Development of the ΝοGap\textsuperscript{CL} Hardware Description Language and its Compiler

Examensarbete utfört i kompilatorer och processordesign vid Tekniska högskolan i Linköping

av

Carl Blumenthal

LITH-ISY-EX--07/3960--SE

Linköping 2007
Development of the $\mathcal{N}o\mathcal{G}ap^{CL}$ Hardware Description Language and its Compiler

Examensarbete utfört i kompilatorer och processordesign vid Tekniska högskolan i Linköping av

Carl Blumenthal

LITH-ISY-EX--07/3960--SE

Handledare: Per Karlström
isy, Linköpings universitet

Examinator: Dake Liu
isy, Linköpings universitet

Linköping, 2 May, 2007
Utveckling av det hårdvarubeskrivande språket NoGapCL och dess kompilator

The need for a more general hardware description language aimed specifically at processors, and vague notions and visions of how that language would be realized, lead to this thesis. The aim was to use the visions and initial ideas to evolve and formalize a language and begin implementing the tools to use it.

The language, called NoGapCL Common Language, is designed to give the programmer freedom to implement almost any processor design without being encumbered by many of the tedious tasks normally present in the creation process. While evolving the language it was chosen to borrow syntaxes from C++ and verilog to make the code and concepts easy to understand.

The main advantages of NoGapCL Common Language compared to RTL languages are:

• the ability to define the data paths of instructions separate from each other and have them merged automatically along with assigned timings to form the pipeline.

• having control paths automatically routed by activating named clauses of code coupled to control signals.

• being able to specify a decoder, where the instructions and control structures are defined, that control signals are routed to.

The implemented compiler was created with C++, Bison, and Flex and utilizes an AST structure, a symbol table, and a connection graph. The AST is traversed by several functions to generate the connection graph where the instructions of the processor can be merged into a pipeline. The compiler is in the early stages of development and much is left to do and solve. It has become clear though that the concepts of NoGapCL Common Language can be implemented and are not just visions.

Nyckelord

NoGAP, HDL, processor, compiler
Abstract

The need for a more general hardware description language aimed specifically at processors, and vague notions and visions of how that language would be realized, lead to this thesis. The aim was to use the visions and initial ideas to evolve and formalize a language and begin implementing the tools to use it.

The language, called NoGap Common Language, is designed to give the programmer freedom to implement almost any processor design without being encumbered by many of the tedious tasks normally present in the creation process. While evolving the language it was chosen to borrow syntaxes from C++ and verilog to make the code and concepts easy to understand.

The main advantages of NoGap Common Language compared to RTL languages are:

- the ability to define the data paths of instructions separate from each other and have them merged automatically along with assigned timings to form the pipeline.

- having control paths automatically routed by activating named clauses of code coupled to control signals.

- being able to specify a decoder, where the instructions and control structures are defined, that control signals are routed to.

The implemented compiler was created with C++, Bison, and Flex and utilizes an AST structure, a symbol table, and a connection graph. The AST is traversed by several functions to generate the connection graph where the instructions of the processor can be merged into a pipeline. The compiler is in the early stages of development and much is left to do and solve. It has become clear though that the concepts of NoGap Common Language can be implemented and are not just visions.
Sammanfattning

Behovet av ett mer generellt hårdvarubeskrivande språk specialiserat för processorer och visioner om ett sådant gav upphov till detta examensarbete. Målet var att utveckla visionerna, formalisera dem till ett fungerande språk och börja implementera dess verktyg.


De största fördelarna med att utveckla i NoGap Common Language jämfört med vanliga RTL språk som verilog är:
- att kunna specificera datavägar för instruktioner separat från varandra och få dem automatiskt förenade med hjälp av tidsangivelser till en pipeline.
- att få kontrollvägar automatiskt dragna genom att aktivera namngivna klau-
suler med kod kopplade till kontrollsignaler.
- att kunna specificera en avkodare som kontrollvägarna kan kopplas till där kodning för instruktioner kan anges.

Kompilatorn som implementerats med C++, Bison och Flex använder sig av en AST struktur, en symboltabell och en signalvägsgraf. AST strukturen traverseras av flera funktioner som bygger upp signalvägsgrafen där processorns instruktioner förenas till en pipeline. Utvecklingen av kompilatorn är ännu bara i de första stadierna och mycket är kvar att göra och lösa. Det har dock blivit klart att det är möjligt att implementera koncepten i NoGap Common Language och att de inte bara är lösa visioner.
Acknowledgments

I would like to thank Per Karlström for being a superb supervisor and always helping me out. He has taught me a lot about proper C++ programming and LaTeX. We have both been coding for the implemented NoGap^CL compiler presented in this thesis and the connection graph classes in Chapter 7 are entirely products of his work. He has also provided Chapter 4 of this report and helped in proofreading it. I thank both Dake Liu and Per for allowing me to do this thesis at the Division of Computer Engineering. It has been interesting and instructive to work on a large project from the beginning and help find the solutions to make it work. I have learnt much about compilers, hardware description languages, and foremost programming in general. I wish Per the best of luck in his Ph.D. studies and the continuing development of NoGap.

Cudos to Scott Andrew Borton for making a tutorial on Emacs mode creation. It helped a lot. Thanks Wikipedia, Google, The Free Dictionary, and fredags-grillen... for always being there. Many thanks to my parents for supporting me during my masters studies.

This thesis was carried out between October 2006 and April 2007 at the Division of Computer Engineering, Department of Electrical Engineering, Linköping University.

Linköping, March 2007 (Year of the Golden Pig)
Carl Blumenthal
Explanations

Abbreviations and Explanations

The following table is ordered by logic reading dependencies first and alphabetic order second.

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree. A tree-structure with data that represents the written code of a program.</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit. A module of code with a defined interface and functionality that can be instantiated in other FUs or serve as the top module for a project.</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Description Language.</td>
</tr>
<tr>
<td>NoGap</td>
<td>Novel Generator of Architectures and Processors. The processor construction framework under development in the Department of Electrical Engineering at Linköping University, Sweden.</td>
</tr>
<tr>
<td>NoGap^{CD}</td>
<td>NoGap Common Description. The unified description of a processor that is created by all inputs to the NoGap framework. It consists of three parts; Mase, Mage, and Castle.</td>
</tr>
<tr>
<td>Mase</td>
<td>Micro Architecture Structure Expression. This part of the NoGap^{CD} contains the signal connection graphs of the pipelines of processors.</td>
</tr>
<tr>
<td>Mage</td>
<td>Micro Architecture Generation Essentials. This part of the NoGap^{CD} contains the AST representations of the FUs in a project.</td>
</tr>
<tr>
<td>Castle</td>
<td>Control Architecture STructure Language. This part of the NoGap^{CD} is an, as of yet, undetermined collection of information that should be used to create an instruction decoder.</td>
</tr>
<tr>
<td>NoGap^{CL}</td>
<td>NoGap Common Language. A processor HDL that is the standard way of describing a processor in the NoGap framework. It is input to a compiler that builds a NoGap^{CD}.</td>
</tr>
<tr>
<td>NoGap^{CG}</td>
<td>NoGap Connection Graph. The signal graph representation of pipelines used for Mase that is made primarily by merging instructions written in NoGap^{CL}.</td>
</tr>
<tr>
<td><strong>ADL</strong></td>
<td>Architecture Description Language.</td>
</tr>
<tr>
<td>----------</td>
<td>-------------------------------------</td>
</tr>
<tr>
<td><strong>ALU</strong></td>
<td>Arithmetic Logic Unit. A piece of hardware that performs arithmetic and logic operations on input operands.</td>
</tr>
<tr>
<td><strong>API</strong></td>
<td>Application Programming Interface. A source code interface that a programming library provides to a computer program.</td>
</tr>
<tr>
<td><strong>ASCII</strong></td>
<td>American Standard Code for Information Interchange. A coding used for representing characters in computers.</td>
</tr>
<tr>
<td><strong>ASIP</strong></td>
<td>Application Specific Instruction set Processor. A processor with an instruction set tailored for a specific application.</td>
</tr>
<tr>
<td><strong>Boost</strong></td>
<td>A collection of open-source template libraries extending the C++ functionality.</td>
</tr>
<tr>
<td><strong>BGL</strong></td>
<td>Boost Graph Library. A Boost template library for making, modifying, and accessing graphs. See [19] for further information.</td>
</tr>
<tr>
<td><strong>Vertex descriptor</strong></td>
<td>A BGL “pointer” to a vertex (node) in a graph.</td>
</tr>
<tr>
<td><strong>BNF</strong></td>
<td>Backus-Naur Form. A syntax to describe the syntax of languages (metasyntax). It is used to express context-free grammars. Further reading can be found in [9].</td>
</tr>
<tr>
<td><strong>Dot</strong></td>
<td>A tool from the Graphviz open-source graph visualization software [2]. It reads graph descriptions in a simple text language and can make diagrams in several useful formats, including postscript.</td>
</tr>
<tr>
<td><strong>Doxygen</strong></td>
<td>A program for automatic generation of documentation from programming source-code that requires special commenting. See [6] for further information.</td>
</tr>
<tr>
<td><strong>DSP</strong></td>
<td>Digital Signal Processing.</td>
</tr>
<tr>
<td><strong>Token</strong></td>
<td>A character sequence (lexeme) coupled with a value. Also called a terminal token or just terminal.</td>
</tr>
<tr>
<td><strong>Non-terminal</strong></td>
<td>A token that represents a sequence of terminal and/or non-terminal tokens.</td>
</tr>
<tr>
<td><strong>Lexical analyzer</strong></td>
<td>A program that recognizes character sequences and is used to read files and return values representing the found tokens to another program (typically a parser). Also called lexer for short.</td>
</tr>
<tr>
<td><strong>Scanner</strong></td>
<td>The first stage of the lexical analyzer that recognizes the character sequences. Usually implemented as a finite state machine.</td>
</tr>
<tr>
<td><strong>Flex</strong></td>
<td>An open-source lexical analyzer generator. Allows you to specify different character sequences to recognize and return values representing tokens. See [14] for further details.</td>
</tr>
<tr>
<td>Term</td>
<td>Description</td>
</tr>
<tr>
<td>--------------</td>
<td>----------------------------------------------------------------------------------------------------------------------------------------------</td>
</tr>
<tr>
<td><strong>Parser</strong></td>
<td>A program that performs syntax analysis with respect to a given grammar by processing a sequence of tokens. It is often used to transform text into a data structure such as an abstract syntax tree (AST).</td>
</tr>
<tr>
<td><strong>Bison</strong></td>
<td>An open-source parser generator. It converts a context-free grammar into a parser. You can specify actions in C++ to be taken when a certain grammatical construct is found. See [5] for further information.</td>
</tr>
<tr>
<td><strong>GNU</strong></td>
<td>A project to develop a Unix like operating system and end-user applications as free software with open source-code. The Emacs editor, Flex and Bison are a part of this project and the Linux operating system is actually GNU with a kernel called Linux.</td>
</tr>
<tr>
<td><strong>GMP library</strong></td>
<td>The GNU Multiple Precision library [23]. It enables unlimited precision for signed integers, rational numbers and floating point numbers.</td>
</tr>
<tr>
<td><strong>MIPS</strong></td>
<td>Microprocessor with Interlocked Pipeline Stages. A family of RISC processors from the MIPS technologies company. See [24] and [25] for information on the MIPS32 processor.</td>
</tr>
<tr>
<td><strong>Netlist</strong></td>
<td>A description of the connectivity of an electronic design.</td>
</tr>
<tr>
<td><strong>NOP</strong></td>
<td>No OPeration. A processor instruction that does nothing.</td>
</tr>
<tr>
<td><strong>Regexp</strong></td>
<td>Regular expression. A pattern describing a certain amount of text and/or other characters. For more information see [7].</td>
</tr>
<tr>
<td><strong>RISC</strong></td>
<td>Reduced Instruction Set Computer. A CPU design philosophy that favors using few and simple instructions.</td>
</tr>
<tr>
<td><strong>RTL</strong></td>
<td>Register Transfer Level. An abstraction level for describing the operation in a digital circuit. Used by verilog and VHDL to make “high-level” representations of written circuits.</td>
</tr>
<tr>
<td><strong>STL</strong></td>
<td>Standard Template Library. A C++ template library included in the “std” namespace with containers, iterators and algorithms.</td>
</tr>
<tr>
<td><strong>Subversion</strong></td>
<td>A program for version handling. See [21] for further information.</td>
</tr>
<tr>
<td><strong>Symbol table</strong></td>
<td>Some kind of container used by a compiler to keep track of named objects defined in the code and their associated information.</td>
</tr>
</tbody>
</table>
Notation

Code Referencing

When referring to variables, constructs, functions, or any other named entities from source code, the names are written in a special style in the report. Example: `operateCheck` refers to a function named “operateCheck” in the AST classes.

When excerpts of code are presented in the report they receive some syntax highlighting, a special font, and line numbers that can be referred to in the text. Example:

```cpp
#include <iostream>

int main()
{
    std::cout << "Hello World!" << std::endl;
    return 0;
}
```

Grammars

Grammars are presented with a BNF syntax [9] as follows:

```
⟨foobar⟩ ::= ⟨direction⟩ IDENTIFIER
      | ⟨direction⟩ CONST_NUM IDENTIFIER

⟨direction⟩ ::= input | output
```

Non-terminal tokens are contained in `<>` and terminals are in upper-case letters. For terminals matching a single string, that string is shown in bold font.
Contents

Explanations ix

1 Introduction 1
  1.1 Background ........................................... 1
  1.2 Purpose ............................................... 1
  1.3 Method ............................................... 2
  1.4 Reading Instructions ................................. 2

I Research 5

2 Processor Construction and Issues 7
  2.1 Hazards ............................................... 7
    2.1.1 Data Hazards .................................... 7
    2.1.2 Branch Hazards .................................. 9
    2.1.3 Structural Hazards ............................... 9
  2.2 Hazard Avoidance .................................... 9
  2.3 Processor Framework .................................. 9
  2.4 Processor HDLs ...................................... 10

3 The NoGap Common Language 13
  3.1 Framework Overview .................................. 13
    3.1.1 The C++ API ..................................... 13
    3.1.2 The NoGap Common Description .................... 15
    3.1.3 Spawners ........................................ 15
  3.2 Basic Features of NoGap CL ........................... 15
  3.3 Basic Syntax of NoGap CL .............................. 16
    3.3.1 Functional Unit Descriptions ..................... 17
    3.3.2 Decoder Descriptions ............................. 23
  3.4 Initial Ideas .......................................... 25
    3.4.1 Very Early NoGap CL .............................. 26
    3.4.2 Starting Point ................................... 28
  3.5 Evolution ............................................. 32
    3.5.1 Functional Units ................................ 32
    3.5.2 Decoders ......................................... 34
3.6 Comparisons with RTL Languages ......................... 35
  3.6.1 Advantages of NoGap$^{CL}$ ......................... 35
  3.6.2 Disadvantages of NoGap$^{CL}$ ....................... 36

4 Related Work ........................................ 37
  4.1 LISA ........................................ 37
  4.2 EXPRESSION ................................ 37
  4.3 ArchC ...................................... 38
  4.4 nML ........................................ 38
  4.5 MIMOLA ..................................... 39
  4.6 ASIP Meister ................................ 39
  4.7 Mescal ..................................... 39
  4.8 bluespec .................................... 40
  4.9 SystemC ..................................... 40
  4.10 Summary .................................... 40

II Implementation ......................................... 41

5 Structure ............................................. 43
  5.1 Basics of the NoGap$^{CL}$ Compiler ................. 43
    5.1.1 Errors .................................. 45
    5.1.2 Output Streams ............................ 45
    5.1.3 main Function ............................. 45
  5.2 Limitations .................................... 45
  5.3 Notes on Development ............................. 46

6 The NoGap Common Language Compiler .................. 47
  6.1 Message Streams .................................. 47
  6.2 Errors ........................................ 48
    6.2.1 Basic Errors Library ....................... 48
    6.2.2 Specialized Errors ......................... 49
    6.2.3 The Error Collector ......................... 51
  6.3 Flex Lexer ..................................... 51
  6.4 Bison Parser .................................... 53
    6.4.1 Basic Operation ............................ 53
    6.4.2 Token Declarations ......................... 53
    6.4.3 Grammatical Rules and Actions ............. 54
    6.4.4 Errors .................................. 55
    6.4.5 Special Grammatical Rules ................ 56
  6.5 Symbol Table .................................... 56
    6.5.1 Scope Tree Structure ....................... 58
    6.5.2 SymTab Interface ........................... 58
  6.6 Abstract Syntax Tree ............................. 60
    6.6.1 Base .................................... 63
    6.6.2 Element .................................. 65
    6.6.3 ElementList ............................... 66
6.6.4  Top ............................................. 66
6.6.5  Statement and Construct ......................... 67
6.6.6  Expression ..................................... 67
6.6.7  ConstNum ...................................... 68
6.6.8  Concatenation ................................... 69
6.6.9  Replication ..................................... 69
6.6.10 BinOp and UnaryOp ............................... 70
6.6.11 Assign ......................................... 70
6.6.12 Identifier ...................................... 71
6.6.13 Connection ...................................... 72
6.6.14 Fu ............................................... 74
6.6.15 Instantiation .................................... 75
6.6.16 FuInstStmt ..................................... 76
6.6.17 Clause .......................................... 76
6.6.18 ClauseActStmt .................................. 77
6.6.19 Pipeline ........................................ 78
6.6.20 Operation ....................................... 79
6.6.21 Other Statements ................................. 80
6.6.22 Other Constructs ................................. 81
6.6.23 The operateCheck function ....................... 82
6.6.24 The timingGen function .......................... 85
6.6.25 The buildGraph function ......................... 87
6.6.26 The genVerilog function ........................ 92
6.6.27 The printDot function ............................ 92
6.6.28 Other Statements ................................. 80
6.7  Compilation Output Examples .......................... 94

7  The NoGap Connection Graph ............................ 97
7.1  Overview .......................................... 97
7.2  Descriptions of the NoGap\textsuperscript{CG} Classes .......... 99
7.2.1  Graph .......................................... 99
7.2.2  Module ......................................... 99
7.2.3  AtomicModule .................................... 100
7.2.4  AtomicAddModule and AtomicMulModule ........... 100
7.2.5  Other Atomic Modules ............................ 101
7.2.6  Unit ........................................... 101
7.2.7  FuUnit .......................................... 102
7.2.8  PortUnit ........................................ 102
7.2.9  InPortUnit and OutPortUnit ....................... 102
7.2.10 ArcInfo ......................................... 103
7.2.11 OutsideArc ...................................... 104
7.2.12 DirectArc ....................................... 104
7.2.13 InternalFuArc .................................. 104
# 8 Future Work

8.1 Language Features ................................................. 107
8.2 Implementation Features ........................................... 108
  8.2.1 Decoder Support .............................................. 108
  8.2.2 Spawners ..................................................... 109
  8.2.3 AST Features ................................................ 109
  8.2.4 \texttt{NoGap} \texttt{CG} Features ............................. 110
8.3 Testing .......................................................... 110

# 9 Conclusions ....................................................... 113

Bibliography .......................................................... 115

A The \texttt{operateCheck} Function of \texttt{Identifier} ......... 117

B The \texttt{timingGen} Function of \texttt{Identifier} ............. 122

C Coding Conventions .................................................. 124

D An Emacs Mode for the \texttt{NoGap} \texttt{CL} ..................... 126
  D.1 Mode Features ................................................ 126
    D.1.1 Keybindings and Menu .................................. 126
    D.1.2 Syntax Highlighting ...................................... 126
    D.1.3 Indenting ................................................ 127
    D.1.4 Commenting ............................................... 127
  D.2 Code ........................................................ 127

## List of Figures

3.1 The \texttt{NoGap} framework ...................................... 14
3.2 Pipeline layout of the processor example ....................... 17

5.1 The structure of the compiler ................................. 44

6.1 Inheritance of the basic errors ............................... 48
6.2 Inheritance of the error classes ................................ 50
6.3 Dot representation of a symbol table .......................... 57
6.4 Dot representation of an AST .................................. 62
6.5 Inheritance of the AST classes ................................ 64
6.6 Connection graph legend ....................................... 88
6.7 Raw graph with all inserted nodes and edges .................. 89
6.8 Graph with combined edges and optimized inserted flip-flops 89
6.9 Finished graph with muxes and names ........................... 90
6.10 A functional unit graph cluster ............................... 91

7.1 Dot representation of a trivial connection graph ............ 98
7.2 Inheritance of the module classes ............................. 99
7.3 Inheritance of the unit classes ........................................... 101
7.4 Inheritance of the arc information classes ......................... 103

List of Tables

2.1 Pipeline snapshot 1 of data hazard ................................... 8
2.2 Pipeline snapshot 2 of data hazard ................................. 8

List of Examples

2.1 Data Hazard ................................................................. 8
2.2 Assembly Instruction ..................................................... 10
3.1 Leaf Functional Unit Description ................................. 19
3.2 Top Functional Unit Description ................................. 22
3.3 Decoder Description ...................................................... 24
3.4 Very Early NoGap CL FU Descriptions ...................... 26
3.5 Very Early NoGap CL MOP ........................................... 28
3.6 Starting Point NoGap CL Visions ............................... 30
6.1 Flex Code ................................................................. 52
6.2 Flex C-style Comments ............................................... 52
6.3 Bison Token Declaration Code .................................... 53
6.4 Bison Signal Declaration Grammar ............................ 54
6.5 From NoGap CL Code to a Symbol Table ..................... 56
6.6 The findObject Function ............................................. 59
6.7 From NoGap CL code to an AST ................................... 60
6.8 Transformation of a Connection Graph ......................... 88
6.9 The printDot Function of Connection .......................... 93
6.10 Compilation Output .................................................... 94
6.11 Compilation Output With Errors ............................... 95
C.1 Doxygen Commenting .................................................. 125
Chapter 1

Introduction

1.1 Background

When designing a new pipelined processor either low-level HDL code like verilog or a specialized tool for processor construction can be used. Using verilog has the advantage of providing complete design freedom but requires substantially more effort than using a specialized tool. A specialized language or tool can, on the other hand, force the programmer into specific architectures by disallowing the direct handling of details. An idea emerged at the Division of Computer Engineering, Linköping University, to develop a processor construction framework, with a processor description language at the core, that strikes the balance between providing design freedom and abstraction in the development process. The framework is called Novel Generator of Architectures and Processors (NoGap) and is supposed to bridge the gap between currently existing RTL languages (verilog, VHDL), and high-level processor construction tools. Like many of the currently existing processor construction tools it is envisioned that NoGap should be able to provide a synthesizable hardware description, an assembler, simulators for debugging, and more for written processors.

This thesis began when the project of developing NoGap was at a point where only concept code and general ideas existed.

1.2 Purpose

The aim of the thesis was to evolve the initial ideas for the processor description language, show that it is useful for real world applications, and begin implementing the tools. When refining the language the focus was to improve ease of use and make it more intuitive and coherent. The general nature of the language had to be kept. Implementation was to commence on a compiler for the language that produced synthesizable verilog code. It is easily recognized that the full implementation of the compiler is beyond the scope of this thesis, due to the time limitation. A goal though, was to achieve a chain of some form from input to
verilog code output. To complete the framework and all its features will be a continuing endeavor.

1.3 Method

When developing the language, concept code with syntax ideas was sifted through to identify the core language features that could realize the basic visions of NoGapCL. These language features were then improved while pseudoimplementing a MIPS processor, mostly by using common sense and borrowing syntaxes from verilog and C++. By making sure all the information needed to create the MIPS processor could be contained in the language it was also verified that it would be feasible to use for a wide array of RISC processors.

It was already decided upon to use Flex to make a lexical analyzer that reads input text, and Bison to make a parser that recognizes grammatical rules, for the compiler. Everything else had to be written in C++. The parser was to create an abstract syntax tree (AST) data structure to represent the written code of programs that in turn uses a symbol table to store and retrieve named objects. This is a common way to create compilers and implementation began with these conditions set. The only other outlines for the implementation was to use “proper” object oriented C++ programming and a few specific coding conventions.

1.4 Reading Instructions

The report is divided into two parts. The research part describes the language, how it fits into the NoGap tool chain, the process of developing it, and what other similar tools exist today. It should be accessible to anyone slightly familiar with computer engineering.

The implementation part describes the implementation of the compiler and requires you to have read the overview in Section 3.1, and about the final syntax of the language in Section 3.3. To understand the specifics of the implementation you should at least be familiar with C++. The specific coding conventions we have used are presented in Appendix C. If you don’t know a specific word or abbreviation, please refer to the Explanations chapter starting on page ix. The following resources are my preferred references for the programming languages used.

- The C++ Programming Language by Bjarne Stroustrup [20]. The beginning chapters give a good overview and refreshes your memory about the general use and syntax of the language. The subsequent chapters go into great detail and it feels like you get a thorough understanding of the inner workings of C++.

- The Bison manual [5]. This is the GNU manual for the Bison parser generator language. It contains a few examples that are worth looking at before reading the implementation part of the report.
• The Flex manual [14]. The GNU manual for the Flex lexical analyzer generator language. Look at the examples and then use it as a reference for how input is matched.
Part I

Research
Chapter 2

Processor Construction and Issues

This chapter explains some of the problems involved in making functional pipelined processors and how they can be solved. It also presents desirable properties of a hardware description language (HDL) or other types of construction tools targeted at making them.

2.1 Hazards

Hazards are problems that arise because of the pipelined nature of processors. The source and nature of these problems are described in this section. There are various ways to avoid or deal with hazards, and they are presented in Section 2.2. One option is not to deal with hazards and force the programmer of the finished processor to be smart and avoid hazards manually. It is nice though to have the option of taking care of it automatically.

2.1.1 Data Hazards

A data hazard is when wrong data is being read or written because of another read or write instruction (or multiple others for that matter). There are three different categories of data hazards:

- Read After Write (RAW). The data being read is wrong due to a prior write not yet having finished. For this to happen, the reading and writing must be using the same memory address or register.

- Write After Read (WAR). The data being read is wrong due to a following write finishing before the read. For this to happen, the reading and writing must be using the same memory address or register.

- Write After Write (WAW). When writing the same operand twice, the second one might finish first and the wrong value will stay.
The WAR and WAW-type data hazards are only possible when there is some parallelism in the reading and writing of memory. An example of a RAW-type data hazard is found in Example 2.1.

--- Example 2.1: Data Hazard ---

Table 2.1 and Table 2.2 show snapshots of the pipeline of a processor. “nop” is a no-operation instruction that does nothing. “addi” is an addition with an immediate operand encoded in the instruction. “load” reads a value from a memory address and puts it in a register.

In Table 2.1 a “load” instruction is reading the value from the memory address 0. This value is what the next instruction, “addi”, means to increase by 10. “addi” has begun executing the addition on the register value and has thus already read a wrongful value from the register file.

In Table 2.2 the “load” instruction writes the memory value to register 1. The “addi” instruction has already calculated a wrongful value that it will write to register 1 in the next clockcycle. So the value staying in register 1 will be the value of register 1 before the “load” instruction plus 10 instead of the value of memory address 0 plus 10.

**Table 2.1.** Pipeline snapshot 1 of data hazard

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>decode</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>execute</td>
<td>addi r1,10</td>
<td>Adding 10 to register 1</td>
</tr>
<tr>
<td>memory access</td>
<td>load r1,0x0</td>
<td>Reading value from memory address 0</td>
</tr>
<tr>
<td>write-back</td>
<td>nop</td>
<td></td>
</tr>
</tbody>
</table>

**Table 2.2.** Pipeline snapshot 2 of data hazard

<table>
<thead>
<tr>
<th>Stage</th>
<th>Instruction</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>fetch</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>decode</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>execute</td>
<td>nop</td>
<td></td>
</tr>
<tr>
<td>memory access</td>
<td>addi r1,10</td>
<td>Waiting to write value back</td>
</tr>
<tr>
<td>write-back</td>
<td>load r1,0x0</td>
<td>Writing to register r1</td>
</tr>
</tbody>
</table>
2.1.2 Branch Hazards

When conditional jumps are performed the processor does not know whether to continue processing the instructions directly following the branch or at the branch target address until the branching instruction is completed. If it begins execution of the wrong instructions, it might cause the processor to behave in ways not intended. Branch hazards are also known as control hazards.

2.1.3 Structural Hazards

These hazards occur when two instructions need the same pipeline resource at once. If, for example, instructions of unequal lengths are implemented, a longer instruction might want the ALU at the same time as the following instruction.

2.2 Hazard Avoidance

All of the hazards described in Section 2.1 can be avoided by finding the problematic situations and stalling or flushing the pipeline until it is safe to continue. This is done with control logic that keeps track of fetched instructions and inserts NOP instructions where the pipeline should stall when a dangerous sequence has been detected. Inserting NOPs in all earlier pipeline stages is called flushing the pipeline. Flushing or stalling reduces performance and better solutions should be used where possible.

To avoid WAR and WAW data hazards you could simply make sure that all reads and writes always take an equal amount of time to complete. RAW hazards can be fixed with forwarding. Forwarding is to input a value not yet written to an instruction in an earlier pipeline stage (as soon as it is available) instead of waiting for the write to finish. This requires extra control logic to know which value to use when reading but can reduce the amount of stalling needed, or eliminate it completely, for RAW hazards. Branch prediction and speculative execution can reduce the impact of branch hazards on execution time. Branch prediction tries to find the most likely branch to be taken by the program and speculative execution means beginning to execute that branch. If the prediction is right then normal operation can simply continue, but if it is wrong all the results from the speculative execution must be discarded. Branch prediction can be everything from very simple things, like assuming branches are never taken (called static prediction), to advanced neural branch predictors [26].

2.3 Processor Framework

There is more to a processor than just the actual hardware. To be a useable and potent platform it needs a framework that enables verification and content creation.
Cycle Accurate Simulator

To verify and test a processor a cycle accurate simulator is often used. These simulators model the processor so that its internal states can be viewed between every clockcycle. The programmer can single-step the execution of a program and see that everything is done the way it should be. These simulators can be programmed, for instance, in C++.

Assembly Language

To smoothly create content for the processor an assembly language and accompanying assembler is needed. An assembly language is composed of simple commands with or without arguments. The assembler is the translator of the assembly language and most assemblers perform one-to-one translations. This means commands are translated directly into the machine code of ones and zeros that is needed to activate the instruction in the processor. Some commands can be pseudo-instructions that have a single command syntax but is actually expanded into several other commands. The language can also contain support for macro definitions, labelling of jump addresses and memory locations, commenting, and more. Example 2.2 explains a simple assembly language command.

--- Example 2.2: Assembly Instruction ---

```assembly
mv r1, r2;
```

The command “mv” means “move” and then the destination and source registers are provided. In the command above, the contents of register 2 is moved to register 1. “mv” can be translated by the assembler directly to the operation code of the “move” instruction. The symbolic names of the registers can be translated into memory locations\(^1\) and added to the instruction.

2.4 Processor HDLs

To create a processor is a big investment of time and effort. An HDL targeted specifically at processors is a way of streamlining the process dramatically by avoiding low-level RTL languages. Many things are desireable for a functional processor HDL.

- There should be a complete tool-chain from some kind of input to hardware description and supporting framework. Only making a hardware description could leave the user with a harder task than usual in making the processor framework from the automatically generated descriptions. Allowing ways to automatically generate a cycle accurate simulator and assembler takes a big load off the programmer.

\(^1\) Depending on the architecture we are translating for, it does not have to be memory locations.
• The language should be intuitive and reflect what hardware it will produce. Making the language easy to use and quick to learn saves time and effort. It can also be good if the user can get some sense of what hardware will be constructed before actually synthesising.

• A finished design should be easily changed. You might, for instance, want to quickly reduce the instruction set or merge some pipeline stages into one without it requiring too much work. This enables fast design space exploration for processors that are being written.
Chapter 3

The NoGap Common Language

This chapter will first give an overview of the NoGap framework and its approach to generation of micro architecture. The main subject is the development and final syntax of the NoGap Common Language (NoGap\textsuperscript{CL}) processor HDL that is envisioned as the standard way of describing processors in the NoGap framework. The initial ideas for the language and how the research proceeded while evolving it into its current form will be presented.

3.1 Framework Overview

Figure 3.1 shows the principle layout of the NoGap framework.

The vision for NoGap is one where different programs, called facets, produce a unified description of a processor called the NoGap Common Description (NoGap\textsuperscript{CD}) from various source input. The standard facet will be the NoGap\textsuperscript{CL} and its compiler. NoGap\textsuperscript{CL} is, as will be presented, specially designed to make a processor as easy to design using NoGap as possible. To make useful output from the NoGap\textsuperscript{CD} programs called spawners are employed. These could produce a verilog description, an assembler, or cycle accurate C++ simulator of the processor.

It should be noted that this is a glorified schematic of how the framework should be implemented. The boundaries of the concepts discussed in this section are somewhat blurred in the actual implemented compiler (presented in the implementation part of this report).

3.1.1 The C++ API

The C++ application programming interface (API) is a common name for the classes and functions of the AST and connection graph. The AST classes are the constituents of the AST structure used to represent the written code of NoGap\textsuperscript{CL}
Figure 3.1. The NoGap framework. This is the layout of the flow of information between different parts of the framework. Boxes contain programs or data structures. Arrows represent the flow of information.
3.2 Basic Features of \texttt{NoGap}^\text{"CL"}

processor descriptions. See Section 6.6 for more details. The connection graph classes are used to create and transform connection graph representations of the pipelines of processors. See Chapter 7 for more details. Facets use the API to construct the \texttt{NoGap}^{CD}.

In the case of the \texttt{NoGap}^\text{"CL"} compiler implemented during this thesis (described in Chapter 6), the API is used from the parser and directly from the \texttt{main} function of the program. When the parser (described in Section 6.4) recognizes the grammatical rules of the language it uses the API to build and connect an AST. The \texttt{main} function uses functions in the AST classes to check and modify that AST. The connection graphs of pipelines are automatically generated from the AST. Other facets could have a different approach.

3.1.2 The \texttt{NoGap} Common Description

This is an intermediate description that the facets produce. It consists of three parts; \texttt{Mase}, \texttt{Mage}, and \texttt{Castle}.

\texttt{Mase}, Micro Architecture Structural Expression, is the connection graph descriptions of pipelines with the merged data paths of all instructions. In the \texttt{NoGap}^{CL} compiler the graphs are created for functional units (FUs, modules of \texttt{NoGap}^{CL} code with a defined interface) containing instruction definitions with the AST-traversing \texttt{buildGraph} function (Section 6.6.25). They are composed of the \texttt{NoGap} connection graph (\texttt{NoGap}^{CG}) classes described in Chapter 7.

The functional units not getting a graph description are included in the \texttt{Mage}, Micro Architecture Generation Essentials. It is the AST descriptions of FUs that contain actual functionality.

The final part of the \texttt{NoGap}^{CD} is the \texttt{Castle}, Control Architecture STructure LanguageE. This is the information necessary to generate a decoder for the processor. \texttt{Castle} has, at the time of writing, not yet been formalized but some of the information exists in different forms in the \texttt{NoGap}^{CD} created with the implemented \texttt{NoGap}^{CL} compiler.

3.1.3 Spawners

The only spawner that has been implemented is a limited verilog spawner. It can make a verilog description from the \texttt{Mase} part of the \texttt{NoGap}^{CD}. Verilog descriptions of functional units in \texttt{Mage} must, at the end of this thesis, still be written by hand. The spawner is implemented as the \texttt{genVerilog} function (Section 6.6.26) of the AST classes.

3.2 Basic Features of \texttt{NoGap}^\text{"CL"}

The following is a short description of a few of the basic defining features of \texttt{NoGap}^{CL}, as developed during this thesis, that help set it apart from other HDLs. The syntaxes of these features are presented in Section 3.3 and a quick comparison with RTL languages is done in Section 3.6.
Code is enclosed in functional units constructs (FUs) that are modules with defined interfaces. The output and input ports of the FUs and their internal signals are the only signals in the language. Signals used for data paths and control paths are treated differently by the language. Control signals are automatically routed to where they should go by activations of named bits of code known as clauses. Clauses are mainly the different choices of a branching switch construct and clause activations determine what kind of functionality is wanted by choosing the branches.

Instructions and their data paths are specified in groups with clause activations to determine the functionality for the instructions. All the instructions are merged automatically into a pipeline with the help of timing information supplied in the written data paths. FUs can be instantiated as many times as needed and used throughout the pipeline.

The codings of instructions are specified in constructs known as decoders. They can then decode incoming instructions and output control signals that have been routed to them and data signals that help define the instructions (such as operand and destination register addresses). The inputs and outputs of the decoders are connected in the pipeline to enable them to do this.

### 3.3 Basic Syntax of \texttt{NoGap}^{CL}

A \texttt{NoGap}^{CL} processor description is, as decided during the course of this thesis, divided into three different design parts; leaf functional units, top functional units, and decoders.

Top FUs define the data paths and behaviour of instructions and thereby define the pipelines of processors. Leaf FUs define basic functionality like memories, register files, and arithmetic logic units (ALUs). Decoders define how instructions are to be decoded and can add control structures like forwarding to the design. They can also define the assembly language commands of the instructions. In their envisioned, not yet finalized, form the decoders can be seen as all-knowing entities that can access any signal from any phase in the pipeline and alter it or build some kind of control structure around it.

Functional units are written in functional unit description files. Files with the “.fud” file extension will be recognized as a \texttt{NoGap}^{CL} functional unit description file in Emacs if the mode described in Appendix D is installed. It will give the code the right syntax highlighting and indenting. Decoders are written in decoder description files with the “.dd” extension. There is no Emacs mode implemented for the decoders yet.

This section uses a very simple processor example to describe the basic syntax of \texttt{NoGap}^{CL}. The processor example has five instructions using an ALU; ADD, SUB, AND, OR, and XOR, employs register forwarding, and allows flushing and stalling of the pipeline. The processor will need a memory where instructions can be read, a register file, a program counter, and of course an ALU. It is limited to operating on values already present in the register file and writing to that same register file. A layout of the pipeline of the processor can be found in Figure 3.2.
3.3 Basic Syntax of NoGapCL

Figure 3.2. Pipeline layout of the processor example. The four pipeline stages of the processor are separated by vertical bars representing the pipeline registers. The arrows represent data paths available to the instructions. Four instantiated FUs are used to describe the functionality.

It should be noted that the description of the language that is given in this section reflects how the language SHOULD work and much of it is still not implemented. Nothing has, for instance, been implemented for decoder descriptions so far. The focus has been on implementing the core features of the top and leaf functional unit descriptions first.

3.3.1 Functional Unit Descriptions

There are two different types of functional units; the ones with operation constructs, top FUs, and those without operations, leaf FUs. Top FUs define instructions in groups with equal data paths in the operation constructs. They are also used to generate the graph descriptions of the Mage. The syntax is a bit limited in these FUs. Switch constructs, for example, are not allowed. In leaf FUs all constructs and statements (except operations and pipelines) are allowed. The leaf FUs are the constituents of the Mage.

Leaf Functional Units

Example 3.1 shows the code for a memory where words (32 bits) can be written and read. The FU is a leaf FU since it does not define any instructions for the processor but only some kind of functionality. Similar to verilog modules or C++ functions the functional unit constructs have a ports-preface section within parenthesis where the input and output ports of the unit are specified. The functionality is specified within curly brackets. The FU is called “mem”, and lines 3-7 of the code is the ports-preface section. Comments are written with standard C++ “//”, and C-style “/**/” syntaxes.
Ports are declared as input or output and with a certain bitrange. Line 3 declares an input port named “data_i” with bits from 31 down to 0. That makes it 32 bits wide. The bitrange can be declared between any two numbers as long as the first number is bigger than or equal to the second. If no bitrange is specified the port defaults to 1 bit (or actually a range from 0 to 0). The output port on line 6 has also got a specified timing offset of one clockcycle. This is the time it takes from a change of the inputs to a valid value on the output port. It is used in assigning and checking the timings in the AST timingGen function (Section 6.6.24). The offset should actually be zero for this port because there are no registers in its path. Using this description would result in the wrong value of “data_o” to be pipelined when instantiating a memory. Only output ports can have offsets.

A signal is declared on line 9. Signals are internal to the functional unit. It is called “memory”, has a bitrange of 7 down to 0, and there are 1024 instances of the signal. Every instance gets 8 bits of 7 down to 0 and they can be addressed one at a time or all at once. Addressing and assigning all 1024 signals at once is generally a big mistake because of the hardware it will produce. Instances can be specified for ports as well.

Reading (line 12-15) is done continuously and, since the memory is byte-addressed, four consecutive addresses (signal instances) must be read to make the output word. The reading is purely combinatorial. The identifiers used to reference the output port and “memory” signal have the same syntax for addressing bitrange and instances as the declarations of ports and signals. The instances number now determines which instance to address instead of how many to create. It is allowed to use a single number for the bitrange to access only one specific bit in the range.

Writing (line 18-33) is done within a cycle construct. The cycle constructs contain clocked logic and all signals assigned therein are made into registers. A switch construct is used to provide the choice of either writing or doing nothing every clockcycle in the cycle construct. It uses a control signal specified within parenthesis as the signal to switch upon. Since “control_w_ci” is a port the switch can only be controlled from the outside of the FU. Had it been a signal, the switch would only be controllable from the inside of the FU where the signal could be found from the current scope. The selections available in the switch are specified as “choice” and “default” clauses. There has to be exactly one “default” clause per switch and its action is done whenever none of the other clauses are active. Names of clauses must always be written like “%NAME”, and in the code presented it has been chosen to write the names in only capital letters to further distinguish between normal identifiers and clauses. Special clause activation statements are used to choose which clause of a switch is active at any given time. The coding of the clauses in the control signal is hidden from the user.

The “WRITE_WORD” clause has an action that writes four consecutive byte instances of the “memory” signal, that is actually a register now since it is assigned in a cycle construct. The “IDLE” clause does nothing and all values in “memory” remain unchanged.
### Example 3.1: Leaf Functional Unit Description

```cl
// Memory functional unit with read/write functionality
fu mem(
    input [31:0] data_i; // Writing data input
    input [31:0] addr_i; // Address input
    input [2:0] control_w_ci; // Control signal for writing
    output {+1+} [31:0] data_o; // Read output data with 1 clk offset
)
{
    signal{:1024:} [7:0] memory; // 1024*8 bits = 1kB of memory

    // Combinatorial read
data_o[7:0] = memory{:addr_i+3:};
data_o[15:8] = memory{:addr_i+2:};
data_o[23:16] = memory{:addr_i+1:};
data_o[31:24] = memory{:addr_i:};

    // Clocked write cycle
    switch(control_w_ci)
    {
        choice: %WRITE_WORD
        {
            memory{:addr_i+3:} = data_i[7:0];
            memory{:addr_i+2:} = data_i[15:8];
            memory{:addr_i+1:} = data_i[23:16];
            memory{:addr_i:} = data_i[31:24];
        }
        default: %IDLE
        {
        }
    }
}
```

### Top Functional Units

Example 3.2 contains the top FU description for our simple processor. Line 1 includes a file with the instruction decoder from Section 3.3.2. Line 2 includes the file containing the descriptions of the ALU, program counter, register file, and the memory from Example 3.1.

No ports are needed for the FU since instructions, we imagine, are already placed in the instruction memory and the instructions only modify internal registers of the processor. Lines 8-12 instantiate the functional units from the descriptions in “test_fus.fud”. The name following “fu::” is the name of the FU description to instantiate and

---

1Pretty useless... but it's an example.
the rightmost name is the name of the instantiation object to create. Between the names you can specify a list of default clauses if you want another clause to be the default one in a switch. In an adder FU you might, for example, want to have unsigned addition instead of signed as the default. The decoder instantiation on line 15 is syntactically analogous to FU instantiations.

The pipeline constructs on lines 18-27 contain the different phases that define the pipeline stages by providing timing information used when defining instructions. First the “fetch_pipe” pipeline is defined with two phases “instruction_fetch_p” and “instruction_decode_p”. The “normal” pipeline adds the “execution_p” and “write_back_p” phases to the ones defined in “fetch_pipe”. It is done by “instantiating” the smaller pipeline with “pipeline::” plus the name of the pipeline. This simply dumps the contents within braces for the instantiated pipeline into the longer pipeline. The list of phases defined within braces has a few special operators interlocking the phases that are used to define if they can be stalled and flushed. Depending on the operators, the timing represented by the phases can either be increased between two phases (left to right), or they can be set to describe the same pipeline stage by making the timing difference between them zero. This enables the pipeline length to be decreased by merging stages, which is useful if there is a large delay in one stage and two other stages can be merged without forcing a lower clock frequency for the processor.

- “>” means that execution progresses in the right phase the next clock cycle. The left and right phase get different timings.
- “-” means that the right and left phase get the same timing, and thereby mark the same pipeline stage.
- “|” means it is allowed to stall the pipeline in the right phase.
- “/” means it is allowed to flush the pipeline in the right phase.

Every pipeline is assigned a decoder (or several). Both “fetch_pipe” and “normal” have “ID” as their decoder. This tells “ID” about all the phases they contain and makes it responsible for decoding all the instructions defined in those pipelines.

The operation constructs on lines 30-62 are each assigned a pipeline within parenthesis. This sets the phases that can be used within the construct and all instructions defined in the operation is added to that pipeline and can be used by its decoder. The functionality of the operation is specified in a special operation-type clause. These clauses cannot be activated like clauses of switches but the name of the clause is important to define the names of the instructions inside.

Line 32 is a phase statement. The “@phasename” syntax is always used to activate phases and set their timing as the current timing. Assignments written after a phase statement take place in the timing of that phase. No assignments can be written in the operation before a phase has been specified.

Line 33 is an assignment of the address input port of the instruction memory with the value of the next program counter output from the instantiated program

---

2 It is unclear what it would mean to define a different decoder for a sub-pipeline than the parent pipeline.
counter FU. The “-” instead of “.”-operator for retrieving ports means that we want a direct connection from that port. A direct connection bypasses any timing considerations that would insert pipeline registers or say the value does not exist yet. Using the “.”-operator when reading in operation constructs means that we want the pipelined value of the signal from where it was first introduced in the operation. When we, for example, assign the operands of the ALU we know that the output value is available the next clockcycle. The output then gets the same timing and place in the pipeline as the phase we assigned the inputs in. The right amount of pipeline registers will be inserted if we use the ALU output at later phases. If we know the output will take 3 clockcycles to complete we must assign the output an offset of 3 in the FU description (in the same way as is done on line 6 in Example 3.1). Line 36 is where the instruction read from the instruction memory is fed to the decoder.

The “FETCH” operation describes how instructions are fetched from the instruction memory. The program counter is used as the address and data is read and fed to the decoder. The instruction fetch procedure is the same for all instructions that could be implemented for this processor and that is why it has been made into its own operation. The “FETCH” operation is “instantiated” in the “ALU” operation on line 42. As in the case of instantiating pipelines it means that we dump the contents within braces of the instantiated operation into the current one.

The “ALU” operation defines the five operations we want for the processor. The register address outputs of the decoder are assigned to the address inputs of the register file in the “instruction decode” phase. For each instruction the decoder knows what bits of the instruction to output as the register addresses, and this way they are used to retrieve the right registers from the register file.

In the “execute” phase the read registers from the register file are input as the operands of the ALU and the various functions of the ALU are applied. Line 55 is a multiple-choice clause activation statement. At this point we can choose to activate either one of the five clauses in the instantiated object named “alu”. This branches the operation from one instruction, named simply “ALU”, into five different ones; “ALU::ADD”, “ALU::SUB”, “ALU::AND”, “ALU::OR”, and “ALU::XOR”. Each one activates a different clause in the multiple-choice activation statement. One or more clauses can also be activated at once without branching the instructions. “alu(%ADD, %SUB, %AND, %OR, %XOR);” would try to activate all the clauses for all instructions. If the clause we want to activate can be found in the current scope, simply “%ADD;” can be used to activate the clause “ADD”.

In the “write back” phase the result from the ALU will be written back to the register file. The output from the ALU is connected to the data input of the register file. The destination register address output from the decoder is connected to the write address of the register file and its “WRITE” clause is activated.

---

3It probably wouldn’t work since they most likely are from the same switch.
Example 3.2: Top Functional Unit Description

```c
#include "test_decoder.dd";
#include "test_fus.fud";

// Simple processor functional unit with 5 instructions
fu test_proc() {
    // Instantiating functional units
    fu::alu(%OR) alu;    // ALU
    fu::register_file(%IDLE) regfile; // Register file
    fu::pc(%IDLE) pc;    // Program counter
    fu::mem(%IDLE) imem; // Instruction memory

    // Instruction decoder instantiation
decoder::test_decoder(%IDLE) ID;

    // Defining a pipeline with 2 phases and decoder ID
    pipeline fetch_pipe(decoder ID) {
        instruction_fetch_p |> instruction_decode_p
    }

    // Defining a longer pipeline by adding to fetch_pipe.
    pipeline normal(decoder ID) {
        pipeline::fetch_pipe |> execute_p |> write_back_p
    }

    // Common fetch for all operations
    operation(fetch_pipe) %FETCH {
        @instruction_fetch_p;
        imem.addr_i = pc->next_pc_o; // Program counter to memory address

        @instruction_decode_p;
        ID.input_i = imem.data_o; // Memory output to decoder input
    }

    // ALU operations with regs
    operation(normal) %ALU {
        operation::%FETCH; // Instantiating FETCH operation

        @instruction_decode_p;
        // Decoded register addresses input to register file
        regfile.addr1_i = ID.reg1_o;
        regfile.addr2_i = ID.reg2_o;

        @execute_p;
        // Read registers input as ALU operands
        alu.data_a_i = regfile.out1_o;
        alu.data_b_i = regfile.out2_o;

        // Choosing ALU function and branching into 5 instructions
        alu(\{%ADD, %SUB, %AND, %OR, %XOR\});
    }
```
### 3.3 Basic Syntax of mCap\(^C\)L

3.3.2 Decoder Descriptions

The decoders are written in much the same way as functional unit descriptions but with many other constructs and built-in functions. They have a port-preface section and their functional content is enclosed within curly brackets. Example 3.3 contains the code of the decoder for our test processor. The decoder is called “test_decoder”.

Operation types define what the bits of an instruction are used for. Line 12 makes an operation type called “R_type”. In this case we have \(2 \times 6\) free bits to distinguish different instructions. They are placed first and last in the 32-bit word. The rest of the bits are connected to different outputs.

Operation types are used when defining instructions in the “operation_codes” construct. The operation codes of this decoder are connected to the “input_i” signal that is 32 bits wide. All the operation types used in the construct must therefore be 32 bits wide also. Instructions from the top functional unit descriptions are identified with their names as defined in the previous section, for example “%ALU::%ADD”. When writing the instruction names in the code the % character has to be used in front of all the clause names to indicate that they are clause names. “%ALU::%ADD” is given its coding on line 20. The “R_type” operation type is used (almost as a function in C++), and the free bits of the operation type are assigned the argument values. “6b100001” means 6 bits of binary value 100001. “ADD” is thereby given the “R_type” operation type and the \(2 \times 6\) bits of free coding are assigned values to enable identification of the instruction. After the “->”-operator the assembly language command of the instruction can be specified. This particular addition is the add unsigned “ADDU” instruction and using it could look like; “ADDU r3,r1,r2”, meaning add register 1 and 2 and put the results in register 3. All the instructions defined in the “ALU” operation of the top FU can be assigned coding and assembler command within the braces after “%ALU::”.

Forwarding (line 32-40) is done with a special “forward” construct. Forwarding is explained in Section 2.2. In the construct the data dependencies between instructions that lead to data hazards can be specified so that the decoder can deal with them. First the decoder needs to be told between which connections the forwarding is to be done. In the processor example it is between the data input and output of the register file. Second the criterion that has to be true for the forwarding to take place must be supplied. The “@from” and “@to” are the
timings of where to fetch the pipelined signals. Pairs of “to” and “from” timings are defined with receding priorities within the braces. If criterions for two different pairs are true at the same time, the pair found at the top is used. Here we check if the destination register of an earlier instruction is the same as a register we want to read from the register file. We check three phases forward and the most recent instruction gets the highest forwarding priority. What actual signals to connect are determined by what instructions are in the “@from” and “@to” phases at the moment. In this case the signal connected to “regfile.data_i” in the “@from”-instruction is forwarded to replace “regfile.out1_o” of the “@to”-instruction. The forwarding in the example is actually incomplete. To complete it, another “forwarding” construct is needed that performs similarly for “regfile.out2_o”.

Switches can also be used in decoders and they are analogous to switches in FU descriptions. On lines 46-66 one is used to enable flushing and stalling. The “FLUSH” choice-clause (lines 48-55) flushes the pipeline in the phase where the clause is activated in the two following clockcycles if flushing is enabled in the phase. Line 44 uses the “@phase” statement to set what phase(s) the “@this” statement (lines 50,58) is allowed to be. The “@this” statement becomes the phase where the clause containing the statement is activated. “@any” means that “@this” can be any defined phase. “@this+$1” references the current phase in the next clockcycle. The “flush_enabled” function returns true if the argument phase is allowed to flush. The “flush” function flushes the pipeline in the argument phase. The “STALL” choice-clause (lines 56-62) is written in a similar fashion.

Example 3.3: Decoder Description

decoder test_decoder {
  input [31:0] input_i;       //Instruction input
  input [auto] flush_stall_ci; //Flushing/stalling control signal
  output [4:0] reg1_o;        //Operand register 1
  output [4:0] reg2_o;        //Operand register 2
  output [4:0] dest_reg_o;    //Destination register
  output [4:0] sa_o;          //Shift amount for shift instructions
}

//Defining operation types
operation_type R_type = {free (6), reg1_o, reg2_o, dest_reg_o, sa_o, free (6)};

//Defining operation codes
operation_codes(input_i)
{
  //ALU operations with regs
  %ALU:
  {
    %ADD = R_type(6b000000, 6b100001) -> ADDU dest_reg_o, reg1_o, reg2_o;
    %SUB = R_type(6b000000, 6b100011) -> SUBU dest_reg_o, reg1_o, reg2_o;
    %AND = R_type(6b000000, 6b100100) -> AND dest_reg_o, reg1_o, reg2_o;
    %OR  = R_type(6b000000, 6b100101) -> OR  dest_reg_o, reg1_o, reg2_o;
    %XOR = R_type(6b000000, 6b100110) -> XOR dest_reg_o, reg1_o, reg2_o;
  }
}

}
3.4 Initial Ideas

The current form of NoGap$^{CL}$ originates from ideas of a language where signals of the control paths and data paths are separated and signals used for control are automatically routed and assigned based on activations of named clauses of code. The code was defined in functional unit modules with defined interfaces. Connections between signals in the pipeline were made with a stack-based system. If you specified an output signal, it was put on the stack, and if you specified an input signal it was assigned an output removed from the top of the stack. If they matched by input and output sizes, whole FU interfaces could be assigned at once. Many of the concepts, like functional units, hiding control routing from the
user, and clauses are, as previously presented, still used today but implemented somewhat differently than described in this section. The code examples shown will also contain many envisioned features that could be included in some form in future NoGap\textsuperscript{CL} revisions.

### 3.4.1 Very Early NoGap\textsuperscript{CL}

To explain the workings of early NoGap\textsuperscript{CL} some old concept code is displayed in Example 3.4 and Example 3.5. In early versions of the language a postfix operator syntax was used for expressions. The feature of defining a separate decoder was still a mystery at this point, but the code is divided into functional unit descriptions and the micro operations (MOP). A micro operations file, with the “.mop” extension, defined the instructions of the processor much like the top functional unit descriptions of the developed NoGap\textsuperscript{CL}. The two upcoming code examples will describe a processor that can only move information between registers of a register file.

Example 3.4 contains the functional unit descriptions. The “DECODER” functional unit (lines 3-12) is acting as the decoder for the processor. It has an input “instr\_i” that gets its size defined automatically by the signal that is later assigned to the input. The automatic size scaling is a feature that will soon be implemented in the current NoGap\textsuperscript{CL}. Every FU has a “cost” variable. It sets the cost of execution time, hardware or any other relevant cost for the FU. It was only a vague concept at this time. Line 4 sets that cost for “DECODER” as 1 of an undetermined unit.

Operations (“op::”) here are the equivalent of the clauses of today. The unnamed operation on lines 8-11 will be performed for all instructions. Its action is the magical decoding of the “instr\_i” signal. A named operation in a FU will produce a separate instruction when the FU is used in the micro operations file.

The “REGISTER\_FILE” register file FU (lines 14-38) has ports with sizes determined by the “native” constant defined on line 1. “src\_a”, “src\_b” and “dst” are declared on lines 21-23 as values coming from the instruction decoder. They are used to access the different memory instances of “rf” declared on line 25. “rf” is used as the actual registers and is read and written in the “WRITE\_ZERO” and “WRITE\_ONE” operations.

---

**Example 3.4: Very Early NoGap\textsuperscript{CL} FU Descriptions**

```
[15:0] native :=;
2 fu := DECODER
4 { 1 cost =;
6   input instr\_i[auto];
8   op::
10   { instr\_i Decode;
12 }```

---
Example 3.5 contains the code of the micro operations file for the processor. First the used FUs are instantiated and then the control (“control::”) and instruction (“instr::”) constructs are defined. The “MIPS” control (lines 6-11) defines the first three stages of the pipeline and its purpose is to make the control signals used in the subsequent instruction constructs that use it. The three instantiated FUs “pc”, “if”, and “de” are used for the “pc_stage”, “if_stage”, and “de_stage” pipeline stages respectively. “{0}” is an absolute reference to the first pipeline stage. The stack-based connecting is used and the outputs of “pc” (making the program counter) must be compatible with the inputs of “if” (fetching the instruction), and so on.

The “MOVE” instruction (lines 13-17) connects the data path of our simple instruction. It is specified as using control signals from the “MIPS” control-generating construct. “<1>” is a relative reference to the first pipeline stage after the ones used in “MIPS”. Line 15 puts the first output of “rf” from stage 3 on the stack and line 16 connects it to the first input of “rf” at stage 7. We hereby specify explicitly what stages we connect signals between and the correct amount of pipeline registers can be inserted. This is done implicitly in current NoGapCL.
Example 3.5: Very Early NoGapCL MOP

```
PROGRAM_COUNTER pc;
INSTRUCTION_FETCH if;
DECODER de;
REGISTER_FILE rf;

c: : MIPS
{
| 0 | pc pc_stage;
| 1 | if if_stage;
| 2 | de de_stage;
}

instr : : MOVE( MIPS )
{
<1> rf [ out : 0 ];
<5> rf [ in : 0 ];
}
```

The concept of operations used here is vague at best and especially about what operations will be used for a specific instruction. The only way of finding out where and when the “WRITE_ONE” operation of “REGISTER_FILE” could be active is by finding places where the input of “REGISTER_FILE” is assigned. Too much of the complexity of control routing is hidden and the user will easily be confused about what instruction and hardware will actually be created. To find the possible outcomes of the data path of an instruction construct while writing the processor, all the FUs used will have to be searched to find operations that would fit. The possible outcomes should be restricted and specified together with the instructions similar to the multiple-choice clause activations of current NoGapCL. This gives a much better overview of the actual instructions created.

Controls and instructions are similar in appearance and functionality and it feels like having both of them is superfluous and could be streamlined to one construct. It also locks the user into an architecture where the first stages of the pipeline must be used to decode instructions.

3.4.2 Starting Point

When I entered the project a lot of pseudo-code had been written to try different approaches and explore the usability of various prototype syntaxes. Example 3.6 contains a collection of such code. Even at this point there were no ideas as to how the decoder descriptions would be written. Below is a list of comments about interesting features and differences compared to the current NoGapCL from the example.

- The converters concept was used to specify what output was wanted from functional units and give directives to the spawners. There is a declaration
of the default converters on line 1 and directives are given to these converters on lines 52-59. The only equivalent to this in current NoGap\textsuperscript{CL} is a verilog replacement statement (Section 6.6.21) that enables us to specify a verilog file to replace a FU in the verilog spawner. More functionality of this type will be added as the spawners become more advanced.

- Interfaces could be defined and later instantiated in the ports-preface section of FUs. It enables a quicker and more space-efficient way of defining the interfaces of FUs as ports. Lines 4-8 define an interface that is instantiated on lines 13 and 62. There is no reason why this would not be implemented and a shell for the interface class has been made during this thesis in the AST classes.

- When using specific ports in an interface the "."-operator was used much like retrieving member variables in C++. This has been expanded to be the main way of referencing a named object in a named object.

- Input and output ports could be written either as control-ports or as data-ports. This distinction is no longer made when declaring the ports as the usage of the ports in the rest of the code will determine what type it will be.

- A special construct known as a control “issuer” had to be declared and connected to control ports. The current “issuer” was supposed to be the decoder that fed control signals to the FUs. Instead of using the “issuer” concept the data paths of the decoders will now be explicitly connected, and the clauses needed for an instruction will automatically have routed control signals to the decoder. A control port is assigned from the active “issuer” on line 12.

- Signals were introduced to represent internal connections in the FUs. They could be assigned in “cycle”-constructs of clocked logic to make registers in a way very similar to VHDL. The contents of “cycle”-constructs were enclosed in named clauses to unify the notion that clauses with names were used to group code in FUs. This notion has since been changed to mark named clauses as something meaningful to activate and choose by way of referencing its name.

- In switches the clauses could either be assigned a specific value of the control signal or written in the “choice”-“default” syntax of today. The same functionality will be implemented in future NoGap\textsuperscript{CL} revisions in some way. It would also be possible to have a way of using an automatically assigned control value in the code without knowing it beforehand.

- Pipeline constructs (lines 66-70) were used to define the possible phases much like today. Instead of only operators interlocking the phases a control signal and a verilog-assignment-type if-then-else statement could define the allowed transitions.
- The stack-based connecting system was still used for the signal connections in the operation constructs that now defined instructions. The timing of when the connections were extracted or inserted was determined by the timing of the currently active phase. Phases were activated with the “@phase” statement.

- Operation “TEST2” on lines 82-91 contains early clause activations on lines 89 and 90 where predefined lists of clauses could be activated. Line 87 is the equivalent of today’s operation “instantiations”.

---

**Example 3.6: Starting Point $	exttt{NoGap}^{CL}$ Visions**

```plaintext
converters::default = {verilog, cpp};
const range native = [15:0];

interface test_interface
{
  input::data [5:0] simple_range_i;
  output::data simple_o;
}

fut test(
  output::data simple_o;
  input::control [auto] simple_ci = issuer::active [auto];
test_interface test_inter ;
) {
  signal [auto] auto_signal ;

cycle
  %SWITCH_TEST_1 {
    switch (simple_ci)
    {
      1b0:
      %CO_IS_ONE {
        simple_o = 1;
        test_inter.simple_o = 1;
      }
      1b1:
      %CO_IS_ZERO {
        simple_co = 0;
        test_inter.simple_o = 0;
      }
    }
  }

cycle
  %SWITCH_TEST_2 {
    switch (simple_ci)
    {
      choice:
      %CO_IS_ONE {
        simple_o = 5b00001;
        test_inter.simple_o = 5b00001;
      }
      %CO_IS_ZERO {
        simple_co = 1;
        test_inter.simple_co = 0;
      }
    }
  }
```
3.4 Initial Ideas

```cpp
     } default:
     %CO_IS_ZERO {
       simple_o = 5b00000;
       test_inter.simple_o = 5b00001;
     }
   }
   
   converter::verilog
   {
     cdl.convertTo(verilog);
   }
   converter::cpp
   {
     file("behav.cpp");
   }

   fu operation_test(test_interface ti;)
   {
     operations
     {
       pipeline base {pc_p > fe_p > de_p};
       pipeline normal {@base > ex_p > wb_p};

       pipeline argument(stall stall_it) {pc_p (stall_it ? | : >) fe_p
                                              (stall_it ? | : >) de_p};

       fu::test t1;
       fu::test <4> tfu;

       operation TEST1(normal)
       {
         @pc_p; t1;
         @fe_p; tfu <0>;
         @de_p; tfu <1>;
       }

       operation TEST2(normal)
       {
         instruction co_one_zero = (<%CO_IS_ONE,%CO_IS_ZERO>);
         instruction co_sr = (<%CO_SR_0,%CO_SR_1,%CO_SR_2>);

         (@pc_p - @de_p); TEST1;
         @ex_p;
         ci_tfu0(<%CO_SR_0>)@[simple_o];
         ci_tfu1(<co_one_zero,co_sr>)@[simple_i];
       }
   }
```
3.5 Evolution

As shown in Section 3.4 there were a lot of ideas to consider for the functional unit descriptions. When developing the language it was a question of filtering out the essential and the best ideas and making them fit together coherently. The syntax of decoder descriptions would have to be designed from scratch.

Below is a list of the core language features that were identified by sifting through concept code and comparing with the main ideas for the functionality of the language. These features were what had to be improved and later implemented in a compiler as far as possible during the thesis.

- Ports and signals are used as external and internal connections for FUs. These connections have a specified bitrange and can have multiple instances.

- The activation of clauses in switch constructs that defines functionality and routes control signals.

- Specifying pipelines and their phases with different flushing and stalling permissions.

- Operation contracts define the data path and possible controls to form the instructions of the processor.

- FUs can be instantiated and used in the operation constructs.

- Some kind of decoder that issues control signals to form different instructions based on an input signal.

Pseudo-implementation began on a real processor to find solutions that contained all the information necessary to create the processor, and in addition was easy to understand. Some things were chosen to make the implementation of the compiler easier. Based on its rather simple architecture and instruction set the MIPS32 processor was decided upon for the implementation. The architecture of the processor is described in [24] and its instruction set in [25].

Changes described in Section 3.5.1 and Section 3.5.2 are changes from the syntax and concepts described in Section 3.4.2.

3.5.1 Functional Units

The need to declare ports as either control or data ports was soon removed from the syntax. It seemed to make sense to let the context in which the ports or signals are used determine their type, as with signals being made into registers with memory functions when assigned in cycle constructs. Ports used for controlling switches are naturally used for control so the ports could be set as control signals upon finding them in the clause construct instead. This made the port declaration syntax more verilog-like and naturally less intimidating to the user. The rest of the syntax of connection declarations proved functional and understandable. Some further alterations were made to make the grammar in the Bison parser (Section 6.4)
3.5 Evolution

Instances are now specified with “{::}” to distinguish it from the lower-than “<” and bigger-than “>” operators of expressions.

When referencing connections, the identifier syntax was chosen to resemble that of C++ where member variables are fetched from classes. C++ or C are languages that should be familiar to most people that are going to be users of NoGapCL. Fetching the signal “bar” from the instantiated FU named “foo” will therefore look like “foo.bar”. If we only want to fetch a certain bitrange of a certain instance of “bar” we add the verilog-type syntax of signals to the identifier; “foo.bar::{1::} [7:0]”. This syntax should be easily understood by anyone familiar with another HDL or programming language.

“cycle”-constructs were stripped of the superfluous named-clause syntax. Naming the code enclosed in a “cycle” construct serves no purpose since referencing it is not needed.

Instantiating functional units is still done as previously shown, but making several instances at once, similar to connections, is not allowed at the moment. Instead a list of clauses can be specified that is used to override the default clauses in switches of the FU. This was not used in the pseudo-implementation of our MIPS processor but it makes written FU descriptions more versatile.

Operations and Pipelines

It was chosen to couple the decoders to the pipeline constructs by referring to the name of an instantiated decoder within parenthesis after the name of the pipeline. This could just as well be changed to the decoders being specified for every operation construct instead. The interaction between decoders and functional units is still not specified in any detail. Connecting pipelines and decoders just seemed to make the most sense.

To unify all instantiation-type statements the “instantiations” of pipelines in pipelines and operations in operations were given the same type of syntax as FU instantiations. An “operation::name” statement now expands the operation “name” in its place.

Operation constructs were given a named clause to hold their content. This fitted into the notion of named clauses being something to activate because these clauses helps define the instructions inside the operation. Instructions are referenced by the clauses that are used to activate them and the first one of every instruction is now an operation clause.

The most obvious change that was made to the operation constructs was replacing the stack-based connection system. Instead of specifying explicitly between which phases a connection is made, an implicit connection system was introduced. The new system uses a more familiar assignment syntax but requires the programmer to be aware of what timings a specific signal will get. Assignments look like normal C++ assignments. “output_o = foo.out_o” in an operation means that the local signal or port “output_o” is assigned the value of the “out_o” output of the “foo” instantiation pipelined from its timing to the current timing. “output_o” is assigned the current timing. Every instantiation gets assigned the timing of the phase where its first input is assigned. The timings of its outputs are
set as the timing of the whole instantiation with added offsets. This offset must be specified for outputs that need one with the \{+offset+\} syntax when declaring the port. In future NoGap\textsuperscript{CL} versions it would be nice to have this generated from the contents of the FU and perhaps some specified dependencies.

Long into the development process it was allowed to use switches and if-else constructs in operations. A switch with multiple choices would act like a multiple-choice activation with the functionality defined directly in the operation. When implementation of the compiler was considered it proved simpler not to allow this and only declare data paths by connecting signals and ports directly in the operations. All functionality had to be moved into other functional units. This had the additional benefit of making the pipeline more easily surveyed when browsing the operations.

### 3.5.2 Decoders

The decoders evolved from only defining the bit-representation of the instructions in its pipeline to also providing advanced control like forwarding.

Since the decoders need special syntax that would not be meaningful in normal functional units they are created in a decoder construct. The decoder construct is similar in appearance to FUs in that it has a ports-preface section and a functionality section. To make the language coherent it was preferred to allow all functionality from leaf FU descriptions in the decoders also. The decoder should in addition be able to reference all connections and instantiated FUs found from the scope of the pipeline and their values in the different pipeline stages if they are pipelined. The decoders also need to be instantiated and can be connected to like FUs. Data paths have to be manually connected with the interface of the instantiated decoder object at this point. It is hard to find a simpler way to connect a single decoder. If multiple decoders are used in a pipeline it would be useful to somehow mark the instruction input signal of the decoders and then use a statement like “issuer::in = instruction\_memory.out\_o;” to assign all marked decoder inputs the instruction memory output at once.

Wrapping the decoders in a shell this similar to FUs had the effect of making them contained visible parts of the design instead of invisible entities interacting with control signals. It also requires less special syntax and gives the language a more unified look and feel.

To define the instructions an “operation\_codes” construct was introduced. The construct is always coupled to an input signal where the instructions are supposed to show up. Inside the construct the instructions of the pipeline assigned to the decoder can be referenced by way of the clauses to activate for them. “ALU::SIGNED::ADD” is supposed to be an instruction from an operation construct with a clause named “ALU”. The “SIGNED” and “ADD” are probably clauses to activate in the ALU of the processor. When implementing the instructions of the MIPS32 processor most instructions could be placed in certain groups of instructions that used the bitrange of the instruction data differently. This spawned the idea of defining operation types where certain ranges of the instruction data could be automatically connected to an output or internal signal.
and other ranges defined as free. The free bits could be used to tell apart the
instructions using that group. This also made it easy to define an assembler rep-
resentation since the arguments of the assembly instruction could be coupled to
named ranges in the operation types.

The problem with the operation type approach appear when many instruc-
tions of a processor does not conform to certain types. It worked reasonably well
for MIPS32 but others might not conform as well. When making a totally new
processor it could give a lot of structure to the decoding and the instructions
though.

The special syntaxes used for advanced control routing was first envisioned as a
genral syntax where, for example, the decoder could check if a certain instruction
was in a certain pipeline stage. Any connection from any pipeline stage could be
fetched and used in switches and if-else constructs. It was never fully formalized
but when using it to pseudo-code the forwarding of MIPS32 it became too complex
very fast. Forwarding was instead moved into the more high-level specialized
“forwarding” construct explained in Section 3.3.2. Regular control structures like
forwarding should get their own constructs but some form of the general language
will probably be needed to cover every contingency.

### 3.6 Comparisons with RTL Languages

Below is listed some of the advantages and disadvantages of using \( \text{NoGap}^{CL} \), in
the form envisioned in Section 3.3, to design a new processor compared to using
RTL languages like verilog. Descriptions of processor HDL tools similar to \( \text{NoGap} \)
and how \( \text{NoGap} \) compares to them is presented in Chapter 4.

#### 3.6.1 Advantages of \( \text{NoGap}^{CL} \)

- Much less micromanagement needed for the details of the processor and the
  pipelining in particular.

- Automatic control signal routing by activating named clauses and specifying
  a decoder.

- Easy to exclude and include instructions in a finished processor design.

- Decoding logic of decoders is automatically created from the operation-types
  with assigned instruction-specific bits and connected to the right control
  signals.

- Advanced control structures like forwarding can be inserted in the connection
  graph from constructs in the decoders and given control logic.

- An assembler can be created for the processor from the command and argu-
  ment specifications in the decoder. A generic assembler template is modified
  with the specific translations for the processor.
3.6.2 Disadvantages of \( \text{NoGap}^{CL} \)

- If a created processor, assembler, or simulator is malfunctioning (due to user error or compiler bugs) it can be harder to debug. Generated code is not as neat and tidy as hand-written code and therefore more difficult to understand.

- If a part of the processor is required to be implemented in a specific way and the bare details must be manipulated, that FU needs to be implemented in verilog also.

- At this point in the development it is yet to be seen how optimized the actual hardware created by the generated verilog code output will be.
Chapter 4

Related Work

Chapter written by Per Karlström (perk@isy.liu.se)

There are quite a few higher level micro architecture and processor construction tools in existence today. All of them offer different support for designing a processor. Many of these tools can construct compilers, assemblers, linkers, simulators, and/or debuggers for the processor being designed. This chapter reviews the current state of the art and describes how NoGap compares to these tools.

4.1 LISA

LISA [15] is one of the major tools for high level descriptions of processors and subsequent compiler and hardware construction. LISA has many appealing features such as compiler, assembler, debugger, profiler, 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 hazard management. LISA, however, assumes a basic architecture of the processor and really novel ASIP processors deviating from this basic architecture cannot be constructed with LISA. The instructions in LISA are restricted to a tree like format where each instruction is composed of sub-instructions which in turn can be divided into sub-instructions. It is up to the designer to assign the binary coding to each sub-module. LISA could in theory, with the correct facets and spawners, fit into the NoGap framework, however NoGapCL assumes less about the architecture from the start than the LISA language. NoGapCL can, unlike LISA, utilize hardware multiplexing to implement a number of instructions with a minimum of hardware.

4.2 EXPRESSION

EXPRESSION [11] is a tool aimed at simulator and compiler retargeting. The processor 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 an architecture component specification, pipeline and data transfer paths, and memory subsystems. A main advantage of EXPRESSION is the ability to easily model various memory subsystems. The FU centric view of the processor, used in EXPRESSION, makes it fairly easy to change to the architecture and to do design space exploration. EXPRESSION is not a tool for generating RTL code, it can only generate simulators and compilers. EXPRESSION has a number of similarities with the NoGap framework like the functional unit centric view. EXPRESSION is, however, not able to generate a real hardware implementation.

4.3 ArchC

ArchC [18] is an ADL for creating processors in SystemC. ArchC 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 generates an assembler and simulator for the processor in question. ArchC is thus a good tool if only an assembler and simulator is needed. ArchC has a co-verification feature where the next refinements of the processor can be verified against the previous refinement. The purpose of ArchC does not seem to be to create a real hardware implementation of the processor described. The SystemC model might in the end be synthesizable but this has to rely on other work done to make SystemC code synthesizable. The designer still has to worry about instruction coding, and the assembler format is defined together with its behavior. To describe a processor in NoGapCL will probably require some more work but NoGap can generate a hardware implementation of the processor described.

4.4 nML

nML [8] is a formalism, or language aiming at describing processors from their instruction set manuals. The description of a processor in nML is based on the information typically found in a programmers manual. nML is a very concise language and, 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 it will save the designer much work. nML is however restricted to processors with a single instruction stream and program counter, thus only fairly standard architectures can be described. Amongst the problems with nML are the difficulties to implement delayed branches, and it falls short when it comes to generating cycle accurate simulators for more complex data paths often found in modern DSP processors. A NoGapCL description will probably be larger but the NoGapCL can encompass a larger set of processor architectures than nML.
4.5 MIMOLA

MIMOLA [17] uses a netlist description of a processor together with algorithms described in a special language to create instructions for the data path that conforms to the algorithm in question. Since MIMOLA generates instructions for the netlist description according to some algorithms, processors for particular algorithms can get optimized. A complete processor can be described if all key parts are put into MIMOLA as algorithms. MIMOLA however 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. NoGap and MIMOLA seems to have some similarities in that functional units and their interconnects are described separately. NoGap supports both multi-cycle instructions and can generate cycle accurate simulators.

4.6 ASIP Meister

ASIP Meister [16] is a processor construction tool. The user can input parameters about the processor like the length of the pipeline, in what pipeline stage the instruction decoding is done, and if register forwarding or delayed branches is to be used. The user also supplies information about what an instruction does and the instruction coding. ASIP Meister can be used to produce a processor, simulator, assembler, and compiler with very little design effort. ASIP Meister also outputs synthesizable code for real hardware implementations of the processor constructed. ASIP Meister is an expert system for RISC architecture construction. This makes it hard to construct a processor deviating from the standard RISC architecture. NoGap and ASIP Meister is similar since they both aim to produce a hardware implementation of the described processor. NoGap differ from ASIP Meister in regards to flexibility. ASIP Meister is restricted to RISC architecture processors while NoGap can be used to describe other architectures as well.

4.7 Mescal

Mescal [12] aims to allow for micro architecture construction not locked to any particular template. Tipi [3], Mescal’s architecture development framework, is a multi-view design entry tool. Tipi uses actors with guarded actions as its atomic elements defining the functionality of leaf modules in a data path. These actors can be connected together to form more complex behavior. Tipi can, for example, be used to construct data paths with the actors as operators. Tipi can then generate all possible connections between all unconnected actor ports in a design. Since all possible instructions are generated for a particular data path it is cumbersome to define a micro architecture from an instruction specification. In this case the designer 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. 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. Mescal and
\textit{NoGap} also shares the concept of having one description for the leaf modules and another description for the data path. However, \textit{NoGap} and Mescal differs in how the architecture is described. In \textit{NoGap} a description of all instructions needed is used and \textit{NoGap} thus guarantees that at least those instructions are implemented. Also, \textit{NoGap} does not make any distinction between a leaf module and a data path module from the outside of a module. It is unclear how pipelines are handled and related to instructions in Tipi and whether or not it is easy to change the number of pipeline stages in a design. This is something \textit{NoGap} handles in a convenient way. Another issue is register forwarding, this is intrinsically supported in the \textit{NoGap}\textsuperscript{CD} but is harder to implement in Tipi.

### 4.8 bluespec

bluespec [1] is a tool for hardware construction aiming at the correct by design approach. bluespec modules are reminiscent of classes in an object oriented language in the respect of having internal state variables and methods to modify and access these variables. The internal state variables in a bluespec module is accessed using guarded atomic actions. bluespec eases the design effort for pipelined architectures and enables design space exploration since the inside of a module can be changed without destroying functionality for other modules. It lacks the concept of operations and instructions which is supported in \textit{NoGap}\textsuperscript{CL}. \textit{NoGap} and bluespec solve different problems. It would be possible to construct a spawner that can generate bluespec code.

### 4.9 SystemC

SystemC [13] is a C++ library for hardware modeling that is gaining ground in the academia and industry. SystemC has a huge advantage in that is has all the expressiveness of C++ but also supports event driven simulation. Thus a designer can model parts at the abstraction level necessary for the task in question. A SystemC model that is synthesizable to hardware is as complicated as any RTL model. \textit{NoGap} and SystemC tries to solve different problems. It would be possible to construct a spawner that generates a SystemC model.

### 4.10 Summary

Many tools offer design support but they usually lock the designer into a specific template design. \textit{NoGap}\textsuperscript{CL} allows for design freedom but still offers support to avoid the micro management needed in normal RTL languages. The tools presented here are just a few of the tools that are available. A good overview of other tools can be found in [10].
Part II

Implementation
Chapter 5

Structure

This chapter is a short description of the general structure and creation of the \texttt{NoGap}^{CL} compiler\footnote{From now on often referred to as compiler only.}. It presents the basic layout of the compiler program and contains short descriptions of the different parts and how they work together. The limitations of the compiler and some notes on how the development proceeded can be found in the last sections. Chapter 6 and Chapter 7 later deals with the specifics of the implementation. Chapter 8 explains what can be done to improve the implemented compiler and the \texttt{NoGap}^{CL}.

5.1 Basics of the \texttt{NoGap}^{CL} Compiler

The compiler is entirely written in C++ except for the lexer and parser that are written in Flex [14] and Bison [5] and compiled to C code. C++ was chosen for its great expressiveness, portability and many supporting tools. Version handling of the project was done with subversion [21].

How the parts of the compiler program interact and how they are arranged in C++ namespaces is shown in Figure 5.1. The compiler is made out of six different main parts:

- The lexical analyzer (lexer), for reading input text and assembling it into tokens (details in Section 6.3).
- The parser, for recognizing grammatical rules as combinations of tokens (details in Section 6.4).
- The abstract syntax tree (AST), for representing the written code in a data structure that allows iteration and modification (details in Section 6.6).
- The symbol table, for storing and retrieving objects named in the code (details in Section 6.5).
• The \texttt{NoGap} connection graph (NCG, \texttt{NoGap}^{CG}), for making and transform-
ing a graph representation of the pipelines of written processors (details in Chapter 7).

• The errors, for having different defined errors that can be thrown and caught throughout the program (details in Section 6.2).

The lexer is called from the parser and returns tokens\(^2\). The parser recognizes the grammatical rules of the language as patterns of tokens. When an include statement is found the parser tries to open the new file and make it the input buffer for the lexer. When sequences of tokens that make statements and constructs are found in the parser, actions are taken that builds the AST representing the code. The Bison parser does not care about checking the validity or functionality of the AST and thus only finds pure syntax errors\(^3\). To separate the Bison parser from the error handling as much as possible all named objects are entered in the symbol table when error-checking the AST. Entering objects into the symbol table produces instant errors if name collisions are detected. All identifying names are entered into their logical scopes and paired with the pointer to the construct (or statement) they represent. If the AST passes its error-checking function, it generates and checks timings for the connections that need it and builds the signal connection graphs with the \texttt{NoGap}^{CG} classes. Signal connection graphs are only built for functional units that contain operations\(^4\) with data paths that need to be merged. The verilog output is made from both the graphs and the AST.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{compiler_structure.png}
\caption{The structure of the compiler. The oval shapes contain the different compiler program parts. The dotted lines mark the boundaries of the namespaces of the program. Arrows represent the most basic flow of information.}
\end{figure}

\(^2\)Or rather the values of the tokens.
\(^3\)Meaning sequences of tokens that do not conform to any grammatical rule.
\(^4\)Constructs that define the instructions of the functional unit (details in Section 6.6.20).
5.2 Limitations

5.1.1 Errors
When any type of error is found, an error object is created and thrown as an exception. This is caught somewhere and saved for later printing in a container known as the error collector (Section 6.2.3), which saves errors catalogued by file and line where they are found. The error collector and other supporting constructs and functions can be found in the “Utils” group. There is a group of basic errors compiled as a separate library that all other errors inherit from. These include the Normal, Warning, Info, and Fatal error classes. When more specific errors are defined they get different severity levels depending on which basic error they inherit from. Fatal errors stop execution and exit the program. The “Parser Errors” group of error classes are specialized basic errors thrown throughout the ::parser namespace.

5.1.2 Output Streams
A couple of output streams are defined in the ::message namespace. These are used throughout the project whenever something should be printed. The targets of the streams are changed easily by defining different variables. The stream definitions are compiled to a dynamically linked library. The beauty of this approach is that the targets of the streams can be changed on-the-fly without re-compiling the entire program.

5.1.3 main Function
The main program creates the AST Top object (Section 6.6.4), the empty symbol table, the error collector and some variables. The parsing function created by Bison links externally to the Top object and uses it to branch the AST from. The symbol table, error collector, and some variables are connected to static variables of the AST and after that the functions in the Top object is the only interface main has to the AST. Some of those functions traverse the AST and get help from the static variables. The static variables also keep track of many other things, like what functional unit and operation construct is being traversed.

5.2 Limitations
Due to the amount of work needed for the implementation there were lots of simplifications made. Some of them are listed below.

- Decoders are not supported at all, which means no automatic instruction decoding, no assembler generation, and no advanced control structures.

- Limited control routing. Control signals are only routed one “instantiation level”, meaning from one instantiated object to the FU where it is used, but not further. The signals are also only routed to input ports of the FU, even if another FU is specified as the decoder.
- No clause dependencies. Clauses in clauses do not save the controls needed to activate the clause they are in as well as their own controls. This means that activating a clause in a clause might not actually make the functionality of it active, unless the “top” clause is also active.

- Verilog output is only made for the contents of the top FUs as put in the connection graphs, which means only for the pipelines of the processors. Verilog generation directly from the AST classes has not been implemented.

- Limited graph representations. Many expressions did not get a connection graph representation during the implementation and assignments to partial signals (meaning not the full bitrange of a signal) is not supported at the time of writing.

5.3 Notes on Development

In the early stages of the implementation there were separate classes representing the actual objects of FUs, connections, instantiations etc. This made a distinction between the actual signal object and the AST signal statement for signal declaration. The object classes were the ones put in the symbol table. We changed this and removed the object classes because they were really just an unnecessary intermediate step. It caused a lot of duplicated information, made that information harder to access, and complicated the program. Now the AST classes are put in the symbol table.
Chapter 6

The NoGap Common Language Compiler

This chapter will describe the functionality of the compiler in detail and how the different parts of it work together. If you don’t feel sufficiently proficient in the programming languages used to understand the text or code examples, then see Section 1.4.

6.1 Message Streams

All messages from the program to the user are channeled into specially defined output streams. They are defined in their own dynamically linked library and can therefore be redefined without re-compiling the entire project. The streams defined are:

- **message::info.** The main information output stream. Usually connected to the normal `std::cout` stream. The information output here tells what files are being parsed, what functions are executing on the AST etc. It is basically saying what is going on in the program at any given time.

- **message::error.** The error output stream. Usually connected to `std::cout` and only outputs the normal and fatal error messages from the error collector.

- **message::warning.** The warnings output stream. Usually connected to `std::cout` and only outputs the warning error messages from the error collector.

- **message::debug.** The debugging output stream. Usually connected to a file-stream called “debug.out”. This stream receives a lot of output that is helpful to debug functions but not interesting to a normal user.
6.2 Errors

6.2.1 Basic Errors Library

The error hierarchy in the project includes a number of basic errors that are compiled as a separate, dynamically linked library in the ::error namespace. They are pure-virtual classes\(^1\), and all other errors inherit from them. The basic error hierarchy is shown in Figure 6.1. The complete error hierarchy of the project at the time of writing can be found in Figure 6.2.

![Inheritance of the basic errors](image.png)

**Figure 6.1. Inheritance of the basic errors**

Base

The *Base* error class basically contains all the functionality of the errors. It has the error message string and the position information composed of the file and line where the error was found. Three functions are used to print this information:

- **printLocation** prints the location of the error in the format “file:line:”.
- **printMessage** prints only the message string.
- **printDebugMessage** prints all the information in the format “file:line: message”.

A function is also present to set the position information. It is used for errors that do not get their position information from the constructor on creation. All errors thrown from the symbol table fall into this category.

Something not yet used in the project is the concept of the error issuer. Every *Base* object stores a copy of the default issuer string defined at the moment. When the AST is issuing errors the string should read “ast”, and when the connection graph is issuing the errors “ncg”. This is done to enable different treatment of the errors in the error collector depending on what part of the program they are from. For now this is not used because only the AST classes output errors at the time of writing.

It is important to preserve the polymorphism when errors are transferred to the error collector. To enable the collector to just store pointers instead of copying the errors to *error::Base* objects (which destroys the polymorphism), we must store pointers to dynamically allocated objects. There is a virtual function in the *Base* class, that is redefined in all lower classes, that creates a dynamically

\(^1\)No actual objects of these classes can be created because of undefined functions.
allocated copy of the object and returns a \texttt{Base} pointer to that object. Pointers to
the actual objects caught cannot be stored because they will always be destroyed
when the catch-construct ends, and the pointer will be rendered useless and faulty.

\textbf{Normal, Fatal, Warning and Info}

The error messages of the error classes are put together solely in the constructors
of the classes. The constructors of these four classes add a severity level indication
at the start of the message and then appends the argument error message. The
severity level indications are:

- \texttt{Normal} adds “error: ” to the error message.
- \texttt{Fatal} adds “fatal error: ” to the error message.
- \texttt{Warning} adds “warning: ” to the error message.
- \texttt{Info} adds “info: ” to the error message.

Having different classes for the different severity levels of an error in the pro-
gram is necessary because they will be caught in different parts of the program.
Fatal errors cause unwinding of the currently executing AST function to a cen-
tralised place where all fatal errors are caught and the program can be exited.
The other severity levels are caught shortly after they have been thrown, mostly
exiting just one function.

The \texttt{MaxErrors} class is also included in this library, but it is not pure-virtual.
It inherits from \texttt{Fatal} and is used when the number of errors found has reached a
certain defined limit and the program should exit in a specific manner.

\subsection{6.2.2 Specialized Errors}

To make usable error classes, the different parts of the program should start their
own error hierarchy from the basic classes in the error library. For now there is only
one continuation of the basic errors; all errors thrown in the \texttt{::parser} namespace
are of the classes defined in \texttt{::parser::error}. For some of these you have to write
the error message manually in the constructor but many are specialized and make
the error message from a couple of string arguments. All the constructors use the
constructor of the class that is directly inherited from. This makes a chain all the
way back to the \texttt{Base} error class.

For errors that are very specific and not often thrown in the code there are
three classes that make general errors, warnings, and infos. They are appropriately
named \texttt{GeneralNormal}, \texttt{GeneralWarning}, and \texttt{GeneralInfo} and inherit di-
rectly from \texttt{Normal}, \texttt{Warning}, and \texttt{Info} respectively. They make it possible to
write your own message and throw an error of the different severity levels. All
warnings except unused connection warnings use this general warning class.

The fatal errors have no general class. In fact, the only fatal errors thrown at
this time use the \texttt{BadProgramming} error class. They are thrown only when the
program does or finds something that is wrong and never should have happened.
Figure 6.2. Inheritance of the error classes
It points out that a programming error has been encountered. These errors are thrown with the location information of where they originate in the source code and not where they are found in the parsed NoGap\(^C_L\) code. This is done by using the \_\_FILE\_\_ and \_\_LINE\_\_ macros predefined in C++.

The ObjectError class is a Normal error and is used for all errors that are connected to a specific object. It takes the name of the object as an extra argument to the constructor and provides a function that returns that name if needed.

The TableError class is the class that all errors used in the symbol table inherit from. It is a typical ObjectError because all these errors concern some object found or not found in the symbol table. Because they are thrown from the symbol table, that does not know what position information to give the errors, they need special treatment in the AST. When any function from the symbol table is used that could throw an error it must be within a try-catch block so that any TableError can be caught and given the right position information.

As is shown in Figure 6.2, there are many predefined error classes for the different types of errors found throughout the AST. The hierarchy is as deep as the AST classes themselves. Since they are all so similar (and just define the error message differently) it is unnecessary to present them all here.

6.2.3 The Error Collector

The error collector is of a class called ErrorCollector in the parser::utils namespace. It saves pointers to error::Base objects arranged by filename and line where the error is found. This is done with a standard map with a string (filename) key and another map with an integer (line number) key as the value part. The second level map with the integer key has the error::Base pointer as value part. The maps are ordered and make it possible to first find the file you want to print errors for and then print them in increasing line number order. There is also a simple list of strings that stores in which order the files are parsed. When the errors are printed this list is used to print errors from different files in the order of error appearance. If the first error is found in “foo.fud”, all errors from that file will be printed first and so on. The polymorphism of the errors is used by the printing function of the error collector to print the messages to different output streams. Normal and fatal errors are printed to the message::error stream, warnings to the message::warning stream, and infos to the message::info stream (see Section 6.1). There are also functions to print all errors to the same argument stream, to delete all entered errors, and to enter errors into the collector.

6.3 Flex Lexer

Flex [14] is used to create a lexical analyzer function. The function is used by the Bison parser (Section 6.4) to retrieve tokens from an input source-file. The lexer includes a file made by the Bison parser where all the tokens declared in the parser are defined. This makes the lexer and parser have a common token description. Patterns and strings are matched with “regular expressions” (regexps)
to recognize the tokens in the lexer. The tokens are returned to the parser that can match them in grammatical rules.

Example 6.1 shows how regexps are coupled to actions that return tokens in Flex code. It makes the string “fu” return the defined value of the token TT_FU and similar for “cycle” and “always”. It then finds identifiers that are any combination of alphanumeric characters and underscore and returns them as the TT_IDENTIFIER token. The return value for TT_IDENTIFIER is a large C-string so all matched text is copied to that return type first. These return types are specified in the Bison parser. Single character tokens like “=” and “/” are returned as their respective ASCII character code.

Example 6.1: Flex Code

```
1 'fu' { return TT_FU; }
2 'cycle' { return TT_CYCLE; }
3 'always' { return TT_ALWAYS; }
4 ( '_' | [[:alnum:]] )+ {
5     strcpy ( ncdval.str , ncdtext );
6     return TT_IDENTIFIER ;
7 }
8 "\*" | '+' | '=' | '-' | '/' {
9     return ncdtext[0];
10 }
```

Everything after “//” on a line (C++-comments) and all whitespaces except linebreaks are ignored. Linebreaks only increase the line counter. When the end of a file is found the lexer uses a function in the parser to see if there are any more buffers to read. If there are, it restores a previous buffer and starts reading the next token.

To enable C-style /**/ comments a special state in the lexer is used. When “/*” is found the lexer enters the exclusive “comment” state, which means that only the lines of the Flex code starting with <comment> will be considered. If linebreaks are found the line counter is increased and if “*/” is found the initial state is entered again. The code for this is provided in Example 6.2.

Example 6.2: Flex C-style Comments

```
%x comment

%%

"/\*" BEGIN(comment);

<comment>"\n" { ncdloc.first_line++; }
<comment>"/\*" BEGIN(INITIAL);
<comment>.
{}
6.4 Bison Parser

Bison [5] is used to construct the parser for the compiler. It is a tool for specifying a context-free grammar that is composed of terminal and non-terminal tokens. Terminal tokens are the tokens returned by the lexer that cannot be divided into other tokens. Non-terminal tokens are created by grammatical rules containing other tokens.

6.4.1 Basic Operation

When the grammatical rules are parsed by Bison the tokens are put on a stack. When a token is added that completes a grammatical rule the tokens composing that rule are reduced (collapsed) to a single non-terminal token representing that rule. When a grammatical rule is reduced there are actions coupled to that reduction. This is what enables us to do things like assemble the AST when parsing the NoGap\textsuperscript{CL} file. If an input file is grammatically correct, tokens will be reduced until only one non-terminal token remains and no more grammatical rules can be applied.

6.4.2 Token Declarations

Terminal tokens has to be declared and, if needed, given an alignment and level of precedence. Precedence defines which tokens are to be reduced first, when there would otherwise be an ambiguity. “*” and “/” have higher precedence than “+” and “-”. The alignment can be defined as left or right and determines which side of the token should be reduced first. A Bison parser with only right-aligned tokens therefore creates a recursive machine that enters all tokens on the stack before being able to reduce a single rule. That is certainly not wanted since the stack depth always is limited. Some tokens, like “=”, has to be right-aligned but otherwise it is preferred to use left-alignment. The right-aligned tokens in NoGap\textsuperscript{CL} are “=”, “.”, “[”, “]”, “,” and “->”. The names of the terminal tokens are automatically assigned integers that are defined in a file that the Flex lexer includes.

Non-terminal tokens\footnote{Also called simply non-terminals.} must be declared with name and return type. The return type is the type of object returned when reducing a rule for the non-terminal. The returned object can be used in the action of a grammatical rule using that non-terminal. Most of the return types in our parser are pointers to AST objects. This makes it possible to connect objects of the AST classes and build the tree structure.

Example 6.3 shows Bison code for declaring “fu”, “cycle”, and “always” as normal tokens. “=” gets right-alignment, and “+” and “-” get left-alignment with lower precedence than “*” and “/”. \texttt{statement} and \texttt{expr} are non-terminals with types \texttt{parser::ast::Statement*} (Section 6.6.5) and \texttt{parser::ast::Expression*} (Section 6.6.6) respectively. The “%union”-construct is where all return types are defined.
6.4.3 Grammatical Rules and Actions

The patterns of tokens to match for different non-terminals are the grammatical rules of the language and must be context-free. Each grammatical rule has an action with C++ code that is executed when that particular rule is reduced. All of the AST objects created and returned by the rules must be dynamically allocated since we do not want them destroyed at the end of the action. The created AST objects are also assigned location information composed of the file and line where they were found in the NoGap CL source file.

Example 6.4 shows the Bison grammar for signal declarations in NoGap CL. The signal_stmt non-terminal returns a pointer to a parser::ast::Signal (Section 6.6.13) object. The constructor of Signal receives the name of the signal from the TT_IDENTIFIER terminal token that has a C-string value. The signal also gets a pointer to a Size object (Section 6.6.12 and Section 6.6.13) that determines the bitrange and number of instances of the signal. The non-terminal size supplies this Size object (or an empty pointer). The bitrange and instances of size are specified with expr non-terminals which are normal expressions (Section 6.6.6) with constants, operators (not including “=”), and identifiers.
instances : '{' ':' expr ':' '}'
{
    $$ = $3;
}

; range : '[ ' expr ':' expr ' ]'
{
    $$ = new std::pair<parser::ast::Expression*,
parser::ast::Expression*>($2, $4);
}

| ' [ ' expr ' ]'
{
    $$ = new std::pair<parser::ast::Expression*,
parser::ast::Expression*>($2, 0);
}

; size : /* empty */
{
    $$ = 0;
} | instances range
{
    $$ = new parser::ast::Size($1, $2->first, $2->second);
    $$->setPos(parseFile, @1.first_line);
}
| range
{
    parser::ast::Expression* foo = 0;
    $$ = new parser::ast::Size(foo, $1->first, $1->second);
    $$->setPos(parseFile, @1.first_line);
}
| instances
{
    $$ = new parser::ast::Size($1, 0, 0);
    $$->setPos(parseFile, @1.first_line);
}

6.4.4 Errors

The second rule for the signal_stmt non-terminal in Example 6.4 includes the predefined error non-terminal. It is matched whenever something does not match anything. This extra rule means that if someone does not type a valid size for the signal the input can still be matched to some extent, and a Signal object without a Size pointer can be made. Otherwise the error would have made the parser unwind the parsed stack and throw away the tokens for the signal declaration. The parser would then continue unwinding until it matches an error non-terminal or the whole stack has been unwound and the parsing has failed.

The error messages for all errors found in the parser are automatically generated by Bison. They are not very specific and might not make much sense unless
you have been developing this program. This is an area that should be improved.
The messages are stored in :error::GeneralNormal objects (Section 6.2.2) and
put in the error collector.

6.4.5 Special Grammatical Rules

When an “include”-statement is found, the programmer wants the parsing to con-
tinue in another file. The current input buffer to the lexer is put on a stack and
the new file is opened and used as input.

Many of the rules assemble some kind of list. The elements rule (see grammar
in Section 6.6.2) is somewhat special in that it can be assembling several lists at
a time and is used frequently. It collects AST Element objects, which can be
everything from full functional units to simple statements and expressions. Until
parsing is finished the top list of FUs is being assembled but many lower constructs
contain lists of elements as well. To enable this single rule to make multiple lists,
a stack is used to push the current working-list when a new list has to be started.

6.5 Symbol Table

A symbol table is a data structure used by compilers to store names used to identify
objects in the code. The names are coupled to information about the objects that
can be retrieved.

The symbol table of the NoGapCL compiler uses a scope tree structure (Sec-
tion 6.5.1) where all defined names are saved in tables contained in the nodes
of the tree and coupled to the pointer to the AST object they represent. The
branching scope tree is only searchable in one direction, towards the “ROOT”
scope, since every scope node only has a pointer to the preceding scope where
it was created. This makes it impossible to find any objects not defined in the
logical “C++-type” scope trail from any given node when using the implemented
standard searching functions (Section 6.5.2). Example 6.5 presents the symbol
table of a simple functional unit.

--- Example 6.5: From NoGapCL Code to a Symbol Table ---

This example shows the symbol table that is constructed for the simple functional
unit description in the code below. The fully filled symbol table is shown in
Figure 6.3. Notice that the name “temp” is used for different objects in the
switch choices. The Dot representation source file is generated completely from
the printDot function in the symbol table.

```plaintext
1  fu  foobar (    
   input [4:0] addr_i;
3   input [31:0] dat_i;
5   output [31:0] out_o;
7   input control_ci;
)
```
6.5 Symbol Table

```
9 {  
  signal [31:0] reg;
11  
  cycle
13 {  
    switch(control_ci)
15 {  
      choice: %WRITE
17 {  
        signal [31:0] temp;
19      temp = dat_i;
21      reg[addr_i:0] = temp;
23      default: %IDLE
25 {  
        constant temp = 32b0;
27      reg = temp;
29    }
31  }
33  out_o = reg;
35 }
```

**Figure 6.3.** Dot representation of a symbol table. The oval shapes are the scope nodes representing the scopes of the program. The squares are the tables of objects contained in the scope nodes. Arrows point to the tables of scope nodes and their sub-scopes.
6.5.1 Scope Tree Structure

The scope tree is almost a totally open structure. It is made with scope nodes of a struct called Scope. All member variables of the scope nodes except for its name are declared public. The contents of a scope node is a table in a standard STL map for storing identifiers, a pointer to the previous scope, a pointer to the ast::Base (Section 6.6.1) object the scope belongs to, and a string with its name. The map contains (string-ast::Base*)-pairs for easy lookup of what object a name refers to. These struct objects are dumb and all intelligent functionality (except Dot printing) of the symbol table is provided through the SymTab class that builds the scope tree and accesses the information in the scope nodes.

6.5.2 SymTab Interface

The SymTab class keeps track of the current and previous scope and provides functions for changing scopes and searching the scopes for named objects in different ways. It also has a list of pointers to all scopes it has created for easy deletion of the scopes in the destructor. The functions for changing the current scope are:

- **setScope**. Simply changes the current scope to an argument scope. All functions changing the current scope also updates the previous scope pointer.
- **prevScope**. Sets the previous scope as the current scope.
- **traceScope**. Sets the next logical scope upwards in the scope tree as the current scope. Uses the previous scope pointer from the current scope.
- **pushScope**. Pushes the current scope on a stack for later recovery.
- **popScope**. Restores a scope from the stack.

Pushing and popping scopes on the stack is practical when you need to make temporary scope changes and/or want to make sure a particular scope is returned to. The enterObject function is the one used for actually putting objects in the symbol table. It takes a string and an ast::Base pointer as arguments and pairs them together. The pair is then entered into the table of the current scope. Two functions are used for making new scopes. One that takes a string and an ast::Base pointer as arguments, enterObjectScope, and one that takes a single string argument, enterScope. enterObjectScope makes a scope with the name of the argument string and sets that it belongs to the argument ast::Base object. enterScope instead uses the object the current scope belongs to for the new scope.

The pointers in the scopes to the objects that created them are used almost completely from the operateCheck function (Section 6.6.23) to see if statements or constructs are found in illogical places. The scopeObjectIs template function is used to find out if the current scope belongs to an AST object of a specific type.

SymTab also has a printDot function for making a Dot graph representation of the symbol table. It functions very similar to the printDot function of the AST classes explained in Section 6.6.27. Since there is no way to traverse the
entire scope tree from one starting point, a scope pointer deletion list (including all created scopes, and later used to delete them) is used to start a printing function in every scope. The scope nodes get a connection to a Dot struct (which is basically a table) where all defined names in its object table are printed. Figure 6.3 is automatically generated with this function.

Scope Searching Functions

All the searching functions are based on the private \texttt{findObject} function. It traverses the scopes towards the “ROOT” node from an argument node searching the tables of the scope nodes for an argument string. If a match is found, the object is returned paired with the scope it was found in. If no match is found, an error is thrown. The code of the function is found in Example 6.6 below.

---

**Example 6.6: The \texttt{findObject} Function**

```cpp
std::pair<ast::Base*, const Scope*> SymTab::findObject(const std::string& s, const Scope* arg_scope) const
{
    typedef Scope::table_t::const_iterator I;
    I iter;

    const Scope* search_pointer = arg_scope;

    // Search scopes in order
    while(true)
    {
        // Error if whole scope tree trail has been searched.
        if(!search_pointer) throw error::Undefined(s);

        // Try to find string in scope table.
        iter = search_pointer->table.find(s);

        // String found?
        if(iter != search_pointer->table.end())
        {
            return std::make_pair(iter->second, search_pointer);
        }

        // Set next logical scope
        search_pointer = search_pointer->previousScope;
    }
}
```

---

The \texttt{findInScope} template function is also private and refines the \texttt{findObject} function with run-time type checking to see if the found object is of the template argument type. Only the pointer to the object is returned. If the object is of the wrong type, an error is thrown.

Only three template functions are available as the public searching interface:
• **find.** Simply uses `findInScope` from the current scope to find an object of the template argument type.

• **extFind.** The extended find function. Instead of throwing an error when an object of wrong type is found, it searches on and quits only when there are no more scopes to search or an object of right type is found. It uses the `findObject` function multiple times and does run-time type checking to see if an object of right type is found.

• **forwFind.** The forward find function. It takes an additional argument list of names with instantiated objects to find. These names are found with the `findInScope` function starting with the current scope, but for each found instantiation its respective scope is used for the next search. When all the instantiations have been found, the last scope is used to find the actual object we are looking for. This function is used when identifiers with “.”-operators are found, like `alu.signal[31:0]`. First the `alu` functional unit instantiation must be found and then a connection named signal in the scope of that FU.

**Checking Connection Usage**

When the `operateCheck` function has been completed, all scopes of the symbol table are searched to find connections that have not been used in any way in the project. Each signal and port has a boolean variable that is set true if an identifier has referred to that connection. Instantiated functional units have separate variables for all the ports of that specific instantiation. This enables the internals of the FU to be checked separate from the interfaces of the instantiations of the same FU. All scopes are simply searched for connections and instantiations and the classes usage-checking functions are invoked to produce warnings to put in the error collector.

### 6.6 Abstract Syntax Tree

An abstract syntax tree (AST) is often used by compilers as an intermediate step in the compilation process. It is a data structure designed to represent written code. It allows functions to easily do multiple passes on the code without having to parse text multiple times. The structure can also be modified at will before using it to generate a desired output. Both of these traits are utilized in the NoGap CL compiler.

Our AST is almost a direct representation of the written code. It is composed of a collection of classes where each object can have pointers to other objects forming a huge tree structure. The tree can only be traversed in one direction from any arbitrary starting point. The Bison parser creates objects of the AST classes with dynamically allocated memory and link them with pointers. We begin with a simple example of how a NoGap CL AST can look, extending Example 6.5 from Section 6.5.
Example 6.7: From NoGap^{CL} code to an AST

This example shows how NoGap^{CL} code is translated into an abstract syntax tree representation. The code is the same as in Example 6.5 but it is printed again for convenience. The simple functional unit description produces the AST in Figure 6.4. The source text for the Dot figure is generated completely by the \texttt{printDot} function in the AST classes (Section 6.6.27).

```plaintext
fu fooobar {
    input [4:0] addr_i;
    input [31:0] dat_i;
    output [31:0] out_o;
    input control_ci;
}

signal [31:0] reg;

cycle
    switch(control_ci)
        choice: WRITE
            signal [31:0] temp;
            temp = dat_i;
            reg[addr_i:0] = temp;
        default: IDLE
            constant temp = 32b0;
            reg = temp;
    }

out_o = reg;
}
```

There are four main functions operating on the AST classes. They are \texttt{operateCheck}, \texttt{timingGen}, \texttt{buildGraph}, and \texttt{genVerilog} and are used on the AST in this order. If errors are found in one function the execution of the main function shouldn’t progress into the next one. Errors found in \texttt{operateCheck} might, for example, cause the program to crash in \texttt{buildGraph} otherwise. The first three functions all have an argument called \texttt{side} that is used to know what side of an assignment is being traversed. The variable is of an enumeration type with \texttt{RIGHT} and \texttt{LEFT} defined. All the main functions are described in further detail later in this chapter.

\texttt{operateCheck} (Section 6.6.23) does syntax/semantic checking and constant folding on the AST classes. It takes a \texttt{side} argument, as previously explained,
Figure 6.4. Dot representation of an AST. The different coloured shapes are the actual AST class objects and the arrows represent the pointers contained in the classes that connect the tree-structure.
and returns a bool telling the function caller to do constant folding or not.

*timingGen* (Section 6.6.24) assigns and checks timings for signal and port connections. The function returns nothing but takes a *side* argument.

*buildGraph* (Section 6.6.25) builds the connection graph description for FUs with operations. It takes a *side* argument and returns a graph vertex descriptor. The vertex descriptor is in essence a pointer to a graph vertex (node) in the \(\text{NoGap}^{CG}\). If no other vertex descriptor can be returned there is a “void” node in every graph that can be returned as a default.

*genVerilog* (Section 6.6.26) generates a verilog description of the FUs described in the AST. It takes an output stream argument and returns an output stream. This function is currently implemented as far as using the verilog generation in the \(\text{NoGap}^{CG}\) and specifying external verilog replacement files. It concerns only a few of the AST classes.

There is a function called *printDot* (see Section 6.6.27) that is implemented for all classes inheriting from the *Base* AST class. It prints a Dot graph description of the AST like the one in Figure 6.4.

The AST classes have an elaborate inheritance and it is helpful to have a picture as a reference while reading about the different classes. Figure 6.5 contains a graph depicting the complete inheritance of the classes.

### 6.6.1 Base

**Inheritance**

*Base* does not inherit from any other class and serves as the base-class for all the AST classes.

**Description**

The *Base* class is pure-virtual and keeps all the common denominators of the AST objects. The four main functions are declared here only as pure-virtual functions securing their inheritance to all AST classes. Any class inheriting from *Base* that should be made into an actual object must therefore have defined these four functions either directly or by further inheritance. The class also keeps a unique identifying string for objects of all derived classes that, among other things, is used for Dot printing.

**The Static AST Variables**

Four important static objects are available to all classes from the *Base* object. They are *parserInfo*, *dotInfo*, *traversingInfo*, and *checkInfo*.

*parserInfo* has pointers to the symbol table and error collector used and keeps all other error information. All the member variables have previously been used by the Bison parser but are copied to the *parserInfo* object when the AST has been built. It also has status functions that prints and empties the error collector and makes the symbol table check for unused connections.
Figure 6.5. Inheritance of the AST classes
**Abstract Syntax Tree**

*dotInfo* helps the printDot function. This static object only keeps two integers. One for counting how many AST elements have been printed and one for keeping the detail level of the current printing run. This enables the printDot function to only have one output stream argument and look like a normal function printing to a stream.

*traversingInfo* is very important for keeping track of the context of what is being traversed when functions are being executed on the AST. Among other things it keeps track of what functional unit is being traversed. For the *buildGraph* function it has a pointer to the current graph being built, its input and output port units, and a list where completed graphs are saved.

*checkInfo* helps the *operateCheck* function with constant folding and with special check modes.

**Grammar**

There is no grammar in the parser that corresponds to the *Base* AST class.

### 6.6.2 Element

**Inheritance**

Inherits from the *Base* (Section 6.6.1) class.

**Description**

The base-class for all classes representing written code elements. It refines the *Base* class with position information keeping the file and line where the *Element* is found in the parsed code. The *operateCheck* function is still pure-virtual so no real *Element* objects can be created. *timingGen* and *buildGraph* are defined as non-modifying functions that can be used by derived classes not needing their own *timingGen* or *buildGraph* function.

**Grammar**

\[
\langle\text{element}\rangle \rightarrow \langle\text{statement}\rangle; \\
| \langle\text{include}\_\text{stmt}\rangle; \\
| \langle\text{construct}\rangle \\
\langle\text{include}\_\text{stmt}\rangle \rightarrow \text{include} \text{ STRING\_CONST}
\]

Elements can be statements, constructs, or the special “include”-statement. The “include”-statement is a statement without a counterpart in the AST classes since its functionality is taken care of directly in the parser. The parser switches input buffers for the lexer in the action coupled to the “include”-statement element. STRING\_CONST is a normal string within quotation marks.
6.6.3 ElementList

Inheritance

Inherits from the Base (Section 6.6.1) class.

Description

ElementList helps provide an easy continuation of the AST every time a class needs the possibility to store an indefinite number of AST elements. It contains a list of Element objects and implements the four main traversing functions by simply invoking the corresponding functions for each of the elements in the order of insertion.

Grammar

\[
\langle\text{elements}\rangle ::= \\
| \langle\text{elements}\rangle \langle\text{element}\rangle
\]

The empty grammar rule makes a new ElementList in its action and the second rule adds an element to an ElementList.

6.6.4 Top

Inheritance

Inherits from the Base (Section 6.6.1) class.

Description

This is always the highest node in the tree and there is always only one per AST. It acts as a handle and sole interface to the rest of the AST. It contains an ElementList where every entry will be a different functional unit and four boolean values keeping the information of whether or not the four main functions finished without errors. The implementations of the four main functions for Top basically just use the corresponding ones in ElementList but include some information printing. After each one has finished, special status-printing functions in parserInfo are used to print all errors and warnings and empty the error collector. If no errors have been found, the boolean “ok”-flag for the function is set true, otherwise no more functions will be executed on the AST and the program will exit.

Grammar

\[
\langle\text{top}\rangle ::= \langle\text{elements}\rangle
\]

The grammar rule sets the ElementList from the <elements> non-terminal as the ElementList of the Top object. When this is done everything will have been parsed and the grammar rules been used to reduce the whole input to a single <top> non-terminal.
6.6 Abstract Syntax Tree

6.6.5 Statement and Construct

These two classes are empty and do not add any extra functionality. They inherit from Element but are only present to divide the lower AST Element-classes into two logical groups. Statements can be written in \texttt{NoGap CL} as a single line ended with a semicolon. Constructs usually span multiple lines and do not end with a semicolon. Another distinction between Statements and Constructs is that Constructs are more of collections of several Statements and/or Constructs that define something together. Statements perform one or a couple of specific tasks while Constructs depend more on their constituents to define their meaning.

Grammar

\begin{align*}
\langle \text{statement} \rangle &::= \langle \text{variable assign statement} \rangle \\
& \mid \langle \text{port stmt} \rangle \\
& \mid \langle \text{expr} \rangle \\
& \mid \langle \text{signal statement} \rangle \\
& \mid \langle \text{phase stmt} \rangle \\
& \ldots \\
\langle \text{construct} \rangle &::= \langle \text{clause cons} \rangle \\
& \mid \langle \text{fu cons} \rangle \\
& \mid \langle \text{cycle cons} \rangle \\
& \mid \langle \text{always cons} \rangle \\
& \mid \langle \text{switch cons} \rangle \\
& \ldots
\end{align*}

There are many more statements and constructs that have been left out of the grammars presented above for convenience.

6.6.6 Expression

Inheritance

Inherits from the Statement (Section 6.6.5) class.

Description

Expressions are combinations of binary and unary operations, constant numbers, identifiers, concatenations, and replications. Assignment is included as an expression in that it is part of the expression classes of the AST, but it has restrictions and receives a lot of special treatment based on the syntax of the \texttt{NoGap CL}. For now only single identifiers can be assigned. Initialization of constants is a special case and is treated as a single statement. That means it does not produce an expression. All the classes used to build expressions inherit from the pure-virtual Expression class. When building the graph description of expressions, modules called atomic modules from the \texttt{NoGap CG} are used (see Section 7.2.3).

Every binary and unary operation has a function that is used to calculate the value of the operation if both operands are constant (\texttt{ConstNum} objects,
Section 6.6.7). The functions are used for constant folding to replace the binary operation with a single ConstNum object. The functions are defined here to only return 0. The assignment function says a bad assignment was found. The assignment operation function is redefined only for identifiers and checks that the sizes of the two operands match for assignment (see Section 6.6.12). In the ConstNum class the other functions are redefined to actually calculate the right values and trigger constant folding.

The virtual compare function is introduced in Expression. It is used by the assignment operator function of identifiers (Identifier, Section 6.6.12) when they are assigned. It helps check and issue a warning if the size\(^3\) of the expression on the right side of an assignment does not match that of the assigned identifier. It gets the bitrange and number of instances of the identifier to compare with as input arguments and returns the calculated bitrange of the expression.

Grammar

\[
\langle \text{expr} \rangle ::= \langle \text{const\_num} \rangle \\
\quad | \langle \text{identifier} \rangle \\
\quad | \langle \text{concatenation} \rangle \\
\quad | \langle \text{identifier} \rangle \\
\quad | \langle \text{expr} \rangle + \langle \text{expr} \rangle \\
\quad | \langle \text{expr} \rangle - \langle \text{expr} \rangle \\
\quad | \langle \text{expr} \rangle \ast \langle \text{expr} \rangle \\
\quad | \ldots
\]

The binary and unary operations are so many and similar that it is unnecessary to include all the grammar rules for expressions here.

### 6.6.7 ConstNum

**Inheritance**

Inherits from the Expression (Section 6.6.6) class.

**Description**

Constant numbers are represented by the ConstNum class and divided into 10-base integer constants (IntNum class) and binary constants (BinNum class). The IntNum and BinNum classes simply initialize the basic ConstNum class differently. The value of a constant number is stored in the helping class called \texttt{parser::utils::Number}. It takes care of storing the value as an unlimited precision integer (by using the GNU Multiple Precision (GMP) library [23]).

The binary and unary operation functions are redefined for ConstNum to calculate the value of the operation performed on the argument expression (if it is a ConstNum) and this object. They then trigger constant folding. The ncg::AtomicConstModule (Section 7.2.5) is used to represent the constant numbers in the connection graph.

\(^3\)Referring to bitrange and number of instances.
Grammar

\( \langle \text{const\_num} \rangle \rightarrow \text{NUM} \\
\mid \text{BIN\_NUM} \)

NUM is a normal 10-base integer number, and BIN\_NUM is a binary number written like “5b10011”. The first rule creates an \textit{IntNum} and the second a \textit{BinNum}.

### 6.6.8 Concatenation

**Inheritance**

Inherits from the \textit{Expression} (Section 6.6.6) class.

**Description**

The concept of concatenation is to fuse several different signals and constants into one signal in a specific order. It contains a list of expressions to concatenate. The AST-traversing functions simply invoke their counterparts in all the expressions in the list. The \textit{compare} function branches \textit{compare} to all expressions in the list and add all the returned bitranges to form the bitrange of the entire concatenation. That bitrange is used to check against the argument bitrange.

An \textit{ncg::AtomicMergeModule} (Section 7.2.5), which takes input signals from all the listed expressions and outputs a single signal, is used to represent concatenations in the connection graph.

Grammar

\( \langle \text{concatenation} \rangle \rightarrow \& \{ \langle \text{concat\_list} \rangle \} \& \\
\langle \text{concat\_list} \rangle \rightarrow \\
\mid \langle \text{expr} \rangle \\
\mid \langle \text{replication} \rangle \\
\mid \langle \text{concat\_list} \rangle , \langle \text{expr} \rangle \\
\mid \langle \text{concat\_list} \rangle , \langle \text{replication} \rangle \)

Replications (Section 6.6.9) had to be left out of the \langle \text{expr} \rangle non-terminal and only allowed in concatenations to escape ambiguities in the language.

### 6.6.9 Replication

**Inheritance**

Inherits from the \textit{Expression} (Section 6.6.6) class.

**Description**

Replication is very similar to concatenation but instead there is a single signal or constant that is concatenated a constant number of times with itself. It contains a pointer to an expression and a pointer to a \textit{ConstNum} saying how many times
to replicate the expression. The compare function finds the bitrange of the expression and multiplies it with the integer value of the constant number to find the bitrange of the entire replication expression. The graph counterpart of replications is similar to concatenations but it uses the same input signal to all the inputs of the \texttt{ncg::AtomicMergeModule}.

\textbf{Grammar}

\[
\langle\text{replication}\rangle ::= \&\{\langle\text{const\_num}\rangle \{\langle\text{expr}\rangle\}\}\&
\]

\textbf{6.6.10 BinOp and UnaryOp}

\textbf{Inheritance}

They both inherit from the \texttt{Expression} (Section 6.6.6) class.

\textbf{Description}

The \texttt{BinOp} and \texttt{UnaryOp} pure-virtual classes represent the binary and unary operations in expressions. They have pointers to their operand expressions and a function pointer to the constant folding operator function used for the particular operation. They also have a string representation for the operations in the function pointer. For example “&&” for the boolean AND operation.

The binary and unary operations derived from \texttt{BinOp} and \texttt{UnaryOp} (like \texttt{Add}, \texttt{Sub}, \texttt{Mult}, and \texttt{Not}) need to define the function pointer and its string representation to be a fully functional operation. They do not have to redefine any of the functions of the \texttt{BinOp} or \texttt{UnaryOp} classes except compare to work. The \texttt{Assign} binary operator class is the exception because it needs to redefine the three main functions with a \texttt{side} argument. The \texttt{Assign} class is explained further in Section 6.6.11.

\texttt{compare} has to be individualised for every binary operation since most of them are different. For instance, \texttt{Add} needs to compare with the greatest of the operand bitranges plus one, and \texttt{Mult} with the sum of the operand bitranges.

The graph representation of the operation classes is at the time of writing pretty short-handed. There are only modules representing the binary add and multiply operations (Section 7.2.4).

\textbf{Grammar}

The grammars for the operations are the different rules of the \texttt{<expr>} non-terminal in \texttt{Expression} except the first three.

\textbf{6.6.11 Assign}

\textbf{Inheritance}

 Inherits from the \texttt{BinOp} (Section 6.6.10) class.
Description

Assign is forced to redefine the operateCheck, timingGen, and buildGraph functions to use different side arguments to the left and right operand expressions.

The graph description of assignments is a single connection between whatever vertex descriptors are returned from the two operand expressions. For the other binary operations a module with two inputs and one output has to be inserted and connected to.

Grammar

\[
\text{variable_assign_statement} ::= \text{identifier} = \text{expr}
\]

Unlike the other binary operations, the assignments have their very own statement non-terminal. This provides better control over the assignments as it leaves them out of the <expr> non-terminal, which prevents them from being nested and used in illogical places in expressions.

6.6.12 Identifier

Inheritance

Inherits from the Expression (Section 6.6.6) class.

Description

When a named object (like a signal) should be used it must be referenced by an identifier. Identifiers basically define search paths in the symbol table with a collection of strings. One list of strings and one main string. The main string is the name of the object to look for and the list says where to look by naming instantiated objects with scope. The language is made so the list always will be empty (meaning search in the current scope) or with just one entry (look in this instantiated object). Two entries in the list would mean to look in an instantiated object in an instantiated object and might be written as “inst1.inst2.signal [31:0]”. This is not allowed.

To know what parts of a connection are addressed the identifiers have a Size construct that has pointers to three expressions. The expressions represent the top and bottom bitrange delimiters and the number of instances to address. The connections use the same Size construct for initialization of signals and ports, but in that case all the expressions must be constant. Here all types of expressions are valid since it is the value of the signal they produce that is important. A check makes sure that if any of the expressions in this Size are constant they are within the limits of the connection referred to by the identifier.

When building the graph description the identifiers just need to return the vertex descriptor to the graph node of the connection referenced. If the identifier has gotten a specified size, a special “tap” or “funnel”-module (depending on if the identifier is read or written) must be inserted to extract or insert the right bits of the connection node. Only the “tap”-module (ncg::AtomicTapModule,
Section 7.2.5 used for reading has been implemented so far. The concept of connection instances has not yet been implemented in the \texttt{NoGap} classes.

If the identifier is written as a direct connection with the \texttt{"->"}-operator, all timing checks for the referenced connection are bypassed and it is connected with direct edges (\texttt{ncg::DirectArc}, Section 7.2.12) in the connection graph.

**Grammar**

\[
\begin{align*}
\langle \text{identifier} \rangle & \rightarrow \text{IDENTIFIER} \ \langle \text{size} \rangle \\
& \quad | \ \text{IDENTIFIER} \ . \ \langle \text{identifier} \rangle \\
& \quad | \ \text{IDENTIFIER} \ -\rightarrow \ \langle \text{identifier} \rangle \\
\langle \text{size} \rangle & \rightarrow \\
& \quad | \ \langle \text{instances} \rangle \ \langle \text{range} \rangle \\
& \quad | \ \langle \text{range} \rangle \\
& \quad | \ \langle \text{instances} \rangle \\
\langle \text{instances} \rangle & \rightarrow \{ : \ \langle \text{expr} \rangle : \} \\
\langle \text{range} \rangle & \rightarrow \left[ \ \langle \text{expr} \rangle : \langle \text{expr} \rangle \ \right] \\
& \quad | \ \left[ \ \langle \text{expr} \rangle \ \right]
\end{align*}
\]

The first rule for \texttt{<identifier>} creates a new \texttt{Identifier}. The second rule adds a parent instantiation name to an \texttt{Identifier}. The third rule adds a parent instantiation name and sets the identifier as a direct connection.

### 6.6.13 Connection

**Inheritance**

Inherits from the \texttt{Statement} (Section 6.6.5) class.

**Description**

Signals and ports in the functional units and the decoders are the only types of connections found in \texttt{NoGap}, and they are very similar. The most important features of a connection are name, size, timing, and offset. Timing and offset are stored as integers. Two additional bools are provided to tell if the connection is used at all and if it is used as a control signal. A connection can either be used as a control signal or as a normal signal, never both. Connections used to switch on in the \texttt{Switch} (Section 6.6.22) constructs are currently the only control signals.

The size of the connection is stored in the same type of \texttt{Size} construct as for identifiers. For a connection declaration the expressions in the \texttt{Size} object must be constant, and thus using only \texttt{ConstNum} objects or identifiers referring to defined constants.

Timings are assigned to connections to be able to insert pipeline registers where needed in the built connection graphs. The timings are checked so the programmer does not use a connection before it is entered into the pipeline and carries valid data. If timings and offsets are poorly defined the processor will behave differently than expected since the wrong values of connections will be pipelined and read.
The timing values stored in the actual connections are only modified outside of *Operation* constructs (Section 6.6.20). If a connection is assigned outside of operations it is stored in the connection otherwise it is stored in the *Operation* construct. This is because different instructions can have different timings for the same connection. If a connection has been assigned outside of operations, it receives timing 0 and cannot be assigned inside of operations. A connection first assigned in operations cannot be assigned again outside of operations. There are functions for reading and setting the timing as well as checking if the timing has been set.

Offset is only used for output ports of instantiations. It is an integer value keeping the timing offset in clockcycles from the assignment of the inputs of an instantiation to when valid data appears on the port. Offset is specified in the declaration of the ports and cannot be altered after the connection has been created. A specified offset is directly added to the timing of the output port of the instantiation it was specified for. See Section 6.6.24 for more info on how timings are handled.

The graph representation of ports are the output and input ports added to the modules (Section 7.2.2) of the FUs used for instantiations and the global ports (Section 7.2.9) added to the graph of the FU. Identifiers referring to a connection fetch one of these vertex descriptors. The choice depends on if the port is referenced from inside its FU or if it is referenced in an instantiation. The graph representation of signals is not yet finalized but should be a simple module with one input and one output port.

**Signals vs Ports**

The biggest differences between signals and ports are that ports can only be declared in the “ports preface” section of a FU and that they are specified as input or output only. Signals can on the other hand be assigned in cycle constructs and turned into registers with memory functionality. Both of the classes naturally have access to the offset variable in the *Connection* class, but ports are the only connections that the grammar permits it to be defined for. To accommodate the differences the *Signal* class has a type enumeration with UNDEFINED, NORMAL, and REGISTER defined. It also has a pointer to the cycle construct where it is assigned if it is a register. *Port* has a boolean value saying if it is an output or input port. Because of their differences, ports and signals are treated differently in the *operateCheck* function of identifiers. For instance, the input or output status of ports must be considered to see if it is allowed to read or write them and register signals cannot be assigned outside their cycle construct.

**Grammar**

\[
\begin{align*}
\langle \text{port stmt} \rangle :& : \langle \text{direction} \rangle \ \langle \text{offset} \rangle \ \langle \text{size} \rangle \ \text{IDENTIFIER} \\
& \ | \ \langle \text{direction} \rangle \ \langle \text{size} \rangle \ \text{IDENTIFIER} \\
\langle \text{signal stmt} \rangle :& : \langle \text{signal} \rangle \ \langle \text{size} \rangle \ \text{IDENTIFIER}
\end{align*}
\]

*Footnote:

4The offset only makes sense for ports in instantiated FUs.
\langle \text{direction} \rangle ::= \text{input} \mid \text{output} \\
\langle \text{offset} \rangle ::= \{ + \langle \text{const\_num} \rangle + \}

<\text{size}> is as previously defined for \textbf{Identifier} (Section 6.6.12).

\textbf{6.6.14} \textit{Fu}

\textbf{Inheritance}

Inherits from the \textbf{Construct} (Section 6.6.5) class.

\textbf{Description}

Functional units (FUs) are the constructs that define the entire functionality of a \texttt{NoGap}\textsuperscript{CL} project. They have specified interfaces of input and output ports and can be instantiated in other functional units to create more complex designs. The \textit{Fu} class inherits from \textit{PortObject}, which contains everything common to classes that all have ports, scope, a graph module, and can be instantiated. No other class other than \textit{Fu} used in the language today fall into this category but shells for the planned \textit{Decoder} and \textit{Interface} classes exist. There is not much to say about \textit{PortObject} other than that it has a pointer to a list of ports, a graph module, and functions to access these member variables.

\textit{Fu} has an \textbf{ElementList} for all the functional content, a pointer to its connection graph description, and bools saying if it contains \textit{Operation} (Section 6.6.20) or \textit{Pipeline} (Section 6.6.19) constructs. Knowing if the FU has operations and pipelines is important because a FU is treated differently in the main functions if it has operations. It also enables some more error checking tests. Full graph descriptions are only created for FUs with operations. In these, the graphs of the instructions defined in operations are merged and edges assigned correct timings to show where pipeline registers should be inserted. The functionality in operation FUs is somewhat restricted for now\textsuperscript{5}, partly because restrictions in the graph representations put restrictions on the FUs. Signals, switches, and if-constructs are, for instance, not allowed anywhere.

The graph classes can automatically generate verilog but FUs not containing operations do not yet have an automated way of generating verilog. You will have to manually write a verilog file to use instead of the \texttt{NoGap}\textsuperscript{CL} code. A special statement (see Section 6.6.21) tells the FU to replace itself with a written verilog file during \texttt{genVerilog}. It is also permitted to override the automatically generated verilog for operation FUs with this statement. This is of course just a temporary fix until automated generation has been implemented for all FUs (though it might be useful to keep this functionality).

\textbf{Controls and Instructions}

During \texttt{buildGraph} the control signals from switches has to be automatically connected. They should be continued from instantiation ports and local signals to

\textsuperscript{5}The restrictions will probably be eased in future \texttt{NoGap} revisions, see Chapter 8.
global outputs of the FU so they can be connected to by a separate decoder. There
is a special class, \textit{Control}, for defining what graph vertex should be assigned which
value for a certain clause to be activated in a switch. A special container class,
\textit{ControlContainer}, handles storing several controls and provides a nice interface
to the objects. Two similar classes handle instructions. The \textit{Instruction} class
couples a list of names\textsuperscript{6} with a \textit{ControlContainer} object to define an instruction.
There is also the \textit{InstructionContainer} class for keeping many instructions with
an effective interface. It has functions for combining the present instruction names
with a list of string names, adding a single name, adding a whole new instruction
and copying another container to this one. In \textit{Operation} all the multiple-choice
clause activations are combined in an \textit{InstructionContainer} to define all instruc-
tions made available by the construct. In \textit{Fu} all the instructions defined by the
operations it contains are stored in an \textit{InstructionContainer} and all controls
activated by default (default-clauses in switches) in a \textit{ControlContainer}.

Grammar

\begin{verbatim}
⟨fu_cons⟩ ::= ⟨fu_cons_start⟩ (⟨port_stmts⟩) { ⟨elements⟩ }
        | ⟨fu_cons_start⟩ () { ⟨elements⟩ }

⟨fu_cons_start⟩ ::= fu IDENTIFIER
⟨port_stmts⟩ ::= ⟨port_stmt⟩;
        | ⟨port_stmts⟩ ⟨port_stmt⟩;
\end{verbatim}

The seemingly redundant \texttt{<fu_cons_start>} non-terminal is used to insert an
action to save the current \textit{ElementList} in the middle of the \texttt{<fu_cons>} non-
terminal grammar rules. The Bison manual \cite{BisonManual} provides more information on
“actions in mid-rule”.

6.6.15 \textbf{Instantiation}

\textbf{Inheritance}

Inherits from the \textit{Statement} (Section 6.6.5) class.

\textbf{Description}

Instantiations are implementations of the \textit{Fu} blueprints that can be connected
inside other FUs in \texttt{NoGap} CL to produce more complex designs. An infinite
number of instances of the same FU can be used and they all share the exact same
implementation. Each FU has to have its own instance of the port interface to
the FU though. If the same port objects were used by all implementations there
would be no difference between instantiations. They therefore make their own list
of ports mirroring the ports in the parent FU but add a layer of extra information
specific to the instantiations. The ports are stored in a special \textit{PortContainer}
object. This makes the actual ports in the \textit{Fu} class internal, and the ports kept
in the instantiations the ones connected to in other FUs. Instantiations also store

\textsuperscript{6}The clauses to activate for the instruction.
their own name, the name of the parent \texttt{Fu}-type, a pointer to the parent \texttt{Fu}-type, and a pointer to the scope of the parent type.

Instantiations have only been referred to as functional unit instantiations thus far, but the \texttt{Instantiation} class is applicable to all \texttt{PortObject} types. But, as stated before, the only real \texttt{PortObject} for now is the \texttt{Fu}. A shell for a decoder instantiation class, \texttt{DecoderInstStmt}, has been implemented though. Section 6.6.16 deals specifically with functional unit instantiations.

\subsection*{Grammar}

No grammar corresponds directly to the \texttt{Instantiation} class. See Section 6.6.16 for the grammar of FU instantiations.

\section*{6.6.16 \texttt{FuInstStmt}}

\subsection*{Inheritance}

Inherits from the \texttt{Instantiation} (Section 6.6.15) class.

\subsection*{Description}

Functional unit instantiations are handled by the \texttt{FuInstStmt} class. It keeps a pointer to an \texttt{ncc::FuUnit} (Section 7.2.7) graph module for the representation of the instantiation in the connection graph. This is a FU cluster as the one shown in Figure 6.10. It is an “instantiation” of the module defined by the ports of the parent FU. \texttt{FuInstStmt} also keeps the list of specified default clauses for this particular instantiation. These can override the “default” default clauses in switches.

\subsection*{Grammar}

\begin{verbatim}
\langle fu_inst_stmt \rangle ::= \texttt{fu:: IDENTIFIER IDENTIFIER}
    | \texttt{fu:: IDENTIFIER ( \langle clause_list \rangle ) IDENTIFIER}
\langle clause_list \rangle ::= \langle clause \rangle
    | \langle clause_list \rangle , \langle clause \rangle
\end{verbatim}

\section*{6.6.17 \texttt{Clause}}

\subsection*{Inheritance}

Inherits from the \texttt{Construct} (Section 6.6.5) class.

\subsection*{Description}

A clause is in essence a bit of code that is identified with a name and can be activated on demand. It is defined in a \texttt{Clause} construct that keeps the code in an \texttt{ElementList}. There are three types of clauses; operation, choice, and default. Operation-type clauses are the ones used in \texttt{Operation} constructs and
are quite different from the other two types. They are not associated with a particular connection for activation but are rather only used by name when defining instructions and when referencing them from decoders. The other two types are obviously the ones used in switches and can thus be activated with clause activation statements. See Section 6.6.18 about ClauseActStmt for more information on clause activations.

When a switch is checked it finds its control signal. A pointer to that connection is stored in all the clauses of the switch along with assigned values in utils::Number objects. The values for activating different clauses are, for now, simply enumerated by order of appearance in the switch construct starting from zero. This will probably be improved upon to enable the programmer to choose his own clause coding or to optimize the coding for simpler hardware. Depending on the connection set as control signal for a clause it can be local to the current FU or available for activation from the outside of an instantiation. A port makes it available from the outside and a signal makes it local.

The connection pointer and a value to assign is all that is needed to make a Control object from a clause. This is done in buildGraph of clause activations to add controls to instructions, and in FUs to make the default controls.

Grammar

\[
\langle \text{clause_cons} \rangle ::= \langle \text{clause_cons_start} \rangle \langle \text{elements} \rangle \\
\langle \text{clause_cons_start} \rangle ::= \langle \text{clause} \rangle \\
\langle \text{clause} \rangle ::= \% \text{IDENTIFIER}
\]

As for functional units the \langle clause_cons_start \rangle non-terminal is used to insert an action to save the current ElementList in the \langle elements \rangle non-terminal.

6.6.18 ClauseActStmt

Inheritance

Inherits from the Statement (Section 6.6.5) class.

Description

Clauses can be activated locally or in instantiations and the activations are stored in the ClauseActStmt statements. Finding the specified clause (or clauses) is similar to the identifiers searching for connections. There is a list of parent instantiations of where to look for clauses and a list of clauses to find. The activations can be written as multiple-choice or normal activations.

The clauses to find must be available in the context of the activation statement. If they are local and an instantiation is being searched they cannot be activated, and the same goes if they are not local and the current scope is being searched. Checks are also made to ensure that operation-type clauses cannot be activated and that multiple-choice activations cannot be outside of operations. As with identifiers only one parent instantiation object is allowed.
When building the graph the instructions are assembled by the activations and the control paths are inserted in the graph. For multiple-choice activations the instructions in the current operation are combined with the names of the clauses to activate. This produces \((\text{clauses}) \times (\text{previous\_instructions})\) instructions for the operation. The names of the clauses to activate are put last in the list of names of the instructions. The instructions are iterated over and the last clause for each instruction is found and made into a \textit{Control} for that instruction. For normal activations all the clauses to activate are made into controls and added to all the instructions in the current operation, but no new instructions are created.

**Grammar**

\[
\langle \text{clause\_act\_stmt\_sing} \rangle \::= \langle \text{clause} \rangle \\
\langle \text{clause\_act\_stmt} \rangle \::= \langle \text{IDENTIFIER} \ (\langle \text{clause\_list} \rangle) \rangle \\
| \langle \text{IDENTIFIER} \ (<\{ \langle \text{clause\_list} \rangle \}>) \rangle \\
| \langle \text{IDENTIFIER} \ . \langle \text{clause\_act\_stmt} \rangle \rangle
\]

Activation of one clause in the current scope got its own non-terminal since it did not fit into the grammar otherwise. The \(<\text{clause\_list}>\) non-terminal is as defined for \textit{FulInstStmt} in Section 6.6.16. The second rule for \(<\text{clause\_act\_stmt}>\) is the grammar for multiple-choice activations.

### 6.6.19 Pipeline

**Inheritance**

Inherits from the \textit{Construct} (Section 6.6.5) class.

**Description**

Pipelines define the phases (pipeline stages) for the operations that use them. When defining a pipeline, previously defined pipelines can be “instantiated” and effectively be extended and combined to form a new pipeline. The pipelines used in defining another pipeline are called sub-pipelines. Every pipeline has its own scope, stores pointers to all sub-pipelines, and knows how many phases it contains.

Names of instantiated pipelines as well as normal phases are saved in \textit{Phase} constructs in a list named \textit{phases} in the pipeline. \textit{Phase} has a variable that tells if it is an instantiated pipeline. When the \textit{phases}-list is checked, the \textit{Phase} constructs that are really pipelines will find that pipeline in the symbol table. The phases of the found pipeline are then copied into the \textit{phases}-list of the current pipeline after the instantiated pipeline phase. Since they are placed after the current phase being checked in the list they will be checked next. Normal phases that are checked will simply enter themselves into the scope of the pipeline, which enables them to be found by the \textit{PhaseStmt} class (Section 6.6.21). When all phases have been checked the phases of “instantiated pipeline”-type are removed from the list. They have been expanded and serve no further purpose. The timing of the individual phases are set by order of appearance in the finished \textit{phases}-list starting with 1. As explained in Section 3.3 the phases can be specified in
the phases-list as allowing stalling, flushing, and as having a timing equal to the preceding phase.

For every pipeline a list of decoders is specified. No functionality for these decoders is implemented beyond checking that whatever is used as a decoder exists. The meaning of having different decoders for sub-pipelines and the full pipeline, or anything similar, has not been defined. Control signals are currently always routed to global inputs, but it would not be hard to route them to another FU specified as decoder for the pipeline.

Grammar

\[
\text{⟨pipeline\_cons⟩} \rightarrow \text{pipeline IDENTIFIER (⟨decoder\_list⟩) \{ ⟨phase\_list⟩ \}}
\]

\[
\text{⟨decoder\_list⟩} \rightarrow
\]

| \text{decoder IDENTIFIER} |
| \text{⟨decoder\_list⟩ decoder IDENTIFIER} |

\[
\text{⟨phase\_list⟩} \rightarrow
\]

| \text{⟨phase\_type⟩} |
| \text{⟨phase\_list⟩ - ⟨phase\_type⟩} |
| \text{⟨phase\_list⟩ /|> ⟨phase\_type⟩} |

\[
\ldots
\]

\[
\text{⟨phase\_type⟩} \rightarrow
\]

| \text{IDENTIFIER} |
| \text{⟨pipeline\_inst⟩} |

\[
\text{⟨pipeline\_inst⟩} \rightarrow \text{pipeline:: IDENTIFIER}
\]

Rule three of the ⟨phase\_list⟩ non-terminal is the syntax for merging two phases by giving them equal timing. Rule four allows the right phase to flush and stall and gives it an increased timing.

6.6.20 Operation

Inheritance

Inherits from the Construct (Section 6.6.5) class.

Description

Operations are the constructs responsible for defining the instructions of a FU. They are represented by the Operation construct and need to be assigned pipelines. The operations of a pipeline are placed in its scope so that phases searched for are found. The Operation constructs are also placed directly in the scope of the FU which forces operations defined in that FU to have unique names. This is necessary to prevent references to instructions in the decoders from being ambiguous. The functional content of an operation is contained in an operation-type clause. It acts almost as a named ElementList.

All connections and instantiations used in the operation gets a timing specific for this operation. When assigning the first input of an instantiation that timing
is saved as the timing of the whole instantiation. Timings of output instantiation ports are calculated by adding their specified offsets to the timing of the instantiation. Input ports simply get the timing of the phase they are assigned in. Two maps are used for storing the timings, one for connections and one for instantiations.

Saving timings in Operation makes it possible to use connections and instantiated objects in different phases for different instructions. The timings could have been saved in the connections themselves if, for example, a map with Operation pointer keys were used to store them. The null pointer key could be the “outside of operation” timing currently stored in the connections\(^7\). The setTiming function used to retrieve timings for connections from Operation objects is explained in Section 6.6.24.

Grammar

\[\langle \text{operation\_cons}\rangle \mapsto \text{operation ( IDENTIFIER ) } \langle \text{clause\_cons}\rangle\]

### 6.6.21 Other Statements

This section provides explanations of some of the smaller and more stand-alone statements and their grammars.

**PhaseStmt**

It represents the phase-changing “@phase”-statements in NoGap\(^CL\). The only purpose of the statement is to change the current timing in traversingInfo to the one in the Phase object it refers to during timingGen and buildGraph. Grammar:

\[\langle \text{phase\_stmt}\rangle \mapsto @ \text{IDENTIFIER}\]

**OperationInstStmt**

The instantiation of operations is different than instantiating functional units. It is more of an “include” statement in the sense that the functional content (i.e. the operation-type clause) of the operation referred to is traversed again and the functionality added to the current operation. To do this the AST-traversing functions simply continue in the operation-type clause of the instantiated operation when this statement is found. Grammar:

\[\langle \text{operation\_inst\_stmt}\rangle \mapsto \text{operation: (clause)}\]

\(^7\)This timing is actually effectively a boolean value since all timings outside of operations are the same.
6.6 Abstract Syntax Tree

ConstantInitStmt

Named constants must be declared and defined in the same statement. The syntax contains a “-”-operator but instead of making an expression of this it is recognized as a ConstantInitStmt. The statement takes an expression to define the constant. It, of course, has to be constant. If it is not folded to a ConstNum after operateCheck an error is thrown saying that constants have to be constant. When identifiers are found to be referring to a named constant the constant folding mechanism is used to fold the identifier to the ConstNum.

Grammar:

\[
\text{constant_init_stmt}::=\text{constant}\ \text{IDENTIFIER} = \langle \text{expr} \rangle
\]

SetTimingStmt

If the programmer for some reason wants to set the timing of a connection or whole instantiation without assigning or reading anything, a special syntax can be used. The “set_timing” built-in $\text{NoGap}^{CL}$ function is translated into a SetTimingStmt in the AST. This class contains an identifier that is checked with a special mode set in the CheckInfo static AST variable. The mode makes the Identifier do an extra check to see if it could be referring to an instantiation.

Grammar:

\[
\text{set_timing_stmt}::=\text{set}\_timing\ (\ \text{IDENTIFIER}\ )
\]

VerilogReplaceStmt

This class is the AST representation of the “verilog(“file.v”);” built-in statement. It is used to define a verilog replacement file used in genVerilog for the FU it is in. Before inserting the file string as the verilog replacement file in the FU construct it is stripped of the string quotation marks. It also gets the path of the current file\(^8\) prefixed. Otherwise the file will be searched for in the path that the program is executed from.

Grammar:

\[
\text{verilog_replace_stmt}::=\text{verilog}\ (\ \text{STRING}\_\text{CONST}\ )
\]

6.6.22 Other Constructs

This section provides explanations of some of the smaller and more stand-alone constructs and their grammars.

Cycle and Always

These constructs keep an ElementList but do not add much to the design themselves. The functionality in the ElementList of a Cycle block is made into clocked logic in verilog, and signals assigned therein are set as registers. Ports

---

\(^8\)The file the statement is written in.
cannot be assigned in a cycle since they are always unclocked. It is forbidden (and totally pointless) to nest cycle-blocks. Always-blocks do not change anything about the code in its **ElementList**, but it can be used to make sure that the contained code is outside of cycle-blocks.

**Grammar:**

\[
\langle \text{cycle\_cons} \rangle ::= \langle \text{cycle\_start} \rangle \{ \langle \text{elements} \rangle \} \\
\langle \text{cycle\_start} \rangle ::= \text{cycle} \\
\langle \text{always\_cons} \rangle ::= \langle \text{always\_start} \rangle \{ \langle \text{elements} \rangle \} \\
\langle \text{always\_start} \rangle ::= \text{always}
\]

**IfElseifElse**

The construct responsible for keeping if-type branches. It has a list of **IfType** constructs with expressions for defining the execution criterion and an **ElementList** with the contents. There are three sub-classes of **IfType**: **If**, **Elseif**, and **Else** inheriting from **IfType** that simply initialize the **IfType** member variables. The functionality and checks for if-type branches are poor and something to be improved upon.

**Switch**

Switches are the main way of branching a **NoGapCL** program. A connection is supplied to switch upon and clauses to activate for different values are defined. The values to assign the connection for activation of different clauses are automatically assigned to the clauses. The switching connection is set by the switch as a control signal. It is actually done in the **operateCheck** function of the identifier specifying the connection after setting a special checking mode in **checkInfo**. There is currently no way of manually assigning control signals or finding and using the activation value for clauses in **NoGapCL**. Functionality of this nature would make the language more flexible.

**Grammar:**

\[
\langle \text{switch\_cons} \rangle ::= \text{switch} \ ( \langle \text{expr} \rangle ) \{ \langle \text{label\_clause\_list} \rangle \} \\
\langle \text{label\_clause\_list} \rangle ::= \langle \text{label} \rangle \langle \text{clause\_cons} \rangle \\
\langle \text{label} \rangle ::= \text{choice} | \text{default}
\]

### 6.6.23 The **operateCheck** function

This function checks the AST built by the Bison parser for errors or peculiarities that should be brought to the programmers attention. It does not check for pure timing errors or structural hazards because they are dependent on the timings having been assigned to connections, which is done in **timingGen** (Section 6.6.24). It also does the constant folding, which is to recognize constant expressions and replacing them with a simple **ConstNum** object.

When checking connection and instantiation declarations the named objects are inserted in the symbol table. When identifiers are checked the symbol table is
asked to locate the object they refer to. If operateCheck has finished without errors it is guaranteed that all identifiers are meaningful and the symbol table contains all objects declared in the code.

When something wrong or noteworthy is found, a message is encapsulated in one of the error classes described in Section 6.2 and thrown as an exception. This exception is caught and stored in the error collector (Section 6.2.3) for the message to be printed at the end of the function. Most checks are small and spread throughout the different classes. Some have been addressed in previous sections. I will now elaborate on a couple of types of checks.

**Identifier and Assignment Size Checking**

Identifiers have a *Size* construct that has three expressions defining the part of the connection referred to by the identifier that is addressed. The size specified in the identifier must be within the limits of the connection. A function in *Size* takes a *Connection* pointer argument and compares the connection limits with the size expressions. Delimiters cannot be compared unless they are constant, so non-constant ones are not compared with the connection. The top bitrange delimiter of the identifier must be lower or equal to the one in the connection and the bottom delimiter must be higher or equal to its connection counterpart. The instance addressed must be between zero and the number of instances specified for the connection. operateCheck is used on the *Size* object before using this function because the constant folding needs to be finished for the expressions.

The assignment binary operator is redefined for *Identifier* to do size checks between the identifier and the expression it is assigned. It finds the bitrange and number of instances of the identifier and uses them as argument values to the *compare* function (see Section 6.6.6) that is then used on the expression. If a bitrange delimiter is non-constant an input value of zero (meaning “don’t care”) for the bitrange is used in the *compare* function. If a delimiter has not been specified the logical default value for it is used (0 for bitrange delimiters and 1 for instances). *compare* is defined differently for each expression class but all compare the argument bitrange and instances with values calculated for the current expression, if possible. *compare* returns the bitrange calculated for the checked expression. It branches down the expression and adds up to complete the whole top expression. If the bitrange of the expression exceeds the argument bitrange an error is thrown. Examples of different *compare* implementations:

- **Add** and **Sub** binary operations. Bitrange is the largest bitrange of the left and right expressions increased by one.

- **Mult** binary operation. Bitrange is the sum of the bitranges of the left and right operand expressions.

- **Concatenation**. Bitrange is the sum of the bitranges of all the concatenated expressions.

- **Replication**. Bitrange is the bitrange of the replicated expression multiplied with the replicator constant number.
• **Identifier.** Finds the size of the identifier just like the assignment operator and compares both bitrange and instance number.

**Context Checking**

The symbol table objects can be entered with a pointer to the object they belong to. This is used throughout the `operateCheck` functions to find the context in which an element is found and to throw errors saying the element was unexpected at this point in the program.

**Symbol Table Errors**

When `operateCheck` uses the scope searching functions (Section 6.5.2) or enters objects in the symbol table, a couple of different errors can be found. An object might not be found or be of the wrong type, and collisions occur when entering objects with the same name in the same scope. These errors are thrown from the symbol table and caught in `operateCheck`. The symbol table does not know where the functions were called from so the caught errors must have location information added before entering them into the error collector.

**Constant Folding**

The binary operations have corresponding functions defined in `Expression`. All these functions are defined here to only return zero. They are redefined in `ConstNum` to return a pointer to the “this” `ConstNum` re-calculated according to the operation if the argument expression is a `ConstNum`. When `operateCheck` is run on a `BinOp` object `operateCheck` is first continued to the operand expression. Then the binary operation function is used on the right operand expression with the left as argument. If both are `ConstNum` constructs it will return a non-zero pointer to a `ConstNum`. The pointer is then put in the `checkInfo` static variable (see Section 6.6.1). The object using `operateCheck` on this object replaces that pointer for the pointer the function was called on. The pointers to the obsolete objects no longer connected to the AST are put in a deletionlist and scheduled for destruction at the end of `operateCheck` in the `Top` object. Folding for unary operations is done the same way as for binary but with only one operand to consider.

**An Implementation Example**

`operateCheck` in `Identifier` is perhaps the most elaborate `operateCheck`. The code is supplied in Appendix A. The function directly and indirectly performs many different checks on the identifier. Some of them are listed below along with what lines in the code they are found on:

- 18-22. Must be in `Fu` or `Operation` construct scopes.
- 25-30. Direct connections cannot be used outside of `Operation` scopes.
6.6 Abstract Syntax Tree

• 32-37. Unable to assign to a direct connection identifier.

• 39-44. Control signals must be in the current scope.

• 62,97. The connection referred to by the identifier must exist.

• 66. When a connection is used as a control signal it is wrong for it to already be used as a normal or control signal. This is checked in \( \text{setUsedControl} \).

• 73,104. Using a control signal as a normal signal is forbidden, and therefore checked in \( \text{setUsed of Connection} \) and \( \text{usePort of Instantiation} \).

• 87-90. If the identifier refers to a defined constant it will be folded into that constant.

• 121-125. The size of the identifier must be within the size of the connection referred to. \( \text{compareObject} \) compares the expressions to the connection referred to by the identifier to the largest extent possible.

• 127-160. If the identifier refers to a port it must logically be available to read or write in the current context.

• 163-166. Cannot assign a port in a cycle construct.

6.6.24 The \textit{timingGen} function

\textit{timingGen} generates and checks the timings of connections. It indirectly also finds some multiple assignment errors. The timings are saved as integers in the connections themselves and in the \textit{Operation} constructs. When connections are used in operations they get a timing assigned specifically for that operation. Connections assigned outside of operations get the current (zero) timing stored in the actual connection or instantiation. This could be replaced with a single boolean variable since the timing value outside of operations is always zero.

A way to simplify the checking of timings and \textit{timingGen} in general would be to remove all functionality extending outside of operations. The timing values and functions in connections and instantiations could be removed and some of the other functions simplified.

Timing Errors

These are the main errors found by \textit{timingGen}:

• Using a connection in operations before it has a timing defined. The connection needs to have a timing defined when the graph is being built.

• Using a connection in a phase with lower timing than it has been assigned is an error because it means the data wanted is not yet available.

• Assigning timing twice for a connection in an operation. It could be meaningful to support assigning the same connection in different phases in the same operation but it would cause implementation difficulties.
Assigning timing for a connection already assigned outside of operations. If a signal is driven somewhere outside the operations it is meaningless to assign it again inside operations. Note that to find these errors the assignment must be before the operation constructs in the code. Assigning before operations ensures that the connection assigned is not assigned in operations. Assigning after the operations instead bypasses assignments to the same connection in operations by inserting a direct edge in the graph description.

If timing for a connection is set differently between different operations there is the possibility of a structural hazard. The Operation constructs of every FU are searched to find these hazards. The errors thrown from the hazard checking are of Info type. It is good to know that the hazards exist but the different timings can also indicate a feature just as much as it can indicate a problem.

Implementation
The function branches down the whole AST but only a few classes actually do some work for timings. PhaseStmt obviously updates the current timing in operations (stored in traversingInfo). Fu does hazard checking when timings for the whole FU have been assigned. The most amount of functionality is in Identifier and Operation.

All connections are used through identifiers, so the timingGen function of Identifier is what assigns and checks the timings. It uses functions in Operation, Instantiation and Connection to do this. Appendix B contains the entire code of the function.

First we will consider identifiers in operations. If the identifier is assigned (line 19-40 in the code), the function called setTiming in the current operation is used. The connection object and possible instantiation object referred to by the identifier is used to produce a key to identify this particular connection. The current timing from traversingInfo is the timing given to that key. setTiming in Operation is quite elaborate and treats every connection-type (input/output-port and signal) differently depending on if it is local or fetched from an instantiation. Three examples:

- Instantiation input port. If timing for the port is already set in the instantiation object there is an error. Otherwise it tries to insert it in the timing-map. If it fails either a double assignment in one phase or a normal timing conflict has been found. If it succeeds it also sets the timing for the whole instantiation in the instantiation-timing-map if it has not previously been set.
- Instantiation output port. Throws a BadProgramming error (Section 6.2.2) since assignments of instantiation output ports should not have passed operateCheck.
- Local signal. If the timing for the signal is already set in the signal there is an error. Otherwise it tries to insert the signal in the timing-map. If it

---

9Like different length instructions.
fails either a double assignment in one phase or a normal timing conflict has been found.

If the identifier is read (line 43-50), the timing of the connection referred to is checked so it is not higher than the current timing. If it is, a timing mismatch error has been found. The timing of the connection is fetched by the `Identifier` member function `getTiming`. It fetches the timing from the current operation, the instantiation, or the connection depending on the context of the identifier. If the timing is not available, an error is thrown. All these checks are bypassed if the identifier is marked as a direct connection. For these, timings do not matter since their timings will not be used in building the graph anyway.

When identifiers are outside of operations (line 53-76), only assignments are of interest. The timings are set with functions in `Connection` and `Instantiation` (`setTiming` and `setPortTiming`). They are totally analogous, except that the function in `Instantiation` has to find the right port to alter in the `PortContainer` of the instantiations. They simply set the timing integer, unless it has already been set and the old value is different from the new, which marks a timing conflict.

### 6.6.25 The `buildGraph` function

The `buildGraph` function makes the connection graph representation from the AST classes. It takes a `side` argument and returns a vertex descriptor. The returned descriptor is used to connect to whatever object the function was executed on. As an example, we can execute `buildGraph` on the operand expression of a binary operation and use the returned descriptor to connect an edge to an input of the `AtomicModule` of the operation. Classes that need to be represented in the graphs make and insert their representation in the current graph, connect input nodes, and possibly return a vertex descriptor for other classes to connect to. `Assign` is special in that it only connects its two operands with an edge and does not have any other representation.

The graphs are only meant for the FUs that contain `Operation` constructs and every such FU gets its own graph. The operations define the data path that the one or more instructions they define need. That path can (and probably will) clash with the path needed for the instructions defined in another operation. To resolve this, all edges (except direct edges) are assigned the name of the operation they are created for. Names of operations are unique to every FU and therefore also for every graph. All edges from all operations in the FU are added to the graph. When the graphs have been filled with the nodes and edges necessary to define all operations in all FUs, modifying transformations are then applied to, among other things, resolve these data path conflicts. Multiplexers are inserted where several operations have assigned the same input. This resolves the conflict but to switch output in the mux another control signal is needed to define the instructions. Control tables for the inserted muxes are automatically generated. Non-direct edges are assigned two timings, one for the source node and one for the target node. A timing difference in one edge means it crosses the timing-gap between two pipeline stages. It is now realized that the necessity of the timings lies in marking these transitions. The same amount of pipeline registers are inserted on the edges as the difference in timing between source and destination node. To
make the graph simpler, all edges with the same timing difference and source and destination node are combined to form one edge instead of many. To preserve the information from all the edges, the operation names and timings are added to the single edge. Combining equal edges is done first, then flip-flop nodes are inserted and optimized by combining the ones that can be considered equal. Lastly the muxes and their input control nodes are inserted. To be able to generate verilog code from the graph, the edges are assigned names and bitrange sizes. After these modifying transformations have been performed on the graphs they are ready to generate verilog. Example 6.8 shows snapshots of a transformation from raw to fully modified graph.

Example 6.8: Transformation of a Connection Graph

Figure 6.7, Figure 6.8, and Figure 6.9 each show a connection graph in different stages of transformation. Look at them in the numbered order to follow the transformation. A legend to accompany the graph figures can be found in Figure 6.6.

Graph Representations

Expressions, instantiations, and ports are currently the AST classes with direct representations in the graphs. All of the unary and many of the binary operations still lack graph modules. The remaining expressions (except Assign) are represented by the ncg::AtomicModule classes. These modules express instantaneous atomic functions. The NoGap CG classes are described in Chapter 7. All graph representations used at the time of writing are listed below.

- Ports add an input or output port of its own name to the Module of the PortObject (Fu) it is found in. If the FU has operations a global input or output port is added to the current graph. Every graph has an InPortUnit and OutPortUnit where global input and output ports are kept.
Figure 6.7. Raw graph with all inserted nodes and edges. The edges with timing and optimized inserted flip-flops. Edges information and the nodes to connect them considered equal from a pipeline point of view (same source and destination nodes classes while traversing the AST). No transformations have been applied.

Figure 6.8. Graph with combined edges and optimized inserted flip-flops. Edges considered equal from a pipeline point of view (same source and destination nodes and timing difference) have been merged and the timing differences evenlyed out by inserting pipeline register flip-flops. The flip-flops have been optimized by only making every needed delay of signals once.
Figure 6.9. Finished graph with muxes and names. Where more than one edge arrived at one input, control muxes have been inserted to be able to separate the instructions. Edges have also been assigned sizes and appropriate unique names, based on what they are connected to, for verilog generation, This graph is now ready to generate verilog.
• **ConstNum.** Uses *AtomicConstModule* to store an unlimited precision integer and provide an output node to connect to.

• **Concatenation** and **Replication** use the *AtomicMergeModule* that is created with a specified number of input nodes and one output node. It translates into a normal Verilog concatenation but is currently used for replications also. Replications connect the vertex descriptor of their one expression to all the inputs of the module while concatenations have a separate expression for every input.

• Binary operations all use different *AtomicModule* sub-classes, but they must have two inputs and one output that are named the same in all the different implementations. The names of the nodes have to be the same to give the *buildGraph* function in *BinOp*, used for all the binary operations, access to them.

• Identifiers can either just return the vertex descriptor of the connection it refers to or, if it is a partial identifier (with defined size), it must provide access only to the addressed part of the connection. Partial identifiers are currently only allowed for reading. The identifier then inserts an *AtomicTapModule*, with the vertex of the connection referred to connected to an input, and returns its output node. The *AtomicTapModule* also has two inputs for the bitrange delimiters. The equivalent “funnel” module for assigning identifiers has not yet been implemented.

• Instantiated functional units make an *FuUnit* from the *Module* of the instantiated FU-type. It receives the right input and output nodes automatically. The port nodes connected to a central FU-node is called a FU cluster. A Dot representation of such a cluster is provided in Figure 6.10.

![Figure 6.10](image)

**Figure 6.10.** A functional unit graph cluster. This is the graph representation of an instantiation of the “ALU” functional unit. It has two input ports, “data_a_i” and “data_b_i”, and one output port, “res_o”.

**Control Routing**

Since manual assignments of control signals are forbidden, edges connecting control signals must be automatically inserted. To be able to assign the signals from
outside the FU, all control signals are routed to new global input ports in the graph. When clauses are activated in operations, Control objects (see Section 6.6.14) are made that specify what vertex should be assigned what value. The controls are added to the instructions of the operation. Control objects keep both the control vertex descriptor (global input) and the normal vertex descriptor of the assigned connection. When control signals that do not have an input control vertex are encountered, a global input is added to the current graph and its vertex descriptor used for the signal. That descriptor is then used for all controls setting that connection.

All instructions from all operations along with the default controls are saved for every FU with operations. The default controls should be applied to connections when no controls of an activated instruction sets it. The edges connecting the control input nodes with the connection nodes are inserted after all instructions and default controls have been gathered from the FU.

The default controls are not fetched as is, but rather the clauses to activate by default for every instantiated FU are gathered. From that information the default controls can be made. If a control signal still does not have a control input vertex when making the default controls, a global input is inserted in the graph and its vertex used. This also means that the control signal is never used by way of activating a clause and the default value can be permanently connected for the signal. For now a direct edge is inserted instead, but in future compiler versions an AtomicConstModule could be permanently connected to the connection vertex and the default control deleted from the FU.

### 6.6.26 The genVerilog function

This function is responsible for making the verilog code from the finished FUs after all the other main functions have been successfully executed. It iterates over the FUs in the ElementList of the Top object to see if any of them have generated a graph representation. If they have got one, the verilog generating function of the graph is executed and verilog code is output to a stream argument. If a verilog replacement file has been specified, that file is copied to the stream instead. Verilog replacement files is the only way to get a verilog output from FUs without a connection graph at the moment.

### 6.6.27 The printDot function

Printing the Dot graph representation of the AST is handled by the printDot function. Figure 6.4 is an example of a graph made by printDot. It is first introduced in the AST tree in the three classes inheriting directly from Base. The Top class has a different interface to the function that allows specifying the name of a file to write the output to and the detail level. The detail level makes the function avoid some branches of the AST and skip printing certain classes. Detail can be specified at four different levels and is put in the dotInfo static variable for easy access by all printing functions. dotInfo also keeps track of how many elements have been printed. The printDot function in Top also prints the
starting line of the Dot output where default colours and style are defined and
encloses the output of the rest of the AST in braces.

The printDot function of Connection is representative of all other printDot
functions and is explained in Example 6.9.

---

**Example 6.9: The printDot Function of Connection**

First the number of printed elements counter is increased in dotInfo (line 3). Then the label of this element is fetched (lines 7-11). This label is the identification string stored in Base (Section 6.6.1). If __NICE_PRINT__ is defined the identification string is fetched without a unique number prefix otherwise added to the string. This makes a nicer looking graph printout but the number can be helpful in debugging since text printouts of objects can be matched to specific nodes in the Dot graph. Every graph node is identified by the pointer to the printed object. This way all nodes are guaranteed to be unique since all pointers must be unique. The “this” pointer is therefore labelled and coloured (lines 13-15). The label is what is shown in the graph so the actual node representation is hidden from the end-viewer.

The Size object owned by the Connection must now be printed (lines 17-24). This is only done if the detail level exceeds 3. First the edge between this connection and the size object is printed and then the actual Size object.

```cpp
std::ostream& Connection::printDot(std::ostream& ost) const
{
    dotInfo.inc();
    std::string dot_label;

    #ifdef __NICE_PRINT__
        dot_label = getIdentType();
    #else
        dot_label = getIdent();
    #endif

    ost << "\" << this << "\" [label="\" << dot_label << "\"];" << std::endl;
    ost << "\" << this << "\" [color=yellow];" << std::endl;

    if(dotInfo.detail() > 3)
    {
        if(size)
        {
            ost << "\" << this << "\" -> \"" << size << "\";" << std::endl;
            size->printDot(ost);
        }
    }
    return ost;
}
```
The printDot function of Element uses the default colour and does not extend to any member objects. It is used as a default printDot function by some leaf AST classes where the printing does not need to continue down the tree. In Fu the printing is done in a Dot subgraph cluster. This boxes the graph of the FU in a rectangle and makes sure its nodes are separated from other subgraphs (FUs).

6.7 Compilation Output Examples

This section will demonstrate the regular output of the compiler to the user. Example 6.10 shows the output from a compiler run on a reduced instruction set MIPS32 processor without a decoder that was written while evolving the language. Example 6.11 shows a compiler run on the same processor with some interposed errors.

--- Example 6.10: Compilation Output ---

The “mips_new.fud” file holds the top FU and “mips_fus.fud” holds the leaf FUs. The warnings that are issued originate from assignments where the size of the assigned connection might not be able to hold the value of the right-hand side expression.

1 Parsing file mips_new.fud
2 Parsing file mips_fus.fud
3 Found end of file.
4 Found end of file.
5 ——— Checking syntax of AST ———
6 mips_fus.fud:34: warning: data might be lost in assignment
7 mips_fus.fud:38: warning: data might be lost in assignment
8 mips_fus.fud:43: warning: data might be lost in assignment
9 mips_fus.fud:126: warning: data might be lost in assignment
10 Completed successfully!
11 ——— Assigning and checking timings ———
12 Completed successfully!
13 ——— Building graph and routing controls ———
14 Completed successfully!
15 ——— Operating on graphs ———
16 Operating on graph of "mips":
17 Combining edges... done
18 Inserting Flip Flops... done
19 Inserting muxes... done
20 Combining equal flip flops... done
21 Assigning edge names... done
22 Setting edge size... done
23 Completed successfully!
24 ——— Generating verilog output ———
25 Inserted file "alu.v" for FU "alu"
26 Generating code for FU "mips"... done
27 Completed successfully!
28 Printing graphs to Dot with prefix "fu_graph_"... done
29 Printing AST with detail 4... done
30 printed 904 elements to file ast.dot
Example 6.11: Compilation Output With Errors

A multitude of errors are found when checking this AST. Because of the errors found the compiler does not progress into the timingGen function.

Parsing file mips_new.fud
Parsing file mips_fus.fud
Found end of file.
Found end of file.

--- Checking syntax of AST ---
mips_fus.fud:19: error: unable to assign constant a non constant \
expression
mips_fus.fud:21: error: no default clause in switch construct
mips_fus.fud:38: warning: data might be lost in assignment
mips_fus.fud:42: warning: data might be lost in assignment
mips_fus.fud:47: warning: data might be lost in assignment
mips_fus.fud:51: error: unable to use "control_ci"; the connection \
is a control signal
mips_fus.fud:53: error: object 'const1' is undefined in this scope
mips_fus.fud:55: error: unable to read 'next_pc_o', port is output
mips_fus.fud:59: error: faulty activation; control signal 'control_ci' \
is not available from this scope
mips_fus.fud:138: warning: data might be lost in assignment

Errors found: 6 !!
Exiting...

Printing AST with detail 4... done
printed 920 elements to file ast.dot
Printing symbol table... done
printed 49 elements to file symtab.dot

Compilation aborted at Sun Mar 25 17:54:03 2007
Chapter 7

The NoGap Connection Graph

This chapter contains a short overview of what the NoGap connection graph (NoGap\(^{CG}\)) is, and brief descriptions of the classes used to create it. Only information needed to understand the collaboration between the AST classes and the NoGap\(^{CG}\) is included. The NoGap\(^{CG}\) classes are an integral part of the compiler and for completeness they need to be presented in short.

7.1 Overview

The NoGap\(^{CG}\) is an annotated graph of connections between functional units and all the other constituents of a NoGap processor pipeline. It is used for the Mase part of the NoGap\(^{CD}\) to describe the merged data paths and control paths of all instructions defined for the processor. The NoGap\(^{CG}\) classes use the Boost Graph Library (BGL) [19] to create the internal graph representation.

A connection graph can be used to generate a verilog design of the described pipeline and, as with the AST and symbol table, a Dot representation of the created graphs can be printed. Even for simple designs the connection graphs get very large. A trivial example of a graph with one instantiated FU and one operation is found in Figure 7.1.
Figure 7.1. Dot representation of a trivial connection graph. The legend in Figure 6.6 is applicable to this graph also. The graph contains an ALU FU represented by a FU cluster as in Figure 6.10, but with an added control input port for switching functionality. The ALU is placed in a pipeline and simply connected to the global inputs and outputs of the graph (representing the ports of the top FU). The “ALU” operation can define several instructions for this example by activating different clauses in the ALU that assigns the “ctrl_i_alu_ctrl_i” control input different values. The “VOID_NODE” is a default node required by the implementation that should never be connected to.
7.2 Descriptions of the NoGap$^G$ Classes

There is much more functionality implemented than will be described in this section. These are only the classes used directly by the AST and those needed to describe them.

7.2.1 Graph

Inheritance

Does not inherit from any class but collaborates with many of the other NoGap$^G$ classes by having them declared as friend classes. Most of the functionality of Graph is only provided to these classes.

Description

The Graph class is basically an advanced wrapper class for the BGL interface of a contained graph. It is tightly coupled to the other NoGap$^G$ classes since all the actual contents of the graph resides in this class. Its public interface, among other things, provides functions for adding edges to the graph and performing the transformations discussed in Section 6.6.25.

Usage in the AST

Pointers to Graph objects are kept in the traversingInfo static AST variable and in every FU. traversingInfo keeps the current graph and has a list of all created graphs. FU objects keep their own graph descriptions if they have one.

7.2.2 Module

Inheritance

Module acts as the base-class for several classes. Figure 7.2 shows the inheritance graph.

![Figure 7.2. Inheritance of the module classes](image-url)
Description

The Module class contains a blueprint of an interface. It has tables of input and output ports with their names coupled with their sizes. These tables are used to enable replication of the interface when, for example, instantiating functional units.

Usage in the AST

Every Fu has a Module that is used when instantiating it. Ports are added to the module of the current FU in the buildGraph function of Port.

7.2.3 AtomicModule

Inheritance

Inherits from the Module (Section 7.2.2) class.

Description

AtomicModule acts as a base-class for specialized modules that have a predefined meaning in the language and atomic function in the graph. It keeps the number of input ports and has a function for retrieving that number.

Usage in the AST

Every binary operation (BinOp) has a pointer to an AtomicModule. The specific binary operations, like Add, make it point to classes derived from AtomicModule that represent their functionality in the graph.

7.2.4 AtomicAddModule and AtomicMulModule

Inheritance

They inherit from the AtomicModule (Section 7.2.3) class.

Description

AtomicAddModule and AtomicMulModule are the specific graph representations of the Add and Mul, addition and multiplication, binary operations. They define a module with two input ports and one output port.

Usage in the AST

When created the Add and Mul classes make a new AtomicAddModule or AtomicMulModule respectively that is used to represent them in the buildGraph function of BinOp.
7.2.5 Other Atomic Modules

These classes inherit from the AtomicModule (Section 7.2.3) class. They are used for specific atomic functions in the graph that are not unary or binary operations.

AtomicConstModule

The AtomicConstModule is used to represent the constant numbers of the ConstNum AST class in the graph. It uses an unlimited precision integer to hold the value and has one output port.

AtomicMergeModule

The AtomicMergeModule is used by concatenations and replications to fuse several incoming connections into one. This module can be created with any number of input ports but it has only one output port.

AtomicTapModule

The AtomicTapModule is used by identifiers when they need to extract a certain bitrange from a connection. The module has one input for the connection to tap bits from and one output for the tapped bitrange. The bitrange is determined by two additional input ports.

7.2.6 Unit

Inheritance

Unit acts as the base-class for several classes. Figure 7.3 shows the inheritance graph.

![Inheritance of the unit classes](image)

**Figure 7.3.** Inheritance of the unit classes

Description

The Unit class contains the basic functionality to implement vertex collections. It is coupled to a specific graph where its content is placed. Vertex descriptors to
the input and output ports of the vertex collection can be fetched by referencing their name with a function.

Usage in the AST

It does not have a direct usage in the AST classes.

### 7.2.7 FuUnit

Inheritance

Inherits from the Unit (Section 7.2.6) class.

Description

Upon its creation, the FuUnit creates a central node and places it in the graph. An argument Module or AtomicModule blueprint supplies the ports to connect to the central node. The ports are connected with edges of the InternalFuArc type (Section 7.2.13). The result is a FU cluster as in Figure 6.10. Vertex descriptors to input and output ports are stored in separate tables coupled with their names to make it possible to retrieve them.

Usage in the AST

Used by instantiations to add a named instance of a FU to the graph by using the Module of the FU. The stored port descriptors are the ones fetched and returned by identifiers referencing ports of instantiations.

### 7.2.8 PortUnit

Inheritance

Inherits from the Unit (Section 7.2.6) class.

Description

PortUnit provides the basic functionality to add and reference global input and output ports in a graph. Vertex descriptors to ports are stored in a table coupled with their names to make it possible to retrieve them.

Usage in the AST

It does not have a direct usage in the AST classes.

### 7.2.9 InPortUnit and OutPortUnit

Inheritance

The classes inherit from the PortUnit (Section 7.2.8) class.
7.2 Descriptions of the NoGap\textsuperscript{CG} Classes

Description

The \textit{InPortUnit} and \textit{OutPortUnit} classes are used to add and reference global input and output ports respectively in a graph.

Usage in the AST

When a new graph is created, one \textit{InPortUnit}, and one \textit{OutPortUnit} is created and coupled to the graph. These can be retrieved from \textit{traversingInfo} to add ports to the current graph. This is used by the \textit{buildGraph} function of \textit{Port} if the current FU has operations. The vertex descriptor of the added port is also stored in \textit{Port}.

7.2.10 \textit{ArcInfo}

Inheritance

\textit{ArcInfo} acts as the base-class for several classes. Figure 7.4 shows the inheritance graph.

![Inheritance Diagram]

\textit{Figure 7.4.} Inheritance of the arc information classes

Description

When edges are created and inserted in the graph they must be supplied a type-object that determines their function and appearance in the graph. \textit{ArcInfo} is the base-class of these type-objects and it only keeps the Dot printing information of the edges.

A more elaborate info class hierarchy also exists for vertices but since it is not directly used from the AST classes it is not presented in this chapter.

Usage in the AST

It does not have a direct usage in the AST classes.
7.2.11 OutsideArc

Inheritance

Inherits from the ArcInfo (Section 7.2.10) class.

Description

The OutsideArc class is used to represent the normal edges inserted in the pipeline. The name of the operation it is used for as well as a source and destination timing must be specified.

Usage in the AST

This type is given to all edges created by buildGraph in operation constructs that should not be direct connections but part of the pipelining. The timing difference between source and destination is used to insert pipeline registers on the edge.

7.2.12 DirectArc

Inheritance

Inherits from the OutsideArc (Section 7.2.11) class.

Description

The DirectArc class marks the type of edge that disregards pipelining and makes a direct connection from one vertex to another. It does not need any timings or operation names.

Usage in the AST

This type is used when edges are inserted outside of operations and when a connection is referenced by an identifier specified as direct. When the control input of a control signal is never assigned, the default control is used to insert a direct edge.

7.2.13 InternalFuArc

Inheritance

Inherits from the ArcInfo (Section 7.2.10) class.

Description

InternalFuArc is used to mark the type of edges used internally in the FuUnit FU cluster representations.
Usage in the AST

It does not have a direct usage in the AST classes.
The NoGap Connection Graph
Chapter 8

Future Work

This chapter will present improvements that can be made to the states of \( \text{NoGap}^{CL} \) and the implemented compiler as they stand at the end of this thesis.

8.1 Language Features

The \( \text{NoGap}^{CL} \) will continue to evolve and new features will be added. Below is a list of possible improvements.

- Ways of marking dependencies between input and output signals of FUs so that offsets for outputs can be added to the timing of when a particular input is assigned.

- The language should provide a way to manually set the coding of clauses in switches. This could be done either in the switch constructs themselves as in Example 3.6 or by writing tables in a separate file that the switch construct can read.

- It should be possible to get the coding of a clause by referencing it with a statement. There should also be a way of overriding the control status of a signal and enable it to be assigned like any other signal. This enables the programmer to manually make the control logic for the control signals if that is wanted.

- If-else constructs are poorly specified at the moment and their meaning, syntax, and hardware translation should be thought about.

When the language has been further refined it would be helpful to have standard libraries to include with preprogrammed FUs like memories, register files, ALUs, and perhaps simple processors.
8.2 Implementation Features

The implemented NoGap compiler is nowhere near finished so there is definitely room for improvement. Some basic features envisioned for the language, like decoders, were omitted for now due to the time constraints imposed by the thesis. Because of this there are many improvements that can be made. Some are smaller and some are bigger changes to the compiler. This section addresses some of the omitted features and some possible improvements found and thought about while implementing.

8.2.1 Decoder Support

The most obvious part missing in the implementation is the lack of support for decoders. Since they are fairly similar to functional units it would be smart to use the same parser and just add extra tokens and rules. Ports and signals can, for example, use the same Bison rules and AST classes. If some type of statement would have to get another meaning in the decoders compared to FUs we could make a new non-terminal with the same rules using other terminal tokens. When a decoder description is included, a special state in the lexer can be entered that returns other tokens for the same matched input. The problem could be if expressions would have to be modified to a greater extent to accommodate a general control syntax. In this case it might be easier to create a totally new parser for decoders that is called from the current parser when decoders are included.

New AST classes must be added for the constructs and statements of the decoder descriptions. An important part is figuring out how the information will be stored in these classes and how they will access and gather information they need from the other AST classes. A large information exchange have to take place to check and assemble the instructions specified in the “operation_codes” construct. Signals to assign from the activated clauses and muxes inserted in the graphs must be fetched for each instruction and coupled with the defining bits of the operation type. This creates a control table that should be used to make the decoding logic.

The “forwarding” construct has to find the graph nodes to forward between and modify the graph with muxes and connections to enable the forwarding. The control tables of the inserted muxes and supplied forwarding criterion must be used to make the extra control logic needed.

To automatically generate an assembler for a processor the information from the operation-types with the free bits specified should be gathered for each instruction and coupled with the specified assembly command and arguments. It should be easy to find the bits in the instruction that each argument represent. This information can then be used with an assembler “template” to produce an assembler for the specific processor.
8.2 Implementation Features

8.2.2 Spawners

The verilog spawner, which is now mostly implemented in the connection graph, has to be expanded with a better AST-traversing function that generates verilog from the Mage part of the \texttt{NoGap} \texttt{CD} as well. Since \texttt{NoGap} \texttt{CL} is pretty similar to verilog in much of the syntax, many AST classes can have a direct translation to verilog. The \texttt{NoGap} \texttt{CL} switch is equivalent to a verilog switch etc. A cycle accurate C++ simulator spawner should also be implemented.

8.2.3 AST Features

- The AST-traversing functions could be implemented separate from the AST classes. Now the functions are members of every AST class, but it might have been better to implement the classes and the traversing functions separately. This would make it easier to add new traversing functions because their implementations would be centralized and easier to get an overview of. It would also increase the distinction between the \texttt{NoGap} \texttt{CD} and the spawners as all the functions used by spawners can then be implemented separately.

- Nested switch constructs are allowed in the language but the controls do not take this into consideration. Activating a clause in a switch enclosed in the clause of another switch only sets the control of the nested clause and not the clause that contains it. It would be preferable if the controls of both clauses were activated. This means we have to keep track of clause dependencies and allow multiple controls for every clause. All of these dependencies should also be checked so clauses that are mutually exclusive cannot be active at the same time. This might be best checked from the generated graphs by checking for multiply assigned control nodes for one operation.

- Allowing switches in FUs with operations but not in the actual operation constructs. \texttt{buildGraph} could disregard the switches and simply use them like any other switch in another FU. The switch has to be connected to a local signal though.

- Automatically finding the offsets of outputs in FUs. It could either be done by default or by explicitly telling the compiler to do it for certain outputs with some kind of syntax like “\{+auto+\}”. Finding these offsets are not trivial because of program branching and such. A special graph will probably have to be created and traversed differently depending on what clauses are active.

- The error messages from the Bison parser must be improved. The automatically generated messages use internal token names and many errors cause large parts of a program to fail during parsing.

- The \texttt{NoGap} \texttt{CG} classes support parsing of simple expressions to automatically set the size of connections. These expressions can reference other connections and add, multiple and such. This should be used in the AST classes to enable the “[auto]” bitrange sizing of connections suggested in Example 3.6.
Future Work

- The automatically assigned codings of switch clauses could somehow be optimized to produce as simple hardware as possible.

- The verilog spawner could put the coding of switch clauses in files with parameters that are included in the verilog design. This enables the programmer to easily change the coding of clauses without having to run the \texttt{NoGapCL} compiler again.

- Control routing currently always route controls to global input ports of the current graph. The specified decoder should instead be taken into consideration and the controls routed to that object instead. This is useful even without a “real” \texttt{NoGapCL} decoder since FU instantiations can be specified as decoders. The contents of these FUs could be replaced with a hand-written verilog file. Future \texttt{NoGapCL} functional unit descriptions might be expressive enough to write the decoding logic as a normal FU.

- When control routing adds global ports in a graph they should also be added to the modules of the FU so that instantiations of the FU get these ports as well. This would enable the control signals to be propagated to a second FU that instantiates the first FU. The implications and logistics of having several layers of pipelined FUs should be thought about.

8.2.4 \texttt{NoGapCG} Features

- More atomic modules should be added. Most of the binary operations and both of the unary operations still lack a graph counterpart. Identifiers also need a “funnel” module to use when assigning specific parts of a connection.

- Signals need a graph representation. It should be as simple as possible with one input port and one output port so that it can be assigned and connected to other nodes.

- The concept of connection instances should be continued to the graph classes. Using connection instances within operations is disregarded when it comes to building the graph in the current state of the \texttt{NoGapCL} compiler.

8.3 Testing

The language and compiler have thus far never been thoroughly tested. We have arrived at a point where there is some content created from the written \texttt{NoGapCL} code, but it needs to be verified that it really reflects the input as it was intended. There are also probably bugs left in the error reporting. Perhaps some errors and warnings need to be more verbose and some checks might be too meticulous and annoy the user. This will have to be adjusted. Some testing might also give insight as to which improvements from this chapter will be the most valuable to work on. I suspect that the concept and syntax of timings for connections in the operations, and the pipelined signals that result, will be the things causing the most confusion.
and erroneous designs for early adopters. More help for the user in this area is probably something that should be addressed soon.

At first many small designs of different character should be written and verified by examining the generated Dot outputs of the AST and connection graphs. Generated verilog from the top FU along with leaf FU descriptions written by hand should also be synthesized and tested. When these designs are verified and the language and compiler are updated according to comments on flaws or inconsistencies, a larger design, like a MIPS with a reduced instruction set, should be verified.
Future Work
Chapter 9

Conclusions

From the initial ideas and visions of a processor HDL that spawned this thesis a few core language features were filtered out that could provide the wanted functionality of the language. By pseudo-implementing a MIPS32 processor these core features were refined and minor features and further details added to adequately describe the processor in a way that was easy to understand. \texttt{NoGapCL} migrated towards borrowing more syntaxes from C++ and verilog. The syntax and meaning of the core features have been finalized and the language is now expressive enough to accommodate simple RISC processors. An implemented design can be divided into three parts:

- Top FUs that define the pipelines, data paths, and wanted functionality for instructions.
- Leaf FUs that define the actual functionality.
- Decoders that define instructions and decode them into control signals. They also define more advanced control structures and logic like forwarding.

Using \texttt{NoGapCL} will be a more efficient way to create processors instead of using RTL languages. This is accomplished by avoiding micromanagement of implementation details associated with control routing and pipelining. The user can focus on specific instructions one by one and then have the data paths of all instructions merged and control paths automatically routed. \texttt{NoGapCL} also holds up well against current processor HDLs in that it allows more design freedom.

The implemented compiler for \texttt{NoGapCL} has been presented in the report. It uses a Flex lexer, a Bison parser, an AST, a symbol table, and functions traversing the AST. The AST-traversing functions can, among other things, find and report many kinds of errors that users can make, assign and check timings of signals used in the pipeline, and build a connection graph description of the processor pipeline using the timings. The compiler is still in the early stages of development and many features, like decoders, have not yet been addressed. The \texttt{NoGapCL} compiler, at the time of writing, only creates synthesizable verilog output for the
pipelines of written processors from their connection graph descriptions. Leaf FUs with functionality must be written by hand.

There are still many implementation hurdles to overcome, as made evident by Chapter 8. During the course of the thesis it has become clear though that the concepts of \( \text{NoGap}^C \) are possible to implement and not just visions.
Bibliography


Appendix A

The operateCheck Function of Identifier

```c++
bool Identifier::operateCheck(const Side::assign side)
{
    try
    {
        // In FU or operation scope?
        if (!parserInfo.table->scopeObjectIs<ast::Fu>() && !parserInfo.table->scopeObjectIs<ast::Operation>())
        {
            throw error::Unexpected("identifier", file(), line());
        }

        // Direct outside of operation scope?
        if (direct && !parserInfo.table->scopeObjectIs<ast::Operation>())
        {
            throw error::Unexpected("direct connection requested outside of operation", file(), line());
        }

        if (direct && side == Side::LEFT)
        {
            throw error::GeneralNormal("cannot assign to direct connection", file(), line());
        }
    }
    // This function does all checks for this element while filling in the symbol table with new objects. Function is continued recursively in all elements pointed to. Constant folding is done by replacing folded expressions with expressions retrieved from parserInfo. If this element is folded the result is put in parserInfo and true is returned. |
    |param[in] side (RIGHT or LEFT side of assignment check) |
    |return Bool (true = fold this element, false = don't fold) |
```
118 The operateCheck Function of Identifier

```cpp
    if (checkInfo.isMode(CheckInfo::CONTROL) && !parents_m.empty())
    {
        throw error::ControlRoutingError(
            "control signal must be in current scope",
            name_m, file(), line);
    }

    if (checkInfo.isMode(CheckInfo::CONTROL) && side == Side::LEFT)
    {
        throw error::BadProgramming(
            "should be unable to assign during control checking",
            __FILE__, __LINE__);
    }

    // More than one -> operator?
    if (direct > 1) throw error::SeveralDirect(name_m, file(), line);

    Connection* temp_conn;

    if (parents_m.empty())
    {
        try
        {
            temp_conn = parserInfo.table->find<ast::Connection>(name_m);

            if (checkInfo.isMode(CheckInfo::CONTROL))
            {
                temp_conn->setUsedControl(file(), line);
            }
            else if (checkInfo.isMode(CheckInfo::SET_TIME))
            {
            }
            else
            {
                temp_conn->setUsed(file(), line);
            }
        }
        catch (error::TableError& err)
        {
            if (checkInfo.isMode(CheckInfo::SET_TIME))
            {
                Instantiation* temp_inst;
                temp_inst = parserInfo.table->find<ast::Instantiation>(name_m);
                instObj = temp_inst;
                return false;
            }
            else
            {
                ConstantInitStmt* cis = parserInfo.table->find<ast::ConstantInitStmt>(name_m);
                checkInfo.setNewElement(cis->getConstant());
                return true;
            }
        }
    }
    else if (parents_m.size() == 1)
```
{ temp_conn = parserInfo.table->forwFind<ast::Connection>(name_m, parents_m);
  ast::Instantiation* inst_obj;
  inst_obj = parserInfo.table->find<ast::Instantiation>(parents_m.front());
  // Mark the Port used in instantiation
  inst_obj->usePort(name_m);
  // Assign the member instantiation pointer for easy future access.
  instObj = inst_obj;
} else //more than one .-operator
{
  throw error::SeveralUndirect(name_m, file(), line());
}

if(checkInfo.isMode(CheckInfo::SET_TIME))
{
  connObj = temp_conn;
  return false;
}

// Check that size of identifier is within object size
if(size)
{
  size->operateCheck(Side::RIGHT);
  size->compareObject(*temp_conn);
}

if(side == Side::RIGHT)
{
  // Using a port?
  if(ast::Port* port_obj = temp_conn->is<ast::Port>())
  {
    if(port_obj->isOutput() && parents_m.empty())
    {
      throw error::PortRead(port_obj->name(),
                             "output", file(), line());
    }
    if(!port_obj->isOutput() && !parents_m.empty())
    {
      throw error::PortRead(port_obj->name(),
                             "input", file(), line());
    }
  }
  
  else if(side == Side::LEFT)
  {
    // Check assigned identifier

    // Assigning a port?
    if(ast::Port* port_obj = temp_conn->is<ast::Port>())
    {
      if(port_obj->isOutput() && !parents_m.empty())
      {

throw error::PortAssign(port_obj->name(),
    "output", file(), line());
}
if (!port_obj->isOutput() && parents_m.empty())
{
    throw error::PortAssign(port_obj->name(),
        "input", file(), line());
}

// Assigning port in Cycle construct?
if (traversingInfo.inCycle())
{
    throw error::PortRegister(port_obj->name(), file(), line());
}
else if (ast::Signal* sig_obj = temp_conn->is<ast::Signal>())
{
    // If in Cycle, signal->register
    if (traversingInfo.inCycle())
    {
        sig_obj->setType(Signal::REGISTER,
            traversingInfo.inCycle(),
            file(), line());
    }
    else
    {
        sig_obj->setType(Signal::NORMAL, 0, file(), line());
    }
}
else
{
    throw error::AssignmentError("can only assign connections", 
        file(), line());
}

// If all test are passed, then assign identifiers connection object.
connObj = temp_conn;
}
catch(error::TableError& err)
{
    err.setPosition(file(), line());
#ifdef __ERROR_COLLECT__
    parserInfo.errorCollector->enterError(err);
#else
    printPosition(message::info);
    message::info << err << std::endl;
#endif
    parserInfo.incError();
}
catch(error::Normal& err)
{
#ifdef __ERROR_COLLECT__
    parserInfo.errorCollector->enterError(err);
#else
    message::info << err << std::endl;
#endif
    parserInfo.incError();
211   }

213   return false;
}
Appendix B

The timingGen Function of Identifier

```cpp
/*!
 * This function finds the timings for signals and ports of all FU:s and
 * enters the specific pipeline timestamps for signals/ports into the
 * operations. At the same time it checks for apparent timing errors like
 * using a port value before it is available.
 */

* [param[in] side (RIGHT or LEFT side of assignment) */

void Identifier::timingGen(const Side::assign side)
{
    if (traversingInfo.inFu() &&
        traversingInfo.inFu()->hasOperations())
    {
        Connection* conn_obj = getConnection();

        Operation* op_obj;
        if ((op_obj = traversingInfo.inOperation()))
        {
            if (side == Side::LEFT)
            {
                if (parents_m.size() == 0)
                {
                    op_obj->setTiming(0, conn_obj, traversingInfo.timing(),
                                       file(), line());
                }
                if (parents_m.size() == 1)
                {
                    ast::Instantiation* inst_obj = getInstantiation();

                    op_obj->setTiming(inst_obj, conn_obj,
                                       traversingInfo.timing(),
                                       file(), line());
                }
                if (parents_m.size() > 1)
                {
                    throw error::BadProgramming(
```
"wrongful identifier in timing function",
    __FILE__, __LINE__);  
}  
else  
{  
  if(!direct)  
  {  
    if(getTiming() > traversingInfo.timing())  
    {  
      throw error::TimingMismatch(conn_obj->name(),
                                  file(), line());  
    }  
  }  
}  
else  
{  
  if(side == Side::LEFT)  
  {  
    if(parents_m.size() == 0)  
    {  
      conn_obj->setTiming(traversingInfo.timing(), file(), line());  
    }  
    if(parents_m.size() == 1)  
    {  
      ast::Instantiation* inst_obj = getInstantiation();  
      inst_obj->setPortTiming(conn_obj->name(),
                              traversingInfo.timing(),
                              file(), line());  
    }  
    if(parents_m.size() > 1)  
    {  
      throw error::BadProgramming(  
        "wrongful identifier in timing function",
        __FILE__, __LINE__);  
    }  
  }  
}
Appendix C

Coding Conventions

We have been using many coding conventions to give all C++ code written for the project a unified look. Example C.1 shows the commenting for an imaginary class with many of the coding conventions used in the project.

- We are using Doxygen [6] to automatically document the code. Comments for classes and functions are therefore written in a way that Doxygen recognizes. Quick comments for member functions are in the “.hh” (header) files and the detailed comments in the “.cpp” files.

- All names of member functions and variables in classes are written like this: firstSecondThirdWord. Member variables should end with “_m” to further accentuate that it is a member.

- Class names are written like member functions but also start with a capital letter, like FirstSecondThirdWord.

- All more complicated types used in a class are defined with typedef and made public at the top of the interface description in the header file. Their names should always end with “_t”.

- Names in enumeration types are all capital letters and names of namespaces are all lower-case.

- It is preferred that all classes define a printing function that print information about the object to a stream and don’t end with a linebreak. The classes that define a printing function should also be printable with the « operator.

- Standard containers and iterators are useful and quick. Use them!
Example C.1: Doxygen Commenting

```cpp
// ! Quick description
/*@ */

/*!
 */

/* |
 */

/* if EXTENDED_INFO
 */

/* |
 */

/* author Carl B.
 */

/* |
 */

/* endif
 */

class TheFooClass
{

public:

/// Enumeration test
enum mode {FIRST, SECOND, THIRD};

/// Type test
typedef std::list<std::string> listType_t;

private:

mode modeEnum_m; // !< The enumeration
listType_t listObj_m; // !< The list

public:

/// Default constructor
TheFooClass {};

/// Default destructor
~TheFooClass {};

/// Member function
bool setMode(mode query_mode) const
{
    return (modeEnum_m == query_mode);
}

};
```
Appendix D

An Emacs Mode for the NoGap$^{CL}$

To make it easier to edit the files in NoGap$^{CL}$ I made a new mode for the Emacs editor. It allows automatic syntax highlighting and indenting of the code and provides a menu with clever commenting and indenting functions. It is based on a tutorial made by Andrew Scott Borton [4] and then extensively modified with help from the GNU Emacs Lisp manual [22]. The lisp code creates a function named “ncl-mode” that activates the new mode. The complete code of the mode is included in Section D.2 in this appendix, mostly because it is hard to find good examples of simple Emacs modes.

D.1 Mode Features

D.1.1 Keybindings and Menu

The mode defines a local keymap of bindings that connect different key-combinations with functions. This map also defines the mode specific menu items as bindings to the fake function key “menu-bar”. The menu items are functions for commenting and uncommenting a region and indenting a region. The keymap is defined on lines 4-30. This map is then set as the local map with “use-local-map” in the “ncl-mode” function on line 272.

D.1.2 Syntax Highlighting

The keywords of the language and other bits of code that would benefit from being highlighted were grouped together in logical groups. I then made optimized regexps with the “regexp-opt” function to recognize the bits of code and bound them to different “font-lock-faces”. Lines 36-57 shows these regexp-“font-lock-face” bindings. Commenting is recognized by C++ style comments being included

---

1 An Emacs construct for defining a highlighting group that a colour can be applied to.
in what is called the syntax table of the mode. They are then bound to the
commenting “font-lock-face”.

### D.1.3 Indenting

All indenting is based on the braces and parentheses in the code. An opening one
increases the indenting width and a closing one decreases the width. This is done
by regexp searching forwards and backwards on the current and as many preceding
lines as needed, counting the braces and parentheses. The main indenting function
is written on lines 94-189. Lines 79-91 contains the function for indenting a whole
region. It indirectly uses the main indenting function.

### D.1.4 Commenting

For easy commenting of code I made a comment region function (lines 193-224)
and an un-comment region function (lines 226-268). They are accessed through
the menu or special keycombinations.

### D.2 Code

```lisp
;; ncl-mode.el --- Mode for editing NoGap Common Language files

(defun ncl-mode-hook ()
  (defvar ncl-mode-map
    (let ((ncl-mode-map (make-keymap)))
      ;; Keybindings
      (define-key ncl-mode-map \[tab] 'indent-according-to-mode)
      (define-key ncl-mode-map \[C-c\C-c] 'ncl-comment-region)
      (define-key ncl-mode-map \[C-c\C-u] 'ncl-uncomment-region)
      (define-key ncl-mode-map \[C-i] 'ncl-indent-region)

      ;; Making menu
      (define-key ncl-mode-map [menu-bar nogap] (cons 'NoGap (make-sparse-keymap 'NoGap)))
      '(menu-item 'Indent Line or Region' ncl-indent-region
        :help 'Indents the lines occupied by the region
        :keys 'C-i'))
      (define-key ncl-mode-map [menu-bar nogap uncommentregion] '(menu-item 'Uncomment Line or Region' ncl-uncomment-region
        :help 'Uncomments the comment that point is in
        :keys 'C-c C-u'))
      (define-key ncl-mode-map [menu-bar nogap commentregion] '(menu-item 'Comment Out Line or Region' ncl-comment-region
        :help 'Comments out the lines occupied by the region between the point and the mark
        :keys 'C-c C-c'))
      (defvar ncl-mode-map
        "Keymap for NCL mode")
    )))
```

SYNTAX HIGHLIGHTING

(defvar ncl-font-lock-keywords ncl-font-lock-keywords-2
  "Default highlighting for NCL mode.")

(defvar ncl-font-lock-keywords-2
  (append ncl-font-lock-keywords-1
    ';; This is added so entity names with underscores can be more easily parsed
    (modify-syntax-entry ?_ "w" ncl-mode-syntax-table)
    ;; Comment styles are same as C++
    (modify-syntax-entry ?/ "", 124b* ncl-mode-syntax-table)
    (modify-syntax-entry ?* "", 23* ncl-mode-syntax-table)
    ncl-mode-syntax-table))

SYNTAX table for ncl-mode")

;; INDENTING

(defun ncl-indent-region (start-point end-point)
  (interactive "r")
  (let (start-temp end-temp)
    (goto-char end-point)
    (end-of-line)
    (setq end-temp (point))
    (goto-char start-point)
    (beginning-of-line)
    (setq start-temp (point))
    (indent-region start-temp end-temp nil)
    (goto-char start-point)))
(defun ncl-indent-line ()
  "Indent current line as NCL code."
  (interactive)
  (beginning-of-line)
  (if (bobp)
      (indent-line-to 0) ; First line is always non-indented
    (let ((not-indent t) cur-indent count_1 count_2 limit1 limit2 prev-indent (backstep t))
      (progn
        (setq count_1 0)
        (setq count_2 0)
        (setq limit2 (point)) ; Finding first row limits
        (end-of-line)
        (setq limit1 (point))
        ;;(setq limit1 (re-search-forward "\n" nil t 1))
        (goto-char limit2)
        (save-excursion
          (while not-indent
            (progn
              (setq not-indent (re-search-forward "\"{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}\" limit1 t))
              (if not-indent
                (setq count_1 (+ count_1 1)))))
            (setq not-indent t)
            (goto-char limit1)
            (while not-indent
              (progn
                (setq not-indent (re-search-backward "\"{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}\" limit2 t))
                (if not-indent
                  (setq count_1 (- count_1 1)))))
              (setq not-indent t)
              (save-excursion
                (while backstep ; Looping to find indentation hint
                  (progn
                    (setq backstep t)
                    (forward-line -1)
                    (setq prev-indent (current-indentation))
                    (setq limit2 (point)) ; Finding next row limits
                    (setq limit1 (re-search-forward "\n" nil t))
                    (goto-char limit2)
                    (while not-indent
                      (progn
                        (setq not-indent (re-search-forward "\"{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}}{{}\" limit1 t))
                        (setq count_1 (+ count_1 1)))))))))
  (return))
(if not-indented
  (setq count_2 (+ count_2 1)))
(setq not-indented t)
(goto-char limit1)
(while not-indented
  (progn
    (setq not-indented (re-search-backward "\\(\\{\\}\\)\\)" limit2 t))
    (if not-indented
      (setq count_2 (- count_2 1)))
    (setq not-indented t))
  (goto-char limit2)

;; End conditions satisfied?
(if (or (> count_2 0) (< count_2 0))
  (progn
    (setq backstep nil) ; End search
    (if (>= count_1 0)
      (if (< count_2 0)
        (setq cur-indent prev-indent)
        (setq cur-indent (+ prev-indent (* count_2 2)))))
      (if (< count_2 0)
        (setq cur-indent (+ prev-indent (* count_1 2)))
        (setq cur-indent (+ (* (+ count_1 count_2) 2)))))
    (if (bobp)
      (progn
        (setq cur-indent 0)
        (setq backstep nil)))))) ; End search
    (if (< cur-indent 0) ; We can’t indent past the left margin
      (setq cur-indent 0))
    (if cur-indent
      (indent-line-to cur-indent)
      (indent-line-to 0))))))

;; COMMENTING

(defun ncl-comment-region (start-point end-point)
  (interactive "r")
  (let (temp-point-1 temp-point-2 search-value)
    (progn
      (goto-char start-point)
      (end-of-line)
      (setq temp-point-1 (point))
      (goto-char end-point)
      (end-of-line)
      (setq temp-point-2 (point))
      (if (eq temp-point-1 temp-point-2)
        (progn
          (beginning-of-line)
          (insert '/*' ))
        (progn
          (setq temp-point-1 nil)
          (setq temp-point-2 nil))
        (setq temp-point-1 temp-point-2))
      (progn
        (beginning-of-line)
        (insert '/*' ))
      (progn
        (setq temp-point-1 nil)
        (setq temp-point-2 nil))
      (setq temp-point-1 temp-point-2))
    (setq temp-point-1 temp-point-2))
  (progn
    (setq temp-point-1 nil)
    (setq temp-point-2 nil))
  (setq temp-point-1 temp-point-2))

;; END OF CL

(beginning-of-line)
(setq temp-point-1 (point))
(goto-char temp-point-2)
(setq search-value
  (re-search-backward
   "^[\t\n]\*" ; "^[\t\n]\*" temp-point-1 t))
(if search-value
  (progn
    (goto-char (+ search-value 1))
    (insert "*/")
    (progn
      (goto-char temp-point-1)
      (insert "*/")
    )
    (goto-char start-point)
    (beginning-of-line)
    (insert "*/"))))
)

(defun ncl-uncomment-region ()
  (interactive)
  (let (start-point temp-point-1 temp-point-2 search-value
         test-search-value)
    (setq start-point (point))
    (beginning-of-line)
    (setq temp-point-1 (point))
    (goto-char start-point)
    (setq search-value (re-search-backward '//' temp-point-1 t))
    (if search-value
      (progn ; Found single line
        (goto-char search-value)
        (delete-char 2)
        (goto-char (¬ start-point 2))
      )
      (progn ; Looking for */ delimiter
        (goto-char start-point)
        (setq test-search-value (re-search-backward '\"*/" nil t))
        (if (not test-search-value)
          (setq test-search-value 0)
        )
        (setq search-value (re-search-backward '\"*/" nil t))
        (if (and search-value (> search-value test-search-value))
          (progn ; Found beginning
            (goto-char search-value)
            (delete-char 2)
            (setq temp-point-2 (point)) ; Looking for */ delimiter
            (setq search-value (re-search-forward '\"*/" nil t))
            (if search-value
              (progn ; Found end
                (goto-char search-value)
                (delete-char -2)
                (goto-char (¬ start-point 2)))
            )
          )
        )
      )
    )
    (print "cannot find end of comment")
    (goto-char temp-point-2)
    (insert '/")
    (goto-char start-point))))
(progn
  (print "cannot find beginning of comment")
  (goto-char start-point)))

;; MAIN PROGRAM

(defun ncl-mode ()
  (interactive)
  (kill-all-local-variables)

  ;; Setting key bindings
  (use-local-map ncl-mode-map)

  ;; Setting syntax table
  (set-syntax-table ncl-mode-syntax-table)

  ;; Set up font-lock
  (set (make-local-variable 'font-lock-defaults)
       '(ncl-font-lock-keywords))

  ;; Register our indentation function
  (set (make-local-variable 'indent-line-function) 'ncl-indent-line)

  (setq major-mode 'ncl-mode)
  (setq mode-name "NCL")
  (run-hooks 'ncl-mode-hook)

  (provide 'ncl-mode)

;; ncl-mode.el ends here
Upphovsrätt

Detta dokument hålls tillgängligt på Internet — eller dess framtida ersättare — under en längre tid från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för icke-kommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmanens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnad som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydda mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida http://www.ep.liu.se/

Copyright

The publishers will keep this document online on the Internet — or its possible replacement — for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for his/her own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: http://www.ep.liu.se/