energy consumption.

The approach presented in this project resembles that in [17], however, instead of trimming, the processor and instruction sets are composed as a best fit of the resource requirements captured from the behavioural input.

# Chapter 3

# Behavioural Synthesis into Nano-Processors

## 3.1 The Nano-Processor Concept

The behavioural synthesis approach, being researched in this project, makes use of the hierarchical nature of the VHDL grammar.

The behavioural subset of VHDL is contained inside process statements. VHDL process statements can be used to describe part or the whole of a system's behaviour as a sequence of VHDL statements. The processes themselves are concurrent statements.

The way high-level descriptions are organised in VHDL makes a fetch-execute instruction set processor structure a straightforward hardware representation for each VHDL process. Each processor structure would need a sequence of hardware instructions to be stored in a program memory. The content of the program memory would represent the equivalent of the original set of sequential statements within the corresponding VHDL

process. These fetch-execute processor structures will be referred to as Nano-Processors throughout the remainder of the report.

The Nano-processor structures should be capable of implementing all logical and arithmetic VHDL operators. A nano-processor should provide the capability of moving data between the different storage units (Register file, Memory, Instruction Register ... etc). A nano-processor can be as capable as a simplistic general purpose microprocessor if the behaviour requires such functionality.

The most important feature of this approach is that every nano-processor structure generated from a VHDL process must only implement the functionality required by the originating behaviour. In other words, each nano-processor structure would only implement a unique instruction set architecture that is an adequate representation of the originating behaviour. Figure 3.1 demonstrates an example VHDL code and a simple form of the corresponding Nano-Processors.

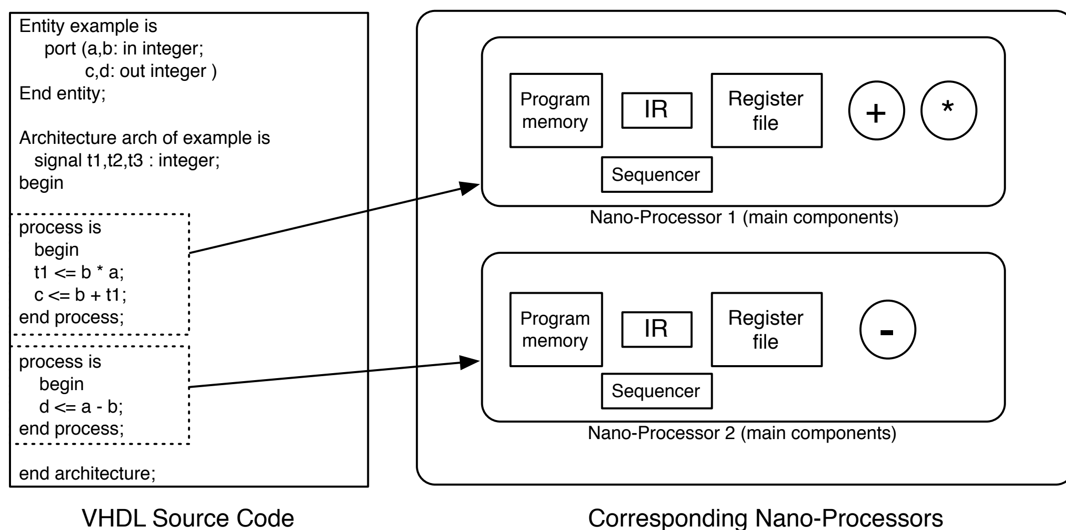


Figure 3.1: Example VHDL code and corresponding Nano-Processors

## 3.2 The Behavioural Synthesis Process

There are many possible implementations of such a behavioural synthesis approach. The strategy taken to implement the system in this project is encapsulated in the flow chart shown in Figure 3.2 and described below.

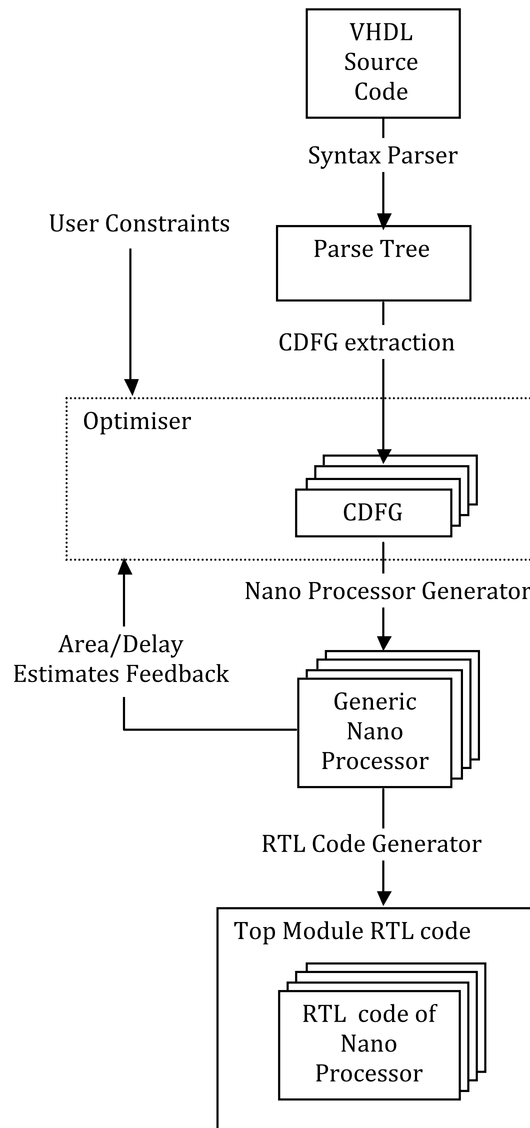


FIGURE 3.2: Flow Chart of the proposed behavioural synthesis process

The input to the system would be VHDL source code describing the behaviour of a circuit. The output of the system would be synthesisable RTL VHDL code of the set of automatically generated Nano-processor structures.

The Nano-processor structures would be wrapped within one single top level entity that defines the external interface to the whole circuit.

## 3.3 VHDL as The Input Language

There are a number of reasons behind the choice of VHDL as the input language of behavioural descriptions in this project:

- The MOODS behavioural synthesis tool, which is used as a test case to compare against, compiles behavioural VHDL descriptions.
- There exist a vast number of digital systems described in the VHDL language, which can be used to test the proposed system.
- A large number of designers are already familiar with VHDL as opposed to new HDLs, such as SystemC, which does not have a wide acceptance among designers.
- A large number of the developed behavioural synthesis tools (other than MOODS) accept VHDL as input, which allows for more comparisons.
- Using VHDL allows a straightforward translation of each VHDL process into a Nano-Processor structure.

## 3.4 Source Code Analysis

The first step in the behavioural synthesis process is to transform the human readable input description into a machine readable tree representation, structured according to the VHDL syntax.

In compiler design, such a tree structure is referred to as the Parse Tree. The parse tree is to be used in the next step of the process in order to extract the information necessary for synthesis, from the source code. All language specific constructs that have no importance to the synthesis process will be ignored.

VHDL constructs of importance to the synthesis process include and are not exclusive to:

- The top entity (ports, generics) would define the external interface to the circuit
- The global signals and variables that may be shared by processes, would define the inter-process communication structure, and any shared storage or interconnect resources.
- The process statements and sequential statements within, would define the architecture of instruction-set Nano-processors.

Figure 3.3 shows an example VHDL code and the corresponding parse tree.

## 3.5 The intermediate representation

The intermediate representation for this system would be a Control and Data Flow Graph. The CDFG would capture the data dependencies and flow extracted from the sequential statements within the VHDL process. Figure 3.4 shows an example CDFG.

This graph representation makes it possible to modify the original CDFG, while preserving the behaviour. The modifications allow the system to explore possible schedules of operations, identify concurrent operations and reduce redundancies.

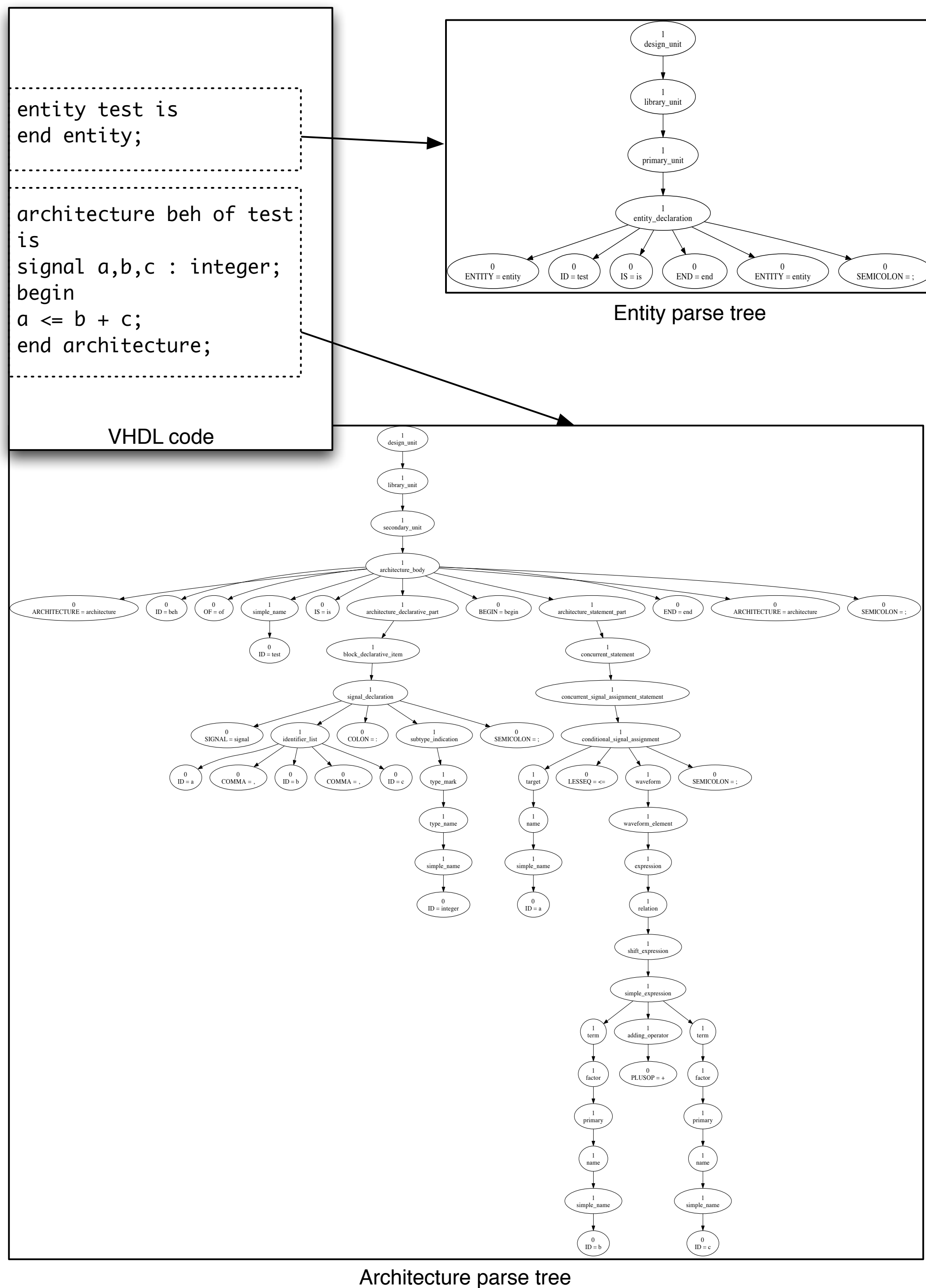


Figure 3.3: VHDL code and corresponding parse tree

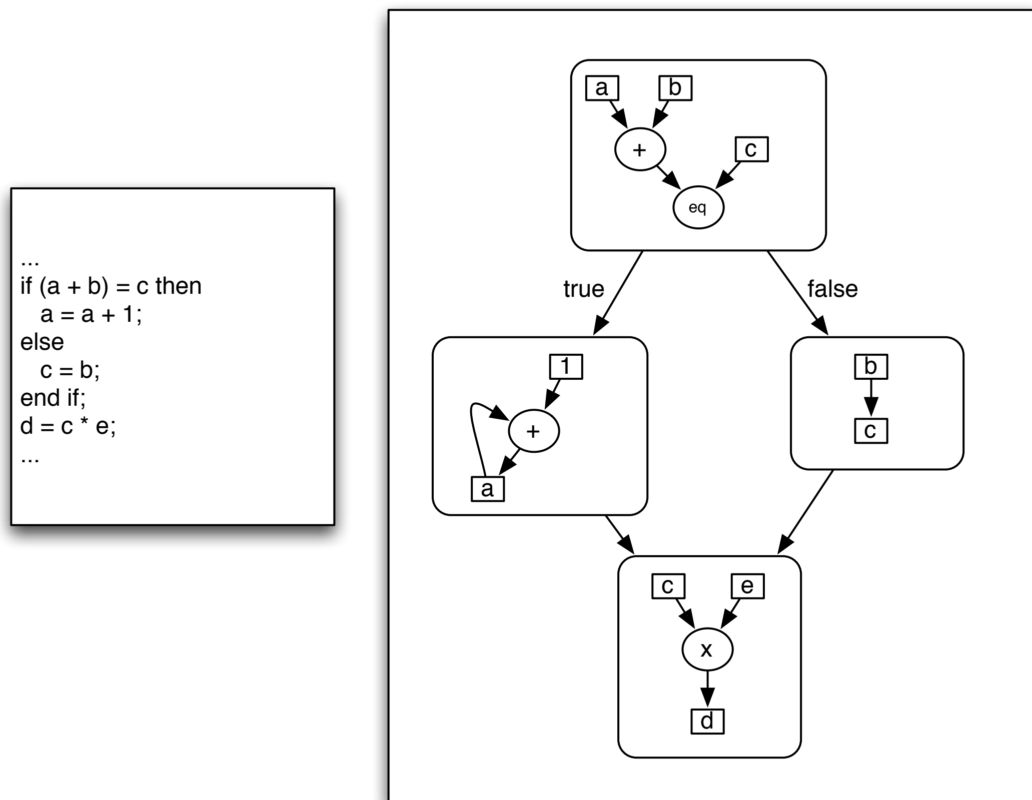


FIGURE 3.4: VHDL code with corresponding Control and data flow graph

## 3.6 Generating Nano-processor structures

Once a schedule of the Control and Data Flow Graph has been defined, the CDFG is transformed into an in-memory Nano-Processor representation. This object representation of the Nano-Processor holds all the necessary information about the components of the nano-prcoessor including the content of the instruction memory.

No matter how the structure of the in-memory representation of the Nano-Processor is defined, the interface to this representation must be the same. Such an interface needs to allow access to the information about the kind and number of components required to implement the nano-processor, the characteristics of those components and their content when required.

## 3.7 Generating RTL description of the circuit of Nano-processors

The processor structure objects will be contained inside a top module object which contains information about the external interface and the global data shared between the different processor structure objects.

At this point, the output of the behavioural synthesis tool is generated according to the top module's contents, and each of the generic nano processor structure objects is used to generate its own unique RTL VHDL code of a Nano-Processor.

## 3.8 Example Transformation

This section gives an example of the desired transformation from a behavioural description to a RTL description. For the purpose of clarity, a very basic example is given.

Three different VHDL processes should be translated into a set of nano-processors. The RTL code for the synthesis result is not given but a graphical view can be seen in Figure 3.5.

The VHDL code is given below. We can see that the processes communicate and share a resource.

```
entity test is

end entity test;

architecture behaviour of test is
    signal a: natural range 0 to 15 ;
    signal c: bit_vector(4 downto 0) := "10010";
    begin
        K: process is
            variable b: natural range 0 to 15 := 5;
            begin
                a <= b;
                b := b + 1;
        end process;
```

```
        L: process is
            variable d: bit_vector(4 downto 0) := "10011";
            begin
                 c <= d and c;
        end process;

        M: process (a) is
            variable e: natural range 0 to 15 := 2;
            begin
                e := e * a;
        end process;

end architecture behaviour;
```

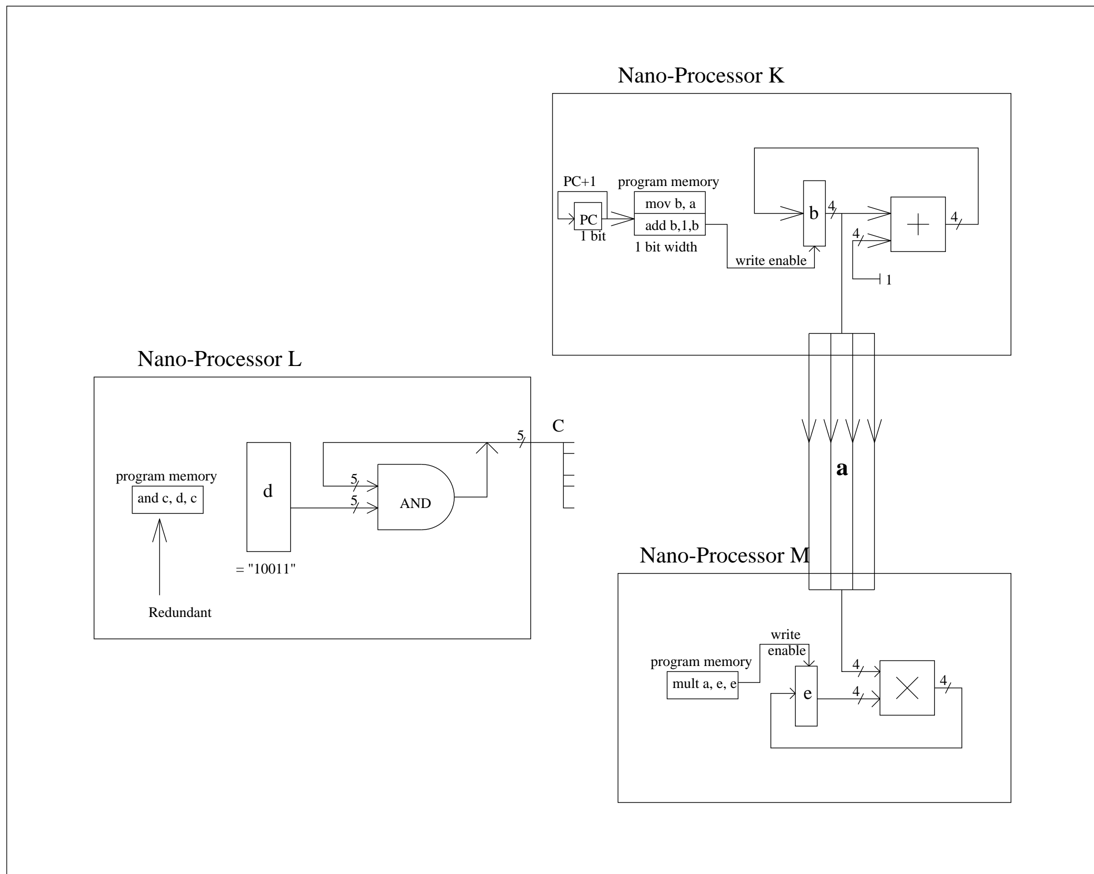


FIGURE 3.5: Block Diagram of the Example Nano-Processor Implementation

The first process (K) will generate a nano-processor that executes only two assembly instructions: [Mov b, a] and [add b, 1, b].

After analysis, we find that we only need one register in the register bank to hold the value of the variable b. We only need one adder (4 bits wide). The program counter

goes from 0 to 1, and the assembly instructions need only be 1 bit wide, because there is only one signal to control and no addressing is required here.

The second process (L) generates a nano-processor that executes only one instruction. Therefore, the presence of program memory is redundant. Variable d is allocated a register, and the signal c is connected to the logic AND gate along with d.

The third process (M) also generates a nano-processor that executes only one instruction. The program memory here is redundant. Only a multiplier is needed here, and variable e is allocated to a register.

# Chapter 4

# Translating a Behavioural Description into a Parse Tree

The first task in building the synthesis tool is to convert the behavioural description from VHDL source code into a hierarchical structure that captures the meaning of the source code as a parse tree.

In order to implement this conversion, a compiler environment needs to be built. This chapter outlines the status of the front-end of the behavioural synthesis tool.

## 4.1 VHDL Compiler Environment

VHDL Compiler Environment (VCE) is the term that refers to the behavioural synthesis tool's front-end. VCE is the implementation of a VHDL compiler which reads source code files written in behavioural VHDL and generates a set of parse trees which are in turn transformed into an intermediate representation.

VCE was developed using the C++ programming language. The C++ Standard Template Library was used in conjunction with the open source STLplus Collection, as a basis for most of the data structures created to achieve the desired implementation. The parts of VCE that have been completed are described in the following sections.

## 4.2 Lexical Analyser

A user of the synthesis tool writes a number of files that describe the behaviour of the system using a behavioural subset of the VHDL language.

The first step in the compilation process is to convert the source code in the files into a stream of tokens. A token is typically a string of characters that represent a single atomic unit of the language (identifiers, keywords, numbers and operators are some examples of tokens).

The lexical analyser implemented in this project was built according to the VHDL lexical notation as described in the VHDL Language Reference Manual [1].

## 4.3 Syntax Parser

A number of parsing approaches have been evaluated before attempting to build the parser.

There exists a number of Parser Generators (known as Parser Parsers) that read an E-BNF syntax description of a language, and automatically generate code that parses files written in that language.

Examples of these parser generators are: the Bison (previously known as YACC) bottom-up or LR parser generator [11], and the PCCTS top-down or LL parser generator [35] (now known as ANTLR [34]).

Although these parser generators are known to be efficient when dealing with simple and small languages, and give the advantage of avoiding having to create the parser manually, the use of such systems was rejected for the reasons below.

Bottom-up parser generators like Bison [11] are found to be unsuitable for the VHDL syntax. The number of reduce-reduce [1] conflicts that exist in the VHDL syntax is large enough to break the original VHDL semantics if those conflicts were to be eliminated.

This shows that any VHDL parser must be a top-down parser in order to preserve the original VHDL semantics. A recent attempt at using Bison to compile the entire VHDL language, by Lorenc et al. [30], confirms this point.

When considering parsing VHDL with a top-down parser generator, an indirect left recursion in the production of the *Name* rule has been eliminated. The following shows the original production of the *Name* rule.

```
Name ::=          simple_name
                | operator_symbol
                | selected_name
                | indexed_name
                | slice_name
                | attribute_name


Selected_name   ::=     Prefix.suffix
Indexed_name    ::=     Prefix ( expression {, expression} )
Slice_name      ::=     Prefix ( discrete_range )
Attribute_name  ::=     Prefix [signature ]' attribute_designator
```

[1] A Reduce-Reduce error is caused when a grammar allows two or more different rules to be reduced at the same time, for the same token. Such a grammar is ambiguous since a program can be interpreted more than one way.

```
Prefix ::=      Name | Function_call

Function_call ::=       Name [(actual_parameter_part)]
```

Because the *Prefix* is also a *Name* and is the *first* symbol in the production of *Name*, there exists an indirect left recursion. The production for the *Name* rule has been modified into the following in order to eliminate this left recursion:

```
Name    ::=       simple_name Rest_of_name
                | operator_symbol Rest_of_name

Rest_of_name ::=  [(actual_parameter_part)] .suffix rest_of_name
      | [(actual_parameter_part)] ( expression {, expression} ) rest_of_name
      | [(actual_parameter_part)] ( discrete_range ) rest_of_name
      | [(actual_parameter_part)] [signature ]' attribute_designator rest_of_name
      | 'blank'
```

After careful analysis of the VHDL syntax, it was clear that VHDL is not a language that can be parsed in a conventional way. The rules governing the use of *Names* in VHDL meant that it is impossible to parse a *Name* without the availability of semantic information.

The problem happens when the parser reaches a stream of symbols of the form:

```
Identifier1(Identifier2)
```

Without the semantic information about the identifier *Identifier1*, it is impossible to decide whether this is a case of a function call, an indexed name, a slice name or a type conversion. However if we know what *Identifier1* refers to, the meaning of the whole name will be known.

The ANTLR parser generator [34] deals with this issue by providing *predicates* which let the programmer systematically direct the parse via arbitrary expressions using semantic and syntactic context. In the previous example, predicates can be added to the rules that govern the use of *Names* which basically check the meaning of the first identifier and decide what the next step will be.

Although parser generators are powerful language tools, a hand written parser gives the advantage of increased flexibility, better error handling, and ease of debugging.

Within the MOODS behavioural synthesis tool [7], there exists the source code for a working top-down VHDL parser and library manager. However, this parser does not completely adhere to the standard VHDL. Therefore, the decision was made to create a new recursive descent VHDL parser with a different parsing approach.

### 4.3.1 Parsing using a hypothesis tree

At the core of the new parser is an algorithm that performs the recursive descent parse in an automatic manner, rather than the traditional approach of using recursive function calls as in the MOODS parser.

The algorithm takes two inputs:

1. A grammar definition graph (which may be hierarchically defined). The nodes of this graph contain lexical tokens - any *legitimate* sequence of input tokens corresponds to a path through the graph.
2. A linear sequence (linked list) of lexical tokens corresponding to the user specified input.

The algorithm functions as follows: (Figure 4.1 gives an example)

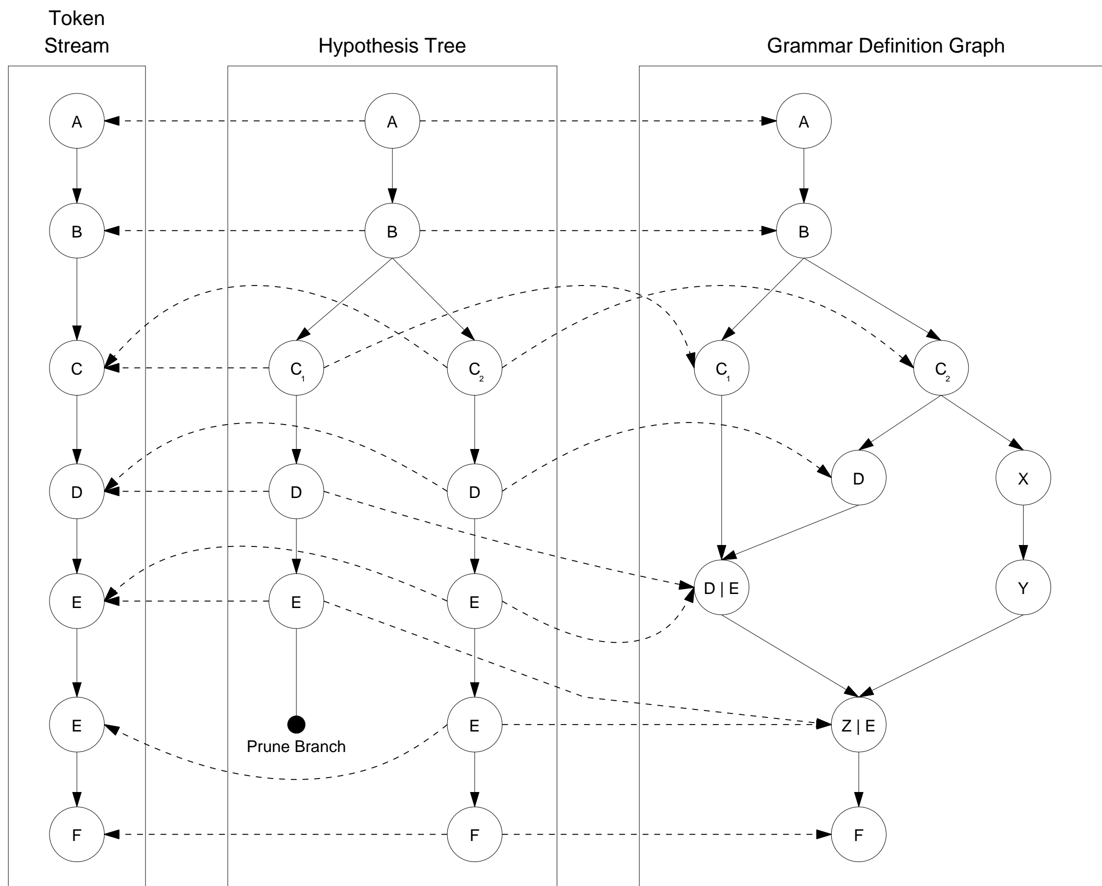


FIGURE 4.1: Example Structure and Use of the Hypothesis Tree

Tokens from the input stream are mapped onto nodes of the definition graph. Where multiple mappings are possible (input token C in Figure 4.1 is an example).

This process is supported by the creation of a *Hypothesis Tree*, that keeps track of all possible *legitimate* mappings as the token stream is processed.

As processing continues, *each* branch of the hypothesis tree is extended. If it is not possible to extend a given branch, (corresponds to an illegal token sequence) the entire hypothesis branch is pruned back to its originating branch point.

At the end of processing, one of the following situations will occur:

- The hypothesis tree is actually a linear list: The input sequence was valid.
- The hypothesis tree is empty: The input sequence was invalid.

- The hypothesis tree contains multiple branches: The input sequence was valid, but ambiguous (which is defined to be *illegal* in the VHDL standard).

**Formal description of the algorithm** Inside a loop, *for* every leaf node of the hypothesis tree, the algorithm:

1. reads the token, **NT**, next to the leaf's corresponding token
2. searches for the *Candidate Terminals* from the syntax graph, that are immediately next to the leaf's corresponding Terminal
3. selects the Candidate Terminals, **MT**, that match the next token, NT.
4. *for* every matching Terminal, MT:
   - the hypothesis tree is extended, by appending a node containing NT and MT onto the current leaf node
5. subsequently the hypothesis tree may grow in width (many possible branches)
6. for a given leaf, if there was no possible match, the branch containing that leaf will be pruned (deleted) from the hypothesis tree

The hypothesis tree would shrink down to a list (a single branch) at the end of the parse. An error is issued if the hypothesis tree could not be extended for all leaf nodes or the hypothesis tree has more than one leaf node at the end of the parse.

## 4.4 VHDL Syntax:

In order to use the parsing algorithm described above, the VHDL syntax must be represented as a graph data structure. The nodes of this graph are either Terminals (tokens) or Non-Terminals (production rules).

A program has been built that reads a file containing the E-BNF description of the VHDL syntax, and automatically creates the graph data structure of all the syntax rules.

Each node of the graph that is a Non-Terminal points to the graph object of the corresponding syntax rule. This makes it simple to query a node in the graph for the Terminal nodes that are immediately next to it.

The automatic building of a graph representation of the syntax was inspired from the approach used by D. Crookes [8]. Further details can be found in Appendix A

## 4.5 Semantic Analysis:

Once the algorithm reaches a syntax rule of a *Name*, the semantic information (about identifiers and operator symbols that compose that *Name*) is necessary in order to disambiguate between possible meanings of the *Name*. This is achieved by calling a special function that parses the *Name* while querying the symbol table for the actual meaning of each identifier or operator symbol.

For this to work, all the declarations in the current VHDL file result in adding the declared identifier or operator symbol into the symbol table while parsing. This has

been achieved by calling special declaration parsing functions for each declaration syntax rule.

Semantic analysis for VHDL is essential because the syntax rules governing *Name* cannot be parsed even with infinite look-ahead tokens.

For example, in the absence of semantic information, using the algorithm described, Name(Name) will extend the hypothesis tree into 4 branches and will not be pruned even when the end of file is reached.

As mentioned earlier, Name(Name) can either be: a slice name, an indexed name, a function call or a type conversion. If another Name(Name) is encountered, the hypothesis tree will extend into 16 branches (4 * 4) and will not be pruned either.

The more of these ambiguous sequences of tokens, the bigger the hypothesis tree will extend without any pruning. Once the hypothesis tree is very wide, (hundreds of branches) the algorithm takes longer and longer time to process the whole file.

The implemented semantic analysis of Name(Name) results in only one possible extension of the hypothesis tree, and therefore eliminates the problem.

# Chapter 5

# Synthesis into Fetch-Execute Structures

## 5.1 Parse Tree to Control & Data Flow Graph

Using the parse tree as a starting point, Control and Data Flow graphs are extracted using a CDFG extraction mechanism.

The first step in this mechanism is the simplification of the parse tree into a stripped down version that only contains the necessary information about the structure of the circuit and its behaviour.

The behavioural part in the parse tree is then used to extract the control flow and within each control part the data flow.

Each VHDL sequential statement is considered a control part. Expressions are the parts that provide the data flow, which can be found in conditions of IF/While statements or in the right hand side of a signal/variable assignment.

The CDFG proposed is a graph of different types of nodes: one type is the control node, which captures the sequence of statements. The other type is a data node which represents variables, constants and signals. Another type of node is the operator node, which represents the operators or procedure calls. Figure 5.1 shows an example CDFG.

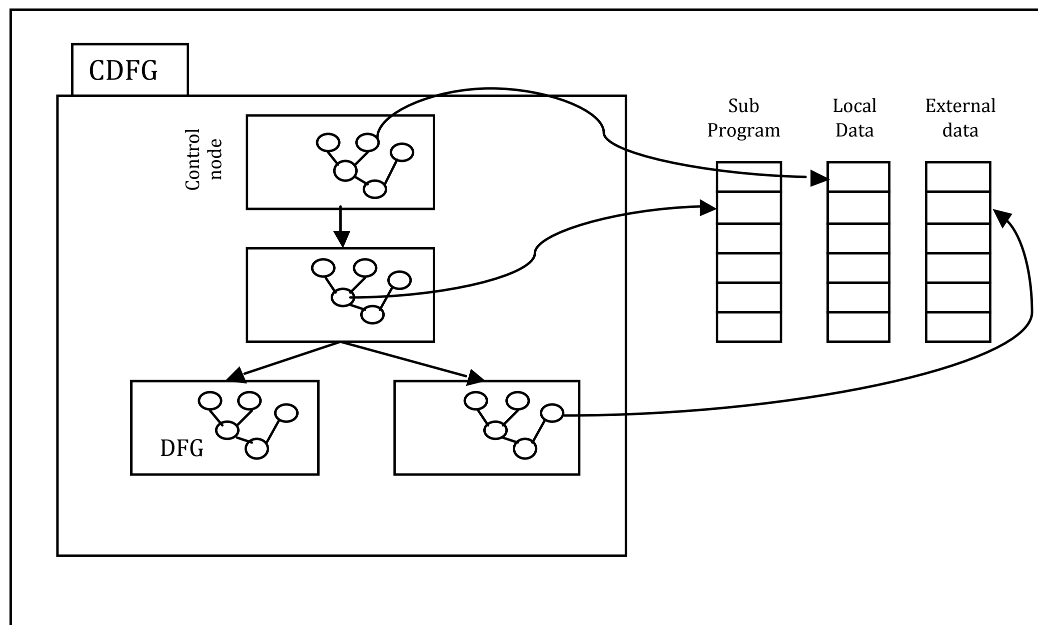


Figure 5.1: Example CDFG extracted from a parse tree

## 5.2 CDFG to Processor Objects

Each process statement from the parse tree is converted into an object form, stored in memory. This form captures all the information essential to generating a corresponding RTL description.

The processor object contains a Control Graph, which is a set of control nodes interconnected based on the sequence of statements originating from the behavioural description.

Each control node points to a Data Flow Graph, which captures the data dependency between the Data nodes and the Operator nodes.

Operator nodes can be a call to a procedure or function's CDFG. Therefore, as well as storing references to the Data nodes, the processor object also stores references to CDFGs of function or procedure definitions that have been called.

Figure 5.2 shows a simplistic view of the object form of a top module, containing two different processor objects. Each processor object is composed of different CDFG that captures the flow of control and data dependency extracted from the behavioural description.

## 5.3 Processor Objects to RTL description

### 5.3.1 Nano-Processor Template

This section describes the generic design of the nano-processor concept. The structure and instruction set of a nano-processor are automatically defined by the behavioural synthesis tool.

The synthesis tool binds each VHDL process with a nano-processor as a one-to-one mapping.

The statements of a VHDL process define what the instruction set and the functional units of the corresponding nano-processor would be. Hence, different VHDL processes are mapped to different nano-processors.

Figure 5.3 depicts a template for such nano-processors. This template has the sole purpose of defining what a nano-processor might be composed of, and how it is organized.

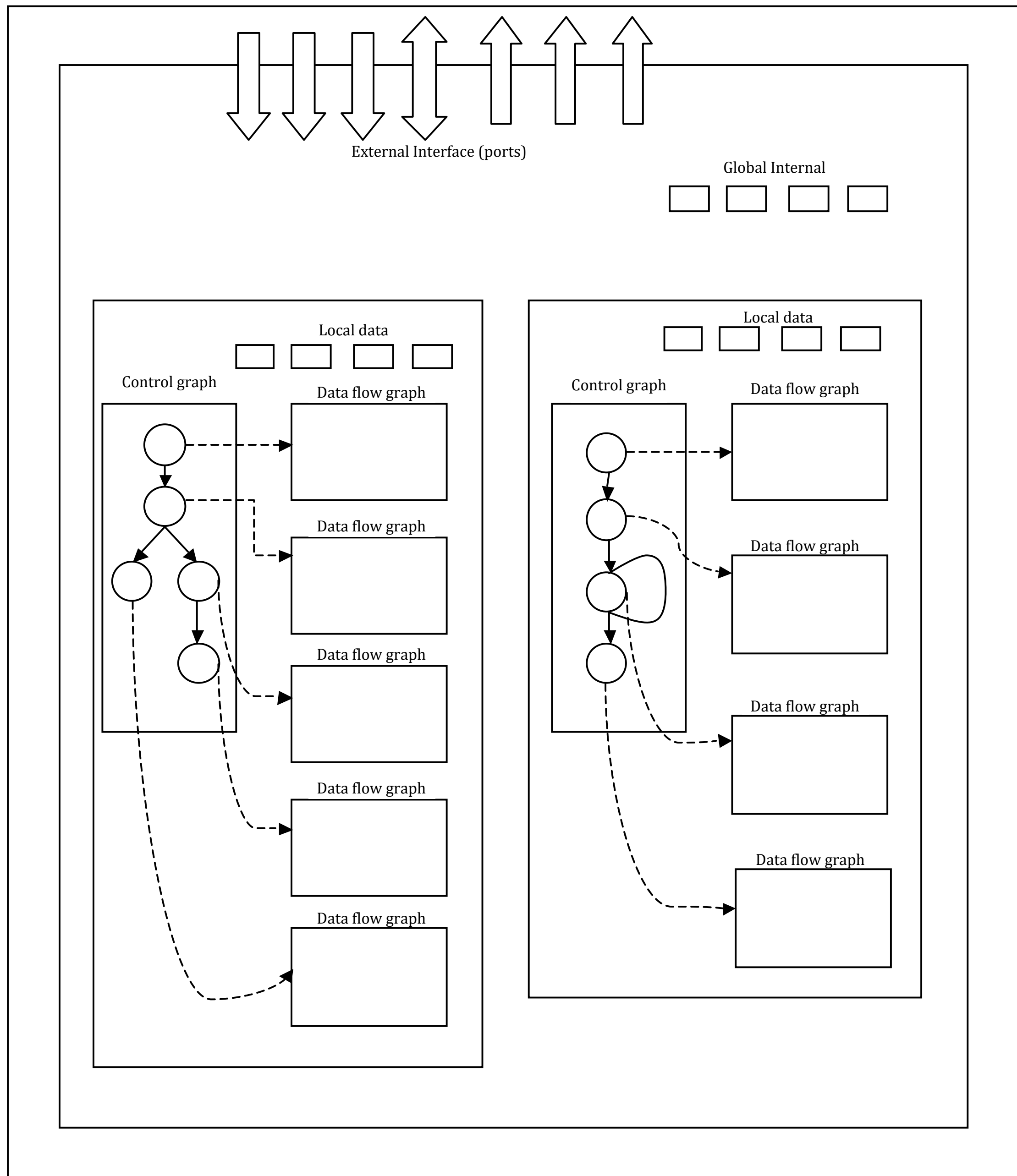


FIGURE 5.2: A sample object representation of a circuit containing two processor objects

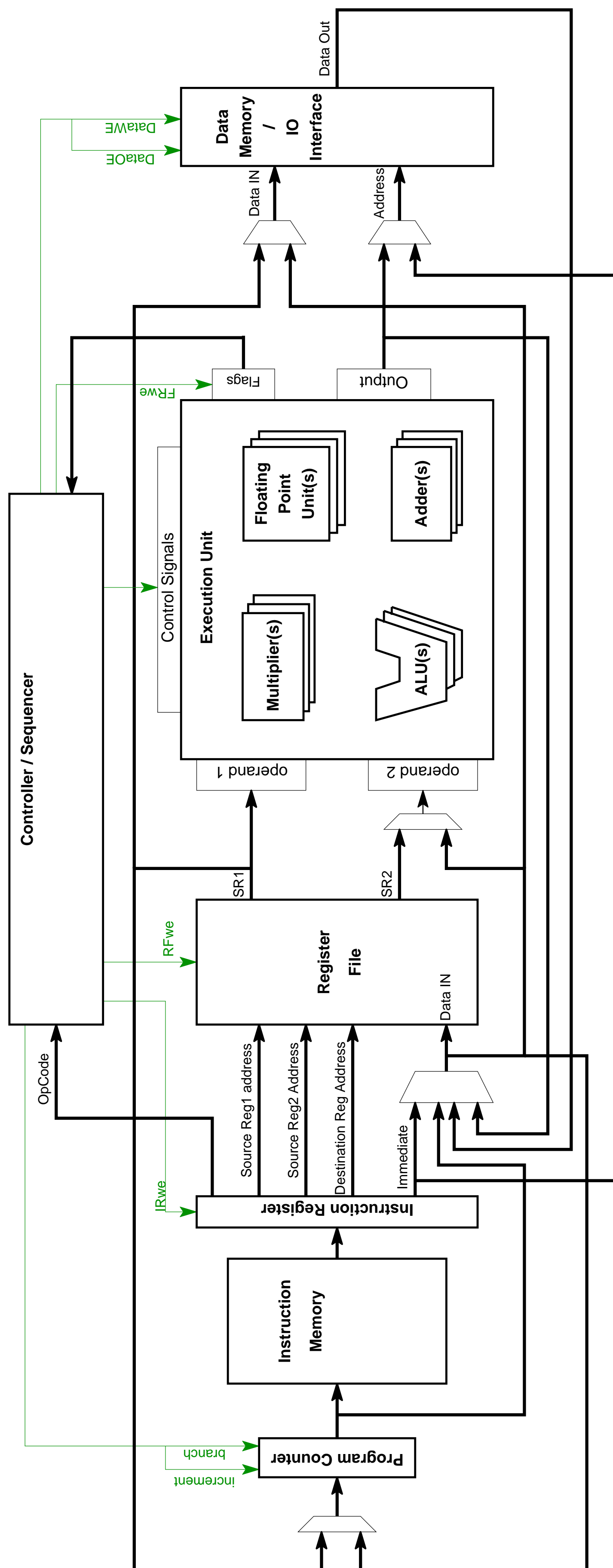


FIGURE 5.3: Template of a Nano-Processor

It is not necessary that all nano-processors contain all the components of the template. The bit widths of the various signals in the template are not shown because they are unknown, until defined for every nano-processor that implements a VHDL process.

**Instruction Memory:** All nano-processors will possess an Instruction Memory. The instruction memory stores the assembly code instructions that represent the corresponding VHDL process statements. At present, the instruction memory is defined as a Read-Only-Memory, which means it contains a fixed micro-code. However, there is no objection to using a Random-Access-Memory to allow for self-modifying programs.

**The Program Counter** will be present in all nano-processors that have more than one instruction in the instruction memory. The program counter is a register that holds the address of the next instruction to be executed.

**The Instruction Register** stores the current instruction being executed by the nano-processor. Depending on the encoding of each nano-processors instruction set, the instruction register will be partitioned into Opcode part, Source Registers addresses, Destination Register address, Immediate part, and other parts as necessary.

The instruction register is connected to the different components of the nano-processor and to the sequencer in order to feed the type and data of the current instruction. These connections differ between nano-processors. All nano-processors have an instruction register.

**The Register File** is an addressable bank of registers. In general purpose microprocessors, these registers are the lowest level of the memory hierarchy. In most cases,

register files are used to read from one or two source registers at the same time, and write to one destination register at a time.

Given the addresses of the target registers, the register file decodes the addresses in order to select the appropriate registers for reading/writing. For writing to a register, a data port holds the data to be written.

For the nano-processor concept, the number and size of registers is not known, until the analysis of the variable types and sizes of the corresponding VHDL process is carried out in order to find the optimum number of registers and their width.

The registers in the register file are used to hold temporary values during instructions execution (such as input operands to a functional unit, the result of an operation, a base memory address for array indexing and so on). They can also be used to store scalar process variables that are frequently used, in order to reduce the number of memory calls.

**The Execution Unit** is the set of functional units that manipulate data. In general purpose microprocessors, the most basic execution unit consists of only an Arithmetic and Logic Unit (ALU).

More powerful processors include multipliers, integer units, floating point co-processors and so forth. In the nano-processor template, an execution unit can contain any number of the most common functional units (ALU, Adders, Multipliers, Floating Point Units).

A nano-processor contains only the functional units necessary to implement the corresponding VHDL process. An adder, if the process only has addition statements, or a multiplier, if the process has multiplication statements, and so forth.

**The Data Memory** is a Random Access Memory, which stores non-scalar variables such as VHDL arrays and records. The addressing system that was implemented for the nano-processor couples Data Memory addresses with Input/Output Interface addresses.

The I/O interface provides a nano-processor with the infrastructure to communicate with other nano-processors, or to connect to shared or global resources within the overall system.

**The Controller/Sequencer module** is the backbone of the nano-processor; it decodes the instruction and sets the appropriate control signals accordingly. The sequencer increments the program counter's address and sets it to an arbitrary value if there is a branch instruction.

### 5.3.2 Binding Processor Objects to Customisations of the Nano-Processor Template

The components of the nano-processor template were implemented using generic RTL VHDL entities so that they can be instantiated to specification if required as part of the synthesis of a Processor Object.

The sequencer/controller of the nano-processor is generated automatically by selecting, from a library of fetch-execute commands, the sequence of commands required to build the Finite State Machine which ultimately orchestrates the flow of data through the nano-processor components.

The byte code of the program memory is automatically generated by concatenating the representations of the different parts of each instruction together. These representations are generated into a Constants package and bound to actual bytes.

The top module linking all the components of the nano-processor together is the last to be generated, within which all the instantiations and binding of signals is described.

# Chapter 6

# Experiments and Results

This chapter captures the results of the experiments that have been carried out to evaluate the performance of the tool that implements the new behavioural synthesis approach.

## 6.1 Evaluation Strategy

In order to evaluate the behavioural synthesis tool that implements the new behavioural synthesis approach, a number of test cases have been used that describe behaviours of varying complexity and size (from single bit inputs up to 32 bit inputs).

The least complex test behaviour used was a logic AND operation, while an arithmetic multiplier was used as a medium complexity behaviour. The more complex behaviour was that of a Fibonacci number calculator.

The performance of the new approach in synthesising these behaviours was tested relative to the performance of the MOODS [7] behavioural synthesis tool. The output RTL representations of both MOODS and the Nano-Processor Synthesis tool were synthesised

using the XILINX ISE Design Suite onto the xc5vlx110t-3ff1136 FPGA as the target device.

Resource utilisation and delay measurements where extracted from the outcome of the synthesis process. The number of slice registers and the number of slice LUTs (Look-Up Tables) give an estimate of the silicon area required. While the speed of the resulting architecture is measured using the delay through the critical path.

The following sections describe the behaviours used in more detail as well as the evaluation results and a comparison to the MOODS performance.

## 6.2 Synthesis Results of the logical AND operator

Listing 6.1 captures the behavioural VHDL code of the logical AND operation used as a test case. Different versions of this code that describe different bit widths of the inputs to the AND gate were compiled and synthesised into a RTL representation by both MOODS and the Nano-Processor synthesis tool.

The RTL representations were then synthesised into the XILINX FPGA device using the XILINX ISE software.

```
entity and_entity is
    port (a, b: in bit_vector (7 downto 0);
                  c: out bit_vector (7 downto 0) );
end entity and_entity;

architecture behavior of and_entity is
        begin
      and_proc: process (a,b) is
                  variable aa,bb,cc,dd,ee,ff : bit_vector (7 downto 0);
        begin
                                        aa := a;
                                        bb := b;

                                        dd := aa and bb;
                                        ee := aa and dd;
                                        ff := bb and ee;
                                        ee := bb and dd;
                                        cc := ee and ff;

                                        c <= cc;
        end process and_proc;
```

```
end architecture behavior;
```

LISTING 6.1: VHDL code of an 8-bit logical AND operation

Figure 6.1 is a graph showing the difference in the usage of the FPGA's Slice Registers between the output of the Nano-Processor Synthesis tool (nano) and both outputs of the MOODS synthesis tool, the one optimised for area (moods a) and the other optimised for delay (moods d). While Figure 6.2 is a graph showing the difference in the usage of the FPGA's Look-Up-Tables (LUTS).

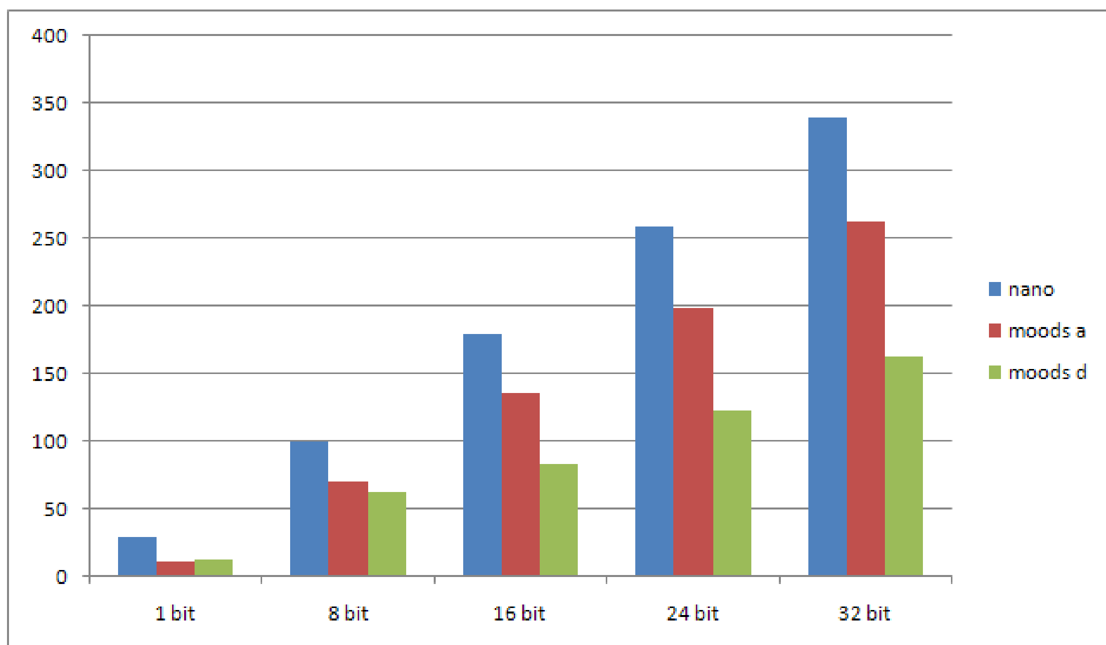


FIGURE 6.1: The number of Slice Registers used to implement the logical AND

The obvious observation is that the number of resources required increases linearly with the width of the variables, for both synthesis approaches.

Compared to MOODS, the nano-processor approach utilises more slice registers, and a greater number of slice LUTs.

Since the nano-processor approach builds a whole processor architecture which includes a Register File, a Program Memory and a Sequencer all connected by several interconnect

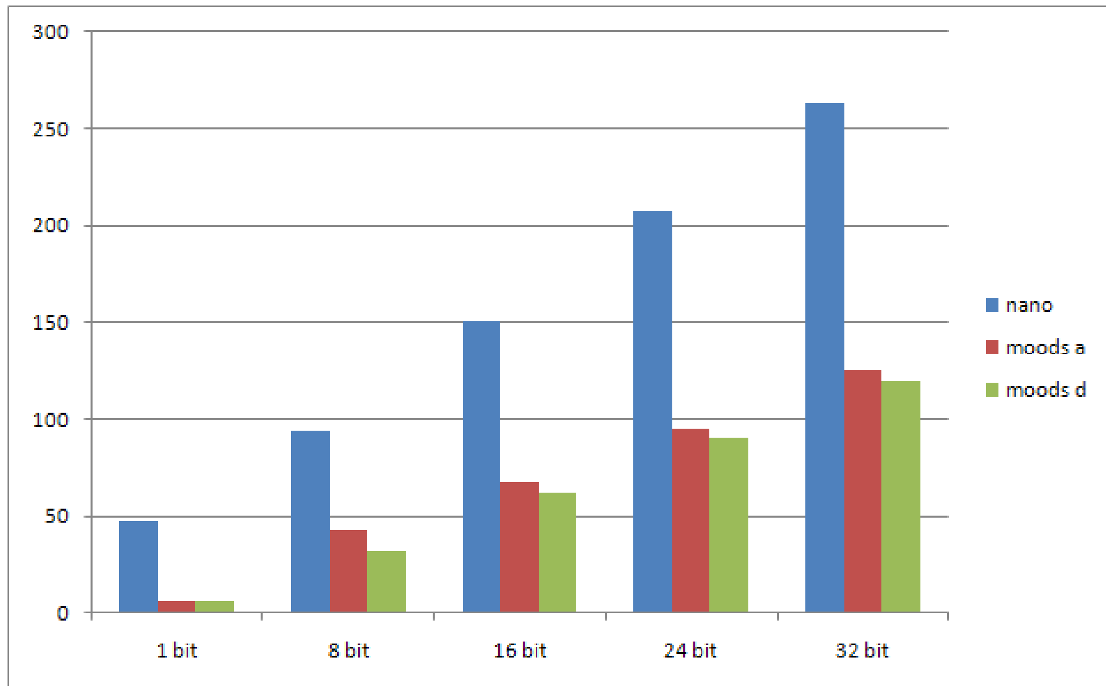


Figure 6.2: The number of LUTs used to implement the logical AND

points around the single AND gate; and because slice LUTS can be used as registers or for interconnect; this outcome was expected.

MOODS also uses optimisation algorithms which detect some redundancies in the behavioural code and as a result reduce the number of resources needed.

Figure 6.3 is graph showing the delay in nano seconds through the critical path of the output of each approach.

We can note that in this case the size of the variables has little effect on the delay in the critical path of the output of the nano-processor approach. This is because the generated architectures of the processor implementing the same behaviour use the same processor components, and therefore have the same critical path. In this case, the main component of the critical path is a simple, non-cascading AND gate, therefore the critical path is independent of the width of the variables.

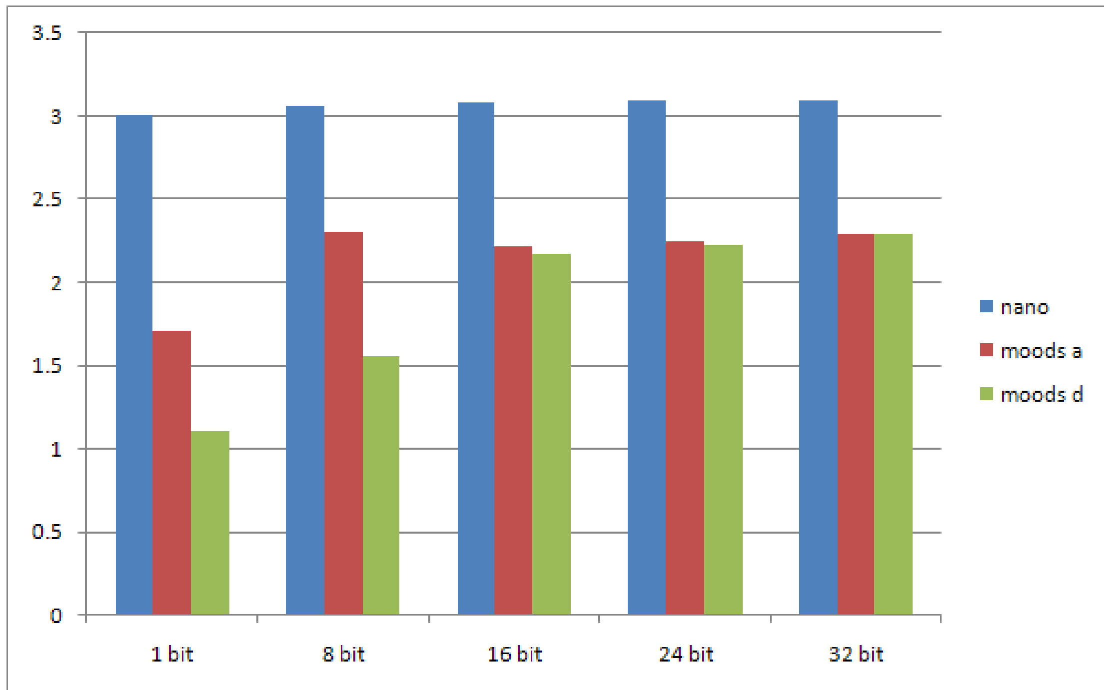


FIGURE 6.3: Delay in nano seconds through the critical path of the logical AND

We can note that the output from the nano-processor approach for this particular example is slower than that of the MOODS synthesis tool.

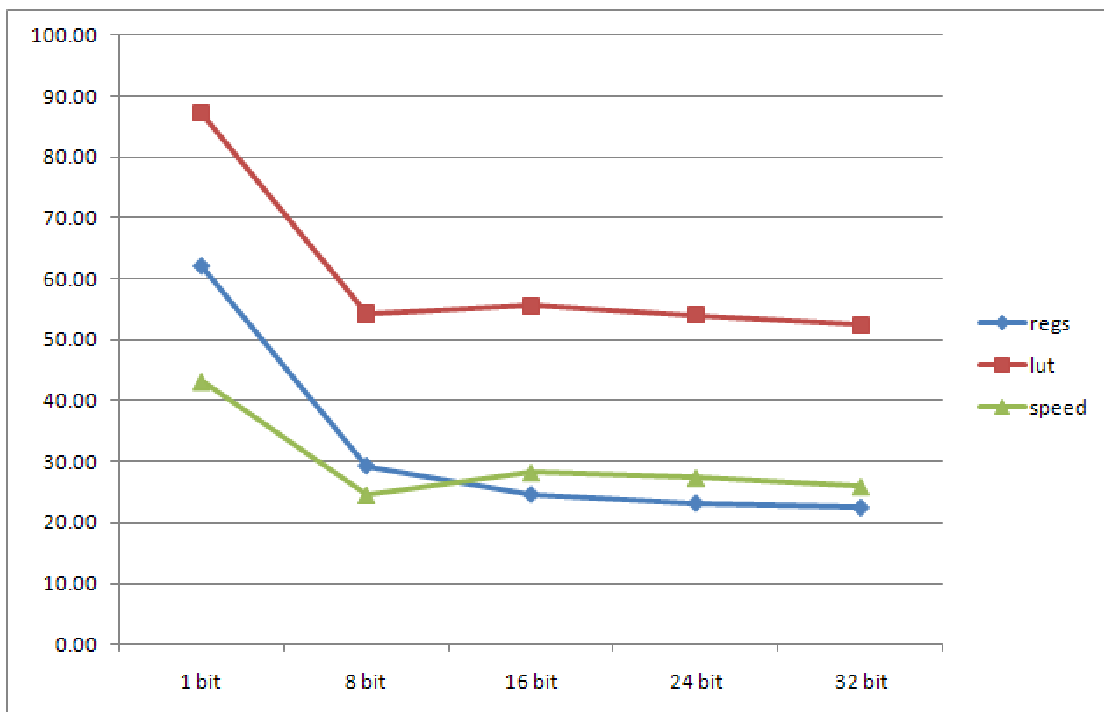


FIGURE 6.4: Percentage difference with MOODS output optimised for area

Figure 6.4 and Figure 6.5 depict the percentage difference in the measurements obtained

from the synthesis results between the output of the nano-processor approach and the output of the MOODS synthesis both optimised for area and optimised for delay.

We can see that for this particular case (the logical AND) the smaller the width of the variable the bigger the difference in resource utilisation is. In other words, the nano-processor approach uses far more resources compared to MOODS if the input behaviour uses smaller variables.

We can also note that the bigger the variables are, the less change in the percentage difference there is.

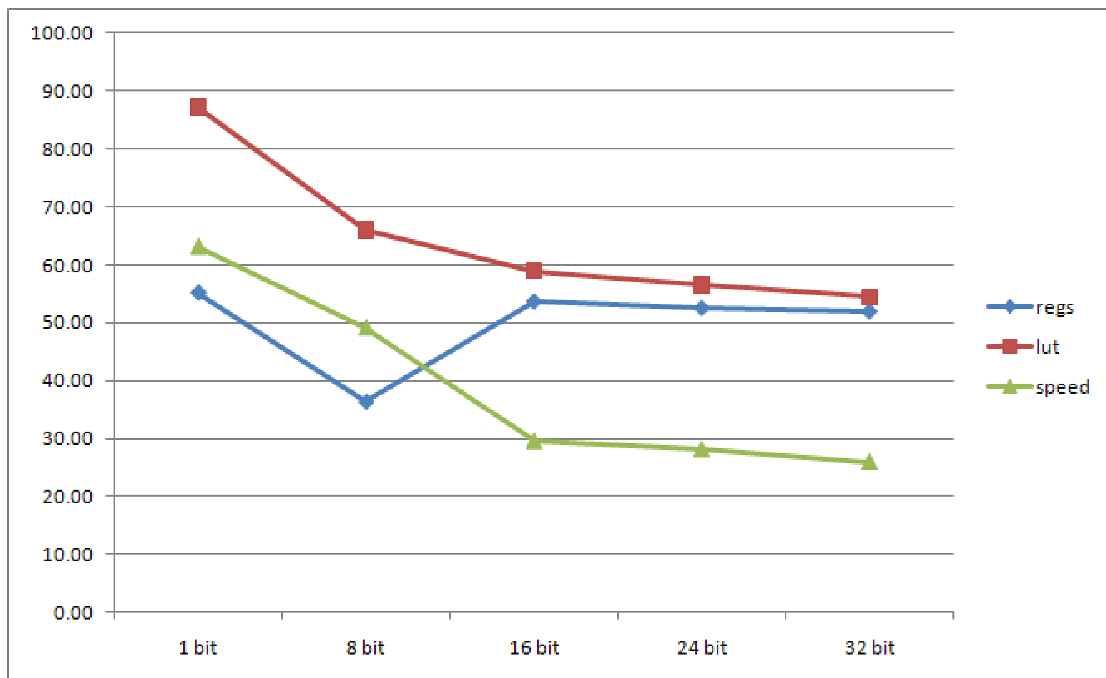


Figure 6.5: Percentage difference with MOODS output optimised for delay

## 6.3 Synthesis Results of the Arithmetic Multiplier

Listing 6.2 captures the behavioural VHDL code of the arithmetic multiplier used as a test case. Again, different versions of this code that describe different bit widths of the inputs to the multiplier were compiled and synthesised into a RTL representation by both MOODS and the Nano-Processor synthesis tool.

The RTL representations where then synthesised into the XILINX FPGA device using the XILINX ISE software.

```
entity mult_entity is
    port (a, b: in integer;
                  c: out integer );
end entity mult_entity;

architecture behavior of mult_entity is

begin

    mult_proc: process (number) is
                  variable factor : integer;
                  variable multipland : integer;
                  variable result: integer;
        begin

                factor := a;
                                        multipland := b;
                                        result := 0;

                                        while factor > 0 loop
                                                result := result + multipland;
                                                factor  := factor - 1;
                                        end loop;

                                        c <= result;
    end process mult_proc;
end architecture behavior;
```

Listing 6.2: VHDL code of an Arithmetic Multiplier

Figure 6.6 is a graph showing the difference in the usage of the FPGA's Slice Registers between the output of the Nano-Processor Synthesis tool (nano) and both outputs of the MOODS synthesis tool, the one optimised for area (moods a) and the other optimised for delay (moods d). While Figure 6.7 is a graph showing the difference in the usage of the FPGA's LUTS.

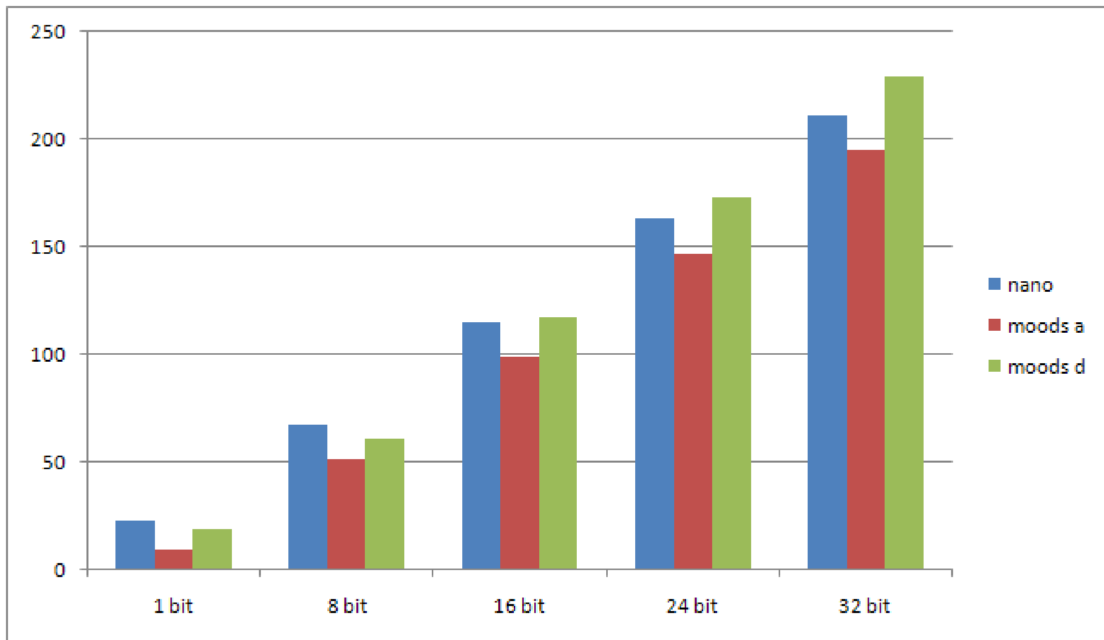


FIGURE 6.6: The number of Slice Registers used to implement the Arithmetic Multiplier

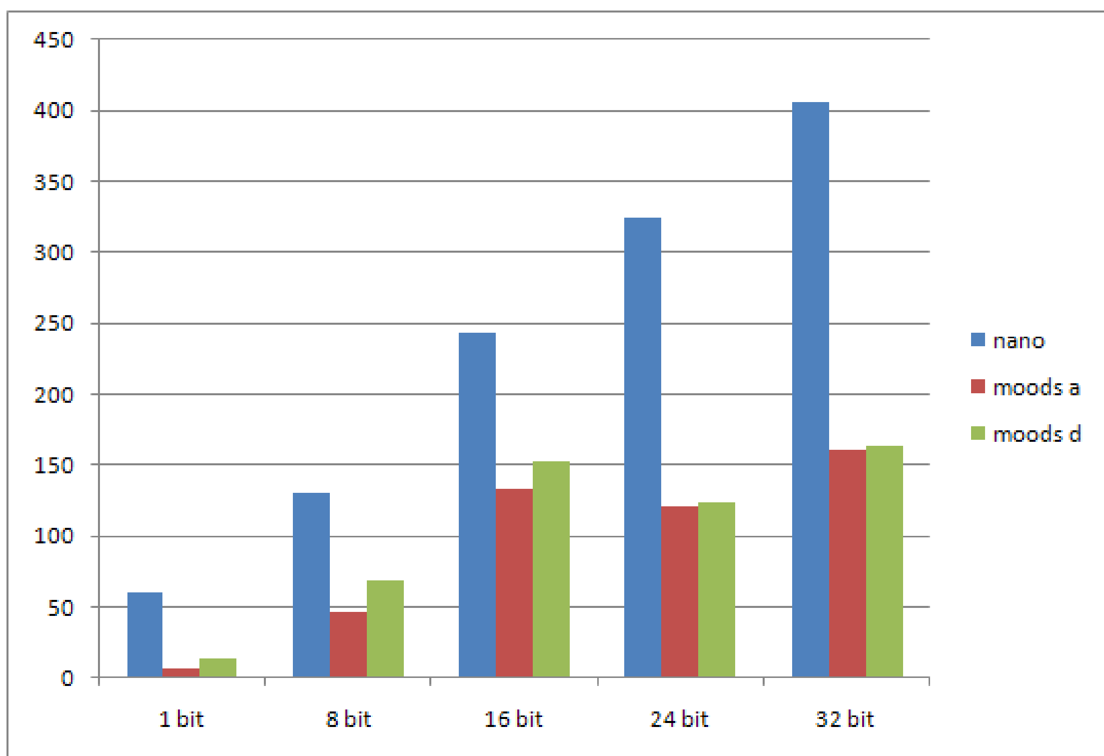


FIGURE 6.7: The number of LUTs used to implement the Arithmetic Multiplier

From Figure 6.6 we can see little difference in Slice Registers usage between the output of the nano-processor approach and that of the moods. However, at bit width of 16 and over, the Slice Register usage of the output of the nano-processor approach is less than that of the Moods output optimised for delay.

This, however, does not mean that the output of the nano-processor approach uses less area than that of the Moods output optimised for delay. The area usage is the aggregate of both Slice Register Usage and LUTs, and looking at the LUT utilisation graph, we can clearly see that at those bit widths the output of the nano-processor approach used a lot more LUTs than that of Moods.

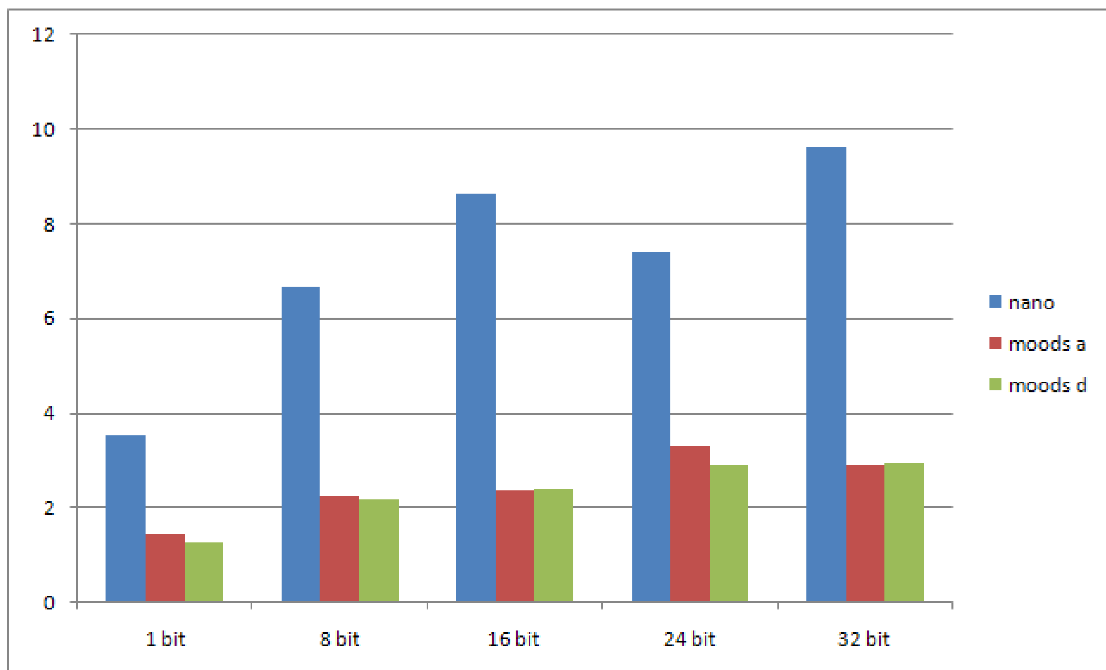


FIGURE 6.8: Delay in nano seconds through the critical path of the Arithmetic Multiplier

Figure 6.8 shows that the output of the nano-processor approach is at least twice as slow as that of Moods. In addition to that, we can see that the delay in this case is affected by the width of the variables. This is mainly because of the nature of the adder used as part of the multiplication operation.

In the tool implementing the nano-processor approach, the RTL output corresponding to an addition operator is a cascade of 1-bit full-adders. Therefore, as the bit width increases, more full-adders are linked together and result in a longer critical path.

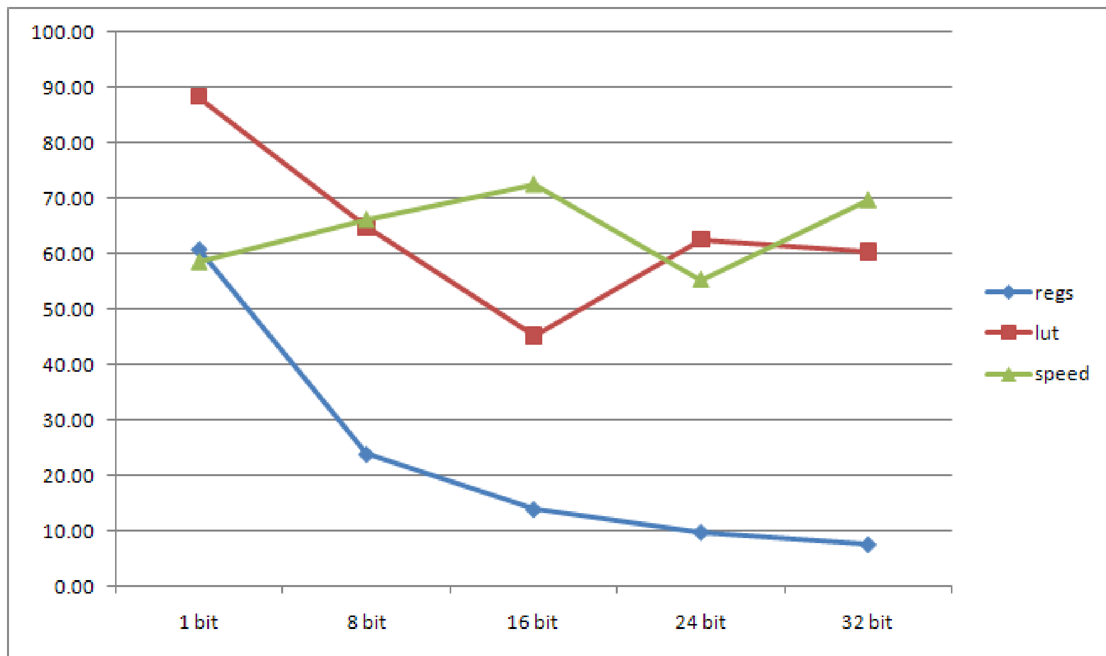


FIGURE 6.9: Percentage difference with MOODS output optimised for area for the Arithmetic Multiplier

From Figure 6.9 and Figure 6.10, the only significant change in percentage difference between the nano-processor and the Moods output is at the Slice Registers usage. For delay and LUT usage, the Moods output clearly outperforms the nano-processor approach output.

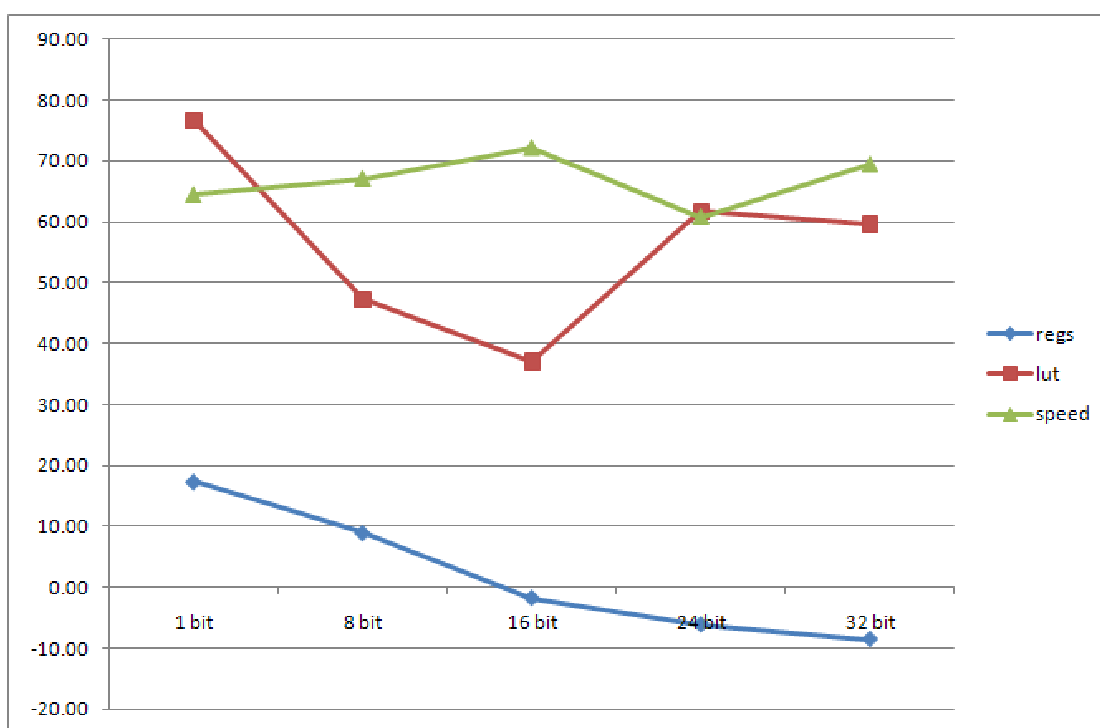


FIGURE 6.10: Percentage difference with MOODS output optimised for delay for the Arithmetic Multiplier

## 6.4 Iterative Fibonacci Calculator

In the Fibonacci sequence, the first two Fibonacci numbers are 0 and 1, and each subsequent number is the sum of the previous two.

Listing C.1 is the VHDL behavioural description of an iterative implementation of the Fibonacci sequence calculation.

```
entity fib_entity is
    port (number: in integer;
           fibonacci: out integer);
end entity fib_entity;

architecture behavior of fib_entity is
begin
        fib_calc: process (number) is
                variable fib_1 : integer;
                variable fib_2 : integer;
                variable count : integer;
                variable result: integer;

                begin
                count := number;
                fib_1 := 1;
                fib_2 := 0 ;

                if count = 0 then
                        result := 0 ;
                elsif count = 1 then
                        result := 1 ;
                else
                        count := count - 1;
                        while count > 0 loop
                                result := fib_1 + fib_2;
                                fib_2  := fib_1 ;
                                fib_1  := result ;
                                count  := count - 1;
                        end loop;
                end if;
                fibonacci <= result;
    end process fib_calc;
end architecture behavior;
```

LISTING 6.3: VHDL code of the fibonnaci sequence calculator

Figure 6.11 and Figure 6.12 show that, at all bit widths, the silicon area required to implement the output of the nano-processor synthesis approach in this case far exceeds that required to implement the output of the MOODS synthesis tool, both optimised for area and delay.

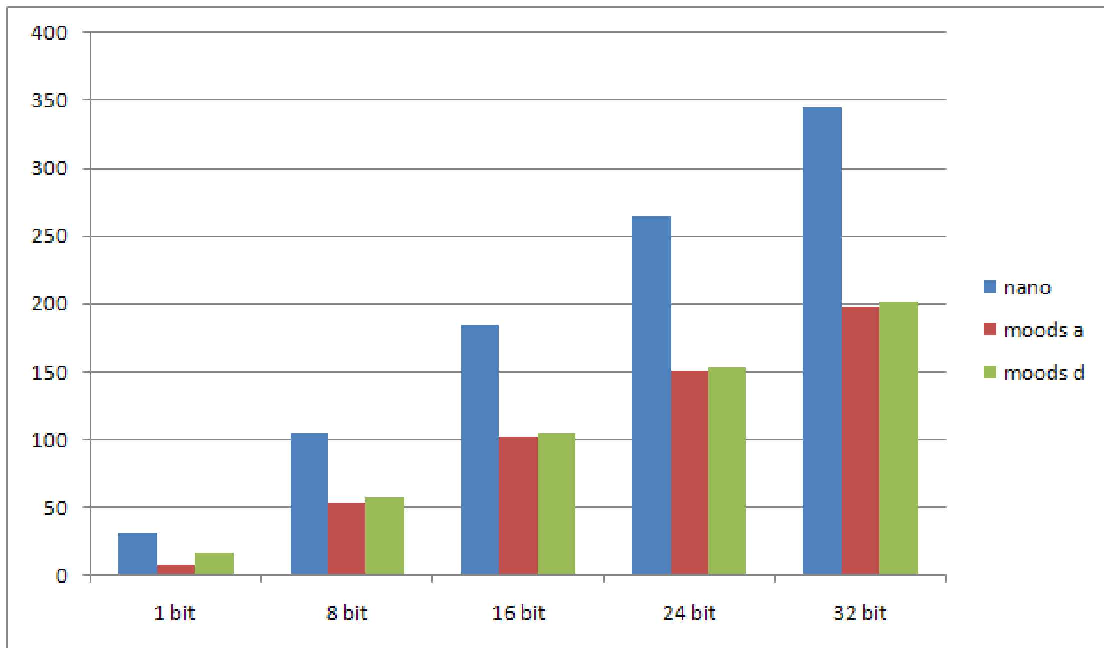


FIGURE 6.11: The number of Slice Registers used to implement the iterative Fibonnaci Calculator

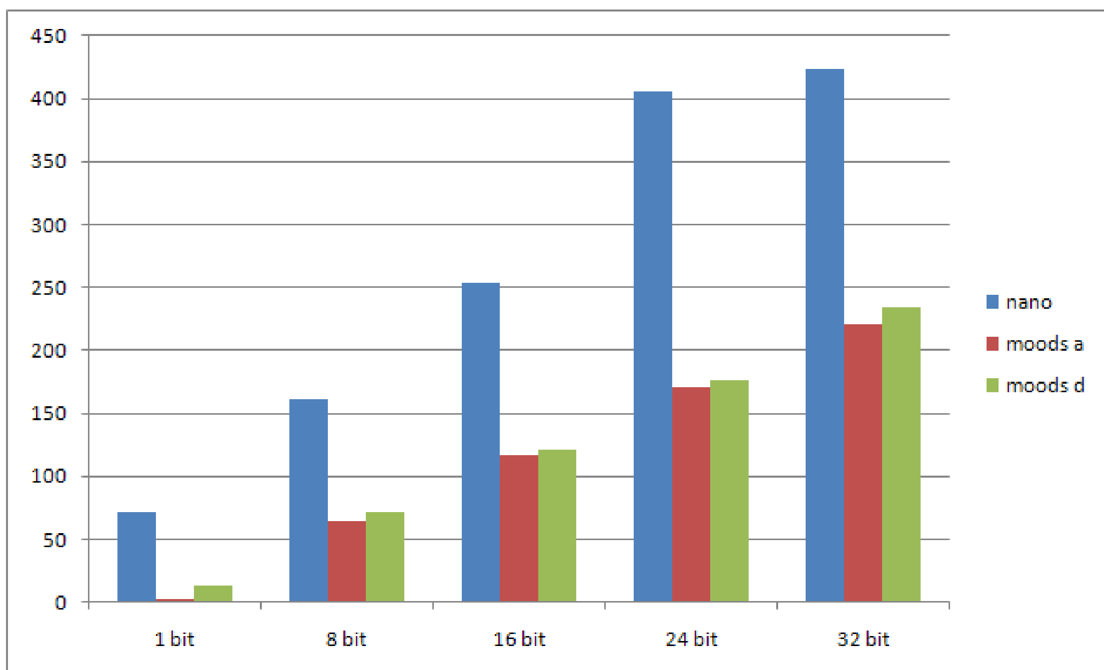


FIGURE 6.12: The number of LUTs used to implement the iterative Fibonnaci Calculator

On top of that, Figure 6.13 shows that the delay in the critical path of the output of the nano-processor synthesis approach is greater than that of the output of the MOODS synthesis tool at all bit widths.

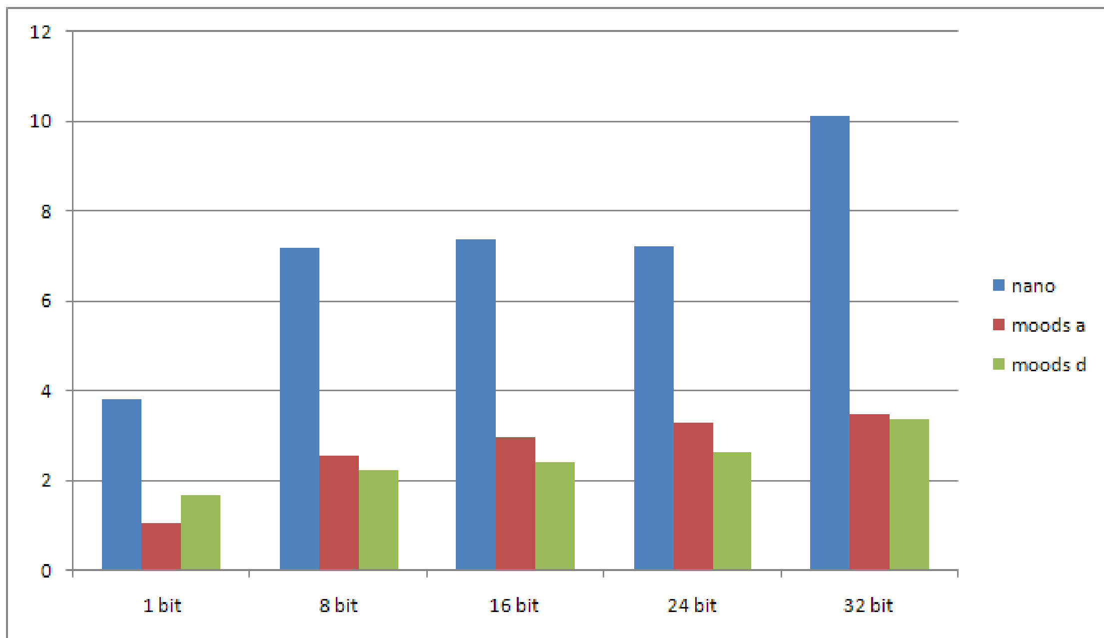


FIGURE 6.13: Delay in nano seconds through the critical path of the iterative Fibonnaci Calculator

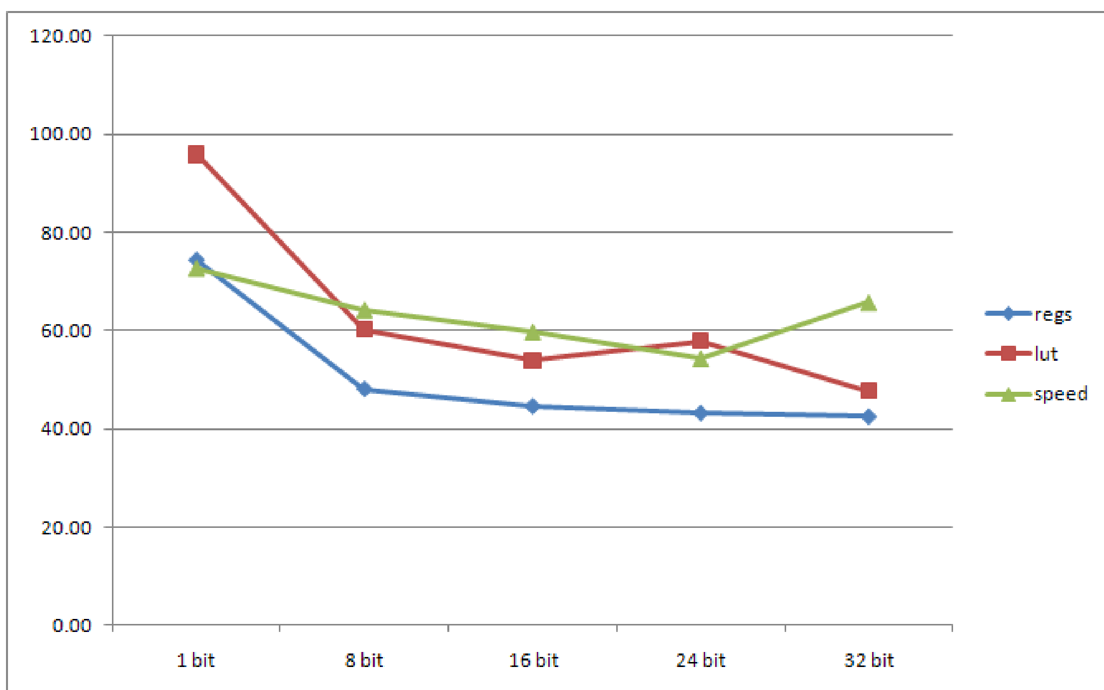


FIGURE 6.14: Percentage difference with MOODS output optimised for area for the iterative Fibonnaci Calculator

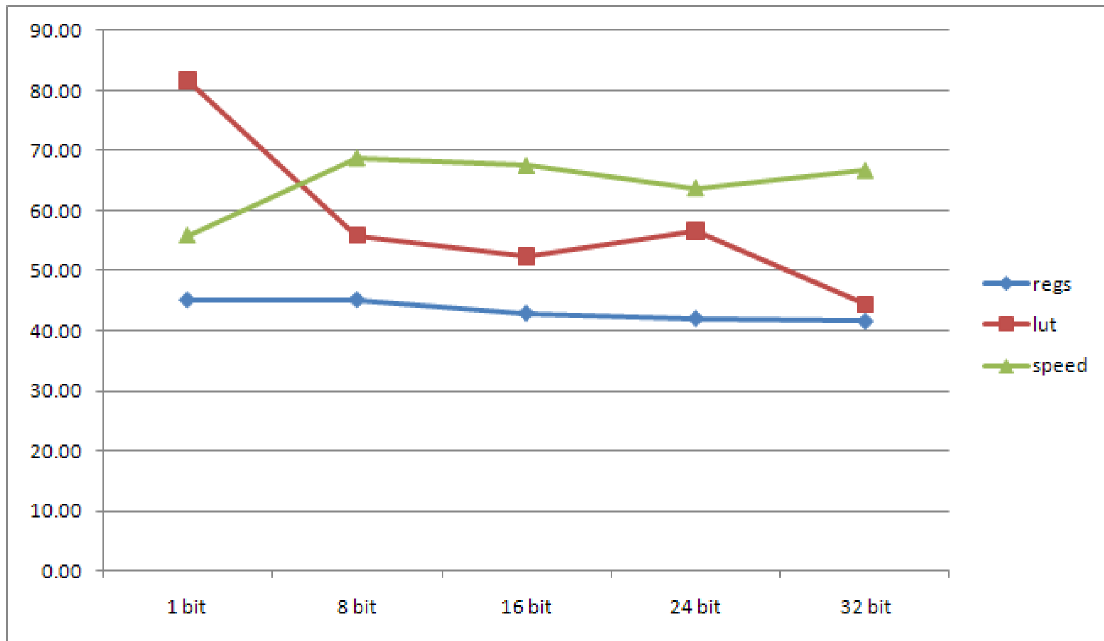


FIGURE 6.15: Percentage difference with MOODS output optimised for delay for the iterative Fibonnaci Calculator

Figure 6.14 and Figure 6.15 confirm that the synthesis output of the Fibonacci sequence calculator using the MOODS synthesis tool is much faster and requires a lot less resources than that using the nano-processor approach.

## 6.5 Further Experiments

### 6.5.1 Cycle Count

In order to compare the performance of nano-processor, the number of clock cycles required to calculate fibonnaci numbers using the nano processor and Moods is captured in table 6.5.1:

| | Nano | Moods |
|---|---|---|
| Fib(4) | 290 | 8 |
| fib(10) | 663 | 15 |
| fib(23) | 1390 | 28 |

TABLE 6.1: Clock cycle count when executing Fibonacci of N

| | Nano | Moods |
|---|---|---|
| Frequency (Mhz) | 137.30 | 287.76 |

TABLE 6.2: Clock frequency of the Fibonacci implementations

Along with the clock frquency of both implementations, captured in table 6.5.1, the results show that Moods clearly outperforms the nano-processor approach in terms of throughput. This can be explained by the fact that the nano processor implementation of the Fibonacci calculator uses a sequence of fetch execute instructions to represent the control flow of the behaviour. Each fetch execute instruction including the no-op instructions is a 5 cycle operation.

### 6.5.2 Longer sequential code

A longer sequential code with multiple loops was used as input to the nano processor synthesis tool and to Moods. The code is listed in Appendix C. It is a 6 times repeat of the Fibonacci loop.

The table below shows the resource utilisation as well as the critical path of the resulting architectures:

| | Nano | Moods |
|---|---|---|
| Number of Registers | 93 | 152 |
| Number of LUTs | 183 | 141 |
| Delay (ns) | 4.942 | 2.795 |

From the results above, we can see that even though Moods continues to produce faster implementations, the resource utilisation of the nano processor did not increase as much as the resource utilisation of the Moods implementation. This can be explained by the fact that the resources needed to implement the shorter sequential code had enough redundant hardware to accommodate the longer piece of code.

# Chapter 7

# Final Remarks

Implementing the newly defined approach to the behavioural synthesis of systems proved to be a highly challenging undertaking. A significant amount of time was spent on the technical implementation of the VHDL compiler and further time was spent on the implementation of the synthesis approach.

The results described previously give an indication that the newly defined approach is most likely less efficient than existing behavioural synthesis techniques when applied to behavioural descriptions of low complexity systems with a small number of variables and operations.

The test cases that have been showcased earlier can only tell part of the story. Therefore, the current inability to support further VHDL constructs (namely procedure/function calls) was the main obstacle against testing the new appraoch on bigger and more complex systems.

# Appendix A

# Details Of The VHDL Parser

## Terminology

The syntax of a programming language is defined as a description of the proper form of its programs, and the semantics of a language define what its programs mean; that is what each program does when it executes.

E-BNF stands for Extended Backus Naur Form. E-BNF is a metasyntax notation used to express context-free grammars. A source code of a programming language consists of Terminal symbols, that is, visible characters, digits, punctuation marks, white space characters, keywords and so forth. The E-BNF description defines Production Rules where sequences of symbols are respectively assigned to a Non-Terminal.

## The VHDL Language Definition Graph

The grammar definition graph mentioned in Section 4.3 is implemented as a set of interlinked rule graphs. Rule graphs are graph representations of the E-BNF VHDL

grammar rules. The nodes of the graph data structure are either Terminals (lexical tokens) or Non-Terminals (reference to a sequence of Terminals and Non-Terminals). Each Non-Terminal node has a pointer to its Rule graph. This structure allows the program to find a path through the set of Rule graphs, by following the pointers of Non-Terminals. A simple function that finds a Terminal immediately next to another Terminal is implemented in this way. Thus the set of Rule graphs is virtually one Grammar Definition Graph.

Figure A.1 shows the graph representation of the entity_declaration syntax rule as defined on the VHDL Language Reference Manual. The following listing is the E-BNF definition of the entity_declaration syntax rule. The name in capital are Terminals, and the other ones are Non-Terminals. In the figure, the pointers inside Non-Terminals such as entity_header and entity_declarative_part are represented by an expansion into another Rule graph. Because of space limitation, not all pointers have been expanded in the figure.

```
entity_declaration ::=
        ENTITY IDENTIFIER IS
                entity_header
                entity_declarative_part
      [ BEGIN
                {entity_statement} ]
        END [ ENTITY ] [ simple_name ] ;


entity_header ::=
        [ formal_generic_clause ]
        [ formal_port_clause ]


entity_declarative_part ::=
        { entity_declarative_item }
```

Listing A.1: VHDL code of the fibonnaci sequence calculator

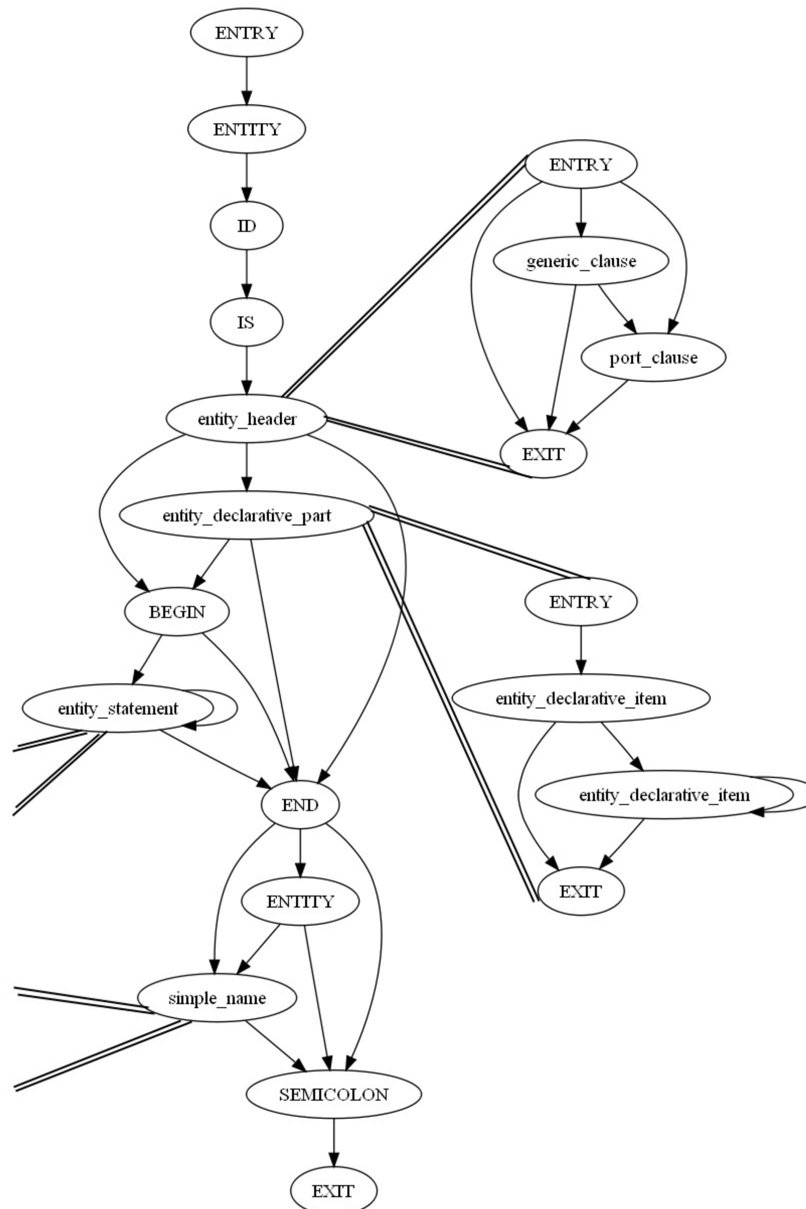


FIGURE A.1: Graph Representation of the Syntax Rule for Entity_declaration

# Appendix B

# RTL Synthesis of Test Cases

## Synthesis of a Fibonacci calculator

The behavioural VHDL code of the Fibonacci calculator described in Listing 6.3 synthesises successfully using the newly developed nano-processor synthesis tool and MOODS. However, the commercial Xilinx ISE 10.1 RTL synthesis tool fails to synthesise the input behaviour because it could not synthesise the loop.

This shows that there are simple behavioural description that cannot be synthesised directly using a commercially available RTL synthesis tool, and therefore require a behavioural synthesis tool to translate that behaviour into a RTL representation which is in turn synthesisable using readily available RTL synthesis tools.

Below is a snippet of the error log from the RTL synthesis process:

```
Release 10.1 - xst K.31 (nt)
Copyright (c) 1995-2008 Xilinx, Inc.  All rights reserved.
...
=========================================================================
*                          HDL Compilation                              *
=========================================================================
Compiling vhdl file "C:/Documents and Settings/Abdeldjalil/Desktop/TEST & SIM/te
st/tes/fib.vhd" in Library work.
Architecture behavior of Entity fib_entity is up to date.
```

```
=========================================================================
*                      Design Hierarchy Analysis                        *
=========================================================================
Analyzing hierarchy for entity <fib_entity> in library <work> (architecture <beh
avior>).


=========================================================================
*                             HDL Analysis                              *
=========================================================================
Analyzing Entity <fib_entity> in library <work> (Architecture <behavior>).
WARNING:Xst:1760 - "C:/Documents and Settings/Abdeldjalil/Desktop/TEST & SIM/tes
t/tes/fib.vhd" line 26: Overflow in constant operation.
....
ERROR:Xst:1312 - Loop has iterated 256 times. Use "set -loop_iteration_limit XX"
 to iterate more.
-->
```

LISTING B.1: Snippet of Error log when synthesising fibonacci code

# Appendix C

# Long sequential code

## Fibonnacci loop repeated 6 times

```
entity fib_entity is
    port (number: in integer range 127 downto -128 ;
                  fibonacci: out integer range 127 downto -128  );
end entity fib_entity;

architecture behavior of fib_entity is

begin

    fib_calc: process (number) is
        variable fib_1 : integer range 127 downto -128 ;
        variable fib_2 : integer range 127 downto -128 ;
        variable count : integer range 127 downto -128 ;
                          variable result: integer range 127 downto -128 ;
        begin

                count := number;
                fib_1 := 1;
                                        fib_2 := 0 ;

                        if count = 0 then
                                result := 0 ;
                        elsif count = 1 then
                                result := 1 ;
                        else
                                count := count - 1;
                                while count > 0 loop
                                        result := fib_1 + fib_2;
                                        fib_2  := fib_1 ;
                                        fib_1  := result ;
                                        count  := count - 1;
                                end loop;
                        end if;

                        count := number;
                fib_1 := 1;
                        fib_2 := 0 ;

                        if count = 0 then
```

```
                result := 0 ;
        elsif count = 1 then
                result := 1 ;
        else
                count := count - 1;
                while count > 0 loop
                        result := fib_1 + fib_2;
                        fib_2  := fib_1 ;
                        fib_1  := result ;
                        count  := count - 1;
                end loop;
        end if;

        count := number;
fib_1 := 1;
        fib_2 := 0 ;

        if count = 0 then
                result := 0 ;
        elsif count = 1 then
                result := 1 ;
        else
                count := count - 1;
                while count > 0 loop
                        result := fib_1 + fib_2;
                        fib_2  := fib_1 ;
                        fib_1  := result ;
                        count  := count - 1;
                end loop;
        end if;

        count := number;
fib_1 := 1;
        fib_2 := 0 ;

        if count = 0 then
                result := 0 ;
        elsif count = 1 then
                result := 1 ;
        else
                count := count - 1;
                while count > 0 loop
                        result := fib_1 + fib_2;
                        fib_2  := fib_1 ;
                        fib_1  := result ;
                        count  := count - 1;
                end loop;
        end if;


        count := number;
fib_1 := 1;
        fib_2 := 0 ;

        if count = 0 then
                result := 0 ;
        elsif count = 1 then
                result := 1 ;
        else
                count := count - 1;
                while count > 0 loop
                        result := fib_1 + fib_2;
                        fib_2  := fib_1 ;
                        fib_1  := result ;
                        count  := count - 1;
                end loop;
        end if;

        count := number;
fib_1 := 1;
        fib_2 := 0 ;
```

```
                        if count = 0 then
                                result := 0 ;
                        elsif count = 1 then
                                result := 1 ;
                        else
                                count := count - 1;
                                while count > 0 loop
                                        result := fib_1 + fib_2;
                                        fib_2  := fib_1 ;
                                        fib_1  := result ;
                                        count  := count - 1;
                                end loop;
                        end if;


                        fibonacci <= result;
    end process fib_calc;


end architecture behavior;
```

Listing C.1: long sequential VHDL code

# Bibliography

[1] 1076 ieee standard vhdl language reference manual. *IEEE Std 1076-2002 (Revision of IEEE Std 1076, 2002 Edn)*, pages 1–300, 2002.

[2] K. Atasu, R.G. Dimond, O. Mencer, W. Luk, C. Ozturan, and G. Diindar. Optimizing instruction-set extensible processors under data bandwidth constraints. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, 2007.

[3] R. Bergamaschi and D.J. Allerton. A graph-based silicon compiler for concurrent vlsi systems. *CompEuro '88. 'Design: Concepts, Methods and Tools'*, pages 36–47, Apr 1988.

[4] A.P. Chandrakasan, S. Sheng, and R.W. Brodersen. Low-power cmos digital design. *Solid-State Circuits, IEEE Journal of*, 27(4):473–484, Apr 1992. ISSN 0018-9200.

[5] Chung-Hsing Chen, T. Karnik, and D.G. Saab. Structural and behavioral synthesis for testability techniques. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 13(6):777–785, Jun 1994. ISSN 0278-0070.

[6] J. Cong, Bin Liu, and Junjuan Xu. Coordinated resource optimization in behavioral synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1267–1272, 2010.

[7] A.J. Currie and K.R. Baker. Multiple objective optimisation in behavioural synthesis. *Formal and Semi-Formal Methods for Digital Systems Design, IEE Colloquium on*, pages 5/1–5/3, Jan 1991.

[8] V. Pickering D. Crookes, R. Fee. Building syntax graphs from syntax equations: A case study in modular programming. *Software: Practice and Experience*, 13(12): 1129–1139, 1983.

[9] A.A. Del Barrio, M.C. Molina, J.M. Mendias, R. Hermida, and S.O. Memik. Using speculative functional units in high level synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2010*, pages 1779–1784, 2010.

[10] S. Devadas and A.R. Newton. Algorithms for hardware allocation in data path synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(7):768–781, Jul 1989. ISSN 0278-0070.

[11] Charles Donnelly and Richard Stallman. *Bison: The YACC-compatible Parser Generator*, 1995.

[12] G. Economakos, P. Oikonomakos, I. Panagopoulos, I. Poulakis, and G. Papakonstantinou. Behavioral synthesis with systemc. In *DATE '01: Proceedings of the conference on Design, automation and test in Europe*, pages 21–25, Piscataway, NJ, USA, 2001. IEEE Press. ISBN 0-7695-0993-2.

[13] Roozbeh Farahbod, Vincenzo Gervasi, and Uwe Glsser. Coreasm: An extensible asm execution engine. In *PROC. OF THE 12TH INTL WORKSHOP ON ABSTRACT STATE MACHINES*, pages 153–165, 2005.

[14] E.F. Girczyc, R.J.A. Buhr, and J.P. Knight. Applicability of a subset of ada as an algorithmic hardware description language for graph-based hardware compilation.

*Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 4(2):134–142, April 1985. ISSN 0278-0070.

[15] R.E. Gonzalez. Xtensa: a configurable and extensible processor. *Micro, IEEE*, 20 (2):60–70, 2000. ISSN 0272-1732.

[16] David Goodwin and Darin Petkov. Automatic generation of application specific processors. In *Proceedings of the 2003 international conference on Compilers, architecture and synthesis for embedded systems*, CASES '03, pages 137–147, New York, NY, USA, 2003. ACM. ISBN 1-58113-676-5.

[17] B. Gorjiara and D. Gajski. Automatic architecture refinement techniques for customizing processing elements. In *Design Automation Conference, 2008. DAC 2008. 45th ACM/IEEE*, pages 379–384, 2008.

[18] S. Gupta, N. Dutt, R. Gupta, and A. Nicolau. Spark: a high-level synthesis framework for applying parallelizing compiler transformations. *VLSI Design, 2003. Proceedings. 16th International Conference on*, pages 461–466, Jan. 2003. ISSN 1063-9667.

[19] M.J.M. Heijligers, L.J.M. Cluitmans, and J.A.G. Jess. High-level synthesis scheduling and allocation using genetic algorithms. *Design Automation Conference, 1995. Proceedings of the ASP-DAC '95/CHDL '95/VLSI '95., IFIP International Conference on Hardware Description Languages; IFIP International Conference on Very Large Scale Integration., Asian and South Pacific*, pages 61–66, Aug-1 Sep 1995.

[20] III Hitchcock, C.Y. and D.E. Thomas. A method of automatic data path synthesis. *Design Automation, 1983. 20th Conference on*, pages 484–489, June 1983. ISSN 0738-100X.

[21] Jinhwan Jeon, Daehong Kim, Dongwan Shin, and Kiyoung Choi. High-level synthesis under multi-cycle interconnect delay. In *Design Automation Conference, 2001. Proceedings of the ASP-DAC 2001. Asia and South Pacific*, pages 662–667, 2001.

[22] D. Johannsen. Bristle blocks: A silicon compiler. *Design Automation, 1979. 16th Conference on*, pages 310–313, June 1979.

[23] K. Kelley, M. Wachs, A. Danowitz, P. Stevenson, S. Richardon, and M. Horowitz. Intermediate representations for controllers in chip generators. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[24] A.A. Kountouris and C. Wolinski. Efficient scheduling of conditional behaviors using hierarchical conditional dependency graphs in the codesis system. In *Euromicro Conference, 2000. Proceedings of the 26th*, volume 1, pages 222–229 vol.1, 2000.

[25] T. Kozlowski, E.L. Dagless, J.M. Saul, M. Adamski, and J. Szajna. Parallel controller synthesis using petri nets. *Computers and Digital Techniques, IEE Proceedings -*, 142(4):263–271, Jul 1995. ISSN 1350-2387.

[26] F.J. Kurdahi and A.C. Parker. Real: A program for register allocation. *Design Automation, 1987. 24th Conference on*, pages 210–215, June 1987. ISSN 0738-100X.

[27] Jiahn-Hung Lee, Yu-Chin Hsu, and Youn-Long Lin. A new integer linear programming formulation for the scheduling problem in data path synthesis. *Computer-Aided Design, 1989. ICCAD-89. Digest of Technical Papers., 1989 IEEE International Conference on*, pages 20–23, Nov 1989.

[28] T.-C. Lee, W.H. Wolf, and N.K. Jha. Behavioral synthesis for easy testability in data path scheduling. *Computer-Aided Design, 1992. ICCAD-92. Digest of Technical Papers., 1992 IEEE/ACM International Conference on*, pages 616–619, Nov, 1992.

[29] Youn-Long Lin. Recent developments in high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst.*, 2(1):2–21, 1997. ISSN 1084-4309.

[30] Lubos Lorenc, Rudolf Schnecker, and Zbynek Krivka. A note on the parsing of complete vhdl-2002. In Alica Kelemenov, Dusan Kolar, and Alexander Meduna, editors, *Workshop on Formal Models*, volume 255 of *CEUR Workshop Proceedings*. CEUR-WS.org, 2007.

[31] M.C. Molina, R. Ruiz-Sautua, J.M. Mendias, and R. Hermida. Area optimization of multi-cycle operators in high-level synthesis. In *Design, Automation Test in Europe Conference Exhibition, 2007. DATE '07*, pages 1–6, 2007.

[32] J.A. Nestor and G. Krishnamoorthy. Salsa: a new approach to scheduling with timing constraints. *Computer-Aided Design, 1990. ICCAD-90. Digest of Technical Papers., 1990 IEEE International Conference on*, pages 262–265, Nov 1990.

[33] B.M. Pangrle and D.D. Gajski. Design tools for intelligent silicon compilation. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(6):1098–1112, November 1987. ISSN 0278-0070.

[34] T. J. Parr and R. W. Quong. Antlr: a predicated-ll(k) parser generator. *Softw. Pract. Exper.*, 25(7):789–810, 1995. ISSN 0038-0644.

[35] Terence John Parr. *Language Translation Using PCCTS and C++*, 1993.

[36] P.G. Paulin and J.P. Knight. Force-directed scheduling for the behavioral synthesis of asics. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 8(6):661–679, Jun 1989. ISSN 0278-0070.

[37] P.G. Paulin, J.P. Knight, and E.F. Girczyc. Hal: A multi-paradigm approach to automatic data path synthesis. *Design Automation, 1986. 23rd Conference on*, pages 263–270, June 1986. ISSN 0738-100X.

[38] Massoud Pedram. Power minimization in ic design: principles and applications. *ACM Trans. Des. Autom. Electron. Syst.*, 1(1):3–56, 1996. ISSN 1084-4309.

[39] Z. Peng. Synthesis of vlsi systems with the camad design aid. *Design Automation, 1986. 23rd Conference on*, pages 278–284, June 1986. ISSN 0738-100X.

[40] M. Potkonjak, S. Dey, and R.K. Roy. Behavioral synthesis of area-efficient testable designs using interaction between hardware sharing and partial scan. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 14(9): 1141–1154, Sep 1995. ISSN 0278-0070.

[41] A. Raghunathan and N.K. Jha. Scalp: an iterative-improvement-based low-power data path synthesis system. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 16(11):1260–1277, Nov 1997. ISSN 0278-0070.

[42] Salil Raje and Majid Sarrafzadeh. Variable voltage scheduling. In *ISLPED '95: Proceedings of the 1995 international symposium on Low power design*, pages 9–14, New York, NY, USA, 1995. ACM. ISBN 0-89791-744-8.

[43] Insup Shin, Seungwhun Paik, and Youngsoo Shin. Register allocation for high-level synthesis using dual supply voltages. In *Design Automation Conference, 2009. DAC '09. 46th ACM/IEEE*, pages 937–942, 2009.

[44] R. Sinha and H.D. Patel. Abstract state machines as an intermediate representation for high-level synthesis. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2011*, pages 1–6, 2011.

[45] D.E. Thomas, III Hitchcock, C.Y., T.J. Kowalski, J.V. Rajan, and R.A. Walker. Automatic data path synthesis. *Computer*, 16(12):59–70, Dec. 1983. ISSN 0018-9162.

[46] H. Trickey. Flamel: A high-level hardware compiler. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 6(2):259–269, March 1987. ISSN 0278-0070.

[47] Weidong Wang, A. Raghunathan, N.K. Jha, and S. Dey. Resource budgeting for multiprocess high-level synthesis. *Computer-Aided Design of Integrated Circuits and Systems, IEEE Transactions on*, 23(7):1010–1019, July 2004. ISSN 0278-0070.

[48] Lin Zhong and N.K. Jha. Interconnect-aware high-level synthesis for low power. *Computer Aided Design, 2002. ICCAD 2002. IEEE/ACM International Conference on*, pages 110–117, Nov. 2002. ISSN 1092-3152.

[49] M. Zuluaga, T. Kluter, P. Brisk, N. Topham, and P. Ienne. Introducing control-flow inclusion to support pipelining in custom instruction set extensions. In *Application Specific Processors, 2009. SASP '09. IEEE 7th Symposium on*, pages 114–121, 2009.