High level synthesis for non-manifest digital signal processing applications

Omar Mansour

Composition of the Graduation Committee:

Prof. Dr. Ir.	Th.	Krol (promoter)
Dr. Ir.	G.J.M.	Smit, UT, department of Computer Science
Prof. Dr. Ir.	C.H.	Slump, UT, department of Electrical Engineering
Prof. Dr.-Ing.	J.	Becker, Universität Karlsruhe, Germany
Prof. Dr.	S.	Vassiliadis, University of Delft
Prof. Dr. Ir.	H.	Corporaal, University of Eindhoven
Dr. Ir.	M.	Bekooij, Philips research

This research is conducted within the High^2 project (TES.5226) supported by the PROGram for Research on Embedded Systems & Software (PROGRESS) of the Dutch organization for Scientific Research NWO, the Dutch Ministry of Economic Affairs and the technology foundation STW.

Group of Computer Architecture,
Design & Test for Embedded Systems.
P.O. Box 217, 7500 AE Enschede, The Netherlands.

Keywords: high-throughput, non-manifest, streaming applications,
coarse grained data flow architecture.

Printed by Ipskamp PrintPartners, Enschede, The Netherlands.
ISBN 90-365-2321-4

HIGH LEVEL SYNTHESIS FOR NON-MANIFEST DIGITAL SIGNAL PROCESSING APPLICATIONS

DISSERTATION

to obtain
the doctor's degree at the University of Twente,
on the authority of the rector magnificus,
prof. dr. W.H.M. Zijm,
on account of the decision of the graduation committee,
to be publicly defended
on Thursday, February 2, 2006 at 15.00

by

Omar Mansour

born on 22 january 1969

in Cairo, Egypt

This dissertation is approved by:

Prof. Dr. Ir. Th. Krol (promoter)

Abstract

High throughput streaming applications operate in an environment where continuous processing of information takes place. An important class of such applications are embedded systems for digital signal processing, *DSP*, in which the algorithms of the system are applied to *streams* of signals. The variety of these applications is large and encompasses algorithms for filtering audio and video signals, algorithms for error correction, compression, picture in picture etc. Modern multimedia systems could not exist without those kind of embedded systems. The algorithms are all characterized by a repeated application of some functions on a stream of input values, eventually resulting in a stream of output values. In order to process the stream samples and provide the desired quality, many functions are performed on the consecutive stream samples.

To maintain the stream throughput and provide outputs that are synchronized with the input signals, the applications require sufficient processing power. Sufficient means that the processing power available within an application should be enough to handle the computational load generated by the stream data. This requirement poses a challenge to system or application designers. System designer have to predict the required load in order to optimize their design for the application needs.

If the algorithms of a high-throughput application are manifest and hence the required load to perform the function is predictable and independent of the stream data, the task is straitforward and usually involves static scheduling of the processing elements.

Unfortunately not all algorithms are manifest. Non-manifest algorithms that are data dependent and produce a variable computational load, do exist. So far, the data dependency was neglected and implementations were based on static scheduling for

the worst case.

In this thesis we show the feasibility of Coarse Grained Data Flow Machines for high-throughput streaming non-manifest applications. The architecture of the Coarse Grained Data Flow Machine is derived from the classical data flow architecture and the scheduling of its processing elements is done dynamically in hardware.

Since the implementation of such an architecture is strongly application dependent, a design flow and supporting software tools, are provided. This gives application designers the means by which the number of processing elements, buffer sizes and latencies of the architecture can be tuned.

Acknowledgements

I am grateful to many people for their support during my Ph.D. First of all, I would like to thank my promoter Thijs Krol for giving me the opportunity to do my Ph.D. at his chair (CADTES) and for his enormous help, support, sharp criticism and quality assurance that made this thesis reach this level of maturity. I also want to thank Gerard Smit for his advice both technically and strategically and for the various versions of my thesis that he had read and corrected. I want to show my gratitude to Bert Molenkamp for the help and support on the VHDL experiments we made and to Michèl Rosien for the implementation support of the High2 Simulator we developed. Actually I have to show my gratitude to many people at the CADTES group for their help and knowledge, Ferdy Hanssen who is in my opinion an expert on many topics but especially when it comes to LaTeX he is the real guru. Paul Heysters, Yuanqing Guo, and Maarten Wiggers my roommates Lodewijk Smit, Gerard Rauwerda, Rick van Rein for the tips and the fruitful discussions we had.
I am also grateful to the group of secretaries Marlous and Nicole for their help and support during my stay at the CADTES group.
Thanks to Pierre Jansen, Hans Scholten, Albert Schouten, whom I had known as lecturers during my study and as colleges during my Ph.D. I am also not forgetting the rest of the DIES group whom we share the floor with. Thank you all is has been a pleasure to work with you.
Last but not least I want thank Djoyce for her help and support during the difficult periods we had.
Finally I dedicate this thesis to my parents who always believed in me regardless of whether I made an agreeable choice or not.

Table of Contents

Abstract **v**

Acknowledgements **vii**

Table of Contents **ix**

1 Introduction **1**

1.1 Introduction 1

1.1.1 Throughput 2

1.1.2 Synchronicity 3

1.1.3 The design process for high-throughput *DSP* applications . . . 4

1.2 Problem Formulation 5

1.3 Proposed Solution 5

1.4 Thesis Outline 6

2 Models and Definitions **9**

2.1 Introduction 9

2.2 Environment 10

2.3 Application 17

3 Non Manifest Algorithms **23**

3.1 Introduction 23

3.2 Examples of Non Manifest algorithms 25

3.2.1 Greatest Common Divider (GCD) 25

3.2.2 Russian Multiplication 28
3.2.3 Cordic Rotation 30
3.2.4 Montgomery Inverse 37
3.3 Statistical properties of non-manifest algorithms 39
3.4 Conclusions 41
3.4.1 Summary 42

4 Dynamic Hardware Scheduling Architectures 45
4.1 Introduction 45
4.2 Static scheduling 46
4.2.1 Phideo 47
4.2.2 Pipelining 49
4.3 Dynamic scheduling in hardware 52
4.3.1 Data flow architectures 53
4.3.2 Scoreboard scheduler 55
4.3.3 Tomasulo scheduler 68

5 High^2 Data Flow Machine 79
5.1 Introduction 79
5.2 The Simple model 80
5.2.1 Design 82
5.2.2 Design Parameters 87
5.2.3 Theoretical calculation of the latency 90
5.2.4 Parameter calculation by simulation 97
5.2.5 Comparison of the theoretical latency-bound and obtained simulation results 101
5.2.6 Design flow of simple model applications 102
5.2.7 Hardware Implementation 107
5.2.8 Improved Hardware Architecture 110
5.2.9 Conclusions of the simple model 114
5.3 The Complex model 116
5.3.1 Problem Description 118

5.4 The High^2 *DFM* 119
5.4.1 Abstract *DFM* Model 120
5.4.2 High^2 *DFM* Implementation 124
5.4.3 Design shortcomings 135
5.4.4 Possible system modifications and improvements 136
5.4.5 Design Flow DFM 137
5.5 Conclusions 140

6 Example of High^2 *DFM* Applications 143
6.1 Introduction 143
6.1.1 Example montgomery-gcd-montgomery 144
6.1.2 Design flow 146
6.2 Conclusions 161

7 Conclusions and Future Work 163
7.1 Non-manifest algorithms 163
7.2 Dynamic hardware scheduling architectures 164
7.3 High^2 Data Flow Machine 165
7.4 Example of High^2 *DFM* Applications 166
7.5 Conclusions 166
7.6 Future work 166

A Scoreboard rules 167

B Tomasulo rules 169

C VHDL code of the simple model architecture 171
C.0.1 Implementation of the processing element 171
C.0.2 Implementation of the scheduler 172
C.0.3 Experimental results 173

Bibliography 181

List of Tables

2.1 Different types of streaming environments 11

3.1 Summary of profiling results 43

4.1 Comparison 77

5.1 System design paremeters 104
5.2 *DFM* Model Summery 121
5.3 Example: cycle count table 138
5.4 Example: reduced cycle count table 139

6.1 Multiplication modulo $P = 7$ 145
6.2 mgcdm calculation 146
6.3 Statistical parameters obtained for simulation 152
6.4 Min, Max, and Average latencies in clock cycles of the *mgcdm* application 156
6.5 Number of Montgomery and Gcd processors versus the obtained simulator latency in clock cycles 158
6.6 Dynamic versus static scheduling results of the *mgcdm* streaming application with a stream throughput of one sample/clock cycle. 159

C.1 Specification of the processing element 172
C.2 Synthesis results 177

List of Figures

2.1 Application within a constant stream environment 12
2.2 Application within a variable stream environment 12
2.3 An example of the *application graph* 20

3.1 Gcd algorithm 26
3.2 (a) Worst case execution versus (b) Best case execution of a Gcd algorithm for 16 bit integer values 26
3.3 *Gcd* distribution for 16 bit integer values, each iteration taken Cgcd=3 clock cycles 27
3.4 Russian Multiplication algorithm 28
3.5 Profiling results of the Russian multiplication algorithm 29
3.6 Cordic rotation algorithm 30
3.7 Calculated *Cordic* error v.s. stop value 31
3.8 Calculated *Cordic* angle v.s. latency in iterations 32
3.9 Calculated *Cordic* frequency distribution, stop value is 10^{-1} 34
3.10 Calculated *Cordic* frequency distribution, stop value is 10^{-4} 35
3.11 Calculated *Cordic* frequency distribution, stop value is 10^{-8} 36
3.12 The Montgomery inverse algorithm 37
3.13 Profile results of the Montgomery inverse algorithm. The latency of the loop body is assumed to be 13 Clock Cycles. Which is an approximate value for the number of instructions of the loop body 38
3.14 Montgomery profiling results for various prime number sets 39
3.15 The Montgomery - Gcd - Montgomery algorithm as an example of a non-manifest loop application 40

3.16 Montgomery-Gcd-Montgomery application distributions 41

4.1 Example of a static (top) v.s. dynamic schedule (bottom). The static schedule always has the worst case latency which is 29 clock cycles in this figure. The dynamic schedule has a data dependent latency between which can range from 11 and 29 clock cycles, based on the actual operands provided 47

4.2 The target architecture of Phideo 49

4.3 A typical pipelined execution behavior 50

4.4 The general organization of the dynamic dataflow model 54

4.5 The CDC 6600 processor courtesy of Control Data Corporation laboratory 56

4.6 Data path of a MIPS processor with a scoreboard scheduling mechanism, the processor contains 5 Functional Units and a register bank. There are separate busses for data communication between the processing elements and the registers. The scoreboard controls the datapath 57

4.7 Scoreboard: the scoreboard consists of a number of fields which are used to keep trace of data dependencies and insure that no hazards can occur due to dependence violations 58

4.8 An example program with its accompanying data dependency graph. In the data dependency graph ellipses indicate *operations*, rectangles indicate *memory* or *registers*, and triangles indicate *constants* 60

4.9 Scoreboard clock 1 to 4 61

4.10 Scoreboard clock 5 to 8 62

4.11 Scoreboard clock 9 to 12 63

4.12 Scoreboard clock 13 to 16 64

4.13 Scoreboard clock 17 to 20 65

4.14 Data path of a MIPS floating-point unit with a Tomasulo Scheduler . 69

4.15 The state of the reservation stations when all instructions are issued and the first instruction has written its result to the CDB 72

4.16 Structure of a processor core with a Tomasulo Scheduler in combination with a reorder-buffer 73

5.1 Simple model of an application with non-manifest loops and its proposed hardware architecture 81
5.2 gcd algorithm 83
5.3 The hardware model of the scheduler, the controller and all control signals are omitted from the figure for simplicity reasons 85
5.4 The flow of data 87
5.5 The definitions 88
5.6 The flow relations 91
5.7 The flow of data 92
5.8 $WIP(t)$ if $IC(t-1,1)= 0$ 94
5.9 $WIP(t)$ if $IC(t-1,1)= 0$ 96
5.10 Simulation results of workload bound B v.s. obtained simulator latency for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$. 99
5.11 Simulation results of workload bound B v.s. queue buffer-sizes for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$. 100
5.12 Theoretical results versus simulation results for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$. 102
5.13 Simulation of the system 105
5.14 Simulation of the system 106
5.15 The new hardware model 112
5.16 Communication busses between scheduler and processing element . . 113
5.17 Communication model between scheduler and processing element using Mealy and Moore models 114
5.18 An example of the *application graph* 116
5.19 A possible static schedule of the application *application graph* 117
5.20 The High^2 *DFM* template 122
5.21 Mapping the *application graph*onto the *DFM* 125
5.22 The High^2 *DFM* template 126

5.23 Architecture of an execution unit 129
5.24 Writing an input operand at the correct position within the operand table 134

6.1 The *mgcdm* application 144
6.2 The *mgcdm* design flow 147
6.3 The Montgomery and Gcd algorithms augmented with profiling instructions 148
6.4 An exemplar of the code used for the profiling process 148
6.5 A fragment of the clock cycle count file produced by the profiling process 149
6.6 *mgcdm* data 150
6.7 *mgcdm* application design flow 153
6.8 *mgcdm* randrow stats 154
6.9 *mgcdm* randrow stats 155
6.10 GUI of the High^2 DFM simulator 157
6.11 *mgcdm* simulator latencies 160

A.1 The scoreboard book keeping rules for each state of an instruction and the actions taken to allow the instructions to advance from one state to the other 167

B.1 The Tomasulo scheduling rules 170

C.1 Generic processing element unfolded in time 173
C.2 Folded version of the generic processing element 174
C.3 *VHDL* code of a generic processing element 174
C.4 The data and bus types 175
C.5 A processing element independent scheduler implementation 176
C.6 The scheduler configuration 177
C.7 The gcd(x,y)processing element implementation 178
C.8 Simulationresults 179

Chapter 1

Introduction

1.1 Introduction

Important classes of embedded systems are systems for digital signal processing, *DSP*, in which algorithms are applied to *streams* of signals. The variety of these applications is large and encompasses algorithms for filtering audio and video signals, algorithms for error correction, compression, picture in picture etc. Modern multimedia systems could not exist without those kind of embedded systems. The algorithms are all characterized by a repeated application of some functions on a stream of input values, eventually resulting in a stream of output values. Important parameters from the application domain which, strongly influence the synthesis method, are throughput and synchronicity.

In this thesis we focus on high-throughput applications. With high-throughput, the stream of signals arrive at a high speed and the inter arrival time of individual signals or packet of signals is shorter than the processing time of an individual signal or packet of signals. Real-time constraints play a crucial role in the design of high-throughput *DSP* applications. The available hardware must be able to perform the specified function at any time and with a sufficient small latency. By small we mean in the order of 100 time steps. In case the input is provided as a synchronous stream of *equally sized* blocks and the latency of the algorithm is not a data dependent, current synthesis tools

for high-throughput applications are available which give (near) optimal schedules. The algorithms within those synthesis tools [4] tend to find a near optimal static schedule using integer linear programming. Unfortunately, if the input streams are not fully synchronous or the algorithms of the high-throughput application have a data dependent execution latency, there are no optimal synthesis tools available.
Scheduling and resource allocation in that case rely on heuristic algorithms. A second problem for those kind of synthesis tools is that they are only suitable for parts of a design and hence they have to cooperate with other tools to provide a total solution. In this research we aim at diminishing those disadvantages, providing design solutions for algorithms with data dependent latencies and handle irregular input data streams, without loosing the advantage of being able to determine optimal schedules.

1.1.1 Throughput

We distinguish between synthesis methods for high- medium- and low-throughput. This distinction is related to the internal speed (i.e. clock speed) that can be obtained with the available technology in relation to the arrival rate of the input sample. Consider the internal clock delay of the processor to be Δ_p and the inter-arrival time of the input samples to be Δ_t. In case $\frac{1}{\Delta_p} < 100 \times \frac{1}{\Delta_t}$ hence $\frac{\Delta_t}{\Delta_p} < 100$, we talk about high-throughput. In case this ratio is 100 to 1000 we talk about medium-throughput and if this ratio is more than 1000 about low-throughput. Because in medium and low-throughput applications the possibility to re-use the functionality provided by the hardware is very high, the implementation in general will lead to one or more von Neumann like processors with a targeted arithmetic and logical unit. So, some program and program store can be distinguished.
In high-throughput applications the re-use of hardware is small. Therefore the implementation in general consists of a number of dedicated processing elements interconnected by a dedicated data path in which the data flow is controlled by a dedicated controller [5].
Typical high-throughput applications are algorithms for video signal processing, typical medium-throughput applications are algorithms for audio signal processing and

a typical low-throughput application is remote control.
From the preceding it becomes clear that synthesis methods for high-throughput applications are quite different from synthesis methods for medium and low-throughput applications.
This research proposal aims at the improvement and extension of the applicability of synthesis tools for high-throughput applications.

1.1.2 Synchronicity

Real time constraints play a crucial role in the design of *DSP* applications. The available hardware must be able to perform the specified function at any time and with a sufficient small latency, for any combination of input streams.
In case the control flow of the algorithm is not data dependent and the input is provided as a synchronous stream of equally sized blocks, the influence of the real-time constraints on the design can be easily determined. In that case, for high-throughput applications, it is possible to calculate optimal schedules by means of integer linear programming [8],[9].
However, in case the inter-arrival time of input samples is irregular and the program flow is data dependent or even an interrupt mechanism needs to be provided, it becomes very difficult to determine the worst case real-time properties and consequently scheduling and resource allocation become difficult. Scheduling and resource allocation in that case rely on heuristic algorithms.
Only proprietary tools are available for high-level design of high-throughput digital signal processing applications [5][4]. The viability and the efficiency of these design tools have been proven by the design of a number of very large and innovative VLSI designs [6],[7]. However, these tools can only be applied when the input is provided as a synchronous stream of equally sized blocks and they do not allow a data dependent control flow. Unfortunately, many specifications do not satisfy these constraints. Moreover, in most cases, only a part or parts of a chip design fall into the category of high-throughput applications.
In this thesis we aim at diminishing these disadvantages without losing the advantage

of being able to determine optimal schedules.

1.1.3 The design process for high-throughput *DSP* applications

The design flow for high-throughput *DSP* applications in most cases starts from an algorithmic specification, often given in C. Such a specification in C is the result of the design activities of the system designer, who uses a simulation environment in C in order to validate the algorithm.

As has been explained before, high-throughput applications tend to an architecture built from dedicated processing elements, a dedicated data path and hardware re-use implemented by a dedicated controller which controls the data flow.

The first step in the design process is to identify the functions in the algorithm which are most suitable for implementation as processing elements. This is a creative process, performed manually by the designer. These processing elements are normally described in *VHDL*, which allows further refinement of the design of the processing elements by means of commercially available tools.

Given the processing elements and the algorithm in C, the behavior expressed by the algorithm is expressed using an *application graph*. In the *Phideo* tool set developed by Philips research [5] the application graph was written by means of a signal flow graph language called *PIF*. The functions in the *PIF* description correspond to the functions performed by the processing elements. The entire translation from the algorithm in C to a set of processing elements each described in *VHDL* and a description in the signal flow graph language *PIF* is a creative process which is performed manually and therefore very error prone. Once the description of the processing elements and the description of the algorithm in *PIF* are ready, effective tools for scheduling and resource allocation are available. These tools are dedicated to the high-throughput *DSP* application domain and are able to determine optimal schedules by means of integer linear programming.

Furthermore, a tool exists which is able to create the required controller from the schedule. Starting from the controller and the processing elements, the final chip is

developed by means of commercially available tools.
The viability and the efficiency of this design flow have been proven by the design of a number of very large and innovative VLSI designs [10].

1.2 Problem Formulation

Unfortunately, the algorithms for scheduling and resource allocation that are based on integer linear programming put severe restrictions on the kind of algorithms which can be treated in the design process sketched above.
All repetitions in the algorithm should be of a fixed number. Consequently the input streams must be fully synchronous and must consist of fixed length blocks.
Clearly the algorithms may not contain conditions. So data dependent loops or algorithms that are data dependent and produce a variable computational load, cannot be treated. The latter type of algorithms are called non-manifest.
Moreover, it is difficult to embed the design results into medium and low-throughput designs and to interact with these designs. In some cases it is possible to work around these restrictions, but in many cases this is not possible for two reasons. Firstly because the optimal schedule depends on the number of repetitions of the different loops in the algorithm and secondly because the signal flow graph model is too restricted to describe irregular data streams.

1.3 Proposed Solution

In this thesis we show the feasibility of Coarse Grained Data Flow Machines for high-throughput streaming non-manifest applications.

In this proposed architecture, the High^2 *DFM* which is derived from the classical data flow architecture, the scheduling is done dynamically in hardware. Since the implementation of such an architecture is strongly application dependent, we provide a design flow and supporting software by means of which the processing elements, buffer sizes and latencies can be tuned.

Applications of the High^2 *DFM* are modeled by an *application graph* where the nodes of the *application graph* represent the algorithms or functions of the application and the edges of the *application graph* represent the data or operand dependencies between the nodes.
To avoid the problems of the classical data flow machines, operations of the High^2 *DFM* are coarse grained by design. In this way the computation to communication ratio and the architectural overhead is acceptable. The *DFM* contains multiple execution units and each execution unit may contain more than one processing element. All the processing elements that belong to an execution unit perform the same operation. This allows the *DFM* to handle an input stream, that does not necessarily have to be synchronous nor do the block lengths of the input samples have to be regular.

1.4 Thesis Outline

Chapter 1: Introduction.

Chapter 2: In this chapter the models and definitions used within this thesis are given. The chapter handles models and definitions related to the *environment* in which a high-throughput streaming *application* has to operate and the *architecture* of the proposed processor.

Chapter 3: In this chapter a description of non-manifest algorithms is provided. There are a number of reasons that cause an algorithm to be non-manifest. The algorithm can have a loop which is data dependent, hence the input data determines the number of loop iterations or it can have data dependent control hence there could be more than one execution path, each with its own latency. A number non-manifest algorithm examples are given, each describing a separate problem. Some algorithms are characterized by having variable iterations of a loop body depending on the operands provided others are characterized by having data dependent control. In order to generalize the model of non-manifest algorithms, we choose to use the number of consumed clock cycles as a measure of the algorithm latency. The number

of consumed clock cycles will vary based on the implementation technology. Hence, designers of the proposed processing elements should profile the non-manifest algorithms on the actual processing element implementations. In the design process of application specific processors, knowledge of the application and its environment is used to design the processor. Important design parameters are, the maximum workload of a single computation and the throughput of the input stream. The stream throughput is a specification requirement and the maximum workload is obtained by profiling the algorithm using realistic data streams.

Chapter 4: In this chapter various dynamic scheduling architectures from literature are explored. A comparison is made between static scheduling and dynamic scheduling techniques. The chapter covers concepts such as the Tomasulo scheduling algorithm, the Scoreboarding algorithm, data flow machines, and pipelining. The Tomasulo and Scoreboard scheduling algorithms schedule fine grained instructions dynamically in hardware. The classical dataflow architecture, also handles fine grained instructions, and thus suffered from architectural overhead problems. Despite those problems the classical dataflow machines had, the dataflow model of execution has attractive properties for high-throughput streaming applications and hence is a motivation for the High^2 dataflow architecture discussed in chapter 5. Those problems were circumvented by adapting the model of fine grain data flow processing to coarse grain processing.

Chapter 5: In this chapter we explore processor design solutions for applications with non-manifest algorithms. Two kinds of application models are classified. Those models are called the *simple model* and *complex model*. In the simple model the application consists of one non-manifest algorithm that has to operate in a streaming environment. The stream samples are independent (each computation depends only on one operand) and form the operands of the application. They arrive with an interval that is much shorter than the execution time of a single operand. Clearly having one processor is not enough to handle this stream load. An architectural solution of a dedicated processor for the simple model is proposed. This dedicated processor

contains many processing elements. Each processing element implements the exact same functionality and the operands of the stream are dynamically scheduled on them. Since the execution time is variable and is based on the actual data content of the operand, the results of the processing elements can over take each other in time. This property is called out-of-order execution. The proposed processor makes use of this property and the property of having many processing elements to cope with the high-throughput stream requirements.

Chapter 6: Chapter 6 describes the design method and tools used for developing applications using the High^2 *DFM* as a target architecture. Using this design method the required system parameters, such as number of processors, memory sizes and system latencies are derived. The chapter describes the design methodology by means of an example. It turns out that the number of processors required can be scaled, and the system can be tuned to the required input processing load. It also turned out that variations in the input throughput, while the load is constant, do not influence the performance of the High^2 *DFM* while the same variations for a statically scheduled architecture may lead to non-optimal scheduling solutions, and worst case requirements for the number of required processing elements.

Chapter 7: Conclusions.

Chapter 2

Models and Definitions

In this chapter we provide the models and definitions needed to describe the material of this thesis.

2.1 Introduction

A high-throughput streaming application usually operates in an environment where continues processing of information takes place and lots of operations, between consecutive input stream samples or data elements, have to be performed. The application usually performs a set of algorithmic functions or operations on a continues stream of input data. It is those set of algorithmic functions, their appropriate interaction and the underlying environment in which they have to operate, that form the characteristics and the limitations of the application.
In order to achieve best results from such applications we need to build hardware architectures which are best suited for such applications, taking into account the environmental aspects in which they have to operate, characteristics and limitations of those class of applications. This is done by analyzing, defining and modeling each aspect separately.
In the rest of this chapter we proceed by analyzing the following subtopics separately:

- Environment

- Application
- Hardware architecture

2.2 Environment

The environment in which an application is to operate has a major influence on the choices designers have to take. We consider only streaming applications, as this thesis is focused on researching such applications and architectures targeted for such applications. In a streaming environment, the life cycle of the application can be seen as consuming and processing input data-samples, and producing output samples.

One restriction that is imposed on high-throughput streaming applications, is that the implementation of the application has to be able to process the input stream within a bounded time interval, also known as the latency of the application. The term *high-throughput* in this thesis refers to the fact that such applications have to keep up with the *high-throughput* of the input stream and in order to do so the system must have sufficient processing power. Sufficient means that the processing power available in the system is more than the processing load provided by the input stream. One way to achieve the high-throughput and latency requirements, is by a highly parallel multiple instantiations of the algorithms of the application. This means that standard processor architectures, which are based on the *Von Neumann* topologies are not suited here for, as they are not capable of performing multiple parallel operations.

The data stream arrives at the inputs, of the application, with either a fixed time interval or variable time interval. If the data elements arrive with a fixed input time interval we say that the system operates in a constant input-stream environment. Finite Impulse Response (FIR) [44][45] filters are a typical example of a constant stream application, the data samples from the analog to digital (A/D) converters arrive at the filter with a constant sampling rate. On the other hand if the data elements arrive at variable time intervals, we say that the system operates in a variable input-stream environment.

input stream	output stream	naming convention
constant	constant	constant stream-environment
constant	variable	constant input variable out stream-environment
variable	constant	variable input constant output stream-environment
variable	variable	variable stream environment

Table 2.1: Different types of streaming environments

A wireless communication medium can be considered an example of a variable stream environment as the data packets or samples can be lost, due to the influences of noise or the medium in which the mobile is operating, or even due to transmission range constraints, and hence delivery is not guaranteed. Playing an MPEG movie from the hard disk of a personal computer could also be considered as a variable input-stream environment, since the input data rate is scene dependent and delivery of the data samples can be hindered by interrupts or other processes which are running on the personal computer.
There are also variations of variable or constant input-streams and variable or constant output-streams. Table 2.1 gives a summary of the different stream-variations and their naming conventions. We shortly summarize the characteristics and properties of the environment that influence the design as follows:

- The way in which the data stream is delivered to the inputs of the application (constant stream / variable stream)
- The way in which the data has to be produced at the outputs of the application (constant stream / variable stream)
- The data granularity (bits, bytes, words, or larger blocks)
- The input and output throughput requirements of the data stream
- The maximum latency between input / output

Stream throughput is the speed of arrival or delivery of the data samples (which are the operands of the functions) and is normally the reciprocal of the average sample interval time. Data values or instruction operands are carried by tokens. In case

of a constant stream the throughput can be expressed as the sample frequency, i.e. the number of samples per second which is δ^{-1} if δ is the time that elapses between two successive samples (the operandinterval). Figure 2.1 gives a simplified model of the environment where streaming applications have to operate, in this figure the environment is a constant stream environment. The input tokens arrive at regular time intervals with a constant time delay between them and the output tokens are also produced with a constant time delay. **Note:** the output inter-token delay does not have to be the same as the input token interval.

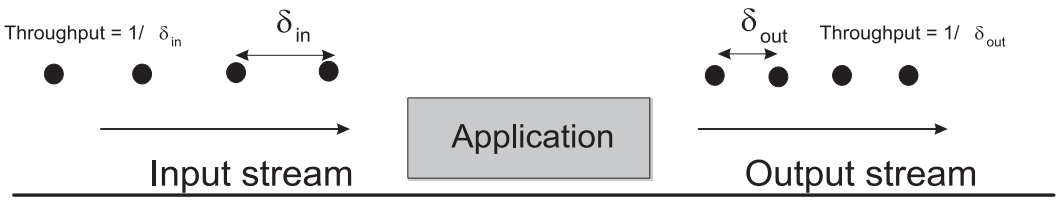


Figure 2.1: Application within a constant stream environment

In figure 2.2 we see the same application but within a variable streaming environment. The input tokens arrive at random or variable time intervals, and hence the inter-token delay is also variable.

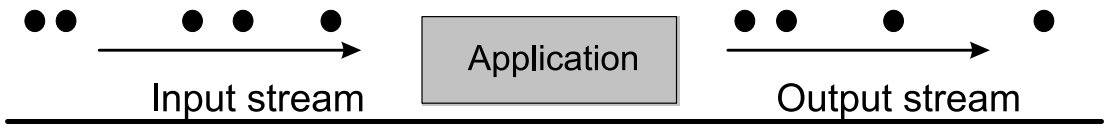


Figure 2.2: Application within a variable stream environment

In order to formalize the concepts just mentioned we give the following definitions: The stream is in the form of data elements which can range from a few number of bytes to blocks of mega bytes. Data elements which belong to a single computational instance are called tokens and are denoted by p. There is an ordering relationship between the tokens of a stream. This ordering relationship dictates that each token of the stream must have a successor and a predecessor.

Tokens are modeled by the set of integers $\mathbb{I}$, each token i, $i \in \mathbb{I}$ has a time stamp $time(i)$, $time(i) \in \mathbb{R}$ and a value $val(i)$, $val(i) \in \mathbb{V}$, in which $\mathbb{V}$ is some set of values. Hence $time \in (\mathbb{I} \rightarrow \mathbb{R})$ and $val \in (\mathbb{I} \rightarrow \mathbb{V})$ where $time$ is a monotonic increasing function, i.e. $i < j \leftrightarrow time(i) < time(j)$.

Definition 1 (stream)**.** An unbounded and ordered set of tokens, where no two tokens can exist at the same time is called a *stream* and is denoted by S. Hence $S = -\infty \cdots p_{i-1}, p_i, p_{i+1} \cdots \infty$,

Note: In practice the value of the tokens range from small sizes such as samples produced from a high speed A/D converter to large blocks of data depending on the producing instrument or device.

Definition 2 (streaming environment)**.** The environment in which an application operates is called a streaming environment, if the tokens the application operates on, arrive as a data stream and/or the application produces its output data as a stream.

Definition 3 (token interval)**.** The token interval $\delta(i)$ is the time interval between two consecutive tokens P_i, P_j and hence $\delta(i) \equiv time(i+1) - time(i)$. Because time is a monotonic increasing function, $\delta(i) > 0$ for all $i, i \in \mathbb{I}$.

Definition 4 (variable streaming environment)**.** A streaming environment is called a variable streaming environment if the tokens do not have a constant time interval δ_i. Hence $\exists i, j : i, j \in \mathbb{I} \wedge \delta(i) \neq \delta(j)$.

Definition 5 (constant streaming environment)**.** The data stream whether consumed or produced by an applications is called a constant stream, if the time delay δ between two consecutive tokens has a constant time value. Hence $\forall i : i \in \mathbb{I} \Rightarrow \delta(i) = \delta$

One major parameter in streaming applications is their input and output throughput. The throughput basically denotes the speed of data tokens. In the simple situation of constant-stream environments we can say that the throughput is the reciprocal of the inter arrival time delay δ. In order to generalize the definition of throughput for both constant and variable-stream environments, we introduce the concepts of maximum throughput and average-throughput based on a sliding window of input

tokens. Sliding windows are used in order to model the input and output mapping relationships.
Since it is the application at hand which dictates the input output relationship, we generalize the definition of throughput. From system theory [15] we know that a system is called an input-output system, if some of the systems signals are designated as inputs and some as outputs. A relationship or rule interrelates the input and output signals.

Definition 6 (Input-Output systems)**.** An *input-output* **(IO)** system is defined by a signal set $\mathbf{U}$ called the input set, a signal set $\mathbf{Y}$ called the output set, and a subset $\mathbf{R}$ of the product set $\mathbf{U} \times \mathbf{Y}$, called the rule or relation of the system. Any pair (u, y) with $u \in \mathbf{U}$, $y \in \mathbf{Y}$, and $(u, y) \in \mathbf{R}$ is said to be an input-output pair of the system, with u the input signal and y a corresponding output signal.

Definition 6 states that for each input value u there is a set $y_u = \{y \in \mathbf{Y} | (u, y) \in \mathbf{R}\}$. If for each input u there exists a single corresponding output y, then the system is called an *input-output mapping* system. I.e. $\mathbf{R}$ is a function. The sets $\mathbf{U}$ and $\mathbf{Y}$ may be of any kind. In practice they will contain streams, hence $\mathbf{U} \in (\mathbb{I} \to \mathbb{V})$ in which $\mathbb{I}$ identifies the tokens and $\mathbb{V}$ the set of all values that can be bound to the tokens.
Further a system is called *memoryless* if the current output of the system at each time instance is fully dependent of the current input value alone and not by past or future values of the input. A system is called a *memory system* or *state system* if it is not memoryless.
A system is called *time-invariant* if a time shift in the input signal causes an identical time shift in the output signal.
A system is called *causal* if the output of the system is independent of future values of the input signal hence dependent only on the previous inputs of the system. Then what does in-dependability actually mean? We say that for the function $Z = F(x_1, \cdots, x_n, y_1, \cdots, y_m)$, Z is independent of $x_1, \cdots x_n$ for $y_1, \cdots, y_m$ iff
$[\forall x_1, \cdots, x_n, x\prime_1 \cdots x\prime_n : F(x_1, \cdots, x_n, y_1 \cdots y_m) = F(x\prime_1, \cdots, x\prime_n, y_1 \cdots y_m)]$

Definition 7 (causality)**.** Let a system be described by $y = F(x)$. in which x and y are streams, i.e. $x, y \in (\mathbb{I} \to \mathbb{V})$.

Let $x_{\leq t}$ and $x_{>t}$ be defined by:

$$\begin{aligned} x = x_{\leq t} + x_{>t} \quad and \quad & x_{\leq t}(i) = 0 \quad if \quad time(i) > t \\ & x_{>t}(i) = 0 \quad if \quad time(i) \leq t \end{aligned} \tag{2.2.1}$$

Then the system F is called causal if

$$\forall_t : \forall x_{\leq t}, x_{>t}, x\prime_{>t} : F_{\leq t}(x_{\leq t} + x_{>t}) = F_{\leq t}(x_{\leq t} + x\prime_{>t}) \tag{2.2.2}$$

In this thesis we restrict our self to the class of systems which are causal and time-invariant. Whether the system is memoryless or not is not an issue as we will cover both situations although, in most real applications, we are limited by the memory requirements of the design and hence systems with an infinite memory requirement are not allowed.

Due to the class of systems we cover, an output token can be dependent upon a limited number of consecutive input tokens, and in the most simple case just one token. We are interested in the throughput and latency requirements of such systems and in order to generalize the model we redefine the throughput formally by introducing the concept of stream windows which is given in definitions 8 and 9. Stream windows can be defined in two ways: first it can be defined as a bounded time interval which contain one or more tokens or it can be defined as a variable time interval (non zero interval) which contains a constant number of tokens.

In the rest of this thesis we assume that in the case of variable streaming environments, the input or output stream has an upper-throughput bound and no lower-throughput bound. Hence there is a maximum input throughput parameter for the designers of the application at hand.

Definition 8 (fixed time stream window)**.** A bounded time interval τ starting from the initial time t is called a *fixed time stream window* and is denoted by $W_{time}(t, \tau)$.

Definition 9 (fixed token stream window)**.** A variable time interval starting from the initial time t and which contains exactly m tokens is called a *fixed token stream window* and is denoted by $W_{token}(t, m)$.

Based on definitions 8 and 9 we can define the following functions: $N(W(t, \tau))$ which gives the number of tokens available within a fixed time window stream and $T(W(t, m))$ which will give the time interval of a fixed token stream window. Now we expand the *throughput* definition given in definition 10 and introduce the concepts of maximum, minimum, and average throughput.
Based on the above window definitions we say that throughput is a function of window length and time.

Definition 10 (Throughput)**.**

$$Throughput(t, \tau) = \frac{N(W(t, \tau))}{\tau} \tag{2.2.3}$$

Definition 11 (minimum token delay)**.** The minimum time delay between two consecutive tokens within the stream window $W(t, \tau)$ is $\delta_{min}(t, \tau)$ and is formally defined as follows:

$$\begin{aligned} &[\exists i : t \leq i < t + \tau \wedge \delta_{min}(t, \tau) = time(i + 1) - time(i)] \\ \wedge &[\forall j : t \leq j < t + \tau \Rightarrow time(j + 1) - time(j) \geq \delta_{min}(t, \tau)] \end{aligned} \tag{2.2.4}$$

Definition 12 (maximum token delay)**.** The maximum time delay between two consecutive tokens within the stream window $W(t, \tau)$ is $\delta_{max}(t, \tau)$ and is formally defined as follows:

$$\begin{aligned} &[\exists i : t \leq i < t + \tau \wedge \delta_{max}(t, \tau) = time(i + 1) - time(i)] \\ \wedge &[\forall j : t \leq j < t + \tau \Rightarrow time(j + 1) - time(j) \leq \delta_{min}(t, \tau)] \end{aligned} \tag{2.2.5}$$

Definition 13 (Average token delay)**.** The average token delay of the stream window $W(t, \tau)$ is $\delta_{avg}(t, \tau)$ and is formally defined as follows:

$$\delta_{avg}(t, \tau) = \frac{\tau}{N(W(t, \tau))} \tag{2.2.6}$$

Note: that definition 10 basically dictates that throughput is $\frac{N(W(t,\tau))}{\tau}$ and, for the case of constant stream windows, if τ is the token time delay δ then $throughput = \frac{1}{\delta}$. In real applications there is a bound on the input throughput. Also the output throughput is bounded by the latency of the application. We say the that the system is *BIBO* stable if it has a Bounded Input throughput and a Bounded Output throughput. Now that we have defined the token delays and the throughput, we commence by defining the latency of the application.

Definition 14 (latency)**.** The latency of a *BIBO stable* application is the time taken to process an output token, given all its required input tokens under the input and output throughput constraints of the environment. The latency is denoted by Lat.

The throughput constraints of the environment and the specification of the application determine the latency specifications of the application. We consider for the class of applications mentioned in this thesis, that the latency is a design parameter and that its constraints are obtained from the environmental throughput constraints and the computational power available for the implementation.

2.3 Application

The next part of the problem analysis is the application itself. If we consider the application on its own without its surrounding environment, it will become obvious that it can be modeled in many ways. In this thesis we develop our own model, as we believe that this model will fit best to the solution approaches presented later in this thesis.

In our approach we model a streaming application as a, *modified*, directed acyclic graph called the *application graph*, where the input tokens d_i will arrive at the start nodes of the *application graph*. We call the model *modified* because there is one extra property added which is not covered by the standard graph definition. In a graph model, ordering of the inputs is not covered by the definition. In real functions and algorithms, ordering of the inputs is eminent, and hence in order to allow for this property we will tag the inputs of a node in order to distinguish between them. The

tagging scheme used basically numbers the ports of a node and distinguishes the inputs from outputs.

Definition 15 (Application Graph "AG")**.** The streaming application is modeled as a modified directed acyclic graph DAG called $AG = \langle V, E \rangle$ where V is the set of nodes representing the functions of the applications and E is the set of edges connecting those nodes. The set of nodes V contains at least two special nodes an input node and an output node. The input and output ports of a node are tagged by a number and a letter in order uniquely identify them, distinguish between inputs/outputs and to allow for the input and output orderings of the function modeled by the node.

Note: Many although not all applications can be modeled in such a way. The *application graph* definition used does not allow for cycles, as they will affect the synchronization mechanism described later in this thesis. The nodes of the *application graph* perform functions on the token values (operands), such that $out(i) = F(in_1(i), in_2(i), \ldots)$ in which $in_k(i)$ denotes the value of token i.

Functions of an application can perform their computation in a constant time or variable time. In this thesis we make a distinction between two kind of functions based on their execution behavior, functions which have an input dependent execution time and functions which have a constant execution time no matter what the input value is. Basically the execution, in a certain implementation technology, of a function commences by triggering a number of operations until the output result is obtained. Searching the age of a person in a data base can also be considered as a function which maps a given name of a person from the set of names (name domain) to the age of that person from the set of $\langle name, age \rangle$ tuples (co-domain). One implementation could be obtained by iterating the complete list of $\langle name, age \rangle$ combinations, the result will be found in a fixed constant time and the number of operations performed in order to find the result is also a constant. Another implementation could be performed by searching the list until the result is found and then skipping the rest of the list, hence the implementation of the function will consume a variable number of operations, based on the name (input) given to the search function.

In order to model this property we have to distinguish between the various implementation characteristics based on their execution behavior. We say that an implementation of a function is *data dependent non-manifest* if it can consume a variable number of operations during its execution and that this variation is based on the input value provided. On the other hand we say the implementation of a function is manifest, if it will consume a fixed number of operations during its execution no matter what the input value is. We give the following definitions in order to formalize this property.

Definition 16 (non-manifest or variable latency function)**.** An implementation of a node V of the *application graph* is called a *non-manifest* functional node, if V is a function of the *application graph* and the number of operations consumed during its execution is dependent upon the input data $d(i)$ provided.

Furthermore the execution of a non-manifest function is bounded between a minimum and a maximum number of cycles.

Definition 17 (manifest or fixed latency function)**.** The implementation of a node V of an *application graph* is called a *manifest* functional node if under all conditions its execution time is constant and independent of its input operands.

Note: In the implementation of functions there are many reasons which make the implementation non-manifest, *if-then-else* in the function body could be one of those reasons, but also input-dependent *while loops*. In this chapter, we will not go in the programming constructs and details which cause the implementation to be non-manifest, as this is covered in chapter 3.

From the above definitions we can deduce that each implementation of function V within the *application graph* has a computational load and we specifically define the computational load of a non-manifest function to be as follows:

Definition 18 (computation load)**.** A data dependent non-manifest function V generates a *computation load* of $CL_V(d_i, d_j, \ldots)$ cycles, depending on its input token values $d_i, d_j \ldots$.

Obviously for function V the following holds:

$$\forall d, d \in \mathbb{I} \qquad CL_{min_V} \leq CL_V(d, \ldots) \leq CL_{max_V}. \tag{2.3.1}$$

In the design of high-throughput streaming applications, we are interested in the work load generated by the stream samples. Hence in order to model this property we define the workload of a streaming window as follows:

Definition 19 (Workload of a streaming window)**.** The workload generated by a streaming window of length m is called $WL(t, m)$ where:

$$WL(t, m) = \sum_{i=t}^{t+m-1} CL_A(i) \tag{2.3.2}$$

$WL(t, m)$ is the workload generated by the tokens $t, t+1, \ldots, t+m-1$ on a processing element A and *sliding window* $t, t+1, \ldots, t+m-1$ of tokens t.

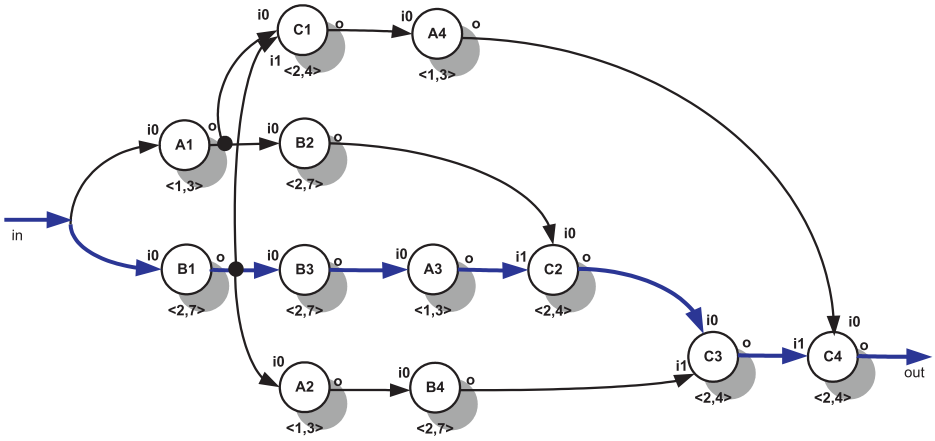


Figure 2.3: An example of the *application graph*

In figure 2.3 we notice that an application is represented as an *application graph* were the nodes of the *application graph* are the functions of the application, the edges

represent the data communication. Nodes of the application are augmented by a minimum and maximum latency in iteration cycles or operations. The inputs of the nodes are tagged in order to preserve the ordering and distinguish the inputs from the outputs. Based on the application type it may (or may not) have multiple functions of the same execution type. If a function of the *application graph* is manifest it will have the same value for its minimum and maximum latency.

Architecture

In the synthesis process we map functions specified in the *application graph* to physical processing elements. Those processing elements are controlled by a clock to synchronize their communication and interaction.

Definition 20 (Processing element)**.** The hardware implementation of a manifest or non-manifest function of an *application graph* is called a *processing element*.

Note: the synthesis process of an *application graph* will result in one or more processing elements depending on the number of functions within the *application graph*. A processing element which is an implementation of a node V, has a computation capacity which we define as follows:

Definition 21 (computation capacity of a processing element)**.** The implementation of a processing element PE of a node function V, consumes C_{res} computation clock cycles per operation.

For the rest of this thesis we assume that all resources of the same type have the same computation capacity, and that one time unit is equivalent to one computation cycle hence $C_{res} = 1$.
Note: For a non-manifest function, computation load is a property of the data and it is the data which causes the functions of an algorithm to consume a variable number of clock cycles during its execution.
In practice the nodes of an *application graph* will map to one or more processing elements. In the design, which will be presented later, it was not always practical to

have the same amount of processing elements as nodes within a *application graph*. So to abstract from this idea while still having the same functionality we have introduced the concept of an *execution unit* which is a collection of identical processing elements in one hardware design. Each execution unit has its own address and can be considered as a separate entity of the total design.

Definition 22 (Execution Unit (EU))**.** A collection of processing elements that implement the same functionality and their underling control and communication mechanism is called an *execution unit* denoted by *EU*.

The designers of an application can map the nodes from an *application graph* to an execution unit but they can not map them directly to the processing elements of this execution unit. As this step is done automatically by the scheduler of this execution unit at run time. Although this might be premature the exact details to the design of the execution units is given in chapter 5.

Definition 23 (Data Flow Machine (DFM))**.** A collection of *EUs* and their interconnection network, which realizes the node communication specified by the *application graph* is called a *DFM*.

From the above definitions we notice that the synthesis process of an *application graph* can result in a *DFM* which may have one or more execution units EUs, where each *EU* has one or more processing element PE. The EUs are connected by an interconnection network. This interconnection network mainly realizes the communication specified by the set of edges of the *application graph*.

Chapter 3

Non Manifest Algorithms

In this chapter we analyze the behavior of non-manifest algorithms. We show how to analyze the algorithm specifications and how to determine non-manifest behavior. Once non-manifest behavior is determined the designers of the system are left with a number of choices which can be exploited. Designers can choose for solutions which stretch the non-manifest behavior of the algorithm to a manifest behavior, hence providing the opportunity to build a processor architecture using static scheduling schemes. The other option is to build a processor architecture which can exploit the non-manifest behavioral properties using dynamic scheduling schemes. Both solutions have their pros and cons; it is the applications at hand and the requirements of the system that dictate what solution to take, hence we will only emphasize on the choices the designer has.

3.1 Introduction

In high-level synthesis of application specific processors (ASIP), algorithm specifications are needed in order to fine tune the implementation of the processors hardware elements. Algorithms can be classified as either manifest or non-manifest. In a manifest algorithm the number of loop body iterations is fixed and the delay of the loop body is constant. This implies that the total latency of the loop computation has a

constant number of clock cycles for a certain processor architecture. This property is used by the compiler tools of statically scheduled architectures, such as the PHIDEO architecture developed by Philips research[4], in order to plan and map the loop computations onto processing elements at compile time. The controller of such an architecture simply activates the required processing element at the pre-programmed clock value and handles the data transfers to and from the required operand memory addresses. In order to do so, the tool chain statically schedules the operations of the application taking into consideration the maximum number and type of processing elements (processing element constraints), the maximum amount memory allowed (memory constraints), and the maximum network bandwidth (communication constraints). The constraints are fed into an Integer Linear Programming (ILP) solver to find a valid (nearly optimal) solution. In order to function properly and find a valid solution the (ILP) solver requires the exact timing constraints of the loop operations of the application or algorithm at hand. This limits the architecture to applications with only manifest loops or inefficient implementations of applications with non-manifest loops. If an application contains loops that have delay variations due to control sequences or due to data dependencies a static schedule is only possible by scheduling the situation with the worst case delay.

In this section we demonstrate the behavior of non-manifest loops, by providing a couple of real life algorithms. We show how to analyze the algorithm specifications and how to recognize non-manifest behavior. Once non-manifest behavior is determined we can profile the algorithm and determine its execution-latency frequency distribution. The frequency distribution is used to estimate the expected workload and the maximum workload. As these parameters are used in the design process to determine the required system parameters, such as number of processing elements, total system latency, and memory of the processors such as described in chapter 5.

Once the non-manifest behavior has been characterized and the application algorithms are profiled, the designers of the system are left with a number of choices which they can exploit.

3.2 Examples of Non Manifest algorithms

In this section we provide a few algorithms with non-manifest behavior; For example the *GCD algorithm, Russian multiplication,* Cordic *rotation, and Montgomery inverse* algorithms. The properties that are of importance are :

- Maximum load generated by a single computation CL_{max}
- Latency in clock cycles of a single iteration of the loop body of the algorithm C_{res}
- Average load consumed by the algorithm when provided with an input stream of operands of known length

In order to obtain this information, the algorithms were implemented in C/C++ and compiled using the gcc compiler. Iteration counting code was added to the source code, and the algorithms were profiled exhaustively by using all possible input combinations. It seems that the number of iterations consumed, of the loop body, multiplied by the load generated by executing a single iteration of the loop body is an indication of the load consumed by a non-manifest algorithm. In turned out in practice that this model is not accurate enough, instead a profiling technique which involves adding profiling code around the control structure of the loop body is much more accurate. Non the less in the upcoming sections of this chapter will use the variations in the iterations of the loop to demonstrate the behavior and properties of non-manifest algorithms.

3.2.1 Greatest Common Divider (GCD)

We use the Euclid's *Gcd* algorithm to demonstrate the behavior of a non-manifest data dependent loop. The algorithm details are given in 3.1, the algorithm basically calculates the greatest common divisor or highest common factor g of a pair x, y of integers. Most modern factoring methods employ the *Gcd* algorithm inside, which is a real tribute to Euclid [1]. One of the properties of the *Gcd* algorithm algorithm is

[1]Euclid of Alexandria: born about 325 BC, died about 265 BC in Alexandria, Egypt. He is the most prominent mathematician of antiquity best known for his treatise on mathematics The

```
1  int gcd(int x, int y){
2      int g;
3
4      g = y;
5      while ( x > 0 ){
6          g = x;
7          x = y % x;
8          y = g;
9      }
10     return (g);
11 }
```

Figure 3.1: Gcd algorithm

```
##  iter     x        y        g  ##
      1 46368,   28657,   28657,
      2 28657,   46368,   46368,
      3 17711,   28657,   28657,
      4 10946,   17711,   17711,
      5  6765,   10946,   10946,
      6  4181,    6765,    6765,
      7  2584,    4181,    4181,
      8  1597,    2584,    2584,
      9   987,    1597,    1597,
     10   610,     987,     987,      maximum number of iterations for 16 bit integer values
     11   377,     610,     610,
     12   233,     377,     377,              is obtained by gcd(46368,28657)
     13   144,     233,     233,
     14    89,     144,     144,
     15    55,      89,      89,
     16    34,      55,      55,
     17    21,      34,      34,
     18    13,      21,      21,
     19     8,      13,      13,
     20     5,       8,       8,
     21     3,       5,       5,
     22     2,       3,       3,
     23     1,       2,       2,
            0,       1,       1,
gcd(46368, 28657) == 1,
num iterations == 23
```

```
##  iter     x        y        g  ##
      1     4,   32768,   32768,
            0,       4,       4,
gcd(4, 32768) == 4,
num iterations == 1
```

Figure 3.2: (a) Worst case execution versus (b) Best case execution of a Gcd algorithm for 16 bit integer values

that it has data dependent latency. The number of iterations of the loop depends on

Elements. The long lasting nature of The Elements must make Euclid the leading mathematics teacher of all time.

the input pair x,y as can be seen in figure 3.2. For 16 bit input values the maximum number of iterations is 23 and the minimum number of iterations is 1. Assuming that, for a specific processor implementation, each iteration has the same number of clock cycles in other words the delay of the loop-body (and hence for a single iteration) is constant[2]. A general non-accurate way to calculate this constant is to count the number of instructions of the loop body and assume that each instruction has a constant delay of one clock cycle. More accurate results are obtained by counting the number of *executed* instructions of a loop body. This involves profiling the application on the real processor. Based on this analysis we may assume the *Gcd* delay for one loop iteration $Cgcd$ is 3 clock cycles.

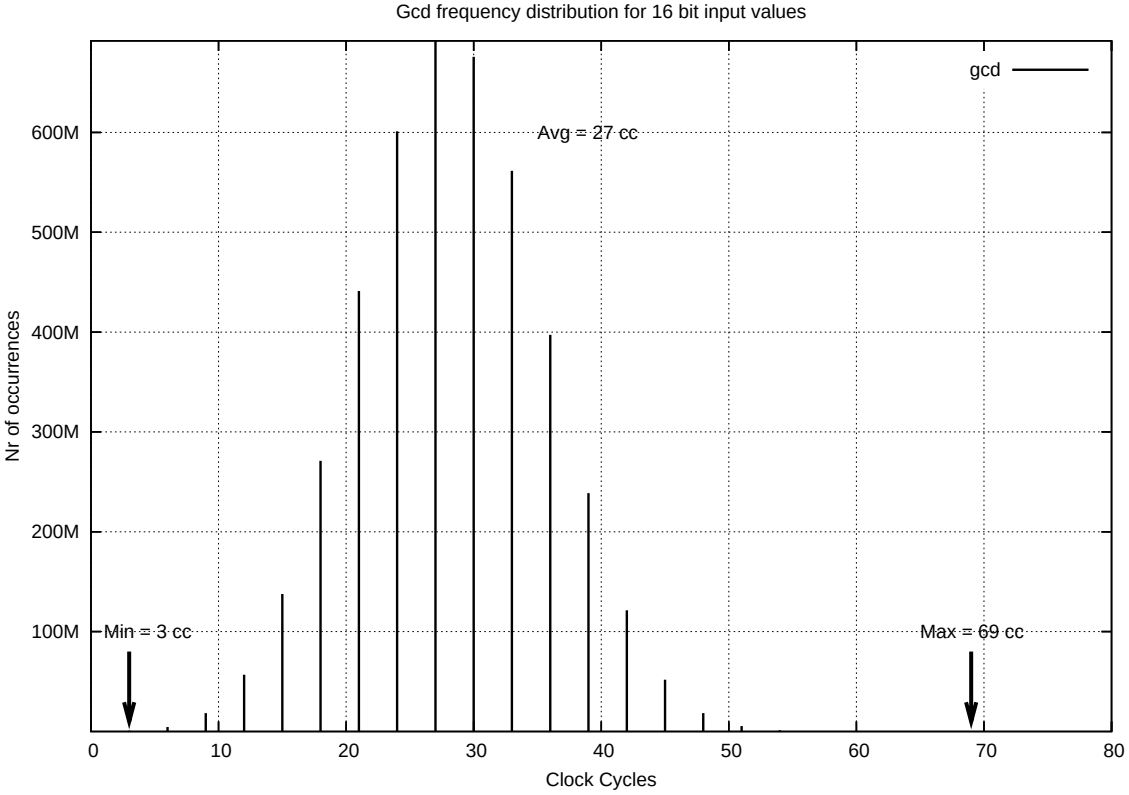


Figure 3.3: *Gcd* distribution for 16 bit integer values, each iteration taken Cgcd=3 clock cycles

Profiling the *Gcd* algorithm for all possible input combinations of 16 bit input values

[2]Off course this constant value depends on the type of processor used

results in the frequency distribution given in figure 3.3. The distribution shows on the x-axis the number of clock cycles the algorithm takes and on the y-axis the number of input combinations that, gave that amount of clock cycles. From the distribution we notice that the average number is around 27 clock cycles, the maximum amount of clock cycles is $23 * 3 = 69$ clock cycles.

Note: The amount of clock cycles is not predictable given the values of x and y (assuming $x \neq y$).

3.2.2 Russian Multiplication

```
1  #define odd(x)      (x & 0x1)
2
3  int rmult(int x, int y){
4      int g;
5
6      g = 0;
7      while ( y > 0 ){
8          if (odd(y)){
9              g += x;
10         }
11         x <<= 1;  // shift left
12         y >>= 1;  // shift left
13     }
14     return (g);
15 }
```

Figure 3.4: Russian Multiplication algorithm

The Russian multiplication algorithm (see 3.4) is another form of a non-manifest algorithm. The loop body has three instructions (including the if statement), hence the computation delay of a single iteration $C_{rmult} = 3$. The algorithm is non-manifest data dependent.

From the frequency distribution given in 3.5 we notice that the maximum number of clock cycles are to the right of the distribution. This is caused by the fact that the latency in clock cycles is proportional to the size of the input value which is the nature of the algorithm, larger numbers take more time to compute. This property

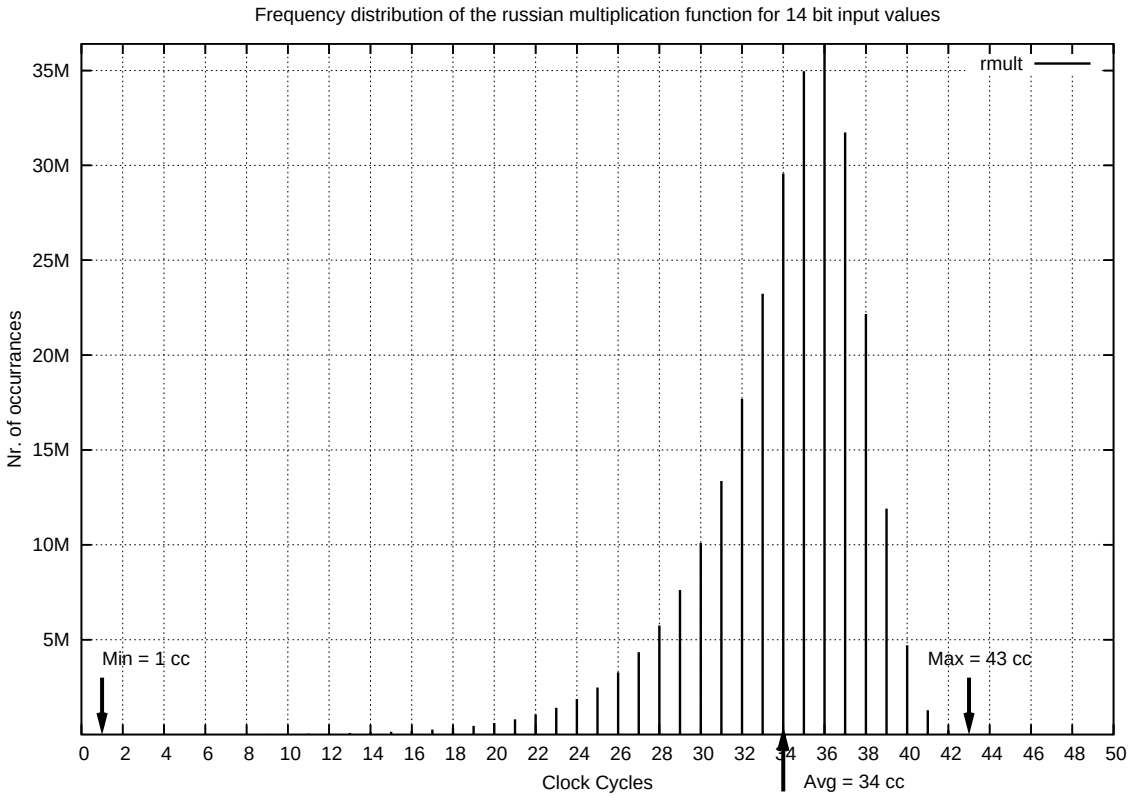


Figure 3.5: Profiling results of the Russian multiplication algorithm

makes the algorithm predictable. Designers can predict how long the algorithm will take based on the maximum number of bits that can represent the input numbers. One might argue that the *if-statement* in line 8 is not always taken and hence the latency of the loop-body is not always 3 clock cycles. This is true, but for the worst case situation the latency is 3 clock cycles. In order to obtain more accurate results, this algorithm should be profiled with the original data set of the environment in which it has to operate.

Note: if the application contains loops that are manifest or non-manifest with embedded control statements, the latency of the loop body would be variable at run time. Such loops are called variable latency loops.

3.2.3 Cordic Rotation

The algorithm presented in figure 3.6 is a version of the *Cordic* rotation algorithm [16]. The *Cordic* algorithm is extensively used in the area of signal processing for approximating trigonometric functions including sine, cosine, magnitude and phase. *Cordic* revolves around the idea of "rotating" the phase of a complex number, by multiplying it by a succession of constant values.

```
1  #define MAX_ITTER 32
2  #define APROXZERO 0.000001
3
4  double rotate(complex c){
5      double z, p, tmpx, k;
6      int l;
7
8      z=0;
9      l=0;
10     while ((l<MAX_ITTER) && (fabs(c.y)>=APROXZERO)){
11         p = phaseTab[l];
12         k = kTab[l];
13         tmpx = c.x;
14         if (c.y >= 0.0){
15             c.x += (c.y * k);
16             c.y -= (tmpx * k);
17             z += p;
18         }else{
19             c.x -= (c.y * k);
20             c.y += (tmpx * k);
21             z -= p;
22         }
23         l++;
24     }
25     return z;
26 }
```

Figure 3.6: Cordic rotation algorithm

However, the multiplies can all be powers of 2, so in binary arithmetic they can be done using just shifts and adds, and hence no actual multiplier is needed. This property makes *Cordic* suited for hardware implementations were no multipliers are

available. The *Cordic* rotation algorithm is a convergent non-manifest loop. This means that the loop converges to the correct answer and the more iterations are consumed the better the quality of the result is. The *Cordic* rotation algorithm was chosen because of its converging behavior which can be exploited by designers in order to tune the system quality parameters.

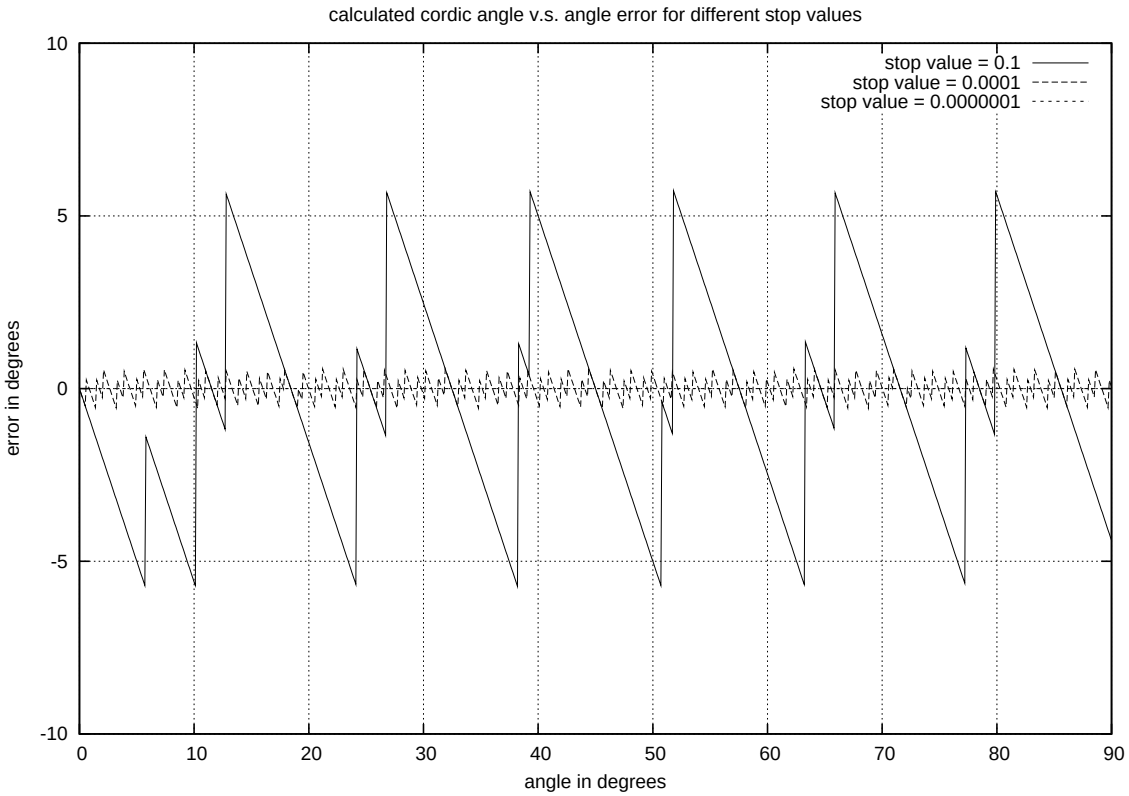


Figure 3.7: Calculated *Cordic* error v.s. stop value

In figure 3.6 the *Cordic* rotation algorithm that is profiled is shown. The algorithm stops when the absolute value of the imaginary component of the complex number c, hence the value of the variable $c.y$ (line 10 of the code example in figure 3.6) is less than the constant value $APPROXZERO$ or when the maximum number of iterations has been reached. The $phaseTab[i]$ is a lookup table for the $\tan^{-1}(\frac{1}{2^i})$ function and $kTab[i]$ is a lookup table for the $\frac{1}{2^i}$ function. Both are needed to determine the angle accumulation during the iterations of the loop.

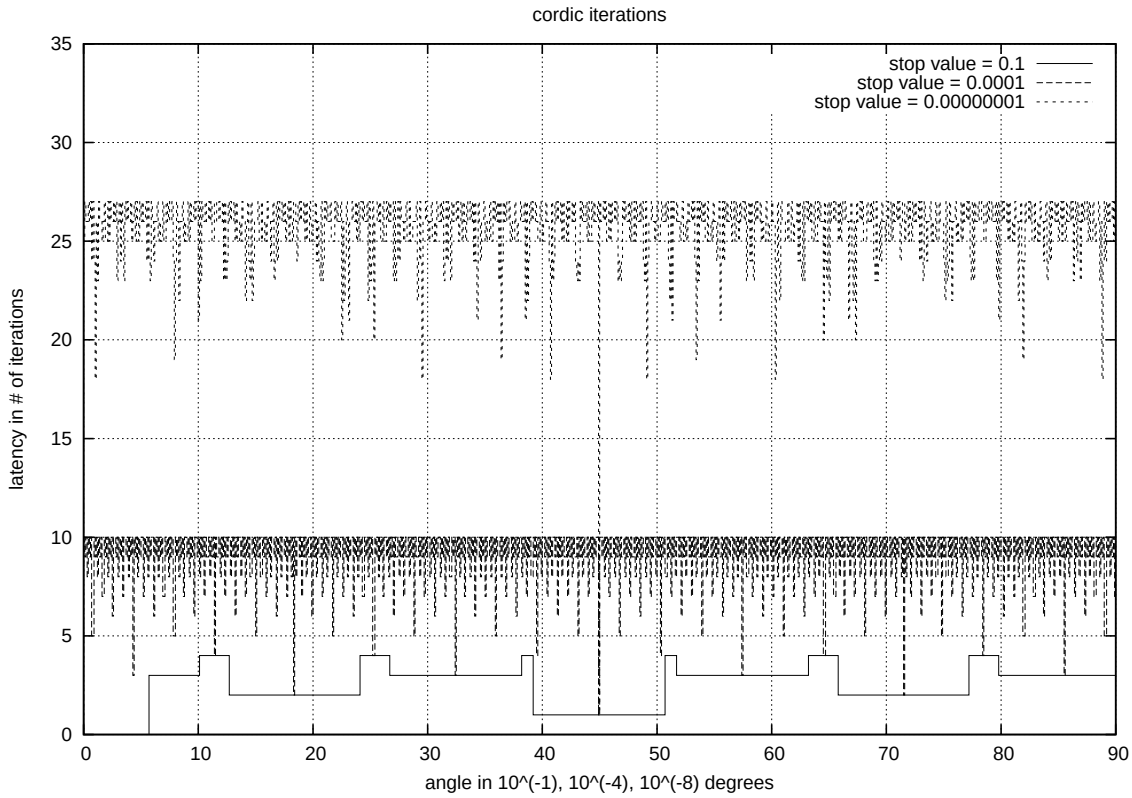


Figure 3.8: Calculated *Cordic* angle v.s. latency in iterations

In order to profile this algorithm, we generated all consecutive angles between 0.0 and 90.0 degrees using steps of 0.1 degree. Those angles were used to generate the complex numbers on the unit circle $c = r.(\cos(\phi) + i\sin(\phi))$. Hence for angle ϕ the real part of the complex number $c.x = \cos(\phi)$ and the imaginary part of the complex number $c.y = \sin(\phi)$ and $r = \sqrt{(}c.x^2 + c.y^2) = 1$. The generated complex numbers were then fed to the *Cordic* rotation algorithm which calculated the angle of the complex number and the loop iterations were profiled. The latency of the loop body of the *Cordic* algorithm is constant because both the latencies of the *if-part* statements $14 \ldots 16$ and *else-part* statements $18 \ldots 20$ are identical. The latency of the loop body can be estimated to $C_{cordic} = 7$ clock cycles, which is an approximate value for the number of instructions consumed during a single iteration.

Figure 3.7 shows the angle error made, in degrees, for stop values of $10^{-1}, 10^{-4}$ and 10^{-8}. Clearly as we can see in figure 3.7 the quality of the resultant angle is

based on the input stop value *APPROXZERO*. Notice that the resultant angles, for $APPROXZERO = 10^{-1}$, varies between $+/-$ 5.7 degrees. The actual error made is periodic and dependant on the resultant phase angle of the complex number c which is an input value to the algorithm.

In figure 3.8 we see, for various values of *APPROXZERO*, the number of iterations consumed for each input complex number. Note the angles of the complex numbers had the range $[0.0 \ldots 90.0]$ degrees. For $APPROXZERO = 0.1$ the loop of the algorithm varied between 1 and 4 iterations and for $APPROXZERO = 10^{-8}$ the algorithm varied between 1 and 27 iterations. For complex numbers with 45 degree angles and $APPROXZERO = 10^{-8}$, the number of loop iterations is always 1 and for complex numbers with 90.0 degrees angles, the number of loop iterations = 2, for real numbers (hence the angle is zero) the number of loop iterations is zero (this also holds for other values of *APPROXZERO*). All other angles have 18 as the minimum number of loop iterations.

Finally in figures 3.9, 3.10, 3.11 the iteration distribution for the stop values $10^{-1}, 10^{-4}$ and 10^{-8} are shown. The distributions were obtained by profiling the *Cordic* algorithm on the range of complex numbers described. Note that the distribution form is not similar, hence the value of *APPROXZERO* influences the form of the distribution. Also in figure 3.11 the maximum number of iterations 27 has the maximum number of occurrences. This may suggest, that static scheduling is the best way to schedule such applications since most of the input operands produce the maximum number of iterations. One should not be fooled by this assumption, since the input data set provided for profiling is whole range of complex numbers in the upper right quadrant of the unit circle. This range has a uniform distribution of angles. In order to obtain realistic distributions, one should profile the application with the actual data set provided by the environment in which the application will operate. Non the less application designers, which intend to use such type of algorithms, now have the quality of the result as an extra criterium in the design process. They can choose to limit the number of iterations hence sacrificing the quality of the results, in order to speed up the calculation.

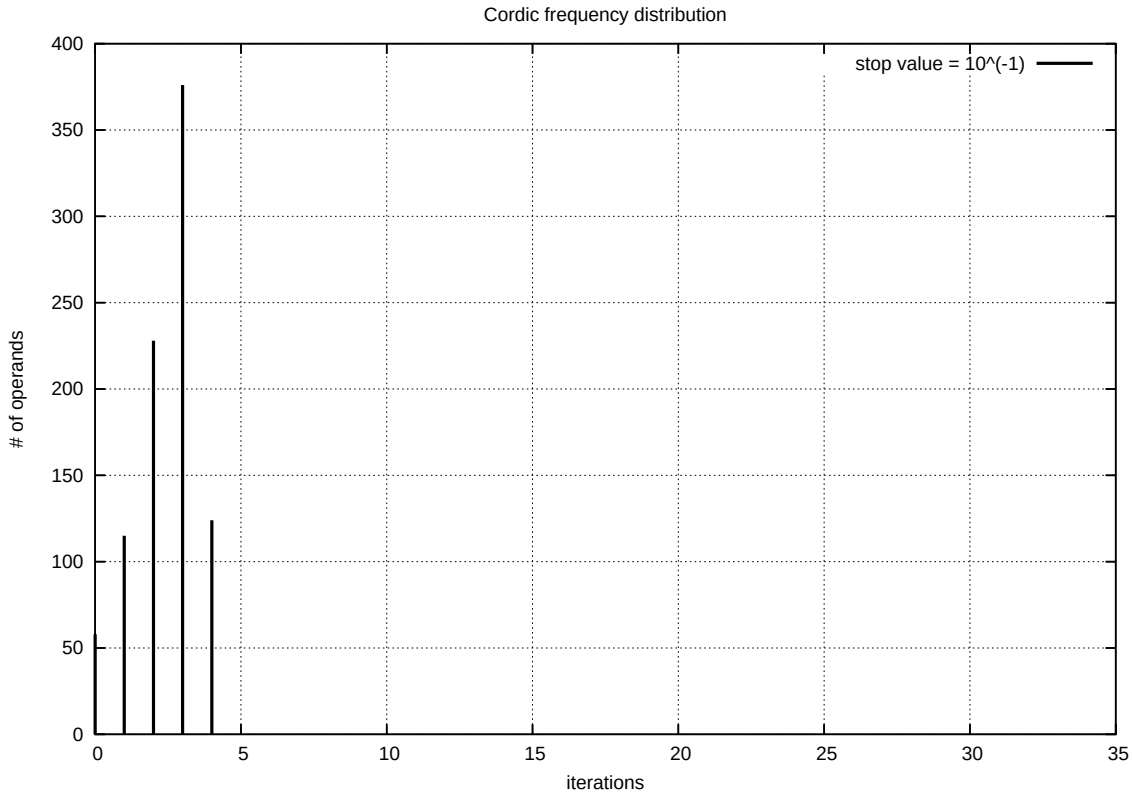


Figure 3.9: Calculated *Cordic* frequency distribution, stop value is 10^{-1}

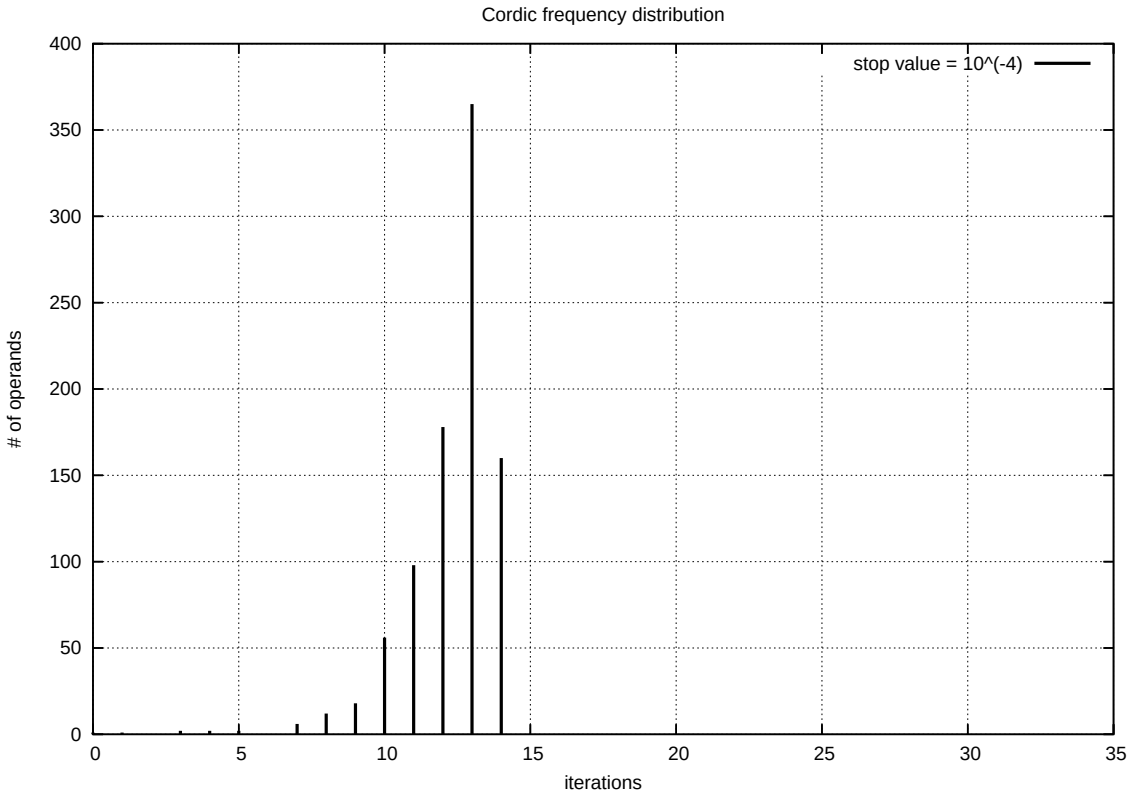


Figure 3.10: Calculated *Cordic* frequency distribution, stop value is 10^{-4}

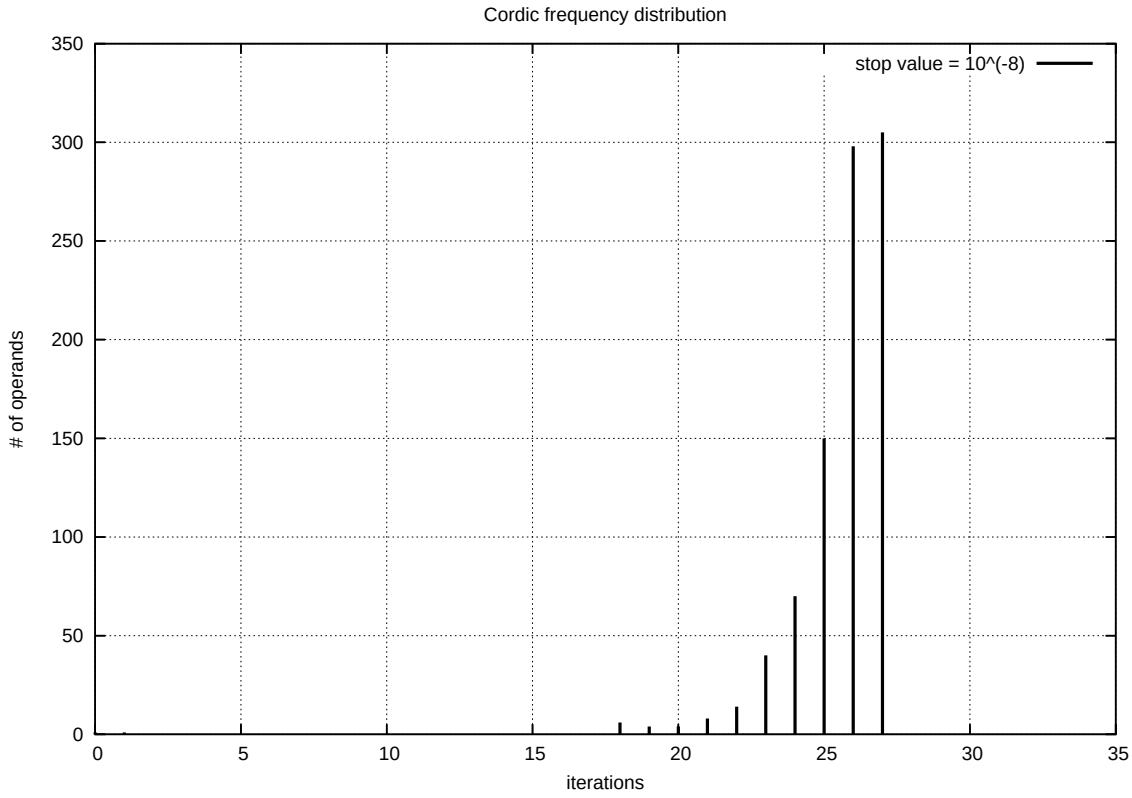


Figure 3.11: Calculated *Cordic* frequency distribution, stop value is 10^{-8}

3.2.4 Montgomery Inverse

```
1  #define even (x)    ((x&0x1) == 0)
2
3  /*
4  * calculates the modular inverse in the Montgomrey domain
5  * inputs: p is a prime number and 0 < a < p
6  * outputs:  x=(a^-1) mod p
7  */
8  double montgom_inv(long p, long a){
9      long u, v, r, s, k;
10
11     u = p; v = a; r = 0; s = 1; k = 0;
12     while(v>0){ // phase i
13         if (even(u)){u/=2; s*=2;}
14         else if(even(v)){v/=2; r*=2;}
15         else if(v>=u){v=(v-u)/2;s+=r;r*=2;}
16         else {u=(u-v)/2;r+=s;s*=2;}
17         k++;  // count the number of itterations
18     }
19     while(k>0){ // phase ii
20         if (even(r)){r/=2;}
21         else{r=(r+p)/2;}
22         k--;
23     }
24     return p-r;
25 }
```

Figure 3.12: The Montgomery inverse algorithm

The Montgomery inverse [41][46] is used to compute the inverse of an integer modulo a prime number. Calculation of modulo inverses is a common operation used within cryptographic systems. We examine the Montgomery algorithm given in figure 3.12 for non-manifest behavior. The Montgomery algorithm expects 2 input operands P,a where P must be a prime number.

The Montgomery algorithm was profiled using the following prime numbers $3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381, 32749, 65521$ as input values of the first operand P and for the second operand we provided all possible combinations up to the value of the first operand P provided. The distribution results for $P =$

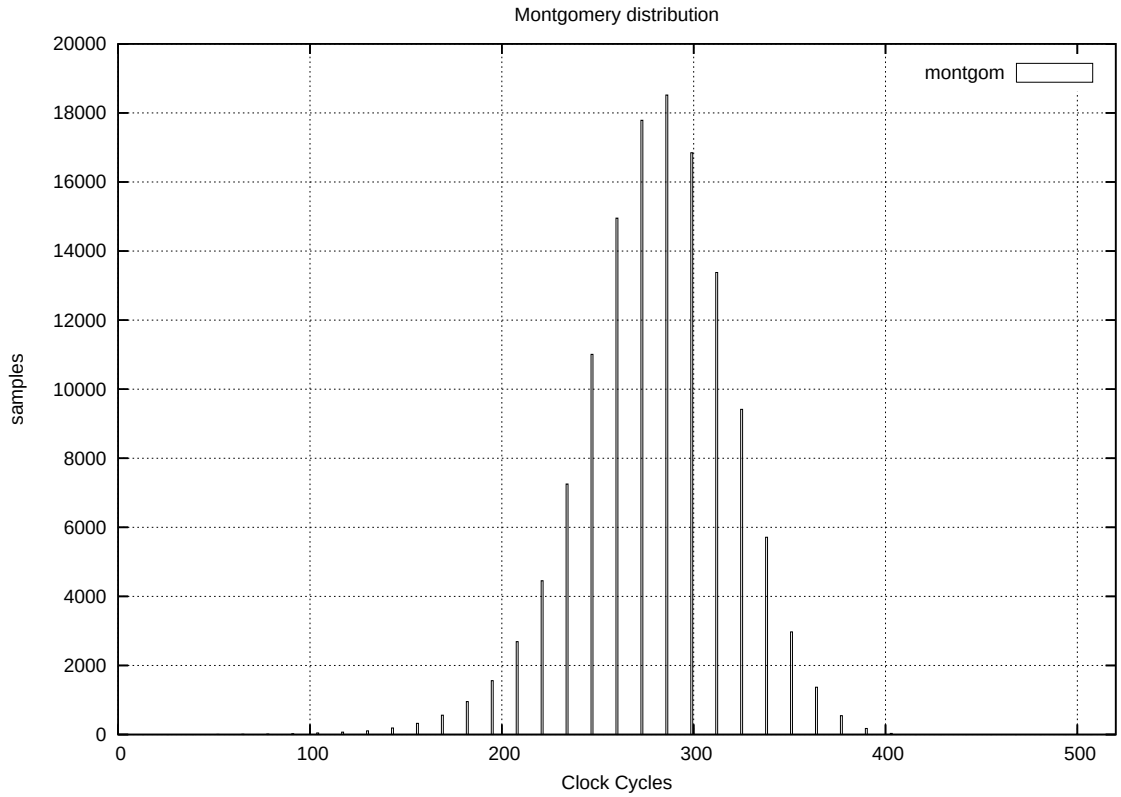


Figure 3.13: Profile results of the Montgomery inverse algorithm. The latency of the loop body is assumed to be 13 Clock Cycles. Which is an approximate value for the number of instructions of the loop body

65521 and a is all combinations from 1 up to P is given in figure 3.13 and the various distributions obtained by fixing the value of the first operand P is given in figure 3.14. Since the actual delay of the loop body is not of any further importance and can be obtained by profiling a single iteration of the algorithm on the target implementation platform. In this distribution we choose the value of $C_{montgom} = 13$ which is approximate value of the number of instructions of the loop body.

From the distribution results we notice that the algorithm has an average latency around $22 * C_{montgom}$ clock cycles and the maximum number of iterations is 31. This results in 286 as the average number of clock cycles and 403 as the maximum number of clock cycles for the Montgomery algorithm. These results are not very accurate

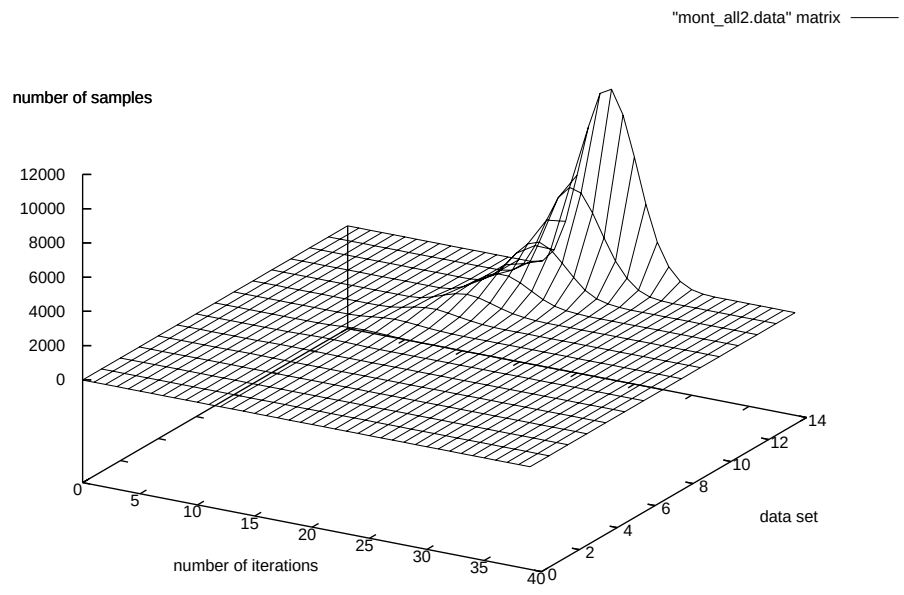


Figure 3.14: Montgomery profiling results for various prime number sets

since the loop body does not have a constant latency and is dependent on the execution path taken. In chapter 6 we use a better technique to profile this algorithm. This technique profiles the algorithm based on the actual path taken during execution, hence the obtained results are much more realistic.

3.3 Statistical properties of non-manifest algorithms

In order to show the behavior of non manifest loops within an application we examine the application given in figure 3.15. The application consists of a Montgomery - Gcd - Montgomery combination of non-manifest algorithms.
If the Montgomery - Gcd - Montgomery application is to be implemented on a statically scheduled architecture the expected latency would be the sum of the maximum

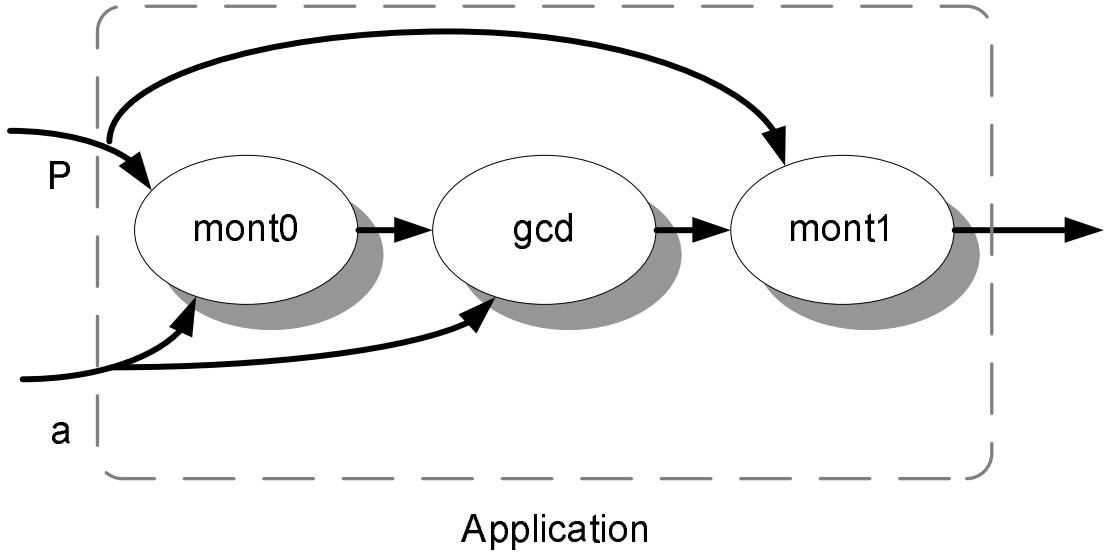


Figure 3.15: The Montgomery - Gcd - Montgomery algorithm as an example of a non-manifest loop application

latency of the individual non-manifest loops which is $2*(31*13)+(23*3) = 875$ Clock cycles. Where we can assume that the maximum number of iterations of Montgomery is 31 and the number of clock cycles of the loop body is 13, the maximum number of clock cycles of the Gcd algorithm is 23 and the number of clock cycles of the loop body is 3.

On the other hand if we would have taken the average latency value of the individual distributions as given in figure 3.3 which is 27 clock cycles and figure 3.13 which is 300 clock cycles, the expected latency would be $2*300+27 = 627$ clock cycles. From the actual distributions of the individual nodes as given in figure 3.16 we can see that the actual distributions are dependent, hence the real average value lies around 500 clock cycles as shown in figure 3.16 (d).

Note: the given distribution is based on an input value set which contains all combinations of possible 16-bit input values. In practice the input data would be a subset

of all possible combinations and hence the actual distributions may vary even further. Also for the Montgomery algorithm the loop body is control intensive and each control path consumes a different number of clock cycles, hence it is not accurate to assume that the loop body has a constant latency.

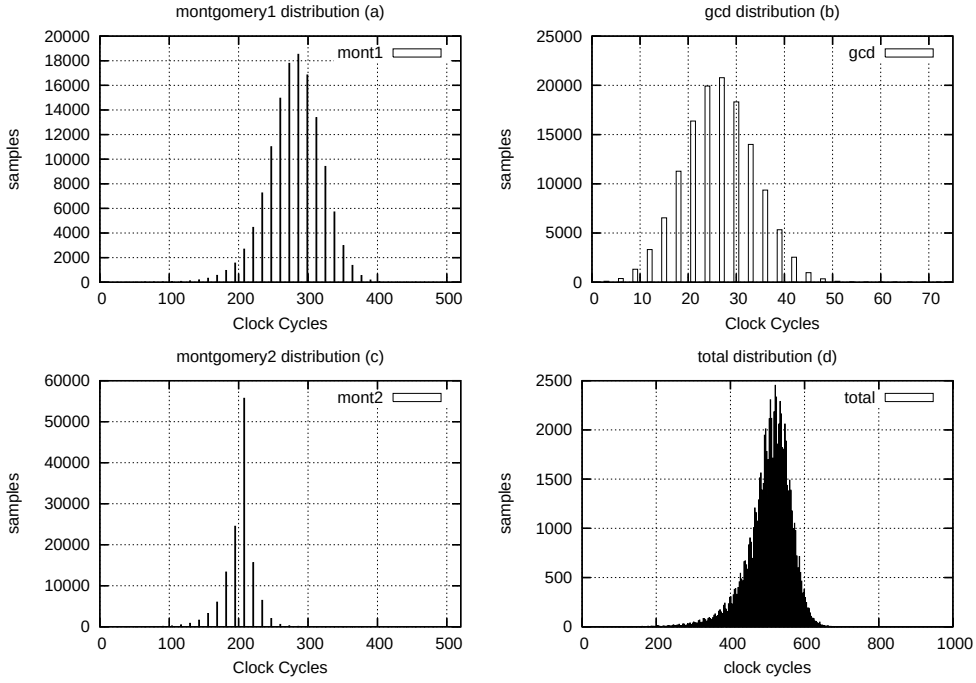


Figure 3.16: Montgomery-Gcd-Montgomery application distributions

3.4 Conclusions

Non manifest algorithms are characterized by having a variable execution latency. The variation in latency may be due to control sequences in the instruction code which will lead to various execution path's. Each execution path has its own delay and hence the actual delay of the algorithm is not known at compile time. The variation in execution latency of the non-manifest algorithm may also be due to input data

dependency. Such non-manifest loops are called non-manifest data dependent. They are characterized by having a variable number of iterations of the loop body. The actual number of iterations depends on the input data. Taking the number of loop iterations as a design parameter is not sufficient. It is not general enough to cover all application examples. A better choice for profiling is to use the actual amount of clock cycles consumed on an implementation processor this is described in chapter 5 and chapter 6.
By profiling the execution latency of such applications we were able to determine the average and maximum execution load. When an application contains multiple inter-connected loop algorithms the distributions of the individual nodes are not independent. In order to determine the average load, the application must be profiled using the exact input data set which would be provided at run-time from the real environment.
The maximum load of a non-manifest algorithm for a single input operand CL_{max} is a design parameter which will be used further on in chapter 5. This parameter is used to determine the amount of processing elements needed to order cope with a certain input stream workload. This parameter is obtained by profiling the application and noting the maximum obtained number of clock cycles for a single computation. Profiling the application with a typical input stream, that represents the stream in which the application will have to operate, will obviously provide the best results.
In some situations the input set provided to a non-manifest algorithm leads to a constant execution latency. If this occurs the designers of a system should analyze the consequences, as a dynamic scheduling scheme might not always be beneficial due to system overhead.

3.4.1 Summary

The class of non-manifest is summarized in table 3.4.1. Non manifest loops can be categorized by being either analytic or convergent.
Table 4.1 summarizes the bench mark results of the non-manifest algorithms discussed in this chapter.

iterations	loop body	algorithm	example
fixed	data dependent	Gcd	analytic
variable	fixed	Cordic rotation	convergent
variable	data dependent	Russian multiplication	analytic

Table 3.1: Summary of profiling results

name	average # cc	maximum # cc	dependency
gcd	27	69	$C_{gcd} = 3$
rmult	43	51	$C_{rmult} = 3$
cordic	175	189	$C_{cordic} = 7$, stop value $= 10^{-8}$
mont	286	403	$C_{mont} = 13$

Chapter 4

Dynamic Hardware Scheduling Architectures

In this chapter we explore various hardware dynamic scheduling architectures from literature. The strong and weak points of each architecture are identified and compared to each other.

4.1 Introduction

In digital signal processing (*DSP*) applications, many algorithms have a repetitive and periodic nature [1]: the same computations must be executed on arrival of each new data sample or block of samples. Some loops within a computation require a constant number of clock cycles in their loop-body and their number of iterations is fixed, they thus have a fixed total execution time. Such loops are called manifest-loops. Non-manifest data dependent loops, on the other hand, are those where the number of iterations are data dependent and hence have a variable total execution length. By high-level synthesis, the functions of the application are translated into hardware elements that will perform the computations. Such hardware elements are called processing elements. The functions for non-manifest algorithms can be classified as: processing elements for (1) *analytic* functions and (2) *non-analytic* or *soft* functions. Analytic functions are those that have exactly one answer and in order to calculate

that answer the processing element requires a variable number of iterations and thus clock cycles. The exact amount of clock cycles is dependent of the input data and is bound between a minimum and a maximum number of clock cycles. Non-analytic functions, on the other hand, converge to the required result in time, hence the upper bound is not fixed and is dependent on both the input data and the required quality of the computation.
Scheduling, which is the task of specifying the order and the allocation in time for each processing element, can be done either at design time (*static scheduling*) or at run time (*dynamic-scheduling*). When scheduling loops of non-analytic functions, one can statically schedule the loop and set its loop count to a fixed number of iterations. The quality of the result in the case of too few iterations would be sacrificed and in the case of too many iterations we unnecessary perform too many clock cycles. This shows that static scheduling of non-manifest non-analytic processing elements is not without costs.
Figure 4.1 shows the difference between static scheduling and dynamic scheduling for the application graph given in chapter 2 figure 2.3. When there are no processing element constraints, the static schedule will consume 29 clock cycles. The dynamic schedule on the other hand is data dependent and hence the operand data of the functions determent the latency. When there are no processing element constraints, the latency will range from 11 clock cycles which is the minimum latency of the critical path (of the *application graph*of figure 2.3) and 29 clock cycles which is the maximum latency of the critical path of the *application graph*.

4.2 Static scheduling

In static scheduling, the order in which the processing elements are to execute is determined at *compile time*. Examples of statically scheduled architectures are Phideo designed at Philips research [4], the Transport Triggered Architecture *TTA* which is an application-specific instruction-set processor (ASIP) architecture template that allows easy customization of processor designs [12][14][13], and the Trimedia processor which is a VLIW digital signal processor from Philips Semiconductors [19]. In order

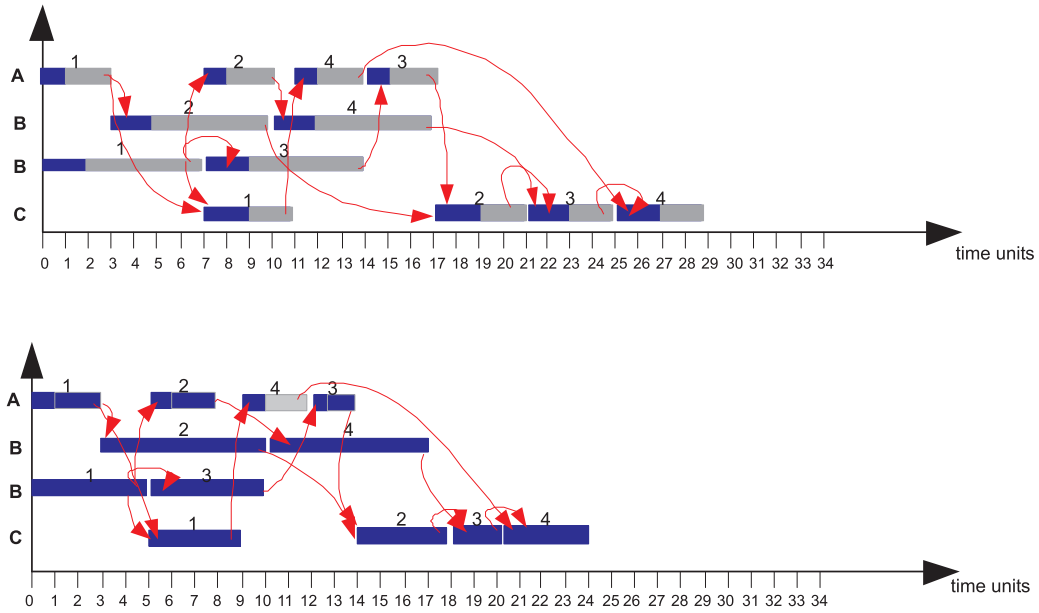


Figure 4.1: Example of a static (top) v.s. dynamic schedule (bottom). The static schedule always has the worst case latency which is 29 clock cycles in this figure. The dynamic schedule has a data dependent latency between which can range from 11 and 29 clock cycles, based on the actual operands provided

to perform this task the high-level synthesis compiler has to know the order (function dependencies) in which the processing elements are to execute and the execution time of each processing element. This information is needed in order to allocate the processing elements to time instances and hence complete the execution schedule. The way in which the processing elements are activated in time is dependent on the control mechanism used.

4.2.1 Phideo

The Phideo [4] design methodology aims at finding hardware structures for video algorithms with minimum integrated circuit area requirements given the video algorithm and its timing constrains. The algorithm is represented by a signal flow graph, which consists of nodes representing operations and arcs representing data dependencies.

The hardware structure consists of processing units (PUs) on which the operations are executed, memories to store the intermediate data, address generators to supply the memories with addresses for reading and writing the intermediate data, interconnections consisting of wires and multiplexers for the transportation of data, and finally a controller to give the correct control signals. Figure 4.2 gives a schematic view of the architecture. The hardware generated by Phideo is synchronous, there is a central clock which rules the synchronization between different blocks. The clock frequency is usually a multiple of the sampling frequency.
The task of the scheduler within Phideo is to determine for each operation within the abstracted signal flow graph (the algorithm specification) a clock cycle on which it has to be started and to assign those operations to processing units. For each of the operations it is specified on which type of processing unit it has be executed and for each processing unit the time shape (which indicates liveness of the processing unit in time) and area costs are specified. Furthermore, timing bounds are given, e.g. on input and output operations. The goal of scheduling is to minimize the total area that is required. Because of the application domain, not only processing units but also memories have a significant contribution to the total area.
Phideo solves the scheduling problem, which is shown to be *NP* hard, by a primal all-integer Integer Linear Program (ILP) [32] algorithm.
The control mechanism of Phideo consists of a number of cascaded counters which are synchronized by a central clock. Each counter triggers an event that will activate one or more operations, transport the data to and from memories and trigger the address generators. To function properly the exact time needed to perform an operation has to be known in advance by the scheduler. Therefore, the Phideo scheduler cannot be used for non-manifest or data dependent algorithms, as their execution latencies are not known in advance.

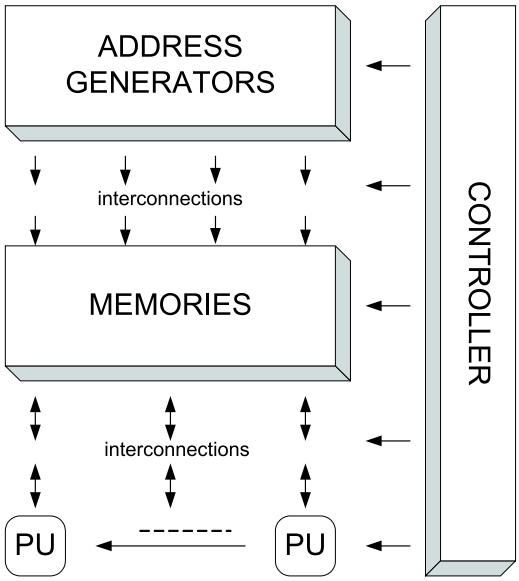


Figure 4.2: The target architecture of Phideo

4.2.2 Pipelining

The basic pipelining mechanism is a static scheduling technique whereby multiple instructions are overlapped in execution. Pipelining takes advantage of the parallelism that exist among the actions needed to execute an instruction. It is nowadays one of the key implementation technique used to improve the performance of CPUs. Pipelining splits the actions needed for instruction execution into a number of phases. The actual number of phases depends on the independent steps needed to execute an instruction. A typical pipelined processor would split up the execution of an instruction into a number of phases[1]:

- **IF:** Instruction Fetch, the actual fetching of the instruction from memory

[1]The pipeline phases described belong to the MIPS processor

- **ID:** Instruction Decode, decoding the instruction and determination of the operation to be performed
- **EX:** Execution, the actual execution of the operation
- **MEM:** Memory Access, memory accesses that might be needed
- **WB:** Write Back, writing back the result of the operation either to memory or registers

Figure 4.3 shows a typical execution behavior of an ideal pipelined architecture without any conflicts. Notice the overlap in phases (pipeline execution stages) of various instructions.

	Clock								
Instruction number	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Instruction i	IF	ID	EX	MEM	WB				
Instruction i+1		IF	ID	EX	MEM	WB			
Instruction i+2			IF	ID	EX	MEM	WB		
Instruction i+3				IF	ID	EX	MEM	WB	
Instruction i+4					IF	ID	EX	MEM	WB

Figure 4.3: A typical pipelined execution behavior

In an ideal situation (a situation with no hazards and were all pipeline stages are of equal length), the speedup from pipelining would be equal to the number of pipeline stages. Superpipelined architectures [27] try to improve the speedup by maximizing the number of pipeline stages, this is mainly due to the fact that pipelined architectures are only capable of achieving a peak throughput of one instruction per clock cycle in the ideal situation. Thus the only way to improve the performance in this case is to increase the clock frequency. A *Superpipelined* CPU in general would have a very high clock frequency in the range of GHz. In such an organization the handling of hazards effectively is very critical. Since any stalling or re-initialization of the pipeline will degrade the performance of the CPU significantly. Moreover, as more pipeline stages are added, and hence further dividing up the combinational logic,

the propagation delay times of the flip-flops begins to dominate. The improvement achieved is even less, when hazards are taken into account, the performance may actually become worse. This is especially the case in control intensive programs which are dominated by branches and jumps.

The various types of hazards that can occur are summarized as follows [29][30]:

Control Hazards: hazards that occur due to branches or jumps and hence the pipeline has to be flushed.

RAW data Hazards: Read after Write, an instruction tries to read a source before a prior instruction has written that source, so it incorrectly gets an older value.

WAR Hazards: Write after Read, an instruction tries to write a destination before it is read by a prior instruction, so the prior instruction incorrectly gets the new value.

WAW Hazards: Write after Write, an instruction tries to write a destination before it is written by a prior instruction. The writes are performed in the wrong order, and hence leaving the wrong value of the prior instruction in the destination.

Structural Hazards: When a processor is pipelined, the overlapped execution of instructions requires concurrent access to some resources (e.g. register files). If some combination of instructions can not be accommodated because of resource conflicts, the processor is said to have a structural hazard.

In statically scheduled pipelined architectures, the compiler tries to avoid pipeline hazards by inserting no-operation (*nop*) instruction in the instruction code of the algorithm. Unfortunately not all hazards can be detected at compile time. Control hazards that lead to severe pipeline stalls are very difficult to detect as they involve prediction of whether branch instructions will be taken or not. Dynamically scheduled pipeline architectures try to solve such hazards at run time by hardware means. This will be fully discussed in the next section.

Finally we summarize by mentioning that:
(1) In an ideal situation speedup is w for a w deep pipeline
(2) Hazards reduce the speedup and
(3) not all pipeline hazards can be detected at compile time

4.3 Dynamic scheduling in hardware

In dynamic scheduling, the order in which processing elements are to be executed is determined at run time by the (hardware) scheduler or controller. We consider only architectures that have multiple FU's that can run in parallel. Dynamic schedulers were introduced to speed up program execution by solving dependencies at run time instead of at compile time. Dynamic hardware schedulers are wide spread nowadays [18] and most of them are used within *superscalar* architectures like the Intel Pentium. In the next sections we discuss dynamic hardware scheduling schemes based on a dataflow model of execution (such as the dynamic dataflow machine) and their counter part the control flow model of execution (such as the Scoreboard [30] and the Tomasulo [22][30] scheduler). But before we delve into the details of the hardware dynamic schedulers it is important to mention that dynamic schedulers try to solve dependency conflicts or hazards at run time. In general purpose processors the program to be executed is provided as a sequence of instructions see for example figure 4.8. Modern processors, aim at parallelizing the instruction execution as much as possible in order to improve performance. The amount of instruction level parallelism (ILP) available within an instruction sequence varies from one algorithm to another. Various techniques and architectures have been developed in order to improve the performance or program speedup. Among those techniques is pipelining which is described in section 4.2.2.

An alternative to superpipelining is the *superscalar* architecture organization. A superscalar architecture tries to achieve a high ILP by using multiple FUs that can run in parallel and at the same time make use of pipelining. A typical superscalar machine would execute multiple instructions per clock cycle with the instruction issue controlled dynamically by hardware. The data dependencies, and hence hazard detection, are controlled by the hardware scheduler. Typically dependencies in the instructions stream are checked for during the issue stage. In the next sections we review hardware scheduling schemes. The dataflow execution model [33] [35] is self scheduling and has the potential of exploiting true parallelism inherent within the application, the Tomasulo [22][30] scheduler and the Scoreboard [30] are both scheduling

schemes which aim at improving ILP while maintaining data dependence and avoiding the hazards that can stall the pipeline.

4.3.1 Data flow architectures

The Data flow machines have been researched actively in the 80th [38][39][35]. Data flow architectures make use of dynamic scheduling. Its dataflow model of execution offers attractive properties for parallel processing. It is asynchronous, because it bases function or instruction execution on the availability of the operands. Synchronization of parallel activities is implicit and it is self scheduling. There is no sequencing order for instructions except for those imposed by true data dependencies of the dataflow program graph. Another attractive property of the dataflow model is that scheduling of non-manifest loops is implicitly solved by the execution model. Since functions will only execute when operands are available, theoretically when there are no resource constraints, there will be no wasted clock cycles.

In the abstract dataflow execution model, the order of instruction execution is determined by the availability of its operands rather than by a program counter. Data values or instruction operands are carried by tokens. Those tokens travel along the edges connecting various nodes in the program graph. Each node represents an instruction. The edges connecting the nodes are assumed to be FIFO queues of unlimited length. Since direct implementation of this model is impossible [35], the data flow execution model has been classified as being either static or dynamic.

Static dataflow execution model: the static dataflow execution model allows at most one instance of a node to be enabled for firing. The node can only execute when all its tokens are available on it's input edges and there are no output tokens on its output edges.

Dynamic dataflow execution model: the dynamic dataflow execution model permits activation of several instances (e.g. due to several iterations) of the same node at the same time during runtime. To be able to distinguish between different instances of a node, each token is tagged. The tag identifies the context in which each particular token was generated. Similar to the static dataflow execution model, a node is

considered executable when all of its input tokens, with identical tags, are available.

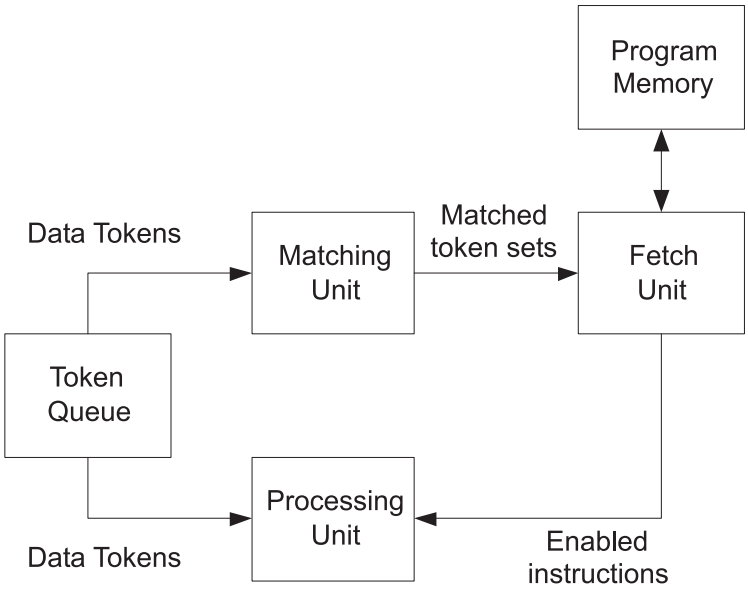


Figure 4.4: The general organization of the dynamic dataflow model

The basic structure of a dynamic dataflow model is given in figure 4.4. Tokens are received by the token matching unit, which is a memory containing a pool of waiting tokens. The units basic operation, is matching tokens with identical tags. If a match exists, the corresponding token is extracted from the matching unit, and the matched token is passed to the fetch unit. If no match is found, the token remains in the matching unit waiting for its matching partner. In the fetch unit, the tags of the token-pair uniquely identify an instruction to be fetched from memory (operations of the dataflow are in this case are either monadic or dyadic, hence at maximum two operand tokens are required to perform a computation). The instruction and the matched token pair form the enabled instruction, which is sent to the processing unit. The processing unit executes the enabled instructions and produces the result tokens to be sent to the matching unit via the token queue.

Despite the dynamic dataflow model's potential for large-scale parallel execution, a number of weak point's have been identified [35]:

- **Overhead:** an efficient implementation of the matching unit is needed to avoid overhead due to the token matching scheme, as it has been shown that the performance depends directly on the rate at which the matching mechanism processes tokens [37]. An associative memory would be an ideal solution. Unfortunately this is not a cost effective solution since the amount of memory needed to store tokens waiting for a match tends to be very large.

- **Instruction cycle:** when compared to the control part of Von Neuman architectures that use a program counter, the instruction cycle of the dataflow model is quite extensive: it involves
  (1) detecting enabled nodes,
  (2) determining the operation to be performed,
  (3) computing the results, and
  (4) generating and communicating the result tokens to appropriate target nodes.
  This leads to slow single thread performance or will degrade the performance of applications with a low degree of parallelism.

- **Data Structures and arrays:** handling of data structures and arrays implies generating an excessive amount of large tokens which will degrade the performance. Basically the data travels with the token and hence arrays would lead to an excessive amount of large tokens.

4.3.2 Scoreboard scheduler

Scoreboarding is a hardware dynamic scheduling technique for allowing instructions to execute out-of-order when there are sufficient processing elements and no data dependencies. It was named after the CDC 6600 which was the first machine to use a scoreboard (see figure 4.5). In addition to scoreboarding, the CDC 6600 was the first processor to make extensive use of multiple processing elements which can run in parallel.

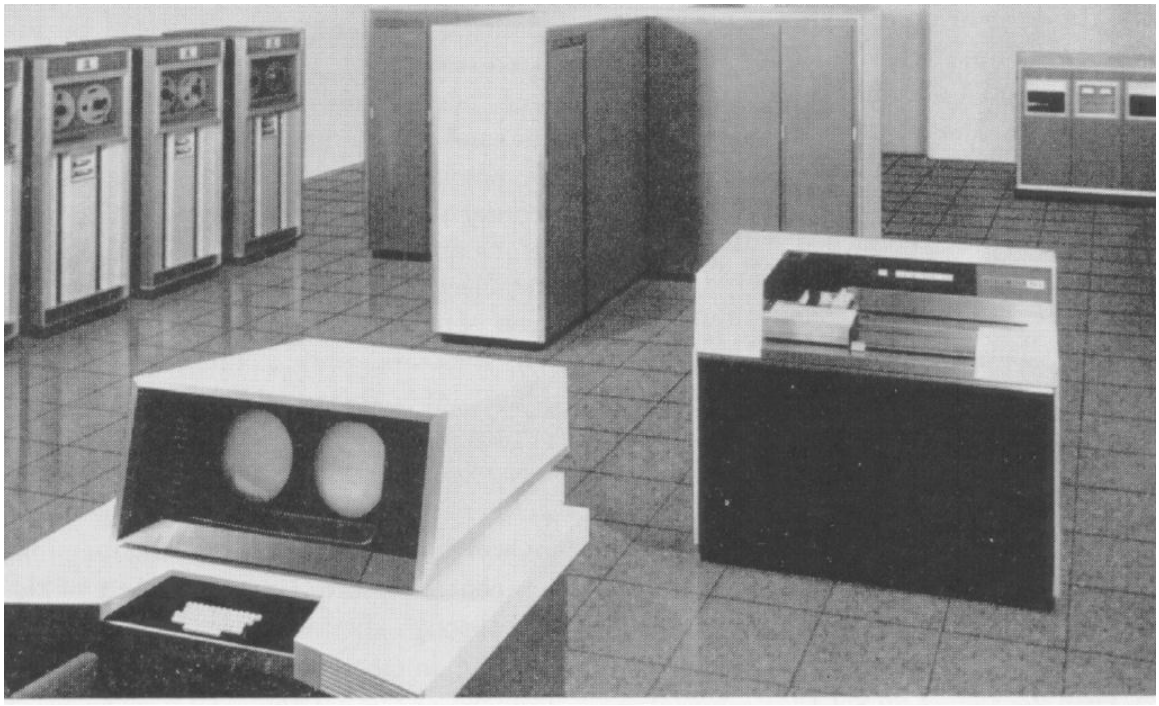


Figure 4.5: The CDC 6600 processor courtesy of Control Data Corporation laboratory

In a dynamically scheduled pipelined architecture, all instructions pass through the issue stage in-order (in-order issue); however, they can be stalled or bypass each other in the second stage (read operands) and thus enter execution out-of-order. Scoreboarding [30] is a technique for allowing instructions to execute out-of-order when there are sufficient processing elements and no data-dependencies.

Operation: Every instruction goes through the scoreboard, were the data dependencies are recorded. The scoreboard determines when the instruction can read its operands and begin execution. If the scoreboard decides that the instruction can not execute immediately, it monitors every change in the hardware and decides when the instruction can execute (the exact book keeping steps taken by the scoreboard controller are depicted in appendix A.1). The scoreboard also controls when an instruction can write its result to the destination register. Hence all hazard detection and resolution is centralized in the scoreboard.

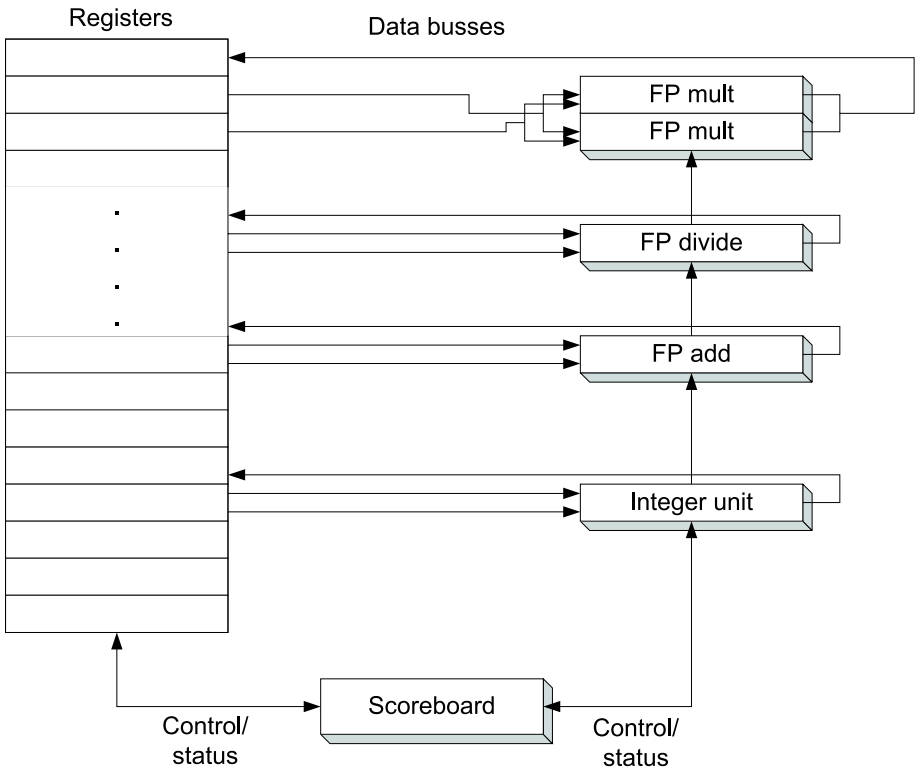


Figure 4.6: Data path of a MIPS processor with a scoreboard scheduling mechanism, the processor contains 5 Functional Units and a register bank. There are separate busses for data communication between the processing elements and the registers. The scoreboard controls the datapath

The scoreboard instruction life cycle consists of 4 steps *Issue, Read Operands, Execution, and Write Result.*

- **Issue:** in the issue stage an instruction is allocated (issued) to a processing element if the processing element is free and there is no other active [2] instruction

[2]An active instruction is an issued instruction that has not reached its write result phase

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result

Register Status	
Register	FU
F0	
F1	
F2	
F3	
F4	
F5	
F6	
F7	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk

Figure 4.7: Scoreboard: the scoreboard consists of a number of fields which are used to keep trace of data dependencies and insure that no hazards can occur due to dependence violations

having the same destination register, hence Write after Write (WAW) hazards, which occurs when an instruction tries to write an operand before it is written by a prior instruction thus leaving the value of the prior instruction in the destination register, are avoided in this way [30]. Upon each issue the scoreboard will update its internal structure. If a structural WAW hazard exists, the instruction issue will stall until the WAW hazard is resolved (e.g. until the previous write instruction is finished).

- **Read Operands:** in the read operands stage the scoreboard continuously monitors the availability of the source operands. The scoreboard recognizes that a source operand of an instruction is available if there exists no other issued active instruction that is going to write that operand. On the availability of the operands the scoreboard tells the processing element to proceed reading the operands from the registers and start the execution. By waiting for the availability of the operands the scoreboard resolves Read after Write (RAW) hazards dynamically. RAW hazards occur when an instruction tries to read a source before it is written by a prior instruction, so it incorrectly gets an older value.

- **Execution:** the execution stage is initiated upon the arrival of the operand of a processing element. Once the computation is complete and the result is available, the processing element will notify the scoreboard that the execution is complete.

- **Write result:** Once the scoreboard has been notified by a processing element that it has completed its execution the scoreboard will check for Write After Read (WAR) hazards. WAR hazards is the situation that occurs when an instruction tries to write a destination before it is read by a prior instruction, so the prior instruction incorrectly gets the newly written wrong value. If WAR hazards are detected the scoreboard will; stall the completing instruction.

The scoreboard holds the status of instructions, registers and functional units (FU's):

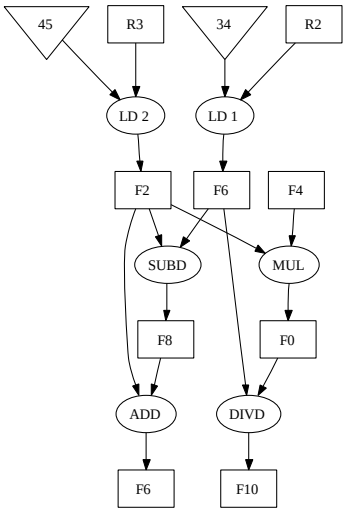
- **1. Instruction status:** the instruction status basically indicates in which part of the instruction life-cycle it is in (see 4.7).

- **2. Functional unit status:** this basically indicates the state of the processing element. There are nine fields for a processing element.
  (1) Busy: indicates whether a unit is busy or not
  (2) Op: The operation to be performed e.g. Mult, Div, Add etc.
  (3) Fi: Destination register of the processing element
  (4) Fj, Fk: Source registers of the processing elements
  (5) Qj, Qk: The Functional units which will produce the data of the source registers Fj, Fk
  (6) Rj,Rk: Flags indicating the status of Fj, Fk e.g "YES" means the input operand is ready, "NO" is not yet written. The flags will be set to "NO" after the operands are read.

- **3. Register result status:** Indicates which processing element will write a register, if an active instruction has the register as destination. The field is set to blank if there are no pending instructions that will write the register.

Based on its own data structure, the scoreboard controls the instruction progression from one step to the next by communicating with the functional units. To avoid structural hazards due to the limited number communication busses, the scoreboard must ensure that the number of functional units within the *Read Operands, Execution* and *Write result* state do not exceed the number of busses available.

Example: scoreboard




```
LOAD    F6  (34+)R2   // F6
<- mem[R2+34]
LOAD    F2  (45+)R3   // F2
<- mem[R3+45]
MULT    F0   F2  F4   // F0  <- F2 * F4
SUBD    F8   F6  F2   // F8  <- F6 * F2
DIVD    F10  F0  F6   // F10 <- F0 / F6
ADDD    F6   F8  F2   // F6  <- F8 * F2
```

Figure 4.8: An example program with its accompanying data dependency graph. In the data dependency graph ellipses indicate *operations*, rectangles indicate *memory* or *registers*, and triangles indicate *constants*

We will demonstrate how scoreboarding works by means of an example which was originally given in [30]. For the example we use the program given in figure 4.8 and its accompanying data dependency graph. In this example we will assume that an addition operation takes 2 clock cycles, multiplication and division both take 4 cycles. We go through the execution of the example given in figure 4.8 step by step using the scoreboard table given in figure 4.7. Figure 4.9 up to 4.13 give the state of the scoreboard for each clock cycle.

CLOCK 1

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1			
LOAD F2 45+ R3				
MULT F0 F2 F4				
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	
F2	
F4	
F6	INTEGER
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F6		R2				YES
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

CLOCK 2

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2		
LOAD F2 45+ R3				
MULT F0 F2 F4				
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	
F2	
F4	
F6	INTEGER
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F6		R2				YES
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

CLOCK 3

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	
LOAD F2 45+ R3				
MULT F0 F2 F4				
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	
F2	
F4	
F6	INTEGER
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F6		R2				YES
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

CLOCK 4

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3				
MULT F0 F2 F4				
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	
F2	
F4	
F6	INTEGER
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F6		R2				YES
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

Figure 4.9: Scoreboard clock 1 to 4

CLOCK 5

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5			
MULT F0 F2 F4				
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	
F2	INTEGER
F4	
F6	
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F2		R3				YES
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

CLOCK 6

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6		
MULT F0 F2 F4	6			
SUBD F8 F6 F2				
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	INTEGER
F4	
F6	
F8	
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F2		R3				YES
MULT1	YES	MULT	F0	F2	F4	INTEGER		NO	YES
MULT 2	NO								
ADD	NO								
DIVIDE	NO								

CLOCK 7

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	
MULT F0 F2 F4	6			
SUBD F8 F6 F2	7			
DIVD F10 F0 F6				
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	INTEGER
F4	
F6	
F8	ADD
F10	
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F2		R3				YES
MULT1	YES	MULT	F0	F2	F4	INTEGER		NO	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2		INTEGER	YES	NO
DIVIDE	NO								

CLOCK 8

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6			
SUBD F8 F6 F2	7			
DIVD F10 F0 F6	8			
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	INTEGER
F4	
F6	
F8	ADD
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	YES	LOAD	F2		R3				YES
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

Figure 4.10: Scoreboard clock 5 to 8

CLOCK 9

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	CLOCK 4	
SUBD F8 F6 F2	7	9	CLOCK 2	
DIVD F10 F0 F6	8			
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	
F8	ADD
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

CLOCK 10

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	CLOCK 3	
SUBD F8 F6 F2	7	9	CLOCK 1	
DIVD F10 F0 F6	8			
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	
F8	ADD
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

CLOCK 11

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	CLOCK 2	
SUBD F8 F6 F2	7	9	11	
DIVD F10 F0 F6	8			
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	
F8	ADD
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

CLOCK 12

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	CLOCK 1	
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8			
ADDD F6 F8 F2				

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	
F8	ADD
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	SUBD	F8	F6	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

Figure 4.11: Scoreboard clock 9 to 12

CLOCK 13

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8			
ADDD F6 F8 F2	13			

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	ADD
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	ADD	F6	F8	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

CLOCK 14

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8			
ADDD F6 F8 F2	13	14	CLOCK 2	

Register Status	
Register	FU
F0	MULT1
F2	
F4	
F6	ADD
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	YES	MULT	F0	F2	F4			YES	YES
MULT 2	NO								
ADD	YES	ADD	F6	F8	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6	MULT1		NO	YES

CLOCK 15

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	CLOCK 4	
ADDD F6 F8 F2	13	14	CLOCK 1	

Register Status	
Register	FU
F0	
F2	
F4	
F6	ADD
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	YES	ADD	F6	F8	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

CLOCK 16

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	CLOCK 3	
ADDD F6 F8 F2	13	14	16	

Register Status	
Register	FU
F0	
F2	
F4	
F6	ADD
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	YES	ADD	F6	F8	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

Figure 4.12: Scoreboard clock 13 to 16

CLOCK 17

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	CLOCK 2	
ADDD F6 F8 F2	13	14	16	17

Register Status	
Register	FU
F0	
F2	
F4	
F6	ADD
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	YES	ADD	F6	F8	F2			YES	YES
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

CLOCK 18

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	CLOCK 1	
ADDD F6 F8 F2	13	14	16	17

Register Status	
Register	FU
F0	
F2	
F4	
F6	
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

CLOCK 19

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	19	
ADDD F6 F8 F2	13	14	16	17

Register Status	
Register	FU
F0	
F2	
F4	
F6	
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

CLOCK 20

Instruction Status				
Instruction	Issue	Read Operands	Execution complete	Write Result
LOAD F6 34+ R2	1	2	3	4
LOAD F2 45+ R3	5	6	7	8
MULT F0 F2 F4	6	9	13	14
SUBD F8 F6 F2	7	9	11	12
DIVD F10 F0 F6	8	15	19	20
ADDD F6 F8 F2	13	14	16	17

Register Status	
Register	FU
F0	
F2	
F4	
F6	
F8	
F10	DIVIDE
F12	
F14	

Functional Unit Status									
Name	Busy	Operation	Fi (dest)	Fj (source)	Fk (source)	Qj	Qk	Rj	Rk
INTEGER	NO								
MULT1	NO								
MULT 2	NO								
ADD	NO								
DIVIDE	YES	DIV	F10	F0	F6			YES	YES

Figure 4.13: Scoreboard clock 17 to 20

In clock 1 the *LOAD F6 (34+)R2* instructions is issued. We can see in the *instruction status* field (see figure 4.9) of the scoreboard that the instructions was issued, in the Functional Unit status we see that the state of the *INTEGER FU* is set to Busy (Note: the *INTEGER FU* handles memory operations in this architecture example); that the operation to be performed is a *LOAD* operation; the destination register of the result is *F6* and that the data of the source register is available hence the *YES* tag in the *Rk* field. Finally, the register status of register *F6* indicates that it will be written by the *INTEGER FU*. In clock 2 the operands (i.e. the contents of the *R2* register) are read, reading the operands of an instruction takes 1 clock cycle. The execution phase is in clock 3. In clock 4 the write results phase takes place, and the content of the register *F6* is updated with the result of the operation. This also takes one clock cycle hence the *INTEGER FU* is free in clock 5. The second load instruction in the instruction stream was not issued by the scoreboard because the *INTEGER FU* was occupied by the first *LOAD* instruction. The *MULT F0 F2 F4* instruction was also not issued by the scoreboard because one of its source operands *F2* was not yet written. Hence the instruction was stalled until the operand is available. (Note: that exchanging the first two *LOAD* instructions would have improved the program). This is however, not possible in the scoreboarding approach and hence a badly written program will result in bad execution performance.

In clock cycle 6 the read operands of the second *LOAD* instruction takes place together with the issuing of the *MULT* instruction. In the processing element status field of the scoreboard we see that the *MULT1 FU* is occupied with a *MULT* operation and that the second source operand *F2* is not yet available, hence the "NO" in the *Rj* field. Since not all operands are available, the instruction can not enter the *read operands* phase. The scoreboard will keep on issuing other instructions if their *FU* and the destination register of the instruction are available.

The reader should note that the scoreboard scheduler avoids data hazards by either stalling the instruction after the issue stage or stalling the write result stage. If a structural hazard exists which is the situation that will occur when an instruction requires a *FU* that is still occupied with another instruction; the second instruction requiring the *FU* will not be issued.

A scoreboard scheduler uses the amount of ILP to minimize the number of stalls arising from the program's true data dependencies. By eliminating those stalls, the scoreboard is limited by the following factors:

- The amount of parallelism available within the program; This determines the number independent instructions that can be found to execute. If each instruction depends on its predecessor, no dynamic scheduling scheme can reduce the stalls.

- The set of instructions examined as candidates for potential execution is called a window. The number of instructions in the window; determine how far ahead in the pipeline the scheduler can look for independent instructions. It is assumed that the size of the window (and hence the scoreboard) does not extend beyond a branch. Hence the window (and the scoreboard) always contain straight line code from a single basic block.

- The number and types of functional units; this determines the probability of structural hazards, which can increase when dynamic scheduling is used.

- The presence of anti-dependencies and output dependencies which lead to WAR and RAW hazards

Discussion

The scoreboard will allow instructions to execute in parallel when there are no data dependencies amongst them and there are no structural hazards. In other words sufficient busses for the data communication and sufficient FUs are available. Since each instruction statically allocates the output register and there is only one physical register for each output, and the scoreboard stalls instructions if their destination registers are occupied. The output registers become a bottleneck and hence the amount of ILP available within program loops is limited.
This problem was solved in the Tomasulo scheduler which is described in the next section. The Tomasulo scheduler uses a register renaming scheme and the concept

of reservations stations. Reservation stations behave as if they are virtual result registers and together with the register renaming scheme there would be more virtual result registers than that is actually used and hence less register conflicts can exist. The register renaming scheme insures that the scheduler can dynamically allocate the result registers instead of the static allocation used within the scoreboard scheduler, hence structural hazards due to register conflicts are avoided. This is the key concept behind the Tomasulo scheduler.

The reader should also note that scoreboarding preserves the instruction order. The first two load instructions are independent. As can be seen in the data dependency graph shown in figure 4.8. Had we exchanged the execution order of the instructions and let the second load instruction *LOAD F2, (45+)R3* execute first, the third *Mult* instruction would not have been stalled during the issue and read operands phase. The scoreboard does not make use of the concept of result-forwarding, which is the situation of forwarding the results as soon as they are available and hence they can be out-of-order, it preserves program dependence by stalling a writing instruction if the destination is still in use. Hence the results within the scoreboard scheduling scheme are always in order. Finally we sum up by mentioning that the scoreboard limits parallelism to basic blocks of code or single line code, it uses the amount of ILP available within the code to minimize its pipeline stalls and a badly written piece of code will result in bad performance.

4.3.3 Tomasulo scheduler

One of the most popular algorithms for *out-of-order* execution is the Tomasulo scheduling algorithm. The Tomasulo algorithm was first published in 1967 by R.M.Tomasulo [22] [21]. It is one of the most competitive scheduling algorithms and is currently used in the Intel Pentium P3, P4 and the IA-32 architectures [42].

As an example we use the basic structure of a MIPS floating point unit which uses a Tomasulo scheduler (see figure 4.16). The structure contains three Functional units; the memory unit, FP adders and FP multipliers. The FP adders and FP multipliers are connected to reservation stations which supply the operands and to the Common

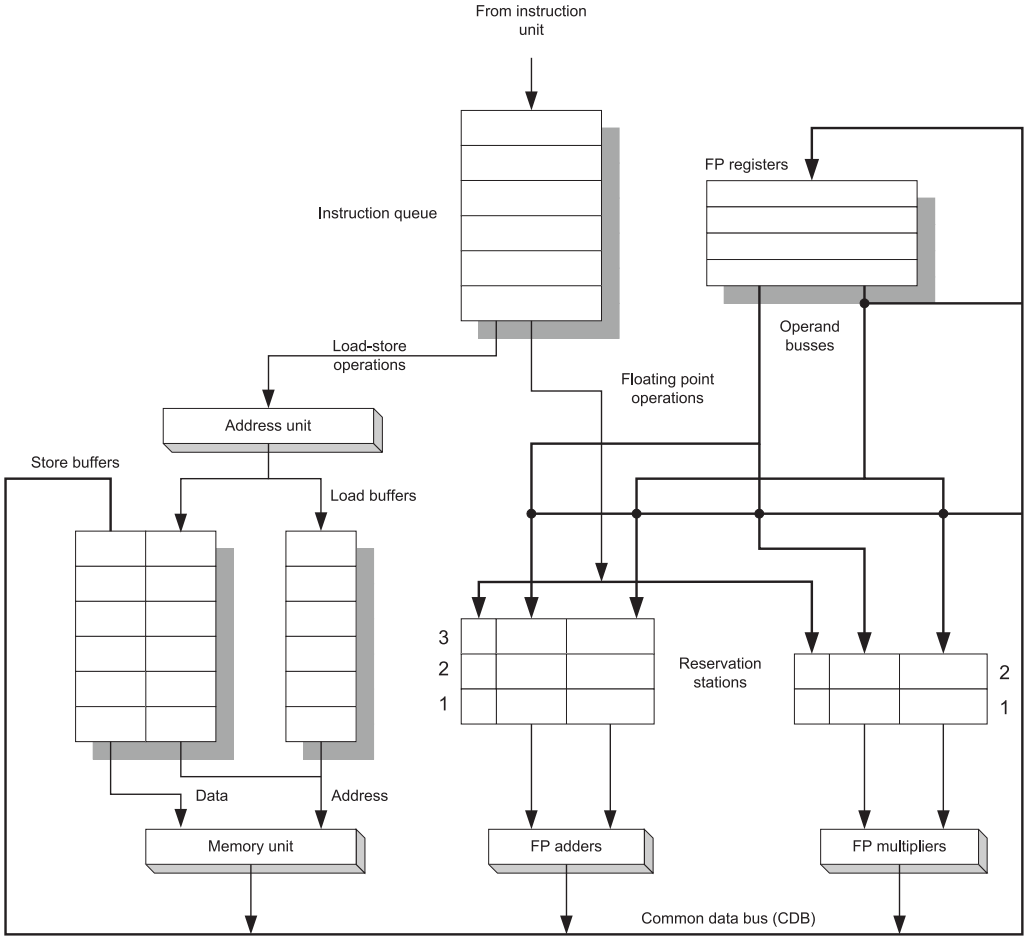


Figure 4.14: Data path of a MIPS floating-point unit with a Tomasulo Scheduler

Data Bus (CDB) for writing their computation results. The memory unit is connected to load and store buffers for loading and storing data from memory. The address unit is responsible for calculating the memory addresses needed by the memory unit. The FP registers hold the registers used within floating point instructions and the instruction queue holds the instructions issued by the instruction unit. The instruction queue issues the instructions in fifo order, hence instructions within Tomasulo are issued in-order. Unlike the scoreboard, instructions go through three phases, Issue,

Execute, and Write result. There is no read operands phase, since the Tomasulo scheduler issues the instructions to reservation stations if the reservation station or load store buffers are free[3], depending on whether the instruction was a floating point instruction or load store instruction. The reservation stations include the operation to be performed as well as the information used for resolving the hazards. The load buffers have three functions: hold the components of the effective address until it is computed, track outstanding loads that are waiting on memory, and hold the results of completed loads that are waiting for the CDB. Store buffers also have three functions: hold the components of the effective address until it is computed, hold the destination memory addresses of outstanding stores that are waiting for the data value to store, and hold the address of the value to store until the memory unit is available. All the results from either the FP units or the load unit are published on the CDB bus, which goes to the FP registers as well as to the reservation stations and store buffers. The FP adders and multipliers perform the floating point addition/-subtraction and multiplication/division. The exact steps an instruction goes through is as follows:

- **Issue:** Get the next instruction from the head of the instruction queue. Instructions are issued in fifo order and hence correct dataflow is maintained. If there is a free matching reservation station the instruction is issued to it with the operand values if they are available in the registers. The operand values are thus copied to the reservation stations. If there is no free reservation station, then there is a structural hazard and the instruction is stalled until a buffer is free or the reservation station is freed. If the operands are not in registers then keep track of the processing elements that will produce them. This step renames registers, eliminating WAW and WAR hazards.

- **Execute:** If one or more operands are not available monitor the Common Data Bus while waiting for it to be computed. When an operand is available it will be

[3]Unlike the scoreboard which will only issue an instruction if the FU is available and the result destination register is not in use.

placed in the corresponding reservation station. When all operands are available the operation can be executed at the corresponding processing element. By delaying the instruction execution until all operands are available, RAW hazards are avoided. Notice that several instructions could become ready in the same clock cycle for the same processing element. Although independent processing elements could begin execution in the same clock cycle, if more than one instruction is ready for the same processing element, the unit will have to choose among them. For the floating point reservation stations this choice may be made arbitrarily. Loads and stores, however, present additional complications since they require a two step execution phase. First the effective address has to be calculated when the base address is available, then the calculated effective address will be placed in the load or store buffer. Loads in the load buffer execute as soon as the memory unit is available. Stores in the store buffer wait for the value to be stored before being sent to the memory unit. Loads and stores are maintained in program-order through the effective address calculation. This will prevent hazards through memory.

- **Write result:** When the result is a available, write it on the Central Data Bus (CDB) and from there into the registers and into any reservation stations (including store buffers) waiting for the result. Stores also write data to memory during this step: when both the address and data value are available, they are sent to the memory unit and the store operation completes.

Register renaming [40][30] In the Tomasulo scheduling scheme, register renaming is provided by the reservation stations. The reservation stations buffer the operands of instructions waiting to be issued. The idea is that a reservation station fetches and buffers the operands as soon as they are available and hence eliminating the need to get the operand from a register. Also, pending instructions designate the reservation stations that will provide their input operands. Finally, when successive writes to a register overlap in execution only the last one is actually used to update the register. As instructions are issued, the register specifiers for pending operands are *renamed* to the names of the reservation station holding those operands. Since there can be

more reservation stations than real registers, this technique can eliminate hazards arising from name dependencies that could have not been eliminated statically by a compiler.

Instruction Status

Instruction	Issue	Execute	Write Result
LOAD F6 34+ R2	1	1	1
LOAD F2 45+ R3	1	1	
MULT F0 F2 F4	1		
SUBD F8 F6 F2	1		
DIVD F10 F0 F6	1		
AddD F6 F8 F2	1		

Register Status

Field	Qi
F0	Mult1
F2	Load2
F4	
F6	Add2
F8	Add1
F10	Mult2
F12	
F14	

Functional Unit Status

Name	Busy	Operation	Vj	Vk	Qj	Qk	A
Load1	NO						
Load2	YES	LOAD				R3	45+Regs[R3]
Add1	YES	SUB		Mem[34+Regs[R2]]	Load2		
Add2	YES	ADD			Add1	Load2	
Add3	NO						
Mult1	YES	MUL		Regs[F4]		Load2	
Mult 2	YES	DIV		Mem[34+Regs[R2]]			

Figure 4.15: The state of the reservation stations when all instructions are issued and the first instruction has written its result to the CDB

For the Tomasulo scheduler we use the same terminology as in the scoreboard scheduling scheme. Each reservation station has seven fields (see figure 4.15):

- **Op:** The operation to perform on source operands S1 and S2.
- **Qj, Qk:** The reservation stations that will produce the corresponding source operand; a value of zero indicates that the source operand is already available in Vj or Vk, or is unnecessary.
- **Vj, Vk:** The value of the source operands. Note that only one of the V fields or the Q field is valid for each operand. For loads, the Vk field is used to hold the offset field

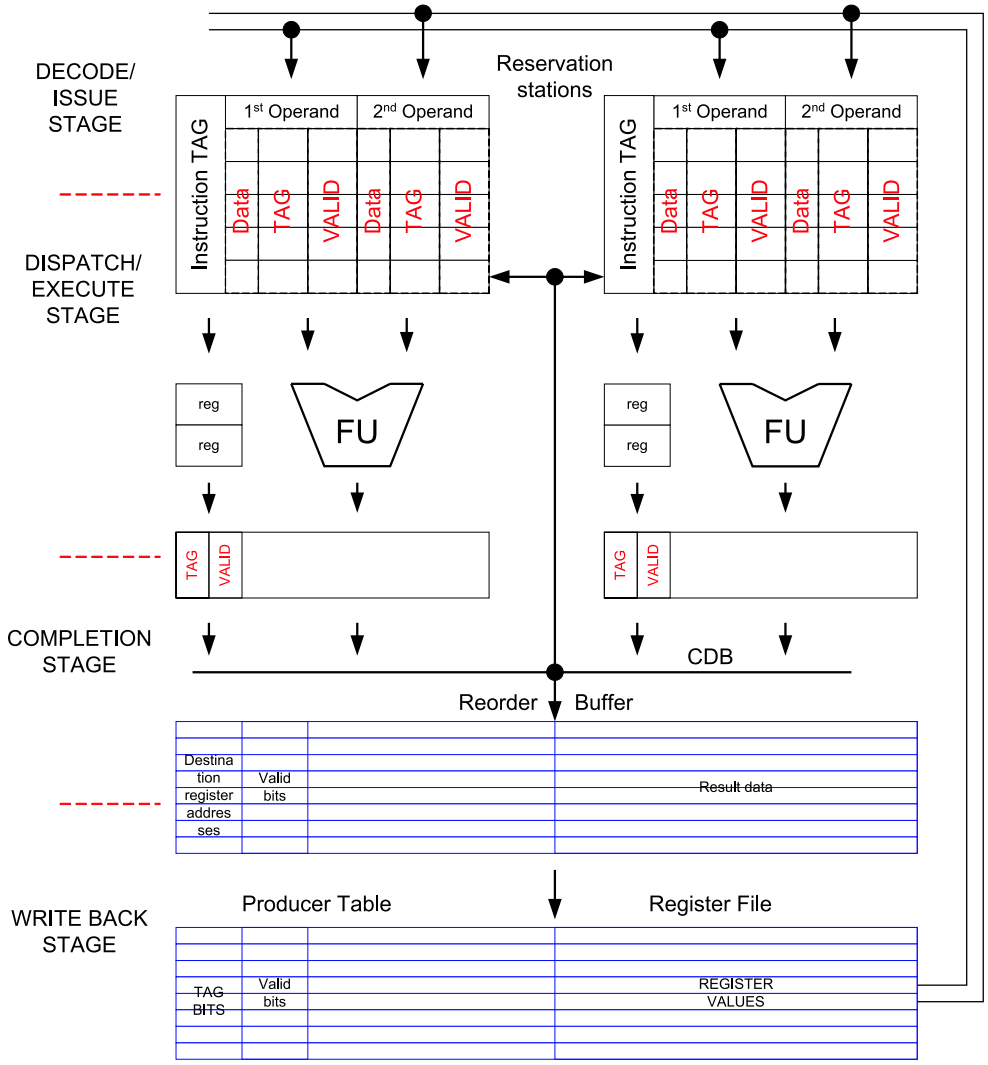


Figure 4.16: Structure of a processor core with a Tomasulo Scheduler in combination with a reorder-buffer

- **A:** Used to hold information for the memory address calculation for load or store. Initially, the immediate field of the instruction is stored here; after the address calculation, the effective address is stored here.

- **Busy:** Indicates that this reservation station and its accompanying processing element are in use.

The register file has the field, **Qi**:

- **Qi:** The indication of the reservation station that contains the operation whose result should be stored into this register. If the value of Qi is blank (or 0), no currently active instruction is computing a result destined for this register, meaning that the value is simply the register contents.

Appendix B provides the bookkeeping rules for the Tomasulo scheduling algorithm. When an instruction is issued, the destination register has its Qi field set to the number of the buffer or reservation station to which the instruction is issued. If the operands are available in the registers, they are stored in the V fields. Otherwise, the Q fields are set to indicate the reservation that will produce the values needed as source operands. The instruction waits at the reservation station until both its operands are available, which is indicated by zero in the Q field. The Q fields are set to zero either when this instruction is issued, or when an instruction on which this instruction depends completes and does its write back. When an instruction has finished execution and the CDB is free, it can commence its write back. All the buffers, and reservation stations whose value of Qj or Qk is the same as the completing reservation station update their values from the CDB and mark the Q fields to indicate that values have been received. Hence, the CDB can publish its results to multiple reservation stations in a single clock cycle. When the waiting reservation stations have their operands, they can all begin execution on the next clock cycle. Note that after the issue stage instructions can enter execution out-of-order. Figure 4.16, is a modified [21] Tomasulo architecture which contains a reorder buffer. By allowing the results to be written out-of-order (out-of-order dispatch) in the Write Results phase to a special reorder buffer, this then reorders the results. Results in the reorder buffer can be forwarded, it can speed up instructions that are waiting on those results. The net effect is that a higher ILP is obtained. According to [18] this scheme is used in the UltraSparc processor series and it is used to compensate for slow FU's. Nevertheless

there are some constraints, namely that at most one instruction is completed per clock cycle and that RAW dependencies are obeyed. Thus instruction dispatch must still stall if the operands are not available or if the instructions compete for the CDB.
In the Tomasulo approach RAW hazards are avoided by executing an instruction only when its operands are available. WAW and WAR hazards, which arise from name dependencies, are eliminated by register renaming. Register renaming eliminates these hazards by renaming all destination registers, including registers with a pending read or write of an earlier instruction, so that out-of-order write does not affect any instructions that depend on an earlier value of an operand.
If we think of registers as names instead of locations, then we can write data to a free register and store the name within a look up table. The contents of the register can then be read using the name as an index within the lookup table. In this way we basically allocate registers to names dynamically at run time and hence avoid stalling instructions due to register conflicts.

Discusion

We have seen that *out-of-order* execution, *out-of-order* completion and register renaming are among the modifications added to the Tomasulo scheduler in order to improve ILP. Even with these sophisticated scheduling mechanisms, we have to look at other concepts such as task level parallelism or coarse grained parallelism, if more parallelism is to be obtained from an algorithmic specification. Since the instruction stream itself is the bottle neck.
A comparison between the dynamic scheduling architectures and models can be summarized as follows. Both Tomasulo and scoreboard schedulers are used in superscaler architectures which are considered to be Von Neumann type architectures. Von Neumann type architectures suffer from the so called Von Neumann bottleneck. Their architectures are controlled by a sequencer which executes the instruction stream. Instructions are fine grained and the program counter points to the next instruction to be executed. The instruction life cycle consists of fetching instructions, executing them and writing the results back to either registers or memory. This is very short

when compared to the instruction cycle of dataflow machines. In order to speedup Von Neumann type of architectures the execution of instructions are pipelined as mentioned in [4.2.2] [4.3.2]and [4.3.3]. Theoretically if there are no hazards the speedup up for a w deep pipelined architecture is w. Unfortunately hazards do occur and they reduce the speedup. Some of the pipeline hazards can be solved (statically) at compile time by aligning, reordering and inserting dummy *nop* instructions. Unfortunately this does not solve all control hazards. The scoreboard and Tomasulo schedulers try to solve those hazards dynamically such as mentioned in 4.3.2 and 4.3.3. The differences between the scorebaording and Tomasulo is that scoreboarding will allow instructions to run in parallel if there are no dependencies amongst them and there are no structural hazards. Tomasulo on the other hand, tries to achieve more ILP by distinguishing between true instruction dependencies and dependencies due to compile time register allocation. The solution to register resource constraints is register renaming and the solution to processing resource constraints is reservation stations. Both concepts are discussed in section 4.3.3. The data flow model of execution does not suffer from false dependencies. Since there are no registers. The operands of an instruction (node) travel through the edges of the data flow graph from node to node. An instruction is considered executable when the processing element implementing the instruction and operands of the instruction are available. Hence the data flow model is hindered only by the true dependencies available within the application. There is no restriction on the instruction granularity within the data flow model although lessons learned from previous research suggest that instructions with fine granularity lead to more overhead and an imbalance in the computation to communication ratio.

Comparison

Despite the problems mentioned, the dataflow model of execution has attractive properties for high-throughput streaming applications and hence is a motivation for the High^2 dataflow architecture discussed in the upcoming chapters. We will show that problems regarding the nature of the pure dataflow model of execution (such as mentioned in section 4.3.1) can be circumvented by an adaptation of the model.

Table 4.1: Comparison

name	Weak points	Strong points
Scoreboard	- Uses ILP to resolve hazards - Instructions can only run in parallel if there are no dependencies amongst them and there are no structural hazards - Only single line of code or basic blocks - Von Neumann bottleneck - Static register allocation implies that bad register allocation leads to bad execution performance	- Short instruction life cycle - Dynamically scheduled pipeline
Tomasulo	- Von Neumann bottleneck	- Dynamically scheduled pipeline - Uses register renaming to resolve register name conflicts
Dataflow	- Overhead due to matching unit - Efficient implementation of the matching unit requires complex multi-ported registerfile which is expensive - Bad single thread performance - Long and inefficient instruction cycle - For fine grained instructions the number of tokens waiting for a match can be very large, which leads to large memories and long latencies - Due to the lack of locality of reference it is difficult to predict when a matching token will arrive - Handling of data structures is not trivial and can lead to deadlocks [35]	- Asynchronous execution of instructions. Instructions are active as soon as their operands are available. - True instruction dependencies - Capable of exploiting true parallelism if available within the program - Capable of scheduling *non-manifest-loops* without loss of clock cycles

Chapter 5

High^2 Data Flow Machine

This chapter discusses data flow machines as a possible solution for applications with non-manifest behavior such as those described in chapter 3. The design of the High^2 data flow machine (High^2 *DFM*), which is an application specific dedicated processor capable of utilizing the parallelism inherent within an application, is provided. The High^2 *DFM* was designed with high-throughput streaming applications in mind, this means that throughput is of major importance. We show that the *DFM* can be tuned at design time to meet certain throughput requirements, this is achieved by varying the number of processing elements available for computing the (non-manifest) functions of the application. In the High^2 *DFM* design we focus on providing solutions for the average case stream processing load and at the same time maintain a high-throughput.

5.1 Introduction

Since in algorithms with non-manifest loops the latency depends on the input data, an architecture which utilizes a static scheduling scheme will have a number of idle processing cycles. These idle cycles are mainly due to the fact that the scheduler plans for the worst case processing latency since it has no prior knowledge of the input data pattern. These idle cycles could be used to enhance the performance of

the architecture in many ways. For example: the processor can disable it's processing elements during idle cycles and hence save energy or free up the processing elements during the idle cycles and make them available for new computations, this effectively increases the processing capacity of the system. By increasing the processing capacity, the system can accept a data stream with a higher throughput or higher processing load. This chapter examines the possibilities of utilizing those idle processing cycles available within non-manifest loops. The solution provided is based on dynamically scheduling the processing elements at run time and freeing those elements as soon as the computation has completed. In section 5.2 a design feasibility experiment, for the simple case of a single non-manifest node, is provided and in section 5.3 we explore the general case for an application with multiple manifest and non-manifest nodes.

5.2 The Simple model

We have seen in chapter 4 section 4.3.2 that the scoreboard scheduling mechanism issues instructions in-order and that instructions enter the execution phase out-of-order if there are no dependencies, hence a better ILP can be obtained depending on how efficient the program was written. This was improved upon in the Tomasulo scheduling mechanism described in section 4.3.3. Instructions are still issued in-order but unlike the scoreboard, instructions can also enter the execution phase out-of-order even in the presence of dependencies. The WAR and RAW hazards that can occur, were solved in the Tomasulo scheduler by introducing the concept of reservation stations and register renaming. The Tomasulo scheduler is more flexible than the scoreboard and achieves better ILP at the cost of more control hardware overhead. In the experiment described in this section, dynamic scheduling on coarse-grained non-manifest data dependent functions is used in combination with out-of order execution and a reorder buffer. Unlike the Tomasulo or Scoreboard (which try to parallelize the instructions, determine their operand-data flow at run time, and solve the hazards) in our experiment we only focus on maximizing the parallelism of *independent* computations on a data stream and at the same time determine the amount of processing elements needed in order to meet a certain stream processing-load.

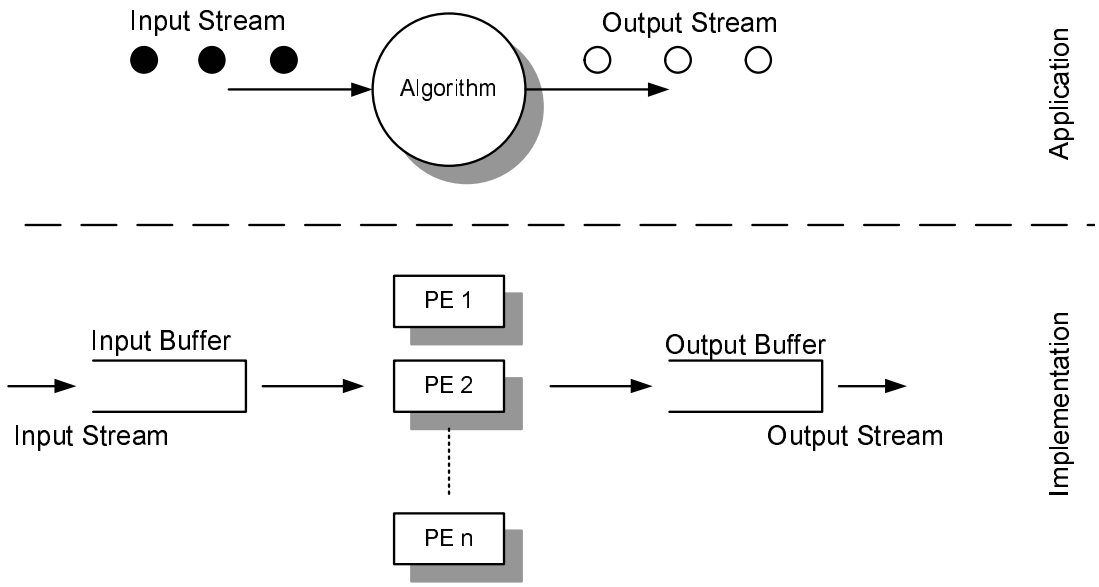


Figure 5.1: Simple model of an application with non-manifest loops and its proposed hardware architecture

The simple-application model and the proposed hardware implementation model are shown in figure 5.1. In the simple-application model the application is one single non-manifest algorithm which has to operate on a high-throughput stream of independent operands. Since the application has to process an input stream that has a time delay between two operands that is shorter than the computation of one single data sample, having only one processing element in the hardware implementation is not enough (since the input workload is larger than the computational capacity). In order to solve this problem we propose a hardware architecture with multiple processing elements. The processing elements will run in parallel, operating on independent stream operands. In order to improve the performance of the hardware architecture even further we propose an out-of-order execution mechanism. This mechanism will ensure that the processing elements are freed-up as soon as possible and hence are ready for a new computation. The proposed architectural model (see figure 5.1) for the application consists the following elements:

- **Input Buffer:** the input buffer will store the incoming stream samples when they cannot be dispatched to a processing element. Stream samples will be stored in the input buffer if the processing elements within the architecture are occupied with an earlier computation.

- **Processing Elements:** the processing elements are dedicated hardware implementations of the non-manifest algorithm. All processing elements, within this architecture, have the same implementation. Since the algorithm is non-manifest, the latencies of individual computations may be different, and hence computations can over take each other in execution thus providing an out-of-order output stream.

- **Output Buffer:** if the application dictates that input stream and the output stream are synchronous, the architecture must somehow reorder the output stream produced by the processing elements. This is implemented by a reorder buffer.

- **Controller:** the controller of the architecture basically dispatches the input stream samples to the processing elements at run-time, hence forming a dynamic scheduler. It basically controls the flow of the stream and performs the needed run-time control calculations which involve buffer locations and activation of the processing elements.

In order to implement such an architecture we need to calculate the input buffer-size, number of processing elements, output buffer-size, implement the scheduling mechanism within the controller and determine the system latency. In the next sections, the process of designing such a system is described.

5.2.1 Design

Take the *Gcd* described in chapter 3 as an example(see figure 5.2). The *Gcd* is a typical example of an analytical non-manifest loop. The function is analytic as it only has one correct answer and is also non-manifest because the number of loop

clock cycles is dependant on the values of x, and y, which are not known at compile time. From chapter 3 we know that the algorithm can consume between $C_{res} = 3$ and $C_{res} * 23 = 69$ clock cycles depending on the input operands.

```
1  int gcd(int x, int y){
2      int g;
3
4      g = y;
5      while ( x > 0 ){
6          g = x;
7          x = y % x;
8          y = g;
9      }
10     return (g);
11 }
```

Figure 5.2: gcd algorithm

The *Gcd* function has to operate in a manifest environment on a continues stream of 16 bit input values $(x_j, y_j), (x_{j+1}, y_{j+1}), \ldots$ each execution $Gcd\ (x_j, y_j)$ would require a computational load between C_{res} and $23 \times C_{res}$. In which C_{res} is the number of clock cycles required for one iteration of the *Gcd* algorithm, and is estimated, at design time, for the target implementation ($C_{res} = 3$ in this case). In case of static scheduling or starting from the worst case computation load, the number of processing elements must be based on a maximum load per input operand of $23 \times C_{res}$ clock cycles. In practice the average load per operand will be much lower, so a large number of execution cycles are wasted. Assume, the workload generated by the input operands does not exceed a certain value B over a sliding window $i, i+1, \ldots, i+m-1$ of m operands. We wish to know:

- The number of processing elements needed
- The maximum latency

In the following sections we show that such a system is feasible using dynamic scheduling. But first we give the exact problem formulation.

Problem formulation

Given an input data stream with known maximum workload bound B on a stream window of size m, hence $WL(t,m) \leq B$ for all t with $WL(t,m) = \sum_{j=i}^{i+m-1} CL_A(v_j)$ see definition 19 on page 20. And given a data dependent non-manifest algorithm A with known computational workload bounds CL_{min_A} and CL_{max_A} (See chapters 2, 3). Devise a real time hardware scheduler that will meet the workload $WL(t,m)$ of the system, produce an output that is synchronous with the input with a latency of at most Lat_{max} time units, and determine the number of processing elements N_{res} needed. We may expect that given B there exist a minimal value for Lat_{max}, moreover a larger number of processing elements could lead to a lower value of Lat_{max}.

Proposed Solution

Processor Architecture: Since the exact amount of computing cycles needed, $CL_A(v_j)$ which is bounded by $0 \leq CL_{min_A} \leq CL_A(v_j) \leq CL_{max_A}$, is not known in advance but is known after the computation of the value v_j has finished, we use the following strategy: Allocate the earliest non calculated value v_j to the first free processing element. If no processing elements are free the value v_j will have to wait in a input buffer until a processing element becomes free. The input buffer handles the input values in a FIFO manner and hence there is no starvation possible. This is similar to the in-order issue phase of the scoreboard, the only difference is that we do not issue instructions but we issue the operands of the functions to the processing elements. To avoid waiting and latency build-up, the design must ensure that there are sufficient processing elements to handle the work load. An architectural solution for this strategy is given in figure 5.3.

The system works as follows: input operands are allocated to the first free processing element (all processing elements implement the same functionality). If non of the processing elements $PE_1, \ldots, PE_n$ is free, the operand will be placed in a data-queue and its accompanying time of arrival is placed in the time-queue. The time-of arrival is needed later in the address selection and calculation unit to calculate the output operand position within the reorder buffer. Since execution of a computation has a

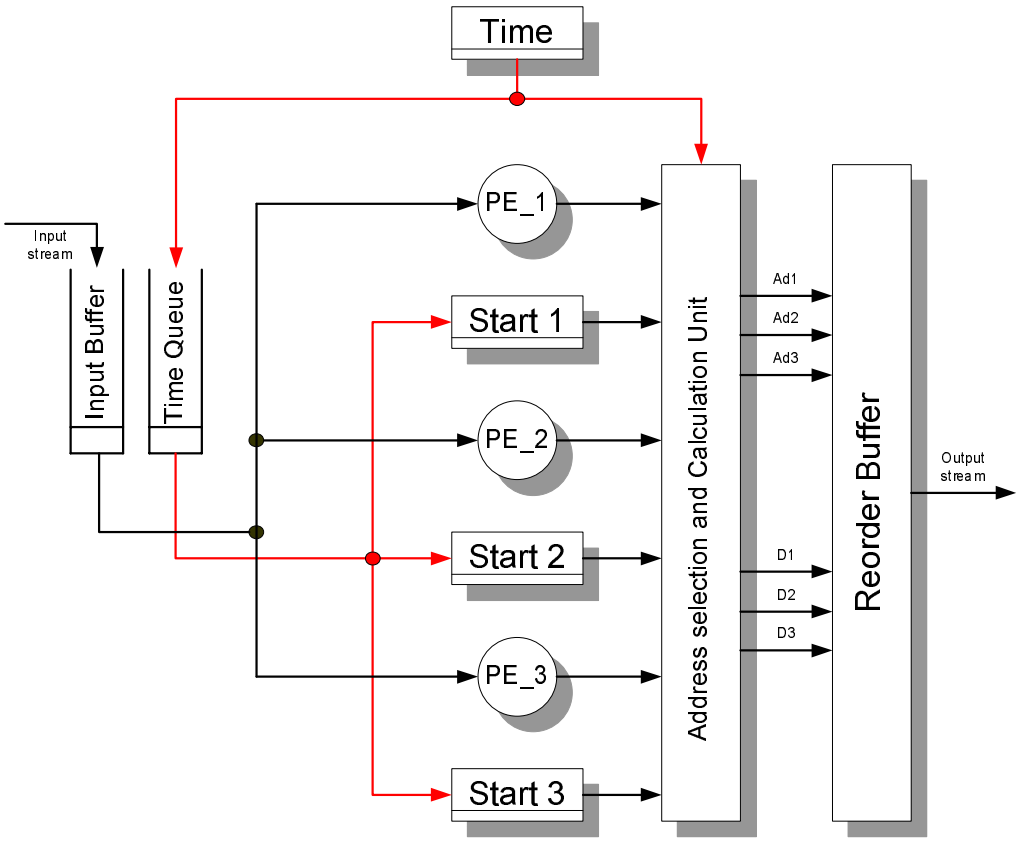


Figure 5.3: The hardware model of the scheduler, the controller and all control signals are omitted from the figure for simplicity reasons

variable length within the interval $CL_{min}, \ldots, CL_{max}$, some computations might produce their output at an earlier stage than their predecessor computations. When this occurs, the produced output stream is out-of-order. In order to ensure a synchronous system, the latency of each single computation must be the same. Note: computations are independent. The system ensures that all computations have the same latency by delaying the output result. The actual delay is the difference between latency and the actual time needed for the computation ($delay_j = Lat - CL(v_j) \quad \forall j$) and $Lat \geq CL_{max}$. This is implemented in the system by means of a reorder buffer and the address selection unit. The reorder buffer has a length Lat which is equal to the latency of the system. The address selection unit computes the position of the

output operand within the reorder buffer and writes it to that address. The address selection unit knows the $CL_(v_j)$ by subtracting the finish time of a computation from its start time which was stored in the system at the arrival of the operands. The newly produced result is placed in the reorder buffer at this address. Finally the reorder buffer positions are shifted, one place on each clock, and the value of reorder buffer at position zero will become the output of the system, which forms the output stream.
Where Lat is the latency of the system. $T_{current}$ is the time an output is produced by a processing element and T_{start} is the time of arrival of the input operand. Because the processing elements are working in parallel, multiple outputs can be produced in the same clock-cycle. All these outputs have to be written to the reorder buffer. Once all outputs have been written to the reorder buffer, all positions of the buffer will shift one place, $Address[lat-1] \rightarrow Address[lat-2] \ldots \rightarrow Address[-1] \rightarrow Address[0]$, hence the data coming out of the least significant address $Address[0]$ is the reordered output data stream.

Design Flow: In order to design such a system we need the following parameters:

1) The latency of the system Lat

2) Number of processing elements N_{res}

3) Buffer sizes

It turned out not to be possible to determine all the design parameters analytically. Only a bound of the latency could be devised. In order to obtain all the design parameters, a special cycle count simulator was developed. This simulator requires as input the workload bound B of the input stream, the streaming window size m, the number of processing elements N_{res} of the architecture, the maximum workload generated by a single input operand CL_{max} and provides the latency Lat and the buffer sizes of the system.
The latency results of the simulator are then verified against the analytical latency solution of theorem 5.2.6.

5.2.2 Design Parameters

We simplify the model used in figure 5.1 as follows (see figure 5.4):

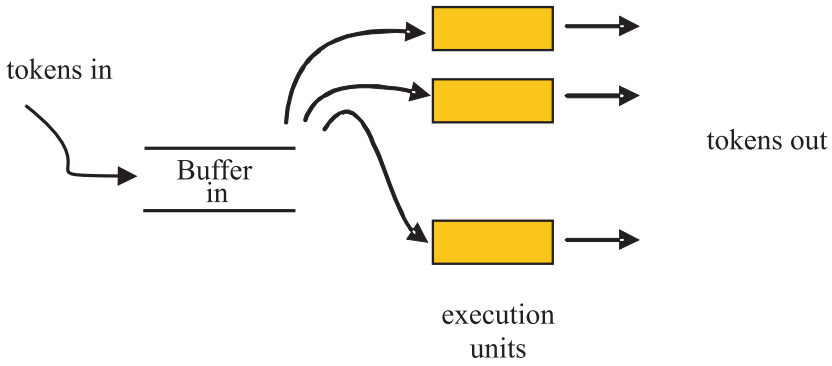


Figure 5.4: The flow of data

The input operands (tokens) v_t are identified by their arrival time t. Hence, each operand v_t is uniquely identified by t and represents a workload $CL(t)$ cycles. Note: $CL(v_j)$, $CL(t)$ are different functions which both produce the workload. For execution N_{res} processing elements are available. Each processing element uses one cycle per time unit. An operand t will enter the system at time t and either go into the buffer or to a processing element. If it was stored in the buffer it will eventually go to a processing element. The operands start execution in the order in which they arrive.

The workload entering the system in an interval of length m between t and $t+m-1$ (bounds included) is defined by:

$$WL(t,m) = \sum_{i=t}^{t+m-1} CL(i) \tag{5.2.1}$$

Let an operand t be released from the buffer at t_b, then its first execution cycle is at t_b and the operand is not in the buffer at t_b. The period the operand resides in the buffer is expressed by $del(t)$ and is $t_b - t$. So the first execution cycle of operand t is at $t + del(t)$. The buffered workload $BWL(t)$ stored in the buffer at time t is the

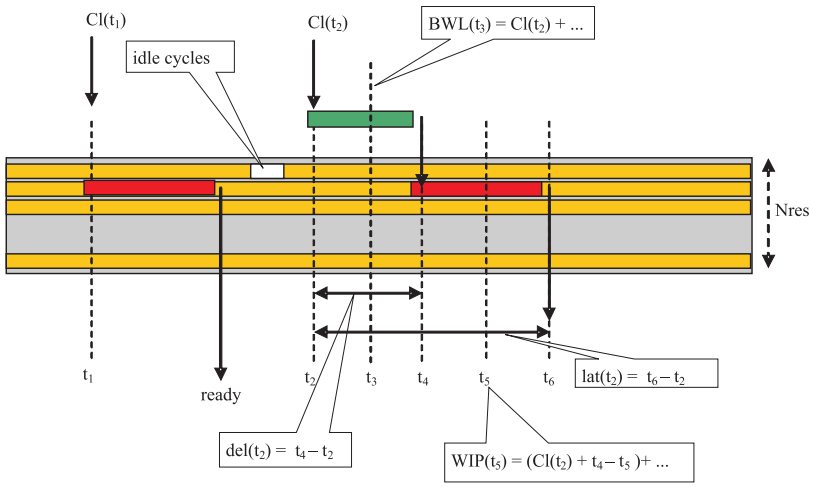


Figure 5.5: The definitions

sum of the workloads associated to the operands stored in the buffer, i.e.

$$BWL(t) = \sum_{i \leq t < t + del(i)} CL(i) \tag{5.2.2}$$

The workload in progress $WIP(t)$ at time t is the sum of the cycles that still have to be processed at time t of the operands that are being processed (by the processing elements) at time t. Hence $WIP(t)$ is the unprocessed workload within the processing elements at time t. So the contribution of an operand t_1 that starts executing at t_2, to the workload in progress $WIP(t_2)$ is $CL(t_1)$ and its contribution at t_2+1 is $CL(t_1)-1$.

The latency $lat(t)$ of an operand t is the time the operand resides in the buffer plus the time that was needed for execution. Hence

$$lat(t) = del(t) + CL(t) \tag{5.2.3}$$

System parameters

We consider a system in which the workload offered to the system for any interval m is at most B, so $WL(t, m) \leq B$. Moreover we only consider systems that are feasible, i.e. systems which can be built from a finite number of components and have a finite

latency, provided the preliminary constraints are satisfied. Clearly, a system to which more workload can be offered than can be processed is not feasible. Processing can be postponed, but not until infinity. Therefore:

Theorem 5.2.1 (Feasible systems)**.** *A system in which $B > m.Nres$ is not feasible.*

Proof. Divide time into blocks of m. For each block i, the workload $WL(i.m, m) = B$. The workload that cannot be processed is stored in a buffer WB. The stored workload at t is $WB(t)$. So the stored workload at $t = i.m$ is $WB(i.m)$. Then the stored workload at $t = (i + i).m$ is $WB(i.m) + B - N_{res}.m$. And thus because $B > m.N_{res}$ holds for all i: $WB((i+1).m) > WB(i.m)$. Hence the system with these parameters is not feasible. □

So a requirement for a feasible system is that $B \leq m.N_{res}$.
Moreover we assume $CL(t) \leq CL_{max}$, which implies that CL_{max} must be chosen $\leq B$.

Theorem 5.2.2 (Maximum number of Processing elements)**.** *The maximum number of processing elements N_{res} needed regardless B is CL_{max}. If $N_{res} \geq CL_{max}$, $Lat_{max} = CL_{max}$*

Proof. If $N_{res} = CL_{max}$, the system can be scheduled statically and a simple cyclic schedule of the successive input values over the N_{res} processing elements suffices. Viz., $IN(i)$ is processed by processing element $k = i \mod N_{res}$. Because $CL(i) \leq CL_{max}$, at $t = i + CL_{max}$ the processing element k will have been released. Processing element k will receive the next sample at $t = i + N_{res}$ which is, because $CL_{max} \leq N_{res}$, after processing element k has been released from its previous task. □

From the preceding cyclic schedule, it immediately follows that for each input sample a processing element is available immediately. Hence the maximum latency equals CL_{max}
Hence, we consider a system in which

$$\forall t \ : \ WL(t, m) \leq B \ , \quad \forall t \ : \ CL(t) \leq CL_{max} \ , \quad B \leq N_{res}.m \ , \quad CL_{max} \leq B \tag{5.2.4}$$

Design constraints

In a design process there is usually a tradeoff between the number of processing elements (N_{res}) and the latency of the system (Lat). Since a processing element can be busy with a computation for at most CL_{max} time units and then it is free for reuse, the maximum number of processing elements ever needed $N_{res_{max}} \leq CL_{max}$.
In order to ensure that the system has more computational capacity than what is needed the following must hold $B \leq m \times N_{res}$. Hence the minimum number of processing elements is limited by $N_{res} \geq \frac{B}{m}$. This means that the following relation holds:

$$\frac{B}{m} \leq N_{res} \leq CL_{max} \tag{5.2.5}$$

Similarly if the system is unconstrained with respect to the number of processing elements the latency of the system will be CL_{max}.

5.2.3 Theoretical calculation of the latency

In this section we obtain an upper bound for Lat_{max} theoretically. In order to achieve a working system with delay Lat_{max} clock cycles, max_{buf} as the maximum number of buffers needed given B as a workload bound, CL_{max} as the maximum number of clock cycles needed for a single computation, and the number of processing elements N_{res}, the following theorem must hold.

Theorem 5.2.3.

$$\boxed{Lat \leq \frac{1}{N_{res}}.(B - CL_{max} - N_{res}.\lfloor \frac{B-CL_{max}}{CL_{max}} \rfloor + (N_{res} - 1).(CL_{max} - 1)) + CL_{max}} \tag{5.2.6}$$

Proof. First we consider the flow relations of the system:
Flow relations
In general not all cycles of the processing elements will be used. Therefore we define the total number of idle cycles in a period starting at t_1 up to and including $t_2 - 1$, as $IC(t_1, t_2 - t_1)$.

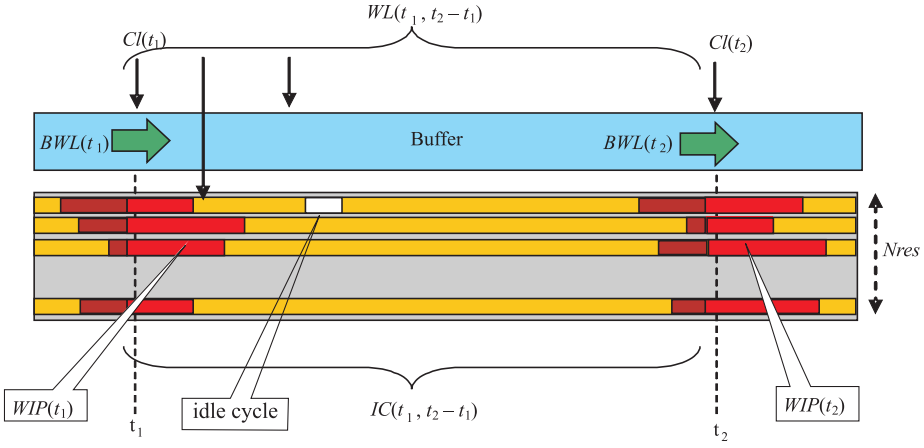


Figure 5.6: The flow relations

From the flow of operands the following can be derived (see figure 5.6):
In the period $t_1, \ldots, t_2 - 1$ a number of $N_{res}.(t_2 - t_1) - IC(t_1, t_2 - t_1)$ cycles are executed.
These cycles satisfy:

- the workload associated with the operands in the buffer at t_1,
- the workload in progress at t_1, and
- the workload of the operands received by the system in the period $t_1, \ldots, t_2 - 1$. Minus
  - the workload associated with the operands in the buffer at t_2 and
  - the workload in progress at t_2.

So

$$N_{res}.(t_2 - t_1) - IC(t_1, t_2 - t_1) = WL(t_1, t_2 - t_1) + BWL(t_1) + WIP(t_1) - BWL(t_2) - WIP(t_2) \tag{5.2.7}$$

For convenience we define:

$$K(t) = BWL(t) + WIP(t) \tag{5.2.8}$$

Then (5.2.7) becomes

$$N_{res}.(t_2 - t_1) - IC(t_1, t_2 - t_1) = WL(t_1, t_2 - t_1) + K(t_1) - K(t_2) \tag{5.2.9}$$

The maximum latency

We wish to determine the maximum latency of a system as described before.
Recall (5.2.3):

$$\forall t \ : \ lat(t) = del(t) + CL(t) \tag{5.2.10}$$

and thus

$$\underset{t}{MAX}\ lat(t) \leq \underset{t}{MAX}\ del(t) + \underset{t}{MAX}\ CL(t) \tag{5.2.11}$$

So we first determine the maximum value of $del(t)$.

The delay of an operand

The delay of an operand t depends on the buffered workload $BWL(t)$ and the work in progress $WIP(t)$, see figure 5.7. In each cycle at most N_{res} cycles are executed.

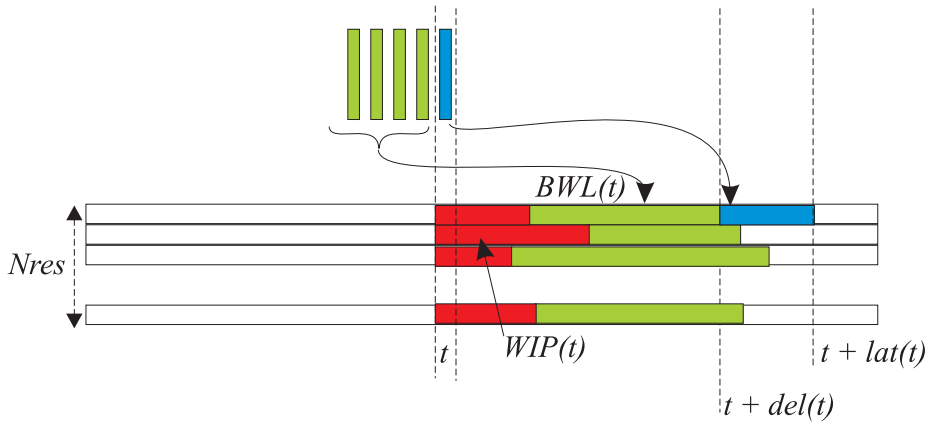


Figure 5.7: The flow of data

So an operand t starts executing just after the buffered workload at t and the workload in progress at t minus the workload at $t + del(t)$ has been executed. Clearly between

$t_1 = t$ and $t_2 = t + del(t)$ are no idle cycles and all workload of the operands after t is executed after $t + del(t)$. So

$$WIP(t_1) + BWL(t_1) - WIP(t_2) = N_{res}.(t_2 - t_1) \tag{5.2.12}$$

The values of $WIP(t_1)$ and $WIP(t_2)$ are bounded by:

$$0 \leq WIP(t_1) \leq N_{res}.CL_{max} \quad \text{and} \quad 0 \leq WIP(t_2) \leq (N_{res} - 1).CL_{max} \tag{5.2.13}$$

From (5.2.11) and (5.2.12) we observe that the maximum delay depends on the maximum value of the buffered workload $BWL(t)$. Therefore we first calculate a bound for $BWL(t)$.

The maximum of the buffered workload $BWL(t)$

First we calculate a bound for $BWL(t)$ for any value of t.

We assume at $t < 0$ the buffer is empty, no work is in progress and all cycles are idle cycles.

As soon as at $t \geq 0$ an operand with $CL(t) > 0$ is entering the system a processing element unit gets occupied. But the remaining execution units are still idle. It might take some time until all processing elements are occupied and the buffer starts filling up. Hence there exists a t_c, $t_c \geq 0$ such that $IC(t_c, 1) = 0$ and $IC(t_c - 1, 1) > 0$.

If there is an idle cycle at $t - 1$, the buffer is empty and operand $t - 1$ will start executing at $t - 1$. At t the buffer is still empty and operand t will also immediately start executing at t. So, if $IC(t - 1, 1) > 0$, the buffer is empty and $BWL(t - 1) = 0$ and $BWL(t) = 0$ too.

However, the buffer might have been emptied at $t - 1$ and have occupied $N_{res} - 2$ processing elements, see figure 5.8. So it can happen that at $t + 1$ all processing elements are used and operand $t + 1$ is directed to the buffer.

From the preceding follows that the work in progress at t is at most $(N_{res} - 1).(CL_{max} -$

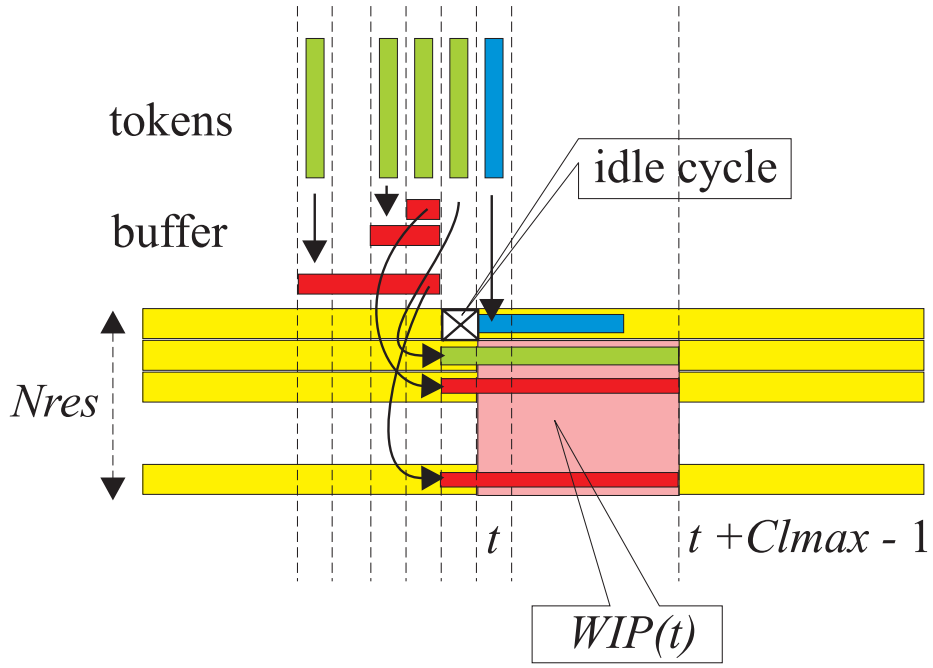


Figure 5.8: $WIP(t)$ if $IC(t-1,1)= 0$

1). I.e.:

$$IC(t_c-1,1)>0 \Longrightarrow \big(BWL(t_c-1)=0 \wedge BWL(t_c)=0\big)$$
$$(IC(t_c,1)=0 \wedge IC(t_c-1,1)>0) \Longrightarrow WIP(t_c) \leq (N_{res}-1).(CL_{max}-1) \tag{5.2.14}$$

and thus

$$(IC(t_c,1)>0 \wedge IC(t_c-1,1)=0) \Longrightarrow K(t_c) \leq (N_{res}-1).(CL_{max}-1) \tag{5.2.15}$$

Yet, consider the workload at t.
From (5.2.9) we know:

$$K(t_2)+N_{res}.(t_2-t_1)-IC(t_1,t_2-t_1)= WL(t_1,t_2-t_1)+K(t_1) \tag{5.2.16}$$

Let t_c-1 be the last empty cycle before t, i.e. $IC(t_c,t-t_c)=0 \wedge IC(t_c-1,1)>0$,

then with $t_2 = t$ and $t_1 = t_c$, we obtain from (5.2.16)

$$K(t) + N_{res}.(t - t_c) = WL(t_c, t - t_c) + K(t_c) \tag{5.2.17}$$

The period $t_c, \ldots, t$ can be divided into a multiple of m and a remainder $t - t'_c$, i.e $t = t'_c + k.m$ with $t - t'_c < m$. So

$$K(t) + N_{res}.(t - t'_c + k.m) = WL(t_c, t - t'_c + k.m) + K(t_c) \tag{5.2.18}$$

or

$$K(t) + N_{res}.(t - t'_c) + k.N_{res}.m = WL(t_c, k.m) + WL(t'_c, t - t'_c) + K(t_c) \tag{5.2.19}$$

From (5.2.4) we know

$$\forall t \ : \ WL(t, m) \leq B \ \text{ and } \ B \leq N_{res}.m \tag{5.2.20}$$

Hence

$$\forall t \ : \ WL(t, m) \leq N_{res}.m \ \text{ and thus } \ \forall t \ : \ WL(t, k.m) \leq k.N_{res}.m \tag{5.2.21}$$

So from (5.2.19) we obtain

$$K(t) + N_{res}.(t - t'_c) \leq WL(t'_c, t - t'_c) + K(t_c) \tag{5.2.22}$$

and with (5.2.15)

$$K(t) + N_{res}.(t - t'_c) \leq WL(t'_c, t - t'_c) + (N_{res} - 1).(CL_{max} - 1) \tag{5.2.23}$$

From (5.2.23) and $WL(t'_c, t - t'_c) = \sum_{i=t'_c}^{t-1} CL(i)$ we see that $K(t)$ is maximal if $WL(t'_c, t - t'_c) = B$ and $t - t'_c$ is minimal (provided $CL_{max} > N_{res}$). So

$$K(t) \leq B - N_{res}.\lfloor \frac{B}{CL_{max}} \rfloor + (N_{res} - 1).(CL_{max} - 1) \tag{5.2.24}$$

or with (5.2.8)

$$BWL(t) \leq B - N_{res}.\lfloor \frac{B}{CL_{max}} \rfloor + (N_{res} - 1).(CL_{max} - 1) - WIP(t) \qquad (5.2.25)$$

So the maximal buffered workload is obtained after the last operand of a block of $\lfloor \frac{B}{CL_{max}} \rfloor$ operands each with a workload CL_{max}. Careful analysis shows that the block can be preceded with one operand in case CL_{max} does not divide B and B mod $CL_{max} < N_{res}$.
Such a block is preceded and followed by empty operands ($CL(t) = 0$), such that $\sum_{i=t}^{t+m-1} CL(i) \leq B$ is satisfied.

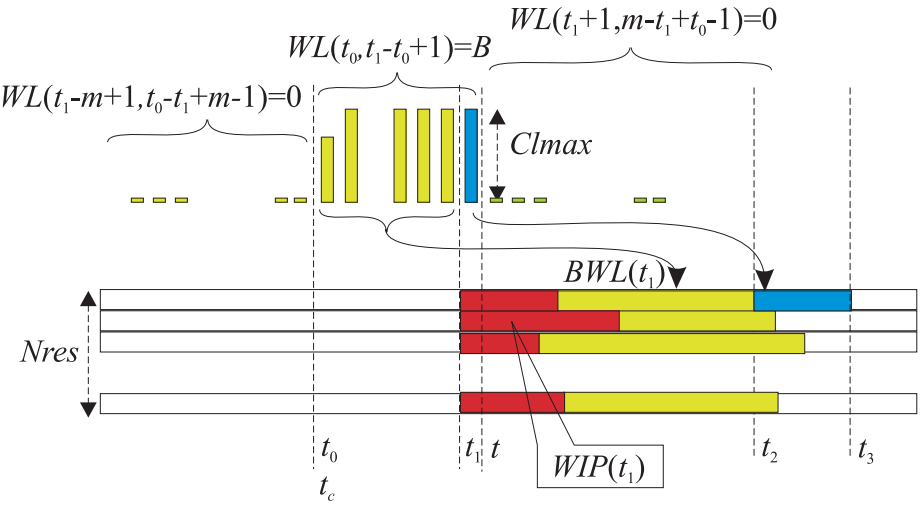


Figure 5.9: $WIP(t)$ if $IC(t-1,1)$= 0

The delay

The maximal latency is determined by the delay of the last operand t_1 in the block that leads to the maximal buffered workload. Obviously the latency is maximal if $CL(t_1) = CL_{max}$. See figure 5.9.
From (5.2.12), i.e.

$$WIP(t_1) + BWL(t_1) - WIP(t_2) = N_{res}.(t_2 - t_1) \qquad (5.2.26)$$

we know that the delay of the last operand t_1 of the block depends a.o. on the

buffered workload at t_1. Formula (5.2.25), however, expresses the buffered workload after the last operand of the block. The maximal buffered workload at t_1 is found by restricting the maximal workload over m operands to $B - CL_{max}$ and then adding the operand t_1 with $CL(t_1) = CL_{max}$. So the buffered workload at t_1 satisfies:

$$BWL(t_1) \leq B - CL_{max} - N_{res}.\lfloor\frac{B - CL_{max}}{CL_{max}}\rfloor + (N_{res} - 1).(CL_{max} - 1) - WIP(t_1) \tag{5.2.27}$$

From (5.2.26) and (5.2.27) we obtain

$$N_{res}.(t_2 - t_1) \leq B - CL_{max} - N_{res}.\lfloor\frac{B - CL_{max}}{CL_{max}}\rfloor + (N_{res} - 1).(CL_{max} - 1) - WIP(t_2) \tag{5.2.28}$$

The unknown value is $WIP(t_2)$. From (5.2.13) we know $0 \leq WIP(t_2) \leq (N_{res} - 1).CL_{max}$, so

$$N_{res}.(t_2 - t_1) \leq B - CL_{max} - N_{res}.\lfloor\frac{B - CL_{max}}{CL_{max}}\rfloor + (N_{res} - 1).(CL_{max} - 1) \tag{5.2.29}$$

And thus

$$\boxed{del(t_1) \leq \tfrac{1}{N_{res}}.\big(B - CL_{max} - N_{res}.\lfloor\tfrac{B-CL_{max}}{CL_{max}}\rfloor + (N_{res} - 1).(CL_{max} - 1)\big)}$$

□

Using the analytical results of theorem 5.2.6 designers of the system can obtain an upper bound of the latency.

5.2.4 Parameter calculation by simulation

In order to check the results of the analytical calculations of theorem 5.2.6 a token-flow simulator was built. Given the system parameters CL_{max}, workload bound B, number of resources N_{res} and window length m, the simulator will generate a data stream with workload $WL(t, m) \leq B$ and then simulate the system for a given number of clock cycles. The final result is the maximum latency obtained during the simulation run and the maximum buffer usage. Running the simulation for a couple of million clock cycles under the specified workload conditions is not a guarantee for the finding

the worst case latency or the buffers of the system, but is a practical enough solution. If $B > m \times N_{res}$ the processing capacity is insufficient and the latency results of the simulator will increase with the number of clock cycles simulated, hence the results are not valid if $B > m \times N_{res}$.

In order to gain an idea of the system behavior under various number of processing elements and load conditions we simulated the system with fixed parameters $CL_{max} = 8, m = 4800, C_{res} = 1$ and then varied the workload bound B from $[1 \dots (m \times CL_{max} = 38400)]$ clock cycles, which is the maximum possible load within that window and the number of processing elements N_{res} from $[1 \dots 8]$ which is the maximum number of needed processing elements for this system. The results shown in figure 5.10 indicate that (for a fixed input workload bound B) increasing the number of processing elements reduces the latency, which was expected, and that for the maximum number of processing elements $N_{res} = 8$ the latency is 8 clock cycles which is confirmed by theorem 5.2.2. The system is capable of handling the workload as long as $B/m \leq N_{res}$ if this is not the case the latency will go to infinity, as there is more workload than computational capacity (i.e. for $N_{res} = 6$ the system is capable of handling a maximum workload of $6 \times 4800 = 28800$ clock cycles).

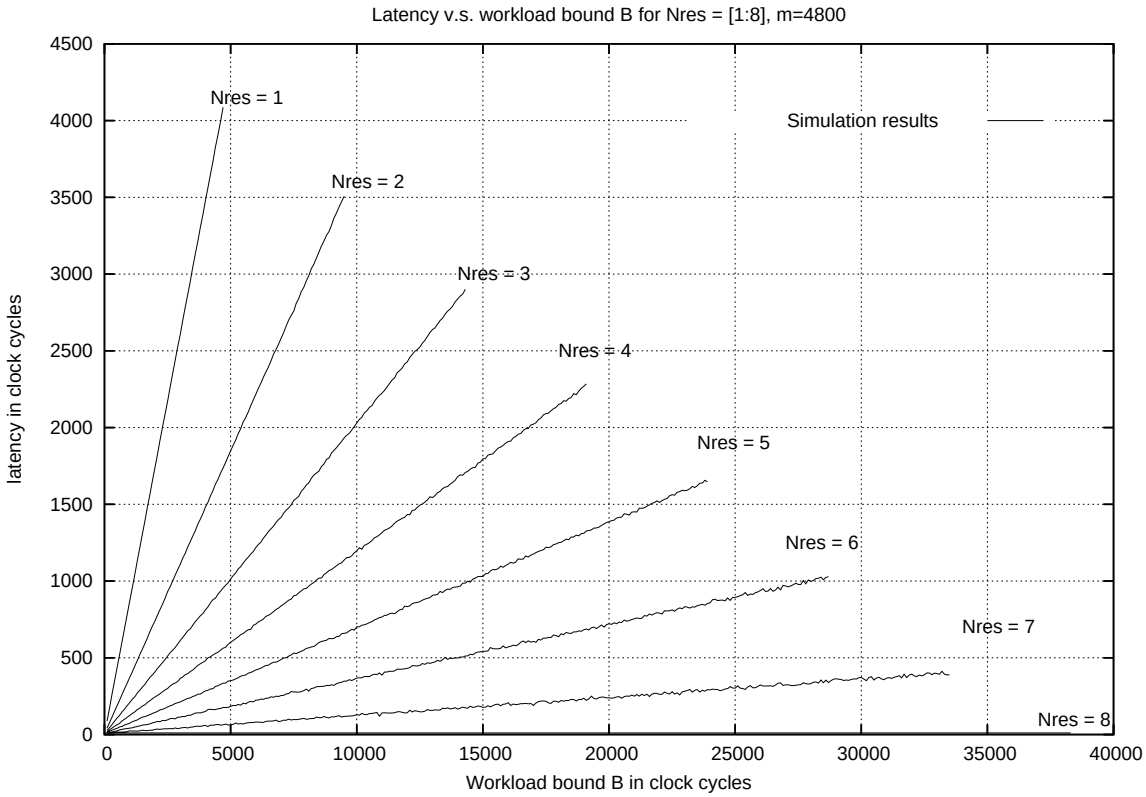


Figure 5.10: Simulation results of workload bound B v.s. obtained simulator latency for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$.

System design parameters such as the maximum latency are obtained from figure 5.10 (i.e. for a workload bound of $B = 10000$ and $N_{res} = 3$ the maximum latency $Lat_{max} \approx 2000$). The simulation also confirms that for $N_{res} = 8$ the system can handle the maximum input workload and at a latency of 8 clock cycles.

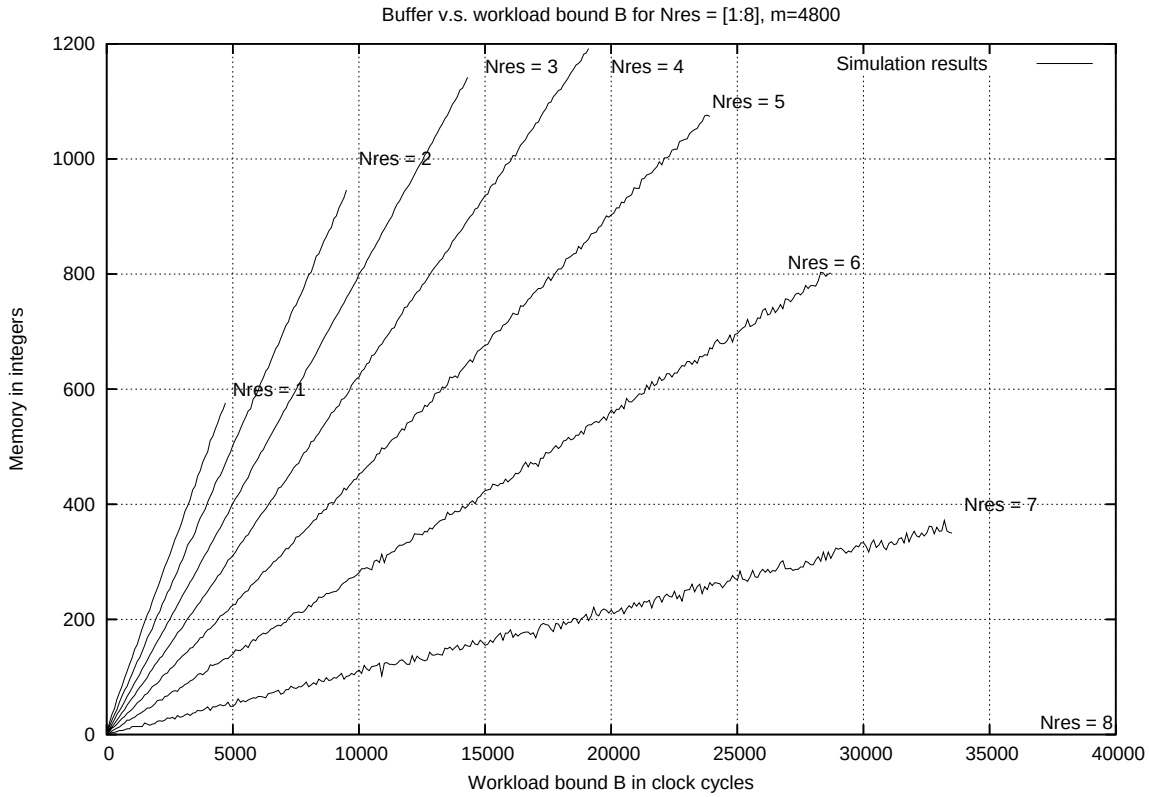


Figure 5.11: Simulation results of workload bound B v.s. queue buffer-sizes for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$.

Figure 5.11 shows the workload bound versus the input buffer size in unit size of the input operands (which are integers in this case) for various values of N_{res}. The figure shows that maximum buffers are obtained when $N_{res} = 4$ and for $N_{res} = 8$ the buffers are 0. This clarifies that if $N_{res} = 8 = CL_{max}$ the latency of the system will also be CL_{max} as there is no buffer build-up. The simulation results also show that if $B/m \leq N_{res}$ and N_{res} is fixed the input workload appears to be linear with the buffer usage.

5.2.5 Comparison of the theoretical latency-bound and obtained simulation results

The theoretical bound given in 5.2.6 can be considered approximately linear and can be simplified as follows:

$$Lat \leq \frac{1}{N_{res}}.\Big(B - CL_{max} - N_{res}.\lfloor\frac{B - CL_{max}}{CL_{max}}\rfloor + (N_{res} - 1).(CL_{max} - 1)\Big) + CL_{max} \tag{5.2.30}$$

Note:

$$\lfloor\frac{B - CL_{max}}{CL_{max}}\rfloor > \frac{B - CL_{max}}{CL_{max}} - 1 \tag{5.2.31}$$

Replacing the l.h.s of equation 5.2.31 with the r.h.s in equation 5.2.30 results in:

$$Lat < \frac{1}{N_{res}}.\Big(B - CL_{max} - N_{res}.\big(\frac{B - CL_{max}}{CL_{max}} - 1\big) + (N_{res} - 1).(CL_{max} - 1)\Big) + CL_{max} \tag{5.2.32}$$

Rewriting equation 5.2.32 results in

$$\boxed{Lat < \tfrac{(CL_{max} - N_{res})}{CL_{max}.N_{res}}.B + (\tfrac{1}{N_{res}}).(1 - 2.CL_{max} + N_{res} + 2.CL_{max}.N_{res})} \tag{5.2.33}$$

Equation 5.2.33 has the form of the single line equation $y = m.x + c$ hence it is linear in B.

Filling in $N_{res} = 1$ and $CL_{max} = 8$ in equation 5.2.33 results in $Lat \leq \frac{7}{8}.B + 0$ and for $N_{res} = 2$ and $CL_{max} = 8$ results in $Lat \leq \frac{3}{8}.B + \frac{19}{2}$.

We know that if $N_{res} = CL_{max}$, the latency is CL_{max}. So for $N_{res} = 8$ the obtained latency Lat should be 8. Unfortunately the bound gives with $B = m.CL_{max}$ and equation 5.2.30 $Lat \leq 14.125$ (notice that Lat turned out to be independent of m in this case).

This indicates that the bound gives higher results than what should be, this is also confirmed by figure 5.12.

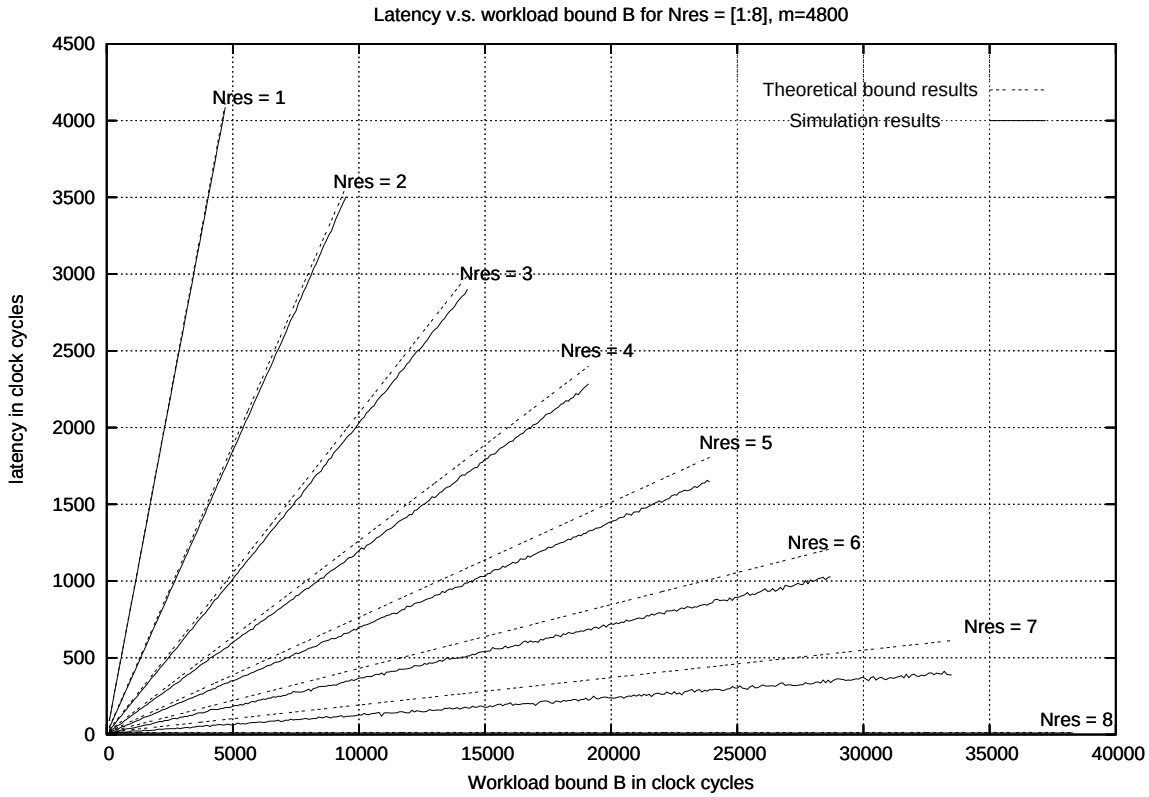


Figure 5.12: Theoretical results versus simulation results for various N_{res}, $CL_{max} = 8$, $m = 4800$, and $0 < B < m.N_{res}$.

In the figure we see the simulator obtained latencies versus the calculated latencies for equation 5.2.6. The figure indicates that the latency Lat is indeed linear with the input workload and that the error made by the theoretical bound increases with N_{res}.

5.2.6 Design flow of simple model applications

The application, to be designed, is a single non-manifest function described in an imperative language such as C. We assume a typical input stream for the simulator is available. Typical implies that "if the system can manage the input stream we are satisfied". The design of the processing elements is part of the design process. During the design of the processing element a good impression on the number of clock cycles needed to execute a particular line or block of source code, is obtained.

Important design parameters are:

- Buffer size
- Number of processing elements
- The maximum latency Lat
- CL_{max}

Note: building a full simulator which will execute real application functions, and its processing elements is not very useful in determining the parameters because of simulation speed.
The design flow can commence as follows: The maximum workload bound B, for different values of m and CL_{max}, is obtained from the typical input stream by a profiling process. Various latencies, and buffer memories are then obtained by simulating the system using the obtained parameters and iterating over different values of N_{res}. Note: $1 \leq N_{res} \leq CL_{max}$.

Example

We demonstrate the scheduling process by means of the *Gcd* example given in section 5.2.1: The *Gcd* algorithm is to be implemented in a synchronous environment. Synchronous means the time between of arrival of the input operand of the system and the production of the accompanying output is constant. The algorithm has to process the input samples on each clock cycle, we know from the specification of the data input stream that it has a $WL(t, 200) \leq 600$ clock cycles (hence $B = 600$), and from profiling results of the algorithm we know that the CL_{max} is 69 clock cycles (See table 5.1). Simulating those parameters using $N_{res} = \lceil \frac{600}{200} \rceil = 3$ produces the maximum latency $Lat_{max} = 256$ and maximum buffer size of 70.
Note calculating the latency using the analytical bound given in equation 5.2.6 produces a latency of 284 clock cycles which is higher than the simulated results.

Table 5.1: System design paremeters

Type	range	units
CL_{max}	69	CLKs
Lat_{max}	256	CLKs
B	600	CLKs
m	200	CLKs
N_{res}	3	Processing Elements
input buffer size	70	CLKs

Simulation results

The simulator of the simple-model is a cycle count simulator. In other words execution of the non-manifest operation is simulated by just decrementing a counter. The counter is initialized by required number of clock cycles obtained from the input stream. In the simulator input stream values represent clock cycles and not the actual operands of the non-manifest function. In this way simulating the application is much simpler and less time consuming. Figure 5.13 and 5.14 show pictures of simple model simulator for the gcd-example. In 5.13 the simulator shows the obtained simulator latency, theoretical bound latency and memory buffers by simulating the system on a random input stream.

The workload bound B of the stream is configured to be below 600 clock cycles and the input stream values do not exceed the value CL_{max} which is 69 clock cycles in this case. The number of processing elements is chosen to be 3. Other parameters of the simulator are the number of clock cycles to execute and the level of randomness. The later allows us to simulate the system with random data around the configured bound value B.

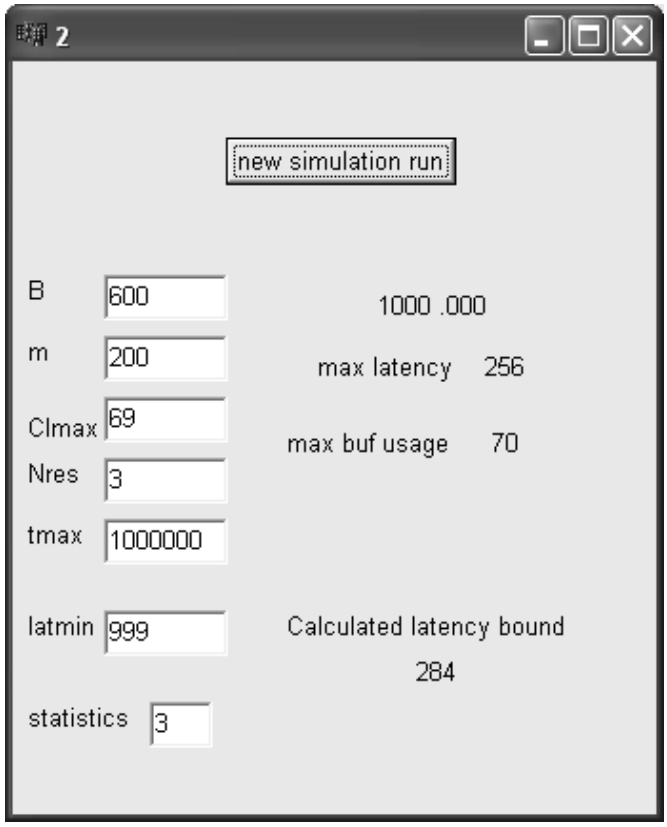


Figure 5.13: Simulation of the system

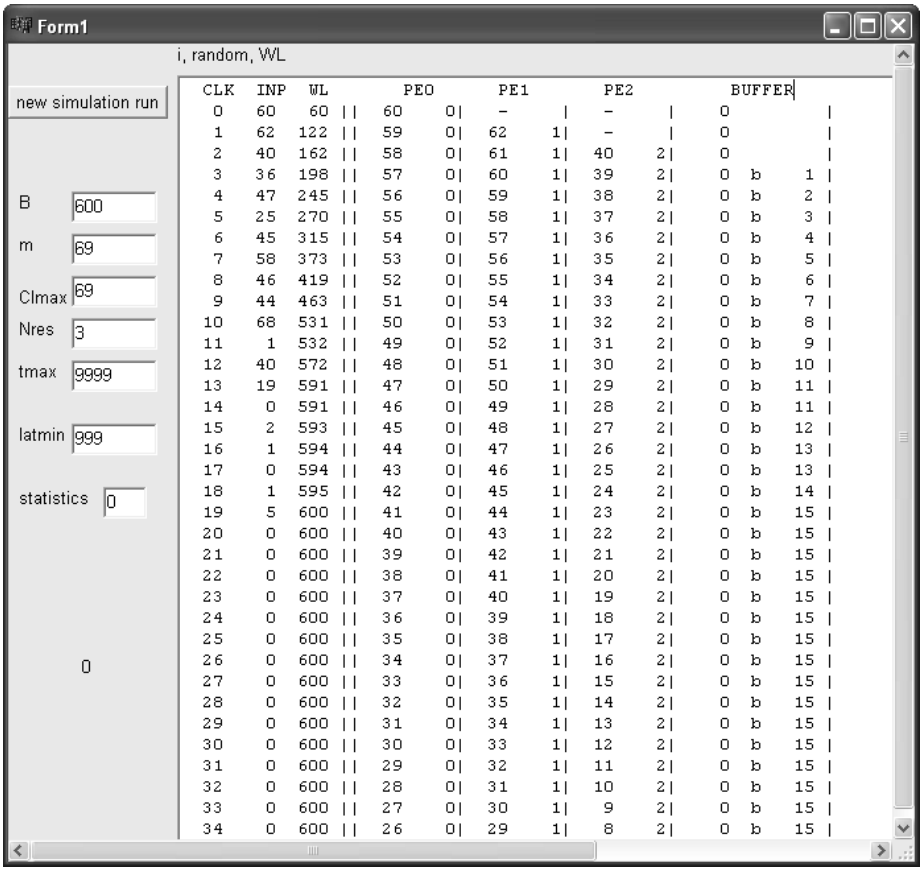


Figure 5.14: Simulation of the system

In figure 5.14 simulation of the scheduling process can be seen. This allows us to monitor the process of dispatching the operands (input stream values) to the processing elements and to monitor the buffers of the system.

5.2.7 Hardware Implementation

In figure 5.3 a block model of the scheduler and the processing elements was presented. The model consists mainly of the following parts:

- Input buffer (FIFO)
- Time queue (FIFO)
- Time counter ($Time$)
- Start time registers ($start_1 \ldots start_n$)
- Processing Elements ($PE_1 \ldots PE_n$)
- Address selection and Calculation Unit (ACU)
- Reorder Buffer (multi ported shift register)

System operation

Since the workload of the system is not constant, all processing elements may be occupied at the moment an input value arrives. In that case the input value has to wait in an input queue until a processing element becomes free. The input values are delivered to the processing elements in order and if a processing element is available at the time an input arrives at the system, the input is transmitted though the queue into the processing element in the same clock cycle. So the input queue acts as a FIFO that has the property that a operand can be accepted and released again in the same clock cycle. Moreover many processing elements can be freed at the same clock cycle and therefore the input queue must be able to release many tokens in the same clock cycle. This might become a design bottleneck.

Since the execution of an operation takes a variable time and because the tokens are delivered to the execution unit in order, some computations might produce their output earlier than preceding computations. In that case the output stream becomes out of order. In order to ensure synchronism between the input and the output stream, the output data of the processing elements are placed in a reorder buffer. The reorder buffer delays the output value such that the time the input value is delayed in the input queue plus the time that was needed for executing the input value plus the time the resulting output stays in the output buffer is constant, viz. Lat . Clearly, Lat is a constant and $Lat \geq Lat_{max}$. This reorder buffer can be modeled by a shift register that shifts right the data one cell each clock cycle and of which all cells have write access. The cells are enumerated from right to left starting with cell 0. Clearly, data entered in cell k will be released by cell 0, k clock cycles later. Because the output values leave the processing elements out of order, they need to be provided with an identifier. For that, the inputs are provided with an identifier being the time they arrive at the system. These identifiers are stored in a time queue parallel to the input queue and are send to the start register of a processing element in parallel with the data that is being send to the input of a processing element. At the moment a processing element releases its output value, the original input value is already $T_{current} - T_{start}$ clock cycles in the system. In which $T_{current}$ is the current time (in clock cycles) and T_{start} is the identifier (time stamp) of the input value. Clearly, the output value still has to be delayed $Lat - (T_{current} - T_{start})$ clock cycles by the reorder buffer and therefore the output value is stored in cell $Lat - (T_{current} - T_{start})$ of the reorder buffer.

Notice that similar to the communication bottleneck between the input queue and the processing elements, there exist a communication bottleneck between the processing elements and the reorder buffer. In worst case, all processing elements could complete their computations during the same clock cycle and thus need to communicate their data in the same clock cycle to the reorder buffer.

These two communication bottlenecks can only be solved in case the average processing time and amount of processing hardware is large compared to the time and hardware that is needed for communication. So in case of coarse grained parallelism.

Architecture properties and shortcomings

Within a synchronous system the life cycle of the stream data samples has the following loop:
1) Waiting to be computed,
2) Dispatching data to Processing element,
3) In execution,
4) Write back of output data to reorder buffer,
5) Waiting (in shift register of reorder buffer) for output dispatch.

The model described in figure 5.3 immediately follows from this loop. However a hardware implementation has a lot of redundant memory.
Notice that if the incoming data cannot be allocated to a processing element directly (because all processing elements are occupied with previous computations), it will be placed in the input buffer together with the time of its arrival (in the time queue). And once a processing element is free this input data will be allocated to the free processing element for processing. The processing element will be busy with the processing for at most CL_{max} cycles. The number of memory places within the reorder buffer equals the latency Lat of the system. This implies that, if an input data is in the input buffer there is always a free place for that data within the reorder buffer, as each data sample, with $CL(v_j) < Lat$, reserves an output position within the reorder buffer.

Notice that the time queue can be replaced by a simple counter that counts the operands dispatched to the processing elements and stores its value in the start buffer of those processing element. This is allowed because the input buffer acts as a FIFO. Obviously this counter equals the time counter if the input queue is empty.
So the architecture can be improved upon, if we devise a system with one buffer (for both the input- and output data) were the number of memory elements, within the buffer, is equal to the latency Lat, and allowing the outputs and inputs to be read and then written to each memory element in a cyclic fashion. Such a system would combine the input buffer and the reorder buffer and get rid of the time queue and

address selection unit of the architecture described in figure 5.3.

5.2.8 Improved Hardware Architecture

In this section we examine an improved hardware architecture in more detail (see figure 5.15). The architecture consists of the following elements:

- A single memory buffer with $N = Lat$ memory elements, each memory element can store the operands of the function implemented by a processing element PE.
- A number of K processing elements $PE_1 \dots PE_K$, each processing element implements the same functionality.
- Two memory pointer registers called $first, last$.
- K registers, one for each PE, that store the id's of the input values that are currently processed.
- A scheduler which controls the scheduling process of the system, and
- Busses for communicating the operand data between the memory buffer and the processing elements.

The system operates as follows: a single memory buffer $MEM[N]$ with N memory elements organized as a cyclic queue, is used to store the incoming data operands (stream samples). The life cycle of the operands is similar to the one described earlier. Two memory pointers are used to store the position of newly incoming operand and output data ($first$) and the position of the oldest operand waiting to be dispatched to a processing element ($last$). At system initiation, *first* and *last* will point to the same memory address. On each clock cycle an output stream sample will leave the system and a new input sample will enter the system. The steps taken to read a new input stream sample and produce an output stream sample are summed up by the following actions which are performed in a cyclic fashion.

- **First:** On each clock cycle, the scheduler will read the operand stored within the memory position pointed to by the register *first* and produce that value on the output stream of the system. After that it will read the newly arrived input data an place it at that same memory position. Finally, it will increase the pointer position of the register $first$ according to the following equation: $first = (first + 1) \mod N$.

- **Second:** the scheduler will scan the processing elements to see whether they have completed the computation of a previously dispatched operand. If some processing elements have completed a computation (and are ready for a new one), the scheduler will write their outputs into the memory buffer MEM using the information stored in the allocation table. I.e. the output is restored in the buffer MEM at the same location as the input was stored. For example if the input with id 4 is executed by PE_3 the id 4 is stored in the id register of PE_3 and the result of the computation is stored at location 4 of the buffer MEM.

- **Third:** the scheduler will try to dispatch the operands stored within the memory range $MEM[first \ldots last]$ starting from the operand stored at $MEM[Last]$ to the available ready processing elements.
  Note: the memory range $MEM[first \ldots last]$ now behave as the input queue of the system. On each successful dispatch of the input value identified by $last$ to the processing element PE_i, the scheduler will store the value $last$, i.e. the id of the operand in the id register of processing element PE_i.

- **Finally:** The processing elements will then perform the required computation, within at most CL_{max} clock cycles. The processing elements will indicate whether they are active or not using an active signal, which is monitored by the scheduler. This scheduler uses the active signal, of the processing elements, and the information in the allocation table to decide whether a processing element is ready for a new computation.

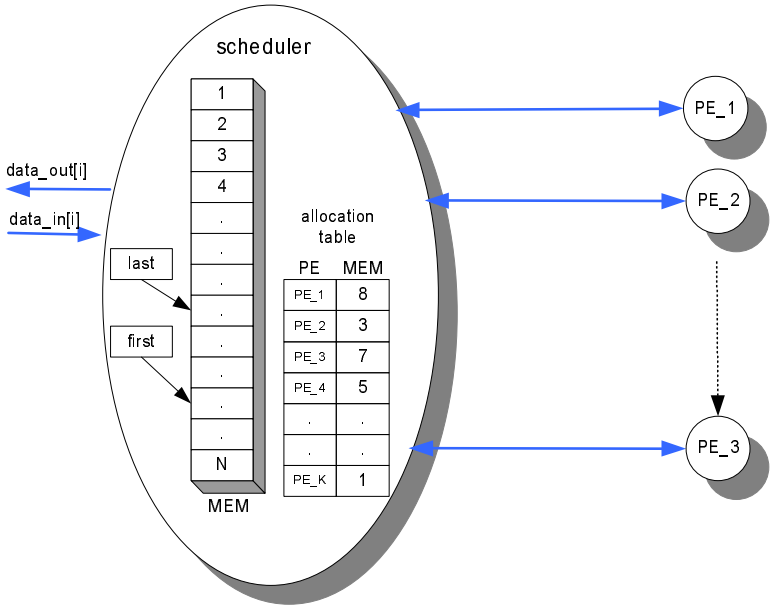


Figure 5.15: The new hardware model

Discussion

This improved architecture uses a single memory buffer for the system operation. The latency of the system *Lat* is predetermined and calculated by simulation as described in section 5.2.4. Also this design faces the same problems of writing many output results at the same time to the memory buffer (multiple communication) and communicating inputs to processing elements in the same clock cycle. Hence the memory buffer should be multi-ported and there should be a sufficient number of busses that communicate the data between the processing elements and the memory buffer. Visa versa, in practice this is an expensive solution. One way of solving this is to delay the system and reserve extra clock cycles for writing the results of the processing elements to the memory. This is a feasible solution if the number of processing elements is not large, as the increase of latency of the system in this case would be proportional to the number of processing elements.

Another design problem is the communication overhead between the processing elements and the scheduler see figure 5.16.

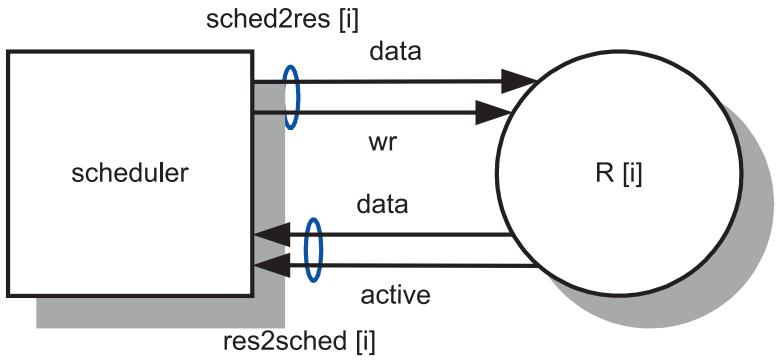


Figure 5.16: Communication busses between scheduler and processing element

If the communication is synchronous, the transport of data operands will take one clock cycle extra for each *wr* action. Since the scheduler has an output register and the procesing elements have an input register and they are both synchronized using the same clock signal. If we assume that each iteration of a non-manifest computation takes one clock cycle, than the complete computation will take at most $CL_{max} + 2$ clock cycles. One solution is to use the design parameter $CL_{max} + 2$ instead of CL_{max}. This will allow us to maintain synchronicity at the cost of extra latency. Another solution is to design the processing element as a Mealy model and the scheduler as a Moore model (see figure 5.17).

In this way, communication within one clock cycle will have a path with only one register which is within the scheduler. This means that if a computation takes one iteration it will consume one clock cycle[1].

[1]**Note:** Although the communication overhead is lower than the synchronous communication solution, the critical path is probably longer.

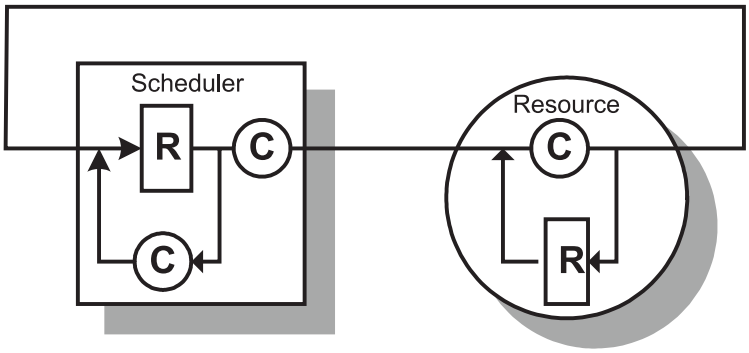


Figure 5.17: Communication model between scheduler and processing element using Mealy and Moore models

5.2.9 Conclusions of the simple model

In 5.2 we have shown how to design and implement a processor capable of handling an operand stream. The processor specification is a single algorithm implemented in an executable specification language such the programming language "C". The algorithm is a non-manifest algorithm which means that the number of clock cycles required to perform a single computation is not known off-line or at compile time. Design parameters of the processor are :

- latency
- number of processing elements
- size of the buffers

The latency could be obtained theoretically by equation 5.2.6 which provides an upperbound to the latency. Better results were obtained by a cycle count simulator which is also capable of providing the memory requirements and other design parameters. Simulation results and analytical calculations show that for a fixed number of processing elements the latency of the system is linear to the workload bound B. Increasing the number of processing elements decreases the latency but the latency

is bounded by CL_{max}. Having more processing elements than CL_{max} is useless and will not improve the latency (see theorem 5.2.2).
The implementation of the processor is strait forward once the design parameters are obtained. We have provided a basic processor design and an improved processor. The improved processor reduced the memory requirements of the system by eliminating redundant memory. The implementation requires that more than one processing element can write its output to memory at the same time, this posses a number of problems and results in a design which scales to the number of processing elements.

5.3 The Complex model

From section 5.2 on page 80 we know that it is possible to perform dynamic scheduling in hardware. For an application with non-manifest data dependant loops, dynamic scheduling in combination with out-of-order execution can save clock cycles that would have been wasted if a static scheduling scheme was used. In section 5.2 we discussed applications that could be modeled as a single non-manifest loop. This is a real limitation, since applications usually consist of multiple interacting functions. Those functions can be manifest or data dependent non-manifest. In chapter 2 we mentioned that such applications can be modeled as a directed acyclic *application graph* see figure 5.18.

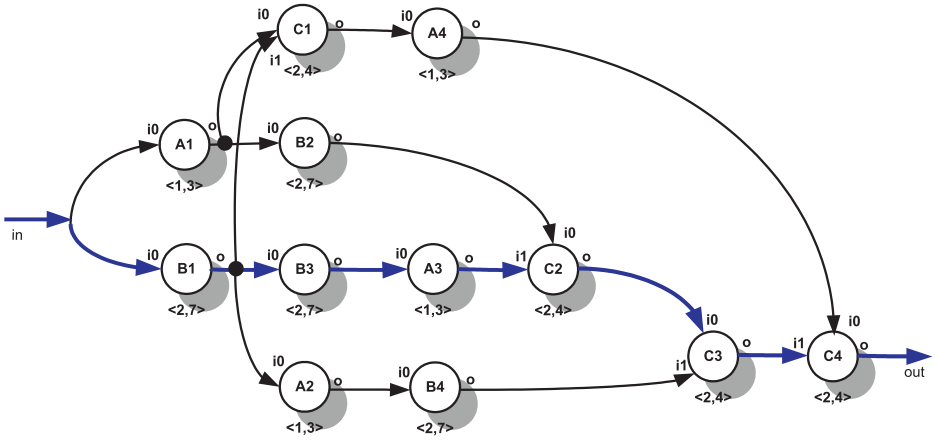


Figure 5.18: An example of the *application graph*

Nodes of the *application graph* represent algorithmic functions. Those functions are usually presented in a programming language as loops. The nodes of the *application graph* are annotated by the minimum and maximum $< min, max >$ number of clock cycles needed for its execution. For example in figure 5.18 node $A1$ has a minimum number of $min = 1$ and a maximum number of $max = 3$ clock cycles. The actual number of clock cycles needed at run time depends on the data (operand values)

that is provided to the function. If the nodes are manifest functions (i.e. min = max), the exact number of clock cycles needed is known in advance. Nodes within the *application graph* do not always have to be represented as loops, they can also be a simple basic-block of the algorithm specification (a sequence of instructions which perform the function needed). In the context of this chapter, we are only interested in the time delay caused by the execution of a node and its dependencies. The edges of the *application graph* represent function dependencies within the application and they carry the operand data from one function to the other. The output operands of the function $A4$, for example, are provided to function $C4$. In this thesis we assume that the *application graph* represents a streaming application: the input-operands arrive at the input-node as a stream of data at regular time intervals and the outputs will be produced at the output node after processing as a stream of output items. In this model there are no dependencies between successive input operands, hence nodes of the application graph do not have to wait for more than one operand on the same input edge and thus the samples arriving at the input nodes lead to independent computations. The execution behavior of a dynamically scheduled architecture for an application with non-manifest data dependent loops, varies in time (i.e. the latency is data dependent) compared to its statically scheduled counter part.

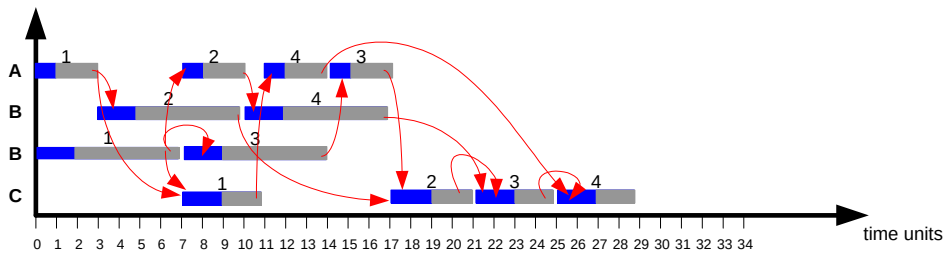


Figure 5.19: A possible static schedule of the application *application graph*

Figure 5.19 shows a possible static schedule of the *application graph* given in figure 5.18. In this schedule we assume there is one processing element of type A, two of type B, and one of type C. Since in a static schedule the number of clock cycles for a function node and hence the execution time behavior has to be known at compile

time. The only possible way to produce a schedule for an application with non-manifest loops is to assume the worst case execution of the processing elements which is the sum of the (worst case) executions of the nodes on the critical path (the path with the longest execution delay). For the *application graph* shown in figure 5.18 this is 29 clock cycles (see figure 5.19). The execution time of a dynamic schedule on the other hand depends on the actual operand values provided at run time. The critical path in this case varies between 11 (the sum of the minimum latencies of the nodes on the critical path) and 29 clock cycles depending on the actual input operands provided.

5.3.1 Problem Description

In the previous section we saw that, for a worst case scenario, a statically scheduled architecture wastes clock cycles when the application contains non-manifest loops. The actual amount of clock cycles saved in a dynamically scheduled architecture depends on the following:

- The applications itself, in other words the difference between the worst case number of clock cycles and the best case number of clock cycles.
- The data set provided at run time (the input stream),
- The number of available processing elements, and finally
- The overhead of the implementation architecture.

Under identical conditions, a dynamically scheduled architecture uses the processing power more efficiently compared to a statically scheduled architecture, because it is capable of using all available clock cycles.
The problem formulation of the complex model is described as follows:

1 Design and implement a processor which is capable of dynamically scheduling the nodes of the application graph, consisting of manifest an non-manifest functions.

2 This processor is to operate in a streaming environment.

3 The inter arrival time δ of two input samples (the minimum time between two successive stream operands) is smaller than the time required to perform the computation of the application graph (it can even be smaller than the computation of a single node), hence, the processor should have the capability of performing parallel computations.

4 The processor should be able to handle the stream throughput at a fixed maximum (to be determined) latency *Lat*.

5.4 The High^2 *DFM*

A possible solution to the requirements mentioned above is the High^2 Data Flow Machine (*DFM*). Which is a coarse grained data flow machine for high-throughput streaming non-manifest applications.

The High^2 *DFM* is derived from the classical data flow architecture and its scheduling is done dynamically in hardware.
Applications of the High^2 *DFM* are high-throughput streaming applications and are modeled by an *application graph* where the nodes of the *application graph* represent the algorithms or functions of the application and the edges of the *application graph* represent the data or operand dependencies between the nodes. High throughput applications are characterized by having a stream sample (operand) rate which is higher than the time needed to perform a single computation. In order to cope with this requirement, the *DFM* consists of a number of high performance execution units (called EU's) interconnected by a network (see figure 5.20). The execution units are implementations of the functionalities within the *application graph* and each execution unit type is capable of computing many independent operands simultaneously. In order to perform multiple computations simultaneously within a single execution unit, the execution units of the *DFM* contain more than one processing element.

Where each processing element is a hardware implementation of a same algorithm type. In order to enhance the performance even further, results of the computations within an execution unit can be out-of-order. Hence, by design, stream operands can overtake each other during execution. This also means that stream operands should be independent of each other, which limits the type of applications suited for this architecture.

5.4.1 Abstract *DFM* Model

DFM structure

An application graph AG consists of number of nodes each identified by a tuple $\langle P, k \rangle$ in which

- P is the type of the node, i.e. the algorithm the node performs, and
- k is the instantiation of that node type. $0 \leq k < inst(P)$ in which $inst(P)$ denotes the number of instantiated node of type P.

Each node is provided with one or more inputs. These inputs of a node are identified by i such that $0 \leq i < inp(P)$ in which $inp(P)$ is the number of input nodes of P. So an input node in AG is uniquely identified by $\langle P, k, i \rangle$. We assume that each node has one output, hence the outputs are uniquely described by $\langle P, k \rangle$. The data flow between the nodes and thus the dependencies are given by the edges between the node outputs and the node inputs in the application graph. Such an edge between the output of node $\langle P_1, k_1 \rangle$ and input $\langle P_2, k_2, i \rangle$ of node $\langle P_2, k_2 \rangle$ is uniquely described by $\langle \langle P_1, k_1 \rangle, \langle P_2, k_2, i \rangle \rangle$.
The successive input values (the operands) are modeled by (identified by) $\langle t, P, k, i \rangle$, in which t is an integer.
The application graph describes the way in which the successive algorithms deal with input stream. The output value of a node only depends on one set of inputs on that node. That means that an output value only depends on the input values $\langle t, P, k, i \rangle$ with $0 \leq i < inp(P)$ of the node $\langle P, k \rangle$. So the consecutive outputs are independent. The dependent operands all have the same t, viz. the time t at which the input

operand entered the system. So the output value $\langle t, P, k\rangle$ of a node $\langle P, k\rangle$ depends on the operands $\langle t, P, k, i\rangle$ with $0 \leq i < inp(P)$.

The High2 *DFM* consists of a number of execution units $EU\_P_0, EU\_P_1, EU\_P_2, \ldots$. Each execution unit EU_P_i is designed for efficiently executing the algorithm specified by the node type P_i in the application graph and all instantiated algorithms $\langle P, k\rangle$ are executed by execution unit EU_P. So the number of execution units in the High^2 *DFM* equals the number of processing types in the application graph. An execution EU_P may contain several processing elements PE that are each capable to execute the algorithm specified by P.

The time difference between two consecutive operands of the input stream is denoted by δ. Hence $\frac{1}{\delta}$ is a measure for the speed of operand arrival (also called data-rate or stream throughput).

Each execution unit EU_P (see figure 5.20) contains an operand table OT, a ready queue RQ, and a set of processing elements $\{PE\}$. RQ is basically the set of matched operands $\{\langle t, k, i\rangle\}$ which await a processing element. The operand table OT holds the arriving operands of the execution unit and stores the destination addresses of each node instance. A summary of the above definitions is given in table 5.2.

Table 5.2: *DFM* Model Summery

name	description
P	process or function type
k	instance number of P
i	node input number ($i \in inp(P)$)
PE	processing element
EU_P	execution unit of node P
RQ	ready queue
OT	operand table
δ	time difference between two stream operands
nodes	$\{< P, k >\}$
inputs of nodes	$\{< P, k, i >\}$
operands	$\{< t, P, k, i >\}$

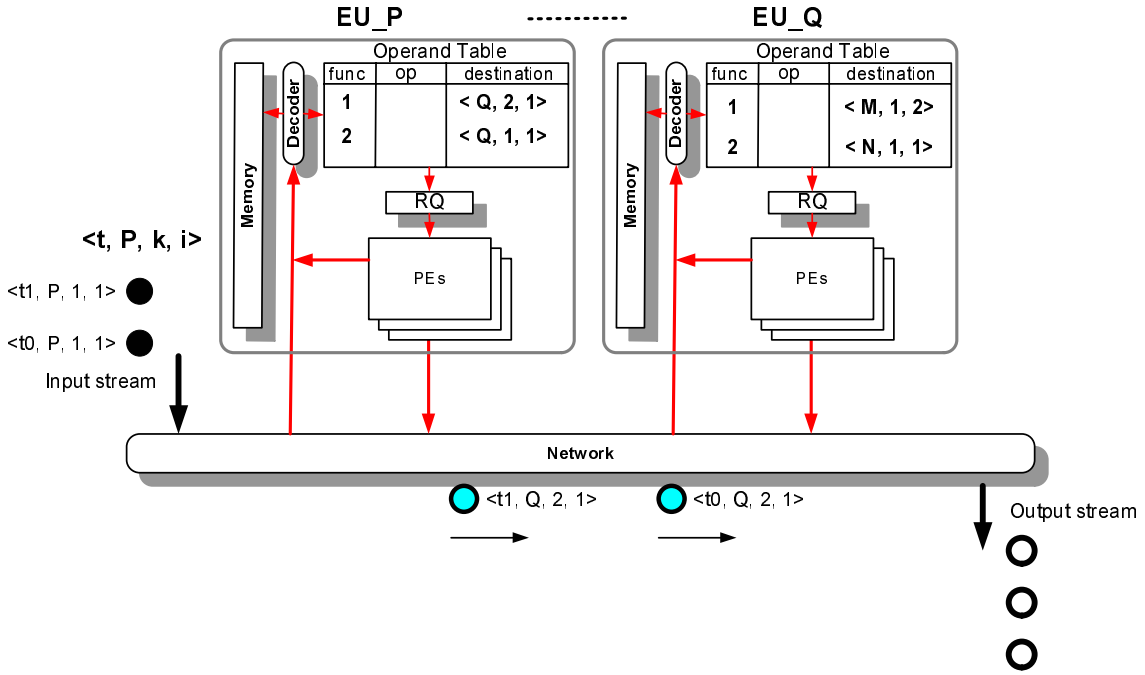


Figure 5.20: The High[2] *DFM* template

DFM operation

The first execution unit within the *DFM*, which is the execution unit representing the first[2] function within the *application graph*, will start processing the input operands. Operands of an execution unit can be in one of the following states:

- **Waiting:** for a matching operand
- **Matched:** all input operands belonging to a function are available
- **Dispatched:** matched operands are dispatched to a free processing element which is executing the function of type P
- **Processing:** the computation on the operands is taking place.

[2]Note: there can be multiple first nodes within this model

- **Transport:** the processing has commenced and the resulting operands are written to their destinations

In order to process the operands of the *DFM* each *EU* performs a number of similar tasks repetitively. The tasks of an *EU* are summarized as follows:

- **Matching**: searching the operand table for matched operands, and writing the found matched operands to the ready queue.
- **Dispatching:** In this step matched operands in the ready queue are dispatched to a free processing element
- **Executing:** In this step the computation of the function identified by node P and operands $\langle t, k, i\rangle$, $\forall i \in inp(P)$ take place in a free processing element.
- **Writing back:** The result of the computation is written to its destination.

Execution of a processing element within an execution unit EU_P can take place as soon as for node instance k and operand time t, all tuples of time t are *matched* (hence all $\langle t, k, i\rangle$ with $0 \leq i < inp(P)$ are in the operand table) and EU_P has a free processing element. In other words, the operands are *matched* and dispatched to a free processing element. In case all processing elements are occupied the triples $\langle t, k, i\rangle$ with $0 \leq i < inp(P)$ are removed from the operand table and a tuple $\langle t, k\rangle$ is added to the ready queue RQ, awaiting a free processing element.

RQ is a set of tuples $\{\langle t, k\rangle\}$, the set represents the matched operands that are ready for execution. Therefore the set $\{\langle t, k\rangle\}$ is ordered lexicographically and as soon as a processing element is free, the top value within RQ, is removed and dispatched to the free PE. During dispatching the operands $\langle t, k, i\rangle$ with $0 \leq i < inp(P)$ are transferred to the processing element.

Once the processing (within the processing unit PE) of the operands identified by the tuple $\langle t, k\rangle$ has been completed, the resulting data is stored and triples $\langle t, k', i'\rangle$ are added to the operand tables of the destination execution units EU_P'. Destination

addresses within the *DFM* comply to the edges in the *application graph*. Hence, the destination address of EU_P is the set of nodes $\{P'\}$ within the *application graph application graph* where there exists an edge within *application graph* that connects node P to the set of subsequent nodes $\{P'\}$. Note: Since each execution unit EU_P within the *DFM* may have more than one processing elements PE and the computations performed by the processing elements are non-manifest, many outputs become available simultaneously. Those output results will become the operands of destination execution units and hence are placed in the operand tables of the destination $EU's$ via a network. This means that the network may eventually form a bottleneck.

DFM implementation details

In each execution unit EU_P the operand $\langle t, k, i \rangle$ is stored for all k in separate tables. In order to synchronize the operands of nodes that have more than one input, an operand indexing scheme is used. This indexing scheme matches operands with identical arrival times t and allows the processing elements of the execution unit to operate on the correct (synchronized) input operands, hence computing correct results. Clearly (due to memory size limitations) the index t cannot be a natural number; hence its range has to be reduced such that no two indices are mapped on the same reduced value. The maximum number of time indices alive in the system is the maximum latency multiplied by the input data-rate $q \geq \frac{Lat_{max}}{\delta}$. So if we chose $idx = t \mod q$ in which q is larger than the maximum number of time indices alive, then $\langle idx, P, k, i \rangle$ are unique. Taking the indices $idx = t \mod q$ makes it impossible to maintain an ordering. This can be solved in different ways. For example by maintaining a value *oldest operand* in each execution unit.

5.4.2 High^2 *DFM* Implementation

The High^2 *DFM* is an implementation of the *DFM* model described in section 5.4.1. It consists of a number of execution units inter-connected by a network and means for scheduling and dispatching the workload offered to the system. Each execution

unit corresponds to a node type in the application graph. For optimizing the number of execution units first a clustering process is used to maximize the number of node types in the *application graph* and at the same time maximize the granularity of these nodes. Maximizing the granularity of the nodes and maximizing the number of node types in the *application graph*, improves the computation to communication ratio and leads to a *DFM* with a minimum number of execution units. Once this is achieved, designing an application based on the High^2 *DFM* merely consists of mapping the *application graph* onto the architecture of the *DFM* and determining a number of design parameters.

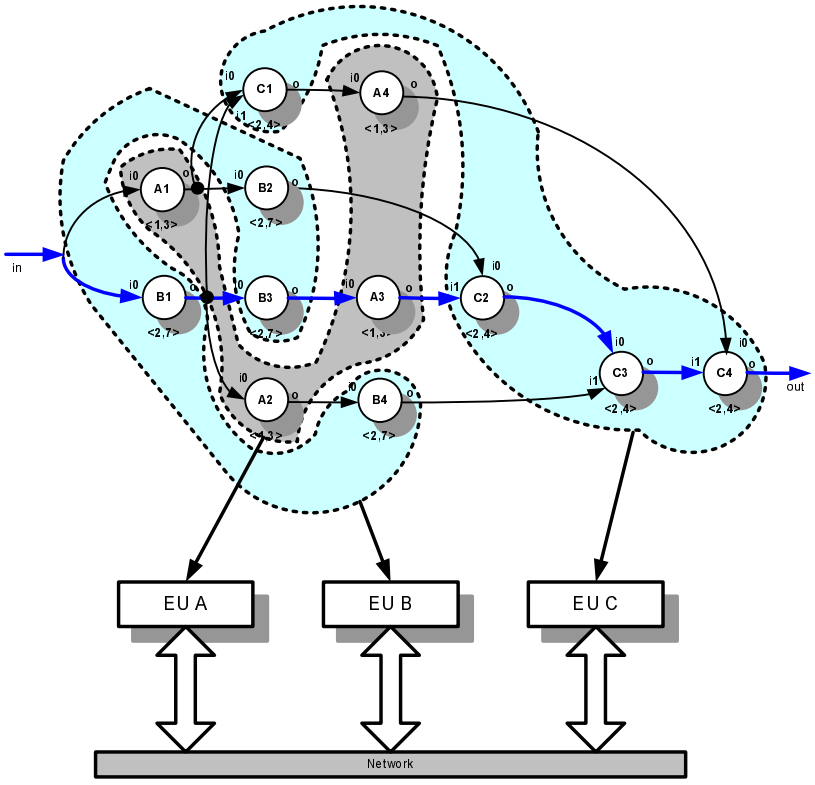


Figure 5.21: Mapping the *application graph*onto the *DFM*

Each node type in the *application graph* maps onto a single execution unit that will

calculate the function specified by that node and each edge maps to a destination address of an execution unit (see figure 5.21). Figure 5.22 gives the *DFM* machine that belongs to the *application graph* of figure 5.18.

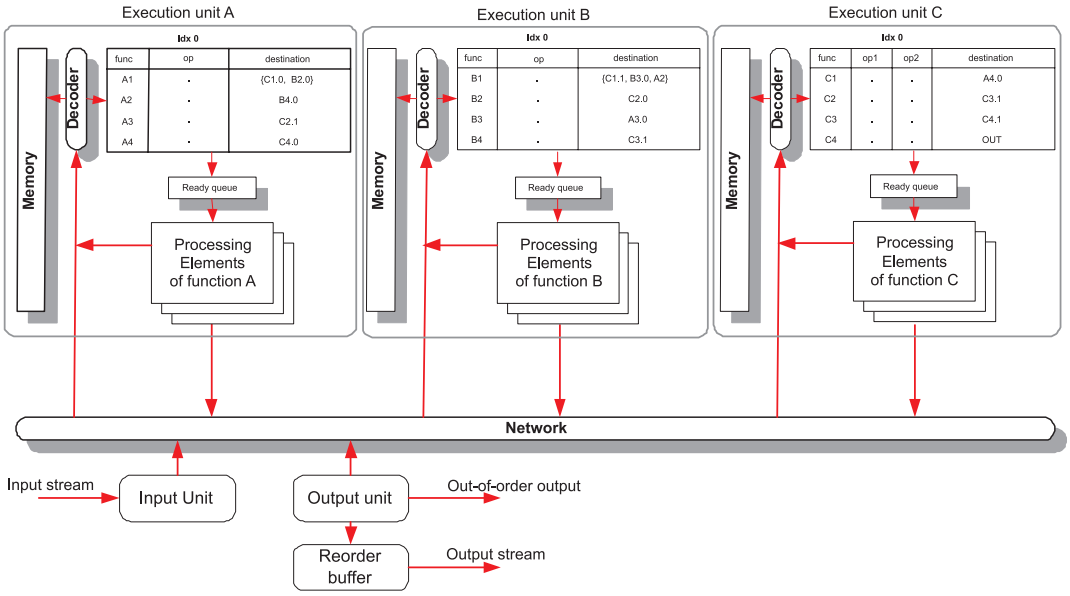


Figure 5.22: The High^2 *DFM* template

Elements of the *DFM* architecture can be summarized as following:

- **Execution Units:** the *DFM* is among others built from a set of execution units
  $\{EU\_P_0, EU\_P_1, EU\_P_2, \ldots\}$. Each execution unit EU_P is responsible for executing a single function-type identified by the node P in the *application graph*. The execution unit can be busy with the execution of many function-nodes simultaneously. This is allowed for because each single execution unit may have more than one processing element PE that will execute the same function. Those processing elements execute in parallel. The number of processing

elements within each EU is one of the design parameters that need to be determined during the design process. The DFM also contains two special types of units that do not perform any algorithmic functions; the input unit and the output unit. The input unit is responsible for transporting the input stream data (operands of the nodes) to the execution unit that implements the first input nodes of the *application graph* (In this case EU_A node $A1$ and EU_B node $B1$ in figure 5.22). The input unit is programmed with those nodes as destinations. The input node issues each new data operand a time index, this time index is incremented according to the indexing scheme described in the DFM model 5.4.1. The output node is also a special node; it is responsible for synchronizing the output stream with the input stream if required. Some applications can benefit from the fact that the output stream is dispatched out-of-order. Similar to the principle of result forwarding used within the Tomasulo scheduler described in section 4.3.3. If the specification requires that the output stream is to be synchronized with the input stream, the time-indices allocated to the operands are used for the synchronization in a similar way as the mechanism used in a reorder buffer. Each execution unit has a memory block for storing the operand-data, an operand table for operand matching and synchronization, a ready queue which will hold matched operands, a number of processing elements; that will perform the actual computation, and finally a memory-controller which is responsible storing the operand-data into the memory and updating the operand table.

- **Network:** The network is responsible for transporting the data operands which contain the operands data, time-index information idx from a source execution unit to the destination execution unit. If the source and destination units are the same unit, transport of data takes place within the execution unit itself. The destinations address (which is programmed within the operand table of the source EU) contains the set of tuples $\{< P', k', i' >\}$. Where P' is the destination EU, k is the instance number of that EU and i is the operand input number. Upon completion of a computation the Processing elements generates the operand tuple $< operand - data, idx, < P', k', i' > >$ and sends it

to the network.

Each execution unit is capable of executing a number of identical functions in parallel. The actual amount of parallel executions it can perform is dependent on the number of processing elements (PE's) it has, and is in fact a design parameter. Parallel operations is needed if the workload per node type is larger than the work that can be performed by one PE. Moreover the High^2 DFM makes use of the inherent parallelism available within the application. Each execution unit within the DFM contains the following elements (see figure 5.23):

- **Memory:** The memory is used to store the operand data. When a new operand data arrives from the network, the decoder places it into a free memory location. The operand data remains in memory until it is needed by a processing unit.

- **Decoder:** The decoder is responsible for allocating new operands to a free memory location within the memory block. After writing the operand to memory, a tuple containing the pointer to the operand data and the of the operand data size, is placed in the operand table. The tuple is written in a three dimensional table at the position, row-id (node instance k), time index idx position, and operand-id (node input number i).

- **Operand Table:** The operand table is used for synchronizing the input operands of a node, in case the function has more than one inputs, and for storing the destination addresses of each node instance. Since the input-operands of a node can overtake each other due to out-of-order execution, they come from different paths at different moments and many nodes of an *application graph* with the same node type can map onto one hardware execution unit, a synchronization mechanism is needed to ensure that the operation is performed with correctly matched input-operands. The operand-table sorts the input operands according to their node-id P, time index idx information, and node input value i called op_id within figure 5.23. The latter indicates at which function input should the operand data be placed.

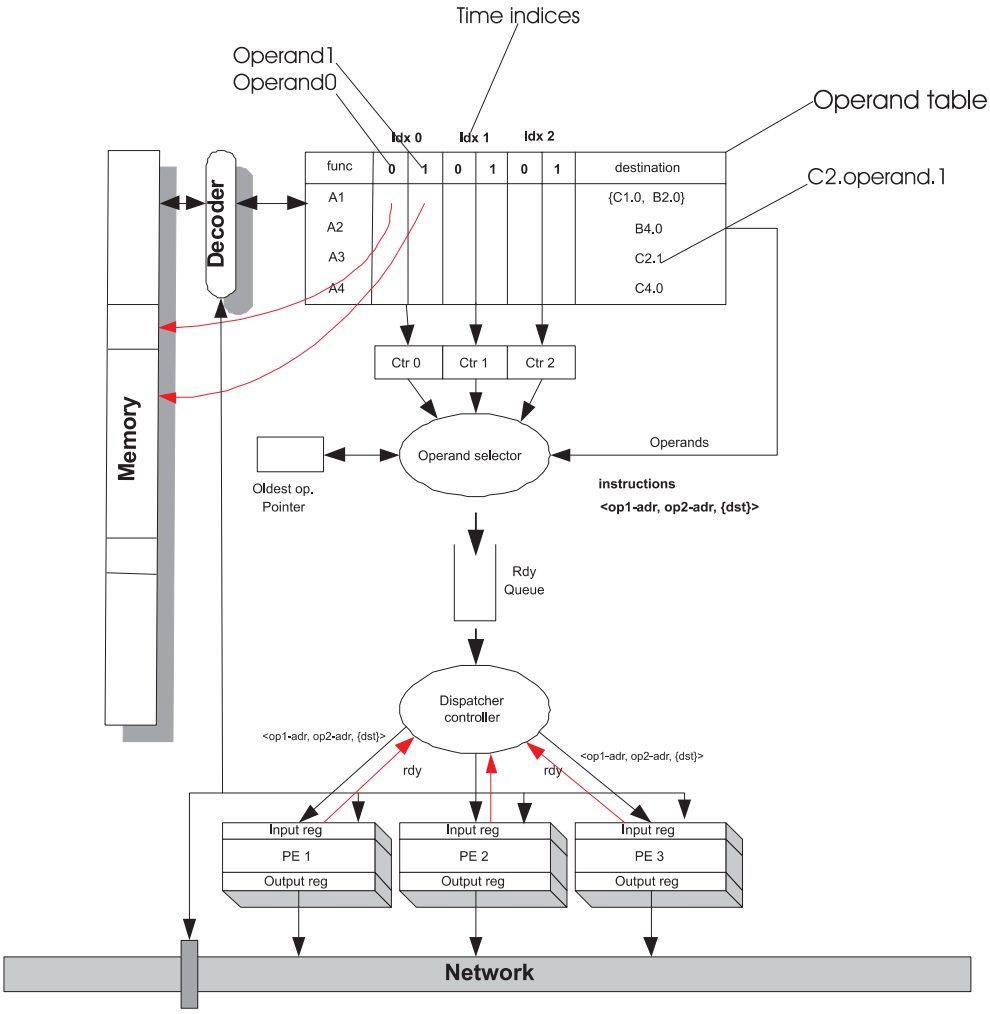


Figure 5.23: Architecture of an execution unit

- **Operand Selector** Once the matching process of newly received operands has taken place. The operand selector will search the operand-table, for those matched operands. Operands are considered matched and ready to be computed if all tuples $\langle idx, k, i \rangle$ with $i \in [0 \ldots inp(P)]$ are available in the operand table and there is a free PE. Once matched operands are found, the operand selector

issues an instruction containing the operands pointers within the local memory of the execution unit in the *DFM* and the destination address information into the ready queue. In order to assure that the oldest operands are serviced first, the operand selector steps through the operand table according to a certain scheme. It first looks for operand-matches at the column indicated by the oldest operand pointer, after that it will look at the other columns. Each column is checked from top-row to bottom-row. This scheme insures that the oldest operands within the operand-table have the highest priority and that no starvation can take place. **Note:** Another scheme that can be used to insure that the oldest operand is serviced first, is to write all matched operands directly to the ready queue once the match has been detected and then sort the ready queue afterwards on the bases of the oldest operand within the queue.

- **Oldest Operand pointer and counters:** The operand selector makes use of an oldest operand pointer and a number of counters (Ctr0, Ctr1, etc. in figure 5.23) whilst selecting matched operands from the operand table. The oldest operand pointer and counters are used to maintain operand ordering within the operand table. The mechanism works as follows: each idx has a separate counter. The counters are used to count the number of matched operands for each specific $idx = t \mod q$ where $\frac{q \geq Lat_{max}}{\delta}$. An overflow in counter $count_{idx}$ will indicate that the all k node instances idx had matched operands and hence, the operands with time idx have left the system indicating that the next operand position is $idx = (t+1) \mod q$ is the oldest operand within the system. The *oldest operand pointer* points to the oldest operand idx and is incremented in a modulo fashion once the counter of the oldest index overflows. The operand selector starts selecting matched operands starting from the position of the oldest index. If a match is found, the matched operands will be written to the ready queue. This will insure that the oldest operand will have the highest priority of being dispatched first and hence the maximum delay of the system is not exceeded by unnecessarily delaying the oldest operand.

- **Ready Queue:** The ready queue basically holds the instructions placed by the

operand selector. Those instructions are stored in fifo-order.

- **Dispatcher:** The dispatcher continuously monitors the state of the ready queue and the ready signal of the processing elements. One or more instruction(s) are dispatched, at the same instance, according to the number of free processing elements and available instructions within the ready queue. After dispatching, the instructions are removed from the ready queue.

- **Processing Elements:** The processing elements are responsible for the actual computation work. The life cycle of the processing elements is as follows:

  - decoding instructions containing the operand pointers and the result destination addresses
  - retrieving operand values stored within the memory of the *EU*
  - executing the function
  - writing back the results and raising the ready signal

  The computation can be a manifest computation or non-manifest. In the case of non-manifest computations, the processing unit will be occupied for a variable number of clock cycles. The processing unit indicates when the computation has finished by raising its ready signal, which is continuously monitored by the dispatcher. Upon completion of a computation, the processing element will write the results to the destination addresses indicated within the instruction. Which could be the execution unit itself or a different execution unit. If the destination execution unit is the execution unit self, the results will written to the local memory of the execution unit via the local busses. If the destination unit execution unit is different, the results will be forwarded to the network which will eventually write those results. The implementation of processing elements can vary from dedicated hardware implementations to general purpose processors. The actual choice depends on the node function and its requirements. Since execution units have different processing element implementations, the complete *DFM* is considered a heterogenous system.

- **Local Busses:** The busses basically transport operand data to and from the processing elements and the memory.

Example

The input node receives the operands from the input stream. The input node packs the operands into network operands. Each operand will contain an index idx, node-id P, operand input number i and the operand-data itself. Indices are given to the incoming operands in a modulo fashion and the total number of indices is calculated by equation 5.4.1:

$$num(indices) = \frac{max_{delay}(application\ graph)}{\delta} \tag{5.4.1}$$

Since operands are allowed to overtake each other in the execution stage, due to the non-manifest properties of the functions, a situation can occur were it would not be possible to properly synchronize the output stream with the input stream. The operand indexing scheme is the mechanism used to allow for the output stream to be synchronized. Operands are indexed upon their arrival and are sorted according to their indices at the output unit. In order to have sufficient memory space to store the maximum number of operands that can coexist at a single execution unit, each operand table will have an operand column field for each index and the memory space requirement is set to be the number of operands of the operand table times the size of the operands data. If the operand-data size is variable, the largest operand data size is used. Notice that the number of time indices calculated in equation 5.4.1 is equivalent to the maximum number of data operands that can coexist in the system at a single time instance.

The input node sends the operands to their destination execution units, which it has been programmed with. Once an operand is received by an execution unit, it will go through a number of steps before being processed. The operand data stored within the operand is copied to a free memory location by the decoder of the execution unit, the decoder will then place another operand containing the address of the operand data, and hence identifying its availability, into the operand table. The position

within the operand table is identified by the node-id P of the *application graph* (e.g. A1 = 1, A2 = 2, etc.), the index number given to the network operand by the input node, and the operand input number i in case the operation to be performed requires more than one input operand see figure 5.24.

In figure 5.24 a number of operands have been written to the operand table of execution unit C. The first column represents index 0 operands the second index 1, etc. The oldest operand pointer has a value of 0 which indicates that the operands within column 0 are the oldest operands within the system. The column counters have the values (0, 1, 0) indicating that there is no operand match in columns 0 and 2 and that there is one operand match in column 1. The operand selector continuously scans the operand table for operand matches starting from the column identified by the oldest operand pointer. If no matches are found in that column the operand selector will scan the next columns. Once an operand match is found, an instruction containing the operand memory pointers, the index of the operand match and the destinations of the operation result is written into the ready queue. The operand match is then removed from the operand table. Figure 5.24 shows that the operand match for node 4 in index 1 has been removed from the operand table and that an instruction has been written to the queue. This instruction points to the operand position in memory and holds the destination address which is the output node. The dispatcher continuously monitors the ready queue and the availability of the processing elements. It will dispatch instructions to the free processing elements. The processing element, will obtain its operands from memory and start the computation. Based on the type of computation that has to be performed a number of clock cycles are consumed. Once the processing element has finished its computation it will write its results to the destination addresses within the instruction. If the destination address is another execution unit, an network operand containing the result data, index idx, node-id P, operand input number i, is written to the network. If the destination is the local execution unit self, the processing element will place the result on its local bus in combination with the index and destination node-id and operand-id. This process is repeated by each execution unit. Once network operands reach the output node, they will be placed in a reorder buffer at a position dependant of idx value. In some

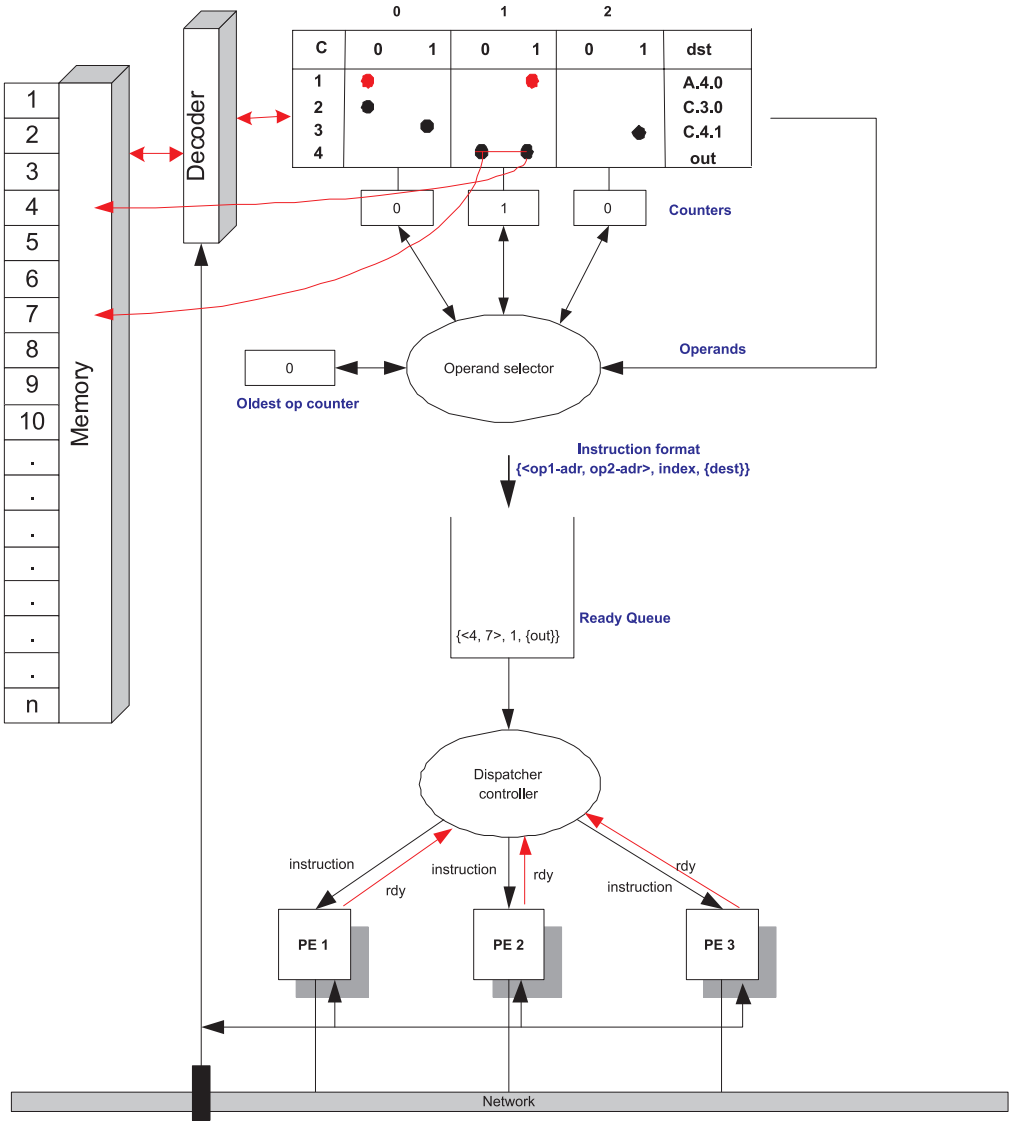


Figure 5.24: Writing an input operand at the correct position within the operand table

situations it is not needed to synchronize the output stream, this is the situation if the order of the operands of output stream is not important. When this occurs the

output will be produced without passing through the reorder buffer.

5.4.3 Design shortcomings

In the High^2 *DFM* described in the previous section a number of shortcomings can be identified:

1 **Global network bottleneck:** due to the out-of-order execution mechanism of the High^2 *DFM*, many processing elements of various execution units can become ready simultaneously. Hence, these processing elements would like to use the network at the same time to transport their operands. The network capacity would on average be capable of handling an average operand load, with a minimum delay. If the situation occurs that all Processing elements of the system are ready at the same time, the network will be overloaded, thus causing extra delays in the system.

2 **Local network bottlenecks:** Another similar problem may exist locally within an execution unit. If all processing elements are released at the same time and the ready queue contains sufficient instructions, all these processing elements will try to use the local busses simultaneously in order to get their new operands from the local memory. Hence, a local bus communication problem will occur.

One way of solving the latter problem is to predict and pre-fetch the operands from the local memory. Prediction would be possible as the execution unit already contains the addresses of the required operands within the instructions dispatched to the ready queue. The pre-fetched operands would be stored in local registers within the processing elements and thus avoiding any extra delays that might be caused by local bus communication bottlenecks.

High^2 *DFM* algorithm properties

We summarize the properties of the High^2 *DFM* as follows:

- The scheduling algorithm is starvation free as all positions within the operand table are checked for a match.

- The scheduling algorithm is deadlock free since there are no cycles within the application *application graph* and the buffering policy ensures that no processing element has to wait on writing its data.

The differences between the High^2 *DFM* architecture and the classical Data flow Machines are:

- Nodes within the High^2 *DFM* are not instructions but coarse-grained pieces of code including non-manifest loops.
- The application domain of the High^2 *DFM* is streaming applications
- The amount of tokens (operands within the High^2 *DFM*) are limited by design

5.4.4 Possible system modifications and improvements

The design provided is just a template and it can be improved in various ways. Some of the possible modifications are summarized below:
Energy aware modifications: by using resizable memory queues we can turn of the queues which are not active and hence be more energy efficient. By varying the number of active processing elements and turning off non-active elements, the system can save energy. Also non active processing elements can be turned off to save power.

Elimination of redundant computations: by checking the operand values within the execution unit memories, the system can detect duplicate operands and hence duplicate identical computations. Since duplicate operands would give the same execution results, the system can avoid duplicate computations by providing the results of the first computation to all successor computations within a constant time frame.

Adaptive functionality or reconfigurable execution units The processing elements of an execution unit could be programmed with more than one function implementation. The actual function is chosen at run time. This allows for dynamic system reconfiguration.

5.4.5 Design Flow DFM

We wish to design a streaming application based on the *DFM* architecture. The application is described by means of an *application graph* which is a set of interconnected nodes. Each node expresses behavior and is described in an imperative language such as C.
We assume that at design time a typical input stream is available. (Typical here implies "if the system can manage this input stream we are satisfied").
We want to optimize the *DFM* system such that it would have a minimal number of processing units with an acceptable overall latency.
The design of the processing elements is part of the design process.
Important parameters of the *DFM* are:

- Number of execution units (is equal to the number of different node types)
- The number of processing elements in each execution unit.
- The maximum latency
- The buffer sizes

In order to determine these parameters a parameterizable *DFM* simulator was developed. Building a full simulator of the *DFM* and its processing elements for determining the parameters is not very useful because of the simulation speed.
We assume that the specification can be simulated on the level of the application graph. I.e. a program is available consisting of the processes that describe the behavior of the nodes of the application graph. For each input sample (value on the input) it calculates all the intermediate values, i.e. the values at the outputs of the nodes.
For each node type a PE is designed. During the design of the PE's we get a good impression of the number of clock cycles needed to execute a particular line or block of source code by the PE. This information is used to augment the process source code with a clock cycle counter. Hence, when simulating the 'typical input stream' we get a good impression about the number of clock cycles that need to be executed by the different $PE's$. This part of the design flow is called the 'Benchmarking'.

So profiling provides us with a file in which for each input value of the typical input stream, the number of clock cycles needed for executing each node in the application graph is given.
So the profiling process produces a table of which the number of rows equals the number of samples of the typical input steam and the number of columns equals the number of nodes in the application graph. The i^{th} row corresponds to the i^{th} sample in the typical input stream and each column is identified by a node identifier. The cell (i, j) contains the number of cycles process j needed to execute for the input sample i.

Table 5.3: Example: cycle count table

	A0	A1	B0	C0	C1
0	11	15	3	55	32
1	7	12	5	17	37
2	15	9	2	23	11
...	...	...	...	...	...

We will call this the 'cycle count table' see table 5.3. Notice that in practice due to its size this file is hardly to inspect visually.
From the cycle count table we can make an estimate of the number of PE's needed in each execution unit.
A first approach would be to determine the average number of cycles per sample and per node type. However most of the workload might be concentrated in a short period which would lead to unacceptable long latencies.
A second approach would be to determine the maximum workload for each node type over all samples. In that case we would get too many $PE's$. Therefore we will derive the maximum workload over a sliding window. Before defining this quantity, we first observe that the number of $PE's$ depends on the cycle count per node type. Hence we reduce the cycle count table by adding per row the cycles of the instances per node type. So the 'reduced cycle count table' of our example will become (see table 5.4.5):
Column A determines the number of $PE's$ needed in execution unit A, column B determines the number of $PE's$ needed in execution unit B, etc. For estimating the

Table 5.4: Example: reduced cycle count table

	A	B	C
0	26	3	87
1	19	5	54
2	24	2	34
...	...	...	...

number of $PE's$, starting at sample i, we determine the number of cycles (work load) needed over a window of m samples, and determine the maximum of this work load over all i. So we determine:

$$WL_{max}(N)(m) = \max_{0 \leq j \leq SIZE-m} \Big(\sum_{i=j}^{j+m-1} CL(i,N) \Big) \tag{5.4.2}$$

in which $CL(i,N)$ is the number of cycles at row i, column N of the reduced cycle count table. I.e. the total number of cycles the nodes of type N have to execute in order to process the input sample i.

Notice that if $m = SIZE$, then $\frac{WL_{max}(N)(m)}{m}$ is the average workload to be executed by the nodes of type N.

The cycle count table also provides us with the minimal latency that can be obtained. We assume a synchronous environment. So the latency of the system is determined by the maximum time it takes to calculate the system output value from an input sample.

Given the cycle count table and the dependencies that follow from the application graph, for each input sample in the cycle count table, the total time (in clock cycles) for calculating the system output can be easily and fast calculated. So we calculate the latency for each sample in the cycle count table starting from the assumption that always immediately a processing element is available (unlimited resources). The maximum taken over these latencies is the minimum system latency of the system, Lat_{min}.
The design flow is supported by software that calculates from a cycle count table the

values for various values of m and the value Lat_{min}.
Starting from $WL_{max}(N)(m)$, we roughly may assume that if we have sufficient processing elements to manage this workload, the data will not have to be buffered much more than m samples before it is executed. Clearly, we want to have a reasonable latency. So a good estimate is to start from $WL_{max}(N)(m)$ with m is about Lat_{min} for each node type N. So the number of processing units $NPE(N)$ of type N is $\frac{WL_{max}(N)(m)}{m}$.
The next step is to determine the exact maximum latency and the buffer sizes. For this purpose we have built a 'token flow simulator' of the DFM. In this simulator no real data is processed. The input samples are considered as tokens that flow though the system and which are delayed in the operand selector (for matching), the ready queue (waiting for a free PE) and in the processing elements. The delay in the processing elements follows from the cycle count table. So the input of the token flow simulator is the cycle count table. The simulator calculates the delay in the buffers, the required buffer sizes, and the maximum latency of the system.
The simulator is fast because only the token flow is simulated.
Playing with different numbers of processing elements in the execution units makes it possible to optimize the system.
Additionally the simulation software provides the possibility to stress the system with maximum workload that just satisfies the maximum workload the system can deal with. I.e. the work load over a period m never exceeds $m \times NPE(N)$

5.5 Conclusions

In this chapter we provided solutions for designing architectures which are capable of handling non-manifest loops in streaming environments. The simple situation, which is the situation of a single non-manifest loop algorithm was discussed in section 5.2. By having multiple implementations of the same algorithm running in parallel on independent stream computations we were able to provide an architectural solution which can handle various stream loads under various throughput conditions. Given a stream load bound we can search for valid solutions in terms of number of processing

elements, latency of the system and memory requirements. Since in practice applications are more complex and can be composed of multiple interacting algorithms, we provided a solution for the general case in section 5.3. This solution is based on ideas taken from data-flow machines. Data flow machines have the property that computations are triggered by the availability of their operands and not by an instruction stream. The combination of data flow-machines with hardware dynamic scheduling provides an architectural solution for the streaming applications we consider in this thesis. The design of the High^2 data flow machine, which is an application specific dedicated processor capable of utilizing the parallelism inherent within an application, is provided in section 5.4.2. The High^2 *DFM* was designed with streaming applications in mind, this means that throughput is of major importance, the *DFM* can be tuned at design time to meet certain throughput requirements, this is achieved by varying the number of processing elements available for computing the (non-manifest) functions of the application. Providing solutions for average case stream processing load and at the same time maintain a high-throughput is possible by varying the design parameters.

Chapter 6

Example of High^2 *DFM* Applications

In this chapter we describe the design method for developing applications using the High^2 *DFM* as a target architecture. The design method is needed to derive important system parameters such as size of buffers, number of processing elements, system latency etc.

6.1 Introduction

In this chapter we show how to design a streaming application, that contains non-manifest algorithms. We will show a detailed example that will describe the High^2 design flow described in chapter 5. The aim of this example is to show how important system parameters are determined and how design choices relating to the number of processing elements, number of buffers, and system latency are made.
The information required to implement an application on a High^2 *DFM* are: the *application graph*, input stream throughput required in the design process, number of processing elements of the execution unit within the *DFM*, and the buffer sizes. The number of processing elements is calculated based on the stream workload-bound (maximum workload within a given stream window of size m) and the input throughput (which is the speed of arrival of the input samples). The input throughput is a

specification parameter and is provided to the designers. The workload-bound can be obtained by profiling the input data stream and measuring the maximum bound within a window of length m samples. The buffer sizes are obtained by simulating the application on the simulator, measuring the maximum buffer sizes needed and multiplying those figures by the size of the input data samples. The rest of the parameters are obtained by a cycle-count simulator, which will use a cycle-count file containing the benchmarked cycle latencies of the application for a uniformly distributed input stream. The simulator will provide the maximum, average, and minimum consumed number of cycles for the provided input stream for a given number of processing elements. This process is described in chapter 5 and will be summarized within the next sections.

6.1.1 Example montgomery-gcd-montgomery

In this section we show how to fine tune a High^2 processor for the Montgomery-Gcd-Montgomery *mgcdm* example introduced in chapter 3.

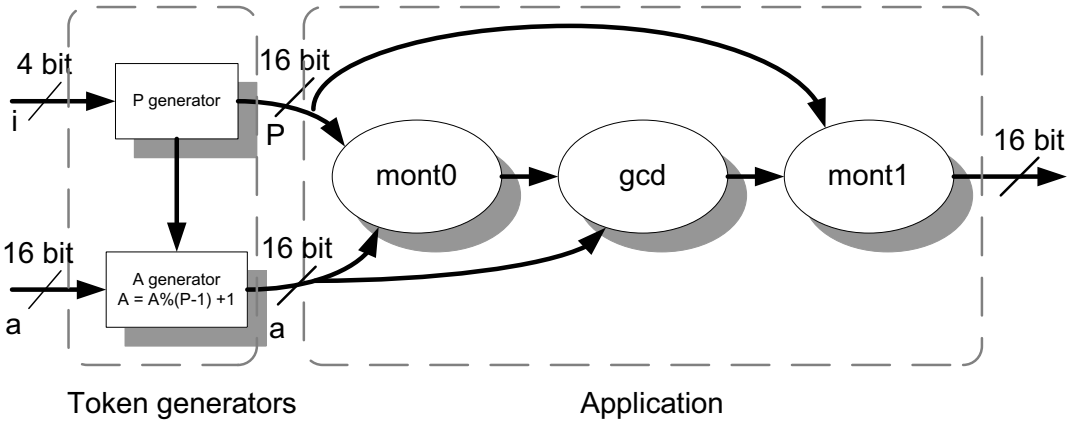


Figure 6.1: The *mgcdm* application

The *mgcdm* application is a fictive application which is used for demonstration purposes only. The Montgomery algorithm within the *mgcdm* application calculates the Montgomery inverse of an integer modulo a prime [41]. It requires the input operand

P to be a prime number and the input operand a to be any value between $[1 \ldots P-1]$. The Montgomery inverses for $P = 7$ can de deduced from table 6.1 by noting that $mont\_inv(a, 7) = a^{-1} \Leftrightarrow a \times_7 a^{-1} = 1$.

Table 6.1: Multiplication modulo $P = 7$

$\times_7$	1	2	3	4	5	6	7
1	**1**	2	3	4	5	6	0
2	2	4	6	**1**	3	5	0
3	3	6	2	5	**1**	4	0
4	4	**1**	5	2	6	3	0
5	5	3	**1**	6	4	2	0
6	6	5	4	3	2	**1**	0
7	0	0	0	0	0	0	0

From the table we can see that $mont\_inv(1, 7) = 1$, $mont\_inv(2, 7) = 4$, $mont\_inv(6, 7) = 6$ etc. In the design flow we need to profile the application and obtain the cycle-count file. In order to provide an input cycle-count file with a uniform input distribution, the application is benchmarked using all possible 16 bit input combinations for the input generator a and all 4 bit input combinations for the input generator P together forming 2^{20} unique input combinations. The reason that the P operand generator is 4 bits, is that we are only interested in the maximum primes under 2^n. The following input prime values 2, 3, 7, 13, 31, 61, 127, 251, 509, 1021, 2039, 4093, 8191, 16381, 32749, 65521, for example are the maximum prime values for $n = [1 \ldots 16]$. In order to do so we have introduced the *P-generator* and *a-generator* operand generator functions to the application graph (see figure 6.1). They will ensure that the algorithms of the application obtain correct inputs (as P input values must be a prime, and the a input values must not exceed the range $[1 \ldots P-1]$). The function *P-generator* in figure 6.1 is basically a lookup table which chooses the highest prime number for a given number of bits e.g. for an input value of i=15, 65521 is the highest prime number under 2^{16} and for the input value i=11, 4093 is the highest prime number under 2^{12}. The input operand a must be limited between 1 and P which is provided by the *A generator* function which basically calculates $a = a \mod (P-1)+1$.

The *Gcd* function within the *mgcdm* application calculates the Greatest common divider of two input integer values.
The output of the application is basically the function
$mont_{inv}(Gcd(mont_{inv}(a, P), a), P)$. Table 6.2 shows the output for $P = 7$ and $a = [1 \ldots 6]$.

Table 6.2: mgcdm calculation

a	P	mont0	gcd	$mont_{inv}(Gcd(mont_{inv}(a, P), a), P)$
1	7	1	1	1
2	7	4	2	4
3	7	5	1	1
4	7	2	2	4
5	7	3	1	1
6	7	6	1	1

6.1.2 Design flow

The design flow is depicted in figure 6.2 and it mainly consists of the following steps:
Step1 profiling the application: to determine the execution time (in clock cycles) for every input combination.
Step2 determine statistics: determine the most important system parameters from the application profile e.g. maximum number of clock cycles taken CL_{max}, maximum workload $WL_{max}(m)$ etc.
Step3 simulation: simulating the application, with a certain workload, using the cycle-count input file provided from the application profile and the parameters obtained from step2. This will provide the latencies of the simulated design and the number of buffers needed.
Step4 Testing: to test the system under extreme workload circumstances a random input generator is used.

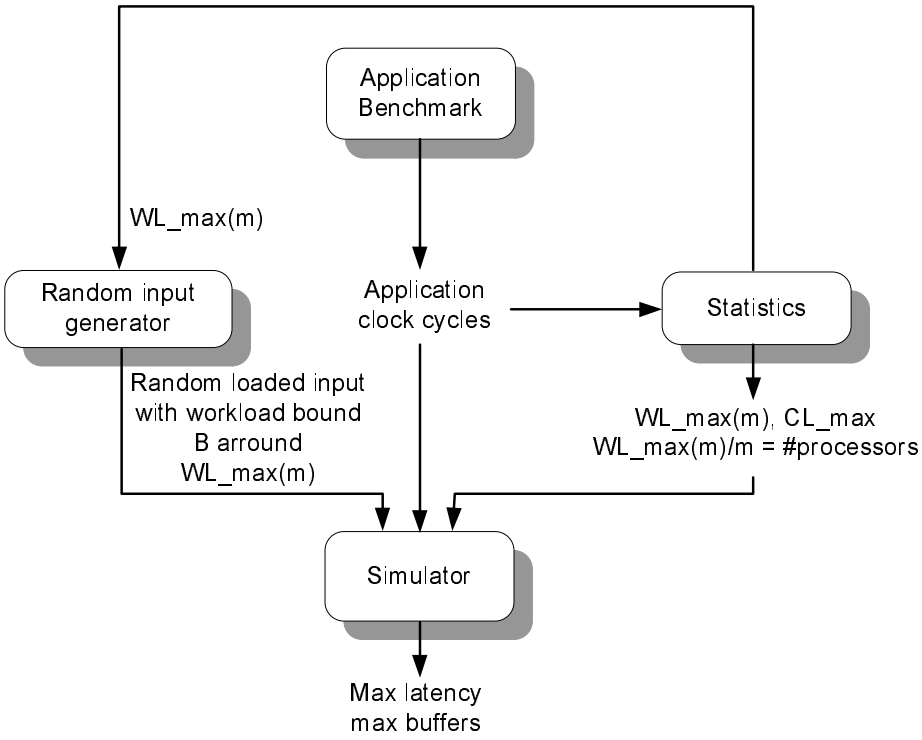


Figure 6.2: The *mgcdm* design flow

Application profiling

The profiling process is basically done by inserting instruction counters into the code of the application see figure 6.3. For each process statement the variable *cl* is incremented. We make the simplification here that each instruction consumes one clock cycle on the implementation of the target architecture. This assumption is inaccurate but sufficient enough for an approximation of the actual number of clock cycles consumed. When more accurate implementation data is available, for instance, when it is known that a code snippet takes n clock cycles, the *cl* variable is incremented by n.

Figure 6.3 shows the augmented code for the Montgomery and gcd algorithms. Each "C" instruction, of the original program, causes the *cl* variable to be incremented.

```
long gcd(long x, long y, long *cl){
    long g, x1, y1;
    g = y;
    (*cl) = 0;           // added benchmarking code
    while ( x > 0 ){
        g = x;
        x = y % x;
        y = g;
        (*cl) += 3;     // added benchmarking code
    }
    return (g);
}

double montgom_inv(long p, long a, long *cl){
    long u, v, r, s, k;

    (*cl) = 0;
    assert ((a > 0) && (a < p));
    {
      u = p; v = a; r = 0; s = 1; k = 0;
      (*cl += 5);      // added benchmarking code
    }
    while(v>0){ // phase i
        if (even(u)){ u/=2; s*=2;
                      (*cl) += 2;      // added benchmarking code
        }else{
            if(even(v)){ v/=2; r*=2;
                         (*cl) += 2;          // added benchmarking code
            }else{
                if(v>=u){ v=(v-u)/2;s+=r;r*=2;
                          (*cl) += 3;       // added benchmarking code
                }else {   u=(u-v)/2;r+=s;s*=2;
                          (*cl) += 3;       // added benchmarking code
                }
            }
        }
        k++;  // count the number of itterations
        (*cl) += 1; // added benchmarking code
    }
    while(k>0){                  // phase ii
        if (even(r)){ r/=2;
                    (*cl) += 1;          // added benchmarking code
        }else{ r=(r+p)/2;
               (*cl) += 1;      // added benchmarking code
        }
        k--;
        (*cl) += 1;          // added benchmarking code
    }
    return p-r;
}
```

Figure 6.3: The Montgomery and Gcd algorithms augmented with profiling instructions

```
for (i=0; i<NUMPRIMES; i++){
  P = PRIMETAB[i];
  for (a=0; a<65536; a++){
    b = (a % (p-1)) + 1;
    res      = montgom_inv (p, b, &cl);
    ..... process the montgomery cycles obtained
    ..... benchmark gcd in a similar fashion
  }
}
```

Figure 6.4: An exemplar of the code used for the profiling process

By inserting code around the control structure of the algorithm, the *cl* variable will be updated based on the execution path taken at run-time. Since the execution

path and the number of iterations are data dependent, we can build an accurate frequency distribution of the amount of clock cycles used for the Montgomery and gcd algorithms. A fragment of the file containing the clock cycle count for each node is provided in figure 6.5.

"P"	"a"	mont0	mont1	gcd	total
31	15	50	35	12	97
31	16	55	40	3	98
31	17	40	35	12	87
31	18	44	35	9	88
31	19	49	35	6	90
31	20	54	40	9	103
31	21	49	39	3	91
31	22	45	40	9	94
31	23	50	35	15	100
31	24	54	40	6	100
31	25	45	44	3	92
31	26	49	40	6	95
31	27	45	35	12	92
31	28	54	40	9	103
31	29	40	35	9	94
31	30	55	55	3	113

Figure 6.5: A fragment of the clock cycle count file produced by the profiling process

In figure 6.5 the first column represents the input values for the P operand, the second column represents the input values for the a operand. The third, fourth and fifth columns represent the clock cycle counts of the mont0, mont1, and gcd nodes respectively and the final column is the total cycles measured for the current input operands provided. Note each row represents the correlated latencies obtained for the provided operand inputs P and a.

A plot of the cycle-count file obtained by the profiling process (see figure 6.4) is shown in figure 6.6. It represents the resulting amount of clock cycles obtained by uniformally distributing the combinations of input values P and a.

The figure basically shows that the latency increases from low, which is the minimum latencies of the critical path of the application given in figure 6.1 to high, which is the maximum latencies of the individual nodes on the critical path.

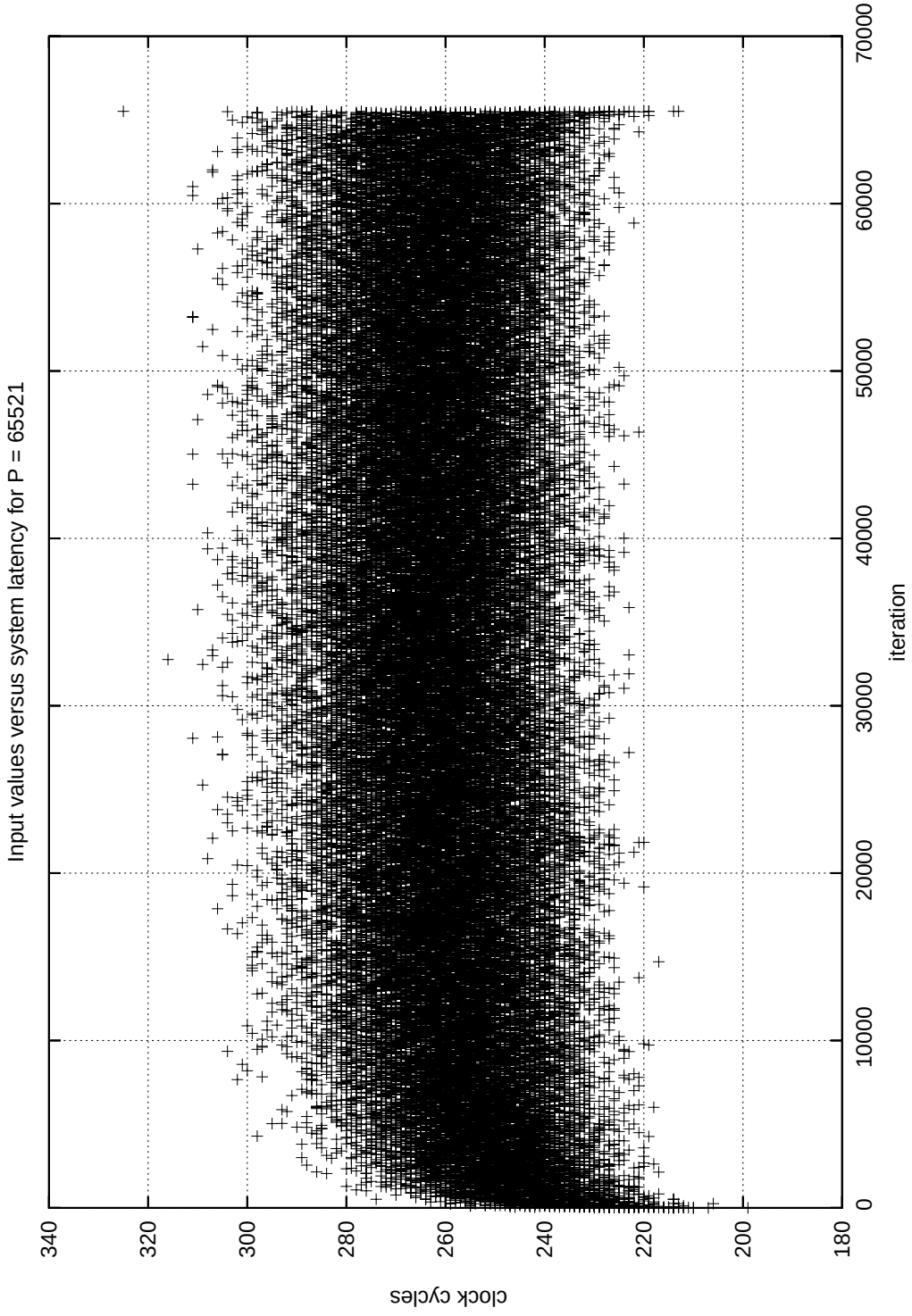


Figure 6.6: *mgcdm* data

Statistics

In this process we obtain statistics from the clock cycle count file of the application; i.e. we obtain CL_{max}, $WL_{max}(m)$ and $WL_{avg}(m)$. CL_{max} is the maximum number of clock cycles taken for a single computation, $WL_{max}(m)$ is the maximum workload in a window of length m, $WL_{avg}(m)$ is the average workload in a window of length m. $WL_{max}(m)$ is used to calculate the required number of processing elements needed. The parameters are calculated as follows:

$$WL_{max}(m) = \max_{j \in [0 \ldots SIZE-m]} (\sum_{i=j}^{j+m-1} CL_i) \tag{6.1.1}$$

$$CL_{max} = WL_{max}(1) = \max_{j \in [0 \ldots SIZE-1]} (CL_i) \tag{6.1.2}$$

$$WL_{avg}(m) = \frac{\sum_{j \in [0 \ldots SIZE-m]} \sum_{i=j}^{j+m-1} \frac{CL_i}{m}}{SIZE - m} \tag{6.1.3}$$

Note: $SIZE$ is the total size of the input data stream.

Calculating the number of required processing elements

The number of processors is calculated as follows:

$$\#procs = \lceil \frac{\frac{1}{m} WL_{max}(m)}{Throughput} \rceil \tag{6.1.4}$$

Which is basically the maximum workload within a window divided by the size of the window divided by the input throughput. Note: this only gives a rough estimate for the required number of processors given a maximum workload within the window. Determining the number of processors required for the final implementation is a design choice. This is described in the next section.

Simulation

In this design step we simulate, tune the design parameters and obtain the system latencies. The input file containing the clock cycle count is randomized by row to obtain an input file for the simulator with an average work load (see figure 6.7). A plot of this randomized input file is shown in figure 6.8. The simulation process is a cycle count measurement based on the randomized input cycle-count file. To check the simulation results a second simulation run can be done using a random input generator which is configured to produce a work load up to $WL_{max}(m)$. The random input generator was designed to stress the application such that the workload for every sliding window of length m is almost $WL_{max}(m)$. This avoids cyclic input values which may mislead the simulator latencies obtained.
This file is first fed into a statistics program which will calculate a number of required parameters as mentioned in the previous section. The simulation parameters obtained are:

Table 6.3: Statistical parameters obtained for simulation

parameter	description
$WL_{max}(m)$	The maximum work load within a window of size m
$WL_{max}(m)/m$	The number of processors needed
CL_{max}	This is equivalent to $WL_{max}(1)/1$
WL_{avg}	This is equivalent to WL_{max} of the total data set

The statistical program also provides the frequency distributions of the application and all its functional nodes. Figure 6.9 shows a plot of the obtained frequency distributions. Since the frequency distributions for the randomized inputs and the uniform inputs are identical only one distribution is shown. In figure 6.9a the frequency distribution of the mont0 and in figure 6.9c the frequency distribution of the mont1 function are given. The distributions are not equivalent which is to be expected. Table 6.4 shows the individual minimum, maximum, and average latencies in clock cycles for the individual function nodes and the total application.

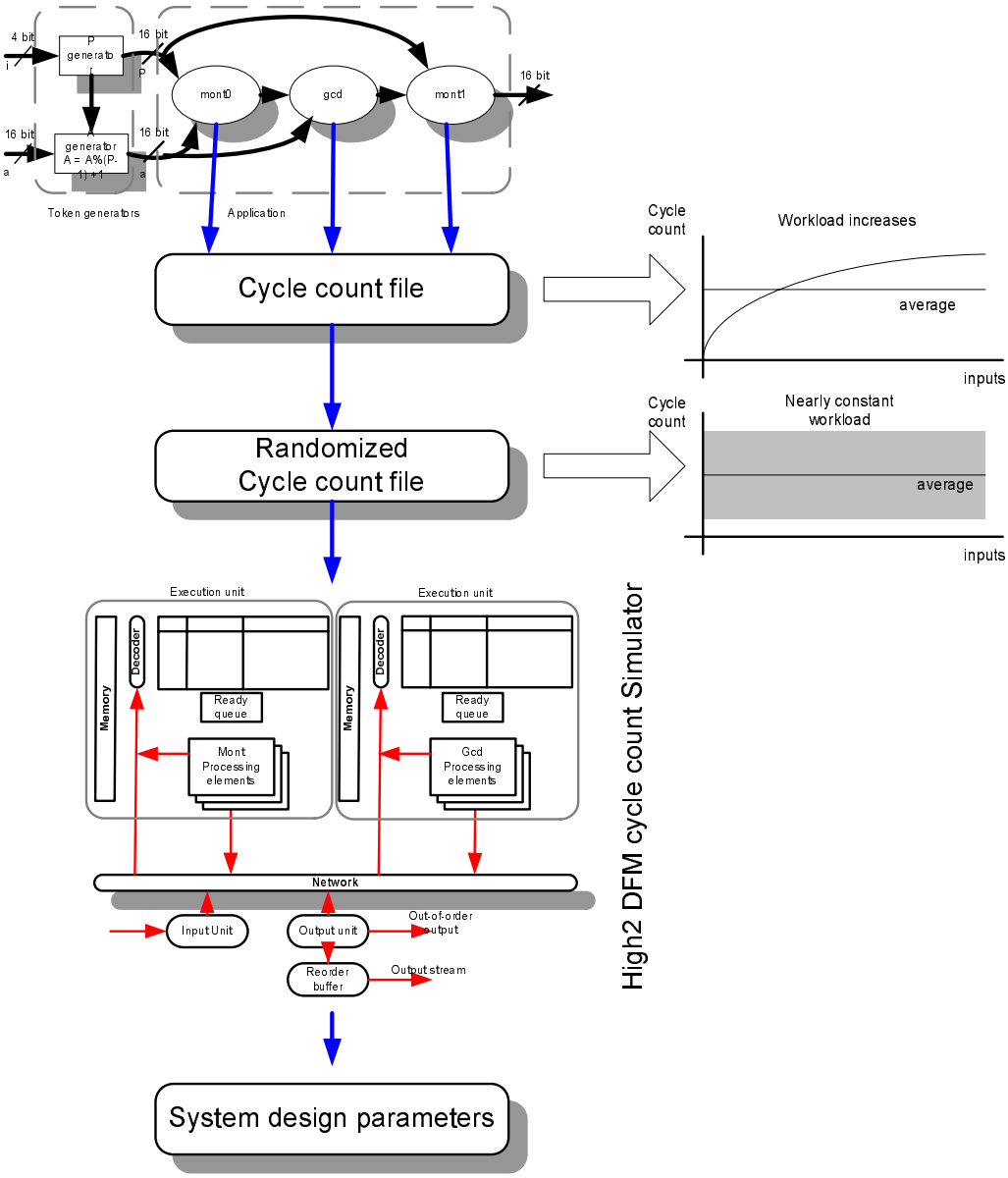


Figure 6.7: *mgcdm* application design flow

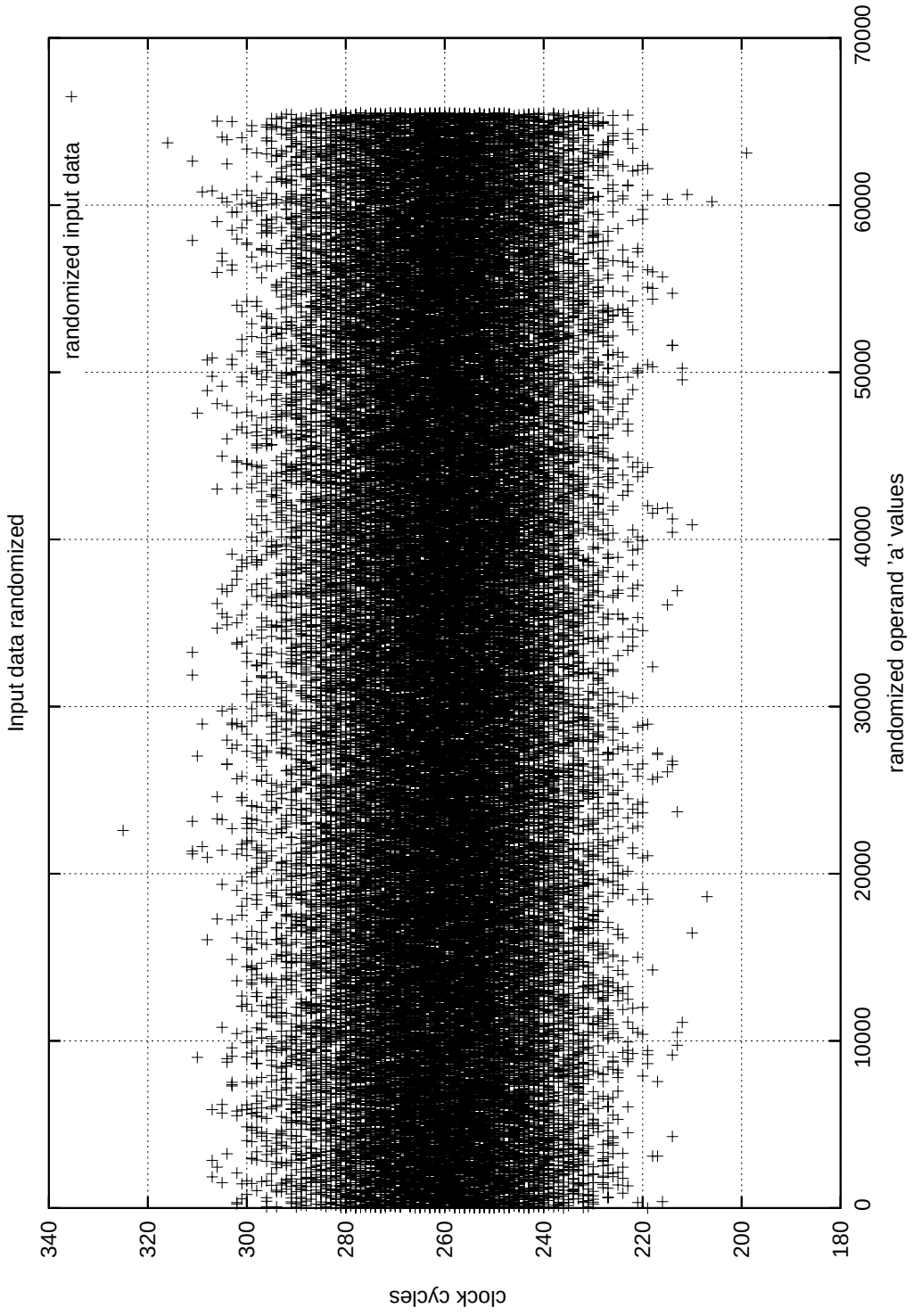


Figure 6.8: *mgcdm* randrow stats

Figure 6.9: *mgcdm* randrow stats

Table 6.4: Min, Max, and Average latencies in clock cycles of the *mgcdm* application

name	Minimum	Maximum	Average
mont0	16	173	73
mont1	16	161	57
gcd	3	60	15
mgcdm	37	325	158

Note that the minimum latency of the total *mgcdm* application is not the sum of the individual minimum latencies. This is due to the fact that the latencies of the individual nodes are correlated.

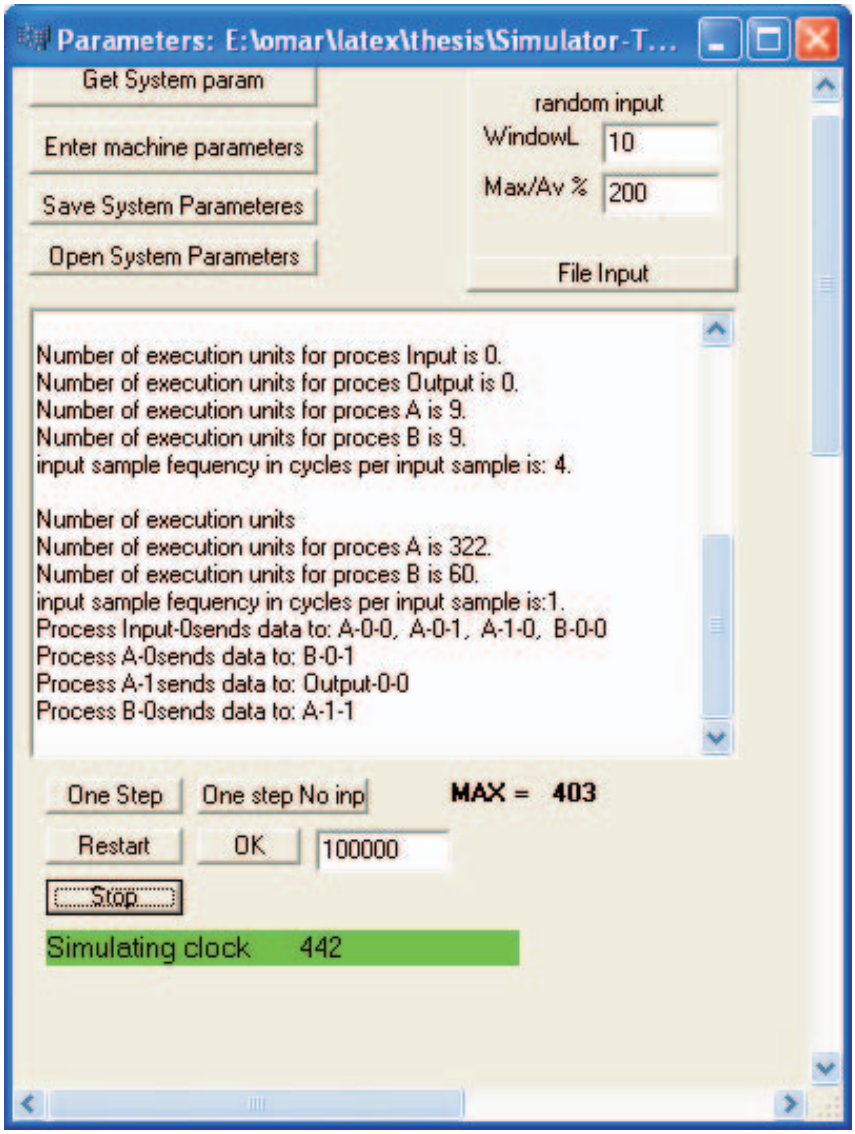


Figure 6.10: GUI of the High^2 DFM simulator

In order to obtain the required number of processing elements for each execution

Mont \ Gcd	16	17	22	30	60
131	813	818	826	827	824
132	416	418	410	405	409
134	341	341	333	332	330
150	325	326	325	325	325
322	325	325	325	325	325

Table 6.5: Number of Montgomery and Gcd processors versus the obtained simulator latency in clock cycles

unit type we simulate with the number of processors given by $WL_{max}(m)/m$ for each processing element type using the High^2 DFM simulator. A screen picture of the GUI is shown in figure 6.10. Figure 6.11 gives a plot of various number of processor combinations possible for different window size m values. The top curve in the plot is for the montgomery node the curve below is for the gcd. We can see the montgomery curve has the value of 320 clock cycles for $m = 1$ which is the correlated maximum number of clock cycles obtained for the mont0 and mont1 nodes of the application graph shown in figure 6.1. On the other hand the gcd function has a CL_{max} value of 60cc. The reason that the CL_{max} value is different from those shown in chapter3 (gcd=69cc), is that the input data set has been influenced by the mont0 function and hence the input operand combinations that lead to more than 60cc do not occur.
The actual latencies obtained from the simulator are shown in table 6.5. In the table we show the number Gcd and Montgomery processors against the total latency obtained by the simulator for a simulation run length of 2^{20} randomized and correlated[1] input data samples. From the table we can deduce that the number of required processors which can handle the input work load must be above 147 (131 mont and 16 gcd) processors for an input data stream with a throughput of 1 sample per clock cycle. If the number of processors is less than 147 the latency will go to infinity as there is more incoming work load than that the system can handle. If we chose the number of processing elements to be 382 (322 mont and 60 gcd) the system will have a minimum latency but would be much more insensitive to input workload fluctuations.

[1]Correlated in the sense that the latencies are obtained from the actual application profiling and not from individual node profiling

poperty	static scheduling	dynamic scheduling
min latency	394 clock cycles	325
max latency	394 clock cycles	827
min nr. processors	334 mont and 60 gcd	mont 131, gcd 16
max nr. processors	334 mont and 60 gcd	mont 322, gcd 60
processor gain	0 %	58.4 % mont and 73.3 % gcd

Table 6.6: Dynamic versus static scheduling results of the *mgcdm* streaming application with a stream throughput of one sample/clock cycle.

In fact it would be capable of handling the maximum possible workload. Reducing the number of processors to 134 mont and 16 gcd processors gives a slightly higher latency but with $1 - \frac{134}{322} * 100 = 58.4\%$ reduction on the number of Montgomery processors and $1 - \frac{16}{60} * 100 = 73.33\%$ reduction on the number of Gcd processors. This implies that if we use the dynamic scheduling approach of the High^2 *DFM* and compare it to the static scheduling approach we gain more than 60% on the number of processors required. If latency is an important design parameter, we can obtain better latency at the cost of more processors. Finally the design is tunable to the input throughput. The figures obtained are for a throughput of one data sample per clock cycle. If we lower the throughput to 1 data sample every 10 clock cycles we can recalculate the number of required processors based on equation 6.1.4. Hence system designers can scale design parameters to meet their application needs.

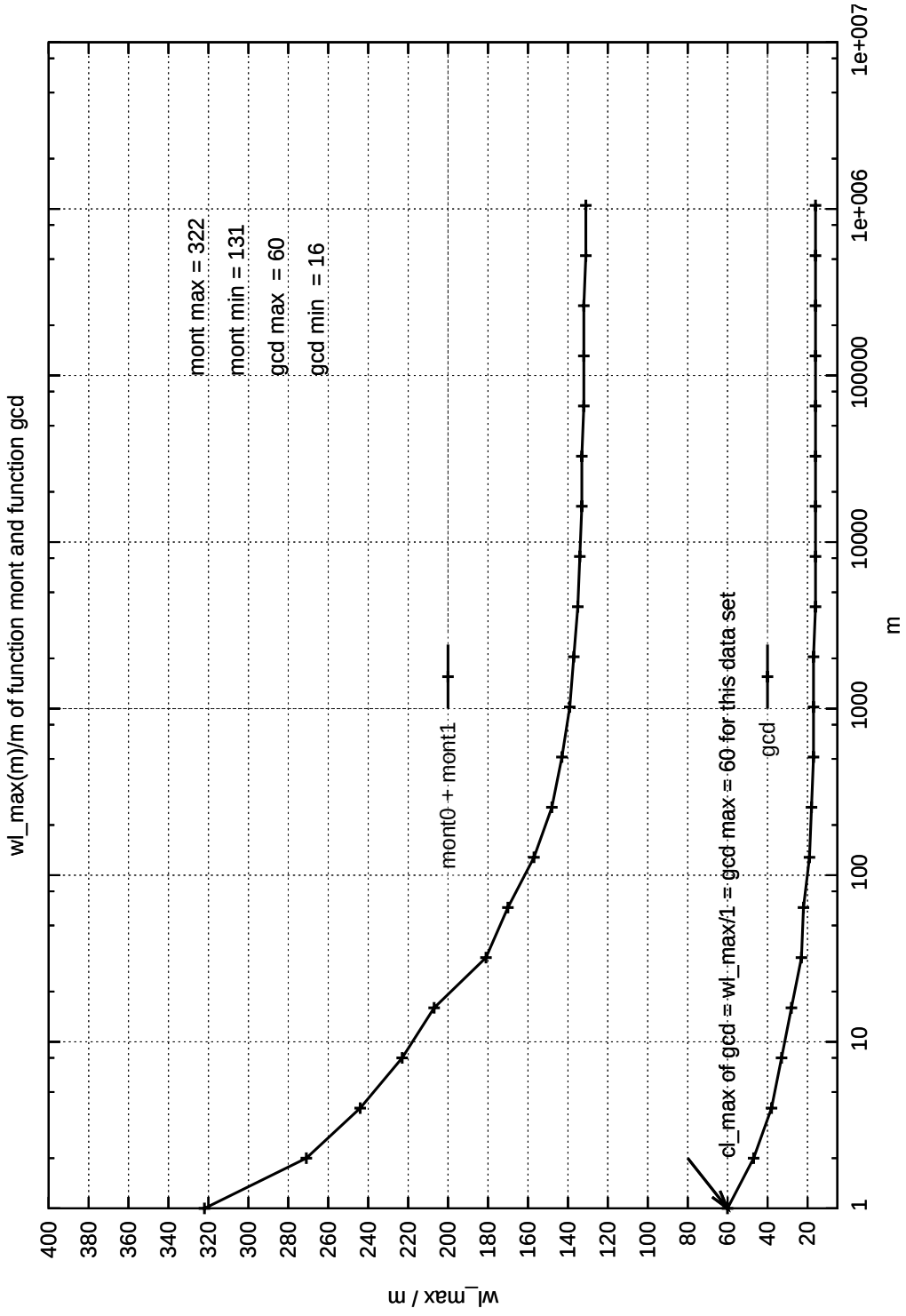


Figure 6.11: *mgcdm* simulator latencies

6.2 Conclusions

In this chapter we described the design of a high-throughput streaming application for the High^2 *DFM* architecture. By profiling the application we were able to determine the number of processors required for the worst case latency and the best case latency scenarios. If we consider the latency, then the gain we obtained is the difference between the average execution latency of the total application and the maximum execution latency of the individual nodes (which is the static scheduling case). It turned out that a latency improvement of $394 - 325 = 69$ clock cycles is possible for the High^2 *DFM* and a reduction of more than 60% of the number of processors.
Since the number of processors is scalable based on the input throughput, the system can be tuned to the required input throughput. Variations in the input throughput do influence the performance of the High^2 *DFM* while the same variations for a statically scheduled architecture may lead to non-optimal scheduling solutions (see chapter 1).

Chapter 7

Conclusions and Future Work

7.1 Non-manifest algorithms

Non manifest algorithms are characterized by having a variable execution latency. The variation in latency may be due to control sequences in the instruction code which eventually will lead to various execution path's or due to input data dependency which may lead to a variation in the number of loop iterations or variation in the control sequences taken. A combination of both is also plausible.

Modeling non-manifest behavior is best done by profiling the non-manifest application on the target processor implementation. Taking the number of loop iterations as a design parameter is not sufficient. It is not general enough to cover all application examples. A better choice for profiling is to use the actual amount of clock cycles consumed.

Important design parameters that are obtained by profiling are the maximum load generated for a single computation CL_{max} and the average load CL_{avg}. In order to obtain accurate results the profiling should be performed using realistic application data.

Obtaining these parameters for complex type of applications (applications with more

than one non-manifest algorithm) is not strait forward, since the frequency distributions of the individual nodes within the application are correlated.

7.2 Dynamic hardware scheduling architectures

Both Tomasulo and scoreboard schedulers are used in superscaler architectures which are considered to be Von Neumann type architectures. Von Neumann type architectures suffer from the so called Von Neumann bottleneck. Their architectures are controlled by a sequencer which executes an instruction stream. Instructions are fine grained and a program counter points to the next instruction to be executed.
The instruction life cycle consists of fetching instructions, executing them and writing the results back to either registers or memory. This is very short when compared to the instruction cycle of dataflow machines. In order to speedup Von Neumann type of architectures the execution of instructions are pipelined.
Theoretically if there are no hazards the speedup up for a w deep pipelined architecture is w. Unfortunately hazards do occur and they reduce the speedup. Some of the pipeline hazards can be solved (statically) at compile time by aligning, reordering and inserting dummy *nop* instructions.
Unfortunately this does not solve all control hazards.
The scoreboard and Tomasulo schedulers try to solve those hazards dynamically such as mentioned in 4.3.2 and 4.3.3.
The differences between the scorebaording and Tomasulo is that scoreboarding will allow instructions to run in parallel if there are no dependencies amongst them and there are no structural hazards. Tomasulo on the other hand, tries to achieve more ILP by distinguishing between true instruction dependencies and dependencies due to compile time register allocation.
Out-of-order execution, *out-of-order* completion and register renaming are among the modifications added to the Tomasulo scheduler in order to improve ILP. Even with these sophisticated scheduling mechanisms, we have to look at other concepts such as task level parallelism or coarse grained parallelism, if more parallelism is to be obtained from an algorithmic specification. Since the instruction stream itself is the

bottle neck.

The data flow model of execution does not suffer from false dependencies. Since there are no registers. The operands of an instruction (node) travel through the edges of the data flow graph from node to node. An instruction is considered executable when the processing element implementing the instruction and operands of the instruction are available. Hence the data flow model is hindered only by the true dependencies available within the application. There is no restriction on the instruction granularity within the data flow model although lessons learned from previous research suggest that instructions with fine granularity lead to more overhead and an imbalance in the computation to communication ratio.

Despite the problems classical data flow machines had to face, the dataflow model of execution has attractive properties for high-throughput streaming applications and hence is a motivation for the High^2 dataflow architecture.

7.3 High^2 Data Flow Machine

High throughput streaming applications come in two categories. The simple model applications and the complex model applications.

Simple model applications consist of a single non-manifest algorithm. By having multiple processing elements (each implementing the same algorithm) running in parallel on independent stream operands, an architectural solution that is capable of handling various stream loads under various throughput conditions, is provided. Design parameters for such an application are obtained by simulation. The latency which is one of those parameters is also verified by an analytical upper bound.

Complex model applications are composed of multiple interacting algorithms which operate on a high-throughput stream of operands.

A possible solution for the complex model applications is the High^2 Data Flow Machine (*DFM*). Which is a coarse grained data flow machine for high-throughput

streaming non-manifest applications. It is derived from the classical data flow architecture and its scheduling is done dynamically in hardware.
The High^2 *DFM* was designed with streaming applications in mind, this means that throughput is of major importance, the *DFM* can be tuned at design time to meet certain throughput requirements, this is achieved by varying the number of processing elements available for computing the (non-manifest) functions of the application.

7.4 Example of High^2 *DFM* Applications

The design flow of the High^2 *DFM* consists of profiling the application and determining the required design parameters. Since the number of processors is scalable based on the input throughput, the system could be tuned to the required input throughput. Variations in the input throughput do influence the performance of the High^2 *DFM* while the same variations for a statically scheduled architecture may lead to non-optimal scheduling solutions.

7.5 Conclusions

It is useful to implement high-throughput non-manifest applications on Coarse grained data flow machines. If the application contains only manifest operations, dedicated processor architectures which are based on static scheduling would have the advantage of having less control overhead and hence a smaller design space. On the other hand if the application contained non-manifest algorithms and the variation, in execution time, between the average case and the worst case is meaningful. Architectures that use dynamic scheduling of their processing elements can benefit from this variation.

7.6 Future work

The work presented in this thesis did not cover the problems of transporting operands between execution units. Processing elements of a many execution units can produce their outputs at the same time and hence the network should be able to handle that.

Appendix A

Scoreboard rules

```
1  switch(state){
2    case ISSUE:
3      while (Busy[inst->fu] || result[inst->destination]){/* wait until */};
4      Busy[inst->fu] = true;
5      Operation[inst->fu] = inst->operation;
6      Fi[inst->fu] = inst->destination;
7      Fj[inst->fu] = inst->S1;
8      Fk[inst->fu] = inst->S2;
9      Qj[inst->fu] = Result[inst->S1];
10     Qk[inst->fu] = Result[inst->S2];
11     Rj[inst->fu] = (Qj[inst->fu] == "") ? "YES" | "NO";
12     Rk[inst->fu] = (Qk[inst->fu] == "") ? "YES" | "NO";
13     Result[inst->destination] = inst->fu;
14     break;
15
16   case READ_OPERANDS:
17     while ((Rj[inst->fu] == false) || (Rj[inst->fu] == false)){/* wait until */};
18     Rj[inst->fu] = "NO";
19     Rk[inst->fu] = "NO";
20     Qj[inst->fu] = 0;
21     Qk[inst->fu] = 0;
22     break;
23
24   case EXECUTION_COMPLETE:
25     while (inst->fu.ready == false){/* wait until */};
26     break;
27
28   case WRITE_RESULT:
29     result = true;
30     while(result){
31       result = true;
32       for (f=0; f<FU.size; f++){
33         if ((Fj[f] != Fi[inst->fu] || Rj[f] == "NO") && (Fk[f] != Fi[inst->fu] || Rk[f] == "NO") ){
34           result = false;
35         }
36       }
37     }
38     for (f=0; f<FU.size; f++){
39       if (Qj[f] = inst->fu) Rj[f] = "YES";
40       if (Qk[f] = inst->fu) Rk[f] = "YES";
41     }
42     Result[Fi[inst->fu]] = "";
43     Busy[inst->fu] = "NO";
44     break;
45 }
```

Figure A.1: The scoreboard book keeping rules for each state of an instruction and the actions taken to allow the instructions to advance from one state to the other

Appendix B

Tomasulo rules

Figure B.1 provides the book keeping rules for the Tomasulo scheduling algorithm. Note that instructions are always issued if there is a free reservation station (in the case of floating point instructions), or there is an empty buffer location (in the case of load/store) instructions. Results of an operation are written as soon as the CDB is free. For the issuing instruction, rd is the destination, rs and rt are the source register numbers. imm is the sign-extended immediate field, and r is the reservation station or buffer that the instruction is assigned to. RS is the reservation stations data structure. The value returned a FP unit or by the load unit is called a result. RegisterStat is the register status data structure, the register file is REGS[].

```
switch(state){
  case ISSUE:
    swith(operation_type){
    case FP_OPERATION:
        while (/* station r not empty */){};
        if(RegisterStat[rs].Qi != 0){
            RS[r].Qj = RegisterStart[rs.Qi];
        }else{
            RS[r].Vj = regs[rs];
            RS[r].Qj = 0;
        }
        if(RegisterStat[rt].Qi != 0){
            RS[r].Qk = RegisterStart[rt.Qi];
        }else{
            RS[r].Vk = regs[rt];
            RS[r].Qk = 0;
        }
        RS[Busy] = "YES";
        RegisterStat[rd].Qi = r;
        break;
    case LOAD_OR_STORE:
        while (/* buffer r not empty */){};
        break;
    case LOAD_ONLY:
        RegisterStat[rt].Qi = r;
        break;
    case STORE_ONLY:
        if (RegisterStat[rt].Qi != 0){
            RS[r].Qk = RegisterStat[rs].Qi;
        }else{
            RS[r].Vk = Regs[rt];
            RS[r].Qk = 0;
        }
        break;
    }
    break;
  case EXECUTE:
    swith(operation_type){
    case FP_OPERATION:
        while (/* (RS[r].Qj != 0) || (RS[r.]Qk != 0) */){};
        { compute result: operands are in Vj and Vk }
        break;
    case LOAD_OR_STORE:
        // step 1
        while (/* (RS[r].Qj != 0) || ({r is not head of load/store queue}) */){};
        RS[r].A = RS[r].Vj + RS[r].A;
        // strep 2
        {Read from Mem[RS[r].A]};
        break;
    }
    break;
  case WRITE_RESULT:
    switch(operation_type){
    case FP_OPERATION:
    case LOAD:
        while (/* (execution not complete at r) || (CDB not available) */){};
        for (x in all register numbers){
            if (RegisterStat[x].Qi) = r){
                Regs[x] = result;
                RegisterStat[x].Qi = 0;
            }
            if (RS[x].Qj = r){
                RS[x].Vj = result; RS[x].Qj = 0;
            }
            if (RS[x].Qk = r){
                RS[x].Vk = result; RS[x].Qk = 0;
            }
        }
        RS[r].Busy = "NO";
        break;
    case STORE:
        while (/* (execution not complete at r) || (Rs[r].Qk == 0) */){};
        Mem[RS[r].A] = rs[r].Vk;
        RS[r].Busy = "NO";
        break;
    }
    break;
}
```

Figure B.1: The Tomasulo scheduling rules

Appendix C

VHDL code of the simple model architecture

C.0.1 Implementation of the processing element

In this appendix we describe how to implement the processing element as a Mealy model. The specification of the processing element is as follows: The processing element will behave as a non-manifest loop and at the same time, the number of loop computations must be controlled by the input data. Hence, if the scheduler provides the data value v where $v \in [1 \ldots CL_{max}]$ the processing element will consume the amount v iterations (clock cycles) during its computation. The result of the computation is always the same value v provided as its input.

This allows us to test the scheduler under various load conditions by providing an input stream with known computation load. Table C.1 shows the specification of the required processing element.

In figure C.1 the unfolded version and in figure C.2 the folded version of a generic processing element are shown. The design process starts by unfolding the specification in time, and in order to design the processing element as a Mealy model, the path from input to output, for a single iteration computation, must be within the same time unit. In other words providing an input to the processing element at time t_i where the processing element would provide its output at time t_{i+1} is not allowed.

Table C.1: Specification of the processing element

control	action
wr & -	$data_i = dataIn$ $count_i = dataIn$ $active = dataIn > 1$ $dataOut = dataIn$
$\overline{wr}$&$active$	$data_i = data_{i-1}$ $count_i = count_{i-1} - 1$ $active = count_{i-1} > 1$ $dataOut = data_{i-1}$
$\overline{wr}$&$\overline{active}$	$data_i = data_{i-1}$ $count_i = count_{i-1}$ $dataOut = data_{i-1}$

The second step is to fold the processing element in time. Registers will be added to data signals which cross the time line. Finally we give names to all the internal signals and write the *VHDL* description of the processing element based on the folded version of the processing element shown in figure C.2.
In the *VHDL* implementation of the processing element we use the data type *DataTp* this is declared in the package type shown in figure C.4. The package mainly contains the needed data structures and some functions used by the scheduler.

C.0.2 Implementation of the scheduler

The scheduler's *VHDL* code is shown in figure C.5. It consists of a memory array which is used to store the data, an allocation table, for keeping track the computations resultant memory address.
The scheduler basically performs the following tasks in a repetitive manner.

- write the result data on to the output stream
- store the valid data from the processing elements to their appropriate memory locations
- allocate waiting data to free processing elements

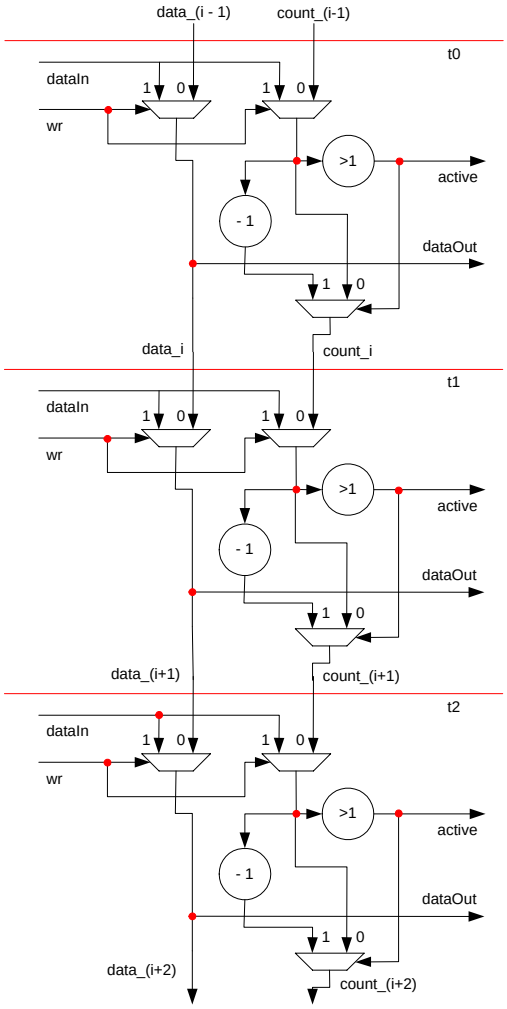


Figure C.1: Generic processing element unfolded in time

The scheduler configuration is given in figure C.6. It basically describes how the connections between the scheduler and the processing elements are established.

C.0.3 Experimental results

The system described was simulated using the simulation tool ModelSim from Model technology [43] and synthesized for a 0.5μ technology using the *LeonardoSpectrum* synthesis engine from Exemplar Logic.

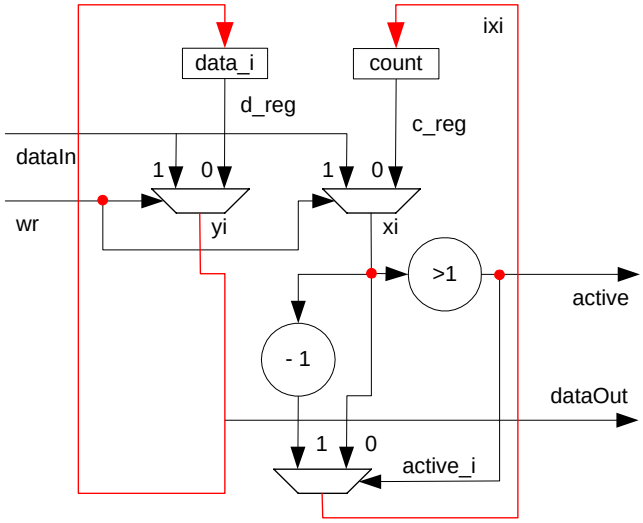


Figure C.2: Folded version of the generic processing element

```
1  ENTITY res IS
2    PORT(clk       : in  std_logic;
3         reset     : in  std_logic;
4         wr        : in  std_logic;
5         active    : out boolean;
6         DataIn    : in  DataTp;
7         DataOut   : out DataTp);
8  END res;
9
10
11 architecture behavior of res is
12 signal d_reg     : DataTp;
13 signal c_reg     : DataTp;
14 signal xi        : DataTp;
15 signal yi        : DataTp;
16 signal active_i  : boolean;
17 signal ixi       : DataTp;
18 begin
19   process (clk, reset)
20   begin
21     if rising_edge(clk) then
22         c_reg <= ixi;
23         d_reg <= yi;
24     end if;
25   end process;
26   xi <= dataIn when wr='1' else c_reg;
27   yi <= dataIn when wr='1' else d_reg;
28   active_i <=  (xi > 1);
29   dataOut <= yi;
30   ixi <= xi - 1 when active_i = true else xi;
31   active <= active_i;
32 end behavior;
```

Figure C.3: *VHDL* code of a generic processing element

The results of the synthesis process for multiple processing elements are shown in table C.2.

```
1  PACKAGE types IS
2    CONSTANT Nres : positive := 3; -- number of resources
3    CONSTANT M    : positive := 14; -- Latency
4
5    --- Data dependent properties
6    CONSTANT maxint : integer := 2**4-1;
7    SUBTYPE int0tomax IS integer RANGE 0 TO maxint;
8    TYPE DataTpIn IS ARRAY(0 TO 1) OF int0tomax;
9    SUBTYPE DataTpOut IS int0tomax;
10
11   -- Memory for storing data
12   TYPE Storage IS RECORD
13     inp  : DataTpIn;
14     outp : DataTpOut;
15   END RECORD;
16   TYPE MemoryTp IS ARRAY(0 TO M-1) OF Storage;
17
18   -- Type used for data transport between resources and scheduler
19   TYPE DataVecTpIn  IS ARRAY (0 TO Nres-1) OF DataTpIn;
20   TYPE DataVecTpOut IS ARRAY (0 TO Nres-1) OF DataTpOut;
21   TYPE IndexVecTp   IS ARRAY (0 TO Nres-1)
22                        OF integer RANGE 0 TO M;
23   TYPE DataSchedResTpIn IS RECORD
24      Data  : DataTpIn;
25      wr    : std_logic;
26   END RECORD;
27   TYPE DataSchedResVecTpIn IS ARRAY(0 TO Nres-1)
28                        OF DataSchedResTpIn;
29
30   TYPE DataSchedResTpOut IS RECORD
31      Data   : DataTpOut;
32      active : boolean;
33   END RECORD;
34
35   TYPE DataSchedResVecTpOut IS ARRAY(0 TO Nres-1)
36                        OF DataSchedResTpOut;
37
38   -- synthesisable functions.
39   FUNCTION incr_mod(i,max : IN integer) RETURN integer;
40   FUNCTION my_mod(a, b: IN integer) RETURN integer;
41 END types;
42
43 PACKAGE BODY types IS
44   FUNCTION incr_mod(i,max : IN integer) RETURN integer IS
45     VARIABLE res : integer;
46   BEGIN
47     IF i < max-1 THEN
48       res:=i+1;
49     ELSE
50       res:=0;
51     END IF;
52     RETURN res;
53   END incr_mod;
54
55   FUNCTION my_mod(a, b: IN integer) RETURN integer IS
56   BEGIN
57     IF a-b < 0 THEN
58         RETURN a;
59     ELSE
60         RETURN (a - ((a/b) * b)); -- b is a poower of 2
61   END my_mod;
62 END types;
```

Figure C.4: The data and bus types

The synthesis results show that the simple model architecture is indeed realizable for Asic design (depending on the required number of processing elements). The implementation scales with the number of processing elements as was expected. Hence there is a physical upper bound on the allowable number of processing elements. Also the obtained frequency of the implementation scales with the number of processing

```
ENTITY scheduler IS
  PORT (clk        : IN std_logic;
        reset      : IN std_logic;
        data       : IN DataTpIn;
        Sched2Res  : OUT DataSchedResVecTpIn;
        Res2Sched  : IN DataSchedResVecTpOut;
        result     : OUT DataTpOut
        );
END scheduler;

ARCHITECTURE behaviour OF scheduler IS
  SIGNAL Sched2Res_int : DataSchedResVecTpIn;
  SIGNAL free, store : std_logic_vector(0 TO Nres-1);
BEGIN
  PROCESS(Res2Sched,Sched2Res_int,free)
  BEGIN
    store <= (OTHERS=>'0');
    FOR i IN Res2Sched'RANGE LOOP
      IF NOT Res2Sched(i).active THEN
        store(i)<=Sched2Res_int(i).wr OR NOT free(i);
      END IF;
    END LOOP;
  END PROCESS;

  PROCESS(clk, reset)
    VARIABLE first, last : integer RANGE 0 TO M-1;
    VARIABLE memory : MemoryTp;
    VARIABLE last_handled : boolean;
    TYPE LutTp IS ARRAY (0 TO Nres-1) OF integer RANGE 0 TO M-1;
    -- stores the index in memory of the result.
    VARIABLE Lut : LutTp;
  BEGIN
    IF reset='1' THEN
      last:=0; first:=0;
      Sched2Res_int <= (OTHERS => ((0,0),'0'));
      free <= (OTHERS => '1');
      last_handled:=false;
      lut := (OTHERS => 0);
    ELSIF rising_edge(clk) THEN
    result <= memory(first).d1;
    -- store valid data from resources to memory
      FOR i IN 0 TO Nres-1 LOOP
        IF store(i)='1' THEN
          -- memory.d1 is used for both input and output data
          memory(Lut(i)).d1:=Res2Sched(i).Data;
          free(i)<='1';
        END IF;
      END LOOP;
      -- assign data to resources
      Sched2Res_int <= (OTHERS => ((0,0),'0'));
      memory(first):=data;
      last_handled:=false;
      FOR i IN free'RANGE LOOP
        IF (free(i)='1' OR store(i)='1') AND NOT last_handled THEN
          -- If resource is free ('0') then it can be used again.
          -- But if store is '1' then the resource has just
          -- finished a job and can allocated to a new one.
          Sched2Res_int(i) <= (memory(last),'1');
          Lut(i) := last;
          free(i)<='0';
          last_handled:=last=first;
          last := incr_mod(last, M);
        END IF;
      END LOOP;
      first := incr_mod(first, M);
    END IF;
  END PROCESS;
  Sched2Res <= Sched2Res_int;
END behaviour;
```

Figure C.5: A processing element independent scheduler implementation

elements. The reason that the frequency decreases with the number of processing elements is that the scheduling algorithm loops over all processing elements checking

```
ENTITY system IS
  PORT (clk     : IN std_logic;
        reset   : IN std_logic;
        data    : IN DataTpIn;
        result  : OUT DataTpOut);
END system;

ARCHITECTURE structure OF system IS
  component scheduler IS
    PORT (clk        : IN std_logic;
          reset      : IN std_logic;
          data       : IN DataTpIn;
          Sched2Res  : OUT DataSchedResVecTpIn;
          Res2Sched  : IN DataSchedResVecTpOut;
          result     : OUT DataTpOut
          );
  END component;
  component resource is
    port(DataIn   : DataSchedResTpIn;
         DataOut  : out DataSchedResTpOut;
         clk      : std_logic;
         reset    : std_logic);
  end component;
  SIGNAL Sched2Res : DataSchedResVecTpIn;
  SIGNAL Res2Sched : DataSchedResVecTpOut;
BEGIN
  sched:scheduler
      PORT MAP (clk,reset,data,Sched2Res,Res2Sched,result);
  resources:FOR i IN Sched2Res'RANGE GENERATE
  inst:resource
      PORT MAP (Sched2Res(i),Res2Sched(i),clk,reset);
  END GENERATE;
END structure;
```

Figure C.6: The scheduler configuration

Table C.2: Synthesis results

$\#PE's$	Frequency (Mhz)	#Gates	Asic technology
1	185.5	3132	0.5 μ
2	166	4447	0.5 μ
4	85.2	7179	0.5 μ
8	52.8	13793	0.5 μ
16	27.2	26088	0.5 μ
32	13.7	54791	0.5 μ
64	6.9	111870	0.5 μ

if they have to write their results to memory. This loop is a sequential implementation and results in a large combinatorial path within the hardware implementation and hence a longer clock period. Better results can be obtained (at the cost of extra busses and multi ported memory) if we parallelize the implementation of this loop.

Having multiple processing elements does indeed allow the system to achieve a higher stream throughput, but it introduces an extra implementation problem, which is that

```
entity resource is
  port(DataIn  : DataSchedResTpIn;
       DataOut : out DataSchedResTpOut;
       clk     : std_logic;
       reset   : std_logic);
end resource;

architecture test of resource is
  alias x : integer is DataIn.Data.d1;
  alias y : integer is DataIn.Data.d2;
  alias wr : std_logic is DataIn.wr;
  alias z : integer is DataOut.data;
  alias active : boolean is DataOut.active;
  signal active_i : boolean;
  signal xreg,yreg,ixreg,iyreg,zi,xi,yi : int0tomax;
  function module(x,y : integer) return integer is
  begin
    if y=0 then
      report "right_operand_0" severity note;
      return x;
    else
      return x mod y; -- for simulation
      -- return my_mod(x,y); -- for synthesis
    end if;
  end module;
begin
  process(clk,reset)
  begin
    if rising_edge(clk) then
      if active_i then
        xreg <= ixreg;
        yreg <= iyreg;
      end if;
    end if;
  end process;
  xi <= x when wr='1' else xreg;
  yi <= y when wr='1' else yreg;
  zi <= module(yi,xi);
  ixreg <= zi;
  iyreg <= xi;
  active_i <= not(zi=0);
  active <= active_i;
  z <= xi;
end test;
```

Figure C.7: The gcd(x,y)processing element implementation

of having to simultaneously write the results of multiple processing elements to memory.

The simulation results in figure C.8 show the waveform produced of the scheduler incombination with three generic processing elements. From the simulation set we can see that the active signal of a processing element directly follows the *wr* signal (This can be seen at clock value $150ns$) and that consecutive writes to the same processing element are handled by the system without any extra delays. Which indicates that we do not lose extra clock cycles for writing the data to and form the processing element.

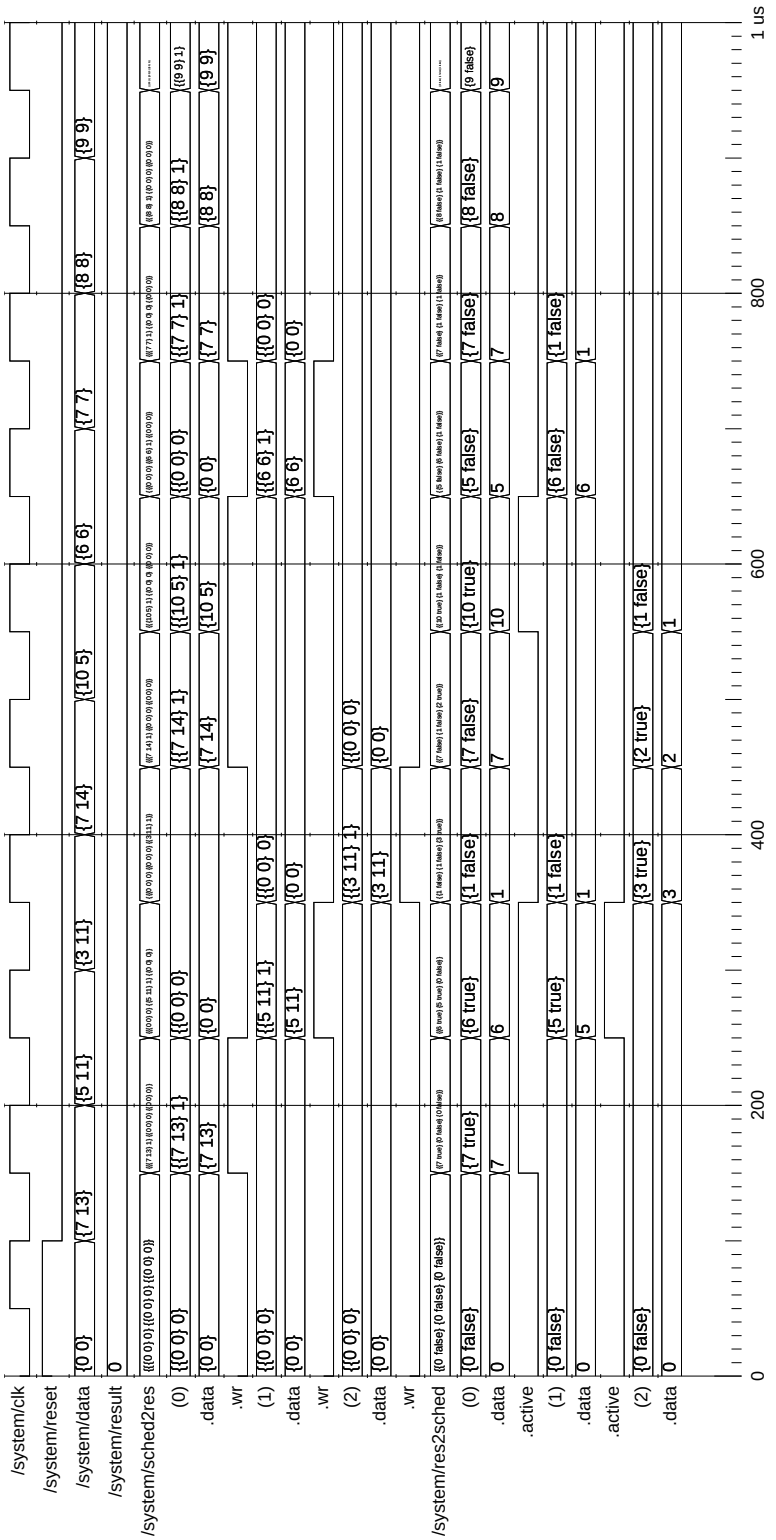


Figure C.8: Simulationresults

FILE

Bibliography

[1] O.Mansour, S.Etalle, T.Krol, "Scheduling and Allocation of Non-Manifest Loops on Hardware Graph Models", PROGRESS Proceedings, 2001, ISBN 90-73461-26-x.

[2] Omar Mansour, Egbert Molenkamp, Thijs Krol, "The synthesis of a hardware scheduler for Non-Manifest Loops", EUROMICRO Symposium on digital system design, Dortmund, Germany 4-6 September 2002, ISBN 0-7695-1790-0.

[3] Omar Mansour, Egbert Molenkamp, Thijs Krol, "Minimum waste scheduling of dynamic variable-latency and non-manifest functional-units", PROGRESS Proceedings, 2002, ISBN 90-73461-34-0.

[4] W. Verhaegh, "Multidimensional Periodic Scheduling", Ph.D Thesis, University of Eindhoven, The Netherlands, 1995, ISBN 90-74445-21-7.

[5] Wim Verhaegh, Gertjan Arnoldussen, Kees Goossens, and Marc Heijligers, "Phideo: architectural synthesis for high-throughput digital signal processing", in Philips Research Bulletin on IC Design, No. 31, 1997, pp. 9-11.

[6] R.P. Kleihorst, A. van der Werf, W.H.A. Brls, W.F.J. Verhaegh, and E. Waterlander, "MPEG2 video encoding in consumer electronics", in Journal of VLSI Signal Processing, vol. 17, 1997, pp. 241-253.

[7] P.Lippens, B. De Loore, G. de Haan, P. Eeckhout, H. Huijgen, A. Lning, B. McSweeney, M. Verstraelen, B. Pham, and J. Kettenis, "A video signal processor for motion-compensated field-rate upconversion in consumer television" in IEEE Journal of Solid-State Circuits, vol. 31, 1996, pp. 1762-1769.

[8] W.F.J. Verhaegh, P.E.R. Lippens, E.H.L. Aarts, J.L. van Meerbergen, and A. van der Werf, "Multidimensional periodic scheduling: Model and complexity", in Proceedings of the Euro-Par, vol. II, 1996, pp. 226-235.

[9] W.F.J. Verhaegh, "Multidimensional periodic scheduling", Ph.D. thesis, Eindhoven University of Technology, Eindhoven, the Netherlands, 1995.

[10] P.E.R. Lippens, P.E.R., J.L. van Meerbergen, W.F.J. Verhaegh, D.M. Grant, and A. van der Werf , "Design of a 30 MHz, 32/16/8-points DCT processor with Phideo", in Proceedings of VLSI Signal Processing VII, 1994, pp. 24-32.

[11] N.G.Busa, A. van der Werf, M.Bekooij "Scheduling Coarse-Grain Operations for VLIW Processors", Philips Research Laboratories, 13th International Symposium on System Synthesis (ISSS'00), September 20-22-2000, Madrid Spain, ACM Press, ISBN:1080-1082.

[12] Henk Corporaal, "Microprocessor Architectures from VLIW to TTA", John Wiley & Sons Ltd, 1997, ISBN 0 471 97157 X

[13] Wikipedia the Free Encyclopedia, http://en.wikipedia.org/wiki/Transport_Triggered_Architectures

[14] Henk Corporaal, "Transport Triggered Architectures Design and Evaluation", Thesis Technische Universiteit Delft, 1995, ISBN 90-9008662-5

[15] Huibert Kwakernaak, Raphael Sivan "Modern Signals and Systems", Prentice Hall, 1991, ISBN: 0-13-809252-4.

[16] Grant R. Griffin "CORDIC FAQ", Dsp, Guru, http://www.dspguru.com/info/faqs/cordic2.htm.

[17] Silvia M. Muller, "On the Scheduling of Variable Latency Functional Units", 11th ACM Symposium on Parallel Algorithms and Architectures SPAA'99.

[18] S.M. Muller, H.W. Leister, P.Dell, N.Gerteis, D.Kroening, "The Impact of Hardware Scheduling Mechanisms on the Performance and Cost of Processor Designs", University of Saarland, Germany, 15th GI/ITG Conference ARCS'99.

[19] S.Rathnam and G.Slavenburg, "Processing the new world of interactive media. The Trimedia VLIW CPU architecture", IEEE Signal Process Mag vol. 15, no. 2 P108117, March 1998

[20] Daniel Kroening, Silvia M. Muller, Wolfgang J.Paul "A Rigorous Correctness Proof of a Tomasulo Scheduler Supporting Precise Interrupts".

[21] Daniel Kroening, "Design and evaluation of a RISC processor with a Tomasulo scheduler", Universitat des Saarlandes, januar 1999.

[22] R.M. Tomasulo, "An efficient algorithm for exploiting multiple arithmetic units", In IBM Journal of Research an development, volume 11 (1), pages 25-33, IBM 1967.

[23] Vijay Raghunathan, Srivaths Ravi, Ganesh Lakshminarayana, "Integrating Variable-Latency Components into High-Level Synthesis", IEEE Transactions on computer-aided design of integrated circuits and systems, October 2000.

[24] L. Benini, E.Macii, M.Poncino, and G.De Micheli, "Telescopic units: A new Paradigm for performance optimization of VLSI designs" IEEE, Trans. Computer-Aided Design of Integrated Circuits and Systems, vol. 19, No. 10, October 2000.

[25] R. Camposano, D. Knapp, D. MacMillen, "A Review of Hardware Synthesis Techniques", NATO Advanced Study Institute, Tremezzo (I), June 1995.

[26] D. Gajski, N.Dutt, A. Wu, S. Lin, "High-level synthesis: Introduction to chip and system design", Kluwer, ISBN 0-7923-9194-2, 1992.

[27] M. Morris Mano, Charles R. Kime, "Logic and computer Design Fundamentals, Third Edition", Prentice Hall, ISBN 0-13-191165-1, 2004

[28] Subbarao Palacharla, "Complexity effective superscalar processors", Phd dissertation, university of wisconsin-madison, 1998.

[29] David A. Patterson, John L. Hennessy, "Computer Organization and Design, the hardware / Software interface", second edition, Morgan Kaufmann Publishers, ISBN 1-55860-428-6.

[30] John L. Hennessy, David A. Patterson, "Computer architecture a quantative approach", third edition, Morgan Kaufmann Publishers, ISBN 1-55860-596-7.

[31] http://www.scd.ucar.edu/computers/gallery/cdc/6600.html.

[32] Wayne L. Winston, "Operations Research Applications and Algorithms", thisrd edition, Duxbury, ISBN 0-534-20971-8.

[33] G. Papadopoulos and D. Culler, "Monsoon: an explicit token-store architecture." In Proceedings of the 17th Annual International Symposium on Computer Architecture, pages 28–31, May 1990.

[34] J. Hicks, D. Chiou, B. S. Ang, and Arvind. "Performance Studies of Id on the Monsoon Dataflow System." Journal of Parallel and Distributing Computing, 18:273–300, 1993.

[35] B. Lee and A. R. Hurson. Dataflow architectures and multithreading. Computer, 27(8):27–39, August 1994.

[36] Kenneth J. Breeding. "Digital Design Fundamentals", Prentice Hall, 1989, ISBN 0-13-211830-0.

[37] V.G. Grafe and J.E.Hoch, "The Epsilon2 Multiprocessor system", J.Parallel and Distributed Computing, vol. 10 1990 pp. 309-318.

[38] J. R. Gurd, C. C. Kirkham, and I. Watson, "The Manchester prototype dataflow computer," Communications of the ACM, vol. 28, no. 1, 1985.

[39] Arthur H. Veen, "Data Flow Architecture", ACM Computing Surveys, Vol 18, No. 4, December 1986.

[40] Mayan Moudgill and Keshav Pingali and Stamatis Vassiliadis. Register renaming and dynamic speculation: an alternative approach. In Proceedings of MICRO-26, pages 202–213, March 1993.

[41] E.Savaş, Ç. Koç, "The Montgomery Modular Inverse- Revisited". IEEE transactions on computers, vol. 49 No 7, July 2000.

[42] http://www.intel.com/design/pentium4/manuals.

[43] ModelSim from Model technology, www.model.com.

[44] Paul M. Heysters, "Coarse-Grained Reconfigurable Processors Flexibility Meets Efficiency", Ph.D Thesis University of Twente, sep 2004,ISBN 90-365-2076.

[45] S.W.Smith, "The Scientist and Engineers's Guide to Digital Signal Processing", second edition, california Technical Publishing, San Diego, CA USA, 1999, ISBN 0-9660176-6-8.

[46] B.S.Kaliski, "The Montgomery Inverse and Its Applications" IEEE Trans. Computers, vol. 44, no. 8, pp. 1,064-1,065, Aug. 1995.