NoGap: Novel Generator of Accelerators and Processors

Per Axel Karlström

Linköping, 2010
NoGap: Novel Generator of Accelerators and Processors
Per Axel Karlström
Linköping studies in science and technology, Dissertations, No. 1347

Copyright © 2010 Per Axel Karlström (unless otherwise noted)
ISBN: 978-91-7393-293-6
ISSN: 0345-7524
Printed by LiU-Tryck, Linköping 2010

Front and back cover: Unification. Per Axel Karlström
Symbolizes the unification of design processes offered by NoGap and how it bridges a design space gap. The image is loosely bases on the complexities seen in Mase graphs.

URL for online version:
http://urn.kb.se/resolve?urn=urn:nbn:se:liu:diva-60192
Errata lists will also be published at this location if necessary.

Parts of this thesis is reprinted with permission from IET and IEEE
The following notice applies to material which is copyrighted by IEEE:
This material is posted here with permission of the IEEE. Such permission of the IEEE does not in any way imply IEEE endorsement of any of Linköping universitet’s products or services. Internal or personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purposes or for creating new collective works for resale or redistribution must be obtained from the IEEE by writing to pubs-permissions@ieee.org. By choosing to view this material, you agree to all provisions of the copyright laws protecting it.
Abstract

Application Specific Instruction-set Processors (ASIPs) are needed to handle the future demand of flexible yet high performance embedded computing. The flexibility of ASIPs makes them preferable over fixed function Application Specific Integrated Circuits (ASICs). Also, a well designed ASIP, has a power consumption comparable to ASICs. However the cost associated with ASIP design is a limiting factor for a more wide spread adoption. A number of different tools have been proposed, promising to ease this design process. However all of the current state of the art tools limits the designer due to a template based design process. It blocks design freedoms and limits the I/O bandwidth of the template. We have therefore proposed the Novel Generator of Accelerators And Processors (NoGap). NoGap is a design automation tool for ASIP and accelerator design that puts very few limits on what can be designed, yet NoGap gives support by automating much of the tedious and error prone tasks associated with ASIP design.

This thesis will present NoGap and much of its key concepts. Such as; the NoGap Common Language (NoGapCL) which is a language used to implement processors in NoGap. This thesis exposes NoGap’s key technologies, which include automatic bus and wire sizing, instruction decoder and pipeline management, how Program Counter (PC)-Finite State Machines (FSMs) can be generated, how an assembler can be gen-
erated, and how cycle accurate simulators can be generated.

We have so far proven NoGap’s strengths in three extensive case studies, in one a floating point pipelined data path was designed, in another a simple Reduced Instruction Set Computing (RISC) processor was designed, and finally one advanced RISC style Digital Signal Processor (DSP) was designed using NoGap. All these case studies points to the same conclusion, that NoGap speeds up development time, clarify complex pipeline architectures, retains design flexibility, and most importantly does not incur much performance penalty compared to hand optimized Register Transfer Language (RTL) code.

We believe that the work presented in this thesis shows that NoGap, using our proposed novel approach to micro architecture design, can have a significant impact on both academic and industrial hardware design. To our best knowledge NoGap is the first system that has demonstrated that a template free processor construction framework can be developed and generate high performance hardware solutions.
Populärvetenskaplig
Sammanfattning


de gör lite olika saker beroende på vilken bilmodell som skall tillverkas. Robotar lämnar sedan över resultatet till nästa maskin, som kan vara vilken annan maskin som helst i fabriken.

Att bygga fabriken blir ett komplicerat problem om vi skall kunna tillverka många olika bilmodeller. Vi måste givetvis se till att maskinerna användes så mycket som möjligt, så att det går att tillverka så många bilar som möjligt per dag. Detta innebär att när en maskin är klar med en uppgift, bör den snarast starta med en ny uppgift, detta leder till att flera tillverkningsordrar är igång samtidigt i fabriken. Men vi kan inte på förhand veta i vilken ordning ordrarna kommer att komma. Detta innebär att ett styrsystemet måste konstrueras som kan ta hänsyn till om det blir konflikt om vissa maskiner.

Det som skiljer en processor från en bilfabrik är att i processorn tillverkas beräkningsresultat istället för bilar och istället för t.ex. lackerings- och plåtböjarmaskiner, har processorn olika beräkningsenheter. Många av de problem som uppstår när en processor skall konstrueras, liknar de som beskrevs med bilfabriken, d.v.s. vilken maskin som skall göra vad och när och var resultatet skall hänvisas till.

Det vore givetvis enklare om vi i fallet med bilfabriken skulle kunna bestämma hur varje bilmodell skall tillverkas, en i taget, i stället för att försöka klura ut hur systemet skall se ut med hänsyn till alla olika bilmodeller som skall tillverkas. Det samma gäller för processorer. NoGap låter oss göra just det för processorer, varje instruktion beskrivs en i taget, datorn tar sedan hand om att sätta in rätt beräkningsenheter på rätt plats och ser till att alla ordrar till beräkningsenheterna kommer med rätt fördröjning.

Det måste dock sägas att det finns verktyg som redan utlovar att göra processortillverkning lättare genom att låta en konstruktör beskriva vad varje order (instruktion) skall göra. Problemet med de verktyg som redan existerar är dock att de utgår i från mallar vilket gör det svårt att bygga nydanande och effektiva processorer som verkligen är skräddarsydda för

Att det blir enklare att bygga processorer gör att nya processorer blir billigare och kan produceras fortare. Att det dessutom går att skräddarsy processorena för deras arbetsuppgift gör att de kan utföra mer beräkningar på kortare tid eller med mindre energi. Detta innebär i sin tur att du t.ex. kan få mer kraftfulla mobiltelefoner till ett lägre pris och med längre batteritider.
Abbreviations,
Explanations, and
Definitions

When you work with somebody for a long period of time, you develop a shorthand with everything.

---

Abbreviations

ALU Arithmetic Logic Unit
API Application Programmer Interface
ASIC Application Specific Integrated Circuit
ASIP Application Specific Instruction-set Processor
AST Abstract Syntax Tree
BBP Base Band Processor

ADL Architecture Description Language
AGU Address Generation Unit
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Full Form</th>
</tr>
</thead>
<tbody>
<tr>
<td>BGL</td>
<td>Boost Graph Library</td>
</tr>
<tr>
<td>BIST</td>
<td>Built In Self Test</td>
</tr>
<tr>
<td>BNF</td>
<td>Backus-Naur Form</td>
</tr>
<tr>
<td>Castle</td>
<td>Control Architecture STucture LanguagE</td>
</tr>
<tr>
<td>CISC</td>
<td>Complex Instruction Set Computing</td>
</tr>
<tr>
<td>CLI</td>
<td>Command Line Interface</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DCS</td>
<td>Dynamic Clause Selection</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DPTP</td>
<td>Data Path Transformation Path</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>$\mathcal{E}$</td>
<td>Graph edge</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FF</td>
<td>Flip-Flop</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language</td>
</tr>
<tr>
<td>HW</td>
<td>Hardware</td>
</tr>
<tr>
<td>IEEE</td>
<td>Institute of Electrical and Electronics Engineers</td>
</tr>
<tr>
<td>IIR</td>
<td>Infinite Impulse Response</td>
</tr>
</tbody>
</table>
OL_{Intra}  Intra-Operation Loop
OL_{Inter}  Inter-Operation Loop
IP         Intellectual Property
IR         Intermediate Representation
ISA        Instruction Set Architecture
Joust      Judgment and Operation Unified Structure
LHS        Left Hand Side
LSB        Least Significant Bit
LUT        Look-Up Table
MAC        Multiply And Accumulate
Mage       Micro Architecture Generation Essentials
E_{Mage}   Mage Edge
V_{Mage}   Mage Vertex
Mase       Micro Architecture Structure Expression
E_{Mase}   Mase Edge
V_{Mase}   Mase Vertex
MIPS       Mega Instructions Per Second
MSB        Most Significant Bit
NRE        Non-Recurring Engineering
NoGap^{CD} NoGap Common Description
NoGap^{CL} NoGap Common Language
NoGap^{FUD} NoGap Functional Unit Description
NoGap      Novel Generator of Accelerators And Processors
NOP        No Operation
PCB        Printed Circuit Board
PC          Program Counter
PC-FSM     Program Counter-Finite State Machine
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PID</td>
<td>Pipelined Instruction Driven</td>
</tr>
<tr>
<td>PU</td>
<td>Parse Unit</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>RHS</td>
<td>Right Hand Side</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Language</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction Multiple Data</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Issue Multiple Tasks</td>
</tr>
<tr>
<td>SPTP</td>
<td>Sequence Path Transformation Path</td>
</tr>
<tr>
<td>SSP</td>
<td>Source Sink Pass</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>TIE</td>
<td>Tensilica Instruction Extension</td>
</tr>
<tr>
<td>TTM</td>
<td>Time To Market</td>
</tr>
<tr>
<td>TUI</td>
<td>Textual User Interface</td>
</tr>
<tr>
<td>UI</td>
<td>User Interface</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modeling Language</td>
</tr>
<tr>
<td>VHDLC</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very-High-Speed Integrated Circuit</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Large Instruction Word</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>WPM</td>
<td>Watts Per MIPS</td>
</tr>
</tbody>
</table>
## Definitions and Explanations

<table>
<thead>
<tr>
<th>Term</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASCII-binary</td>
<td>ASCII representation of binary code.</td>
</tr>
<tr>
<td>accelerator</td>
<td>A device that efficiently can perform a specific type of raw computations, but usually does not have any flow control logic.</td>
</tr>
<tr>
<td>AsmGen</td>
<td>Part of ( \text{NoGap} ) that uses ( \text{NoGap} ) Common Description (( \text{NoGap}^{CD} )) to generate ( \text{NoGap} \text{Asm} ).</td>
</tr>
<tr>
<td>assembly instruction</td>
<td>Assembly code representing one binary instruction.</td>
</tr>
<tr>
<td>assembly program</td>
<td>Assembly code written by a user for a processor to perform a task.</td>
</tr>
<tr>
<td>capability</td>
<td>How much something can do, both in terms of Mega Instructions Per Second (MIPS) and the number of tasks it can perform.</td>
</tr>
<tr>
<td>child scope</td>
<td>The scope following a construct.</td>
</tr>
<tr>
<td>class</td>
<td>In the context of C++, a collection of variables and functions.</td>
</tr>
<tr>
<td>construct statement</td>
<td>A complex ( \text{NoGap}^{CL} ) construction, e.g. an entire if-then-else code section.</td>
</tr>
<tr>
<td>data path</td>
<td>Hardware consisting of a number of FUs that can perform computations.</td>
</tr>
<tr>
<td>decoder</td>
<td>FU that translates an incoming instruction into all needed control signals for a data path.</td>
</tr>
<tr>
<td>device</td>
<td>A hardware system that can perform some function.</td>
</tr>
<tr>
<td><strong>dot</strong></td>
<td>Part of the graphviz tool suite for generating images of graphs from a textual description.</td>
</tr>
<tr>
<td><strong>E-descriptor</strong></td>
<td>A pointer to a Graph edge ($E$) in a Boost Graph Library (BGL) graph.</td>
</tr>
<tr>
<td><strong>energy</strong></td>
<td>A thermodynamic quantity equivalent to the capacity of a physical system to do work, measured in Joules (J).</td>
</tr>
<tr>
<td><strong>function</strong></td>
<td>In the context of C++ code, it refers to a subroutine.</td>
</tr>
<tr>
<td><strong>FU cluster</strong></td>
<td>A arrangement of vertices and edges in a Micro Architecture Structure Expression (Mase) graph representing an Functional Unit (FU) with input, output, and inout ports.</td>
</tr>
<tr>
<td><strong>instruction</strong></td>
<td>An array of bits, usually in a coded form, that determines what some hardware shall do.</td>
</tr>
<tr>
<td><strong>introduced symbol</strong></td>
<td>A unique symbol, created and added to a Mase Edge ($E_{\text{Mase}}$) during automatic port and bus sizing.</td>
</tr>
<tr>
<td><strong>method</strong></td>
<td>In the context of C++ code, a method refers to a function that is part of a class.</td>
</tr>
<tr>
<td><strong>NoGapAsm</strong></td>
<td>The generated executable, which controls the parser, that converts an assembly program to a binary file for the processor.</td>
</tr>
<tr>
<td><strong>object</strong></td>
<td>In the context of C++, a unique instantiation of a class.</td>
</tr>
<tr>
<td><strong>power</strong></td>
<td>Rate of energy conversion, measured in Watts (W).</td>
</tr>
</tbody>
</table>
price  Monetary cost of something usually measured in some kind of currency.

$S(\alpha)$  size in bits of signal $\alpha$.

source $\mathcal{V}$  In a directed graph, the Graph vertex ($\mathcal{V}$) from which an $\mathcal{E}$ starts. Or if used about a $\mathcal{V}$, the source $\mathcal{V}$s are all $\mathcal{V}$s that has an $\mathcal{E}$ going into the $\mathcal{V}$ in question.

SSP partition  A partition of a $\text{Mage}$ graph into source, sink, and pass $\mathcal{V}$s.

symbol table  A table containing all symbols in a PU

$\mathcal{V}$-descriptor  A pointer to a $\mathcal{V}$ in a BGL graph.

---

Typographical Conventions

The following typographic conventions are used in the text:

- \texttt{for(;;)} represent program code in text.
- \texttt{add} instructions.
- $\langle \text{nt} \rangle$ Non terminal in a BNF grammar.
BNF Grammars

Backus–Naur Form (BNF) grammars are used in various places in the text. An example of how a grammar will look is shown in the grammar below.

BNF grammar example
⟨non_terminal⟩→terminal1 terminal2 | (non_terminal2) □
⟨non_terminal2⟩→terminal3 □

The following conventions are used:

(non_term) non terminal

terminal terminal

→ Separates the right hand side of a rule from the left hand side.

| Separates alternatives in a rule.

(space) Separates members in an alternative.

■ Marks the end of a rule.

Code Listings

Longer sections of code are presented in listings like the one below.

1 if(a == b | c <= b)
2 {
3    sig = 3;
4 }
Acknowledgments

Bernard of Chartres used to say that we are like dwarfs on the shoulders of giants, so that we can see more than they, and things at a greater distance, not by virtue of any sharpness of sight on our part, or any physical distinction, but because we are carried high and raised up by their giant size. If I have seen longer than anybody else is because I have been standing on the shoulders of giants.

John of Salisbury

I have a lot of people to thank for being able to produce this thesis. It is the culmination of about five years work with the NoGap project.

First of all I want to thank Dake Liu, my professor and supervisor. I want to thank him for letting me work in his team and all the support he has given me. But most importantly I want to thank him for that he believed in me and my research in times when I did not even do so myself.

I want to thank Wenbiao Zhou, my valued colleague in the NoGap project, for being part of this project for the last two years. His tireless work and support has been invaluable for the continued development of this work.

I want to thank Andreas Ehliar, Johan Eilert, and Di Wu for all the intriguing discussions we have had and for the support and good cooperation in everything from FPGA synthesis to domino programs.
I want to thank all other persons in the department for their valuable support and interesting discussions.

I want to thank Carl Blumenthal, Lyonel Barthe, Faisal Akhlaq, Sumathi Loganathan, Ching-han Wang and Luis Medina Valdes, all very competent master thesis students, who have contributed with invaluable insights and development work.

I want to thank Wei Zhang, my beloved partner, who has supported and stood by my side for the last three years. I would not be here without you.

I want to thank Sten Karlström and Agneta Karlström von Göes, my parents. They have always believed in me and taught me that I can do anything if only I try hard enough.

I want to thank Lisa Stensdotter and Magnus Göes Karlström, my siblings. I would not be the person I am without them.

I want to thank P.G. von Göes, my late grandfather, who made sure mom and dad bought a computer, since he thought it would be an important thing to learn about.

I want to thank Martin Jansson for being like a big brother to me, often helping me to pave my way in life. I also want to thank Kurt, Kerstin, and Henrik Jansson, they have been like a second family for me. And also for letting me use their 286 computer when having computers at home was more of an exception than rule.

I want to thank all my friends for having believed in me and helped me to think about other things than just my work.

I want to thank Sofia Karlström, one of my cousins, she motivated me to start my Ph.D. Partly because I promised her, on her dissertation party, that I would have one myself in about 10 years or so.

I want to thank the Swedish research council for their generous funding.

Finally I want to thank all, known and unknown, people who have contributed to GCC, C++, Make, Emacs, Linux, Boost, STL, Inkscape,
GIMP, $\LaTeX$, ModelSim, SystemVerilog, Subversion, the x86 architecture, and all other peoples work I have directly or indirectly used in my research.

All you people are my giants and its only thanks to you I have been able to see a little bit further than anybody else.
Preface

In the beginning the Universe was created. This has made a lot of people very angry and been widely regarded as a bad move.

Douglas Adams

This thesis presents my research from May 2004 to November 2010. This thesis is written with the hopes that it might serve as a reference for anyone interested in NoGap and its further development.

The work presented in this thesis outlines the status of NoGap at the time of writing, depending on when you read this thesis, certain things might have changed.

The research behind NoGap started from scratch. Dake Liu and I were the sole responsible for this work in the first years. Starting from scratch meant that much time had to be spent setting up goals, doing background investigations, and making the basic system design. But much time has also been devoted to the actual C++ implementation, system design improvements, and verification/validation of processor and tool chain generation.

Other research interests

During my time as a Ph.D. student I have also taken part in developing an advanced computer system course, where an FPGA is used to host a
small computer system. Some of my research time has also been spent on efficient processor architectures for video and general DSP tasks.
Publications

1. High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4 [17]
   P. Karlström, A. Ehliar, D. Liu

2. High Performance, Low Latency FPGA based Floating Point Adder and Multiplier Units in a Virtex 4 [18]
   P. Karlström, A. Ehliar, D. Liu

   A. Ehliar, P. Karlström, D. Liu

   P. Karlström and D. Liu

5. NoGapCL: A Flexible Common Language for Processor Hardware Description [48]
   W. Zhou, P. Karlström, D. Liu

6. Operation Classification for Control Path Synthetization with NoGap [23]
   P. Karlström, W. Zhou and D. Liu

7. Automatic Assembler Generator for NoGap [20]
   P. Karlström, S. Loganathan, F. Akhlaq and D. Liu

8. Automatic Port and Bus Sizing in NoGap [21]
   P. Karlström, W. Zhou and D. Liu

9. Cycle Accurate Simulator Generator for NoGap [16]
   P. Karlström, F. Akhlaq, S. Loganathan, W. Zhou and D. Liu

    P. Karlström, W. Zhou, C.-h. Wang and D. Liu
11. Implementation of a Floating Point Adder and Subtractor in NoGap, a Comparative Case Study [22] (Accepted, not yet published) P. Karlström, W. Zhou and D. Liu
Contents

I  Prologue 1

1  Assumed Reader Knowledge 3
   1.1  C++ 4
   1.2  Verilog/VHDL 4
   1.3  Hardware Design 4
   1.4  Processor Design 4
   1.5  Graphs 5
      1.5.1  Graphs in \texttt{NoGap} 5

II  Background 7

2  Introduction 9
   2.1  Rationale 10
   2.2  Introduction to \texttt{NoGap} 15
   2.3  Thesis Organization 17

3  Related Work 19
   3.1  LISA 20
      3.1.1  Strengths of LISA 20
      3.1.2  Weaknesses of LISA 20
      3.1.3  LISA in Comparison with \texttt{NoGap} 20
<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2</td>
<td>MESCAL/Tipi 20</td>
<td></td>
</tr>
<tr>
<td>3.2.1</td>
<td>Strengths of MESCAL/Tipi 21</td>
<td></td>
</tr>
<tr>
<td>3.2.2</td>
<td>Weaknesses of MESCAL/Tipi 21</td>
<td></td>
</tr>
<tr>
<td>3.2.3</td>
<td>MESCAL/Tipi in Comparison with NoGap 21</td>
<td></td>
</tr>
<tr>
<td>3.3</td>
<td>EXPRESSION 22</td>
<td></td>
</tr>
<tr>
<td>3.3.1</td>
<td>Strengths of EXPRESSION 22</td>
<td></td>
</tr>
<tr>
<td>3.3.2</td>
<td>Weaknesses of EXPRESSION 22</td>
<td></td>
</tr>
<tr>
<td>3.3.3</td>
<td>EXPRESSION in Comparison with NoGap 22</td>
<td></td>
</tr>
<tr>
<td>3.4</td>
<td>ArchC 22</td>
<td></td>
</tr>
<tr>
<td>3.4.1</td>
<td>Strengths of ArchC 23</td>
<td></td>
</tr>
<tr>
<td>3.4.2</td>
<td>Weaknesses of ArchC 23</td>
<td></td>
</tr>
<tr>
<td>3.4.3</td>
<td>ArchC in Comparison with NoGap 23</td>
<td></td>
</tr>
<tr>
<td>3.5</td>
<td>nML 23</td>
<td></td>
</tr>
<tr>
<td>3.5.1</td>
<td>Strengths of nML 23</td>
<td></td>
</tr>
<tr>
<td>3.5.2</td>
<td>Weaknesses of nML 24</td>
<td></td>
</tr>
<tr>
<td>3.5.3</td>
<td>nML in Comparison with NoGap 24</td>
<td></td>
</tr>
<tr>
<td>3.6</td>
<td>SimpleScalar 24</td>
<td></td>
</tr>
<tr>
<td>3.6.1</td>
<td>Strengths of SimpleScalar 24</td>
<td></td>
</tr>
<tr>
<td>3.6.2</td>
<td>Weaknesses of SimpleScalar 25</td>
<td></td>
</tr>
<tr>
<td>3.6.3</td>
<td>SimpleScalar in Comparison with NoGap 25</td>
<td></td>
</tr>
<tr>
<td>3.7</td>
<td>MIMOLA 25</td>
<td></td>
</tr>
<tr>
<td>3.7.1</td>
<td>Strengths of MIMOLA 25</td>
<td></td>
</tr>
<tr>
<td>3.7.2</td>
<td>Weaknesses of MIMOLA 26</td>
<td></td>
</tr>
<tr>
<td>3.7.3</td>
<td>MIMOLA in Comparison with NoGap 26</td>
<td></td>
</tr>
<tr>
<td>3.8</td>
<td>MDES 26</td>
<td></td>
</tr>
<tr>
<td>3.8.1</td>
<td>Strengths of MDES 26</td>
<td></td>
</tr>
<tr>
<td>3.8.2</td>
<td>Weaknesses of MDES 27</td>
<td></td>
</tr>
<tr>
<td>3.8.3</td>
<td>MDES in Comparison with NoGap 27</td>
<td></td>
</tr>
<tr>
<td>3.9</td>
<td>ASIP Meister 27</td>
<td></td>
</tr>
<tr>
<td>3.9.1</td>
<td>Strengths of ASIP Meister 27</td>
<td></td>
</tr>
<tr>
<td>3.9.2</td>
<td>Weaknesses of ASIP Meister 27</td>
<td></td>
</tr>
</tbody>
</table>
3.9.3 ASIP Meister in Comparison with NoGap . . . . . 27

III NoGap Internals 29

4 System Architecture 31
  4.1 Architecture Overview . . . . . . . . . . . . . . . . 31
  4.2 Action Register . . . . . . . . . . . . . . . . . . . . 34
  4.3 The Joust architecture . . . . . . . . . . . . . . . . 36

5 Parse Unit 39
  5.1 Parse Units . . . . . . . . . . . . . . . . . . . . . . 39
  5.2 PC-FSM . . . . . . . . . . . . . . . . . . . . . . . . 42
  5.3 Mage Dependency Graph . . . . . . . . . . . . . . . 42
  5.4 Clause Access Rules . . . . . . . . . . . . . . . . . 42
  5.5 Pipelines . . . . . . . . . . . . . . . . . . . . . . . 44
  5.6 Control Path . . . . . . . . . . . . . . . . . . . . . 46
    5.6.1 Operation Info . . . . . . . . . . . . . . . . . 46
  5.7 Operation Classes . . . . . . . . . . . . . . . . . . 48
  5.8 Instruction Decoder . . . . . . . . . . . . . . . . . 50
  5.9 Instruction Table . . . . . . . . . . . . . . . . . . . 52
    5.9.1 Instruction Format . . . . . . . . . . . . . . . 52
  5.10 Needs Clock Input . . . . . . . . . . . . . . . . . . 53
  5.11 SSP Partition . . . . . . . . . . . . . . . . . . . . 53

6 Mage 59
  6.1 Introduction . . . . . . . . . . . . . . . . . . . . . . 59
  6.2 $\mathcal{V}_{\text{Mage}}$ Types . . . . . . . . . . . . . . 62
  6.3 $\mathcal{E}_{\text{Mage}}$ Types . . . . . . . . . . . . . . . 65
  6.4 Mage Dependency Graph . . . . . . . . . . . . . . . 65
Contents

7 Mase 69
7.1 Introduction . . . . . . . . . . . . . . . . . . . . . . . . . . 69
7.2 V\textsubscript{Mase} Types . . . . . . . . . . . . . . . . . . . . . 71
7.3 \(\varepsilon\textsubscript{Mase}\) Types . . . . . . . . . . . . . . . . . . . . . 75
7.4 FU Cluster . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 75
7.5 In-line Expression FU . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 78
7.6 Loops in the Mase . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
7.7 Predefined Mase Transformations . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 79
7.7.1 Combine Equal Edges . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 80
7.7.2 Insert Flip-Flops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.7.3 Combine Equal Flip-Flops . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 81
7.7.4 Insert Multiplexers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.7.5 Set Wire Names . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.7.6 Set Edge Sizes . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 85
7.8 Connect Stalling FFs . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 88

8 Symbol Table 89
8.1 Symbol Table Description . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 89
8.2 Symbol Creation Process . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 92

9 \textit{N}o\textit{G}ap Common Language 97
9.1 Introduction to NoGap\textit{CL} . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 97
9.2 Operators . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 100
9.3 Identifiers . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
9.4 Concatenation . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 105
9.5 Keywords and Special Characters . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 106
9.6 FU Specification . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 107
9.7 Control Structures . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
9.7.1 if / else if / else Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 109
9.7.2 switch Structure . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 110
9.8 Cycle and Comb Blocks . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 111
Contents

9.9 Clause Name ................................................. 111
9.10 Port and Signal Declarations .............................. 112
  9.10.1 Port Timing Offset .................................... 114
9.11 Manual FU Instantiation ............................... 115
9.12 Mase FU Specification ..................................... 116
9.13 Phase Description ....................................... 117
9.14 Stage Description ....................................... 117
  9.14.1 Stall Operation Description ......................... 118
9.15 Pipeline Description ..................................... 119
9.16 FU Instantiation in Mase FUs .......................... 122
9.17 Pipelined Operations .................................... 123
  9.17.1 Operation Code Assignment ......................... 126
9.18 FU Usage List ............................................ 127
9.19 Inline FU Expressions .................................. 131
  9.19.1 Port Sizing of Inlined FUs ......................... 132
9.20 Forwarding Path Description ......................... 132
9.21 Static Connections ..................................... 133
9.22 Flow Control ............................................ 137
9.23 Design Template ....................................... 138
  9.23.1 PC-FSM Template .................................... 139
  9.23.2 Instruction Decoder Template ....................... 143
9.24 Instruction Declarations .............................. 145
9.25 NoGap System Call ..................................... 147

10 Dynamic Bus and Port Sizing .......................... 149
  10.1 Introduction to Dynamic Port Sizing ............... 149
  10.2 The Sizing Algorithm ................................. 151
    10.2.1 Hardware Multiplexing Multiplexers ............. 153
    10.2.2 Annotation Phase ................................. 153
    10.2.3 Solver Phase ..................................... 157
11 Pipeliner Generation

11.1 Controlling Mage FUs ........................................... 165
11.2 Dynamic Clause Selection ...................................... 166
11.3 Data Paths and Control Paths ................................. 166
11.4 Instruction Format Generation ................................. 167
11.5 Decoder Generation .............................................. 167
11.6 Pipeliner Generation ............................................. 168
    11.6.1 Operation Classification ................................. 170
    11.6.2 Pipeline Usage Vector ................................... 172

IV Spawners

12 Generators and Spawners ...................................... 177

12.1 Introduction .................................................. 177
12.2 Generator System .............................................. 178
12.3 Generator Register ............................................ 183
12.4 Generator Usage ............................................... 184
12.5 The Flow File .................................................. 185
12.6 The Generated Data Template ................................. 186
12.7 NoGap Folders ................................................. 186

13 SystemVerilog Spawner ....................................... 189

13.1 Introduction .................................................. 189
13.2 PC-FSM Verilog Generation .................................... 190
13.3 Mage Verilog Generation ...................................... 191
    13.3.1 Preparatory Mage Transformations ...................... 191
    13.3.2 Verilog Generation of Transformed Mage ............ 191
13.4 Mage Verilog Generation ...................................... 196
13.5 Instruction Decoder Generation ............................. 198
17.3 Implementation with Joust approach ..... 234
17.4 Verification ............................................. 237
17.5 Results .................................................... 238
   17.5.1 ASIC Flow ........................................... 239

18 SENIOR ................................................. 241
   18.1 Introduction ........................................... 241
   18.2 Architecture ........................................... 242
   18.3 Instruction Set ........................................ 242
      18.3.1 Move-Load-Store Instructions ................. 246
      18.3.2 Short ALU Instructions ......................... 246
      18.3.3 Long Arithmetic Instructions ................. 247
      18.3.4 Flow Control Instructions .................... 247
   18.4 NoGap CL for SENIOR ................................ 247
      18.4.1 Mage in SENIOR .................................. 247
      18.4.2 Mase in SENIOR .................................. 247
   18.5 Implementation Details ............................... 252
      18.5.1 FPGA Flow ....................................... 252
      18.5.2 ASIC Flow ....................................... 253

VI Conclusions and Future Work ................................ 257

19 Conclusions ............................................. 259

20 Future Work ............................................. 261
   20.1 Instruction Level Parallelism ....................... 261
   20.2 Instruction Format Specification .................. 262
   20.3 Compiler Generation .................................. 262
   20.4 Cycle Accurate Simulator ............................. 262
      20.4.1 Better SSP Generation ......................... 262
      20.4.2 OL_Inter Resolving .............................. 263
List of Figures

2.1 The basic idea of NoGap ......................... 15
4.1 NoGap system architecture ....................... 32
4.2 NoGap system architecture ....................... 32
4.3 Mase and Mage relationship ...................... 34
4.4 NoGap system flow ............................... 35
4.5 Transformation path and bridge ................... 38
5.1 Instruction format example ....................... 53
6.1 Mage example .................................... 60
6.2 Mage class diagram ............................... 63
7.1 VertexInfo inheritance hierarchy ................. 71
7.2 V_Mase colors 1 ................................... 73
7.3 V_Mase colors 2 ................................... 74
7.4 ArcInfo inheritance hierarchy ..................... 75
7.5 E_Mase colors .................................... 77
7.6 FU cluster example ................................ 78
7.7 OL_intra ........................................... 80
7.8 OL_inter ........................................... 80
7.9 Simple Mase graph ................................ 81

xxxv
<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>7.10</td>
<td>Equal $E_{\text{Mase}}$ combination example</td>
<td>82</td>
</tr>
<tr>
<td>7.11</td>
<td>FF insertion example</td>
<td>83</td>
</tr>
<tr>
<td>7.12</td>
<td>Equal FFs combination example</td>
<td>84</td>
</tr>
<tr>
<td>7.13</td>
<td>Multiplexer insertion example</td>
<td>86</td>
</tr>
<tr>
<td>7.14</td>
<td>Set edge sizes example</td>
<td>87</td>
</tr>
<tr>
<td>7.15</td>
<td>Stalling FF connection example</td>
<td>88</td>
</tr>
<tr>
<td>8.1</td>
<td>Symbol table architecture</td>
<td>90</td>
</tr>
<tr>
<td>8.2</td>
<td>Naming and type name $V_{\text{Mase}}$</td>
<td>95</td>
</tr>
<tr>
<td>9.1</td>
<td>FU placement example</td>
<td>129</td>
</tr>
<tr>
<td>9.2</td>
<td>Add/Sub-AND/OR XOR</td>
<td>130</td>
</tr>
<tr>
<td>9.3</td>
<td>Static connections</td>
<td>136</td>
</tr>
<tr>
<td>9.4</td>
<td>Jump prologue and epilogue</td>
<td>137</td>
</tr>
<tr>
<td>9.5</td>
<td>PC-FSM example with action and condition tree</td>
<td>141</td>
</tr>
<tr>
<td>9.5.1</td>
<td>PC-FSM graph</td>
<td>141</td>
</tr>
<tr>
<td>9.5.2</td>
<td>Condition tree</td>
<td>141</td>
</tr>
<tr>
<td>9.5.3</td>
<td>Action tree</td>
<td>141</td>
</tr>
<tr>
<td>10.1</td>
<td>Small sizing example</td>
<td>152</td>
</tr>
<tr>
<td>10.2</td>
<td>Size annotation traversal</td>
<td>154</td>
</tr>
<tr>
<td>10.3</td>
<td>Simple relation graph</td>
<td>159</td>
</tr>
<tr>
<td>10.4</td>
<td>Simple relation graph reduced for symbol6</td>
<td>161</td>
</tr>
<tr>
<td>10.5</td>
<td>Parallel dependent loops</td>
<td>162</td>
</tr>
<tr>
<td>10.6</td>
<td>Side loop</td>
<td>162</td>
</tr>
<tr>
<td>10.7</td>
<td>Loop in loop</td>
<td>162</td>
</tr>
<tr>
<td>11.1</td>
<td>Pipeliner slice</td>
<td>169</td>
</tr>
<tr>
<td>11.2</td>
<td>Generated pipelining $\text{Mase}$</td>
<td>171</td>
</tr>
<tr>
<td>11.3</td>
<td>Pipeline usage vector examples</td>
<td>172</td>
</tr>
<tr>
<td>12.1</td>
<td>Generator system</td>
<td>178</td>
</tr>
</tbody>
</table>
List of Figures

13.1 SystemVerilog spawner flow . . . . . . . . . . . . . . . . . . . . 190
14.1 NoGap assembler flow . . . . . . . . . . . . . . . . . . . . . . . . 201
15.1 Simulator overview . . . . . . . . . . . . . . . . . . . . . . . . . . 211
15.2 Mage dependency graph for a register file . . . . . . . . . . . . . 213
15.3 Before SSP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
15.4 After SSP . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 215
15.5 Loop removal thanks to SSP partitioning . . . . . . . . . . . . . . 215
16.1 Adder architecture overview . . . . . . . . . . . . . . . . . . . . . . 224
17.1 PIONEER . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 233
17.2 PIONEER with Joust approach . . . . . . . . . . . . . . . . . . . . . 235
17.3 Verification on DaFk system . . . . . . . . . . . . . . . . . . . . . . . 237
18.1 Pipeline architecture for normal instruction in SENIOR . . . . . . . 244
18.2 Pipeline architecture for convolution instruction in SENIOR . . . . . 245
# List of Tables

5.1 PU IR types explained ............................... 41

7.1 *M*ase graph *V*$_{M}$ase types ...................... 72
7.2 *M*ase graph *E*$_{M}$ase types ...................... 76
7.3 FU cluster types .................................... 78
7.4 Predefined *M*ase transformations ................... 80

9.1 Operators in NoGap$^{CL}$ .............................. 101
9.2 Keywords in NoGap$^{CL}$ ................................ 106
9.3 Special character combinations in NoGap$^{CL}$ ........ 107
9.4 Sizing functions for inline expression operators ....... 132
9.5 PC-FSM clauses ........................................ 142
9.6 NoGap system calls .................................... 148

11.1 Class selection example .............................. 169

12.1 Available generators (A-M) .......................... 180
12.2 Available generators (N-V) .......................... 181
12.3 Generator interface .................................. 182

13.1 HDL writer required sub-classes ..................... 194

16.1 Performance in various devices ...................... 227
16.2 Adder resource utilization in Virtex 4 ............... 228
16.3 Comparison with USC adder Virtex-II ............... 229
16.4 Comparison with Nallatech adder Virtex-II .......... 229
16.5 Comparison with Xilinx adder Virtex-4 ............... 229
16.6 Comparison with Xilinx adder Virtex-5 ............... 230

17.1 FPGA utilization by part ......................... 239

18.1 Pipeline Specification ........................... 243
18.2 Explanation of pipeline stages for SENIOR ......... 243
18.3 Move-load-store instructions ...................... 246
18.4 Short ALU instructions ......................... 248
18.5 Long ALU instructions .......................... 249
18.6 Flow instructions ............................... 251
18.7 FPGA utilization by part ......................... 254
18.8 Comparison for SENIOR in FPGA ................. 255
18.9 Comparison for SENIOR in ASIC ................. 255
List of Grammars

9.1 Concatenation grammar ........................................ 105
9.2 FU specification grammar ....................................... 107
9.3 If statement grammar ........................................... 109
9.4 Switch structure grammar ....................................... 110
9.5 Cycle and comb grammar ........................................ 111
9.6 Clause grammar .................................................. 111
9.7 Port and signal definition grammar ............................. 112
9.8 Manual FU instantiation grammar ............................... 115
9.9 Phase definition grammar ....................................... 117
9.10 Stage declaration grammar ..................................... 117
9.11 Pipeline description grammar .................................. 119
9.12 FU use definition grammar ..................................... 122
9.13 Operation definition grammar .................................. 123
9.14 FU usage list grammar ........................................... 127
9.15 Instruction declaration grammar ............................... 145
9.16 NoGap directive grammar ....................................... 147
14.1 NoGapAsm grammar .............................................. 203
LIST OF GRAMMARS
List of Algorithms

10.1 FU input port processing ........................................ 155
10.2 Size annotation algorithm ...................................... 156
10.3 Relation graph construction .................................... 159
10.4 Maximum substitution algorithm ............................... 164

11.1 Operation mergability ............................................. 173

12.1 NcFap’s special folder creation and/or use ..................... 187

15.1 Mage dependency graph generation ............................ 212
15.2 SSP Partitioning .................................................... 216
15.3 Mage sequentialization ............................................ 218
List of Listings

1.1 BGL graph definition ...................................... 6
4.1 Action register usage example ............................. 37
5.1 PU excerpt .................................................. 40
5.2 Essential clause access rule code ......................... 43
5.3 Essential pipelines code .................................... 45
5.4 Essential control path code .................................. 46
5.5 Essential operation info code ............................ 47
5.6 Essential operation classification code .................... 49
5.7 Essential instruction decoder code ....................... 51
5.8 Instruction decoder data ..................................... 51
5.9 Essential instruction table code ........................... 52
5.10 Essential SSP partition code ............................... 53
5.11 Transcript of a Mase PU ................................. 54
5.12 Transcript of a Mage PU ................................... 56
6.1 Mase graph BGL type ....................................... 61
6.2 Statement declaration ....................................... 64
6.3 Instruction declaration ...................................... 65
6.4 Mage Declaration ............................................. 66
6.5 Essential Mage dependency graph code .................. 67
7.1 Mase graph BGL type ....................................... 70
8.1 Symbol table Declaration ................................... 91
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>8.2</td>
<td>op_i definition</td>
<td>92</td>
</tr>
<tr>
<td>8.3</td>
<td>FU instantiation AST navigation methods</td>
<td>94</td>
</tr>
<tr>
<td>8.4</td>
<td>FU instantiations</td>
<td>94</td>
</tr>
<tr>
<td>8.5</td>
<td>Symbol table excerpt for cp and dp</td>
<td>94</td>
</tr>
<tr>
<td>9.1</td>
<td>FU Specification</td>
<td>108</td>
</tr>
<tr>
<td>9.2</td>
<td>If control structure example</td>
<td>109</td>
</tr>
<tr>
<td>9.3</td>
<td>Switch control structure example</td>
<td>110</td>
</tr>
<tr>
<td>9.4</td>
<td>Signal and ports example</td>
<td>113</td>
</tr>
<tr>
<td>9.5</td>
<td>Port offset</td>
<td>114</td>
</tr>
<tr>
<td>9.6</td>
<td>Manual FU instantiation example</td>
<td>116</td>
</tr>
<tr>
<td>9.7</td>
<td>Stage stall example</td>
<td>118</td>
</tr>
<tr>
<td>9.8</td>
<td>Pipeline description examples</td>
<td>121</td>
</tr>
<tr>
<td>9.9</td>
<td>Operation FU example</td>
<td>125</td>
</tr>
<tr>
<td>9.10</td>
<td>FU usage list examples</td>
<td>128</td>
</tr>
<tr>
<td>9.11</td>
<td>Inline FU usage example</td>
<td>131</td>
</tr>
<tr>
<td>9.12</td>
<td>Forwarding path descriptions</td>
<td>133</td>
</tr>
<tr>
<td>9.13</td>
<td>Static connection in Mase</td>
<td>135</td>
</tr>
<tr>
<td>9.14</td>
<td>PC-FSM template specification</td>
<td>140</td>
</tr>
<tr>
<td>9.16</td>
<td>Decoder instantiation example</td>
<td>143</td>
</tr>
<tr>
<td>9.17</td>
<td>NoGap call example</td>
<td>147</td>
</tr>
<tr>
<td>10.1</td>
<td>FU port sizing example</td>
<td>151</td>
</tr>
<tr>
<td>12.1</td>
<td>Generator interface</td>
<td>179</td>
</tr>
<tr>
<td>12.2</td>
<td>Generate method</td>
<td>179</td>
</tr>
<tr>
<td>12.3</td>
<td>Generator register usage</td>
<td>183</td>
</tr>
<tr>
<td>12.4</td>
<td>Generator usage</td>
<td>184</td>
</tr>
<tr>
<td>12.5</td>
<td>Generate through PUs</td>
<td>185</td>
</tr>
<tr>
<td>12.6</td>
<td>Generator flow file</td>
<td>186</td>
</tr>
<tr>
<td>13.1</td>
<td>HDL code generation method</td>
<td>192</td>
</tr>
<tr>
<td>13.2</td>
<td>HDL writer class example</td>
<td>193</td>
</tr>
<tr>
<td>13.3</td>
<td>Generated Mase SystemVerilog code</td>
<td>195</td>
</tr>
<tr>
<td>Section</td>
<td>Title</td>
<td>Page</td>
</tr>
<tr>
<td>---------</td>
<td>---------------------------------------------------</td>
<td>------</td>
</tr>
<tr>
<td>13.4</td>
<td>Mage AST generation excerpt</td>
<td>197</td>
</tr>
<tr>
<td>13.5</td>
<td>Generated decoder excerpt</td>
<td>198</td>
</tr>
<tr>
<td>14.1</td>
<td>Instruction set for our test processor</td>
<td>206</td>
</tr>
<tr>
<td>14.2</td>
<td>Program excerpt</td>
<td>207</td>
</tr>
<tr>
<td>14.3</td>
<td>ASCII-binary with comments</td>
<td>208</td>
</tr>
<tr>
<td>15.1</td>
<td>Block Example</td>
<td>211</td>
</tr>
<tr>
<td>15.2</td>
<td>Register Values</td>
<td>219</td>
</tr>
<tr>
<td>16.1</td>
<td>Mase example</td>
<td>226</td>
</tr>
<tr>
<td>17.1</td>
<td>PIONEER in NoGap(^CL) description</td>
<td>235</td>
</tr>
<tr>
<td>18.1</td>
<td>Mage in SENIOR</td>
<td>250</td>
</tr>
<tr>
<td>18.2</td>
<td>Mase in SENIOR</td>
<td>251</td>
</tr>
<tr>
<td>20.1</td>
<td>Improved Sizing Expression</td>
<td>265</td>
</tr>
</tbody>
</table>
Part I

Prologue
1

Assumed Reader Knowledge

We now have a whole culture based on the assumption that people know nothing and so anything can be said to them.

---

Stephen Vizinczey

I have written this thesis with the assumption that the reader has a certain level of understanding of some key concepts and technologies. This since both time and space constraints do not allow me to include text oriented toward novices. I have however, in this Chapter, tried to list the most important key items needed to understand this thesis. The additional resources I cite here should cover most of the knowledge base needed to fully understand this thesis.
Chapter 1: Assumed Reader Knowledge

1.1 C++

It is assumed that the reader is well-adversed with C++ [41]. Concepts such as STL [15], Boost [46] and templates [44] should be well known to the reader. If they are not the reader is advised to turn to the references just cited.

1.2 Verilog/VHDL

It is assumed that the reader is familiar with either Verilog [33] or VHSIC Hardware Description Language (VHDL) [40] and has basic understanding how they are used to model hardware at the Register Transfer Language (RTL).

1.3 Hardware Design

It is assumed that the reader is familiar with RTL design and understands concepts such as binary calculations, pipelining, and hardware multiplexing. It is further assumed that the reader knows what Field Programmable Gate Arrays (FPGAs) and Application Specific Integrated Circuits (ASICs) are and the methods used to synthesize designs for these devices. There are numerous resources that handles the topic of hardware design at various levels. Some suggestions for further information are [37, 10, 25]

1.4 Processor Design

It is assumed that the reader is familiar with basic processor design concepts, such as decoders, register files, data paths, instructions, micro-operations, etc. For more information about processor design see [26].
1.5 Graphs

It is assumed that the reader is familiar with elementary graph theory. Concepts such as vertices, edges, loops, and trees should be well understood. It is further assumed that the reader is familiar with some graph algorithm such as Depth First Search (DFS) and topological sort. I will therefore not discuss any of that in this Chapter. For more information about graph theory and algorithms, see [7].

1.5.1 Graphs in NoGap

All graphs in Novel Generator of Accelerators And Processors (NoGap) uses the Boost Graph Library (BGL). For a full documentation of BGL, the reader is referred to the relevant part of the Boost documentation [46].

BGL graphs only specifies and deals with Graph vertices ($V$s) and Graph edges ($E$s). BGL does not care what data is actually associated with the $V$s and $E$s. Some BGL algorithms do require some knowledge of this data, but in that case the programmer has to inform the BGL how to extract the needed data.

In NoGap the data associated with the BGL graph is normally a class for the $V$ and another class for the $E$. There are two approaches used to distinguish different types of $V$s and $E$s. For more complex graphs with many different types, class hierarchies are used. For simpler graphs with only a few different $V$ and $E$ types, enums are used to mark the type.

An example of a BGL graph definition can be seen in Listing 1.1, where the class NodeData is used for $V$ data and the class EdgeData is used for $E$ data. The meaning of the other parameters is outside the scope of this thesis.
Listing 1.1: BGL graph definition

typedef boost::adjacency_list<boost::listS,
  boost::listS,
  boost::bidirectionalS,
  NodeData,
  EdgeData,
  boost::listS> variable_deps_t;
Part II

Background
Introduction

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

Arthur Machen

When designing a processor or a pipelined data path nowadays, one can either choose to use an Hardware Description Language (HDL) such as Verilog or VHDL or one can choose to use one of many high level design tools, often called Electronic Design Automation (EDA) tools. Both roads have drawbacks and advantages. For example when using an HDL one gets almost complete control at the register transfer level, but at the same time one has to manage all minuscule details regarding hardware multiplexing, instruction decoding, control signal generation, and control signal pipelining. On the other end of the spectrum, one can use an existing EDA tool. In that case, one will not have to think about secondary details, instead one can be focused on what one wants to accomplish. However, at the same time one will loose control over the final hardware, often being stuck with a design that is more a product of the possibilities of the EDA tool used, rather than truly novel and
creative architectures.

With **NoGap** we have tried to solve this conundrum of choosing either a low level or a high level tool. **NoGap** offers low level control at the register transfer level, if so desired, while at the same time offering the possibility to ease construction of instruction controlled data paths, something found in both programmable accelerators and, naturally, in processors. **NoGap** gives humans control over what humans do best, being creative and solving one thing at a time. **NoGap** then gives the computer control over what the computer does best, handling multiple variables at a time and doing computations. Using **NoGap** one can specify reasonably sized modules as freely as in any HDL but one gets support combining these modules together into complex programmable data paths. **NoGap** also produces assemblers, cycle accurate simulators and synthesizable HDL code, all from the same source. Thus the assembler, simulator, and HDL code will always be functionally coherent, let alone relieving designers from these secondary design tasks. This thesis will go into detailed descriptions about what **NoGap** is and how **NoGap** is implemented. If this brief description of **NoGap** seem interesting for you, it is my hope that you dwell deeper into my thesis to see if **NoGap** is a tool that might suit your needs.

### 2.1 Rationale

The art of digital electronic design is at a crossroads. At the same time as feature sizes are shrinking for every new generation of silicon fabrication technology, the task of designing a fully functional system, which can make use of all the transistors at its disposal, is getting ever harder. Having gone from full custom designs, via schematic entry followed by first and second generation RTLs, today a large number of EDA tools have emerged in the academy and industry to alleviate the burden put on designers. Each of these tools have their advantages and disadvantages.
The main disadvantage is that the abstraction level removes a designer from the micro architecture and the currently existing EDA tools will therefore hamper the development of novel architectures. More details about some of the EDA tools available today can be found in Chapter 3.

As a consequence of the increasing transistor density for every new manufacturing generation, digital devices are becoming an ubiquitous part in our society. Also people in general have gotten used to devices getting smaller and more competent, like small electronic Swiss-army knives. This demand has of course resulted in a huge market advantage for the company that can sell the best product to the lowest possible price. However the main part of the price for a mass fabricated electronic devices today is the Non-Recurring Engineering (NRE) price. The actual manufacturing price per unit will be relatively low. The quest is therefore to reduce the price tag of the NRE.

The current trend, and customer demand, for ever more capable devices has led to a fierce competition where the Time To Market (TTM) is an important factor since the time window, where a certain product is profitable, is usually very small. The current trend is also towards mobile devices, which should be able to operate with at most one recharge per day. So in essence the demand is for devices and systems that are ever more capable using ever less power. Usually ASICs have been viewed as the devices that gives the lowest power over Mega Instructions Per Second (MIPS) ratio or Watts Per MIPS (WPM) and has therefore been the premier choice for devices with low WPM requirements. ASICs however, have a very limited flexibility, meaning that it is often hard to reuse them in future generation systems or modify the functionality of an already existing system. On the other end of the spectrum are general purpose processors, e.g. Athlon II, Core-i7, or ARM. They are very flexible and can be adapted to perform most of the tasks that are requested of them. They however usually have a high WPM ratio. Meaning that they are not suited for high performance mobile
devices. The middle ground here is to use a device that is sufficiently flexible while still maintaining a reasonably low WPM. Limiting the requirement to that the device only has to be flexible within a specific domain, it is usually possible to reach a low WPM while maintaining domain flexibility. These devices usually take the form of some kind of Application Specific Instruction-set Processor (ASIP). Another way of reaching a low WPM while still maintaining some flexibility is to use a simple micro controller supplemented with one or more programmable accelerators. This configuration can be the basis of a general design platform that can be adopted to new demands either by reprogramming the accelerators, or if needed, designing new accelerators. These accelerators could be ASICs, but if programmable accelerators are used, the same accelerator can be used for slightly different tasks, thus the same hardware can be reused to perform a multitude of functions. The Base Band Processor (BBP) [43], which introduced the Single Issue Multiple Tasks (SIMT) concept is a good example of how such a system can be very power and area efficient.

There is however a number of problems associated with developing an ASIP or a programmable accelerator, among those are the design time and the NRE price. This can make, going down this design path, a daunting prospect, therefore a less optimal design solution might be used. The NRE cost of ASIP or accelerator design can however be alleviated by using an EDA tool. Chapter 3 will give a survey of a number of existing tools and their limitations. Different EDA tools are useful for either ASIP or accelerator design. The problem with existing ASIP design tools are that they limit the design to a certain template design and makes too many assumptions about how an ASIP should be designed. Basically they take on the problem of processor design from a too high level. And the problem with the current accelerator design tools are that they create an accelerator for a very specific algorithm, in effect creating an ASIC.
The traditional approach, taken by processor design tools, is to start from an Instruction Set Architecture (ISA) description. This approach carries with it a number of benefits and drawbacks. Some benefits are shorter design times for both Hardware (HW) and compilers. Furthermore they are usually user friendly and enables inexperienced designers to design functional processors. These tools are suitable for relatively normal applications and simple instruction acceleration. On the other hand, the drawbacks are plentiful and they all stem from the fact that a traditional, ISA based, processor design tool has to make a lot of assumptions about the micro architecture. In fact the tool will become an expert system, designing a micro architecture which is a version of how the tool’s designers envisions a processor micro architecture. Therefore the performance of the design will be limited by the tool. There might be a number of issues, such as memory bandwidth limitations, degraded computation performance for advanced architectures, too high WPM. In short it is very hard to develop truly novel architectures with the standard tools.

In our opinion, the data path architecture for efficient ASIPs or flexible accelerators has to be designed by humans. As of yet computers are not creative enough to do this in a reliable manner. Then, if you know the data path architecture, you do not need a tool to generate it for you. As of yet these limitations in EDA tools means that designers often revert back to an HDL such as Verilog or VHDL. HDLs offer almost complete design flexibility but at the same time forces the designer to handle the tedious and error prone task of managing every little detail in the data path and its control signals.

We have therefore proposed a new tool that we call Novel Generator of Accelerators and Processors (NoGap). NoGap assumes very little about what is being designed, but has a number of ways of supporting ASIP or Accelerator designs. In essence NoGap lets a designer make all the creative decisions and then, if the designer has asked for it, synthe-
sizes hardware multiplexing and the needed control paths. An important aspect of NoGap is that it does not do anything unexpected. This can be likened with the situation of programming in higher level software languages such as C. A C programmer knows, that a function call, will result in assembly instructions for parameter passing, the actual call instruction, and stack handling. The programmer, does not need to know the details of how this is done but the programmer knows that this is a determinable effect of the C code’s function call\(^1\). NoGap works much in the same way, i.e. a designer can, at design time, know exactly what NoGap is going to do, although s/he does not need to know the exact details of what is being synthesized by NoGap. To achieve design freedom a central concept in NoGap is compositional design, where each part is independent and does not know about the whole, but putting all individual parts together can create very powerful and flexible architectures.

The general principle of NoGap is based upon the idea that a number of Functional Units (FUs) are interconnected in some kind of interconnection network. This principle is depicted in Figure 2.1, where a number of FUs in one way or another communicates with each other.

In principle NoGap can be used to describe any single clock domain hardware architecture. This since the Micro Architecture Generation Essentials (Mage) descriptions are as expressive as normal RTL code. However using NoGap in this way gives little or no advantage over using a well designed RTL such as Verilog or VHDL. What NoGap is good at is describing micro architectures using instruction driven pipelines using a time stationary decoding and much of the work presented deals with how to manage and handle these kinds of architectures.

\(^1\)Assuming no optimizations
Chapter 2: Introduction

2.2 Introduction to NoGap

This section gives a quick introduction to NoGap and its main components. A more detailed explanation can be found in Part III.

NoGap approaches the problem of giving support while not limiting the designer by making few assumptions about the system being designed. The underlying design principle in NoGap is based on the assumption that designing individual modules, e.g., adders, multipliers, or Arithmetic Logic Units (ALUs) of a pipelined architecture is generally a fairly simple task for a human, even for a relatively inexperienced designer. Specifying the temporal and spatial relations between these modules on a per instruction basis is also something humans are good at. The hard part for a human is to merge all these instructions into a pipelined ASIP architecture having the necessary control signals, multiplexers, and associated delays. On the other hand this is a fairly easy
task for a computer since in principle no more creative decisions has to be made. For this reason the design input to NoGap is descriptions of the individual FUs (can be supplied as a user defined library), an instruction set\(^2\), the per instruction data path architecture, and a set of constraints. NoGap then compiles this information to an intermediate representation that can be used to generate, for example, assemblers, simulators, and synthesizable RTL code. This flow is shown in Figure 4.2

NoGap consists of a number of different components. The system can be divided into three main parts, NoGap Common Description (NoGap\(^{CD}\)), facets, and spawners.

NoGap\(^{CD}\) (NoGap Common Description) is an intermediate representation of the system being designed, NoGap\(^{CD}\) is generated from Abstract Syntax Tree (AST) graph descriptions of the individual FUs. This AST description is constructed using a C++ Application Programmer Interface (API).

A facet is a tool that constructs a NoGap\(^{CD}\) through the C++ API. One facet has been implemented as a language, exposing all functionality of NoGap, this language is called NoGap Common Language (NoGap\(^{CL}\)). A facet can also indirectly use the C++ API by generating a NoGap\(^{CL}\) description and rely on the NoGap\(^{CL}\) parser to generate the needed ASTs. NoGap\(^{CL}\) is explained in more details in Chapter 9.

A spawner is a tool that reads the NoGap\(^{CL}\) to construct some useful output. For example, synthesizable Verilog code, a cycle and bit accurate simulator, or an assembler for a generated processor. Some notable spawners are described in Part IV.

This layered approach allows for a flexible system, where spawners are independent of the facets. Using NoGap as a back end, it will be easy to design a more dedicated facet, e.g. a Digital Signal Processor (DSP) processor design facet or just a simple data path designer facet. The

\(^2\)Fixed function data paths has an instruction set consisting of a single instruction
spawners will still work and output the correct output.

For example, a cycle accurate simulator spawner could have been implemented for an earlier project, but a new facet would ease the design effort for a new project. In this case only a new facet has to be implemented and time/money can be saved by reusing the old spawner.

\(\text{NoGap}^{CD}\) contains information about all FUs in the system. The three most important components of \(\text{NoGap}^{CD}\) are \textit{Micro Architecture Structure Expression (Mase)}, \textit{Mage} and \textit{Control Architecture ST ructure Language (Castle)}.

\textit{Mase} is an annotated data flow graph describing all spatial and temporal connections between FUs for all operations defined on this particular data path. \textit{Mase} is further described in Chapter 7.

\textit{Mage} is a somewhat modified and optimized version of the original AST. \textit{Mage} FUs are seen as black boxes in the \textit{Mase} graph. \textit{Mage} FUs can be seen as operators in \textit{Mase}. For example, simple FUs such as adders or multipliers, but also more complex \textit{Mase} FUs can be used, such as ALUs or a complete MAC unit. It is up to the designer to decide how complex the \textit{Mage} units shall be. \textit{Mage} is further described in Chapter 6.

\textit{Castle} contains the information needed to construct instruction decoders, e.g. specifying instruction formats, source-, and destination operands.

### 2.3 Thesis Organization

**Part I** aims to inform readers about what kind of knowledge is needed in order to understand this thesis (Chapter 1).

**Part II** aims to give a background of this research (Chapter 2) and then goes on to survey some other tools which have functionalities similar to \(\text{NoGap}\)’s (Chapter 3).
Part III aims to give the reader a deeper understanding of NoGap. First an architecture overview of NoGap is presented (Chapter 4), then an in depth look at NoGap\textsuperscript{CD} where some key pars are discussed. The key parts discussed in this thesis are; the Parse Unit (PU) (Chapter 5), Mage (Chapter 6), Mase (Chapter 7), and the symbol table (Chapter 8). This part then goes on to present NoGap\textsuperscript{CL} an architecture description language, developed as part of my research work (Chapter 9). This is followed by a description about how NoGap, automatically, can set the correct sizes for ports and buses (Chapter 10). And finally a description about how control signals to a pipelined data path are handled by creating a pipeliner (Chapter 11).

Part IV aims to give the reader a deeper understanding of the spawner and generator concept (Chapter 12). Then three actual spawners, developed as part of my research work, are presented, these spawners are; a SystemVerilog spawner (Chapter 13), an assembler spawner (Chapter 14), and a simulator spawner (Chapter 15).

Part V presents three different case studies done to prove and verify NoGap’s usability as a processor and accelerator design tool. In the first study a single precision floating point adder and subtracter was implemnts using NoGap (Chapter 16). In the second study a simple Reduced Instruction Set Computing (RISC) processor was implemented using NoGap (Chapter 17), and in the third study a complex RISC DSP processor was implemented using NoGap (Chapter 18).

Part VI wraps this thesis up, by first giving some conclusions (Chapter 19) and then discussing future work, needed and/or desired, to improve NoGap (Chapter 20).
Related Work

No man or woman is an island. To exist just for yourself is meaningless. You can achieve the most satisfaction when you feel related to some greater purpose in life, something greater than yourself. 

Denis Waitley

There are many tools available that promise to ease the processor construction process from the RTL. Many of these tools also construct compilers, assemblers, linkers, simulators, and/or debuggers. This is done since the actual processor hardware is just a small part of a successful processor. Of equal importance is the supporting tool chain making the processor easy to use and integrate into larger systems. This chapter will review the state of the art today and describe what NoGap has to offer compared to these other tools. A good summary of available processor design tools is presented in [29].
3.1 LISA

LISA [35] is one of the major tools for high level descriptions of processors. LISA can generate both the relevant tool chain, such as compilers, simulators, assemblers, and synthesizable HDL code.

3.1.1 Strengths of LISA

LISA has many appealing features such as compiler, assembler, debugger, and hardware generation. The language used in LISA allows for a large range of processors to be described. LISA also supports construction of pipelines with data and structural hazards management.

3.1.2 Weaknesses of LISA

Many different processor can be constructed with LISA, but LISA still assumes a basic architecture of the processor and really novel ASIP processors deviating from this basic architecture is if not impossible to describe with LISA at least very cumbersome. The instructions in LISA is restricted to a tree like format where each instruction is composed of sub-instructions, which in turn can be sub-instructions. It is up to the designer to assign the binary coding to each sub-module.

3.1.3 LISA in Comparison with NoGap

NoGap assumes less about the architecture from the start then LISA, further NoGap can utilize hardware multiplexing to implement multiple functions using fewer hardware components. NoGap can also, in contrast to LISA, generate a binary coding by itself.

3.2 MESCAL/Tipi

MESCAL [28] is a micro architecture construction framework not locked to a particular design. It uses actors with guarded actions as its atomic
elements. These actors can then be connected together to form more complex behavior. Tipi is a graphical design environment built on top of MESCAL and is used to construct data paths with the actors. The tool will then generate all possible connections between all unconnected actor ports.

### 3.2.1 Strengths of MESCAL/Tipi

The flexibility of MESCAL enables almost any micro architecture to be described and the designer is not locked into a particular design by the tool.

### 3.2.2 Weaknesses of MESCAL/Tipi

To define a micro architecture from an instruction specification in MESCAL can be a bit tricky since a number of instructions are generated and the designer then has to go through all possible instructions to sort out the interesting ones and also impose the correct restrictions on the use of the actors.

### 3.2.3 MESCAL/Tipi in Comparison with NoGap

MESCAL and NoGap has a number of common ideas but also a number of differences. The concept of a common intermediate description is shared by the two frameworks. While NoGap has a unified language for all components in the design, MESCAL has one description for the leaf modules and another description for the data path. However, NoGap and MESCAL differs in how the architecture is described. In NoGap a description of all instructions needed is used. NoGap thus guarantees that at least those instructions are implemented.
3.3 EXPRESSION

EXPRESSION [14] is a tool aimed at simulator and compiler re-targeting. The processor behavior is specified in two descriptions: a behavior specification and a structure specification. The behavior specification consists of operation specification, instruction specification, and operation mappings. The structure specification consists of architecture components specification, pipeline and data transfer paths, and memory subsystems.

3.3.1 Strengths of EXPRESSION

A main advantage of EXPRESSION seems to be the ease of modeling memory subsystems. Its functional unit view of the processor makes it fairly easy to make changes to the architecture and to design space exploration.

3.3.2 Weaknesses of EXPRESSION

EXPRESSION is not built to be a processor construction tool. Thus details needed to get a real processor working is not possible to express in the language.

3.3.3 EXPRESSION in Comparison with NoGap

There are a number of similarities to NoGap but EXPRESSION is not aiming to get the processor description down to hardware in the end. Thus some expressiveness is missing from the EXPRESSION language.

3.4 ArchC

ArchC [38] is an Architecture Description Language (ADL), creating SystemC models of the processor described, this model is then used as a platform to simulate the processor in question.
Chapter 3: Related Work

3.4.1 Strengths of ArchC

The expressiveness of ArchC allows for a large range of processors to be described. It can first be used to develop a functional model of the processor and later this model can be refined into a cycle accurate model. ArchC also outputs an assembler for the processor in question. ArchC seems to be a perfect tool if only an assembler and simulator is needed. ArchC also has a Co-verification feature so the next refinements of the processor can be verified against the previous refinement.

3.4.2 Weaknesses of ArchC

The purpose of ArchC does not seem to be to create a real hardware processor. The SystemC model might in the end be synthesizable but this has to rely on other work done to make SystemC synthesizable. The designer still has to worry about instruction coding and the assembler format is intermixed with its behavior.

3.4.3 ArchC in Comparison with NoGap

To describe a processor in NoGap will probably require some more work but on the other hand the final product of NoGap is a processor model which can be synthesized to hardware.

3.5 nML

nML [11] is a formalism, or language aiming at describing processor from their instruction set manuals. The description of a processor in nML is based on the information typically found in a program manual.

3.5.1 Strengths of nML

nML is a very concise language for processors fitting a certain domain, short descriptions will suffice to capture the behavior of the processor. If
an instruction set manual exist and fits what can be described in nML, nML is a good choice for a designer.

3.5.2 Weaknesses of nML

nML is restricted to processors with a single instruction stream and PC. Thus only fairly standard architectures can be described. How nML handles stalls, especially stalls with a random delay, is unclear. Furthermore nML seems to be aimed at constructing a simulator and code generator from an already existing instruction set manual.

3.5.3 nML in Comparison with NoGap

If the processor being designed fits the assumptions of nML, nML is a very powerful tool. nML is although quite cumbersome to use as a tool for processor development. A NoGap description will probably be larger, but NoGap can then encompass a larger set of processor architectures than nML.

3.6 SimpleScalar

SimpleScalar [6, 1] is an infrastructure for computer system modeling and program performance analysis. A designer can, for example, model novel cache architectures, instruction level parallelism, or novel branch prediction schemes. The tool is in extensive use in academia, especially in the computer science domain.

3.6.1 Strengths of SimpleScalar

SimpleScalar’s versatility makes it a very competent tool when e.g. testing how a new cache architecture will affect the performance of a program.
3.6.2 Weaknesses of SimpleScalar

SimpleScalar is, however, more of a performance profiling tool and is restricted to a number of instruction sets such as Alpha, PISA, ARM and x86. SimpleScalar is not a good tool to use for trying out novel processor architectures with novel instruction sets.

3.6.3 SimpleScalar in Comparison with NoGap

SimpleScalar and NoGap are far apart from each other. SimpleScalar is aiming at simulating novel modifications to an architecture running a known instruction set. NoGap is a tool for constructing novel processors and accelerators.

3.7 MIMOLA

MIMOLA was a pioneer in processor synthesis, it was introduced in the late 70’s [27]. MIMOLA, also described in Chapter 3 in [29], uses a net list description of a processor together with algorithms described in a PASCAL-like language to create instructions for the data path that conforms to the algorithm in question. MIMOLA can also be used to implement a micro-coded interpreter for a “normal” instruction set, and in this way a programmable processor can be generated with MIMOLA.

3.7.1 Strengths of MIMOLA

MIMOLA makes it easy to develop processor for certain algorithms, although in later years, this ability has been superseded by tool like Catapult C [13]. But at the time MIMOLA was developed it was a great step forward from what previously existed. MIMOLA also eased the task of designing programmable processors, and supporting tool chains. Another good feature is the automatic test program generation that MIMOLA supports.
3.7.2 Weaknesses of MIMOLA

There is no notion of a pipeline in MIMOLA. It seems MIMOLA assumes that all instructions are single cycle and it is thus hard to use MIMOLA for modern DSP processors with multi cycle pipelines. It is also unclear if cycle accurate simulators can be generated. Much of these weaknesses can stem from the fact that MIMOLA was a pioneering tool and at that time the concept of instruction pipelined integrated processors, were still in its infancy.

3.7.3 MIMOLA in Comparison with NoGap

NoGap and MIMOLA seems to have some similarities in that functional units and their interconnects are described separately. However NoGap supports multi cycle instructions and can generate cycle accurate simulators.

3.8 MDES


3.8.1 Strengths of MDES

The MDES language makes it easy to describe parametrization of a base processor architecture, which then automatically is used to generate the more detailed information to the retargetable compiler.
3.8.2 Weaknesses of MDES

The Trimaran package all works around a base processor architecture thus novel processor architectures can not be explored with MDES.

3.8.3 MDES in Comparison with NoGap

NoGap and MDES is far from each other, while MDES is aimed at describing parametrization of an existing processor architecture NoGap is a general language for FU centric machines.

3.9 ASIP Meister

ASIP Meister [36] is a processor construction tool. The user enters parameters about the processor like the length of the pipeline, what pipeline stage the instruction decoding is done, and if register forwarding or delayed branches is to be used. The user also supply information about what an instruction does and the instruction coding.

3.9.1 Strengths of ASIP Meister

ASIP Meister can be used to produce a processor, simulator, assembler, and compiler with very little design effort. ASIP Meister also generates synthesizable RTL code.

3.9.2 Weaknesses of ASIP Meister

ASIP Meister is an expert system for RISC architecture construction. Which makes it hard to construct a processor deviating from the standard RISC architecture.

3.9.3 ASIP Meister in Comparison with NoGap

NoGap and ASIP Meister is similar since they both aim to produce hardware implementations of the described processor. NoGap can how-
ever be used to design a RISC construction expert system. But its core language makes NoGap much more versatile.
Part III

NoGap Internals
4.1 Architecture Overview

The top level architecture of NoGap can be split into three parts: facets, NoGap-core, and spawners. An overview of the architecture can be seen in Figure 4.1. The basic operating principle of NoGap is that one or more facets uses a C++ API to construct a number of ASTs that is fed to the NoGap compiler. The NoGap compiler processes the ASTs and constructs the NoGap$^{CD}$. Finally a number of spawners can read the NoGap$^{CD}$ and produce something useful, like synthesizable SystemVerilog code (Chapter 13), an assembler (Chapter 14), or a cycle accurate simulator (Chapter 15).

One particular facet has been developed in conjunction with the development of NoGap-Core, and that is NoGap$^{CL}$ which is further described in Section 9.
Figure 4.1: NoGap system architecture

Figure 4.2: NoGap\textsuperscript{CL} Flow
Chapter 4: System Architecture

\textit{NoGap}^{CD} is a collection of PUs, more information about the PU can be found in Chapter 5. There will be at least as many PUs in \textit{NoGap}^{CD} as there are FUs in the design. There can however be more PUs generated during the compilation process. Each PU can in turn be divided into three important parts; \textit{Mase}, \textit{Mage} and \textit{Castle}.

\textit{Mase} is an annotated data flow graph describing all spatial and temporal connections between FUs for all operations defined on this particular data path.

\textit{Mage} is a somewhat modified and optimized version of the original AST. \textit{Mage} FUs are seen as black boxes in the \textit{Mase} graph. \textit{Mage} FUs can be seen as operators in \textit{Mase}. For example, simple FUs such as adders or multipliers, but also more complex \textit{Mase} FUs can be used, such as ALUs or a complete MAC unit. It is up to the designer to decide how complex the \textit{Mage} units shall be.

\textit{Castle} contains the information needed to construct instruction decoders, e.g. specifying instruction formats, source-, and destination operands. In effect this is all other information stored in the PU.

Worth noting is that the input to \textit{NoGap} compiler is always AST graphs. However if the AST specifies a data path FU the PU will get its \textit{Mase} part generated for it and then it is considered to be a \textit{Mase} PU. Otherwise the PU will be regarded as a \textit{Mage} PU. In a system designed with \textit{NoGap}, the hierarchical relationship between \textit{Mase} and \textit{Mage} PUs is that of a tree, where the \textit{Mase} PUs never can be leaf vertices. This is illustrated in Figure 4.3, where boxes represent \textit{Mase} FUs and ovals represents \textit{Mage} FUs.

A top level flow chart \textit{NoGap}'s compilation and generation flow, is shown in Figure 4.4. This flow chart start with a \textit{NoGap}^{CL} description of some hardware unit, \textit{NoGap}^{CL} is further described in Chapter 9. The first step is the parsing step, in which the \textit{NoGap}^{CL} parser, generates a number of ASTs, these ASTs are also called \textit{Mages}. At the same time one PU, for each FU defined in the input file, is created, more informa-
Chapter 4: System Architecture

Figure 4.3: Mase and Mage relationship

The next step is the compilation stage, in this step a number of generators processes the Mages, more details about generators are presented in Section 12.2. During the compilation stage, additional data will be written to the PUs and new PUs might be generated. The final stage in the flow is the generation stage. In this stage another set of generators produce some useful outputs, e.g. synthesizable SystemVerilog code, cycle accurate simulators, or assemblers. A collection of generators that produces a concrete output is called a spawner. More details about the currently available spawners is presented in Part IV

4.2 Action Register

There are a number of cases in the NoGap C++ code where it might have seem natural to use a polymorphic solution, e.g. having a print_verilog method in each Mase Vertex ($V_{Mase}$). However this would cause a
Chapter 4: System Architecture

Figure 4.4: NoGap system flow
lot of dependencies within the system. All \( V_{\text{Mas}} \) would have to be modified as soon as a new generator is constructed. This is not desirable as the \( V_{\text{Mas}} \)'s should not be aware of how they are used. Instead the design opted for, is to dynamically bind certain functionality to types during run time. To facilitate this dynamic polymorphism the \texttt{utils::ActionRegister} template was created. The details of this template is outside the scope of this thesis but a usage example is presented in Listing 4.1. The helper class \texttt{PortPrint}, declared on line 1–21 is used as an example. A struct, \texttt{Arg}, is declared to carry the arguments. Three types, \texttt{return_type}, \texttt{argument_type}, and \texttt{function_ptr_type}, required by \texttt{ActionRegister} is declared on line 9–11. Finally, two usable methods are declared on line 14–18. Later in the code, on line 23–27, a \texttt{ActionRegister} is defined and later the two print functions, \texttt{slice_print} and \texttt{control_print}, are bound to the two types \texttt{SlicePortInfo} and \texttt{ControlPortInfo}, respectively. Finally, on line 29, \texttt{act} method of the \texttt{print_action} action register is called, the \texttt{act} method which will resolve the type of \texttt{port_data} and execute the corresponding registered function. A function argument is also passed using the \texttt{PortPrint::Arg} struct.

### 4.3 The Joust architecture

The Judgment and Operation Unified STructure (Joust) architecture concept stems from a realization that there is no inherent difference between what traditionally is called a data path and a control path. Both the control and data paths move and transform data. The exact movement and transformations are controlled from an external entity. In a traditional, single issue RISC machine, the external entity for the data path is the control path and the external entity for the control path is the data path. The problem with the traditional data path/control path partitioning is that it does not in a consistent way encapsulates
Listing 4.1: Action register usage example

```cpp
class PortPrint
{
public:
    struct Arg
    {
        Arg(std::ostream & ost, const std::string & name);
    }

    typedef std::ostream & return_type;
    typedef const Arg & argument_type;
    typedef return_type (* function_ptr_type)
    (const arch_data::InstructionPortInfo &, argument_type);

public:
    static return_type
    control_print(const arch_data::InstructionPortInfo & ipi,
                  argument_type arg);

    static return_type
    slice_print(const arch_data::InstructionPortInfo & ipi,
                argument_type arg);

    ...;
    utils::ActionRegister<PortPrint> print_action;
    print_action.register_operation
    <arch_data::SlicePortInfo>(PortPrint::slice_print);
    print_action.register_operation
    <arch_data::ControlPortInfo>(PortPrint::control_print);
    ...
    print_action.act(port_data,
                     PortPrint::Arg(branch_code,
                                    "instr")
                     );
}
```
feedback paths from the data path into the control path as needed for e.g. conditional jumps. In the traditional view instructions only flow from the control path to the data path. Closer to the truth is that there are also instructions flowing from the data path to the control path, for example condition codes for jumps can be viewed as instructions for the control path. In NoGap terminology transformation paths communicate with other transformation paths via control bridges, this is depicted in Figure 4.5, where the grey elements constitute control bridges. Note that what traditionally is called the control path is split into a transformation path and a control bridge.

Figure 4.5: Transformation path and bridge
One of the most untruthful things possible, you know, is a collection of facts, because they can be made to appear so many different ways.

---

Karl A. Menninger

### 5.1 Parse Units

The PU is a very central concept in NoGap, a parse unit can roughly be said to correspond to an FU. A PU contains all relevant Intermediate Representation (IR) for one FU. Listing 5.1 shows an excerpt of the PU class. It can be seen that a PU contains a number of different member variables used for the IR. Table 5.1 give a short explanation for these member variables. All, except `ast_m`, of the main information carrying types are wrapped in the `generator::GeneratedData` container, which is explained in Section 12.6.
namespace parser
{
    ...

class ParseUnit : public policy::OutStreamable
{
    friend class ParseData;
    ...
    ast::Graph ast_m;
    ...

generator::GeneratedData<ncg::Graph> mase_m;
generator::GeneratedData<SymbolTable> symbol_table_m;
generator::GeneratedData<arch_data::PcFsm> pc_fsm_m;
generator::GeneratedData<VarExprDepGraph> var_expr_dep_graph_m;
generator::GeneratedData<VarExprDepGraph> mage_dep_graph_m;
generator::GeneratedData<ClauseAccessRules> clause_access_rules_m;
generator::GeneratedData<Pipelines> pipelines_m;
generator::GeneratedData<ControlPath> control_path_m;
generator::GeneratedData<ncg::PipelineClassifier> operation_classes_m;

generator::GeneratedData<arch_data::InstructionDecoder> instruction_decoder_m;


generator::GeneratedData<ncg::InstructionTable> instruction_table_m;
generator::GeneratedData<bool> needs_clock_input_m;
generator::GeneratedData<SSP_Partition> ssp_partition_m;
...
}
}
### Table 5.1: PU IR types explained

<table>
<thead>
<tr>
<th>Line</th>
<th>See Section</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>6</td>
<td>The Mage graph</td>
</tr>
<tr>
<td>10</td>
<td>7</td>
<td>The Mase graph</td>
</tr>
<tr>
<td>11</td>
<td>8</td>
<td>The symbol table</td>
</tr>
<tr>
<td>12</td>
<td>5.2</td>
<td>The Program Counter-Finite State Machine (PC-FSM) description</td>
</tr>
<tr>
<td>14</td>
<td>5.3</td>
<td>The mage dependency graph</td>
</tr>
<tr>
<td>15</td>
<td>5.4</td>
<td>The clause access rules</td>
</tr>
<tr>
<td>16</td>
<td>5.5</td>
<td>The pipelines</td>
</tr>
<tr>
<td>17</td>
<td>5.6</td>
<td>The control path</td>
</tr>
<tr>
<td>19</td>
<td>5.7</td>
<td>The operation classed</td>
</tr>
<tr>
<td>22</td>
<td>5.8</td>
<td>The instruction decoder</td>
</tr>
<tr>
<td>24</td>
<td>5.9</td>
<td>The instruction table</td>
</tr>
<tr>
<td>25</td>
<td>5.10</td>
<td>The clock input need</td>
</tr>
<tr>
<td>26</td>
<td>5.11</td>
<td>The Source Sink Pass (SSP) partition</td>
</tr>
</tbody>
</table>

*a*in Listing 5.1
5.2 PC-FSM

A PU can hold information about one synthesized PC-FSM, this will be used if the PC-FSM was constructed using a PC-FSM template. This description is the synthesized result, presented in a form that should be easy to utilize when designing a spawner. This section only outlines the representation of the PC-FSM in NoGap\textsuperscript{CD}. A more detailed description of the PC-FSM can be found in Section 9.23.1.

The PC-FSM representation is a BGL graph, its $V$s does not contain any important data more than the state name, while its $E$s contains two \texttt{Mage} Vertex ($V_{\texttt{Mage}}$)-descriptors \texttt{condition.m} and \texttt{action.m}. Both of these $V_{\texttt{Mage}}$-descriptors points to the root of an expression sub-tree, for the associated condition and action respectively, in the \texttt{Mage} graph.

5.3 \texttt{Mage} Dependency Graph

The \texttt{Mage} dependency graph shows the dependencies between variables in a \texttt{Mage} FU. More details about the \texttt{Mage} dependency graph can be found in Section 6.4

5.4 Clause Access Rules

Clauses can be named and used to model instruction controlled FUs, this is what is referred to as Dynamic Clause Selection (DCS), which is further explained in Section 11.2. However during the compilation processes NoGap will generate a collection of rules for how clauses should be accessed in an FU. A clause access rule describes what values input ports has to have to access a certain clause within the FU. This Section outlines how the clause access rules are represented in NoGap\textsuperscript{CD}

Listing 5.2 shows the essential C++ code that define clause access rules. A single rule, \texttt{rule_t}, defined on line 4, is the actual rule type.
The collection of rules, \texttt{rules\_t}, defined on line 15, details what values a set of ports has to have to ensure that a particular clause in the FU is accessed. The \texttt{std::string} in the \texttt{rules\_t} type is the name of a clause. The \texttt{port\_rule\_t} maps the actual symbols, which should be input ports, to a clause access rule. All this means that more than one input port can be incorporated in a rule to access a particular clause. And that for each input port a range of values, rather than just one value, can be recorded as possible values for clause access. Normally just one of these values will be selected when designing a decoder, but having multiple possible values to choose from opens for the possibility to optimize the decoder in later stages.

Some examples of what clause access rules might look like is shown in Listing 5.12 on line 10–18. For example, the first rules says that to access the \%ADD clause, that the symbol \texttt{op\_i} (shown before |), needs to have a two bit value set to a value in the range of \texttt{00\_2} and \texttt{00\_2} (shown after |).

**Listing 5.2:** Essential clause access rule code

```cpp
1 class ClauseAccessRule : public policy::OutStreamable
2 {
3   ...
4   typedef std::vector<boost::numeric::interval<utils::BitNum>> rule_t;
5 private:
6     rule_t rule_m;
7   ...
8   }
9
class ClauseAccessRules : public policy::OutStreamable
10 {
11   ...
12   public:
13   typedef std::map<const Symbol*, ClauseAccessRule> port_rule_t;
14   typedef std::map<std::string, port_rule_t> rules_t;
15 private:
16     rules_t rules_m;
17   ...
18   }
19 }
```
Chapter 5: Parse Unit

5.5 Pipelines

The concept of pipelines and their use in NoGap, is discussed in Section 9.15 and Section 9.17. This Section will only outline how they are stored in the PU.

The essential code for the representation of pipelines is shown in Listing 5.3. As described in the Sections just referred to, a pipeline essentially consists of phases and stages. This is reflected in the Phase class, on line 1, and the Stage class, on line 9. The Phase class records the name and timing of a phase and the Stage class records the name and type of a stage. All phases will be stored in a phase pool and the stages in a stage pool. They are then accessible by their names from these pools.

The actual pipeline is represented by the Pipeline class, on line 17. The pipeline stores the set of phases and stages used, if the pipeline is the extension of another pipeline, and the name of the pipeline. The stages and phases are only stored by name, to be retrieved from the stage and phase pool.

All pipelines in the PU is stored in the Pipelines class, seen on line 32. This class collects all the individual pipelines into a common class. The phase and stage pools are members of this class.
Listing 5.3: Essential pipelines code

class Phase : public policy::OutStreamable
{
    private:
        std::string name_m;
    boost::optional<arch_data::time_t> timing_m;
    ...
};

class Stage : public policy::OutStreamable
{
    ...
    std::string name_m;
    Type::enum_t type_m;
    ...
};

class Pipeline : public policy::OutStreamable
{
    ..., public:
        typedef std::set<std::string> phases_t;
        typedef std::set<std::string> stages_t;
    ...
    private:
        phases_t phases_m;
        stages_t stages_m;
        boost::optional<const Pipeline&> base_pipeline_m;
        std::string name_m;
        ...
};

class Pipelines : public policy::OutStreamable
{
    ...
    public:
        typedef std::set<Pipeline*> pipelines_t;
        typedef std::multimap<std::string, Pipeline*> pipe_names_t;
        typedef PE_Pool<Phase> phase_pool_t;
        typedef PE_Pool<Stage> stage_pool_t;
    private:
        pipelines_t pipelines_m;
        pipe_names_t pipe_names_m;
    public:
        ...
    phase_pool_t phase_pool_m;
    stage_pool_t stage_pool_m;
    ...
};
5.6 Control Path

The control path together with the operation classes (Section 5.7), the instruction decoder (Section 5.8), and the instruction table (Section 5.9) contains the information needed to synthesize control paths. As can be seen in Listing 5.4, the ControlPath stores information about operations using the OperationInfo class.

Listing 5.4: Essential control path code

```cpp
class ControlPath : public policy::OutStreamable
{
    ...

    typedef std::map<std::string,
                     boost::shared_ptr<arch_data::OperationInfo> >
                     operation_info_t;

    private:
    operation_info_t operation_info_m;
    ...
};
```

5.6.1 Operation Info

The operation info class, shown in Listing 5.5, is used to store all needed information about operations in a Mase FU. In order to understand the operation info description the three classes, SubOperation, CanonicalOpName, and OpCode must first be explained. This section will only give a brief introduction to OperationInfo, there are more parts to this class, but they are outside the scope of this thesis.

SubOperation, declared on line 1–7, specifies the use of a unique FU port at a unique time in a Mase, e.g. rf.wr_en_i@3.

CanonicalOpName, declared on line 9–16, specifies what clause should be accessed for one SubOperation, e.g. [rf.wr_en_i@3](%WRITE).

OpCode, declared on line 18–23, is a collection of CanonicalOpNames.
Chapter 5: Parse Unit

It stores one of the possible instructions created from one Mouse-operation, e.g. [rf.wr_en_i@3](%WRITE)[t1.op_i@1](%ADD).

OperationInfo, declared on line 25–35, collects all possible OpCode\(\text{s}\) and thus specifies all possible instructions that can be created from one Mouse-operation.

**Listing 5.5: Essential operation info code**

```cpp
class SubOperation : public policy::OutStreamable{
private:
    std::string fu_name_m;
    std::string port_name_m;
    arch_data::time_t timing_m;
    ...
};

class CanonicalOpName : public policy::OutStreamable{
public:
    typedef std::vector<std::string> use_names_t;
private:
    SubOperation fu_inst_name_m;
    use_names_t use_name_m;
    ...
};

class OpCode : public policy::OutStreamable{
    typedef std::vector<CanonicalOpName> op_code_t;
private:
    op_code_t op_code_m;
    ...
};

class OperationInfo : public policy::OutStreamable{
public:
    typedef std::vector<OpCode> opcodes_t;
    typedef std::map<std::string,utils::BitNum> constant_assignments_t;
    ...
protected:
    std::string operation_name_m;
    opcodes_t opcodes_m;
    constant_assignments_t constant_assignments_m;
    ...
};
```
5.7 Operation Classes

The operation classification is further described in Section 11.6.1. Presented here is a description of the operation classes representation. Listing 5.6 shows the essential code needed to understand how operation classes are represented in NoGap.

The pipeline usage vector, declared on line 8–20, is the core part of the operation classes representation. Section 11.6.2 describes the pipeline usage vector in detail.

The operation classification table, on line 45, is an unordered map that associates each pipeline usage vector with a set of operations. This means that this class stores the merged pipeline usage vectors together with a set of operations that has that specific pipeline usage vector.
Listing 5.6: Essential operation classification code

```c++
class OperationClass : public policy::OutStreamable
{
private:
  unsigned int class_id_m;
...}

class PipelineUsageVector : public policy::OutStreamable
{
public:
  typedef boost::optional<arch_data::time_t> phase_t;
private:
  typedef std::map<ncg::types::vdesc_t,phase_t> usage_vector_t;
public:
  typedef usage_vector_t::const_iterator const_iterator;
  typedef usage_vector_t::value_type value_type;
private:
  usage_vector_t usage_vector_m;
...}

class OpClassInfo : public policy::OutStreamable
{
public:
  typedef std::set<OperationClass> class_id_set_t;
private:
  class_id_set_t class_id_set_m;
...}

class PipelineClassifier : public policy::OutStreamable,
  boost::noncopyable
{
private:
  typedef std::set< const arch_data::OperationInfo*> operation_set_t;
public:
  typedef OperationClass class_id_t;
  typedef std::map<arch_data::time_t,OpClassInfo> fu_class_config_t;
  typedef unordered_map<OperationInfo*,
                        std::map<OperationClass*,
                                 std::vector<OpClassInfo*> >
                        op_class_table_t;
private:
  typedef std::map< const PipelineUsageVector*,
                   class_id_t> class_id_map_t;
private:
  op_class_table_t op_class_table_m;
  class_id_map_t class_id_map_m;
...}
```
5.8 Instruction Decoder

The InstructionDecoder class stores information about how instruction decoders shall be generated. Listing 5.7 shows the essential code needed to understand how instruction decoders are represented. The key member is mase_port_info_m, on line 12. It is a table, that for each output port of a decoder, lists control information for each op code. There are two types of data in the table CONTROLL and SLICE. CONTROLL denotes that operation codes needs to be converted to micro architecture control codes. SLICE denotes that parts of the instruction word shall be sliced and sent to the data path verbatim, this is also referred to as an immediate field. An example of such control table is shown in Listing 5.8, on line 2–12. For example line 2 states that the output port alu_op_i shall generate and output control signals, e.g. output 002 if the op-code is 0, while port imm_a_o shall slice out bit 11 to 8 from the instruction word for all operation codes, note that different operation codes can cause different parts of the instruction word to be used for the immediate data.

The other member shown in Listing 5.7, op_code_slices_m, on line 13 stores information about which bits in the instruction word is regarded as op-code bits. At the time of writing, only one continuous field in the instruction word can be used for the op-code.
Chapter 5: Parse Unit

Listing 5.7: Essential instruction decoder code

```cpp
class InstructionDecoder : public policy::OutStreamable
{
private:
    typedef std::map<std::size_t,
        boost::shared_ptr<InstructionPortInfo>> opcode_port_info_t;
    typedef std::map<std::string, opcode_port_info_t> mase_port_info_t;
    typedef std::vector<boost::shared_ptr<InstructionPortInfo>> op_code_slices_t;

    private:
        mase_port_info_t mase_port_info_m;
        op_code_slices_t op_code_slices_m;

        ..
}
```

Listing 5.8: Instruction decoder data

```plaintext
MASE PORT INFO:
( alu_op_i :(0: CONTROLL : 2<16 > u0 ), (1: CONTROLL : 2<16 > u2 ),
(2: CONTROLL : 2<16 > u3 ), (3: CONTROLL : 2<16 > u1 ))

( imm_a_o :(0: SLICE : [11:8]), (1: SLICE : [11:8]),

( imm_b_o :(0: SLICE : [7:4]), (1: SLICE : [7:4]),
(2: SLICE : [7:4]), (3: SLICE : [7:4]))

( imm_c_o :(0: SLICE : [3:0]), (1: SLICE : [3:0]),
(2: SLICE : [3:0]), (3: SLICE : [3:0]))

( oc_i :(0: CONTROLL : 1<16 > u1 ), (1: CONTROLL : 1<16 > u1 ),
(2: CONTROLL : 1<16 > u1 ), (3: CONTROLL : 1<16 > u1 ))

( regfile_wr_en_i :(0: CONTROLL : 1<16 > u1 ), (1: CONTROLL : 1<16 > u1 ),
(2: CONTROLL : 1<16 > u1 ), (3: CONTROLL : 1<16 > u1 ))

OP CODE SLICES:
SLICE: [14:12]
```
5.9 Instruction Table

The instruction table is a table of the generated instruction formats, instruction formats is explained in Section 5.9.1. Its construction is straightforward and the essential code is shown in Listing 5.9. The InstructionTable class, on line 8 is composed of a number of InstructionTableElements.

Listing 5.9: Essential instruction table code

```cpp
struct InstructionTableElement {
    std::string op_name_m;
    arch_data::InstructionFormat instruction_format_m;
    const arch_data::OperationInfo* op_inf_m;
    boost::optional<std::size_t> op_code_m;
};

class InstructionTable : public policy::OutStreamable {
    typedef std::vector<InstructionTableElement> ins_format_coll_t;
    public:
    ins_format_coll_t instructions_m;
};
```

5.9.1 Instruction Format

NoGap synthesizes the instruction format so that it always consists of an op-code, padding bits, and immediate data fields. An immediate field in NoGap denotes an instruction field that will be copied verbatim from the instruction word into the data path, e.g. register addresses will be considered immediate data as well as what is traditionally considered immediate data in an instruction. So far NoGap can only handle fixed size instructions, for this reason padding bits are used to pad shorter instructions. An example of this general instruction format is shown in Figure 5.1.
NoGAP will automatically synthesize the needed instruction formats for a particular data path/processor. Therefore each new processor will probably have an unique instruction format and instruction set, although all instructions will have some commonalities.

![ instruction format example ]

5.10 Needs Clock Input

The need clocks input says if this PU needs to have clock and reset inputs. It is a boolean, if its true, clock and reset inputs are needed if its false no clock or reset inputs are needed.

5.11 SSP Partition

The Source Sink Pass (SSP) partition class records witch ports belong to the source, sink, and pass partitions. The SSP concept is further explained in Section 15.3. Listing 5.10 shows part of the SSP_Partition class. As can be seen it is simply composed of three VmMap sets, one for each of the SSP partitions.

Listing 5.10: Essential SSP partition code

```cpp
1 class SSP_Partition : public policy::OutStreamable
2 {
3  ...
4 private:
5   ast::vertex_set_t sources_m;
6   ast::vertex_set_t sinks_m;
7   ast::vertex_set_t passes_m;
8  ...
9 );
```
Listing 5.11: Transcript of a Mase PU

1  ===== Parse Unit: data_path (DATA_PATH) =====
2  AST : data_path
3  MASE : data_path
4  PIPELINES:
5  Group phases: {DE [0]} {EX [2]} {OF [1]} {WB [3]}
6  Group stages: {ff-register}
7  pipe=
8  Base Pipe: *
9  Phases: {DE [0]} {EX [2]} {OF [1]} {WB [3]}
10 Stages: {ff-register}
11 NEEDS CLOCK: YES
12 SWITCH CODING:
13 Switch codings:
14
15 CLAUSE ACCESS RULES:
16
17 RELATED UNITS:
18 Related as: DECODER
19 dec
20 SYM. TAB.:
21 Clause Names:
22
23 ===== Scope (global) [*] =====
24 ===== Symbols =====
25 0: data_path: N6parser3ast4info16FuImplementationE (*) <>>
26  [untitled]
27 === Child Scopes ===
28 ===== Scope (untitled) [global] =====
29 ===== Symbols =====
30 0: DE: N6parser3ast4info16PhaseDeclarationE (*) <>> [*]
31 1: EX: N6parser3ast4info16PhaseDeclarationE (*) <>> [*]
32 2: OF: N6parser3ast4info16PhaseDeclarationE (*) <>> [*]
33 3: WB: N6parser3ast4info16PhaseDeclarationE (*) <>> [*]
34 4: alu: N6parser3ast4info13FuDeclarationE (alu) <>> [*]
35 5: alu_op: N6parser3ast4info9OperationE (pipe) <>> [untitled]
36 6: dat_o: N6parser3ast4info9OperationE (pipe) <>> [untitled]
37 7: dec: N6parser3ast4info21FuTemplateDeclarationE (decoder_spec)
38  <>> [*]
39 8: ff: N6parser3ast4info5StageE (*) <>> [untitled]
40 9: instr_i: N6parser3ast4info11TemplateArgE (*) <>> [*]
41 10: load: N6parser3ast4info9OperationE (pipe) <>> [untitled]
42 11: move: N6parser3ast4info9OperationE (pipe) <>> [untitled]
43 12: op_i: N6parser3ast4info4PortE (input) <>> [*]
44 13: out: N6parser3ast4info9OperationE (pipe) <>> [untitled]
Chapter 5: Parse Unit

14: pipe: N6parser3ast4info8PipelineE (*) <-> [untitled]
15: regfile: N6parser3ast4info13PuDeclarationE (regfile) <-> [*]

--- Child Scopes ---

+++++ Scope (untitled) [untitled] ++++

--- Symbols ---

--- Child Scopes ---

+++++ Scope (untitled) [untitled] ++++

--- Symbols ---

--- Child Scopes ---
Listing 5.12: Transcript of a Mage PU

1  === Parse Unit: alu (MAGE) ====
2  AST : alu
3  MASE : *
4  PIPELINES: *
5  NEEDS CLOCK: NO
6  SWITCH CODING:
7  Switch codings:
8  0x6621e0 Switch codes:
9  <0x65aa70,0>:<0x65c830,1>:<0x65e470,2>:<0x6600c0,3>
10  CLAUSE ACCESS RULES:
11  % ADD
12  op_i : N6parser3ast4info4PortE (input) <-> [*] | [2<16>u0,2<16>u0]
13  % AND
14  op_i : N6parser3ast4info4PortE (input) <-> [*] | [2<16>u2,2<16>u2]
15  % OR
16  op_i : N6parser3ast4info4PortE (input) <-> [*] | [2<16>u3,2<16>u3]
17  % SUB
18  op_i : N6parser3ast4info4PortE (input) <-> [*] | [2<16>u1,2<16>u1]
19  RELATED UNITS:
20  *
21  SYM. TAB.:
22  Clause Names:
23  0x65bd20 % ADD:0x65f710 % AND:0x661360 % OR:0x65dac0
24 _scope (global) [*] ======
25  ===== Symbols =====
26  0: alu: N6parser3ast4info16FuImplementationE (*) <-> [untitled]
27  +++== Scope (untitled) [global] ======
28  ===== Symbols =====
29  0: a_i: N6parser3ast4info4PortE (input) <-> [*]
30  1: b_i: N6parser3ast4info4PortE (input) <-> [*]
31  2: op_i: N6parser3ast4info4PortE (input) <-> [*]
32  3: res_o: N6parser3ast4info4PortE (output) <-> [*]
33  +++== Scope (untitled) [untitled] ======
34  ===== Symbols =====
35  +++== Scope (untitled) [untitled] ======
36  ===== Symbols =====
37  +++== Scope (% ADD) [untitled] ======
38  ===== Symbols =====
39  +++== Scope (untitled) [untitled] ======
40  ===== Symbols =====
41  +++== Scope (% ADD) [untitled] ======
42  ===== Symbols =====
43  +++== Scope (untitled) [untitled] ======
44  +++== Scope (untitled) [untitled] ======
Chapter 5: Parse Unit
Blood Mage: "Hi, my name is Roy. I'm a magic addict."
Group: "Hi Roy."

The Mage description is an annotated graph, and is therefore often referred to as just Mage graph. Both forms will be used interchangeably throughout this thesis.

6.1 Introduction

The Mage graph is a Directed Acyclic Graph (DAG) that describes an AST, an example of a Mage graph is shown in Figure 6.1. There are a number of different Mage Edge ($E_{\text{Mage}}$) types, further described in Section 6.3, and a number of different $V_{\text{Mage}}$ types, further described in Section 6.2.

The input to the NoGap compiler are ASTs, however after the compilation some new PUs might have been created and yet other PUs might have had Mage graphs generated for them. However there will still be
Figure 6.1: Mage example
a number of PUs that only contains the AST, these FUs are considered to be Mage FUs.

The Mage graph is implemented as a the class parser::ast::Graph, which is composed of, as its most important member variable, a Boost graph of type parser::ast::ast_t, as seen in Listing 6.1 line 16. The definition of ast_t is shown on line 5–10. The definition shows, among other things that, on line 8, the parser::ast::info::NodeInfo type is used for V_{Mage} data. And that, on line 9, the parser::ast::info::EdgeInfo type is used for E_{Mage} data.

Listing 6.1: Mage graph BGL type

```cpp
namespace parser
{
namespace ast
{

typedef boost::adjacency_list<boost::listS,
                           boost::listS,
                           boost::bidirectionalS,
                           info::NodeInfo,
                           info::EdgeInfo,
                           boost::listS> mcg_t;

class Graph
{

  ast_t theGraph_m;

};

}
}
```

Both the parser::ast::info::NodeInfo and the parser::ast::info::EdgeInfo classes are wrapper classes for the underlying types, parser::ast::info::Statement and
The wrapper class approach is a legacy design choice, and is subject to change in the future.

6.2 Mage Types

There are a number of different Mage types, each Mage is a class and the class diagram for all Mage types is shown in Figure 6.2, although the text is very small, it can at least be seen that there are many different Mage types, this section will only outline the most important aspects of this class hierarchy, since giving detailed explanations for each class would make this thesis far too long.

As can be seen in Figure 6.2 the Statement class is the root of the class hierarchy. Its declaration is shown in Listing 6.2. There are four important methods in this class. The first three; childScope on line 19, namingNode on line 20, and typeNameNode on line 21, are all intended to be overloaded with methods that return a path to its child scope, naming node, and type name node, respectively. A child scope is the scope associated with a construct statement. A naming node is a Mage containing an identifier associated with the statement node, giving the statement a name. A type name node is a Mage containing an identifier for the name of the type associated with the current statement. These functions return a list of integers, these integers are then used to traverse the AST by using the Mage order numbers, the three functions are collectively called AST navigation functions. The fourth method on line 22 is a delegation method for the template method with the same name on line 11, the delegation method is used to so that the correct type can be resolved, even using polymorphism. The buildSymbol method uses the childScope, namingNode, and typeNameNode methods to build a symbol of itself that can be inserted into the symbol table. More details about the symbol creation process is found in Section 8.2.
Figure 6.2: $\mathcal{V}_{\text{Mage}}$ class diagram
Listing 6.3 shows an example of how a $V_{Mage}$ is declared, it utilizes a macro to ease the creation of new $V_{Mage}$ types. The macro is called on line 1, then the three AST navigation methods are overloaded to guide the symbol creation process. Finally on line 11 the delegation to the template method is done. The need for this delegation method stems from that only the Instruction class itself knows what type it is, therefore only a delegation function can call the template function with the correct types.

Listing 6.2: Statement declaration

```cpp
class Statement : public policy::OutStreamable,
                  public property::PolyNewClonable<Statement>
{
  // NAMESPACE POLY_NEW_CLONABLE(Statement);
  public:
    typedef std::list<int> pathList_t;
    typedef boost::optional<pathList_t> propertyEdge_t;
    typedef boost::optional<Symbol> buildSymbol_t;

  protected:
    std::string statementName_m;
    template<typename T>
    static buildSymbol_t buildSymbol(const T& t,
                                      const ast::Graph& graph,
                                      const vdesc_t& vertex,
                                      bool is_constant=false);

  public:
    Statement(const std::string& name);
    virtual ~Statement();
    virtual const propertyEdge_t childScope() const
    virtual const propertyEdge_t namingNode() const
    virtual const propertyEdge_t typeNameNode() const
    virtual buildSymbol_t buildSymbol(const ast::Graph&,
                                       const vdesc_t& vertex)
                          const
    virtual buildSymbol_t buildConstantSymbol(const ast::Graph&,
                                             const vdesc_t& vertex)
                          const
    virtual std::ostream& print(std::ostream& ost) const;
    virtual std::ostream& dotLabel(std::ostream& ost) const;
};
```
6.3 \(\mathcal{E}_{\text{Mage}}\) Types

The AST has types associated with its edges the base \(\mathcal{E}_{\text{Mage}}\) type is the \texttt{Edge} class, its declaration is shown in Listing 6.4. The actual \(\mathcal{E}_{\text{Mage}}\) type has no meaning in the current design, although an important property of an \(\mathcal{E}_{\text{Mage}}\) is its order, on line 7. As seen in Figure 6.1 all out edges for a node has an unique number. This number can be used to navigate the graph.

6.4 \texttt{Mage} Dependency Graph

The \texttt{Mage} dependency graph is a graph of the dependencies between variables and comb/cycle blocks in a \texttt{Mage} FU. This Section outlines the representation of the \texttt{Mage} dependency graph in a PU. A detailed description of what a \texttt{Mage} dependency graph is, and how it is generated, can be found in Section 15.2.

The \texttt{Mage} dependency graph is stored in the \texttt{NoGap} as a graph
Listing 6.4: $E_{\text{Mage}}$ Declaration

```c++
class Edge : public policy::outStreamable,
public property::PolyNewCloneable<Edge>
{
NG_PROPERTY_POLY_NEW_CLONABLE(Edge);

private:
std::string name_m;
EdgeOrder order_m;

public:
bool pruned;
Edge(const std::string& name = "foo",
const EdgeOrder& order = EdgeOrder(0));
virtual ~Edge();
const std::string& name() const;
const EdgeOrder& order() const;
std::ostream& dot_label_text(std::ostream& ost) const;
std::ostream& print(std::ostream& ost) const;
Edge& operator++();
};
```

using BGL. Listing 6.5 shows the essential code for the $E_{\text{Mage}}$ dependency graph representation.

The NodeData subclass, on line 5, represents $V$ data. As can be seen a $E_{\text{Mage}}$ dependency graph $V$ can be of four different types; VARIABLE, EXPRESSION, COMB_BLOCK, and CYCLE_BLOCK representing variables, expressions, comb blocks, and cycle blocks, respectively. If the $V$ is of type EXPRESSION, the expr_root_m member, points to the root $V_{\text{Mage}}$ of the expression in question.

The EdgeData subclass, on line 14, represents $E$ data. As can be seen a $E_{\text{Mage}}$ dependency graph $E$ can be of three different types CYCLE_ASSIGN, COMB_ASSIGN, or OPERAND. CYCLE_ASSIGN $E$'s points to variables that are assigned in a cycle block. COMB_ASSIGN $E$'s points to variables that are assigned in comb blocks. And OPERAND $E$'s points to variable that are operands in either comb or cycle blocks.

The actual BGL graph is declared on line 29.
Listing 6.5: Essential $\textsc{Mage}$ dependency graph code

```cpp
class VarExprDepGraph : public policy::OutStreamable
{
  ...
  public:
  struct NodeData
  {
    enum node_type_t { VARIABLE, EXPRESSION, COMB_BLOCK, CYCLE_BLOCK};
    node_type_t node_type_m;
    std::string text_m;
    ...
  };

  struct EdgeData
  {
    enum edge_type_t { CYCLE_ASSIGN, COMB_ASSIGN, OPERAND};
    edge_type_t edge_type_m;
    ...
  };

typedef boost::adjacency_list<boost::listS,
    boost::listS,
    boost::bidirectionalS,
    NodeData,
    EdgeData,
    boost::listS> variable_deps_t;

  private:
  variable_deps_t variable_deps_m;
  ...
};
```
7

Mase

Just as treasures are uncovered from the earth, so virtue appears from good deeds, and wisdom appears from a pure and peaceful mind. To walk safely through the maze of human life, one needs the light of wisdom and the guidance of virtue.

Buddha

The Mase description is an annotated graph, and is therefore often referred to as just Mase graph. both forms will be used interchangeably throughout this thesis.

7.1 Introduction

The Mase graph is a directed graph that describes a data path. There are a number of different Mase Edge ($E_{\text{Mase}}$) types, further described in Section 7.3, and a number of different $V_{\text{Mase}}$ types, further described in Section 7.2.

The Mase graph is implemented as the class nco::Graph, which is composed of, as its most important member variable, a BGL graph of
type \texttt{ncg::types::ncg\_t}, as seen in Listing 7.1 line 16, who’s definition is shown in Listing 7.1 line 5–10. The definition shows, among other things, that on line 8, the \texttt{ncg::info::NodeInfo} type is used for $V_\texttt{Mase}$ data. And that, on line 9, the \texttt{ncg::info::EdgeInfo} type is used for $E_\texttt{Mase}$ data. For more details about BGL, the reader is referred to [46].

Both the \texttt{ncg::info::NodeInfo} and the \texttt{ncg::info::EdgeInfo} classes are wrapper classes for the underlying types, \texttt{ncg::info::VertexInfo} and \texttt{ncg::info::EdgeInfo}, respectively. The wrapper class approach is a legacy design choice, and is subject to change in the future.
Chapter 7: Mask

7.2 Mask Types

All Mask types are derived from the ncg::info::VertexInfo class, which in turn is derived from the ncg::info::Base virtual base class. The complete inheritance hierarchy for all possible Mask types is shown in Figure 7.1. A detailed description of all Mask types can be found in Table 7.1. Each node has a specific color, although some node types share color with other node types. Figure 7.2 and Figure 7.3 show these colors. It is not important to remember all these colors since all nodes have a textual description that should suffice. However, the colors are presented here as an informative reference.

Figure 7.1: VertexInfo inheritance hierarchy
Table 7.1: $\mathcal{M}$ase graph $\mathcal{V}_{\mathcal{M}}$ase types

<table>
<thead>
<tr>
<th>$\mathcal{V}_{\mathcal{M}}$ase Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fu</td>
<td>Central $\mathcal{V}_{\mathcal{M}}$ase in an FU cluster.</td>
</tr>
<tr>
<td>Decoder</td>
<td>Central $\mathcal{V}_{\mathcal{M}}$ase in a decoder FU cluster.</td>
</tr>
<tr>
<td>Source</td>
<td>Source $\mathcal{V}_{\mathcal{M}}$ase after SSP partitioning.</td>
</tr>
<tr>
<td>Sink</td>
<td>Sink $\mathcal{V}_{\mathcal{M}}$ase after SSP partitioning.</td>
</tr>
<tr>
<td>Pass</td>
<td>Pass $\mathcal{V}_{\mathcal{M}}$ase after SSP partitioning.</td>
</tr>
<tr>
<td>InlineExpression</td>
<td>Central $\mathcal{V}_{\mathcal{M}}$ase in a inline expression FU cluster.</td>
</tr>
<tr>
<td>PassWithoutPu</td>
<td>A pass $\mathcal{V}_{\mathcal{M}}$ase lacking a parse unit.</td>
</tr>
<tr>
<td>InsertedMux</td>
<td>Inserted multiplexers.</td>
</tr>
<tr>
<td>InsertedFlipFlop</td>
<td>Inserted Flip-Flops (FFs).</td>
</tr>
<tr>
<td>InputFlipFlop</td>
<td>Write part of FFs split during SSP partitioning.</td>
</tr>
<tr>
<td>OutputFlipFlop</td>
<td>Read part of FFs split during SSP partitioning.</td>
</tr>
<tr>
<td>PipelineClassControl</td>
<td>Pipeline class control</td>
</tr>
<tr>
<td>Port</td>
<td>Base class for ports.</td>
</tr>
<tr>
<td>SignalNode</td>
<td>Represent signals in the $\mathcal{V}_{\mathcal{M}}$ase</td>
</tr>
<tr>
<td>FuInPort</td>
<td>Input port $\mathcal{V}_{\mathcal{M}}$ase in an FU cluster</td>
</tr>
<tr>
<td>FuOutPort</td>
<td>Output port $\mathcal{V}_{\mathcal{M}}$ase in an FU cluster</td>
</tr>
<tr>
<td>GlobalIn</td>
<td>Input port for the entire $\mathcal{M}$ase graph</td>
</tr>
<tr>
<td>MuxControl</td>
<td>Output port for the entire $\mathcal{M}$ase graph</td>
</tr>
</tbody>
</table>
Figure 7.2: $\nu_{\text{Mase}}$ colors 1
Figure 7.3: $\mathcal{M}_{\text{ase}}$ colors 2
Chapter 7: 

7.3 $E_{\text{Mase}}$ Types

All $E_{\text{Mase}}$s are derived from the \texttt{ncg::info::EdgeInfo} class, which in turn is derived from the \texttt{ncg::info::Base} virtual base class. The complete inheritance hierarchy for all possible $E_{\text{Mase}}$ types is shown in Figure 7.4. A detailed description of all $E_{\text{Mase}}$s can be found in Table 7.2. The different $E_{\text{Mase}}$ types have different colors. Figure 7.5 shows these colors and also the information related to each $E_{\text{Mase}}$. It is not important to remember all these colors since all edges have a textual description that should suffice. However, the colors are presented here as an informative reference.

![Figure 7.4: ArcInfo inheritance hierarchy](image-url)

7.4 FU Cluster

An FU cluster consists of a central FU $V_{\text{Mase}}$ connected to a number of input $V_{\text{Mase}}$s and a number of output $V_{\text{Mase}}$s. In an FU cluster, the input ports are source $V_{\text{Mase}}$s of the central $V_{\text{Mase}}$ and the output
### Table 7.2: Mase graph $\mathcal{E}_{\text{Mase}}$ types

<table>
<thead>
<tr>
<th>Edge Type</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>InternalFuArc</td>
<td>Connects $\mathcal{V}_{\text{Mase}}$ within an FU cluster.</td>
</tr>
<tr>
<td>OutsideArc</td>
<td>Connects $\mathcal{V}_{\text{Mase}}$ outside FU clusters.</td>
</tr>
<tr>
<td>BypassingArc</td>
<td>Connects $\mathcal{V}_{\text{Mase}}$ involved in bypassing logic.</td>
</tr>
<tr>
<td>ClassControlArc</td>
<td>Part of the class control logic in the pipeliner, further described in Chapter 11.</td>
</tr>
<tr>
<td>DirectArc</td>
<td>Has no delay or instruction information, forwards data to another $\mathcal{V}_{\text{Mase}}$.</td>
</tr>
<tr>
<td>ControlArc</td>
<td>Used for connections to FU control input ports.</td>
</tr>
<tr>
<td>InsertedMuxControlArc</td>
<td>Connects from multiplexer control $\mathcal{V}<em>{\text{Mase}}$ to the controlled multiplexer $\mathcal{V}</em>{\text{Mase}}$.</td>
</tr>
</tbody>
</table>
Figure 7.5: $E_{\text{Mase}}$ colors
ports are the target \( \mathcal{V}_\text{Mase} \) of central \( \mathcal{V}_\text{Mase} \). An example of an FU cluster is depicted in Figure 7.6. There are a number of different types of FU clusters, these types are listed in Table 7.3. In principle the, only difference between the FU clusters, is the use of different types for the central FU \( \mathcal{V}_\text{Mase} \).

**Figure 7.6:** FU cluster example

<table>
<thead>
<tr>
<th>Central ( \mathcal{V}_\text{Mase} )</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Decoder</td>
<td>Used for decoders.</td>
</tr>
<tr>
<td>InlineExpression</td>
<td>Used for inline expressions.</td>
</tr>
<tr>
<td>Fu</td>
<td>Used for all other FUs.</td>
</tr>
</tbody>
</table>

**Table 7.3:** FU cluster types

7.5 **In-line Expression FU**

An in-line expression FU is an FU that is synthesized by the \texttt{NoGap} compiler directly from expressions in an operation description. This type of
FUs will be created in a \texttt{Mase} graph when an in-line expression has been used in an operation description, an example of in-line expression usage, is shown in Listing 9.11 on line 9,10, and 11. The in-line expression FU will be created as an FU cluster where the central $V_{\text{mase}}$ is of type \texttt{necg::info::InlineExpression}. The input port sizes are always set to the incoming bus size and the output port size is calculated from the expression being inlined.

The sizing assumptions used to calculate the output port size of in-line expressions are listed in Table 9.4.

\section*{7.6 Loops in the Mase}

Loops in a \texttt{Mase} graph are either Intra-Operation Loop (OL\texttt{Intra}), meaning that the loops are part of the data path for this operation, e.g. a MAC type operation would cause these loops. Or Inter-Operation Loop (OL\texttt{Inter}), meaning that the loops are not part of any one operation, rather the loops are due to architecture artifacts occurring when some operations are merged into a single data path. Figure 7.7 shows an example of an OL\texttt{Intra}. And Figure 7.8 shows an example of an OL\texttt{Inter}, where two operations, A and S are merged into one data path, A uses the adder first, then the subtracter, S uses the subtracter first and then the adder. Both of these cases are legal loops, since in the case of the OL\texttt{Intra} loop, it is broken by a register and in the case of the OL\texttt{Inter} loop, the synthesized control path guarantees that all multiplexer are set so as to break the otherwise combinational relationship.

\section*{7.7 Predefined Mase Transformations}

There are a number of predefined transformations implemented in the \texttt{Mase} graph class. This Section will give a brief explanation of these predefined transformations, a summary of the available transformations
can be seen in Table 7.4. In order to better illustrate the effect of the transformations, the simple Mase graph, presented in Figure 7.9, will be used as a starting point. The figure shows two input ports and two output ports that can be connected in five different ways, OP1–OP5.

<table>
<thead>
<tr>
<th>Method name</th>
<th>Explanation in</th>
</tr>
</thead>
<tbody>
<tr>
<td>Graph\ncg::Graph::combineEqualEdges()</td>
<td>Section 7.7.1</td>
</tr>
<tr>
<td>Graph\ncg::Graph::insertFlipFlops()</td>
<td>Section 7.7.2</td>
</tr>
<tr>
<td>Graph\ncg::Graph::combineEqualFlipFlops()</td>
<td>Section 7.7.3</td>
</tr>
<tr>
<td>Graph\ncg::Graph::insertMuxes()</td>
<td>Section 7.7.4</td>
</tr>
<tr>
<td>Graph\ncg::Graph::setWireNames()</td>
<td>Section 7.7.5</td>
</tr>
<tr>
<td>Graph\ncg::Graph::setEdgeSize_()</td>
<td>Section 7.7.6</td>
</tr>
</tbody>
</table>

### 7.7.1 Combine Equal Edges

Combines $E_{\text{Mase}}$ that are regarded as equal. Two $E_{\text{Mase}}$ are considered to be equal if they go between the same two $V_{\text{Mase}}$ and the traversal time is the same. The edges are then combined but the respective operation
and timing information retained and stored in the resulting, combined, $E_{\text{Mase}}$. Figure 7.10 shows the effect of running equal $E_{\text{Mase}}$ combination on the $\text{Mase}$ graph presented in Figure 7.9.

### 7.7.2 Insert Flip-Flops

Inserts FFs in accordance with the delay times (shown as $[a,b]$) on $E_{\text{Mase}}$. Figure 7.11 shows the effect of running FF insertion on the $\text{Mase}$ graph presented in Figure 7.10.

### 7.7.3 Combine Equal Flip-Flops

Combines FFs that are regarded as equal. Two FFs are considered to be equal if the same data will be latched in them. Figure 7.12 shows the effect of running equal FF combination on the $\text{Mase}$ graph presented in Figure 7.11.
Figure 7.10: Equal $\epsilon_{\text{Mase}}$ combination example
Figure 7.11: FF insertion example
Figure 7.12: Equal FFs combination example
7.7.4 Insert Multiplexers

Inserts multiplexers as needed to resolve multiple connections to the same port. Figure 7.13 shows the effect of running multiplexer insertion on the \text{Masc} graph presented in Figure 7.12.

7.7.5 Set Wire Names

Gives all \text{Masc} unique names that can be used for code generation by a spawner. The name of an edge is a function of the \text{Vasc} it comes from. This reflects how signal names are used in normal HDL, where the same signal name can be used as source data in a number of different places. Figure 7.14 shows the effect of running both \text{setWireNames()} and \text{setEdgeSize()} on the \text{Masc} graph presented in Figure 7.13. Note how the \text{Easc} data, previously named \textit{unnamed}, has been changed to useful names.

7.7.6 Set Edge Sizes

Sets the size of all \text{Easc}. The size is calculated based on sizing functions of the FUs in the \text{Masc} graph. This is also referred to as “dynamic bus and port sizing”, a detailed description of this topic is presented in Chapter 10. Figure 7.14 shows the effect of running both \text{setWireNames()} and \text{setEdgeSize()} on the \text{Masc} graph presented in Figure 7.13. Note how the \text{Easc} data, previously written as \{\} = \{\}, has been filled with values. In this case all values are numeric, symbolic sizes can also be used.
Figure 7.13: Multiplexer insertion example
Figure 7.14: Set edge sizes example
7.8 Connect Stalling FFs

Certain FFs can be constructed as stallable, the need for using stallable FFs over normal FFs, depends on the pipeline design. If stalling should occur stall control logic is synthesized and stall FFs gets a stall control port. The connect stalling FFs method is responsible for connecting the stalling control logic to the stallable FFs. Figure 7.15 shows the effect of connecting stalling FFs in a Mase graph (not related to the previous presented Mase graphs). Note how all FFs have been connected to a stall signal generator.

Figure 7.15: Stalling FF connection example
A painting is a symbol for the universe. Inside it, each piece relates to the other. Each piece is only answerable to the rest of that little world. So, probably in the total universe, there is that kind of total harmony, but we get only little tastes of it.

Corita Kent

The symbol table is responsible for recording and handling all parsed symbols. It is used for look up and validation of symbols during the compilation process.

8.1 Symbol Table Description

The symbol table is an hierarchical table of identifiers and it is created early on in the compilation process from the input ASTs. Figure 8.1 shows an Unified Modeling Language (UML) diagram for the symbol table. As can be seen, the symbol table is implemented as the class `parser::SymbolTable`, this class is composed of one `parser::Scope` objects. The `parser::Scope` object represents what in MoGapCL is called
a clause. Therefore the `parser::Scope` is composed of zero or more new `parser::Scope` objects, one for each scope in the current scope. The `parser::Scope` class is also composed of zero or more `parser::Symbol` objects. These are the symbols that is native to the current scope. A `parser::Symbol` object is a leaf in the symbol table hierarchy.

![Symbol table architecture](image)

**Figure 8.1:** Symbol table architecture

The essential code for the `parser::Symbol` class is shown in Listing 8.1. The notable member variables are as follows. `type_m` on line 6 which is the name of the C++ type used to represent the symbol type. It is found using the template method on line 20–25. `declaringVertex_m`, on line 7, represent the `Vertex` used as the root for the creation of this symbol. `typeName_m`, on line 8, represents the given name of this type, e.g. the type name of an FU is the name given to the FU when it is defined. `name_m`, on line 9, is the name if a particular symbol, also called an identifier. `value_m`, on line 10, can be set to a numeric value, either if the symbol represents a constant value or if the symbol has a default value. `constant_m`, on line 11, is set to true if the symbol represents a constant value.


Listing 8.1: Symbol table Declaration

```cpp
class Symbol : public policy::OutStreamable
{
public:
  typedef boost::optional<utils::BitNum> value_t;
private:
  std::string type_m;
  const ast::vdesc_t declaringVertex_m;
  boost::optional<std::string> typeName_m;
  const std::string name_m;
  value_t value_m;
  bool constant_m;
  const Scope* childScope_m;
public:
  Symbol(const std::string& name, const ast::vdesc_t& declaringVertex);
  ~Symbol();
  const ast::vdesc_t& declaringVertex() const;
  template<typename T>
  Symbol& setType()
  {
    type_m = typeid(T).name();
    return *this;
  }
  template<typename T>
  bool isType() const;
  boost::optional<const std::string&> typeName() const;
  Symbol& typeName(const std::string& tn);
  const std::string& name() const;
  Symbol& setValue(const utils::BitNum& v);
  const value_t& value() const;
  Symbol& setConstant();
  bool isConstant() const;
  Symbol& setChildScope(const Scope* s);
  boost::optional<const Scope*> childScope() const;
  std::ostream& print(std::ostream& ost) const;
};
```
Symbols in the symbol table are created from the Mage graph using the factory method `parser::Statement::buildSymbol`. It uses the three navigation methods of each statement node, as described in Section 6.2, to find the right values for each of the member variables.

A textual representation of two symbol tables can be found in Listing 5.11 and Listing 5.12. The symbols are written according to the following format:

```
NAME: C++_TYPE_NAME (TYPE_NAME) <VALUE> [CHILD_SCOPE]
```

For example in Listing 5.11 on line 43, the `op_i` symbol is written, repeated here for the readers convenience:

```
op_i: N6parser3ast4info4PortE (input) <*> [*
```

The symbol name is `op_i`, the C++ type is `N6parser3ast4info4PortE`, it is of type `input`, it has no predefined constant value, and it has no child scope. The corresponding `NoGapCL` code from where this symbol is derived is shown in Listing 8.2. Note that the size information is not part of the symbol. Instead this is contained in the AST and can be found via the `declaringVertex_m` member.

### Listing 8.2: `op_i` definition

```cpp
input [7:0] op_i;
```

#### 8.2 Symbol Creation Process

As stated in Section 6.2, the `parser::Statement` class, or any derived class thereof, are part of the symbol creation process. The `parser::Statement::buildSymbol` method uses the `parser::Statement::childScope`, `parser::Statement::namingNode`, and `parser::Statement::typeNameNode` methods to build a symbol.
of itself that can be inserted into the symbol table. Listing 8.3 shows how the AST navigation methods are overloaded for an FU instantiation $V_{Mage}$. Looking at the overloaded AST navigation methods it is possible to follow the symbol creation process of the manual FU instantiations shown in Listing 8.4, which corresponding AST is shown in Figure 8.2. The overloaded $\texttt{childScope}$ method, on line 4, says that the child scope is found following the $E_{Mage}$ with order 2, thus finding the two respective child scopes sub trees, represented as triangles under the receptive FU instantiations. The overloaded $\texttt{namingNode}$ method, on line 5, says that the naming $V_{Mage}$ is found following the $E_{Mage}$ with order 1, thus finding $dp$ and $cp$ respectively. The overloaded $\texttt{typeNameNode}$ method, on line 6, says that the type name $V_{Mage}$ is found following the $E_{Mage}$ with order 0, thus finding $\texttt{data\_path}$ and $\texttt{control\_path}$ respectively. Listing 8.5 shows the resulting symbols, in this case the child scope had no name, hence the [untitled] entries.
Listing 8.3: FU instantiation AST navigation methods

```cpp
typedef std::list<int> pathList_t;
typedef boost::optional<pathList_t> propertyEdge_t;

const propertyEdge_t childScope() const { return pathList_t(1,2); }
const propertyEdge_t namingNode() const { return pathList_t(1,1); }
const propertyEdge_t typeNameNode() const { return pathList_t(1,0); }
buildSymbol_t
buildSymbol(const ast::Graph& graph, const vdesc_t& vertex) const { return Statement::buildSymbol(*this, graph, vertex); }
```

Listing 8.4: FU instantiations

```cpp
component data_path dp
{
  //Child scope for dp
}

component control_path cp
{
  //Child scope for cp
}
```

Listing 8.5: Symbol table excerpt for cp and dp

```cpp
cp: N6parser3ast4info15FuInstantiationE (control_path) <+ [untitled]
...
dp: N6parser3ast4info15FuInstantiationE (data_path) <+ [untitled]
```
Chapter 8: Symbol Table

Figure 8.2: Naming and type name ¥mage¥
9

NoGap Common Language

In making a speech one must study three points: first, the means of producing persuasion; second, the language; third the proper arrangement of the various parts of the speech.

Aristotle

9.1 Introduction to NoGap\textsuperscript{CL}

NoGap\textsuperscript{CL} is a the default NoGap facet used to construct the Maze, Mage and Castle descriptions. It is also a flexible language for general hardware description.
The main features of NoGap\textsuperscript{CL} are as follows:

1. Less micromanagement needed for control path construction.
2. No processor template restriction, providing more freedom for the designer.
3. Support of dynamic port sizes.
5. Pipeline stages can be adjusted easily, different pipelines can be defined for different operations.

NoGap\textsuperscript{CL} borrows heavily from VHDL, Verilog, and C. So hardware designers familiar with these languages should have no problems understanding NoGap\textsuperscript{CL}.

Although it is possible to write code that describes combinational loops, it is not advisable to do so, partly because it is considered bad design practice when designing at the RTL, but mainly because it will hinder NoGap from synthesizing cycle based software simulators for the device constructed.

NoGap\textsuperscript{CL} also exposes all the features that are present in the NoGap compiler, thus correctly used it will give the designer access to NoGap’s ability to ease the design of novel accelerators and processor without hampering the design flexibility otherwise only offered by using an RTL language.

This chapter aims to introduce NoGap\textsuperscript{CL}, mostly focusing on the syntax and semantics of the language. Adding descriptions about best practices for using NoGap\textsuperscript{CL} would make this thesis far too long and is therefore left for future publications. In most cases there will be a Backus–Naur Form (BNF) grammar detailing the use of certain language constructs. The presented grammars are based on the grammars used in the actual parser implementation but are often modified for clarity. Also
Chapter 9: NoGap Common Language

there are many cases where some statements are syntactically correct but will fail semantic checks. I have tried to extend the grammars to encompass also most of the associated semantic rules. The associated text, for each case, will explain more. The grammars presented here is by no means the full grammar for NoGap\textsuperscript{CL}, presenting the full grammar would make this thesis far to long.

NoGap\textsuperscript{CL} designs are centered around the FU concept, much like modules and architectures/entities in VHDL. However there are two main kinds of FUs in NoGap, the Mage FU and the Mase FU. Most of their syntax are similar or rather the Mage FU used a subset of the NoGap\textsuperscript{CL} language. This Chapter will first go through all common features and then dwell into the parts of the language that is related only to Mase FUs. However it is sometimes unavoidable to completely leave out features that are Mase specific in the first part, in that case it will be clearly stated witch parts are Mase FU specific.
9.2 Operators

The operators that can be used in $\text{NoGap}^{\text{CL}}$ are similar to those of Verilog. Table 9.1 list the operators that can be used in $\text{NoGap}^{\text{CL}}$. Operators are grouped according to their precedence, least precedence first. The precedence groups are delimited by double horizontal lines, e.g. the first precedence change is between <-> and <. The following notes are used in the table.

1. The two arguments are aligned at the Least Significant Bit (LSB) the shorter argument is zero-extended to match the longer argument.

2. A value is false if it is zero, otherwise is is true.

3. If the assignee can not store all bits of the assigned expression, the out of bound Most Significant Bits (MSBs) of the assigned expression are discarded. If on the other hand the assignee can store more bits than it is assigned, the final result is zero extend at the MSB side to fit the assignee.

4. Selecting bits that are out of bound is considered an error.
### Table 9.1: Operators in NoGap\(^{CL}\)

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha = \beta)</td>
<td>assignment, (\alpha) is assigned the value of (\beta)</td>
<td>3</td>
</tr>
<tr>
<td>(\alpha \rightarrow \beta)</td>
<td>Signal connection in manual FU instantiation, (\alpha) connects to input port (\beta), see Section 9.11. Also used to connect stages and phases in pipeline descriptions, see Section 9.15</td>
<td>3</td>
</tr>
<tr>
<td>(\alpha \leftarrow \beta)</td>
<td>Signal connection in manual FU instantiation, (\alpha) connects to output port (\beta), see Section 9.11</td>
<td>3</td>
</tr>
<tr>
<td>(\alpha \leftrightarrow \beta)</td>
<td>Signal connection in manual FU instantiation, (\alpha) connects to in-/output port (\beta), see Section 9.11</td>
<td>3</td>
</tr>
<tr>
<td>(\alpha &lt; \beta)</td>
<td>Less than comparison, if (\alpha)'s value is less than (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
<tr>
<td>(\alpha &gt; \beta)</td>
<td>Greater than comparison, if (\alpha)'s value is greater than (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
</tbody>
</table>

*continued on next page*
<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>(\alpha \leq \beta)</td>
<td>Greater than or equal comparison, if (\alpha)'s value is greater than or equal to (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
<tr>
<td>(\alpha \geq \beta)</td>
<td>Greater than or equal comparison, if (\alpha)'s value is greater than or equal to (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
<tr>
<td>(\alpha \ll \beta)</td>
<td>Logical left shift, (\alpha) is shifted left (\beta) bits.</td>
<td></td>
</tr>
<tr>
<td>(\alpha \gg \beta)</td>
<td>Logical right shift, (\alpha) is shifted right (\beta) bits, the new MSB is filled with zero.</td>
<td></td>
</tr>
<tr>
<td>(\alpha \ggg \beta)</td>
<td>Arithmetic right shift, (\alpha) is shifted right (\beta) bits, the new MSB is filled with a copy of itself.</td>
<td></td>
</tr>
<tr>
<td>(\alpha = \beta)</td>
<td>Equality comparison, if (\alpha)'s value is equal to (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
<tr>
<td>(\alpha \neq \beta)</td>
<td>Inequality comparison, if (\alpha)'s value is different from (\beta)'s value the expression evaluates to true, false otherwise.</td>
<td>1</td>
</tr>
<tr>
<td>Op.</td>
<td>Description</td>
<td>Note</td>
</tr>
<tr>
<td>-----</td>
<td>-------------</td>
<td>------</td>
</tr>
<tr>
<td>α &amp;&amp; β</td>
<td>Boolean and, α and β are interpreted as boolean values and the entire expression returns true if both arguments are true, false otherwise.</td>
<td>2</td>
</tr>
<tr>
<td>α</td>
<td></td>
<td>β</td>
</tr>
<tr>
<td>α</td>
<td>Bitwise or, each bit in α is or:ed with the corresponding bit in β.</td>
<td>1</td>
</tr>
<tr>
<td>α ^ β</td>
<td>Bitwise xor, each bit in α is xor:ed with the corresponding bit in β.</td>
<td>1</td>
</tr>
<tr>
<td>α &amp; β</td>
<td>Bitwise and, each bit in α is and:ed with the corresponding bit in β</td>
<td>1</td>
</tr>
<tr>
<td>α + β</td>
<td>Addition, α is added with β.</td>
<td></td>
</tr>
<tr>
<td>α − β</td>
<td>Subtraction, β is subtracted from α.</td>
<td></td>
</tr>
<tr>
<td>α * β</td>
<td>Multiplication, α is multiplied with β.</td>
<td></td>
</tr>
<tr>
<td>α / β</td>
<td>Division, α is divided with β.</td>
<td></td>
</tr>
<tr>
<td>− α</td>
<td>Negation, negates the value of α, equivalent to 0 − α</td>
<td></td>
</tr>
<tr>
<td>&amp;α</td>
<td>Reduction and, all bits in α and:ed together.</td>
<td></td>
</tr>
</tbody>
</table>

*continued on next page*
### Chapter 9: NoGap Common Language

continued from previous page

<table>
<thead>
<tr>
<th>Op.</th>
<th>Description</th>
<th>Note</th>
</tr>
</thead>
<tbody>
<tr>
<td>$</td>
<td>\alpha</td>
<td>$</td>
</tr>
<tr>
<td>$^\wedge \alpha$</td>
<td>Reduction xor, all bits in $\alpha$ xor:ed together.</td>
<td></td>
</tr>
<tr>
<td>$\sim \alpha$</td>
<td>Complement (bitwise not), Each bit in $\alpha$ is inverted.</td>
<td></td>
</tr>
<tr>
<td>!$\alpha$</td>
<td>Logical not, $\alpha$ is interpreted as a boolean value and if its false, the expression evaluates to true, false otherwise.</td>
<td></td>
</tr>
<tr>
<td>$\alpha[\beta : \gamma]$</td>
<td>Bit slicing, bits $\gamma$ to $\beta$ are sliced from $\alpha$</td>
<td>4</td>
</tr>
<tr>
<td>$\alpha[\beta]$</td>
<td>Bit slicing, bit $\beta$ is sliced from $\alpha$</td>
<td>4</td>
</tr>
</tbody>
</table>
9.3 Identifiers

Regular expression 9.1 Identifiers in NoGap\(^{CL}\)
\( (\_\mid[:alnum:])^+ \)

An identifier in NoGap\(^{CL}\) is any string matching Regular expression 9.1.

9.4 Concatenation

Grammar 9.1 Concatenation grammar
\( \langle \text{concatenation} \rangle \rightarrow &\{ \langle \text{expr_list} \rangle \} & \)

The concatenation operation is used to concatenate two or more expressions. The size of resulting bit vector is the sum of the sizes of the individual expressions. A concatenation simply copies the bits of each expression and puts them after each other, e.g. \&\{a,b,c\} \equiv 011101_2 \) if \(a \equiv 01_2, b \equiv 1_2, \) and \(c \equiv 101_2\).
9.5 Keywords and Special Characters

Table 9.2 lists the keywords in NoGap$^{CL}$. Keywords can not be used for identifiers.

<table>
<thead>
<tr>
<th>Keyword</th>
<th>NoGap$^{CL}$</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>fu</td>
<td>template</td>
<td>forward</td>
</tr>
<tr>
<td>forwarding</td>
<td>bypassing</td>
<td>condition</td>
</tr>
<tr>
<td>source</td>
<td>destination</td>
<td>instruction</td>
</tr>
<tr>
<td>immediate</td>
<td>cycle</td>
<td>comb</td>
</tr>
<tr>
<td>always</td>
<td>if</td>
<td>else if</td>
</tr>
<tr>
<td>else</td>
<td>input</td>
<td>output</td>
</tr>
<tr>
<td>inout</td>
<td>include_decoder</td>
<td>instruction_decoder</td>
</tr>
<tr>
<td>pipeline</td>
<td>operation</td>
<td>signal</td>
</tr>
<tr>
<td>parameter</td>
<td>switch</td>
<td>choice</td>
</tr>
<tr>
<td>default</td>
<td>auto</td>
<td>stage</td>
</tr>
<tr>
<td>reset</td>
<td>constant</td>
<td>interface</td>
</tr>
<tr>
<td>set_timing</td>
<td>verilog</td>
<td>function</td>
</tr>
<tr>
<td>return</td>
<td>component</td>
<td>phase</td>
</tr>
<tr>
<td>integer</td>
<td>advance</td>
<td>flushpipe</td>
</tr>
<tr>
<td>delay</td>
<td>jump</td>
<td>loop</td>
</tr>
<tr>
<td>nogap</td>
<td>ffi</td>
<td>ffo</td>
</tr>
</tbody>
</table>

Certain combinations of characters has special meanings in NoGap$^{CL}$. These character combinations are listed in Table 9.3. Their meanings are explained in Section 9.2.
Table 9.3: Special character combinations in NoGap CL

<table>
<thead>
<tr>
<th>Symbol</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>::</td>
<td>-&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;-</td>
</tr>
<tr>
<td></td>
<td>&lt;-&gt;</td>
</tr>
<tr>
<td></td>
<td>==</td>
</tr>
<tr>
<td>!=</td>
<td>&lt;=</td>
</tr>
<tr>
<td></td>
<td>&gt;=</td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td>&amp;{</td>
<td>}&amp;</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;</td>
</tr>
<tr>
<td></td>
<td>&gt;&gt;&gt;</td>
</tr>
<tr>
<td></td>
<td>&lt;&lt;</td>
</tr>
</tbody>
</table>

9.6 FU Specification

Grammar 9.2 FU specification grammar

Grammar 9.2 FU specification grammar

\[
\text{fu} \text{ cons} \rightarrow \text{fu} \ (\text{identifier}) \ (\text{statement}) \\
| \text{fu} : \text{template} \ (\text{template_arg}) \ (\text{identifier}) \ (\text{statement}) \\
(\text{template_args}) \rightarrow \langle \ (\text{template_arg_list}) \rangle | \langle \space > \rangle \\
(\text{template_arg_list}) \rightarrow (\text{template_arg_list}) \ , \ (\text{template_arg}) \\
| (\text{template_arg}) \\
(\text{template_arg}) \rightarrow (\text{identifier}) \\
\]

The FU is the central design concept of NoGap much like the module is for Verilog and the entity/architecture is for VHDL. An example of an ALU FU can be seen in Listing 9.1. An FU specification starts with the FU keyword and its name, as seen on line 1. Following is the FU’s input and output ports definitions and their sizes, as seen on line 3–7. Section 9.10 gives a thorough explanation of port definitions. The rest of (the FU is used to describe its functionality using either a Mage or Mase level description.

Common to all FUs is that no clock or reset ports need to be declared, NoGap will automatically detect if clock or reset signals are needed. Note that NoGap will resolve this correctly even for hierarchically instantiated FUs, i.e. if a sub FU, in need of clock and reset inputs, is instantiated in an FU that does not have a cycle block, NoGap will still synthesize a clock and reset input for the top level FU.
Listing 9.1: FU Specification

```plaintext
fu alu
%LEAF{ // All clauses can be named.
  input [31:0] a_i;
  input [31:0] b_i;
  input [2:0] op_i;
  output [31:0] res_o;
  output [3:0] flag_o;
  signal auto("res_o+1") res;

  comb {
    switch(op_i) {
      0: %ADD { res = a_i+b_i; }
      1: %SUB { res = a_i-b_i; }
      2: %AND { res = a_i&b_i; }
      3: %OR { res = a_i|b_i; }
      4: %XOR { res = a_i^b_i; }
      5: %NOT { res = ~a_i; }
      6: %INC { res = a_i+1; }
      7: %DEC { res = a_i-1; }
    }
  }

  comb { res_o = res[31:0]; }
  comb {
    flag_o[3] = (res_o == 0);
    flag_o[2] = res_o[31];
    flag_o[1] = res[32];
    flag_o[0] = 0;
  }
}
```
Chapter 9: NoGap Common Language

9.7 Control Structures

The control structures supported by NoGap are explained in this Section. A reader familiar with C/C++, Verilog, or VHDL should have no problem understanding these. These control structures can only be used in Mage FUs, except for limited use in the stage definitions in the Mase graph.

9.7.1 if / else if / else Structure

The if/else if/else control structure should be familiar to anyone who has ever programmed in a C like language. The grammar governing the if construct is shown in Grammar 9.3 The ⟨cond_expr⟩ can be any valid expression, however the result of the expression is casted to a boolean, where zero is interpreted as false and everything else is interpreted as true. A small code example can be found in Listing 9.2

Listing 9.2: If control structure example

```c
1 if(a == b || c <= b)
2 {
3    sig = 3;
4 }
5 else if(a == c)
6 {
7    sig = 2;
8 }
9 else sig = 1;
```
9.7.2 switch Structure

The `switch` construct is similar to the `switch` construct in C-like languages, with some minor modifications. The grammar governing the `switch` construct is shown in Grammar 9.4. As can be seen it is syntactically correct to use any language construct that is a ⟨statement⟩ as the final nonterminal in the first rule, however this is not semantically correct. Semantically this ⟨statement⟩ has to be one or more ⟨switch_case_cons⟩ nonterminals, usually contained in a clause.

The selection expression ⟨⟨sel_expr⟩⟩ needs to be an expression that evaluates to a constant number in compile time. It is also possible to specify the expression with x-es. An x matches any bit value at that position of the selection expression. This is similar to Verilog’s `casex` statement.

The `default` case value is will match any selection that is not matched by a selection expression.

An example of how to use the `switch` statement is shown in Listing 9.3. Here the use of x-es is displayed on line 4, which will match `0000_2`, `0001_2`, `0010_2`, and `0011_2`.

**Listing 9.3:** Switch control structure example

```java
 1 signal [3:0] a;
 2 switch (a)
 3 {
 4   4'b00xx:{b = 0;}
 5   8:{b = 2;}
 6   default:{b = 3;}
 7 }
```
Chapter 9: NoGap Common Language

9.8 Cycle and Comb Blocks

Grammar 9.5 Cycle and comb grammar

\[
\begin{align*}
(\text{comb}_\text{cons}) & \rightarrow \text{comb} \ (\text{statement}) \\
(\text{cycle}_\text{cons}) & \rightarrow \text{cycle} \ (\text{statement})
\end{align*}
\]

All logic, written in NoGap\textsuperscript{CL}, takes place within either a cycle or a comb block. The comb block describes combinatorial logic and the cycle block describes synchronous logic. Grammar 9.5 describes the grammar for the cycle and comb blocks. A usage example of both the comb and cycle blocks can be seen in Listing 9.4 on line 26–30 and line 20–24 respectively.

9.9 Clause Name

Grammar 9.6 Clause grammar

\[
\begin{align*}
(\text{clause}) & \rightarrow (\text{clause}_\text{body}) \ | \ (\text{clause}_\text{name}) \ (\text{clause}_\text{body}) \\
(\text{clause}_\text{body}) & \rightarrow \{ (\text{statements}) \} \ | \ \{ \} \\
(\text{clause}_\text{name}) & \rightarrow \% \ (\text{identifier})
\end{align*}
\]

All clauses, i.e. all code blocks starting with \{ and ending with \} can be named. This is important for operation constructions since an operation on a leaf FU is constructed by declaring which of the clause(s) shall be reached. In Listing 9.1, eight clauses are named, as can be seen on line 12–19. The clause names represent different ALU operations. NoGap uses these clause names to transparently synthesize the needed control vectors and associated control path if the FU in question is used as an instruction controlled FU. This process is called DCS and is further described in Section 11.2.
### 9.10 Port and Signal Declarations

Port and signal definition grammar

\[
\begin{align*}
\langle \text{port} \rangle & \rightarrow \langle \text{port\_direction} \rangle \ \langle \text{port\_offset} \rangle \ \langle \text{port\_signal} \rangle \ \langle \text{identifier} \rangle ; \\
\langle \text{port\_direction} \rangle & \rightarrow \text{input} | \text{output} | \text{inout} \\
\langle \text{port\_offset} \rangle & \rightarrow \{ + \langle \text{expr} \rangle + \} \\
\langle \text{port\_signal} \rangle & \rightarrow \langle \text{signal\_stmt} \rangle | \langle \text{size} \rangle \\
\langle \text{size} \rangle & \rightarrow \langle \text{range} \rangle | \langle \text{range} \rangle : \langle \text{range} \rangle \\
\langle \text{range} \rangle & \rightarrow [\langle \text{expr} \rangle : \langle \text{expr} \rangle] | [\langle \text{expr} \rangle] | \langle \text{auto\_range} \rangle \\
\langle \text{auto\_range} \rangle & \rightarrow \text{auto} (\ \langle \text{string} \rangle ) \\
\langle \text{signal\_stmt} \rangle & \rightarrow \text{signal} \ \langle \text{size} \rangle \ \langle \text{identifier} \rangle ; \\
\end{align*}
\]

Ports are used to define the interface of an FU and signals are used for assignments and data flow within an FU. They are similar in their definitions and use. To implement a clocked register within an FU a signal has to be used. The BNF grammar for ports and signals are shown in Grammar 9.7.

An example of port and signal definitions are shown in Listing 9.4, line 2 and 3, show two simple port definitions, \( a_i \) is one bit wide and \( b_i \) is eight bits wide. Line 5 and 6 show the definition of two input ports using the auto range for their sizes. The auto range concept is further explained in Chapter 10. \( c_i \) will be set to the size of the signals connected to port \( c_i \). \( d_i \) will then be set to double the size of \( c_i \). The output ports from line 8 to line 12, should not be any problem to understand. However some words needs to be mentioned about the use of the automatic range on line 12. To be able to use an automatic output port size the FU in question, must be a \textsc{Mase} FU\(^1\).

The signal definitions are similar to the port definitions. Worth not-

---

\(^1\)This is the case at the time of writing this thesis, there are however advanced plans to implement this feature for \textsc{Mage} FUs as well.
Chapter 9: NoGap Common Language

ing is the definition for a two dimensional signal on line 18. This is how memories can be declared in NoGap CL. This particular memory is a synchronous write and asynchronous read memory, as can be deduced from line 23 and line 29.

Listing 9.4: Signal and ports example

```plaintext
fu port_example {
  input a_i;
  input [7:0] b_i;
  input auto("#") c_i;
  input auto("2*c_i") d_i;
  output e_o;
  output [7:0] f_o;
  output auto("(a_i*b_i)+1") g_o;
  output auto("#") h_o;
  ...;
  signal s1;
  signal [7:0] s2;
  signal auto("c_i+d_i") s3;
  signal auto("#") s4;
  signal [7:0][255:0] mem;
  cycle (s1 = a_i;
          mem[b_i] = c_i;
  }
  comb { e_o = s1;
          f_o = mem[b_i+1];
  }
}  
```
9.10.1 Port Timing Offset

As seen in Grammar 9.7, ports can be given timing offsets. This is useful if the FU have internal pipelining. NoGap can be notified about this by setting timing offsets for the ports affected. Both positive and negative offsets can be used. A port that has an offset will be placed a number of cycles earlier/later (as specified by the offset count) in the data path pipeline from its normal placement. This is shown in Listing 9.5, where, on line 2, dat_i is offset by -1 cycles and, on line 3, dat_o is offset by +2 cycles. Note that (at the time of writing this thesis) the offset directives will only be adhered to if the FU is placed in Master graph. The offset directives will be ignored if the FU is manually instantiated in another FU, as described in Section 9.11.

Listing 9.5: Port offset

```plaintext
fu port_offset_example {
  input {+ -1 +}[7:0] dat_i;
  output {+ 2 +}[7:0] dat_o;
  ...;
}
```
9.11 Manual FU Instantiation

**Grammar 9.8 Manual FU instantiation grammar**

\[
\begin{align*}
\langle \text{fu\_instantiation\_cons} \rangle & \rightarrow \text{component} \ \langle \text{fu\_name} \rangle \ \langle \text{instance\_name} \rangle \\
& \quad | \ \text{component} \ \langle \text{fu\_name} \rangle \ \langle \text{para\_pass} \rangle \ \langle \text{instance\_name} \rangle \\
& \quad | \ \langle \text{statement} \rangle \\
\langle \text{fu\_name} \rangle & \rightarrow \langle \text{identifier} \rangle \\
\langle \text{instance\_name} \rangle & \rightarrow \langle \text{identifier} \rangle \\
\langle \text{para\_pass} \rangle & \rightarrow \# \ (\ \langle \text{para\_byname\_list} \rangle \ ) \\
\langle \text{para\_byname\_list} \rangle & \rightarrow \langle \text{para\_value\_byname} \rangle \ | \ \langle \text{para\_byname\_list} \rangle , \ \langle \text{para\_value\_byname} \rangle \\
\langle \text{para\_value\_byname} \rangle & \rightarrow . \ \langle \text{identifier} \rangle \ (\ \langle \text{expr} \rangle \ ) \ | \ . \ \langle \text{identifier} \rangle \ (\ ) \\
\langle \text{connection} \rangle & \rightarrow \langle \text{expr} \rangle \ \langle \text{connection\_direction} \rangle \ \langle \text{port\_name} \rangle \\
\langle \text{connection\_direction} \rangle & \rightarrow <- | -> | <-> 
\end{align*}
\]

Manual instantiation of FUs is the same as instantiating a module in Verilog, however the syntax in NoGapCL is different. The grammar for manual FU instantiation is given in Grammar 9.8. Although syntactically any statement is accepted at the end of the component instantiation rule. Semantically this has to be of type \langle \text{connection} \rangle or a clause consisting of \langle \text{connection} \rangle statements. The three \langle \text{connection\_direction} \rangle options connects output ports (<-), input ports (->), and inout ports (<->). These operators are also called connection arrows. The instantiated component ports are on the right hand side of the connection arrows and the connecting signals, from the instantiation level, are on the left hand side. Thus the connection arrows will point in the data flow direction. NoGap compiler will use the direction of the connection arrow to validate the connections. Erroneous connections, such as connecting a top level input port to an output port of the instantiated FU, will result
Listing 9.6: Manual FU instantiation example

```plaintext
fu manual_fu_inst_example {
  input [31:0] top_a_i;
  input [31:0] top_b_i;
  output [31:0] top_res_o;
  ....;
  signal [2:0] op;
  signal [3:0] flags;
  ....;
  component alu alu_inst {
    top_a_i -> a_i;
    top_b_i -> b_i;
    op -> o_p_i;
    top_res_o <- res_o;
    flags <- flag_o;
  }
}
```

in a compilation error.

Listing 9.6 shows an example of a manual FU instantiation, where the ALU, shown in Listing 9.1, is instantiated in another FU. Here it becomes clear how the connection arrows help to clarify in what direction data flows for the different connections. Just remember that the instantiated FU’s ports are on the right and the top level signals are on the left.

9.12 Mase FU Specification

The rest of this Chapter will deal with features only available in Mase FU specifications.

A Mase FU specification is written in much the same way as a Mage FU, but with an extended set of language constructs, but also with some limitations from what can be written in Mage FU. A Mase FU contains one or more operation constructs, further described in Sec-
Chapter 9: NoGap Common Language

A NoGap FU specification can include: phase statements, pipeline constructs, and/or operation constructs. Using phases, the operation’s pipeline timing can be computed by NoGap. Assignments written after a phase statement take place in the timing of that phase. The operation construct define the operations of the processor, and also define the connection relationship between different Mage FUs. The operation construct is further described in Section 9.15.

9.13 Phase Description

Grammar 9.9 Phase definition grammar

\[
\text{phase} \langle \text{identifier} \rangle
\]

Phases are used in operation constructs to define timings for a number of pipelined operations. The grammar for a phase definition is shown in Grammar 9.9.

Examples of phase definitions can be seen in Listing 9.8 on line 1–7.

9.14 Stage Description

Grammar 10 Stage declaration grammar

\[
\text{stage} \langle \text{stage_id} \rangle \text{ ( (stall_ctrl) ) (statement) }
\]

Stages are templates that describes how data propagates from one phase to another. The grammar governing stage declarations is shown in Grammar 9.10.
Semantically the (statement) has to be a clause containing a `comb` or `cycle` block, using the `ffi` and `ffo` keywords for relating how data flows from one stage to the next. `ffi` means data flowing into the stage and `ffo` means data flowing out. The two simplest stage declarations are the wire model and the FF model. They are shown in Listing 9.8, where the wire model stage is shown on line 19–25 and the FF model is shown on line 11–17.

9.14.1 Stall Operation Description

A real world processor should have the ability to be stalled depending on conditions outside of the processor’s control. This is needed for example if the processor is going to use a cache or simply communicate over a shared resource with random delays. \( \text{NoGap} \) handles this by giving the designer the ability to insert stallable stages. Listing 9.7 shows an example of a stallable stage on line 6–14.

The stallable stage has a single input controlling if stalling should be performed or not. The stall signal is assumed to come from an FU in the data path. If there is a stallable stage in the processor \( \text{NoGap} \) will insert a load enable signal for all registers in the design. As long as stall is asserted, no registers in the processor will be written, essentially stopping time for the processor. There must however be some registers which are still writable, e.g. if the stall logic is controlled by a Finite State Machine (FSM). For this reason, some registers in \( \text{NoGap} \) can be marked as \textit{unstallable}. Note that this stalling does not describe the stalling needed to resolve resource conflicts in the data path.

Listing 9.7: Stage stall example

```plaintext
1 fu stall_example
2 {
3   ...
4   fu::stall_logic(%STALL) stall_logic;
5  
6  stage ff(stall_logic.stall_o)
```
Chapter 9: NoGap Common Language

```c
7 {
8     input stall_i;
9     cycle
10     {
11         if(stall_i) ffo = ffo;
12         else ffo = ffi;
13     }
14 }
15
16 pipeline the_pipe //normal pipeline
17 { main_decoder -> ff -> OF -> ff -> EX -> ff -> WB; }
18 ...;
19 }
```

9.15 Pipeline Description

**Grammar 9.11** Pipeline description grammar

- `(pipeline_cons)→pipeline (identifier) (statement) □`
- `(pipe_description)→(pipe_start) -> (pipe_desc_list)`
  - `(phase_id) □`
- `(pipe_start)→(fu_inst_id) | (phase_id) | (pipeline_id) □`
- `(pipe_desc_list)→(pipe_desc_list) -> (pipe_desc_elem)`
  - `(pipe_desc_elem)→(stage_id) -> (phase_id) □`
- `(stage_id)→(identifier) □`
- `(phase_id)→(identifier) □`
- `(fu_inst_id)→(identifier) □`
- `(pipeline_id)→(identifier) □`

The pipeline construct describes how a number of phases are related to each other. The pipeline construct grammar is shown in Grammar 9.11. The `(statement)` nonterminal in the first rule has, semantically, to be a clause containing a `(pipe_description)` statement.
The purpose of a pipeline declaration is to set the timing of a number of phases. These phases are later used in the operation construct to design pipelined logic. Each phase is linked to another phase via a stage. Phases and stages are described in Section 9.13 and Section 9.14, respectively.

Listing 9.8 shows four different pipeline declarations. The first pipeline declaration, the_pipe on line 27–30, is the simplest. There are three phases, OF, EX, and WB, connected with two stages, ff and wire. Note that the wire stage means that the EX and WB phases will have the same timing, i.e. there will be no pipeline register separating signals going from phase EX to phase WB. The second pipeline declaration, long_pipe on line 32–35, is almost the same as pipeline the_pipe just longer. Something which is important to note here though is that the same phase must have the same timing in all pipelines. This is however something that needs to be modified in NoGap. The intended way for this to work is that the individual phase timings are associated with a particular pipeline and are not global timings. When describing an operation one pipeline has to be chosen and then it defines the timings of the phases within that operation.

The third pipeline declaration, extend_pipe on line 37–40, displays the pipeline extension feature, used to extend an already declared pipeline with extra phases and stages. First the name of a declared pipeline is used and then connected to more phases via a stage. In this particular case the_pipe is extended with two more phases. It is important to note that the phases added can not be any of the phases in the original pipeline. The last pipeline declaration, extend_from_fu_pipe on line 42–45, shows how a pipeline can be extended from a pipeline declared in an instantiated FU. Here all phase names need to be unique. Which in this case means that the phase names in the some_decoder pipeline can not be OF, EX, or WB.
Listing 9.8: Pipeline description examples

```java
1  phase OF;
2  phase EX;
3  phase EX2;
4  phase EX3;
5  phase WB;
6  phase WB2;
7  phase WB3;
8
9  fu::decoder<instr_i,flush_i>() some_decoder
10
11 stage ff()
12 {
13    cycle
14    {
15      ffo = ffl;
16    }
17 }
18
19 stage wire()
20 {
21    comb
22    {
23      ffo = ffl;
24    }
25 }
26
27 pipeline the_pipe
28 {
29  OF -> ff -> EX -> wire -> WB;
30 }
31
32 pipeline long_pipe
33 {
34  OF -> ff -> EX -> ff -> EX2 -> wire -> WB2;
35 }
36
37 pipeline extend_pipe
38 {
39  the_pipe -> wire -> EX3 -> ff -> WB3;
40 }
41
42 pipeline extend_from_fu_pipe
43 {
44  some_decoder -> ff -> OF -> ff -> EX3 -> ff -> WB3;
45 }
```
9.16 FU Instantiation in Mase FUs

**Grammar 9.12** FU use definition grammar

\[
\begin{align*}
(fu\_definition) & \rightarrow fu :: (fu\_name\_id) \ (\{fu\_use\_arg\_list\}) \\
(instance\_id) & \mid fu :: (fu\_name\_id) \ (\{fu\_use\_arg\_list\}) \\
 & \ (para\_pass) \ (instance\_id) \\
(fu\_name\_id) & \rightarrow (identifier) \\
(instance\_id) & \rightarrow (identifier)
\end{align*}
\]

FU instantiation in Mase FUs differs from the manual instantiation described in Section 9.2. At first the FU to be used has to be declared for use according to Grammar 9.12. Then the declared FUs can be used in the operation construct, described in Section 9.17. The \(\{fu\_use\_arg\_list\}\) must resolve to one unique clause in the instantiated FU. This will be called the default instruction. The default instruction is used when the FU should perform a No Operation (NOP). For stateless FUs this is of little importance. But for state full FUs this matters a lot, e.g. a register file can only be allowed to be written at specific cycles. If the FU has parameters, they can be set here.

Some examples of FU definitions in a Mase FU can be found in Listing 9.9 on line 3.
9.17 Pipelined Operations

A pipelined operation is an important concept in a Mase description. A pipelined operation defines a pipelined hardware structure and a set of instructions for the FUs that are part of the operation. The grammar governing operation definitions is shown in Grammar 9.13. The (pipe_id) specifies what pipeline the operation is associated with as subsequently the meaning of the phase names. The (op_name_id) is the name of the operation. The (code_expr) specify the op-code range for this operation, this is further described in Section 9.17.1. The (default_op) is used to mark one of the operations in the Mase FU as being the operation that is performed as the NOP operation. The (statement) has semanti-
Chapter 9: NoGap Common Language

cally to be one or a clause of \(\text{pipeline\_statement}\). Pipeline statements are either phase specifications \(\langle \text{phase} \rangle\) or a signal assignment, defined by \(\langle \text{signal\_assignment} \rangle\). The \(\langle \text{fu\_use\_statment} \rangle\) is further described in Section 9.18. Grammar 9.13 also shows that \(\langle \text{source} \rangle\) can be an \(\langle \text{expr} \rangle\). This is called inline FU usage and is further explained in Section 9.19.

Some of the advantages of describing a pipeline this way compared to other HDLs are:

- Signals used further down in the pipeline does not need to be manually delayed, NoGap will insert as many flip flops as needed to keep all signals synchronized.

- The pipeline is visible to the designer and it is no question where in the pipeline each FU is placed. Normally the pipeline architecture has to be described in a separate document, whereas using NoGap this information is captured in the code.

- It is easy to merge and split pipeline stages, just change the delay between the phases and NoGap will regenerate a new data path structure.

Listing 9.9 shows an example where the operation construct is used. The \texttt{alu\_inst} operation defines eight operations for the processor, the different operations are specified by different clause names. The connection relationships of \texttt{main\_decoder}, \texttt{alu}, and \texttt{regfile} are also defined. Using the defined connection relationships, NoGap constructs a \texttt{Mace} graph description of the processor’s data path.
Chapter 9: NoGap Common Language

Listing 9.9: Operation FU example

```plaintext
fu machine {
  output [31:0] dat_o;
  fu::alu( %ADD ) alu;
  ...
  phase OF;
  phase EX;
  phase WB;
  //stage ff( "register";)
  stage ff() { cycle { ffo = ffi; } }
  fu::decoder< instr_i > main_decoder;
  ...
  pipeline the_pipe {
    main_decoder -> ff -> OF -> ff -> EX -> ff -> WB; }
  operation( the_pipe ) alu_inst(main_decoder.alu_sq) {
    OF;
    rf;
    rf.addr_a_i = main_decoder.rf_a;
    rf.addr_b_i = main_decoder.rf_b;
    EX;
    alu( %ADD, %SUB, %AND, %XOR, %XOR, %NOT, %INC, %DEC;)
    alu.a_i = rf.data_a_o;
    alu.b_i = rf.data_b_o;
    WB;
    rf.WRITE;
    rf.addr_a_i = main_decoder.rf_a;
    rf.data_i = alu.res_o;
  }
  ...
}
```
9.17.1 Operation Code Assignment

The \texttt{\langle code_expr \rangle} seen in 9.13 is used to assign op codes to the operation. At the moment there are three ways of doing this:

1. Using a decoder template

2. Assigning a constant value

3. Assigning no value.

When using a decoder template, as seen in Listing 9.9 on line 16, \texttt{NoGap} will assign the operation codes it sees fit. Note that one operation in this case can have multiple operation codes. There will be one for each possible sub operation.

Assigning a constant value, e.g. \texttt{operation \ldots \{op_i == 2\}} means that an input to the \texttt{Mase} FU called \texttt{op_i} shall be two for this operation to execute. In this case, operations can not have any sub operations.

Using an empty \texttt{\langle code_expr \rangle}, e.g. \texttt{operation \ldots \{\}} means that this operation will always execute. No other operation can be present in the FU. This is used to design a data path that only utilizes \texttt{NoGap}'s ability to keep track of pipeline stages and synchronize signals.
Chapter 9: NoGap Common Language

9.18 FU Usage List

Grammar 9.14 FU usage list grammar

\[
\begin{align*}
& \langle \text{fu}_\text{use}_\text{stmt} \rangle \rightarrow \langle \text{id} \rangle \ast \langle \text{fu}_\text{use}_\text{arg}_\text{list} \rangle \ast \\
& \langle \text{fu}_\text{use}_\text{arg}_\text{list} \rangle \rightarrow \langle \text{fu}_\text{use}_\text{arg}_\text{list} \rangle , \langle \text{fu}_\text{use}_\text{arg} \rangle \\
& \quad \mid \langle \text{fu}_\text{use}_\text{arg} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{clause}_\text{name} \rangle \\
& \quad \mid < \langle \text{fu}_\text{use}_\text{combination}_\text{list} \rangle > \\
& \langle \text{fu}_\text{use}_\text{combination}_\text{list} \rangle \rightarrow \langle \text{fu}_\text{use}_\text{combination}_\text{list} \rangle , \\
& \quad \{ \langle \text{name}_\text{list} \rangle \} \\
& \quad \mid \{ \langle \text{name}_\text{list} \rangle \} \\
& \langle \text{name}_\text{list} \rangle \rightarrow \langle \text{name}_\text{list} \rangle , \langle \text{name} \rangle \\
& \quad \mid \langle \text{name} \rangle \\
& \langle \text{name} \rangle \rightarrow \langle \text{id} \rangle \mid \langle \text{clause}_\text{name} \rangle \\
\end{align*}
\]

An FU usage list is used to define how a particular FU should be used in a pipelined operation. Pipelined operations are described in Section 9.17. The FU usage list informs NoGap about what possible operations the FU should be able to perform and at what stage in the pipeline the FU is placed. The grammar governing FU usage lists are shown in Grammar 9.14.

FU placement affects where in the pipeline the FU is considered to reside. The FU’s output ports reside in the same pipeline stage as the FU. However its input ports can be connected in other pipeline stages. Figure 9.1 shows part of the \textit{Mast} graph generated from the operation definition in Listing 9.9. The FU is placed at the OF phase, so all its outputs are also placed in this phase. The ALU is placed in the following pipeline phase, EX, and all its inputs and outputs are used in this phase. Finally in the WB phase the output of the ALU is connected to the input port of the register file. The effect of this can be seen in Figure 9.1,
Listing 9.10: FU usage list examples

```plaintext
1 operation (pipe) ex_op (dec.ex_op)
2 {
3     ...
4     @ PX;
5     fu_placed;
6     fu_single '%ADD';
7     fu_multiple '%ADD,%SUB,%AND,%OR';
8     fu_combination '<{%ADD,%SUB},{%AND,%OR}>.XXOR';
9     ...
10 }
```

where the register file’s input port resides in the WB phase.

Listing 9.10 shows some example of how the FU usage statement can be used. The first version, on line 5, is simply a placement statement, forcing the `fu_placed` to be placed in phase PX. No additional operations or control paths will be synthesized for this FU. The second version, on line 6, means that `fu_single` is placed in phase PX executing its `%ADD` clause. The third version, on line 7, means as usual that `fu_multiple` is placed in phase PX, however specifying more than one clause access directive means that this particular FU can execute any of these functions. This means that for this example, ignoring the last FU usage directive, that `ex_op` in reality will be synthesized into four different instructions, one instruction for each of the clause access statements. The last version, on line 8, shows how directives can be combined. Imagine that `fu_combination` has an adder/subtractor connected to a logic block that either take the AND or OR of the result from the adder. Also this FU can, completely unrelated to the other logic, just perform an XOR operation. This is specified as shown in the code example. First the combination list, `<{%ADD,%SUB},{%AND,%OR}>`, instructs NoGap that there are two unrelated but controllable units within this FU. Therefore NoGap will synthesize four instructions for this, i.e. all
Figure 9.1: FU placement example
possible combinations of the two directive sets. The single XOR directive just adds one additional instruction for this FU, since it is unrelated to the combination list. All in all the fu_combination will have five instructions, Figure 9.2 shows an example for how such an architecture might look. NoGap will generate one instruction for each possible combination from all used FUs. Which means that ex_op will cause $1 \cdot 1 \cdot 4 \cdot 5 = 20$ instructions to be synthesized. This means that using too many FUs and combination lists within one operation can cause the number of instructions to explode. It might therefore sometimes be wise to split certain intended behaviors into several operations. Future research will focus on better specifying legal clause select combinations to avoid the current multiplying effect.

Figure 9.2: Add/Sub-AND/OR XOR
Listing 9.11: Inline FU usage example

1 fu machine
2 {
3    input auto("#") op_a_i;
4    input auto("#") op_b_i;
5    ...;
6    operation(the_pipe) ex_op(dec.type_bar)
7    {
8        @p1;
9        l1_a_i = op_a_i | op_b_i;
10       sig = 0xf & op_c_i;
11       fu_a.port_i = fu_b.port_o + signal;
12        @p2;
13        l1.b_i = sig;
14        l1.%ADD;
15        l2_a_i = l1.c_o;
16        l2_b_i = l1.world_byte_o;
17        l2.%SUB;
18        @p3;
19        res_o = l2.c_o;
20        ...
21    }
22}

9.19 Inline FU Expressions

Functionality can be entered directly into an operation. Three examples of inline expression usage is shown in Listing 9.11 on line 9, line 10, and line 11.

The use of expression directly in operation constructs is called an inline FU expression because this causes NoGap to generate an anonymous FU that will be inserted into the Masr graph. The port sizes of the generated FU is calculated automatically by NoGap, this is further described in Section 9.19.1.
Chapter 9: NoGap Common Language

Table 9.4: Sizing functions for inline expression operators

<table>
<thead>
<tr>
<th>Form</th>
<th>Size Expression</th>
<th>⊖</th>
</tr>
</thead>
<tbody>
<tr>
<td>⊖α</td>
<td>S(α)</td>
<td>−,ᶜ</td>
</tr>
<tr>
<td>α ⊖ β</td>
<td>max(S(α), S(β)) + 1</td>
<td>+,−</td>
</tr>
<tr>
<td>α ⊖ β</td>
<td>S(α) + S(β)</td>
<td>×,÷,&lt;,≤,≥,≠,&amp;&amp;,∥,≤,¬</td>
</tr>
<tr>
<td>α ⊖ β</td>
<td>S(α)</td>
<td>&gt;&gt;,≫</td>
</tr>
<tr>
<td>α ⊖ β</td>
<td>max(S(α), S(β))</td>
<td>&amp;,</td>
</tr>
<tr>
<td>α ⊖ β</td>
<td>1</td>
<td>=,&gt;,≥,≤,≠,&amp;&amp;,∥,≤,¬</td>
</tr>
<tr>
<td>C</td>
<td>S(C)</td>
<td>C is a constant value</td>
</tr>
<tr>
<td>α[β:γ]</td>
<td>β − γ + 1</td>
<td>bit-field slice in α</td>
</tr>
</tbody>
</table>

9.19.1 Port Sizing of Inlined FUs

The size of both input and output ports of a generated inline FU has to be correctly calculated. The input sizes are set to Incoming Bus Size (#) and the output size expression for the created FU is automatically synthesized by NoGap. The sizing assumptions made for inline expressions are listed in Table 9.4, whereᶜ is bitwise inversion, < is left shift, > is logical right shift, ⊕ is concatenation and logical or, ⊖ is bitwise xor, && is logical and, || is logical or, ≤ is bit index, ¬ is logical not, and the other operators have their normal meaning. S(α) returns the size in bits of α. Using these rules the expression is evaluated and a symbolic expression for the output size will be generated and inserted into the Mase graph.

9.20 Forwarding Path Description

Pipelined processors often have read-after-write (RAW) data hazards. To avoid RAW hazards, forwarding and bypassing paths can be used. Forwarding paths are typically used in a processor where the operands
are read and the results are written in separate pipeline stages. The designer can explicitly specify in \texttt{NoGap} how to solve data hazard using forwarding path. It can be constructed by specifying the path’s source and destination, seen in Listing 9.12, and the pipeline phase of source and destination port is also specified. Based on their description in \texttt{NoGap}, the Verilog generation tool can automatically implement the necessary hardware for forwarding path.

9.21 Static Connections

A static connection is a connection in a \texttt{Mase} FU that is independent of the pipeline. Both \texttt{comb} and \texttt{cycle} blocks can be used. Static connections are very useful when linking external modules to \texttt{NoGap} generated code. For example cores that have already been written for other projects. External modules can be linked by using interface modules where some ports are connected to the data path controlled by \texttt{NoGap} and some of the ports are statically connected to top level input/output ports.
Listing 9.13 uses some static connections. A \texttt{comb} static connection is used on line 32–35 and a \texttt{cycle} static connection is used on line 37–41. Using \texttt{comb} blocks means that the connections are made without any delay. Using the \texttt{cycle} block, means that the signals are registered through one or more FFs. The corresponding \texttt{Mace} graph is shown in Figure 9.3, where it is made clear that there is one connection going directly from \texttt{dat_a_i} to \texttt{val_o}, as described in the \texttt{comb} static connection, and one connection going through two FF from \texttt{add1.res_p} to \texttt{val2_o}, as described in the \texttt{cycle} static connection. Note that the first FF caused by the \texttt{cycle} block is merged with the pipeline FF created from the operation.
Listing 9.13: Static connection in Mase

```plaintext
fu data_path
{
  input [31:0] dat_a_i;
  input [31:0] dat_b_i;
  input add_sub_i;
  output [31:0] res1_o;
  output [31:0] val_o;
  output [31:0] val2_o;

  phase P1;
  phase P2;

  stage ff(){ cycle{ ffo = ffi; }}
  fu::adder(%ADD) add1;
  signal auto("#") the_sig_ff;
  pipeline pipe{ P1 -> ff -> P2; }

  operation(pipe) add_sub()
  {
    @P1;
    add1.add_sub_i = add_sub_i;
    add1.a_i = dat_a_i;
    add1.b_i = dat_b_i;
    @P2;
    res1_o = add1.res_o;
  }

  comb
  {
    val_o = dat_a_i;
  }

  cycle
  {
    the_sig_ff = add1.res_o;
    val2_o = the_sig_ff;
  }
}
```
Figure 9.3: Static connections
9.22 Flow Control

An instruction driven pipeline data path architecture usually has a sequencer consisting of an instruction memory using a program counter to point to the current running instruction. Such architectures can in most cases do flow control via flow control instructions, i.e. instructions that can move execution to an instruction not directly following the current instruction. All these kinds of instructions will be referred to as jump operations. A jump operation in NoGap is closely tied to the PC-FSM, which is further described in Section 9.23.1. In order to perform a jump, an output port from the data path must be connected to the jump input port of the sequence path. It is then up to the operation to set the jump port to one if a jump shall be taken. In the sequence path, this signal will most likely be connected to the PC-FSM.

If the PC-FSM template is utilized, NoGap has to determine how many instructions needs to be flushed before execution can start again. The number of instructions to flush depends partly on the data path architecture and partly on the control path architecture. The data path dependency is called the jump prologue. The jump prologue is the number of cycles the system has to wait until both the target address and
the jump decision has been computed. The control path dependency is called the jump epilogue. The jump epilogue is the number of instructions that are in the pipeline between the program counter and decoder input. NoGap can determine both the prologue and epilogue length automatically. An example is given Figure 9.4 where PC represent the program counter and PC-FSM and JD/TA represents the Jump Decision logic and Target Address output. The data path is the dark gray region. The distance from the decoder to the JD/TA is two and thus the prologue length is two, the distance from the Program Counter (PC) to the decoder is one and thus the epilogue is one. In total three instructions needs to be flushed after a jump instruction.

9.23 Design Template

In a processor design, not all FUs can be independent from all other FUs. One example is a PC-FSM that should handle delay slots and flushing needed for program flow control. Not only the pipeline depth of the data path, but also the distance in cycles from the PC register to the decoder input, will determine how many cycles of flushing has to be done.

For this reason some design elements can be expressed as template FUs. NoGap will then instantiate real FUs during the compilation process when NoGap have full knowledge of the system. By utilizing template designs it is possible to design, with NoGapCL, general library FUs that can be reused in many different designs. Much like the standard template library in C++. In Listing 9.14, a program counter template for jump operation is defined. When constructing a jump operation, NoGap can generate the real program counter according with the template.

It should be noted that a design template in NoGap is a helpful feature that can be used to ease the design process. But is not a required component. For example the PC-FSM in SENIOR, described in Chapter 18, was designed with normal NoGapCL code.
9.23.1 PC-FSM Template

The PC-FSM template is used to model PC-FSMs. A general description of program flow control in NoGap can be found in Section 9.22.

The template has a predefined port interface, as shown in Listing 9.14 on line 3–13, where D, A, L, and P are sizes determined by the architecture. The input signals should be used in a switch statement as shown on line 20 to select one of several possible actions, as shown on line 23–31. The meaning of these clauses are described in Table 9.5. These clauses are intended to control the outputs shown on line 15–16. Note that the PC-FSM template does not contain the PC register. It should only update it with its next value.

An example of the PC-FSM, generated from the code in Listing 9.14, can be found in Figure 9.5.1. Included are also part of the condition tree, in Figure 9.5.2, and action tree, in Figure 9.5.3, for the transition going from $S_2$ to $S_3$. The transition in question has been set in a different font. One or two $\text{Mage}$-descriptors are associated with each transition, shown as hexadecimal numbers in the figure. The first $\text{Mage}$-descriptor, before the colon, points to the root of a condition tree, i.e. the condition that needs to be fulfilled for the transition to happen. If there is no $\text{Mage}$-descriptor before the colon, the transition is taken if no other transition is taken from the FSM. The second number, after the colon, points the action tree, i.e. the tree that specifies what should happen if the transition is taken. The expression and condition tree should be directly translatable to a standard C like language or a standard HDL.
Listing 9.14: PC-FSM template specification

```c
fu::template<> pc_spec
{
    input [D-1:0] delay_slots_i;
    input [A-1:0] target_addr_i;
    input jump_instruction_i;
    input loop_instruction_i;
    input [L-1:0] loop_begin_addr_i;
    input take_jump_i;
    input loop_end_i;
    input single_loop_end_i;
    input [P-1:0] curr_pc_i;
    input loop_begin_eq_end_i;
    input stall_i;

    output [P-1:0] next_pc_o;
    output flush_o;

    advance
    {
        switch &{
            loop_begin_eq_end_i, loop_instruction_i, loop_end_i,
            single_loop_end_i, jump_instruction_i, take_jump_i
        }

        6b000001: %JUMP {...}
        6b000010: %JUMP_PROLOGUE {...}
        6b0001xx: %SINGLE_LOOP_END {...}
        6b001xxx: %LOOP_END {...}
        6b01xxxx: %LOOP_BEGIN {...}
        6b1xx1xx: %SINGLE_INST_LOOP {...}
        6b11xxxx: %SINGLE_INST_LOOP_END {...}
        6b000011: %ERROR {...}
        default: %INC {...}
    }
}
```
Chapter 9: NoGap Common Language

9.5.1: PC-FSM graph

9.5.2: Condition tree

9.5.3: Action tree

Figure 9.5: PC-FSM example with action and condition tree

Figure 9.5: PC-FSM graph
### Table 9.5: PC-FSM clauses

<table>
<thead>
<tr>
<th>Name</th>
<th>Intended Action</th>
</tr>
</thead>
<tbody>
<tr>
<td>%JUMP</td>
<td>take a jump</td>
</tr>
<tr>
<td>%JUMP_PROLOGUE</td>
<td>handle the jump prologue</td>
</tr>
<tr>
<td>%SINGLE_LOOP_END</td>
<td>handle the end of a single iteration loop</td>
</tr>
<tr>
<td>%LOOP_END</td>
<td>handle the end of a loop</td>
</tr>
<tr>
<td>%LOOP_BEGIN</td>
<td>handle the beginning of a loop</td>
</tr>
<tr>
<td>%SINGLE_INST_LOOP</td>
<td>handle a loop with a single instruction</td>
</tr>
<tr>
<td>%SINGLE_INST_LOOP_END</td>
<td>handle the end of a single instruction loop</td>
</tr>
<tr>
<td>%ERROR</td>
<td>handle erroneous signal combinations</td>
</tr>
<tr>
<td>%INC</td>
<td>increment the PC to the next instruction</td>
</tr>
</tbody>
</table>
9.23.2 Instruction Decoder Template

An instruction decoder template is used if NoGap shall help with the design of the instruction decoder. Listing 9.15 shows an example of a decoder template. The two template parameters INSTRUCTION and FLUSH should be bound to the signals representing instructions and the flush signal. An example of a decoder instantiation can be seen in Listing 9.16, the actual decoder is instantiated on line 1, then on line 7–9 the actual operation (op_i), jump signal (jump_o), and flush signal (flush_i), are assigned from/to the decoder. As can be seen on line 3 the \texttt{dec_unit.alu\_op} is selected for the operation type. It should be noted that a decoder belongs to the data path in NoGap it is therefore required to connect the decoder the proper way for all operations. NoGap will then synthesize the control bridge according to the decoder template and the constructed data path. The instruction decoder template makes use of instruction declarations, described in Section 9.24.

\textbf{Listing 9.16:} Decoder instantiation example

```plaintext
1 f u::decoder\_spec< instr\_i, flush\_i >() dec\_unit;
2 ...;
3 operation(long\_pipe) out(dec\_unit.alu\_op)
4 {
5 @DE;
6 dec\_unit;
7 dec\_unit.instr\_i = op\_i;
8 jump\_o = dec\_unit.jump\_o;
9 dec\_unit.flush\_i = flush\_i;
10 ...;
11 }
```
Listing 9.15: Instruction declaration example

```cpp
fu::template<INSTRUCTION,FLUSH> decoder_spec
{
  input auto("#") INSTRUCTION;
  input auto("#") FLUSH;

  output [4:0] rf_a;
  output [4:0] rf_b;
  output [4:0] rf_w;
  ...

  immediate [4:0] imm_rf_a;
  immediate [4:0] imm_rf_b;
  immediate [4:0] imm_rf_w;

  instruction alu_op
  {
    source
    {
      rf_a = imm_rf_a;
      rf_b = imm_rf_b;
    }
    destination
    {
      rf_w = imm_rf_w;
    }
    jump_o = 0;
    ret_o = 0;
  }
}
```
Chapter 9: NoGap Common Language

9.24 Instruction Declarations

**Grammar 9.15** Instruction declaration grammar

\[
\begin{align*}
\langle \text{instruction_cons} \rangle &\rightarrow \text{instruction} \ (\text{instr_id}) \ (\text{statement}) \\
\langle \text{source_cons} \rangle &\rightarrow \text{source} \ (\text{statement}) \\
\langle \text{destination_cons} \rangle &\rightarrow \text{destination} \ (\text{statement}) \\
\langle \text{immediate_def} \rangle &\rightarrow \text{immediate} \ (\text{size}) \ (\text{imm_id}) \\
\langle \text{simple_assign} \rangle &\rightarrow (\text{port_id}) = (\text{simple_rhs}) \\
\langle \text{simple_rhs} \rangle &\rightarrow (\text{imm_id}) \mid \text{constant number} \\
\langle \text{instr_id} \rangle &\rightarrow \text{identifier} \\
\langle \text{imm_id} \rangle &\rightarrow \text{identifier} \\
\langle \text{port_id} \rangle &\rightarrow \text{identifier}
\end{align*}
\]

Instruction declarations are used in instruction decoder templates to specify what parts of the instruction that are needed. An \langle instruction_cons \rangle needs a clause with a list of \langle source_cons \rangle, \langle destination_cons \rangle, or \langle simple_assign \rangle statements. A \langle source_cons \rangle or \langle destination_cons \rangle, can only be followed by one or a clause of \langle simple_assign \rangle statements. An important component of the instruction declaration is the immediate type declarations. Defining an immediate, like imm_rf_a on line 10 in Listing 9.15 creates a placeholder for immediate data. An instruction declaration crates an instruction type to be used by one or more operation in a Mase FU. An example of an instruction declaration can be seen in Listing 9.15 on line 14–27.

The instruction declaration consists of source/destination blocks and simple constant assignments. The source and destination declarations marks immediate data fields as either source or destination fields, information which is important for assembler spawners. Assigning an immediate place holder to an output port of a decoder, e.g. as on line 18, tells NoGap that for this particular instruction type there has to be a
five bit immediate in the instruction word that shall be sent out through port rf_a of the final decoder. It is possible to use the same immediate place holder for more than one output port in a single instruction declaration. Doing so means that both these ports will output the same immediate value. The size of the immediate place holder declares how large the immediate field needs to be. Simple constant assignments are used to set a fixed value to some ports for certain instruction declarations. For example the assignment of jump_o on line 25, denotes that for an operation using this particular instruction type port jump_o of the decoder will always be zero.
Chapter 9: NoGap Common Language

9.25 NoGap System Call

Grammar 9.16 NoGap directive grammar

\[
\text{nogap} :: \text{<function_call>}
\]

\[
\text{<function_call>} \rightarrow \text{<function_id>} \left( \text{<function_arg_list>} \right)
\]

\[
\text{<function_id>} \rightarrow \text{<identifier>}
\]

A NoGap system call, is a function call intended for the underlying NoGap system, rather than having any functionality for describing the hardware. An example of such a NoGap system call is the \text{meminit} call, shown in Listing 9.17 on line 7. The \text{meminit} NoGap system call is used to give memory elements predefined data during simulation. Grammar 9.16 shows the related grammar, the \text{<function_arg_list>} is a comma separated lists of arguments. The currently available NoGap system calls are presented in Table 9.6.

Listing 9.17: NoGap call example

```
fu nen{
  input [7:0] adr_i;
  output [2:0] dat_o;

  signal auto("dat_o"):auto("adr_i-2") memory;
  nogap::meminit(memory,"dp_mem_init.bin","b");

  comb %NOP
  { 
    dat_o = memory[adr_i];
  }
}
```
Table 9.6: NoGap system calls

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>halt</td>
<td>Halts a simulation run when reached.</td>
</tr>
<tr>
<td>meminit(a,b,c)</td>
<td>Initialization values for memory elements during simulation.</td>
</tr>
<tr>
<td></td>
<td>a: name of memory element, e.g. mem</td>
</tr>
<tr>
<td></td>
<td>b: name of data file as a string, e.g. &quot;mem.dat&quot;</td>
</tr>
<tr>
<td></td>
<td>c: data type in the data file, &quot;b&quot; for ASCII-binary, &quot;h&quot; for ASCII-hex, and &quot;B&quot; for binary.</td>
</tr>
</tbody>
</table>
Dynamic Bus and Port Sizing

The competent programmer is fully aware of the limited size of his own skull. He therefore approaches his task with full humility, and avoids clever tricks like the plague.

Edsger Dijkstra

10.1 Introduction to Dynamic Port Sizing

\texttt{NoGap} can dynamically determine the size of wires and ports, even if the input port sizes to a \texttt{Masc} FU are not known at compile time. This extends the functionality already existing in Verilog and VHDL with parameters and generics respectively. While the parameterized modules can have their port sizes set from the outside, they can not determine their own port sizes depending on where in the data path they end up. The dynamic port sizing in \texttt{NoGap} ensures that adding an instruction
will not break already existing functionality even if the same FU is used by other instructions with hardware multiplexing.

A Mage description using dynamic input port sizing will usually require dynamic output port sizing as well. The output port sizes will be set during the construction of the Mase graph. The size of an output port can be written as a function of one or several input port sizes. Apart from the normal four operator (add, subtract, divide, and multiply) two additional operators can be used; binary maximum ($\text{\$}$) and binary minimum ($\text{\@}$), for example, in Listing 10.1 two possible port interfaces are shown. Note the output port expression $(a \_i \text{\$} b \_i) + 1$, where $a \_i$ and $b \_i$ are the input port sizes.

The sizing problem is complicated by the fact that a Mase graph might contain port size dependency loops, i.e. port size expressions that in one way or another depends on itself. A more detailed discussion about loops in the Mase graph can be found in Section 7.6. The algorithm presented in this thesis is able to handle these loops in a consistent manner.

The sizing algorithm is split into two main phases. First an annotation phase and then a size solving phase. The annotation phase annotates all sizes with either values or symbolic expressions if no numeric value is available. The size solving phase then uses the result from the annotation phase and resolves all dependencies, loops are resolved using a simple heuristic, described in Section 10.2.3.

The sizing algorithm operates on Mase graphs, Mage FUs therefore needs to be connected in a Mase graph to be affected by the sizing algorithm. Mase graphs in turn might also have dynamic input port sizes, in this case the input port sizes are set to a symbolic values and all sizing computations must be done with symbolic expressions. To simplify the computations needed for solving the sizing problem a Mase Size Relation Graph (Mase$_{\text{relation}}$) is created, where only the relevant size expression relations are exposed.
Chapter 10: Dynamic Bus and Port Sizing

Listing 10.1: FU port sizing example

```
1 f u m a s e _ e x a m p l e
2 {
3   i n p u t a u t o ( "#" ) a _ i ;
4   i n p u t [ 2 : 0 ] b _ i ;
5   i n p u t a u t o ( "#" ) op _ i ;
6   o u t p u t a u t o ( "( a _ i \& b _ i )+1" ) c _ o ;
7 ...
8 }

f u m a s e _ e x a m p l e
10 {
11   i n p u t a u t o ( "#" ) a _ i ;
12   i n p u t a u t o ( "#" ) b _ i ;
13   o u t p u t a u t o ( "#" ) c _ o ;
14 ...
15 }
```

10.2 The Sizing Algorithm

The overall algorithm for sizing ports can be summarized as follows.

1. Make an initial annotation in the `Mase` graph of all edges to be sized, as described in Section 10.2.2.

2. Build a graph of all non trivial relations.

3. Reduce the graph for each non trivial symbol, as described in Section 10.2.3.

4. Set edge sizes in the `Mase` graph according to the reduced graphs.

The `#` is used to denote the size of the bus coming into this port. `Mage` descriptions can use the `#` for both their input and output port width expressions. `Mage` FUs using dynamic input port sizing will probably also require dynamic output port sizing as well. The size of an output port can be written as a function of one or several input port sizes.
Two new operators can be used when specifying an automatic port size; binary maximum ($\|$) and binary minimum ($\@\$), e.g. an adder would probably have the output port expression set to $(o\alpha)p + 1$, where $o\alpha$ and $o\beta$ are the input port sizes. Listing 10.1 shows an example how these features are implemented in NoGap\textsuperscript{CL}, on line 3 the port $a_{\_i}$ is set to have the same width as a bus connected to this port. On line 6, the output port $c_{\_o}$ is set according to an expression that depends on the size of the two input ports; $a_{\_i}$ and $b_{\_i}$. In the same listing, on line 14, it is shown how also output ports in Mase graphs can be set to $\#$, in this case it means that the output port will be set to the size of the bus going to it from inside the Mase.

A small example of how the bus sizing works is shown in Figure 10.1, where text in square brackets represents the size of the signal and text in curly braces represent output expressions. The input ports of the adder are named $i1$ and $i2$, the input ports of the multiplexer are named $A$ and $S$. Remember that: $\alpha\$\beta = \max(\alpha, \beta)$. 

\begin{figure}[h]
\centering
\includegraphics[width=0.4\textwidth]{figure10.1.png}
\caption{Small sizing example}
\end{figure}
Chapter 10: Dynamic Bus and Port Sizing

The dynamic sizing functionality presents two problems. First if one or several \texttt{Mase} inputs are dynamically sized, i.e. its size is not known at compile time, symbolic size expressions have to be computed. Secondly a \texttt{Mase} graph might contain loops that must be handled correctly. The loop problem makes it impossible to use a direct solution. Without loops a recursive solution starting at the output ports would set all sizes correctly. We propose a method that solves the sizing problem in a two phased approach. The annotation phase, presented in Section 10.2.2 and the solver phase, presented in Section 10.2.3.

Some extra care must be taken to handle inline expressions and inserted multiplexers. These issues are outlined in Section 9.19.1 and Section 10.2.1, respectively.

10.2.1 Hardware Multiplexing Multiplexers

The synthesized multiplexers, used for hardware multiplexing of operations are assigned the output function of $\max(S(i_0), S(i_1), \ldots, S(i_n))$, where $i_\xi$ is input port number $\xi$.

10.2.2 Annotation Phase

In the annotation phase all edges in the \texttt{Mase} graph are assigned two size expressions one called the source expression and one called the target expression. Source expressions are set when processing output nodes and target expressions are set when processing input nodes. The algorithm processes the \texttt{Mase} graph with a wave front traversal. Figure 10.2 shows how this algorithm traverse a simple \texttt{Mase} graph. The dotted line bubbles shows the different active node sets. The numbers in the active node sets show the order in which they are considered. Each edge has two dots a source and a target dot. If the dot is black, it has been assigned a valid expression, if the dot is white, it has been assigned a new symbol, also called an introduced symbol.
The wave front traversal starts from a set of initial $\mathcal{V}_{\text{Mase}}$s, in specific the following node types are used for the start set:

- `ncg::info::GlobalIn`
- `ncg::info::SignalNode`
- `ncg::info::InsertedFlipFlop`
- `ncg::info::MuxControl`
- `ncg::info::FuOutPort` if the FU has no input ports.

Note that the entire $\mathcal{Mase}$ graph is searched when this $\mathcal{V}_{\text{Mase}}$ set is constructed.

If the traversal reaches an FU input port, it checks all other input ports of the found FU and sets the sizes according to Algorithm 10.1. In short, $\mathcal{NoGap}$ checks if the other input port sizes can be found from
Algorithm 10.1: FU input port processing

input: FU input port \( p_{in} \)

\[
P_{in} \leftarrow \text{all_fu_input_ports}(p_{in});
\]

\[
\text{foreach } p \text{ in } P_{in} \text{ do}
\]

\[
e_{in} \leftarrow \text{in_edge_of}(p);
\]

\[
\text{if target_expression}(e_{in}) = \emptyset \land \text{source_expression}(e_{in}) \neq \emptyset \text{ then}
\]

\[
\text{target_expression}(e_{in}) \leftarrow \text{source_expression}(e_{in});
\]

\[
\text{end if}
\]

\[
\text{else if target_expression}(e_{in}) = \emptyset \text{ then}
\]

\[
\text{target_expression}(e_{in}) \leftarrow \text{new_symbol()};
\]

\[
\text{end if}
\]

\[
\text{end foreach}
\]

already calculated sizes. If an already calculated size can not be found, the input port size is set to a new symbolic value.

Synthesized multiplexers are handled in the same way as FUs. The reason for doing this is to avoid complications if there are dependency loops in the \( \text{Mase} \) graph. This way the annotation run is always stable, independent of loops in the \( \text{Mase} \) graph.

Processing FU input nodes does not result in any new expression. When processing an input node, the expression is just propagated in the graph. FU output nodes can result in new expressions, if the output node is automatically sized it should have an output expression that depends on the sizes of some input nodes. The new expression for the output node is found by substituting the relevant input expressions into the output expression. The top level annotation algorithm is presented in Algorithm 10.2.
Algorithm 10.2: Size annotation algorithm

**input:** initial node set $N_{init}$

$N_{active} \leftarrow N_{init}$;

**while** $N_{active} \neq \emptyset$ **do**

$N_{next} \leftarrow \emptyset$;

**foreach** $n$ in $N_{active}$ **do**

    **if** $n$ is output node **then**
        **foreach** $e$ in out-edges from $n$ **do**
            $e.size_{source} \leftarrow n.size$;
        **end** **foreach**
    **end if**

    **else if** $n$ is input node **then**
        **foreach** $e$ in in-edges to $n$ **do**
            **if** $e.size_{source} \neq \emptyset$ **then**
                $e.size_{target} \leftarrow e.size_{source}$;
            **end if**
            **else if** $e.size_{source} \equiv \emptyset$ **then**
                $e.size_{target} \leftarrow$ new_symbol();
            **end if**
        **end** **foreach**
    **end if**

    $N_{next} \leftarrow N_{next} \cup next_nodes(n)$;

**end** **foreach**

$N_{active} \leftarrow N_{next}$;

**end while**
It should now be possible to understand what is going on in Figure 10.2.

1. The global input ports are processed since they constituted the initial $V_{\text{Set}}$ set. Their respective source expressions are set.

2. The FU input ports of three different FUs are processed according to Algorithm 10.1. In this stage two of the FUs have input ports that need to be set to symbolic values, since their respective source expressions are empty at this stage.

3. The output port expressions for all output ports of the three FUs are calculated and set as the output ports source expressions.

4. The global output ports are processed and their respective target expressions are set from the relevant FU output ports.

### 10.2.3 Solver Phase

When the annotation phase is completed, all $E_{\text{Set}}$ are annotated with one source expression and one target expression. As an effect of Algorithm 10.2, four types of relations can arise, these are according to Equation 10.1–10.4.

\begin{align*}
\text{Numericvalue} &= \text{Numericvalue} \\
\text{IntroducedSymbol} &= \text{Numericvalue} \\
\text{IntroducedSymbol} &= \text{Symbolicfunction} \\
\text{Symbolicfunction} &= \text{Symbolicfunction}
\end{align*} \tag{10.1–10.4}

The relations in Equation 10.1 and Equation 10.4 do not cause any problem. By design Right Hand Side (RHS) should be equal to Left Hand Side (LHS), if they are not equal a critical error has occurred. The relation in Equation 10.2 is also easy to solve, since it means that
the introduced symbol is in reality a numeric value and thus the symbol is directly substituted with its numeric value. It is the relation in Equation 10.3 that are most problematic. Therefore the main part of the solver is dedicated to solve this relation and the loops that it can cause.

The first step in the solver is to create a relation graph for all non trivial relations. Figure 10.3 shows an example of a small relation graph. The graph is also be mathematically described according to Equation (10.5)–(10.6). The relation graph is built with two types of vertices, Relation Graph Target Vertices (\(V_{rel}^{trg}\)) (\(T:\)) and Relation Graph Source Vertices (\(V_{rel}^{src}\)) (\(S:\)), corresponding to the target and source expressions of the \(E_{\text{Mases}}\). Thus two nodes are created for each \(E_{\text{Mases}}\), where the \(V_{rel}^{trg}\) points to the associated \(V_{rel}^{src}\). The value of a \(V_{rel}^{trg}\) are either a numeric value, a symbolic expression, or a single introduced symbol, while the value of a \(V_{rel}^{src}\) might be either a numeric value or a symbolic expression.

Algorithm 10.3 outlines the process with which a relation graph is constructed. First \(V_{rel}^{trg}\) for all symbols are created, then \(V_{rel}^{src}\) for all symbolic expressions are created. Relation Graph Edges (\(E_{rel}\)) are added from the symbolic expression to its participating symbols. Finally an \(E_{rel}\) is added, going from the RHS symbol to the LHS symbolic expression.
Chapter 10: Dynamic Bus and Port Sizing

Algorithm 10.3: Relation graph construction

\textbf{input:} Mass size relations $R$

$R_s = \text{all\_symbolic\_relations\_of}(R)$;

$S = \text{all\_symbols\_of}(R_s)$;

\textbf{foreach} symbol $s$ in $S$ \textbf{do}

\hspace{1em} add\_vertex\_for(s);

\textbf{end foreach}

\textbf{foreach} relation $r$ in $R_s$ \textbf{do}

\hspace{1em} $v = \text{add\_vertex\_for(rhs(r))}$;

\hspace{1em} \textbf{foreach} symbol $s$ in rhs($r$) \textbf{do}

\hspace{2em} add\_edge(v \text{ to vertex\_of}(s))

\hspace{1em} \textbf{end foreach}

\hspace{1em} add\_edge($\text{vertex\_of(lhs(r)) to v}$)

\textbf{end foreach}

\textbf{Figure 10.3:} Simple relation graph

![Simple relation graph](image-url)
A heuristic is employed to handle loops. The first step is to use the relation graph to compute the maximum substitution for each function. The algorithm for finding the maximum substitution for a relation graph is given in Algorithm 10.4. All looped expression has to be solved simultaneously since hardware loops can be parallel and simultaneously dependent on each other. A simple example of this is depicted in Figure 10.5. After the algorithm has been run, for symbol6, on the example graph in Figure 10.3, the new relation graph reached is presented in Figure 10.4.
Figure 10.4: Simple relation graph reduced for symbol6.
Figure 10.5: Parallel dependent loops

Figure 10.6: Side loop

Figure 10.7: Loop in loop
Chapter 10: Dynamic Bus and Port Sizing

When Algorithm 10.3 algorithm is run for both symbol7 and symbol6, Equation (10.7)–(10.8) can be extracted from the graph.

\[ S_7 = f(S_6, b_i, c_i) \]  \hspace{1cm} (10.5)

\[ S_6 = h(S_7, a_i, b_i) \]  \hspace{1cm} (10.6)

\[ S_7 = f(h(S_7, a_i, b_i), b_i, c_i) \]  \hspace{1cm} (10.7)

\[ S_6 = h(f(S_6, b_i, c_i), a_i, b_i) \]  \hspace{1cm} (10.8)

\[ S_7 = f(h(0, a_i, b_i), b_i, c_i) \]  \hspace{1cm} (10.9)

\[ S_6 = h(f(0, b_i, c_i), a_i, b_i) \]  \hspace{1cm} (10.10)

This algorithm will correctly handle complex cases, such as loops within loops (Figure 10.7) and side loops (Figure 10.6). At this point there is no real solution since there are still self references in the set of equations. This is solved with a simple heuristic; assume that from the start all symbols starts a loop with no data and thus have a size of zero. Thus, after the first iteration in the loop, the sizes of \( S_7 \) and \( S_6 \) are according to Equation (10.9)–(10.10). Since all size expressions in the rest of the \texttt{M}ase graph is constructed from either constant input sizes, symbolic input sizes, or introduced symbols, substituting \( S_6 \) and \( S_7 \) with their looped expression will yield the correct result.

The algorithm presented here does only a single iteration in all looped expressions. It is desirable to be able to specify a maximum number of iterations that a connection should be used. The algorithm then needs to be modified so as instead of disallowing all substitutions of already substituted variables (this happens when a loop in the graph has been completed). A variable should be allowed to be substituted as many times as the minimum allowed iterations in a loop.

Please note that numeric sizes always take precedence over symbolic expressions. There will thus be no iteration calculations for numeric sizes.
Algorithm 10.4: Maximum substitution algorithm

**input:** Relation graph $G$

**input:** Function vertex $v_f$

**while** More substitutions to do **do**

$V_{symbols} = \text{next_vertices}(v_f)$;

**foreach** $v$ in $V_{symbols}$ **do**

**if** out_degree($v$) > 0 **then**

$V_{function} = \text{get_function_vertices}(v)$;

**if** substituted($v$.symbol) && !has_direct_loops($v$, $V_{function}$) **then**

substitute ($v$.symbol with $V_{function}$ in $v$.size_expression);

$V_{new-symbols} = \text{next_vertices}(V_{function})$;

**foreach** $nv$ in $V_{new-symbols}$ **do**

add_unique_edge(source:$v_f$, target:$nv$);

end foreach

remove_edge(source:$v_f$, target:$v$);

end if

else

No more substitutions possible (end loop);

end if

end foreach

end while
11

Pipeliner Generation

There are no big problems, there are just a lot of little problems.

Henry Ford

11.1 Controlling Mage FUs

As NoGap’s central design principle is compositional design. Individual Mage FUs can not be aware of what an operation is, an FU should be usable as either an instruction driven module, e.g. an ALU with an operation select input, to be used in the operation construct in Mage descriptions. Or the FU should be usable as a normal hardware module, perhaps as sub modules in a Mage description. For this reason an FU can not have any special operation select constructs, but there still needs to be a way to access different code parts depending on different inputs to the FU. A novel approach is used to solve this dilemma in NoGap. We call this approach Dynamic Clause Selection (DCS) and it is further described in Section 11.2.
11.2 Dynamic Clause Selection

Dynamic Clause Selection (DCS) is used to make FUs behave as operation controlled FUs.

DCS works as follows. As described in Section 9.9, clauses can be named. An FU with named clauses can be used in an operation construct where the clause to be accessed can be specified. NoGap will then calculate which, and how, input signals needs to be set to reach that specific clause. This problem then comes down to a satisfiability problem and is as such NP-complete. But any practical module will have a limited set of problems to solve, thus keeping this approach practical. If the problem becomes too complex to solve in a reasonable amount of time, the FU is considered to be illegal for DCS. Note that this is solved once for each FU and only affects compile times. Some cases are also unsatisfiable, e.g. if a clause is unreachable or the clause can not be uniquely activated, thus not every FU is a legal dynamic clause selection FU. But a large set of FUs can be designed to be legal dynamic clause selection FUs. The input ports used to control clause access will be marked as control inputs and becomes part of the synthesized control path. The ports marked as control inputs can never be used as data input ports.

11.3 Data Paths and Control Paths

Immediate data signals and control signals are handled in different ways in a $\mathcal{M}$as$\mathcal{E}$ graph. Immediate data signals from the decoder is added into the $\mathcal{M}$as$\mathcal{E}$ as any other signal with the appropriate delays. Control signals including the control signals for multiplexers however are directly connected to their respective FU control ports and therefore needs to be delayed after the decoder and perhaps multiplexed if different operations that use the same FU but at different times. These pipeline timing
Chapter 11: Pipeliner Generation

multiplexers are controlled by pipeliner $V_{\text{Mates}}$. Section 11.6.

A $\text{Mage}$ FU will have data inputs and possibly also control inputs. The control inputs to a $\text{Mage}$ FU are the ports needed to satisfy the DCS for all instructions in the system. Two resources need to be controlled to correctly execute the different instructions; the clause selection input ports of $\text{Mage}$ FUs and the multiplexers controlling delay path selection and hardware multiplexing.

11.4 Instruction Format Generation

Instruction formats for the data path are automatically generated by NoGap. The instruction format in NoGap can have three types of fields, control, immediate, and padding. Control fields are used to encode a wider set of control signals for the data path, i.e. the instruction opcode. Immediate fields are used for all instruction fields containing data, e.g. constant data or register file addresses. The padding fields are used to pad bits in instruction formats not using all bits.

The instruction format generation requires directions from the designer about the number and size of the immediate formats needed for an instruction. Together with the operations in the data path a minimal set of needed instruction formats are generated.

11.5 Decoder Generation

The bridge between the generated instruction formats and the data path is the decoder. The decoder uses the generated instruction format to either generate the correct control signals to the data path based on the operation code or slice of bits of the instruction word as immediate data for the data path.
11.6 Pipeliner Generation

\textit{NoGap} assumes that the decoder can decode an instruction in one clock cycle. But since the data path will usually be pipelined, a layer between the decoder and the data path needs to be constructed, this layer is called the pipeliner in \textit{NoGap}. The pipeliner essentially consists of two parts, delay lines and class selectors. Each instruction is assigned an operation class, further described in Section 11.6.1, several instructions can be assigned to the same operation class. As part of decoding an instruction, the decoder generates the instruction’s corresponding pipeline class identifier. The operation class is delayed through as many FFs as there are stages in the data path pipeline. At the end of the operation class delay line, is one class selector unit per control port output. This is depicted in Figure 11.1, for control port \( P \). Port \( OC \) from the decoder outputs the current operation class identifier. Observe that there is only one operation class output from the decoder regardless of how many control outputs there are from the decoder.

Each controllable resource in the data path needs its own class selection logic. \textit{NoGap} considers a controllable resource in the data path to be either a \textit{Mage} FU input port used for clause access or inserted multiplexers.

For example, a system with at least three instructions, using a controllable resource in three different stages of the pipeline, will have at least three different operation classes, e.g. C1, C2, and C3 where C1 uses the resource at stage one, C2 at stage two, and C3 at stage three. A picture of how \textit{NoGap} would synthesizes the pipeliner layer for this example is depicted in Figure 11.1. The class selection nodes then prioritize the longest instruction as the one to get control of the resource in question. This means instructions using the same resource sooner in the pipeline will be shadowed by longer instructions. This normally translates to that the first started instruction will be allowed to complete uninterrupted.
Table 11.1 gives examples of which class is given priority. The first six examples are trivial, the two last shows the priority order, where first a $C_1$ and $C_2$ operation is shadowed by a $C_3$ operation, and finally how a $C_2$ operation shadows a $C_1$ operation. Note that the classes will progress down a pipeline as deep as the deepest depth of the data path pipeline. In our example, a $C_1$ class is only valid in the first position of the vector. This, since it uses the FU at this stage in the pipeline, a $C_2$ class will only be valid in the second stage and so on. This class prioritization
needs to be done for each pipeliner node, since there is no guarantee that all control signals for different FUs over different operations are in the same sequence. One operation might use FU A before FU B whereas another might use B before A.

An example of a small Masr with an inserted pipeliner is shown in Figure 11.2.

11.6.1 Operation Classification

For the pipeliner to work, it needs information about the current instruction’s pipeline architecture. For this reason the instruction decoder also generates an operation class identifier. To save hardware and thus lessen the risk that NoGap-generated components are part of the critical path, we developed a method by which a number of operations could be classified as having the same pipeline architecture class. We call this method operation classification. A key concept in operation classification is the operation mergeability. Algorithm 11.1 is used to determine operation mergeability. If two operations are mergeable they are said to belong to the same operation class. There might exist several valid partitionings among a set of instructions. NoGap uses a greedy approach to solve this problem. Future research will determine if a more optimized method can improve performance of the generated hardware.
Figure 11.2: Generated pipelining layout.
11.6.2 Pipeline Usage Vector

The pipeline usage vector is a central concept of the operation classification process. The pipeline usage vector can best be described as a two-dimensional map over when which resource is used for a particular operation. The resources in this case are inserted multiplexers and FU input ports used for clause access. An illustrative example of two pipeline usage vectors is shown in Figure 11.3, where the × mark when a resource is used, e.g., in Figure 11.3.1, $\text{FU}_1.\text{op}_i$ is used at time step 0. If a column has no × mark then that resource is not used in the operation in question, e.g., $\text{FU}_2.\text{op}_i$ in Figure 11.3.2.

It is easy to realize that two instructions can be merged if all their resource usages are exactly the same and that they are not mergeable if one or more resources are used at different stages in the pipeline in the different operations. The question here is if one operation $O_1$ has an empty slots, i.e., it does not use a particular resource, but the other operation $O_2$ makes use of this resource. Can then this used resource in $O_2$ be merged into the empty slot of $O_1$? It turns out that the answer is that it can be merged if the resource in question in $O_2$ is at its earliest possible usage time.

The reason for this is that if a resource that is not used at its earliest usage time in the pipeline is merged into an empty usage slot of an operation it can have unforeseen consequences. Say that resource $A$ can be used in pipeline stage two and three. If operation $O_1$, which does
Algorithm 11.1: Operation mergability

input: Op A resource usage vector ru1
input: Op B resource usage vector ru2
input: Base usage vector baseUsage

foreach Resource r in baseUsage do
  earliestTime = earliest_time_in_pipeline(r);
  if r is in both ru1 and ru2 then
    if time of r in ru1 != time of r in ru2 then
      return false;
    end if
  else if r is only in ru1 then
    if time of r in ru1 != earliestTime then
      return false;
    end if
  else if r is only in ru2 then
    if time of r in ru2 != earliestTime then
      return false;
    end if
  end if
end foreach
return true;
not need to use \( A \) and therefore has an empty resource usage slot for \( A \), belongs to operation class \( C_1 \) which puts \( A \) in pipeline stage three. And operation \( O_2 \) has to use \( A \) and belongs to operation class \( C_2 \) which puts \( A \) at pipeline stage two. Then if \( O_1 \) is first executed and then \( O_2 \) there will be a resource contention for \( A \). But since \( O_1 \) should not use \( A \), this resource contention should not arise and can not be foreseen by a programmer. Therefore including \( O_1 \) into operation class \( C_1 \) is clearly illegal. However if instructions with empty usage slots are included into an operation class which has resource usages for these empty slots, but the resource is used at its earliest possible time in the pipeline, the problem previously described can not arise. Due to that, no other operation can use a resource earlier than its earliest possible usage time.
Part IV

Spawners
12

Generators and Spawners

I do not think there is any thrill that can go through the human heart like that felt by the inventor as he sees some creation of the brain unfolding to success... such emotions make a man forget food, sleep, friends, love, everything.

Nikola Tesla

12.1 Introduction

It is important to understand the distinction between a spawner and a generator. A generator performs a single specific task, while a spawner uses one or more generators to accomplish its task. The spawner is a design concept not directly represented in code, while the generators have a clear defined coding structure. So in order to understand the spawners, one has to understand the generator system.
12.2 Generator System

To manage all tasks the NoGap compiler has to perform, while still avoiding and strong dependencies between software components, NoGap uses, what we call, a generator system.

A generator is a software component that processes PUs. The first step taken by a generator is to test if a PU, given to it, is eligible for processing. If it is, the generator goes ahead and performs its tasks. A generator may or may not modify the PU it uses for processing. An example of a modifying generator is the Mase graph construction generator and an example of a non modifying generator is the dot printer generator. A generator might also create new PUs that are going to be processed by later generators. An example of a generator that creates new PUs is the Mase generator.

A simple example of the generator system is depicted in Figure 12.1, where a PU goes through three generators, G1, G2, and G3. G1 modifies the PU by generating some new data and writes this data back to the PU, G2 synthesizes a new PU, G3 output the ASTs in the dot-language.

It is important to know that one generator will process all available PUs before another generator starts its processing.

![Figure 12.1: Generator system](image-url)
A generator is implemented as a C++ class derived from the generator::Generator virtual base class, its class definition is shown in Listing 12.1. The most notable methods and their purpose are listed in Table 12.3. The available generators are listed in Table 12.1 and Table 12.2.

**Listing 12.1: Generator interface**

```cpp
class Generator {
    protected:
        virtual const bool generate_predicate(const parser::ParseUnit&) const;
        virtual void operator()(parser::ParseUnit& pu) = 0;
    public:
        Generator();
        virtual ~Generator();
        bool generate(parser::ParseUnit& pu);
        virtual void initialize();
        virtual void finalize();
        virtual void reset();
};
```

**Listing 12.2: Generate method**

```cpp
bool Generator::generate(parser::ParseUnit& pu)
{
    if(generate_predicate(pu))
    {
        operator()(pu);
        return true;
    }
    return false;
}
```
### Table 12.1: Available generators (A-M)

<table>
<thead>
<tr>
<th>Generator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AssemblerGenerator</td>
<td>Generates an assembler, see Chapter 14.</td>
</tr>
<tr>
<td>AstDot</td>
<td>Generates a dot graph for the AST.</td>
</tr>
<tr>
<td>BridgePipiliner</td>
<td>Inserts a pipeliner into the Mase, see Chapter 11.</td>
</tr>
<tr>
<td>ClauseAccessRules</td>
<td>Generates the clause access rules, see Section 5.4.</td>
</tr>
<tr>
<td>ControlBridgeGenerator</td>
<td>Inserts a decoder into the Mase.</td>
</tr>
<tr>
<td>ControlPathGenerator</td>
<td>Builds operation information needed for decoder construction.</td>
</tr>
<tr>
<td>FoldConstants</td>
<td>Fold constants in the AST.</td>
</tr>
<tr>
<td>Immediates</td>
<td>Builds immediate fields for the operations.</td>
</tr>
<tr>
<td>InstructionDecoder</td>
<td>Builds information for the instruction decoder, see Section 5.8.</td>
</tr>
<tr>
<td>MageDepDot</td>
<td>Generates a dot graph from a Mage dependency graph in the PU.</td>
</tr>
<tr>
<td>MageDepGraph</td>
<td>Generates a Mage dependency graph, see Section 15.2.</td>
</tr>
<tr>
<td>Mase</td>
<td>Builds a Mase from an input AST, see Chapter 7.</td>
</tr>
<tr>
<td>MaseDot</td>
<td>Generates a dot graph for the Mase in the PU.</td>
</tr>
<tr>
<td>MaseDotAllTransforms</td>
<td>Generates a dot graph from a fully transformed copy of the Mase in the PU, see Section 7.7.</td>
</tr>
</tbody>
</table>
## Table 12.2: Available generators (N-V)

<table>
<thead>
<tr>
<th>Generator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NeedsClockInput</td>
<td>Checks if the PU needs a clock input if instantiated in real hardware, see Section 5.10</td>
</tr>
<tr>
<td>OperationClassifier</td>
<td>Classifies operations into pipeline classes, see Section 11.6.1</td>
</tr>
<tr>
<td>Outstream</td>
<td>Serializes a PU to an <code>std::ostream</code></td>
</tr>
<tr>
<td>PcFsm</td>
<td>Generates a PC FSM, see Section 5.2.</td>
</tr>
<tr>
<td>Pipelines</td>
<td>Constructs the pipelines and resolves the timing of its phases, see Section 5.5.</td>
</tr>
<tr>
<td>SSP_Partition</td>
<td>Generates an SSP partition of the Mage, see Section 15.3.</td>
</tr>
<tr>
<td>SetFileName</td>
<td>Set the correct file name to all Mage files.</td>
</tr>
<tr>
<td>SimulatorGenerator</td>
<td>Generates the simulator driver program.</td>
</tr>
<tr>
<td>SimulatorMaseGenerator</td>
<td>Generates the simulator code for Mase graphs, see Chapter 15.</td>
</tr>
<tr>
<td>SymbolTable</td>
<td>Generates the symbol table, see Chapter 8.</td>
</tr>
<tr>
<td>ValidateIdentifiers</td>
<td>Validates identifiers in the AST.</td>
</tr>
<tr>
<td>VerilogGenerator</td>
<td>Generates System Verilog code for the PU, see Chapter 13.</td>
</tr>
</tbody>
</table>
### Table 12.3: Generator interface

<table>
<thead>
<tr>
<th>Line</th>
<th>Purpose</th>
<th>Default Operation</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Inspects the PU in the argument and returns true if the PU should be processed by the generator.</td>
<td>return true</td>
</tr>
<tr>
<td>6</td>
<td>Performs the actual generation</td>
<td>—</td>
</tr>
<tr>
<td>10</td>
<td>Entry point for application code, see Listing 12.2.</td>
<td>As described</td>
</tr>
<tr>
<td>11</td>
<td>Executed before any PU is processed by a particular generator.</td>
<td>Nothing</td>
</tr>
<tr>
<td>12</td>
<td>Executed after all PUs are processed by a particular generator.</td>
<td>Nothing</td>
</tr>
<tr>
<td>13</td>
<td>Executed before each PU is processed, mainly for resetting values for the next PU.</td>
<td>Nothing</td>
</tr>
</tbody>
</table>
12.3 Generator Register

To manage the generators, the `generator::Register` class, which is a collection of generators, is used. Initially generators have to be registered to the generator register. Listing 12.3 shows how the generator registration is implemented and later used. The `register_generator` method (line 1–8) shows code for how a generator is registered to the generator register. On line 14 the actual registration is performed, the single string argument gives the generator a name by which it can be found in the register. This name is important for the “flow file” system, described in Section 12.5. Note also how, on line 15, the just registered generator (md) is assigned some additional data. If and what type of additional data a generator expects, depends on how the generator is implemented. The function `default_registration` is a function used to create a default registration of the current available generators.

Listing 12.3: Generator register usage

```cpp
template<typename GENERATOR> GENERATOR&
Register::register_generator(const std::string& generator_id)
{
    register_m[generator_id] =
        boost::shared_ptr<GENERATOR>(new GENERATOR());
    return *(static_cast<GENERATOR*>(&(*(register_m[generator_id]))));
}

void default_registration(Register& reg)
{
    generator::MaseDot& md =
        reg.register_generator<generator::MaseDot>("dot_mase");
    md.set_output_path("mase_graphs");
    ...
}
```
12.4 Generator Usage

Listing 12.4 shows the intended way of using the generators and the generator register. The generator register is first instantiated and then assigned a default registration, as discussed in Section 12.3. Later, the `while` loop, executes all registered generators. The returned `std::vector<bool>`, consists of as many elements as processed PUs, if the an element is true that particular PU was valid for generation and had the current generator executed for it. The mechanics for determining if a PU is valid for generation is discussed in Section 12.2. Listing 12.5 show how a generator is applied to all currently available PUs. It is worth noting how, on line 6–8, a temporary vector of all current PUs is created, and then, on line 10–13, this temporary vector is used to process the PUs. This is done since the `parse_unit_map_m` is not a stable vector when the actual processing takes place, which is due to that some generators might add new PUs to `parse_unit_map_m`. The temporary vector approach ensures that only the PUs, that existed before the generation process of a particular generator, are processed and added PUs does not interfere with the process.

Listing 12.4: Generator usage

```cpp
1 generator::Register gen_reg;
2 generator::default_registration(gen_reg);
3 ...
4 while(!gen_reg_test.empty())
5 {
6   const generator::Register::value_type& g = gen_reg.top();
7   std::vector<bool> status =
8   parser::ParseData::get().generate_through_parse_units(*g);
9   gen_reg_test.pop();
10 }
```
Chapter 12: Generators and Spawners

Listing 12.5: Generate through PUs

```cpp
std::vector<bool> ParseData::generate_through_parse_units(generator::Generator& gen)
{
    std::vector<bool> ret_status;
    gen.initialize(); std::vector<ParseUnit*> gen_pus;
    BOOST_FOREACH(parse_unit_map_t::value_type & pu, parse_unit_map_n){
        gen_pus.push_back(pu.second);
    }
    BOOST_FOREACH(ParseUnit* pu, gen_pus){
        gen.reset();
        ret_status.push_back(gen.generate(*pu));
    }
    gen.finalize();
    return ret_status;
}
```

12.5 The Flow File

To control in which order the generators are executed a flow file is used. Listing 12.6 is an example of a flow file. The flow file syntax is simple, one directive per line. C++ style comments can be used, as seen on line 1. The directives are names of generators, where the name of the generator is set when it is registered to the generator register, as described in Section 12.3. The generators are then executed in the order in which they are listed in the flow file. The additional finish directive is also recognized and means that no more generators are to be executed. For example gen_construct_switch_codings on line 11 is not executed due to the previous finish directive.
12.6 The Generated Data Template

As can be seen in Listing 5.1, data that is generated for a PU is wrapped in the `generator::GeneratedData` template. This is used to keep track of which values have been initialized by and generated by other generators. The `GeneratedData` mimics a pointer for most part but must be explicitly initialized and cannot be deleted by application code.

12.7 NoGap Folders

In order to avoid destroying user created folders, `NoGap` tags all folder it creates by creating an empty sub folder within the created folder. The name of the sub folder follows the naming scheme `nogap_generator_name`, where `generator_name` is the name of the generator creating the folder. If a generator named `gen` wants to create a folder named `gen_output` the steps performed are outlined in Algorithm 12.1.
Algorithm 12.1: NoGap’s special folder creation and/or use

Look for a folder named gen_output;

if gen_output exist then
    if gen_output/nogap_gen exist then
        Use this folder and overwrite any data in it;
    end if
    else
        Report an error saying that folder can not be created;
    end if
else
    Create folder gen_output;
    Create folder gen_output/nogap_gen;
end if
13 SystemVerilog Spawner

People who are really serious about software should make their own hardware.

Alan Kay

13.1 Introduction

The SystemVerilog spawner is used to generate synthesizable SystemVerilog code from NoGapCD. The Verilog spawner corresponds to the generator::VerilogGenerator generator. There are however four different ways in which the final SystemVerilog code can be generated. An illustration of the SystemVerilog spawner flow is shown in Figure 13.1

- The PU models a PC-FSM, Section 13.2.
- The PU models a Mage, Section 13.3.
- The PU models an instruction decoder, Section 13.5.
- The PU models a Mage, Section 13.4.
The resulting SystemVerilog code is, at the time of writing, put in a directory called `verilog_code`.

Figure 13.1: SystemVerilog spawner flow

13.2 PC-FSM Verilog Generation

To generate the PC-FSM SystemVerilog code from the PC-FSM graph is a simple task using the pre-generated PC-FSM graph in the PU. More details about the PC-FSM description can be found in Section 5.2. The generated SystemVerilog code of the PC-FSM is designed according normal best practices for how synthesizable FSMs should be written. The state names are taken from the PC-FSM’s Vs. The default state can be generated in two ways, if the PC-FSM template defines a default state with the `default` keyword, the default state information will be constructed accordingly, otherwise the PC-FSM V with index zero will be the default state.

The combinational always blocks are generated from the state transfer condition and action information, which is represented by the PC-FSM graph’s Es. There will be two combinational always blocks. One
13.3 *Mase* Verilog Generation

13.3.1 Preparatory *Mase* Transformations

In order to use the *Mase* graph to generate HDL code, it has to be transformed so it better models real hardware. The following transformations are applied to the *Mase* graph.

1. Calculate and set the wire and bus sizes, further explained in Chapter 10.
2. Insert FFs, further explained in Section 7.7.2.
3. Combine equal FFs further explained in Section 7.7.3.
4. Connect stalling FFs further explained in Section 7.8.
5. Combine equal *E*Mase*s, further explained in Section 7.7.1.
6. Set wire names, further explained in Section 7.7.5.

13.3.2 Verilog Generation of Transformed *Mase*

The actual SystemVerilog generation is performed in the `ncg::Graph::genHDLCode` template method, using the `ncg::HDLwriter::VerilogWriter` as the template argument. `ncg::Graph::genHDLCode` works by first collecting all relevant *V*Mase*s in an `std::vector` and then calling the code generation method of a predefined sub class of the template argument class. An example of this is shown in Listing 13.1, which is responsible for generating the module’s interface. First, on line 8–20 the `igvl` collection is filled with the relevant *V*Mase*s. When the collection is filled an `InterfaceWriter`
object is instantiated on line 22, and then called to actually generate the SystemVerilog code.

The template argument `GENERATOR` is required to be a class with a number of sub classes. An example is given in Listing 13.2, where the `InterfaceWriter` used in Listing 13.1, is defined on line 12–20. All the required sub classes and their purpose are listed in Table 13.1, in which, the last column, in all cases except one, refers to line numbers found in Listing 13.3. These line numbers points to the code generated by the respective writer.

**Listing 13.1: HDL code generation method example**

```cpp
1 template<typename GENERATOR>
2 std::ostream& Graph::genHDLCode(std::ostream& ost) const
3 {
4     vertex_list_t igvl;
5     boost::tie(vi, vi_end) = vertices();
6     for (; vi != vi_end; ++vi)
7     {
8         const info::NodeInfo& ninf = theGraph_m[*vi];
9         if (ninf.getInfo()->is<info::GlobalIn>() ||
10             ninf.getInfo()->is<info::GlobalOut>())
11             { 
12                 igvl.push_back(*vi);
13             } 
14         else if (ninf.getInfo()->is<info::MuxControl>())
15             { 
16                 if (in_degree(*vi) == 0) igvl.push_back(*vi);
17             } 
18     } 
19     typename GENERATOR::InterfaceWriter iw(*this);
20     iw.generateCode(ost, igvl);
21     ..
22 }
```
Listing 13.2: HDL writer class example

```
struct VerilogWriter
{
    class Writer : public policy::CodeWritable
    {
    protected:
        const ncg::Graph& graph_m;
    public:
        Writer(const ncg::Graph& graph);
        ~Writer();
    ...

    class InterfaceWriter : public Writer
    {
    public:
        InterfaceWriter(const ncg::Graph& graph);
        ~InterfaceWriter();
        virtual std::ostream& generateCode(ostream& ost,
            const ncg::vertex_list_t& verList) const;
    ...

```

```
<table>
<thead>
<tr>
<th>Sub class</th>
<th>Purpose</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>PreambleWriter</td>
<td>Writes code to the beginning of the output.</td>
<td>1</td>
</tr>
<tr>
<td>InterfaceWriter</td>
<td>Writes code for the module interface, e.g. module name, input and output</td>
<td>2–11</td>
</tr>
<tr>
<td>FinalizationWriter</td>
<td>Writes code at the end of the output.</td>
<td>79</td>
</tr>
<tr>
<td>VariableDeclarationWriter</td>
<td>Writes code for variables, e.g. wire, reg, and the needed parameters.</td>
<td>12–19</td>
</tr>
<tr>
<td>FuWriter</td>
<td>Writes code for instantiated FU</td>
<td>29–35</td>
</tr>
<tr>
<td>InlineExpressionWriter</td>
<td>Writes code for inlined expressions.</td>
<td>37–43</td>
</tr>
<tr>
<td>GlobalOutWriter</td>
<td>Writes code for connections to output ports.</td>
<td>23–25</td>
</tr>
<tr>
<td>GlobalInWriter</td>
<td>Writes code for connections to input ports.</td>
<td>20–22</td>
</tr>
<tr>
<td>InsertedFlipFlopWriter</td>
<td>Writes code for inserted FFs.</td>
<td>72–78</td>
</tr>
<tr>
<td>InsertedMuxWriter</td>
<td>Writes code for inserted multiplexers.</td>
<td>44–61</td>
</tr>
<tr>
<td>PipelineClassControlWriter</td>
<td>Writes code for pipeline class controllers.</td>
<td>62–71</td>
</tr>
<tr>
<td>DecoderWriter</td>
<td>Writes code for decoders.</td>
<td>Section 13.5</td>
</tr>
<tr>
<td>SignalNodeWriter</td>
<td>Writes code for signal nodes.</td>
<td>26–28</td>
</tr>
</tbody>
</table>
Listing 13.3: Generated Mace SystemVerilog code

1 'default_nettype none
2 module data_path(
3 dat_o_wi,
4 ...
5 wr_en_i_wi, clk_i, rst_i);
6 parameter dat_o_wi__WIDTH = 16;
7 ...
8 input wire [dat_o_wi__WIDTH-1:0] dat_o_wi;
9 ...
10 output wire [wr_en_i_wi__WIDTH-1:0] wr_en_i_wi;
11 ...
12 parameter dat_o_wi_0_W__WIDTH = 16;
13 parameter __FF_58___0_W__WIDTH = 1;
14 ...
15 wire [dat_o_wi_0_W__WIDTH-1:0] dat_o_wi_0_W; 
16 wire [op_i_0_W__WIDTH-1:0] op_i_0_W;
17 reg [__MUX_0___0_W__WIDTH-1:0] __MUX_0___0_W;
18 reg [__MUX_1___0_W__WIDTH-1:0] __MUX_1___0_W;
19 ...
20 assign dat_o_wi_0_W = dat_o_wi; 
21 assign op_i_0_W = op_i; 
22 ...
23 assign wr_en_i_wi = dm_wr_en_i_wi_0_W;
24 assign dat_i_wi = dm_dat_i_wi_0_W;
25 ...
26 //SignalNode: temp
27 assign temp_0_W = status_reg_status_o_0_W;
28 ...
29 acc acc ( 
30 //Inputs
31 .macreg_i(__FF_15___0_W_),
32 .mul_i(__FF_35___0_W_),
33 .op_i(CS_acc_op_i_0_W_ ),
34 //Outputs
35 .res_o(acc_res_o_0_W);
36 ...
37 function [EXPR_res_o_5_W__WIDTH-1:0] inline_function_0;
38 input [__FF_7___0_W__WIDTH-1:0] alu_dot_res;
39 begin
40
41 end
42 endfunction 
43 assign EXPR_res_o_5_W = inline_function_0(__FF_7___0_W);
44 parameter EXPR_res_o_2_W_MIN___MUX_0___0_W =
Chapter 13: SystemVerilog Spawner

13.4 Mage Verilog Generation

The process of generating SystemVerilog code from a Mage is a straightforward task. This since Mage code in many ways resemble SystemVerilog code. The generation first ensures that the needed module interface is generated and then the needed internal variables and parameters are
written to the resulting SystemVerilog file. Then the rest of the AST is processed and depending on what type of node that is encountered, its corresponding code is generated. Listing 13.4 shows an excerpt of how this is done. First, on line 7–11, casts are performed to find what type the \( V_{\text{Mage}} \) is, then on line 15–23 code for handling a vase \( V_{\text{Mage}} \) is shown.

The reason for not using a polymorphic solution here is motivated in Section 4.2.

**Listing 13.4: \( V_{\text{Mage}} \) AST generation excerpt**

```cpp
bool VerilogGenerator::generate_tree(const parser::ast::vdesc_t & root, std::ostream& ost) const
{
    const parser::ast::Graph & ast = parse_unit_m.ast_m;
    const parser::ast::info::NodeInfo & ninf = ast[root];

    const parser::ast::info::Comb * comb =
        ninf.is_data<parser::ast::info::Comb>();

    const parser::ast::info::NogapCall * nogap_call =
        ninf.is_data<parser::ast::info::NogapCall>();

    bool default_traverse = true;

    if (comb)
    {
        parser::ast::Graph::vertexList_t nv = ast.nextVertices(root);
        ost << "always@*" << std::endl;
        current_code_env_m = code_env::COMB;
        generate_tree(nv.at(0), ost);
        current_code_env_m = code_env::GLOBAL;
        return false;
    }

    ... false;
}
```


13.5 Instruction Decoder Generation

The instruction decoder is generated much in the same way as a Magic FU, described in Section 13.4, when it comes to interface generation. However, the internal functionality is derived from arch_data::InstructionDecoder::mase_port_into_m, described in Section 5.8. Each line in the decoder port information table will give rise to one case statement for that particular port. For example, line 6 in Listing 5.8, will result in the code shown in Listing 13.5. NoGap does not try to optimize cases where it is obvious that all options give rise to the same output, this is instead left for the synthesis tools.

Listing 13.5: Generated decoder excerpt

```verilog
always_comb begin
  imm_b_o = 0;
  case(___op_code)
  3'd3: begin
    imm_b_o = instr[7:4];
  end
  3'd2: begin
    imm_b_o = instr[7:4];
  end
  3'd1: begin
    imm_b_o = instr[7:4];
  end
  3'd0: begin
    imm_b_o = instr[7:4];
  end
  end
```

...
When designing a processor, the actual hardware is just one part of the entire design process. Of equal importance is the supporting tool chain, i.e. simulators, assemblers, and compilers. Currently NoGap has not been aimed at compiler generation, although it would be an interesting research project. This chapter will describe the techniques used for constructing an assembler generator for NoGap. The assembler generator described in this chapter can generate an assembler for all types of processors that can be constructed with NoGap.

The implementation of the assembler spawner was done by Sumathi Loganathan and Faisal Akhlaq, as part of their master thesis project. Many of the design decisions were mine, but they did the actual implementation. More details about this work can therefore be found in [5].
Chapter 14: Assembler Spawner

14.1 NoGap Assembler

AsmGen, discussed in Section 14.2, generates NoGapAsm. The purpose of NoGapAsm is to convert an assembly program written by the user into machine code for a processor generated with NoGap.

NoGapAsm can handle assembly program written for any processor created with NoGap. This is due to the generic process used for assembler construction. A general purpose parser is used, which means that all assembly programs will have the same basic syntax. When a new processor is generated, the only thing that will be changed is the definition file.

One of the problems of implementing the assembler generator is that NoGapAsm has to be a stand alone executable. This means there must be a way for AsmGen to pass information to an as of yet non existing program. This problem was solved by storing the necessary information in intermediate files. This flow, as well as the flow of running NoGapAsm, is shown in Figure 14.1, where the top gray area represents components that are part of the NoGap executable, and the bottom gray area represents the components that are part of the NoGapAsm executable. AsmGen will extract instruction details from NoGapCD, and write this information to a definition file. The definition file is loaded by NoGapAsm into a data structure which is used by the parser to verify the assembly program. The parser and lexical analyzer are constructed using Bison [8] and Flex [34] respectively. The mnemonic alias, instruction table, and definition file are described in Section 14.2.

The processes of assembling an assembly program is done in two passes. In the first pass, the syntax of the input program is verified by the parser against the instruction details provided in the definition file. If the first pass is successful, labels are handled in the second pass, finally generating the binary for the processor.
Figure 14.1: NoGap assembler flow
14.2 NoGap Assembler Generator

The assembler generator is a NoGap spawner used to create assemblers for generated processors. The assembler generator spawner, also called AsmGen, extracts the needed information from NoGap CD. For each instruction the following will be extracted/generated:

- The mnemonic name.
- The operation code, its size and position in the instruction.
- Number of padding bits needed and their position in the instruction.
- Immediate field names, their respective sizes, and positions in the instruction.

This information is written to a definition file, using the Boost serialization package. The definition file is later used as input to NoGap Asm as discussed in Section 14.1.

AsmGen will generate the following three files as shown in Figure 14.1.

1. **Instruction table** containing all available assembly instructions, used as a reference for the user. Each assembly instruction is listed with its mnemonic and details about the operands.

2. **Definition file** containing definitions for the generated processors instructions.

3. **Mnemonic aliases** containing aliases for the automatically generated mnemonics. An alias can be used instead of the lengthy mnemonic generated from NoGap.

14.2.1 Lexical Analyzer

Flex is used to generate the lexical analyzer, which is the same for all generated processors. The lexical analyzer analyzes the assembly program
Chapter 14: Assembler Spawner

and produces tokens for the parser which is constructed using Bison. The
lexical analyzer, will generate generic tokens for the mnemonics, register
and label names. The generic tokens are the strength of NoGapAsm,
this way any new instruction can be incorporated without changing the
existing assembler programs.

14.2.2 Parser

Bison is used to generate the parser. The grammar rules for the as-
semble language is always the same, independent of the processor gen-
erated. The general grammar for the parser is simple and is shown in
Grammar 14.1.

\[
\text{Grammar 14.1 NoGapAsm grammar}
\]

\[
\begin{align*}
&\text{(statement)} \rightarrow \text{(label)} \mid \text{(directive)} \mid \text{(instruction)} \\
&\quad \mid \text{(newline)}
\end{align*}
\]

\[
\begin{align*}
&\text{(directive)} \rightarrow \text{(org)} \ \text{(number)} \ \text{(newline)}
\end{align*}
\]

\[
\begin{align*}
&\text{(label)} \rightarrow \text{string} : \ \text{(newline)}
\end{align*}
\]

\[
\begin{align*}
&\text{(instruction)} \rightarrow \text{mnemonic} \ \text{(oplist)} \ \text{(newline)}
\end{align*}
\]

\[
\begin{align*}
&\text{(mnemonic)} \rightarrow \text{string}
\end{align*}
\]

\[
\begin{align*}
&\text{(oplist)} \rightarrow \text{(oplist)} \ \text{(operand)} \mid \text{(operand)}
\end{align*}
\]

\[
\begin{align*}
&\text{(operand)} \rightarrow \text{string} : \ \text{(label)} \\
&\quad \mid \text{(string)} : \ \text{(number)}
\end{align*}
\]

\[
\begin{align*}
&\text{(org)} \rightarrow \text{.org}
\end{align*}
\]

\[
\begin{align*}
&\text{(newline)} \rightarrow \text{end of line}
\end{align*}
\]

\[
\begin{align*}
&\text{(number)} \rightarrow \text{numeric literal}
\end{align*}
\]

\[
\begin{align*}
&\text{(string)} \rightarrow \text{alphanumeric string}
\end{align*}
\]

A label is an alphanumeric string which is followed by a colon(:). The
label must be written on a separate line, not followed by an instruction or
directive or another label in the same line. When a label is encountered
in the program, the name of the label and its memory location is stored
in an Standard Template Library (STL) map.

Checks are made to ensure the parsed instructions have the format specified in the definition file. Operand values are checked against the permissible range. If all checks are OK, all values are written to a data structure. Another option would have been to write each parsed and assembled instruction directly to the output file. The advantage of writing the assembled results to an intermediate format first, is that it is easy to extend the functionality of NoGapAsm with new output file generators. The currently supported output formats are ASCII-binary, ASCII-binary with inlined assembly code\(^1\), and pure binary.

The only directive recognized by the generated parser is .org. When the program has .org followed by a number (in this case memory address), the instruction following the .org directive is loaded in the memory address specified with the .org. If there is empty space left in the memory before .org, it is filled with zeros.

### 14.2.3 Mnemonic Generation

As discussed in Section 14.2, the mnemonics for each instruction had to be generated. Mnemonic generation for an instruction is a two pronged problem. The first part is to find mnemonics for the different operations, e.g. add, sub, mul. The second part is to identify and generate a sensible operand syntax.

Mnemonics are automatically generated according to the following pattern:

\[
\text{opname}(\%CA_1), (\%CA_2), \ldots, (\%CA_N)
\]

where \text{opname} is the operation name and \%CA_X is a clause access name. The \text{opname} and clause access name(s) are collected from the \text{Masc} description of the data path. To understand the meaning of a clause access

---

\(^1\)Suitied for Verilog memory initialization.
name and what it has to do with an instruction, the reader is referred to [19].

Each assembly instruction can have zero or more operands. The following syntax is used for operands.

funame:N

where funame is the name of the FU connected to the respective immediate output port from the decoder and N is a number. For example, a resulting assembly instruction can look like this: add(%ADD)(%WRITE)

rf:10 rf:4 rf:3

14.2.4 NoGapAsm

Another part of the assembler generation process is to generate C++ code for NoGapAsm.

NoGapAsm consist of three static source files, one C++ file, one Flex file, and one Bison file. The Flex and Bison files capture the syntax and grammar, and since they are static, the basic syntax and grammar is always the same, independent of the generated processor. NoGapAsm will read a definition file, generated by AsmGen and setup the data structures that specifies the exact format of the instructions.

14.3 Testing NoGapAsm

To test the assembler generator and the resulting assembler a simple RISC processor, developed with NoGap, was used. Listing 14.1 shows the assembler instructions extracted from NoGapCD for this processor. The numbers in the square brackets are the respective immediate operands bit positions in the generated instruction format.

As discussed earlier, all instructions consists of a mnemonic followed by zero or more operands. The operand names are followed by the range describing where the operand is positioned in the instruction, This also
shows the permissible range of values that can be used for the operand. In total the instruction length for this processor is 41 bits.

In the assembly instruction;
r[4:0] occupies bit 4 to 0, rf[9:5] occupies bit 9 to 5, and rf[14:10] occupies bit 14 to 10. Since the instruction is 41 bits and the op-code is four bits, the op-code will be stored in bit 40 to 37. The rest of the instruction, i.e. bit 36 to 15 padded with zeros.

As can be seen, all rf immediates, in our test processor, are five bits wide, and hence the user can address the registers from 0 to 31. For example, add(%ADD)(%WRITE) rf:12 rf:31 rf:23 is a valid assembly instruction while add(%ADD)(%WRITE) rf:12 rf:40 rf:23 is invalid.

If the op-code, for the mnemonic add(%ADD)(%WRITE), is 0010, the assembly instruction add(%ADD)(%WRITE) rf:12 rf:31 rf:23 is converted to the following instruction:
0010 00000000000000000000 01100 11111 10111

Since some of the extracted mnemonic names are lengthy, a look up file with a list of existing mnemonic names is provided to the user. If the user wants to use a short name, s/he can edit the mnemonic file and add the new convenient name next to the mnemonic. For example, instead
of having to use the mnemonic `add(%ADD)(%WRITE)`, the user can define an alias for it, e.g. `add_easy`.

A small excerpt from an assembly program is shown in Listing 14.2. From the excerpt it can be seen that labels can be used as general numbers, e.g. the memory address of `label1` and `ca` are used in place of the memory address for one of the operands in line 5 and 9.

This program was successfully assembled and a correct binary file was generated using the command:

`./nogapasm AssemblyFile.S BinaryOutput.dat`

Listing 14.3 shows the output as ASCII-binary with comments.
Listing 14.3: ASCII-binary with comments

1 //nap
2 1001_0000000000000000000000000000000000000
3 //add(%ADD)(%WRITE) rf:1 rf:10 rf:23
4 0010_0000000000000000000000000001_01010_10111
5 //label1:
6 //add(%SUB)(%WRITE) rf:5 rf:0 rf:12
7 0011_0000000000000000000000_00101_00000_01100
8 //add2(%SUB)(%WRITE) rf:0 rf:10 rf:label1
9 0001_0000000000000000000000_00000_01010_00010
10 //nop
11 1001_0000000000000000000000000000000000000
12 //ca:
13 //io_out rf:30
14 0100_00000000000000000000000000000000_00000
15 1110
16 0101_00000000000000000000000000000000_00101

14.4 Remarks

By having a fixed grammar for all kinds of processor that can be generated by NoGap, it was possible to specify a fixed lexical analyzer, parser, and driver program. This also means that a user will be familiar with the assembly language syntax independent of the generated processor, which makes it easier to implement assembly programs for new processors. The use of a fixed parser with a definition file makes it easy to extend NoGapAsm with new functionality in the future, which is important since the NoGap project is still evolving.
Simulator Spawner

A technologically mature “posthuman” civilization would have enormous computing power. Based on this empirical fact, the simulation argument shows that at least one of the following propositions is true: (1) The fraction of human-level civilizations that reach a posthuman stage is very close to zero; (2) The fraction of posthuman civilizations that are interested in running ancestor-simulations is very close to zero; (3) The fraction of all people with our kind of experiences that are living in a simulation is very close to one.

...Unless we are now living in a simulation, our descendants will almost certainly never run an ancestor-simulation.

Nick Bostrom

An important aspect of processor generation tools are their ability to generate a supporting tool chain such as assemblers, simulators, and compilers. This chapter presents the techniques used to construct a cycle accurate simulator generator for NoGap.

The implementation of the simulator spawner was done by Sumathi Loganathan and Faisal Akhlaq, as part of their master thesis project.
Many of the design decisions were mine, but they did the actual implementation. More details about this work can therefore be found in [5].

### 15.1 Simulator Overview

The simulator generator described here is a cycle and bit accurate simulator. The simulator simulates one clock cycle at a time. The simulator generator is therefore centered around generating the cycle execute function, which is a C++ function that simulates the execution of one clock cycle. For this reason parallel architectures have to be translated into sequential code, we call this translation step sequentialization.

An overview of the simulator generation flow is shown in Figure 15.1. *NoGap* will generate *Mage* dependency graphs as part of the *NoGapCD*. The simulator generator uses the *Mage* dependency graph and other parts of the *NoGapCD* to sequentialize both the *Mage* FUs and the *Mase* graph. The simulator generator generates C++ code from the sequentialized units as well as a driver program using the generated C++ code. The generated C++ code is finally compiled to the actual simulator. The basic operation of the simulator is to execute a function modeling the behavior during a cycle, and then execute a function for finishing the cycle, i.e. one function for the duration of the clock period and one function for the rising edge of the clock, this is done as many times as needed.

### 15.2 Mage Dependency Graph Generation

The *Mage* dependency graph is a graph of the dependencies between variables and comb/cycle blocks in a *Mage* FU. A block consists of all statements within a cycle or comb statement. An example is shown in Listing 15.1, in which two blocks are defined, one comb and one cycle block. This Section will describe how the *Mage* dependency graph is generated and represented.
Chapter 15: Simulator Spawner

Figure 15.1: Simulator overview

Listing 15.1: Block Example

```sh
fu some_fu // Fu declaration and name
input a_i; // Port specifications
...
comb {
  o1 = reg;
  ...
}
cycle {
  reg = ~a_i;
  ...
}
}
```
An example of a Mage dependency graph for a simple register file is shown in Figure 15.2. BLOCK_0 is a combinational block and BLOCK_1 is a sequential block. Operand edges (Op) go from operands to the blocks where they are read. Assignment edges go from a block to the variables being assigned in that block, these assignment edges can be of two types, cycle assign (Cy=) or combinational assign (Co=). Cycle assign edges are used if a variable is assigned in a cycle block and thus forms a register. Combinational assign edges are used if a variable is assigned in a comb block and thus forms combinational logic. The algorithm used to construct a Mage dependency graph is described in Algorithm 15.1.

**Algorithm 15.1: Mage dependency graph generation**

```plaintext
input: Mage FU FU
insert_block_vertices(FU);
insert_symbol_vertices(FU);
foreach block ∈ FU do
    S_in = find_input_set(block);
    S_out = find_output_set(block);
    foreach vertex ∈ S_in do
        add_edge(source:vertex, target:block);
    end foreach
    foreach vertex ∈ S_out do
        add_edge(source:block, target:vertex);
    end foreach
end foreach
```
15.3 \textit{Mase} Source-Sink-Pass Partitioning

To be able to build a cycle based software simulator, there can not be any combinational loops in the system. Combinational loops can arise internally in \textit{Mage} FUs or be introduced when connecting \textit{Mage} FUs in the data path of a \textit{Mase} FU. The issues of loops in the \textit{Mase} graph is discussed in Section 7.6.

Combinational loops in a \textit{Mage} FU can be checked for, by removing the cycle assign edges \((Cy=)\) in the \textit{Mage} dependency graph and then run a standard loop detection algorithm. If a loop is found it will be considered as an error and the simulator generation process will be aborted.

FUs in the \textit{Mase} has to be handled in some way, they could be considered to be purely combinational and thus any OI-\textit{Intra} not broken by a pipeline register would be considered to be an illegal design. This view is however too conservative. Instead the \textit{Mage} dependency graph is used to split each central FU \(V_{\text{Mase}}\) into three nodes; sink, source, and pass, this can be seen in Figure 15.5. Input ports which have no combinational effect on any output port are connected to a sink \(V_{\text{Mase}}\), output ports that have no combinational dependency on any input port are connected to a source \(V_{\text{Mase}}\), and finally input/output ports which

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{simulator_spawner.png}
\caption{\textit{Mage} dependency graph for a register file}
\end{figure}
have a combinational dependency are connected to the pass $\mathcal{V}_{\mathcal{M}a5r}$. We call this process SSP partitioning of the $\mathcal{M}a5r$ graph. An example of how this works can be seen in Figure 15.3 and 15.4. The pipeline in Figure 15.3 shows a pipeline that needs to be serialized. Figure 15.4 shows how this pipeline would look after an SSP partitioning. First of all, the FFs have been split into input-FFs and output-FFs. But also the individual FUs have been split into one source partition (light gray $S$ circle), one sink partition (the other $S$ circle), and one pass partition. The SSP partitioning will break false loops as can be seen in Figure 15.5.
Chapter 15: Simulator Spawner

Figure 15.3: Before SSP

Figure 15.4: After SSP

Figure 15.5: Loop removal thanks to SSP partitioning
Algorithm 15.2: SSP Partitioning

```
input: G: Magic Dep. Graph
output: $S_{sink}$
output: $S_{source}$
output: $S_{pass}$

$S_{in} = \text{find\_inport\_set}(G)$;
$S_{out} = \text{find\_outport\_set}(G)$;
$G = \text{remove\_cy\_edge}(G)$;

foreach vertex $\in S_{in}$ do
    $S_{visited} = \text{DFS}(\text{start:vertex},\text{graph:G})$;
    $S_{pass} = (S_{visited} \cap S_{out}) \cup S_{pass}$;
end foreach

foreach vertex $\in S_{in}$ do
    $S_{visited} = \text{DFS}(\text{start:vertex},\text{graph:G})$;
    if $S_{visited} \cap S_{out} = \emptyset$ then
        $S_{sink} = \text{vertex} \cup S_{sink}$;
    end if
end foreach

$S_{source} = S_{out} - S_{pass}$;
```

The SSP partition of an FU is calculated accordingly to Algorithm 15.2, where \textit{find\_inport\_set}/\textit{find\_outport\_set} builds a set with all input/output ports, i.e. vertices in the dependency graph with zero in edges/out edges, \textit{DFS} is a depth first search starting at node \textit{start} and returning all visited vertices.

Then, in the Magic graph, for each FU, connect all FU input ports either to a \textit{sink} or \textit{pass} $V_{\text{Magic}}$, according to the \textit{sink set} and \textit{pass set}, all FU output ports are connected to either a \textit{pass} or a \textit{source} $V_{\text{Magic}}$, according to the \textit{pass set} and \textit{source set}. 
15.4 Mage Sequentialization

Each Mage FU will have a class generated for it, each Mage FU instance will then be an object of that class. A Mage module class consists of three member collections; input port values, output ports values, and register values. Each Mage class also has three execution methods `execute_sink`, `execute_source`, and `execute_pass`, used to execute the functionality for the corresponding SSP partitioned FU. Since port sizes can vary from each instance of the Mage module, the exact port and register sizes are set when the corresponding object is instantiated. Computations are done with a custom written bit exact type, this bit exact type is then wrapped in a `boost::optional` container so that uninitialized values can be tracked.

The problem we had to solve was how to translate a Mage FUs, which describe parallel execution, into sequential C++ code. Each block in a Mage FU is a sequential execution description. Assignments in comb blocks are blocking and assignments in cycle blocks are non blocking. All in accordance with best practices in Verilog. However the execution between blocks are parallel and needs to be sequentialized. This sequentialization is done by performing a topological sort, on relevant parts of the Mage dependency graph.

Algorithm 15.3 describes how a Mage FU is sequentialized. Since the SSP partitioning requires one C++ method to be generated for each partition, three methods will be generated; `execute_sink`, `execute_source`, and `execute_pass`.

15.5 Mase Sequentialization

The Mase graph must be sequentialized so that sequential code can be generated that models the Mase graph. The first step is to create an SSP partitioned version of the Mase graph, referred to as an SSP-graph.
Algorithm 15.3: Mage sequentialization

input: $G$: Mage dependency graph
input: ssp_partitions: SSP partitions for this Mage
output: One C++ function per SSP partition

foreach partition $\in$ ssp_partitions do
  $G' = \text{keep}_{-\text{dependent}}_{\text{graph}}(G)$;
  $G'_{\text{-cy}} = \text{remove}_{-\text{cy}}_{\text{edge}}(G')$;
  ordered_vector = topological_sort($G'_{\text{-cy}}$);
  foreach vertex $\in$ ordered_vector do
    write_cpp_code(vertex, partition);
  end foreach
end foreach

FUs are partitioned as discussed in Section 15.4 and pipeline register are split into a source and a sink $V_{\text{masr}}$. Some other special $V_{\text{masr}}$s also has to be handled, they are: synthesized multiplexer, class selector $V_{\text{masr}}$, inline-expressions, and decoders. They are all replaced with an FU with just a pass $V_{\text{masr}}$. After the SSP-graph has been generated it is topologically sorted to find an execution order for all the $V_{\text{masr}}$s in the SSP-graph. Finally the graph is traversed in the sorted order and for each vertex visited, the corresponding C++ code is written to the cycle execute function.

All pipeline registers are collected in one special data structure, consisting of two vectors of all pipeline registers, a read vector and a write vector. During the execution of a single cycle, data is read from the read vector and written to the write vector. At the end of the cycle the contents of these vectors are swapped. This can be done by just swapping two pointers, thus saving execution time.
Chapter 15: Simulator Spawner

15.6 Results

To test the generated simulator we used a simple RISC processor which can perform load, addition and subtraction developed with NoGap. An assembly program was written which loads values to operand registers, add and subtract loaded values, and store the results of addition and subtraction in different registers. Binary code was generated for the assembly program through the NoGap assembler, described in Chapter 14. With the binary code as input, the NoGap cycle-accurate simulator was ran for the number of clock cycles specified by the designer. Values in all registers can be checked at the end of each execution cycle. Listing 15.2 shows the register values after 14 cycle executions. An Empty value means that it is an uninitialized value.

In Listing 15.2 register 0, 1 and 2 were loaded with values 8, 9 and 7. The result of addition of register 1 and 2 was saved to register 3 which is 10 in hexadecimal representation. Register 5 and 4 were subtracted and result was stored into register 7. The processor consists of 16 registers, and those registers which are not used for any processing are displayed with blank values.
Part V

Experimental Results
Speed is a great asset; but it’s greater when it’s combined with quickness - and there’s a big difference.

Ty Cobb

### 16.1 Introduction

In order to test how NoGap compares to less abstracted languages, such as Verilog, we have used NoGap to reimplement the floating point adder/subtractor (hereafter referred to as only “adder”), first described in [17] and further elaborated in [18]. The architecture of the adder is presented in Figure 16.1. For anymore details the reader is referred to [18].

If the floating point adder could be implemented in pure NoGap code, it would prove that NoGap can be used to implement complex data paths. We would also have an highly optimized Verilog version to compare against. Although both a floating point multiplier and adder is
presented in [18], the adder was chosen for this test since its architecture is more complex than that of the multiplier.

The floating point adder in question is almost IEEE 754 compliant [4]. The binary format is the same as for IEEE 754, i.e. a 23 bit mantissa with an implicit one, an 8 bit exponent, and one sign bit. Our adder does not handle denormalized numbers, nor does it honor NaN or Inf. The rounding is done using the round to nearest even method. The rounding method can not be changed as is mandated by IEEE 754 to be fully IEEE 754 compliant. Denormalized numbers are excluded due to the large overhead of correctly dealing with these numbers. Denormalized numbers are often excluded from high performance systems, e.g. the Cell processor does not use denormalized numbers for the single precision format in its synergistic processing units [31].

16.2 Implementation

Implementing the floating point adder in NoGap required two steps. The first step required us to rewrite all modules except the top level pipeline as Mage FUs. This was a fairly straightforward process since Mage FU code bears a lot of resemblance with Verilog code. The second step was to model the pipeline using the previously constructed Mage FUs. Part of the Mage description is shown in Listing 16.1. This step was actually easier than writing the code directly in Verilog, thanks to NoGap’s ability to synchronize pipelined signals.

A NoGapCL excerpt from the actual implementation is shown in Listing 16.1.

The generated Verilog code was then synthesized and targeted to different FPGAs to measure and compare the performance of the generated Verilog code.
Figure 16.1: Adder architecture overview
Chapter 16: Floating Point Core

Listing 16.1: Example

```c
fu fp_add {
  output [22:0] man_o;
  //...more code...
  stage ff() { //One clock cycle phase separation.
    cycle { ffo = ffl; }
  }
  stage wire() { //Zero clock cycle phase separation.
    comb { ffo = ffl; }
  }
  pipeline add_pipe{
    IN_REG -> wire -> CMP -> ff -> SEL -> ff
    -> ALIGN -> ff -> ADD -> ff -> FL1 -> ff
    -> NORM_1 -> ff -> NORM_2 -> ff -> ROUND
    -> POST_PROCESS -> wire -> OUT_REG;
  }
  //...more code...
  fu::sticky_bit_prep(%OP) sticky_bit_a;
  //...more code...
  operation(add_pipe) add_sub() {
    @ CMP;
    //Sticky bit generation FU placed at this phase
    sticky_bit_a;
    sticky_bit_a.man_i = a_man_i;
    //...more code...
    @SEL;
    sticky_vec_sel; //Sticky bit selection at this phase
    //Reading data from output port of sticky_bit_a
    sticky_vec_sel.dat_a_i = sticky_bit_a.sticky_prep_o;
    //...more code...
    @OUT_REG;
    //Result written to the output port
    man_o = man_out_mux.dat_o;
    //...more code...
  }
}```
16.3 Implementation Details

Porting the existing adder to NoGap\(^{CL}\) took three work days with the added benefit of getting a clearer picture of the pipelined data path.

Xilinx' place and route tool was used to determine the maximum clock frequency by changing the timing constraints until timing closure could not be achieved. The clock frequencies reported by us are the maximum frequencies for which timing closure occurred, rounded down to the nearest integer. Also the clock frequencies reported, assumes a clock with no jitter.

The final adder architecture has a latency of 8 clock cycles. The optimized adder presented in [18] is listed as DA and the adder implemented using NoGap is listed as NoGap. Table 16.1 lists various performance metrics over different devices and speed grades.

<table>
<thead>
<tr>
<th>Device</th>
<th>DA</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>XC4VSX-10</td>
<td>290</td>
<td>217</td>
</tr>
<tr>
<td>XC4VSX-11</td>
<td>302</td>
<td>256</td>
</tr>
<tr>
<td>XC4VSX-12</td>
<td>377</td>
<td>276/326(^a)</td>
</tr>
<tr>
<td>XC5VLX-1</td>
<td>317</td>
<td>285</td>
</tr>
<tr>
<td>XC5VLX-2</td>
<td>366</td>
<td>321</td>
</tr>
<tr>
<td>XC5VLX-3</td>
<td>419</td>
<td>379</td>
</tr>
</tbody>
</table>

\(^a\)With hand optimized adder sub-module.

To get an idea of where resources are consumed in the two implementations, Table 16.2 lists the resource utilization, for a Virtex-4, of the steps in Figure 16.1. To avoid the extra delays associated with the FPGA I/O pins, two extra pipeline stages before and one stage after were inserted into the top module. These extra FFs are not included in the
resource utilization metrics. As can be seen the Look-Up Table (LUT) utilization is about the same except for the mantissa adder where the heavily optimized adder naturally has a lower resource utilization. Also for the NoGap adder some FFs have been moved out of the individual modules into the top level pipeline, this explains the larger discrepancies seen between the FF utilization.

Table 16.3, 16.4, 16.5, and 16.6, compares our results (NoGap) against some other publications.

Please note that the figures are obtained from units with slightly different features in terms of IEEE 754 compliance. The numbers of LUTs in the case of the Nallatech unit is an estimation since Nallatech only publish data for how many slices their design occupies. The number of LUT is in this case estimated to be twice as many as the slices since there is two LUT per slice in a Virtex-II. Although the comparisons here are not completely fair they still give a good picture of how the performance of our floating point units compare to other FPGA implementations.

| Table 16.2: Adder resource utilization in Virtex 4 |
|----------------|----------------|----------------|
|                | DA             | NoGap          |
|                | LUT  FF        | LUT  FF        |
| Compare/Select | 111 22         | 95 21          |
| Align          | 134 66         | 118 0          |
| Add            | 36 29          | 111 0          |
| Normalization  | 436 191        | 418 223        |
| Round          | 8 0            | 7 0            |
| Other          | 121 121        | 160 289        |
| Total          | 846 429        | 909 533        |
### Table 16.3: Comparison with USC adder [12] in Virtex-II (XC2VP-7).

<table>
<thead>
<tr>
<th></th>
<th>USC</th>
<th>DA</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>19</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>LUTs</td>
<td>548</td>
<td>760</td>
<td>882</td>
</tr>
<tr>
<td>FFs</td>
<td>801</td>
<td>516</td>
<td>532</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>250</td>
<td>278</td>
<td>233</td>
</tr>
</tbody>
</table>

### Table 16.4: Comparison with Nallatech adder [30] in Virtex-II (XC2VP-6).

<table>
<thead>
<tr>
<th></th>
<th>Nallatech</th>
<th>DA</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>14</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>LUTs</td>
<td>&lt; 580&lt;sup&gt;a&lt;/sup&gt;</td>
<td>758</td>
<td>882</td>
</tr>
<tr>
<td>FFs</td>
<td>?</td>
<td>517</td>
<td>532</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>184</td>
<td>242</td>
<td>211</td>
</tr>
</tbody>
</table>

<sup>a</sup>Value estimated from number of slices

### Table 16.5: Comparison with Xilinx adder [47] in Virtex-4 (XC4VSX-10).

<table>
<thead>
<tr>
<th></th>
<th>Xilinx</th>
<th>DA</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Pipeline depth</td>
<td>13</td>
<td>8</td>
<td>8</td>
</tr>
<tr>
<td>LUTs</td>
<td>578</td>
<td>846</td>
<td>876</td>
</tr>
<tr>
<td>FFs</td>
<td>594</td>
<td>429</td>
<td>533</td>
</tr>
<tr>
<td>Clock frequency (MHz)</td>
<td>368</td>
<td>290</td>
<td>217</td>
</tr>
</tbody>
</table>
16.4 Discussion

As can be seen from the result the hardware generated directly from NoGap code could not reach the same clock frequencies as the hand optimized code. The critical path for the NoGap architecture was through the mantissa adder/subtractor, this unit was optimized using directly instantiated LUTs in the original implementation and it is therefore not surprising that the NoGap implementation could not reach the same clock frequencies. As a final experiment the Verilog code of the adder generated from NoGapCL code was replaced with the Verilog code for the hand optimized adder and resynthesized for a Virtex-4SX35 (speed grade -12) timing closure was then met at 338 MHz, compared to 377 MHz for the hand optimized implementation. On the other hand by using NoGap the design of the top level pipeline was greatly simplified for thanks to the features of NoGapCL. Many times time to market is as important as performance. Also the ease with which critical path components can be substituted with hand optimized code, makes it possible to employ an iterative process, where NoGap is used for the initial implementation and then focus can be directed to improving the critical path.
It is not easy to be a pioneer - but oh, it is fascinating! I would not trade one moment, even the worst moment, for all the riches in the world.

Elizabeth Blackwell

17.1 Introduction

In order to test and verify NoGap, both in terms of usability for a designer and in terms of performance for the generated hardware, we started out with a simpler processor called PIONEER.

Ching-han Wang, a master thesis student, did the actual implementation of PIONEER, using NoGap. The results presented in this chapter are based on his work. Therefore, more details can be found in his master thesis [45].
17.2 PIONEER Overview

17.2.1 Architecture

The PIONEER processor is a single issue RISC processor with DSP extensions, its architecture is based on the Harvard architecture. PIONEER is divided into a data path and sequence path. The data path is comprised of a 16×16 bit register file, a 16 bit ALU, a 16×16 bit multiplier, 4×36 bit accumulator register, a data memory, a condition checker and status registers including the stack pointer, ALU flags, and Multiply And Accumulate (MAC) flags. The sequence path consists of a PC-FSM and an instruction memory. An overview of the architecture is presented in Figure 17.1.

The sequence and data path is communicating through control bridges, resulting in the structure shown in Figure 17.2. The data path has two pipeline configurations and the sequence path has one.

17.2.2 Instruction Set

PIONEER’s instruction set consists of four types of instructions; short arithmetic and logic, long arithmetic, move-load-store, and flow control.

The short arithmetic and logic instructions performs 16 bit arithmetic and logic operations. All operands are acquired from the register file. The arithmetic and logic instructions includes addition, subtraction, increment, decrement, and, or, not, and xor.

The long arithmetic instruction performs 32 bit arithmetic operations. The long arithmetic operations includes; mul (multiply), mac (multiply and accumulate, and mdm (multiply and diminish). All long arithmetic instructions are carried out with four guard bits, resulting in 36 bits internal resolution.

Move-load-store instructions performs 16 bit data transfers between general registers, status registers, accumulators (guard, higher and lower
Figure 17.1: PIONEER
parts) and the data memory. The general registers can also load data from immediate values carried by the load instructions.

Flow control instructions deals with program flow, and includes conditional jump, call, and return. PIONEER uses a software stack stored in the general data memory.

### 17.3 Implementation with Joust approach

PIONEER is constructed with one data path and one sequence path. According to the Joust architecture, each path contains a transformation path and a control bridge. The data path consists of a control bridge with an instruction decoder and a Data Path Transformation Path (DPTP) which has one short pipeline, for normal instruction and one long pipeline, for MAC instruction. The sequence path consists of a control bridge with an empty decoder and an Sequence Path Transformation Path (SPTP). This architecture is roughly depicted in Figure 17.2. The gray control bridge of the sequence path is minimal with an empty decoder and pipeliner. All of the FUs used in PIONEER described are instantiated in either DPTP or SPTP.

Listing 17.1 shows a NoGap CL excerpt from PIONEER. The division, seen in the code, into a data path and sequence path connected in a top level unit, is a good illustration of the Joust approach.
### Listing 17.1: PIONEER in NeSapCL description

```plaintext
1 fu top_level {
2 .................;
3 component data_path dp {
4 op -> op_i;
5 take_jump <- take_jump_o;
6 pc -> pc_i;
7 .................; }
8 component seq_path sp {
9 op <- op_o;
10 take_jump -> take_jump_i;
11 pc <- pc_o;
12 .................; }
14 }
15
16 fu seq_path {
17 input take_jump_i;
18 input [7:0] target_address_i;
19 ...
20 fu::program_counter(%INC_PC) pc;
```
21    stage ff() { cycle { ffo=ffi; } }
22    phase PC,IF,DEC;
23
24    pipeline sequence_pipe
25    { PC -> ff -> IF -> ff -> DEC; }
26
27    operation(sequence_pipe) flow() {
28
29        @ PC;
30        pc;
31        pc.take_jump_i=take_jump_i;
32
33        @ IF;
34        imem;
35        imem.addr_i=pc.current_pc_o;
36
37        @ DEC;
38        op_o = imem.dat_o;
39    }
40
41    fu data_path {
42
43        input [31:0] op_i;
44        ...
45        fu::alu(%NOP) alu;
46        fu::decoder_spec<instr_i>() dec_unit;
47
48    stage ff() { cycle { ffo=ffi; } }
49    phase DE, OF, EX, WB, EX2, WBL;
50
51    pipeline normal_pipe
52    { DE -> ff -> OF -> ff -> EX -> fff -> WB; }
53
54    pipeline long_pipe
55    { DE -> ff -> OF -> ff -> EX -> ff -> EX2 -> ff -> WB; }
56
57    operation(npipe) nop(dec.nop) {
58        @DE;
59        dec_unit;
60        dec_unit.instr_i = op_i;
61        jump_o = dec_unit.jump_o;
62
63    ...
64
65    operation(long_pipe) mac_inst(dec_unit.mac) {
66        @DE;
67        dec_unit;
68        ...
69        @OP;
17.4 Verification

The PIONEER processor was verified in hardware using the DAFK system [39]. The DAFK system is a soft computer system utilizing the OR1200 Central Processing Unit (CPU) from Open cores [32]. The testbed architecture is shown in Figure 17.3. An assembler for PIONEER...
was generated by NoGap’s assembler generator, AsmGen (Chapter 14). A number of assembly programs were written to test the processor. For example some Finite Impulse Response (FIR) filters and an $8 \times 8$ Discrete Cosine Transform (DCT) were used as test cases. The generated assembler was used to assemble these programs into binaries for PIONEER on a standard PC. These binaries were downloaded to the DAFK system test bed, running $\mu$Clinux, through an ethernet connection. A program executing on the OR1200 was then used to transfer the data, through the wishbone bus, into the PIONEER’s instruction memory, PIONEER was then started. When PIONEER had completed its program, the execution results, now in PIONEER’s data memory, were red back to the OR1200 processor and verified against precomputed values, all results were found to be correct.

17.5 Results

The SystemVerilog code generated by NoGap was synthesized to a Virtex-4 LX80 speed-grade 12 FPGA. Precision was used for synthesis and Xilinx ISE 10.1 was used for mapping and place & route. The generated code contains no FPGA specific optimizations. The data and instruction memory are not part of the processor design, they are instantiated externally, using the block Random Access Memories (RAMs) present in the Virtex-4. The memories are integrated into the data path of PIONEER using external interface FUs.

Table 17.1 displays the hardware usage by part in the synthesized result for FPGA. Two numbers are given for each column, A/B, A is the number of elements that belong to that specific hierarchical module, B is the total number of elements from that hierarchical module and any lower level hierarchical modules below.

The generated processor met timing closure at 204MHz in the Virtex-4 flow as compared to xi2 [9] at 334MHz and Microblaze at 200MHz.
Table 17.1: FPGA utilization by part

<table>
<thead>
<tr>
<th>Part</th>
<th>Slices</th>
<th>LUTs</th>
<th>DSP48</th>
<th>RAMB16</th>
</tr>
</thead>
<tbody>
<tr>
<td>+ data path</td>
<td>475/832</td>
<td>248/637</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>++alu</td>
<td>41/41</td>
<td>66/66</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++decoder</td>
<td>96/96</td>
<td>123/123</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++macreg</td>
<td>169/169</td>
<td>108/108</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++register</td>
<td>33/33</td>
<td>65/65</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++sp</td>
<td>18/18</td>
<td>27/27</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ sequence path</td>
<td>0/4</td>
<td>0/0</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++pc</td>
<td>4/4</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>+ Memory</td>
<td>87/87</td>
<td>16/16</td>
<td>2/2</td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>923</td>
<td>708</td>
<td>1</td>
<td>2</td>
</tr>
</tbody>
</table>

When comparing these values it is important to remember that both xil2 and Microblaze are heavily optimized for FPGA usage, whereas PIONEER is completely generated by NoGap. Although this comparison is not entirely fair since all processors are different, it gives a ball park figure about how NoGap generated processors can compare to heavily optimized implementations. The most important conclusion about the NoGap generated processor comes when looking at the critical path. The critical path of PIONEER in both the FPGA and ASIC flow are through the carry chain of the 36 bit adder/subtracter and flags computation, meaning that the NoGap synthesized control logic and other related hardware, does not incur any performance penalty.

17.5.1 ASIC Flow

The SystemVerilog code generated by NoGap was also targeted to an ASIC flow. For this, a tool chain supplied by Synopsys was used. The
area usage and power consumption are 24815 $\mu m^2$ and 1.607 mW (estimated) respectively.

Cadence tools were used to do a layout for chip design in 65 nm CMOS technology. The RoChap generated System Verilog code could be synthesized without any errors.

The generated processor met timing closure at 300 MHz with the ASIC flow just described.
18

SENIOR

Life must be understood backwards; but... it must be lived forward.

Soren Kierkegaard

18.1 Introduction

After having seen that NoGap could be used to implement PIONEER, we wanted to test NoGap further by implementing a more complex processor. For this we choose to reimplement SENIOR, a processor that has been under development in our department for the last 4.5 years.

Ching-han Wang, a master thesis student, did the actual implementation of SENIOR, using NoGap. The results presented in this chapter are based on his work. Therefore, more details can be found in his master thesis [45].
18.2 Architecture

The SENIOR processor is a single issue RISC processor with DSP extensions, based on the Harvard architecture. SENIOR is divided into a data path and sequence path. The data path consists of a single precision 16 bit ALU, a double precision MAC unit, two data memories with dedicated address generated units, a condition checker, flag computation units, a loop controller, and memory units including a $32 \times 16$ bit general purpose register files unit, 18 single precision special purpose registers, and four double precision MAC register with eight guard bits. The sequence path contains a program counter, a PC-FSM, and an instruction memory. More details about SENIOR can be found in the SENIOR manual [42].

The SENIOR processor has two kinds of pipelines, normal pipeline and longer pipeline, which are listed in Table 18.1 (with memory access, and MAC instructions in their own columns but in the same pipeline as the normal instruction). Description for each stages is shown in Table 18.2. The different pipeline architectures are also illustrated in Figure 18.1 and Figure 18.2.

The sequence and data path are communicating through control bridges, resulting in the structure shown in Figure 17.2. The data path has two pipeline configurations and the sequence path has one.

18.3 Instruction Set

SENIOR has 4 type of instructions. A detailed description of all of SENIOR’s instructions can be found in the SENIOR manual [42]. This Section will give a short overview of the instruction set. Section 18.3.1 outlines the move-load-store instructions, Section 18.3.2 outlines the short ALU instructions, Section 18.3.3 outlines the long arithmetic instructions, and Section 18.3.4 outlines the flow control instructions.
### Table 18.1: Pipeline Specification

<table>
<thead>
<tr>
<th>Stage</th>
<th>Normal</th>
<th>MAC</th>
<th>Memory</th>
<th>Convolution</th>
</tr>
</thead>
<tbody>
<tr>
<td>P0</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
<td>PC</td>
</tr>
<tr>
<td>P1</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
<td>IF</td>
</tr>
<tr>
<td>P2</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
<td>DE</td>
</tr>
<tr>
<td>P3</td>
<td>OF</td>
<td>OFAG</td>
<td>OFAG</td>
<td>OFAG</td>
</tr>
<tr>
<td>P4</td>
<td>EX1</td>
<td>EX1</td>
<td>MEM</td>
<td>MEM</td>
</tr>
<tr>
<td>P5</td>
<td>W1</td>
<td>EX2</td>
<td>W1</td>
<td>W2</td>
</tr>
<tr>
<td>P6</td>
<td>WB</td>
<td>WB</td>
<td>WB</td>
<td>MUL</td>
</tr>
<tr>
<td>P7</td>
<td></td>
<td></td>
<td></td>
<td>ACC</td>
</tr>
</tbody>
</table>

### Table 18.2: Explanation of pipeline stages for SENIOR

<table>
<thead>
<tr>
<th>Stage</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PC</td>
<td>Program counter</td>
</tr>
<tr>
<td>ID</td>
<td>Instruction decode</td>
</tr>
<tr>
<td>OF</td>
<td>Operand fetch</td>
</tr>
<tr>
<td>AG</td>
<td>Address computation</td>
</tr>
<tr>
<td>MEM</td>
<td>Memory access</td>
</tr>
<tr>
<td>EX1</td>
<td>Execution</td>
</tr>
<tr>
<td>EX2</td>
<td>Execution for MAC</td>
</tr>
<tr>
<td>W1</td>
<td>Wait for other instruction to solve structural hazards</td>
</tr>
<tr>
<td>W2</td>
<td>Send data from MEM to MUL</td>
</tr>
<tr>
<td>WB</td>
<td>Write back to register file</td>
</tr>
<tr>
<td>MUL</td>
<td>Multiplication</td>
</tr>
<tr>
<td>ACC</td>
<td>Accumulation</td>
</tr>
</tbody>
</table>
Figure 18.1: Pipeline architecture for normal instruction in SENIOR
Figure 18.2: Pipeline architecture for convolution instruction in SENIOR
18.3.1 Move-Load-Store Instructions

Move-load-store instructions handle 16-bit data transfer operations, between general purpose registers, special registers, accumulators, data memories and I/O ports. The move and load instructions can work with most of the registers, special registers and accumulators for data access with plenty of options, as the load-store instructions can use various addressing modes for data memory access. Table 18.3 lists the move-load-store instructions.

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description/specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>move</td>
<td>copy data to/from registers, special registers, and accumulators</td>
</tr>
<tr>
<td>set</td>
<td>set constant to register and special registers</td>
</tr>
<tr>
<td>ld0</td>
<td>load register from data memory 0</td>
</tr>
<tr>
<td>ld1</td>
<td>load register from data memory 1</td>
</tr>
<tr>
<td>st0</td>
<td>store register to data memory 0</td>
</tr>
<tr>
<td>st1</td>
<td>store register to data memory 0</td>
</tr>
<tr>
<td>dblld</td>
<td>double load</td>
</tr>
<tr>
<td>dblst</td>
<td>double store</td>
</tr>
<tr>
<td>in</td>
<td>read IO port to register</td>
</tr>
<tr>
<td>out</td>
<td>write register to IO port</td>
</tr>
</tbody>
</table>

18.3.2 Short ALU Instructions

The short ALU instructions handle 16-bit arithmetical, logical, and shift instructions, logical. All instructions can make use of conditional execution (depending on the status of ALU/MAC flags) if no constant operands are used. Because of code size limitations in the original SE-
NIOR, constant operands are restricted to a size of 12 bits, but they are always sign extended to 16 bits before use. Only the `cmp` instruction can carry a 16 bit constants. Table 18.4 lists the short ALU instructions.

### 18.3.3 Long Arithmetic Instructions

The long arithmetic instructions performs 32-bit arithmetic operations using the MAC unit. Operations are carried out with 8 guard bits on all operands, giving 40 bits internal resolution in the MAC unit. Table 18.5 lists the long ALU instructions.

### 18.3.4 Flow Control Instructions

The flow control instructions determine the program flow. Instruction `jump` can make use of conditional execution. Table 18.6 lists the flow control instructions.

### 18.4 NoGap for SENIOR

#### 18.4.1 Mage in SENIOR

Mage FUs are used to describe leaf module with functionality, like most of the components in Figure 18.1. An example can be seen in Listing 18.1, which is the Address Generation Unit (AGU) in SENIOR. The AGU uses DCS to implement 10 different addressing modes for data memories accessing.

#### 18.4.2 Mase in SENIOR

For every operation, the Mase FU defines the interconnection and clause selection for every Mage in the data path or control path. Double load, `(dblId)`, which is a special instruction for accelerating some application such as DCT or Fast Fourier Transform (FFT), is shown in Listing 18.2, which shows the data path of SENIOR.
<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description/specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>add</td>
<td>addition of registers and constants</td>
</tr>
<tr>
<td>addn</td>
<td>addition of registers and constants with no flag change</td>
</tr>
<tr>
<td>addc</td>
<td>addition of registers and constants with carry</td>
</tr>
<tr>
<td>adds</td>
<td>addition of registers and constants with saturation</td>
</tr>
<tr>
<td>sub</td>
<td>subtraction of registers and constants</td>
</tr>
<tr>
<td>subn</td>
<td>subtraction of registers and constants with no flag change</td>
</tr>
<tr>
<td>subc</td>
<td>subtraction of registers and constants with carry</td>
</tr>
<tr>
<td>subs</td>
<td>subtraction of registers and constants with saturation</td>
</tr>
<tr>
<td>cmp</td>
<td>compare registers and constants</td>
</tr>
<tr>
<td>max</td>
<td>return maximum of registers and constants</td>
</tr>
<tr>
<td>min</td>
<td>return minimum of registers and constants</td>
</tr>
<tr>
<td>abs</td>
<td>return absolute of a register</td>
</tr>
<tr>
<td>and</td>
<td>logic and between registers and constants</td>
</tr>
<tr>
<td>andn</td>
<td>logic and between registers and constants with no flag change</td>
</tr>
<tr>
<td>or</td>
<td>logic or between registers and constants</td>
</tr>
<tr>
<td>orn</td>
<td>logic or between registers and constants with no flag change</td>
</tr>
<tr>
<td>xor</td>
<td>logic xor between registers and constants</td>
</tr>
<tr>
<td>xorn</td>
<td>logic xor between registers and constants with no flag change</td>
</tr>
<tr>
<td>asr</td>
<td>arithmetic shift right</td>
</tr>
<tr>
<td>asl</td>
<td>arithmetic shift left</td>
</tr>
<tr>
<td>lsr</td>
<td>logic shift right</td>
</tr>
<tr>
<td>lsl</td>
<td>logic shift left</td>
</tr>
<tr>
<td>ror</td>
<td>rotate right</td>
</tr>
<tr>
<td>rol</td>
<td>rotate left</td>
</tr>
<tr>
<td>rcr</td>
<td>rotate right through carry</td>
</tr>
<tr>
<td>rcl</td>
<td>rotate left through carry</td>
</tr>
</tbody>
</table>
## Table 18.5: Long ALU instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description/specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>addl</td>
<td>long addition between accumulators and registers</td>
</tr>
<tr>
<td>subl</td>
<td>long subtraction between accumulators and registers</td>
</tr>
<tr>
<td>sublst0</td>
<td>long subtraction and store register to data memory 0</td>
</tr>
<tr>
<td>sublst1</td>
<td>long subtraction and store register to data memory 1</td>
</tr>
<tr>
<td>cmpl</td>
<td>long compare between accumulators and registers</td>
</tr>
<tr>
<td>abs1</td>
<td>long absolute of accumulator or register</td>
</tr>
<tr>
<td>negl</td>
<td>long negation of accumulator or register</td>
</tr>
<tr>
<td>movel</td>
<td>long move of register to accumulator</td>
</tr>
<tr>
<td>clr</td>
<td>clear of accumulator</td>
</tr>
<tr>
<td>postop</td>
<td>post accumulator operation</td>
</tr>
<tr>
<td>mul</td>
<td>multiply</td>
</tr>
<tr>
<td>mulld0</td>
<td>multiply and load register from data memory 0</td>
</tr>
<tr>
<td>mulld1</td>
<td>multiply and load register from data memory 1</td>
</tr>
<tr>
<td>muldbld</td>
<td>multiply and double load</td>
</tr>
<tr>
<td>mac</td>
<td>multiply and accumulate</td>
</tr>
<tr>
<td>macld0</td>
<td>multiply, accumulate and load register from data memory 0</td>
</tr>
<tr>
<td>macld1</td>
<td>multiply, accumulate and load register from data memory 1</td>
</tr>
<tr>
<td>mdm</td>
<td>multiply and diminish</td>
</tr>
<tr>
<td>conv</td>
<td>convolution</td>
</tr>
</tbody>
</table>
Listing 18.1: Magc in SENIOR

```vhdl
fu agu_0
2{
3    input [2:0] op_i;
4    ...; // other port declarations
5    comb
6    {
7        switch(op_i)
8        {
9            0:%INC{ //post-incremental
10               addr_o=areg_i;
11               areg_o=areg_i+step_i;
12           }
13            1:%DEC { //pre-decremental
14               addr_o=areg_i-step_i;
15               areg_o=areg_i-step_i;
16           }
17            2:%OFF { //offset
18               addr_o=areg_i+imm12_i;
19               areg_o=areg_i; //
20           }
21            3:%MINC { //modulo post-inc
22               addr_o=areg_i;
23               areg_o=modulo;
24           }
25            4:%BRV { //bit-reversal
26               addr_o=bitrev_i;
27               areg_o=areg_i+step_i;
28           ...; //other addressing modes
29           }
30        }
31    }
32}
```
Chapter 18: SENIOR

Table 18.6: Flow instructions

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Description/specification</th>
</tr>
</thead>
<tbody>
<tr>
<td>repeat</td>
<td>repeat instructions in sequence a certain number of times</td>
</tr>
<tr>
<td>jump</td>
<td>jump to address</td>
</tr>
<tr>
<td>call</td>
<td>call to subroutine</td>
</tr>
<tr>
<td>ret</td>
<td>return from subroutine</td>
</tr>
<tr>
<td>nop</td>
<td>no operation, only used for time shimming</td>
</tr>
</tbody>
</table>

Listing 18.2: Race in SENIOR

```c
1 fu data_path
2 {
3   input [33:0] op_i;
4       // ports declaration
5   fu::decoder_spec<instr_i, flush_i>() dec_unit;
6   fu::agu_0(XINDR) agu_0;
7   fu::agu_1(XINDR) agu_1;
8   fu::addr_reg(XNOP) areg;
9       //FU declaration
10  phase DE, OF, EX, EX2, WBL;
11  phase ME, WA, MU, AC;
12  stage ff()
13      { cycle {ffo=ff1;}}
14  pipeline conv_pipe
15      {
16      DE -> ff -> OF -> ff -> ME -> ff -> WBL -> MU -> ff -> AC;
17      }
18  pipeline long_pipe
19      {
20      DE -> ff -> OF -> ff -> EX -> ff -> EX2 -> ff -> WBL;
21      }
22  operation(long_pipe) dblld(dec_unit.dblld)
```
18.5 Implementation Details

18.5.1 FPGA Flow

The SystemVerilog code generated by NoGap was synthesized to a Virtex-4 LX80 speed-grade 12 FPGA. Precision was used for synthesis and Xilinx ISE 10.1 was used for mapping and place & route. The generated code contains no FPGA specific optimizations. The data and instruction memory are not part of the processor design, they are instantiated externally, using the block RAMs present in the Virtex-4. The memories are integrated into the data path of PIONEER using external interface FUs.
Table 18.7 displays the hardware usage by part for the synthesized result. Two numbers are given for each column, \( A/B \), \( A \) is the number of elements that belong to that specific hierarchical module, \( B \) is the total number of elements from that hierarchical module and any lower level hierarchical modules below.

The FPGA comparison between original SENIOR and NoGap SENIOR is shown in Table 18.8. In the table, \((o)\) and \((i)\) stands for the pipeline registers at the output and input of the modules.

The timing performance results could make the observant reader a bit surprised to see that the maximum frequency in NoGap SENIOR is higher than in original SENIOR which is heavily optimized and has been designed for a long time. The reason is that original SENIOR have register forwarding implemented which means that forwarding multiplexers are inserted and thus increase length of the critical path. The critical path in NoGap SENIOR is however, almost the same as the critical path in original SENIOR except without a forwarding multiplexer.

### 18.5.2 ASIC Flow

The System Verilog code generated by NoGap was also targeted to an ASIC flow. For this we used a tool chain supplied by Synopsys. A comparison of ASIC implementations, between original SENIOR and NoGap SENIOR is shown in Table 18.9. All the memories, peripherals, and Direct Memory Access (DMA) units, are removed from both versions of SENIORs. The ASIC synthesis tool reported the critical path as well. In the table, \((o)\) and \((i)\) stands for the pipeline registers at the output and input of the modules.

The resulting maximum frequency of the ASIC design was 215 MHz, which is only 10% less than original SENIOR in 65 nm technology with 100% low power logic. The area and power consumption are almost the same in both SENIORs. The critical paths are the same as in the FPGA
Table 18.7: FPGA utilization by part

<table>
<thead>
<tr>
<th>Part</th>
<th>Slices</th>
<th>LUTs</th>
<th>DSP48</th>
<th>BRAM</th>
</tr>
</thead>
<tbody>
<tr>
<td>SENIOR</td>
<td>1250/7240</td>
<td>1211/7508</td>
<td>0/1</td>
<td>3/3</td>
</tr>
<tr>
<td>+data path</td>
<td>1312/5878</td>
<td>800/6136</td>
<td>1/1</td>
<td></td>
</tr>
<tr>
<td>++acc</td>
<td>22/22</td>
<td>42/42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++agu0</td>
<td>104/104</td>
<td>142/142</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++agu1</td>
<td>123/123</td>
<td>166/166</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++alu_pre_b</td>
<td>14/14</td>
<td>17/17</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++addr reg</td>
<td>142/142</td>
<td>213/213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++bitrev0</td>
<td>22/22</td>
<td>34/34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++bitrev1</td>
<td>16/16</td>
<td>31/31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++decoder</td>
<td>2354/2354</td>
<td>2364/2364</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++loopr</td>
<td>92/92</td>
<td>137/137</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++lu</td>
<td>16/16</td>
<td>16/16</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++mflag</td>
<td>24/24</td>
<td>34/34</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++modulo</td>
<td>181/181</td>
<td>167/167</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++mac register</td>
<td>173/173</td>
<td>213/213</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++saturation</td>
<td>27/27</td>
<td>42/42</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++rf</td>
<td>824/824</td>
<td>1090/1090</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++round</td>
<td>28/28</td>
<td>20/20</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++scale</td>
<td>80/80</td>
<td>155/155</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++shift</td>
<td>193/193</td>
<td>334/334</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++stack pointer</td>
<td>33/33</td>
<td>54/54</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++special reg</td>
<td>89/89</td>
<td>122/122</td>
<td></td>
<td></td>
</tr>
<tr>
<td>+sequence path</td>
<td>82/112</td>
<td>74/105</td>
<td></td>
<td></td>
</tr>
<tr>
<td>++pc fsm</td>
<td>30/30</td>
<td>31/31</td>
<td></td>
<td></td>
</tr>
<tr>
<td>total</td>
<td>7240</td>
<td>7508</td>
<td>1</td>
<td>3</td>
</tr>
</tbody>
</table>
### Table 18.8: Comparison for SENIOR in FPGA

<table>
<thead>
<tr>
<th></th>
<th>Virtex-4</th>
<th>Original</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max frequency</td>
<td>75.1MHz</td>
<td>85.02MHz</td>
<td></td>
</tr>
<tr>
<td>Slices usage</td>
<td>5928</td>
<td>7508</td>
<td></td>
</tr>
<tr>
<td>Critical path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>mac_reg(o)→scale→</td>
<td>mac_reg(o)→scale→</td>
<td></td>
<td></td>
</tr>
<tr>
<td>acc→round→sat→</td>
<td>acc→round→sat→</td>
<td></td>
<td></td>
</tr>
<tr>
<td>fwd→mul(i)</td>
<td>mac_reg(i)</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

### Table 18.9: Comparison for SENIOR in ASIC

<table>
<thead>
<tr>
<th></th>
<th>CORE65LPSVT</th>
<th>Original</th>
<th>NoGap</th>
</tr>
</thead>
<tbody>
<tr>
<td>Max frequency&lt;sup&gt;a&lt;/sup&gt;</td>
<td>238 MHz</td>
<td>215 MHz</td>
<td></td>
</tr>
<tr>
<td>Area</td>
<td>63812 μm&lt;sup&gt;2&lt;/sup&gt;</td>
<td>63878 μm&lt;sup&gt;2&lt;/sup&gt;</td>
<td></td>
</tr>
<tr>
<td>Power</td>
<td>3.0511 mW</td>
<td>3.1328 mW</td>
<td></td>
</tr>
<tr>
<td>Critical path</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>scale(o)→round→acc→</td>
<td>scale(o)→acc→round→</td>
<td></td>
<td></td>
</tr>
<tr>
<td>round→sat→fwd(i)</td>
<td>sat→mac_reg(i)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Designing time</td>
<td>&gt; 4.5 years</td>
<td>~ 3 man-weeks</td>
<td></td>
</tr>
</tbody>
</table>

<sup>a</sup>Using ultra low power library
flow.

The design time comparison, shown in Table 18.9, is a bit biased since it was an implementation done from an already existing specification. Still C-H. Wang, who performed the actual NoGap\textsuperscript{CL} implementation had limited previous experience of processor design and specifically had not worked on the SENIOR implementation before. Also the 4.5 years given in the table includes several iterations of SENIOR and addition of DMA and interrupt controllers. Anyhow I am confident enough to draw the conclusion that NoGap eased the design process and C-H. Wang would have had to spent much more time to do the same implementation in Verilog code.
Part VI

Conclusions and Future Work
Conclusions

I think and think for months and years. Ninety-nine times, the conclusion is false. The hundredth time I am right.

Albert Einstein

In this thesis, I have tried to do a comprehensive presentation of the NoGap design framework, a novel generator of accelerators and processors. NoGap puts very few limits on what type of architecture that can be designed, thus enables designers to use NoGap for novel processors design.

A number of cases studies were done in order to prove NoGap’s capability to describe complex pipelined data paths. A floating point adder/subtractor was realized using NoGap. The final hardware generated by NoGap showed comparable results to hand optimized Verilog code. Yet, with a design time significantly less then the original Verilog design. A simple RISC processor with DSP extension, PIONEER, was completely implemented using the NoGap design tools. Proving that processor can in fact be designed using NoGap. Finally a complex DSP processor, SENIOR, was implemented using NoGap. The hardware generated by
\texttt{NoGap}, had almost the same resource utilization and timing characteristics as the version implemented in Verilog. Yet again the design time, for the \texttt{NoGap} implementation, was significantly less than the design time of the original Verilog implementation.

A very important conclusion from all these case studies comes from looking at their respective critical paths. In all cases the critical path is never through any \texttt{NoGap} generated parts. The critical path is through the same elements as they would have been in a normal Verilog design, e.g. adders and multipliers.

With all this in mind I dare say that \texttt{NoGap} is a novel tool that can greatly improve the design process for pipeline based instruction driven architectures. But without locking designers into any predefined architecture, yet still giving support where it is best needed.
Although NoGap has been developed for the last five years and its core principles are proven and tested to be satisfactory, there are still much to do. This Chapter outlines some of the ideas about how NoGap can be improved in the future. Some of the ideas will be easy to implement, while others will require a substantial research work.

20.1 Instruction Level Parallelism

Although it is possible to write processors and accelerators using instruction level parallelism, e.g. Single Instruction Multiple Data (SIMD), Very Large Instruction Word (VLIW), super scalar, or multi core, in NoGap there is no direct support to ease that design process either. Therefore we have started an investigation into what the current limitations are and from there find possibilities for improvements.
20.2 Instruction Format Specification

Currently \texttt{NoGap} automatically synthesizes the instruction formats for a processor. However there needs to be a way to specify the binary format of the instructions generated. This is important, e.g. if a \texttt{NoGap} generated processor needs to conform to an already existing instruction set.

Furthermore, there is no way to limit the number of instructions generated by specifying either illegal or valid combinations.

Related to this area is binary code compression and the trade off between decoder complexity. Having these things automatically generated opens for the possibility to automatically scale the decoder depending on the data path’s critical path.

Future research will focus on investigating what the best methods are for letting designers limit and decide upon the final instruction format.

20.3 Compiler Generation

Currently \texttt{NoGap} can not generate a compiler for processors designed with \texttt{NoGap}. One problem with compiler generation is that \texttt{NoGap} descriptions are too general to be used as input for a compiler generator. However a more limited facet, perhaps a limited version of \texttt{NoGap}^{CL}, could be used as a starting point for compiler generation.

20.4 Cycle Accurate Simulator

20.4.1 Better SSP Generation

The SSP partitioning is done in order to break up false loops in \texttt{Maple} graphs. However there are still cases where this three fold split will cause legal cases to be seen as illegal combinational loops. Consider the simple case of modeling a 74LS04 (hex-inverter) circuit. The SSP partitioning
would connect all inputs and outputs to the same pass node and thus any chaining of the inverters would, wrongfully, be considered an illegal combinational loop. Future research will expand the SSP concept by inserting multiple pass nodes and doing so modeling the true combinational dependencies of the FU. For the 74LS04 case it would mean six pass nodes, one for each inverter.

20.4.2 OL\textsubscript{Inter} Resolving

The simulator spawner can not resolve OL\textsubscript{Inter}s. An idea for a solution is outlined as follows. The first step is to find the minimum set of inserted multiplexer that has to be removed to break all possible OL\textsubscript{Inter}s. Then looking at all possible instructions, a minimum set, of removed multiplexer configuration vectors for the data path, can be generated. Each configuration vector describes a different pipeline architecture to sequentialize. Therefore, for each configuration vector, the corresponding sequentialized code, can be generated, i.e. one cycle execution function for each of the possible configuration vectors. Thus depending on the configuration vector at the start of a cycle, the corresponding cycle execution function is called.

20.4.3 Debugger

On a par with the simulator, a debugger is an important tool to understand where and why something went wrong during the simulation. Therefore future research will investigate the best methods for how a debugger can be generated in conjunction with the simulator.

20.4.4 UI Generation

Having a good User Interface (UI) when running a simulator is important if the simulator shall be useful. The simpler case would be to generate a Command Line Interface (CLI) front end for the simulator core. But
as using text input and output sometimes has its merits, there are other times when Graphical User Interfaces (GUIs) are more convenient, such a case is running simulations. It would therefore be highly beneficial to be able to generate a GUI front end for the simulator. This does not only apply to the cycle accurate simulator, but is likewise crucial for other simulator versions.

20.5 Assembler Generator

The assembler generator so far generates a basic assembler, lacking a number of useful features.

20.5.1 Offline Pipeline Conflict Checking

Currently the programmer is responsible for ensuring that no pipeline conflicts occur and there is no warning or error messages if this will happen. There are however a lot of cases where the pipeline conflict can be statically checked and thus giving the programmer much needed feedback. The information needed to do the pipeline checking can already be found in the NoGapCD, therefore it is a matter of finding the time and/or personnel to implement this improvement.

20.5.2 Constant Definitions

There is currently no way to define constants in memory using the generated assembler. This is not a hard problem, again it is just a question of time and/or personnel.

20.6 Linker Generator

Any serious software development will need a linker to manage libraries of code. It would be beneficial if NoGap could also generate a linker as part of the generated tool chain.
Listing 20.1: Improved Sizing Expression

```plaintext
fu better
{
  parameter IN_PARAM=10;
  input [IN_PARAM-1:0] a_i;
  input [b_i::size-1:0] b_i;
  output [b_i::size+IN_PARAM-1:0] res_o;
  ...
}
```
20.8 Hardware Generation

20.8.1 VHDL Generation

As of yet NoGap can only generate Verilog code, but knowing that many design houses works with VHDL, it would be of great value if NoGap could also generate VHDL code. There are no practical difficulties to do this, its just a question of writing a VHDL generator.

20.8.2 Data Stationary Decoding

As of yet NoGap gives no support for pipelines using a data stationary decoding. Future research are planned to look into how to modify NoGap to handle even data stationary decoding modes.

20.8.3 Target Specific Hardware Generation

NoGap can generate standard SystemVerilog code that is fully synthe-
sizable. Although without any device specific optimizations. Future research will look into how to incorporate custom HDL code depending on synthesis target and target specific spawners.

20.8.4 Hardware Testing

Another important aspect of hardware design is automatic veri fication
and validation. Using NoGap it would be possible to also generate as-
associated test benches and validation code. With more research, NoGap
could also assist Built In Self Test (BIST) generation, useful for chip validation.
20.9 NoGap-Core Improvements

20.9.1 Better Error Reporting
The current error reporting system is not very helpful. Improving this is essential if NoGap shall be usable by any other people than the development team.

20.9.2 GUI
Text interfaces and configuration files have their merits, but integrating it all in a GUI environment might appeal to some engineers and can make the design process easier to follow for engineers new to NoGap. Therefore it would be interesting to investigate the possibilities and merits of adding a GUI front end to NoGap.

20.9.3 Multiple Clock Domains
Currently, NoGap can only be used for designs using a single clock domain. Although this covers a large set of possible design, some designs will be impossible to implement. Investigating how NoGap can be extended to handle multiple clock domains is an interesting future research topic. NoGap might even be able to help with the tricky task of clock domain crossing.
Chapter 20: Future Work


