Examensarbete

Assembler Generator and Cycle-Accurate Simulator Generator for NoGap

Examensarbete utfört i Reglerteknik
vid Tekniska högskolan i Linköping
av

Faisal Akhlaq and Sumathi Loganathan

LiTH-ISY-EX--2010/4335--SE
Linköping 2010
Assembler Generator and Cycle-Accurate Simulator Generator for ΝοGap

Examensarbete utfört i Reglerteknik
vid Tekniska högskolan i Linköping
av

Faisal Akhlaq and Sumathi Loganathan

LiTH-ISY-EX--2010/4335--SE

Handledare: Per Karlström
ISY, Linköpings universitet

Examinator: Dake Liu
ISY, Linköpings universitet

Linköping, 27 May, 2010
System-on-Chip is increasingly built using ASIP (Application Specific Instruction set Processor) due to the flexibility and efficiency obtained from ASIPs. NoGap (Novel Generator of Accelerator and Processor framework) is an innovative approach for ASIP design, which provides the advantage of both ADL (Architecture Description Language) and HDL (Hardware Description Language) to the designer.

For the processors designed using NoGap, software tools need to be automatically generated, to aid the designer in programming and verifying the processor. As part of the master thesis work, we have developed two generators namely Assembler generator and Cycle-Accurate Simulator generator for NoGap using C++. The Assembler generator automatically generates an assembler, which is used to convert the assembly code written by a programmer into relocatable binary code. The Cycle-Accurate Simulator generator automatically generates a cycle-accurate simulator to model the behavior of the designed processor. Both these generators are static, and can be used to generate the tools for any processor created using NoGap.

In this report, we have detailed the concepts behind the generators, and the implementation details of the generators. We have listed the results obtained from running assembler and cycle-accurate simulator on a test processor created using NoGap.
Abstract

System-on-Chip is increasingly built using ASIP (Application Specific Instruction set Processor) due to the flexibility and efficiency obtained from ASIPs. NoGap (Novel Generator of Accelerator and Processor framework) is an innovative approach for ASIP design, which provides the advantage of both ADL (Architecture Description Language) and HDL (Hardware Description Language) to the designer.

For the processors designed using NoGap, software tools need to be automatically generated, to aid the designer in programming and verifying the processor. As part of the master thesis work, we have developed two generators namely Assembler generator and Cycle-Accurate Simulator generator for NoGap using C++. The Assembler generator automatically generates an assembler, which is used to convert the assembly code written by a programmer into relocatable binary code. The Cycle-Accurate Simulator generator automatically generates a cycle-accurate simulator to model the behavior of the designed processor. Both these generators are static, and can be used to generate the tools for any processor created using NoGap.

In this report, we have detailed the concepts behind the generators, and the implementation details of the generators. We have listed the results obtained from running assembler and cycle-accurate simulator on a test processor created using NoGap.
Acknowledgments

We thank Almighty immensely for everything.

We would like to express our sincere gratitude to our examiner Dake Liu and our supervisor Per Karlström for providing us opportunity to work in this thesis and to publish conference paper on the work done.

Many thanks to Per Karlström for his guidance, ideas and C++ mentoring throughout the thesis. We thank Dake Liu for his suggestions and permitting us to redraw the diagram from his book. We also would like to thank Wenbiao Zhou for clarifying our doubts in NoGap, and Ching-han Wang for explaining us the pioneer processor he had created as his master thesis using NoGap\textsuperscript{CL}. We also thank ISY staff Anders Nilsson, Thomas Johansson and Ylva Jernling for cheerfully helping us anytime with support issues.

We thank our master program co-ordinator Kristian Sandahl for providing us stepping stone in C++ programming through mandatory C++ courses under our master program. We also thank our professor Vivian Vimarlund whose courses served as foundation for us in report writing.

We thank everyone who had lent us helping hand to complete this thesis.

We are ever thankful to our family and friends who stand by us forever in good and bad times.
Contents

1 Introduction .................................................. 3
   1.1 Thesis Introduction ........................................ 3
   1.2 Purpose .................................................... 4
   1.3 Intended readers ........................................... 4
   1.4 Prerequisites ............................................... 4
   1.5 Structure of thesis ........................................ 4

2 ASIP Design and Software Tools ............................. 7
   2.1 ASIP ........................................................ 7
      2.1.1 ASIP Design ........................................... 8
      2.1.2 ASIP Design Automation ............................... 8
      2.1.3 Software tools for ASIP .............................. 9
   2.2 Assembler .................................................. 10
   2.3 Simulator .................................................. 11
      2.3.1 Bit Accurate ........................................... 11
      2.3.2 Cycle Accurate ........................................ 11
      2.3.3 Pin Accurate .......................................... 11
      2.3.4 Pipeline Accurate ..................................... 11

3 Related Work ................................................ 13
   3.1 Assembler Generator ....................................... 13
   3.2 Cycle-Accurate Simulator Generator ........................ 14

4 Background Theory .......................................... 17
   4.1 Graph Theory .............................................. 17
   4.2 Graphviz ................................................... 20
   4.3 Boost C++ Libraries ....................................... 21
   4.4 Boost Graph Library ....................................... 21
      4.4.1 adjacency_list ....................................... 21
      4.4.2 depth_first_search .................................... 21
      4.4.3 dfs_visitor .......................................... 22
      4.4.4 topological_sort ..................................... 22
   4.5 Flex and Bison ............................................ 22
      4.5.1 Flex .................................................. 23
      4.5.2 Bison ................................................. 23
## 5 NoGap Overview

5.1 NoGap Components ........................................ 27
  5.1.1 NoGap Common Description (NoGap\textsuperscript{CD}) .............. 27
  5.1.2 Spawners ............................................. 28

5.2 NoGap Common Language (NoGap\textsuperscript{CL}) ............... 28
  5.2.1 Functional Unit (FU) .................................. 29
  5.2.2 Mage FU .............................................. 29
  5.2.3 Mase FU ............................................... 30
  5.2.4 Parse Unit ............................................ 31

5.3 NoGap Connection Graph NCG ................................. 31
  5.3.1 Mage Graph ........................................... 31
  5.3.2 Variable Dependency Graph .............................. 31
  5.3.3 Mase Graph ............................................ 34

5.4 Uniqueness of NoGap ........................................ 36

## 6 NoGap Assembler Generator Implementation .................. 39

6.1 NoGap Assembler ............................................ 40

6.2 Instruction Format .......................................... 40

6.3 NoGap Assembler Generator (AsmGen) ......................... 42
  6.3.1 Definition file ......................................... 46
  6.3.2 Instruction table ....................................... 46
  6.3.3 Mnemonic aliases ....................................... 46

6.4 Assembler Program ........................................... 47

6.5 C++ Program - Assembler Driver .............................. 48
  6.5.1 Getting input from user .................................. 48
  6.5.2 Reading the definition file ............................... 49
  6.5.3 Controlling the Flex and Bison programs ................. 49
  6.5.4 Performing the second pass for the assembler ............. 49
  6.5.5 Generating binary output files ........................... 49

6.6 Lexical Analyser ............................................. 52

6.7 Parser ...................................................... 52
  6.7.1 Directive ............................................... 54
  6.7.2 Label .................................................. 54

6.8 Mnemonic Generation ......................................... 54

6.9 Output Files ................................................ 55

6.10 Error Handling .............................................. 55
  6.10.1 Error Handling by Bison Parser ......................... 56
  6.10.2 Error Handling by Assembler Driver Program ............ 56

6.11 Results ..................................................... 57

6.12 Conclusion ................................................ 59

## 7 Overview of NoGap Cycle-Accurate Simulator Generator ........ 61

7.1 Major Steps in implementation .............................. 62
  7.1.1 Step 1: Sequentialization of Mase graph .................. 64
  7.1.2 Step 2: Initialization of processor elements .............. 66
  7.1.3 Step 3: Write Execute methods ........................... 69
Contents

7.1.4 Step 4: Execute cycle .............................................. 69

8 NoGap Cycle-Accurate Simulator Generator Implementation 71
8.1 Simulator Generator ................................................. 71
  8.1.1 Generating Base Class ........................................ 73
  8.1.2 Generating PU Classes ......................................... 73
  8.1.3 Generating Multiplexer Class ................................ 74
  8.1.4 Generating Inline Expression Classes ....................... 75
  8.1.5 Generating Flip-Flop Class .................................... 75
  8.1.6 Generating Global Input Class ............................... 76
  8.1.7 Generating Global Output Class .............................. 76
  8.1.8 Generating Pass Node Class .................................. 77
  8.1.9 Generating Decoder Class .................................... 77
  8.1.10 Generating Pipeline Class Control Classes ............... 77
  8.1.11 Other methods of Simulator Generator ..................... 77
8.2 Simulator Mase Generator ......................................... 77
  8.2.1 SSP Graph Generation .......................................... 78
  8.2.2 Topological Sorted List Generation ......................... 87
8.3 Simulation Runner Class Generation .............................. 87
  8.3.1 Instantiating Objects ......................................... 88
  8.3.2 Initializing Port Sizes and Values .......................... 88
  8.3.3 Generating Execution Cycle ................................... 88
8.4 Simulator Driver .................................................... 89
8.5 Results ....................................................................... 89

9 Conclusion ................................................................... 97
  9.1 Limitations and Future Work ................................. 97
    9.1.1 Assembler Generator ......................................... 97
    9.1.2 Cycle-Accurate Simulator Generator .................... 98
Bibliography .................................................................. 99

A Code Excerpt .......................................................... 103

B Abbreviations ........................................................... 116
List of Figures

2.1 ASIP ................................................. 7
2.2 Automatic ASIP design flow (tool researcher’s view) .......... 9
2.3 Simulator Categories .................................. 12

4.1 Simple graph example .................................. 18
4.2 DFS Example Graph .................................. 20
4.3 Flex and Bison ....................................... 23
4.4 Assembler using Flex and Bison ......................... 26

5.1 NoGap System Architecture .............................. 28
5.2 NoGap Mase graph ..................................... 33
5.3 NoGap variable dependency graph ....................... 34
5.4 Mase graph node class hierarchy ...................... 35
5.5 NoGap Mase graph ..................................... 37

6.1 NoGap assembler flow .................................. 41
6.2 Instruction format example ............................... 42
6.3 Overview of NoGap Assembler Generator .................. 42
6.4 NoGap AsmGen Mechanism ............................... 43
6.5 NoGap AsmGen serialization data structure ............. 44
6.6 Writing definition file using serialization ................. 44
6.7 NoGap AsmGen internal data structures and generated files .... 44
6.8 NoGap AsmGen Flow Chart ............................... 45
6.9 Assembler Program Structure ............................ 48
6.10 Assembler Driver Containers ........................... 49
6.11 Assembler Driver Flow Chart ........................... 50
6.12 Assembler Generator Class diagram .................... 51
6.13 Lexical Analyser generated using Flex .................. 52
6.14 Parser generated using Bison ......................... 53

7.1 Example for pipeline ................................... 62
7.2 Simple Mase graph ..................................... 63
7.3 Simple Mase graph with Flip-flops and Multiplexers ........ 64
7.4 Source Sink Pass node creation from FU .................. 66
7.5 Mase graph transformed to SSP graph .................... 67
7.6 SSP graph with order of execution ....................... 68
7.7 Class diagram of FU ..................................... 68

8.1 Simulator Generator Overview ........................... 72
8.2 Simulator Generator generated classes .................... 73
8.3 NoGap Simulator Mase Generator flowchart ............... 78
8.4 Variable Dependency Graph of regfile FU ................ 80
8.5 Variable Dependency Graph of regfile FU after removing cy edges .... 80
8.6 Variable Dependency Graph of sub FU ................... 81
8.7 Adder and Subber FUs and Flip-Flop 4 and 5 .............. 82
8.8 Adder and Subber Pass node and Flip-Flop 4 and 5 Source nodes . 83
8.9 Multiplexer to Pass conversion flowchart . . . . . . . . . . . . . . . 85
8.10 Multiplexer node . . . . . . . . . . . . . . . . . . . . . . . . . . . 86
8.11 Multiplexer Pass node . . . . . . . . . . . . . . . . . . . . . . . 86
8.12 Flip-Flop Sink nodes . . . . . . . . . . . . . . . . . . . . . . . . . . 87
8.13 NoGap Mase graph of test processor . . . . . . . . . . . . . . . 91
8.14 SSP graph of test processor . . . . . . . . . . . . . . . . . . . . . 92
8.15 Class diagram for the C++ classes generated through SimGen . . 94
Chapter 1

Introduction

1.1 Thesis Introduction

With the rapid development in Integrated Chip industry, VLSI design is becoming more sophisticated and it is possible to manufacture a complete system on a single silicon chip. Engineers tend to design more advanced products and processors, where larger number of transistors are packed tightly in small space with complex interconnections. This tends to make design of the processor a challenging approach which demands more time. Designers have to meet the changing market needs by designing flexible processors for various applications. Design efficiency is directly related to quick manufacturing of the device and fast time to market. [1]

To make the design process versatile, various ADLs (Architecture Description Language) [29] [3] and HDLs (Hardware Description Language) are available in the market. ADLs are used to describe hardware and software architecture of a system. LISA [27], uML [7], MIMOLA [21], ArchC [29], ASIP meister [28] are few ADL tools to mention. HDL describes the hardware of digital systems. VHDL [3] and Verilog [32] are the well known HDLs. Designers can choose either an ADL or a HDL to design a processor. ADL tools are easy to use, but come with a predefined architecture template. This limits the flexibility of the design process and the designer has to fit the new processor into the existing architecture template, and thereby compromise the original design he/she had in mind. On the other hand, HDL provides the designer with flexibility in designing the processor with complete support for tasks like register forwarding, pipeline control. At the same time, using HDL languages increase the complexity and the designer has to be very careful to avoid errors. NoGap strikes a balance between these two approaches, by not providing a predefined architecture but supporting pipelined instruction control architecture. Hence the designer can create any kind of processor with support to handle complex tasks. NoGap is used to automate the design of ASIPs and provides the advantages of both ADL and HDL to the designer. [15]
1.2 Purpose

The purpose of the thesis is to develop generators that automatically generate the tools assembler and cycle-accurate simulator for any processor created using \texttt{NoGap}. From the user perspective, these tools will assist the designer in testing the processor before it is taped out.

1.3 Intended readers

This thesis report will assist researchers who develop a processor construction framework to understand how to design the assembler and cycle-accurate simulator generators. Programmer who will develop these tools, can benefit from the report by understanding the implementation of the generators.

The report includes an introduction to \texttt{NoGap} which might be of interest to researchers.

1.4 Prerequisites

Familiarity with digital signal processor design and VHDL or verilog is preferred but not mandatory, to facilitate the overall understanding of the project. Being ’Software Engineering” students, we had attempted to explain the hardware part of \texttt{NoGap} as we had understood during the thesis. The reader can look into the book Embedded DSP Processor Design by Dake Liu [23] to learn more about processor design in detail.

Knowledge of C++ programming is essential for the reader of the thesis to understand the implementation of the generators. The C++ Programming Language book by B. Stroustrup [31] can serve as a good introduction to C++ or to refresh the C++ knowledge. Being 'Software Engineering’ students, we had attempted to explain the hardware part of \texttt{NoGap} processor as we had understood in the few months spent for the thesis. The reader can look into the book ‘Digital Signal Processor’ [23] to know about processors in detail.

1.5 Structure of thesis

The thesis report is organized into nine chapters.

**Chapter 1** provides a general introduction to the thesis along with intended readers and prerequisites to understand the thesis.

**Chapter 2** gives brief introduction to the reader about ASIP design and software tools needed for ASIP user. This is to provide the reader with the background theory of the thesis.
Chapter 3 presents the related work on tools similar to NoGap and the assembler and simulator generators that were already developed in research field.

Chapter 4 briefly introduces the NoGap system and NoGap^CL.

Chapter 5 Basic graph theory, boost C++ graph libraries used in the thesis, and a short explanation of the software tools flex and bison in general are explained in this chapter.

Chapter 6 explains the overview and C++ implementation of the NoGap assembler generator, with explanation of all the programs used to generate the assembler and the results achieved.

Chapter 7 In this chapter, the overview of NoGap cycle-accurate simulator generator (or simgen) is discussed using a simple example graph.

Chapter 8 is the explanation of cycle-accurate simulator generator implementation in C++, with description of the results achieved in simulation.

Chapter 9 presents the conclusion, discussion, limitation and the future work of the thesis.
Chapter 2

ASIP Design and Software Tools

NoGap is used to design ASIP. In this chapter ASIP, design of ASIP, automation of ASIP design and software tools that aid ASIP designer and user are discussed. Basic functionality of the assembler and simulator is explained at the end of the chapter.

2.1 ASIP

Based on the programmability and flexibility, ASIP (Application-Specific Instruction-set processors) fall in-between Microprocessors and ASIC (Application-Specific Integrated Circuit) as in 2.1. Microprocessors are flexible and can be programmed for any kind of application. ASIP’s instruction set is designed for a particular application domain and is programmable for a specific domain. An ASIC is fixed for an explicit application and can not be programmed [35].

![Figure 2.1: ASIP](image-url)
An ASIP is derived from a general purpose processor by adding instructions specific to the application domain. Its instruction set is targeted to the most used functions in the specific application domain unlike general purpose instructions which supports wide applications. One way to implement such specific instructions in the hardware is by the use of runtime reconfigurable units [23, 24].

2.1.1 ASIP Design

We aim to provide a brief introduction to ASIP design as a prelude to our master thesis, and for more detailed description of ASIP design, the reader is referred to [23].

The design of ASIPs varies from general purpose processors in the emphasis placed on performance, power consumption and hardware cost. ASIPs aim for high performance in specific application domains at low hardware cost and low power consumption, while providing flexibility through programmability [23]. ASIP is an intermediate solution between general purpose processors and ASIC. General ASIP design includes stages namely analysis of the application domain (The target application is analyzed to understand the requirements and specifications of the application), Architectural design space exploration (Suitable architecture for the application is explored based on the application analysis, which will also fit constraints like cost, performance and power consumption), Generation of Instruction set (Instruction set apt for the application and architecture is decided), Code synthesis (includes generation of object code from architecture template and instruction set, generators for compiler, assembler and simulators) and Hardware synthesis (design and construction of microarchitecture) [13].

2.1.2 ASIP Design Automation

Dake Liu in his book [23] explains the automation of the ASIP design process as below.

To overcome disadvantages in manual ASIP design, tools are employed for designing ASIP automatically. The automation process involves three major steps namely architecture exploration, ADL specification, generations and verifications. There are many profilers in the market to decide the architecture and assembly instruction set in the architecture exploration phase. Once an architecture is decided, an ADL is specified to model the instruction set and architecture. This is a crucial phase, as the type of ADL decides the efficiency in modeling the instruction set and the microarchitecture. The ADL should be able to support all the information about the architecture, and should not be too complicated for the designer’s use. The main purpose of NoGap is to balance this aspect and is explained more in chapter 5 on NoGap. The last phase includes generation of software tools namely Compiler, Assembler, Simulator, and finally verification/testing the ASIP design. Figure 2.2 from [23] illustrates ASIP design automation.
For, NoGap we have written C++ programs to automatically generate Assembler and Simulator. This C++ development of the two generators constitutes our master thesis.

Figure 2.2: Automatic ASIP design flow (tool researcher’s view)

2.1.3 Software tools for ASIP

The purpose of software tools for the ASIP user is to write programs for the processor and verify if the designed processor works according to the instructions written in the high level language or assembly language. If a high level language is used to code the source program, a compiler is normally used to convert the program to assembly language. The other option is to manually translate the source program to assembly language. In this later option, the programmer must be careful to avoid errors. An Assembler converts the assembly program to binary code, which can be loaded directly into the memory location. A Simulator is
used to execute the binary code modeling the actual processor, and thus enables the programmer to debug and verify if the processor works or not as expected [23].

From the ASIP designer’s perspective, the purpose of software tools is categorized into three types namely Code Analysis, Code Generation, Code Modeling, and these names explain the purpose of the tools. In this section, emphasis is given to the tools employed in \texttt{NoGap} namely Assembler and Simulator. Out of scope tools like Semantic Analyzer, Compiler, Profiler, Linker and Debugger are not explained in detail. Reader is referred to [23] to learn more on these topics.

Code Analysis is done using a Lexical Analyzer and a Syntax Analyzer, and this analysis also constitutes the initial part of Assembler development. Lexical Analyzers or Scanners read the input program from left to right and groups the characters into lexical tokens. These tokens are used as input to Syntax Analysis. We have used Flex [10] to perform Lexical analysis in \texttt{NoGap}. The Syntax Analyzer or parser consists of grammar rules made of lexical tokens to verify if the input program conforms to the grammar of the language in which the program is written, and generate parse tree which will serve as input to the semantic analyzer or other tools. \texttt{NoGap} doesn’t need semantic analyzer for the language \texttt{NoGap}^{CL} as the grammar used in \texttt{NoGap}^{CL} is context-free. Bison [11] is the parser generator used in \texttt{NoGap} for syntax analysis. Flex and Bison are explained briefly in Chapter 4.

As mentioned previously, an Assembler is used to convert the assembly code into binary code. A Simulator is used to model the behavior of the processor. User can execute the code when the hardware of the processor do not exist by using the simulator.

\section{2.2 Assembler}

The theory presented here explains a two pass assembler. \texttt{NoGap} assembler is a two pass assembler. Different types of assemblers are explained in [4].

An assembler takes an assembly code file as its input and generates an object file. The output object file from the assembler is not a complete binary file. The output file contains binary code with some other information as well. The translation is performed in two steps. The object file generated through assembler is not executable because of reference to variables that are in other files. Therefore the assembler sends its output with information of reference to the linker which links it to the libraries and forms an executable file [23].

Two pass assembler performs its translation task in two steps. In the first step, it reads the input assembly program line by line. Lexemes are generated for each instruction. During the first pass if the assembler comes across a label it saves the label with its address into a symbol table. During the second pass the assembler
takes first pass data and converts all the instructions into their equivalent binary. If the symbol table contains a reference to a external symbol and their address is not in the symbol table then they are left unresolved [23].

2.3 Simulator

Simulator theory is explained from Dake Liu’s book on Embedded DSP Processor Design and for further explanation the reader is directed to his book [23]. Simulator for assembly language are of two types namely instruction set simulator(ISS) and process architecture simulator.

"Bit and Cycle accurate assembly language execution can be exposed by using an instruction set simulator. This will not even need linking to the actual hardware implementation. A processor architecture simulator is the executable hardware behaviour exposing implementation details and the pipeline and bus transactions accurate assembly language execution." [23]

2.3.1 Bit Accurate

"Bit accurate means the outputs of the ISS to data memories and to registers is exactly the same as the outputs to data memories and registers in the hardware core." [23]

2.3.2 Cycle Accurate

"Cycle accurate means the clock cycle consumed by running instructions including running branch instructions, handling interrupts, and handling I/O ports is the true number of clock cycles." [23]

2.3.3 Pin Accurate

"Pin accurate means that input and output to and from each pin of the simulator is the same as the input and output to and from each pin of the processor RTL code." [23]

2.3.4 Pipeline Accurate

"Pipeline accurate means that the execution of architecture simulator and the execution of RTL code are exactly synchronous on the machine clock. That is, all registers and memories get data and send data at exactly the same time (clock cycle) in both the architecture simulator and RTL code. However, all accuracies are compared and required on architecture level instead of microarchitecture level, meaning operations inside modules are not exposed and compared." [23]

Figure 2.3 shows few types of simulators and an example of a commercial product available under that category. As we move from top to the bottom the details
increase but the speed decreases.

<table>
<thead>
<tr>
<th>Name</th>
<th>Example</th>
</tr>
</thead>
<tbody>
<tr>
<td>Functional Simulator</td>
<td>Virtutech Simics</td>
</tr>
<tr>
<td>Cycle Accurate Simulator</td>
<td>NoGAP Simulator</td>
</tr>
<tr>
<td>Event Driven Simulator</td>
<td>Modelsim</td>
</tr>
<tr>
<td>SPICE Simulator</td>
<td>ngSpice</td>
</tr>
</tbody>
</table>

**Figure 2.3:** Simulator Categories
Chapter 3

Related Work

3.1 Assembler Generator

A number of tools such as LISA [37], EXPRESSION [9], nML [7], MIMOLA [20], ArchC [29], and ASIP Meister [12], are tools that support processor design. All of these tools however force a designer into a predefined template architecture. On the other end of the spectrum of design tools are HDLs such as Verilog, VHDL or SystemC [25]. These tools however require manual handling of all minuscule details of an RTL design. NoGap offers a unique trade off between these two extremes. No template design is assumed but support is given for managing details regarding pipelined instruction controlled architectures.

The ISDL assembler described by George Hadjiyiannis et al in [8] works much in the same way as AsmGen, where a tool was developed, that when given an ISDL description of a processor generates an assembler for it. The generator tool for ISDL assembler generates new lex and yacc files for each new processor.

S.Kumari [19] developed a similar assembler for Sim-nML language as her master thesis, in which a tool called asmg takes processor model in its intermediate form (IR) and generates an assembler. The IR is generated by another tool irg. The asmg tool generates the two pass assembler consisting of lex, Yacc and keyword files.

Sim-nML assembler works very similar as the ISDL assembler and they both differ from NoGapAsm in the way that NoGapAsm has a stable lexical analyzer and parser, i.e. they are not generated with AsmGen. Rather they use a definition file generated from AsmGen.
3.2 Cycle-Accurate Simulator Generator

The Cycle-Accurate simulator designed in [34] can execute multiple operations in a single cycle. The simulator can execute instructions either from an assembler or the instructions produced from another processing element. For this simulator, the order of execution of operations within the instruction does not matter. The simulator performs read at the beginning of the cycle and perform write at the end of the cycle. A special port called the *instruction*, controls the simulation. When the cycle starts the value written to this instruction port is the instruction to be executed.

Lsimpp described in [33] is a cycle-accurate, multiprocessing simulator based on LISA ADL. Lsimpp consists of the following concepts:

- *Automatic Tool Generation from ADL*
- *Processor Model System Architecture*
- *Interconnect Modules*

Lsimpp is targeted as a cycle-accurate simulator for multi-core SoC. The simulator can be invoked in GUI or batch mode.

Various standard buses are integrated through interconnect. These interconnects are modeled through System-C. For integration of user defined modules, a C++ library is used. A tool chain including C-compiler, assembler, linker and few other tools are generated automatically and these tool chain provide the wrapper modules for the simulation. Wrapper includes the kernel functions and bus adaptors. The modules are connected through bus and memory modules. Different simulator parts and the generated wrappers are compiled and linked to the simulator. For the simulator to be a multiprocessor simulator, external shared memory is used. An arbiter is used to handle simultaneous requests from different processors in the same clock cycle.

Sleipnir (described by Tor E. Jeremiassen in [14]) is a generator tool for writing instruction-level simulators. Sleipnir also supports to generate cycle accurate simulators for most embedded processors. Goals of Sleipnir simulator generator are:

- Writing simulators for different architectures easily
- Generated Simulators should provide statistics, profiling and timing information

Sleipnir has generated simulators for different architectures including cycle accurate simulation. Another important aspect of simulators, generated through Sleipnir is the portability to different host platforms [14].
For simulation, Sleipnir generates C source files and some functionality is provided in libraries. A pre-decode mechanism is used by the generated simulators. The target instruction is decoded once and stored in a C structure. The C structure that stores the intermediate representation of the instruction is known as \textit{target instruction descriptor (TID)} [14]. Pre-decode mechanism adds to the speed of Sleipnir generated simulators. A \textit{semantic function} implements the semantics of the instruction and these semantics are bound to the instructions TID. The main simulation loop consists of four steps[14]:

1. Computing the TID address of the target instruction
2. Execution of C code copied from machine description
3. Instruction Dispatch
4. Execution of C code also copied from machine description

Stefan et al. in [27] describe LISA language and the simulator developed for it. They state that it is not possible to produce cycle accurate simulators for nML [7]. LISA language has some similar ideas as that of nML but has better features in different areas such as, support of compiled simulation techniques. To generate the compiled simulators, \textit{LISA features conditional structures on the operation level that evaluate at compile time} [27].

A complex processor (TMS320C6201) was modeled using LISA language and the model was realized in less than two months. An environment of retargetable development tool was developed, configured by LISA descriptions. A retargetable compiled simulator was implemented. The simulator was generated through an intermediate data base. The intermediate database comes from a parser that reads the LISA models and translates them into an intermediate database. Stefan Pees in [27] mentions that the translation of the complex processor (TMS320C6201) model into a simulator took less time and the simulator was successfully verified.

A common trait for all the simulator generation tools described in this section is that they all start from some form of sequential instruction description and from that they generate the simulator. \textit{NoGap} is different in that it starts from parallel descriptions of leaf modules and a parallel hardware multiplexed data path graph containing functionality for all instructions. None of the previous works have presented the techniques needed to generate a cycle accurate simulator from an inherently parallel description language.

One possible solution would have been to use a discrete event simulator, however discrete event simulators are ineffective compared to simulating an entire cycle at a time, but the simulated architecture can then not contain any combinational loops. The simulator described in this report assumes an architecture free from combinational loops and can as such be cycle based.
Chapter 4

Background Theory

In this chapter, the theory behind graphs is explained shortly, so that is easy for the reader to understand the graphical representation of the NoGap design.

Introduction to software tools graphviz, Flex and Bison is presented and the important boost C++ graph libraries used in our programming are detailed.

Terms related to NoGap namely NoGap\(^{CD}\), Mase graph are explained in Chapter 5.

4.1 Graph Theory

A Graph is an abstract form of a problem to be solved. Many applications have the pattern of nodes, and arcs which connect these nodes. Worldwide Web, Electronic circuits, data flow in computer are few examples of problems that can be abstracted in graph form.

NoGap\(^{CD}\) is primarily graphs, and the cycle-accurate simulator generator we had developed is based on the graph, which is a copy of the Mase graph. In this section we aim to cover the basics of graph theory from [22] and [30], which is essential for the reader to understand the implementation of the simulator. Readers who have graph theory knowledge can skip this section, as it primarily deals with basic terms in graph.

**Graph** Mathematically a graph can be defined as \( G = (V, E) \), where \( V \) is set of vertices or nodes, and \( E \) is set of edges or arcs. \( V \) is a finite set and \( E \) is the binary relationship on \( V \). An edge is a pair \((x, y)\) where \( x \) and \( y \) are the vertices that the edge connect. A simple example graph is shown in Figure 4.1

In this example graph, vertices are \( V = a, b, c, d, e, f \)

Edges are \( E = (a, b), (a, c), (a, d), (c, e), (c, f) \)

the edges are named as \( g, h, i, j, k \) respectively in the example diagram.
Graph is $G = (V, E)$

**Directed Graph** A graph is called directed graph or digraph, when the all edges between any two vertices are directed i.e. each edge has a source vertex and target vertex. All the edges are ordered pairs, i.e. for every edge $e = (x, y)$, $x$ is the tail vertex or source vertex and $y$ is the head vertex or target vertex. The edge $e$ is directed from its tail to the head vertex. The example graph is a directed graph. The direction is indicated by arrow.

**Undirected Graph** In undirected graph, all the edges are unordered pairs. Here the connection between two vertices doesn’t have any specific direction, i.e the edge can be traversed from either direction. Undirected graph can not contain self loops. An edge is called self loop when its origination and destination vertices are the same.

![Simple graph example](image)

**Adjacent Vertex** A vertex $x$ is adjacent to another vertex $y$, if both the vertices are connected by an edge and the vertices $x$ and $y$ are called neighbors. In the example, vertex $c$ and $e$ are adjacent vertices.

**Adjacent Edge** Two edges are called adjacent edges if they have a connection to a common vertex. In the example graph, edges $j$ and $k$ are adjacent to each other.

**In-Edge** In a directed graph, for the vertex $y$, the edge $(x, y)$ is the in-edge. Example - $h$ is the in-edge of vertex $c$.

**Out-Edge** In a directed graph, for the vertex $x$, the edge $(x, y)$ is the out-edge. Example - $k$ is the out-edge of vertex $c$.

**Degree** The degree of a vertex is the number of edges connected to it. Degree of vertex $a$ is 3 in the example.

**In-Degree** In a directed graph, the number of in-edges to a vertex $x$ is called in-degree of $x$. In degree of vertex $c$ in example is 1.
Out-Degree In a directed graph, the number of out-edges from a vertex x is called out-degree of x. In degree of vertex c in example is 2.

Path Path P, in a non-empty graph can be defined as (V,E), where V = a0, a1, a2, ..., an

and E = a0a1, a1a2, a2a3, ..., an-1an, where the vertices a0 to an are linked by path P in sequence. The target vertex of each edge is the source vertex of the next edge in the path. In the example, (a, c), (c, e) is a path.

Cycle When the start vertex and end vertex in a path are same vertex, and there are no repeated edges, the path is called a Cycle.

Acyclic Graph If there are no cycles in any of the paths, the graph is called Acyclic.

Tree A path between two vertices which do not have any cycle forms a tree.

Root Vertex A vertex on the top of the tree without any in-edges or ancestors is called root vertex.

Leaf Node A node in the tree which do not have out-edge or child node is called leaf node.

Forest Union of trees in a graph is called forest.

Attributes The example graph shown is a high level abstraction where details of node and edge are not defined. To make this graph usable to solve a problem, the nodes and edges are assigned attributes. Color is a common attribute assigned to both edges and nodes. Mathematically vertex or edge attribute is a function from vertex or edge set to set of possible attribute values.

In NoGap, attributes are assigned through various classes for different types of nodes and edges.

Adjacency List Representation of Graph

Adjacency list is one of the data structures used to define the way the graph is stored for computation within programs. In an Adjacency list, the vertices of the graph are stored as a list. Each vertex contains a linked list of its adjacent vertices. Details of the edges are not stored in the adjacency list. For the simple graph depicted in the example, adjacency list can be given as below

a -> b, c, d
b
c -> e, f
**Depth-First Search (DFS) Algorithm** DFS algorithm is one of the graph search algorithms which traverses through the graph. DFS visits all the vertices in a depth-first manner i.e., whenever DFS has to visit a new vertex in the search tree, the next deeper adjacent vertex is chosen, until there are no more vertices to visit. When there are no more vertices to visit, the algorithm backtracks to the previous vertex and continues the depth first search of other unvisited nodes. This first forms a depth-first tree and these trees together form depth-first forest. Illustration of Depth first search on the example graph is shown in Figure 4.2. The number on the arrows indicates the order of search beginning from start vertex a.

The order in which vertices will be traversed is a - b - a - c - e - c - f - c - a - d

Among these visited vertices, the finished vertices and their order is b - e - f - c - d - a.

**Figure 4.2:** DFS Example Graph

**Topological sort** Depth first search is used to do topological sort on a directed acyclic graph (DAG). In topological sort, the nodes are ordered linearly, such that for any edge(i,j), the node i comes before the node j in the ordered graph. If the graph has any cycle, it can not be topologically sorted. Topological sort re-arranges the graph in such a way that the nodes are placed according to the precedence of their execution.

### 4.2 Graphviz

The open source graph visualization software *Graphviz* is used in the *NoGap* project to visualize graphs. The *dot* language is used to draw directed graphs and *dotty* is the graph editor which is used to view and edit the graphs. More details about graphviz can be found in [2].
4.3 Boost C++ Libraries

Boost from [5] are peer-reviewed, portable open source C++ libraries, which are written for wide range of applications. Boost C++ libraries are aimed at performing advanced tasks in C++ and can be used with C++ STL libraries. Boost creators claim that the boost libraries increase productivity, reduce bugs, save time and costs in re-writing code. Boost works on most of the modern operating systems and are available for both commercial and non-commercial use.

NoGap makes use of numerous libraries provided by boost for efficient programming. Boost graph library is the foundation of programming in NoGap where NoGapCD is described in terms of graphs. The boost graph library classes which we have used in our thesis are briefly described in the following section.

4.4 Boost Graph Library

Boost Graph Library (BGL) [30] provides support to represent graphs that can be used in software development. BGL has a user-friendly interface to access, traverse and utilize the graph algorithms and data-structures.

4.4.1 adjacency_list

adjacency_list is a BGL class that is used to implement the adjacency list graph data structure. It is a template class with below format:

adjacency_list<OutEdgeList, VertexList, Directed, VertexProperties, 
                EdgeProperties, GraphProperties, EdgeList>

The OutEdgeList is the parameter for the container of edges of each vertex in the graph. VertexList defines the container for the vertices of the graph, and EdgeList that of edges in the graph. Directed is the option to create the graph as either directed or undirected or bidirectional. As the name implies, VertexProperties, EdgeProperties and GraphProperties define the properties of nodes, edges and graphs respectively. There are default values available for these parameters.

adjacency_list class provides functions namely add_vertex(), remove_vertex(), add_edge(), remove_edge() to add and remove vertices and edges. Functions vertices() and edges() give the list of all vertices and edges in the graph.

4.4.2 depth_first_search

depth_first_search function implements the DFS algorithm as described in graph theory. The algorithm keeps track of the vertices by using different colours. The vertices yet to be visited are marked white, vertices which have adjacent vertices to be discovered are marked gray, and the vertices that are discovered and do not
have any undiscovered neighbors are marked black. A color-map and vertex index
map are provided as input to this algorithm in addition to the graph on which the
depth first search is performed.

4.4.3 dfs_visitor

Visitors in BGL are similar to functors in STL. Visitors are used to extend the
functionality of a graph algorithm, by adding the steps necessary for the developer.
Each graph algorithm has different event points and the respective visitors have
corresponding methods. Among the different visitors available in BGL, we had
used dfs_visitor in association with the depth_first_search graph algorithm.

A class is defined extending the dfs_visitor, and the object of this class is passed
to the depth_first_search algorithm. Methods are defined within dfs_visitor
which can be used to perform additional functionality that the developer wishes
for. Among the available methods namely initialize_vertex, start_vertex, dis-
cover_vertex, examine_edge, tree_edge, back_edge, forward_or_cross_edge, fin-
ish_vertex, we have used examine_edge and finish_vertex. These methods are
in-turn the event points for the algorithm. Examine_edge method is called whenever
the search algorithm finds an out-edge from a vertex, and finish_vertex is
called for every finish vertex encountered in the depth first search.

4.4.4 topological_sort

The template class topological_sort of BGL, performs the topological sort as ex-
plained in the graph theory. The class takes the directed acyclic graph and an
output iterator as template parameters. The resulting vertex list of the sorted
DAG graph are stored in the output iterator in reverse topological order.

4.5 Flex and Bison

In the Assembler developed for NoGap, Flex [10] is used as Lexical Analyzer gen-
erator and Bison [11] as Parser generator. The C code generated from Flex and
Bison is used for actual lexical analysis and parsing.

Lexical analysis is the first step where input character stream is converted into
tokens. An instance of token is called lexeme. The tokens are used by parser to
take actions based on the grammar rules. Usage of Flex and Bison together is
illustrated through a simple example of printing opcode values for mnemonics in
a assembly language. The input file will contain mnemonics and the output is the
value of opcode for each of these mnemonics.

A brief overview of Flex and Bison is presented below. We do not aim to cover
complete explanation of these two tools. The reader is referred to the Flex [10]
4.5 Flex and Bison

4.5.1 Flex

Flex (fast lexical analyzer) is a tool used for creating a program that matches pattern in a given text. Using Flex, a program called scanner or tokenizer is created, which recognizes patterns in the user’s input file. In the Flex program, rules are created as pairs of regular expression and C code. When the executable of the Flex program is run against the user’s input text, patterns matching the regular expressions are identified and for a matching pattern found, the corresponding C code in the rule is executed.

Flex program consists of three parts namely definitions, rules and user code, each of them separated by a line with the symbol `%%`. Definitions section consists of name definitions and declaration of start conditions. Rules section consists of pair of regular expression (or pattern) and the corresponding action to be taken, which is specified using C code. The user code is optional, and if present, the code is copied verbatim to the output without any change.

Listing 4.1 shows a simple Flex code for the example stated above. Variables are declared in the definition section, pattern matching is done in rules section. The file `new.tab.h` is used by Flex to communicate with Bison about details of the tokens. For each unique mnemonic in the example assembly language, a token is returned by Flex to Bison. In this example, any character sequence other than the mnemonics will be written to the output stream without any action executed against them.

4.5.2 Bison

Bison is a tool used to convert any context free grammar into a LALR(1) (Look Ahead Left-to-right Rightmost derivation) parser or a GLR (Generalized Left-to-right Rightmost derivation) parser for that grammar. A context free grammar (CFG) consists of set of terminal symbols (or tokens), non-terminal symbols, production rules and a start symbol. Terminal symbols are the character sequence that occur in the actual input program. Non-terminal symbols function as placeholders for terminal symbols. Production rule defines non-terminal symbols in terms of other non-terminal symbols and terminal symbols. One of the non-terminal symbol is marked as a start symbol, and the validation of the language begins from this symbol. For example, an essay in English language is composed of paragraphs. Paragraphs are made of sentences and sentences are made of words conforming to English grammar. Here essay is the start symbol and paragraph, sentence are
Listing 4.1: Simple Flex example

```
 %{ 
 #include "new.tab.h"
 extern YYSTYPE yylval;
 }%

 // examples of non-terminal symbols. Each word is terminal symbol or token.

 A Bison file consists of three sections separated by %%. The first section consists of the Bison declarations and also the declaration of variables used in the C or C++ code written within Bison. The second section consists of grammar rules and the last section consists of the code user wants to use. Only the rules section is mandatory, and other two are optional. Each rule has a action written against it, which can performs the action specified by the user.

 Listing 4.2 shows a simple Bison code for the example stated above. The variable declarations are enclosed within %{ and }%. Bison declarations for the tokens are listed next in declaration section. Only these tokens can be used by Flex which is used with Bison. The rules section begins with the rule that the non-terminal symbol `input` is composed of instructions. Input is the start symbol. `Instruction` is another non-terminal symbol which is composed of the tokens (or terminal symbols). Against each terminal symbol in the rule, action is specified to print the value of opcode for the corresponding mnemonic. Codes section of Bison consists of C code to print any error message and to open the input file containing the input mnemonics.

 The concept of basic example illustrating usage of Flex and Bison together is used in assembler generator. Figure 4.4 explains the basics of assembler generator, which is explained in detail in Chapter 6.
Listing 4.2: Simple Bison example

```c
#include <bitset>
#include <iostream>
int yylex();
void yyerror(const char* s);
extern int yylex();
extern FILE* yyin;
%

%token TT_MOVE_WRITE TT_LOAD_WRITE TT_ADD2_ADD_WRITE
    TT_ADD_ADD_WRITE TT_ADD_SUB_WRITE TT_ADD2_SUB_WRITE
    TT_NOP TT_IO_OUT TT_JUMP_0 ALWAYS
%
input:
    | input instruction
    ;

instruction: TT_MOVE_WRITE   { std::cout << "Op Code : "
    | TT_LOAD_WRITE    { std::cout << "Op Code : "
    | TT_ADD2_ADD_WRITE { std::cout << "Op Code : "
    | TT_ADD_ADD_WRITE { std::cout << "Op Code : "
    | TT_ADD_SUB_WRITE { std::cout << "Op Code : "
    | TT_ADD2_SUB_WRITE { std::cout << "Op Code : "
    | TT_NOP          { std::cout << "Op Code : "
    | TT_IO_OUT       { std::cout << "Op Code : "
    | TT_JUMP_0 ALWAYS { std::cout << "Op Code : "
    ;
%
void yyerror(const char* s)
{    printf ("%s\n", s);
}

bool set_assembler_input_file(const std::string& fileName)
{    return (yyin = std::fopen(fileName.c_str(), "r");
}
Figure 4.4: Assembler using Flex and Bison
Chapter 5

NoGap Overview

All the contents in this section is that of the Ph.D. work of our thesis supervisor Per Karlström, to develop the complete NoGap framework. We have attempted to explain NoGap briefly from the knowledge gained through NoGap research papers, existing code and discussion with Per and Wenbiao. Understanding the basics of NoGap is essential for the reader to understand our development work of the NoGap tools, especially cycle-accurate simulator generator.

NoGap (Novel Generator of Accelerator and Processor Framework) is an accelerator and processor construction framework. It is a tool aimed at ASIP design, utilizing hardware multiplexed data paths. NoGap provides design freedom by imposing few limits on the architecture.

5.1 NoGap Components

This section describes the main components of NoGap that are specifically related to simulator and assembler generators.

5.1.1 NoGap Common Description (NoGap\textsuperscript{CD})

NoGap uses NoGap Common Description (NoGap\textsuperscript{CD}) to define the micro-architecture of the processor. NoGap aims to generate NoGap\textsuperscript{CD} from one of high level languages, which are called facets in NoGap. NoGap common language (NoGap\textsuperscript{CL}) is the default facet used in the project, which after compilation by NoGap\textsuperscript{CL} parser produces NoGap\textsuperscript{CD}. NoGap architecture diagram from Per’s work is shown in figure 5.1.

NoGap\textsuperscript{CD} consists of Mase (Micro-architecture Structure Expression), Mage (Micro-architecture Generation Essentials) and Castle (Control Architecture STructure Language). Mage is the AST (Abstract Syntax Tree) representation of an FU (functional unit). FU is any basic building element of the micro-architecture, like register or adder. Mase is a graph which connects all the FUs together and
thereby representing the micro-architecture in graph form with data and control
paths. \textit{Castle} contains directives for generation of instruction decoders.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{system_architecture.png}
\caption{\textit{NoGap} System Architecture}
\end{figure}

5.1.2 Spawners

From the \textit{NoGap}^{CD}, software tools useful for designer are generated through the
automatic code generators called spawners. Spawners are used not only for software
tool generation, but also for generation of HDL code like verilog. The Assembler
Generator and Cycle-accurate Simulator Generator are the spawners that have been
created as our master thesis. Using these spawners, assembler and cycle-accurate
simulator can be generated automatically. The major advantage of having spawner is that, it can be re-used for different facets to generate the
required software tool, where only the new facet has to be implemented while re-utilizing the existing spawners. The only existing spawner in \textit{NoGap} is the verilog
generator, which generates verilog code from \textit{NoGap}^{CI}.

5.2 \textit{NoGap} Common Language (\textit{NoGap}^{CL})

\textit{NoGap} Common Language (\textit{NoGap}^{CL}), which is the default facet of \textit{NoGap}, is
used to describe the hardware architecture. Though \textit{NoGap}^{CL} resembles VHDL
and Verilog, there are noteworthy differences in NoGap\(^{CL}\). As Wenbiao et al states in [36], the main advantages of NoGap\(^{CL}\) are

1) Less micromanagement needed for control path construction.
2) No processor template restriction, providing more freedom for the designer.
3) Support of dynamic port sizes.
4) Automatic decoder generation.
5) Pipeline stages can be adjusted easily; different pipelines can be defined for different operations.\(^*\)

Since our cycle-accurate simulator simulates the processor created through NoGap\(^{CL}\), we attempt to explain NoGap\(^{CL}\) briefly in this section, which will serve as foundation to understand functioning of the cycle-accurate simulator generator.

### 5.2.1 Functional Unit (FU)

Functional units (FU) form the basic building blocks in NoGap\(^{CL}\). FUs are used to represent the functionality, data path and control path of a processor. FUs in NoGap can be either a leaf FU or top FU. Mage is created from these leaf FUs, and in this report we’ll refer to them as Mage FU. Only the top FU contain operations of the processor, decoder and pipeline implementation and this FU generate the Mase, and hence we’ll call it Mase FU.

#### 5.2.1.1 Template FU

Template FU is the FU with template option just like C++ templates.

#### 5.2.1.2 Inline FU

Each Arithmetic operation within the Mase FU is represented as an Inline FU.

#### 5.2.2 Mage FU

Mage in NoGap is similar to VHDL and Verilog and is a parallel hardware description language. A key feature differentiating Mage from HDL is that combinational loops are not allowed. An example Mage FU is shown in Listing 5.1. The input represents input port and the values within the brackets [ and ] indicates the upper and lower range of the bit size of the ports. For example [3:0] represents a 4 bit port. Output defines the output ports for the FU. One of NoGap’s advantage is dynamic port sizing. When no size is defined, the default size of 1 bit is assumed. Signal represents a signal within the FU. Cycle and Comb block represent combinatorial logic and clocked logic respectively. In the example Mage FU, all reading is done through Comb. Writing is done within cycle and any signal which gets assigned within cycle is considered a register of the processor. Based on the Signal value, Switch construct provides a choice of either reading or writing of doing nothing. (corresponds to NOP instruction). A specialty of NoGap is that there is no clock or reset inputs due to the way Mage FUs are coded with cycle blocks.
Listing 5.1: Mage FU

```vhdl
fu reg_file
{
    input [3:0] addr_a_i;
    input [3:0] addr_b_i;
    input [3:0] addr_w_i;
    input [5:0] dat_i;

    output [15:0] dat_a_o;
    output [15:0] dat_b_o;

    input wr_en_i;

    signal [15:0]:[15:0] register;

    cycle
    {
        switch(wr_en_i)
        {
            1: \%WRITE
            {
                register[addr_w_i] = dat_i;
            }
            0: \%NOP{}
        }
    }
}
```

The code written within the brackets { and } constitutes a clause within an FU. Clauses can be given name which is used by operations in Mase FU to control the Mage FU.

5.2.3 Mase FU

An example Mase FU is shown in Listing 5.2. Phase, Stage, Operation and Pipeline constructs are the key elements defined exclusively in Mase FU. Phase declares the different phases or delays in pipeline management like fetch, decode, execute and write back. Every Pipeline construct defines a pipeline (example: normal pipeline or long pipeline) with relationship between the different phases. The phases are connected to each other within a pipeline through Stage. A Stage is a template description of how input and output of phases are related. A cycle within
stage indicates one delay and comb within stage means 0 delay. Stage represents
data flow from one phase to the next one. This data connection can be either
direct, where two phases are connected in same time step (using comb), or through
flip-flop, where the phases are separated by one time step (using cycle). The
Operation construct consists of the pipeline for an instruction and the decoder for
the instruction. In this Operation construct, action to be taken for each instruction
at different phases in the pipeline is defined. The decoder itself is defined as a
template FU within Mase and it in-turn defines source and destination operands,
immediates and constants for every instruction [36, 18].

5.2.4 Parse Unit

A Parse unit in NoGap is the unit that contains functionality for each module in
the system. Mage FU, Mase FU, or a template FU for decoder are all instances
of a parse unit.

5.3 NoGap Connection Graph NCG

Following graphs come under NoGap connection graph.

- Mage Graph
- Variable Dependency Graph
- Mase Graph

5.3.1 Mage Graph

The NoGapCL is parsed and an Abstract Syntax Tree (AST) is created for each
FU. Mase FU too will have this abstract syntax tree graph. These will be called
as Mage graphs in this report. Every Mage graph corresponds to the abstract
syntax tree for an FU in the NoGapCL.

To know about parsing in general the reader can refer to the Chapter 6 on assem-
bler generator in this report where usage of the tools Flex and Bison are explained.
Knowledge of how the Mage syntax tree is created is irrelevant for this report.

A Mage graph generated within NoGap is shown in Figure 5.2.

5.3.2 Variable Dependency Graph

Every FU has a corresponding variable dependency graph, which explains the
dependencies between variables in the FU. Example for a variable dependency
graph is shown in Figure 5.3 which is derived from the example Mage FU in
Listing 5.1. co= is the combinatorial dependency which corresponds to the comb
clause of the FU. cy= is the cycle assign which is part of cycle clause of FU.
During cycle assign, value is assigned to a register. Target nodes of Op edges
are the operands of a expression. cy= in the variable dependency graph is used
Listing 5.2: Mase FU

```c
fu data_path
{
  input [31:0] op_i;
  output [7:0] target_address_o;
  fu::reg_file(\%NOP) rf;
  fu::alu(\%ADD) alu;

  phase DE;

  stage ff()
  {
    cycle
    { ffo=ffi; }
  }

  pipeline normal_pipe
  {
    DE -> ff -> OF -> ff -> EX -> ff -> WB;
  }

  comb
  { dat_i_wo = dm.dat_i_wo;
    .......... 
  }

  operation(normal_pipe) nop(dec_unit.nop)
  { @DE;
    dec_unit;
    dec_unit.instr_i = op_i;
    jump_o = dec_unit.jump_o;
  }

  operation(normal_pipe) alu_inst(dec_unit.alu_sq)
  { @DE;
    dec_unit;
    dec_unit.instr_i = op_i;
    jump_o = dec_unit.jump_o;

    @OF;
    rf;
    rf.addr_a_i = dec_unit.rf_a;
    rf.addr_b_i = dec_unit.rf_b;

    @EX;
    alu\%ADD,\%SUB,\%AND,\%XOR,\%OR,\%NOT,\%INC,\%DEC\\
    alu_flag\%UPDATE\\
    alu.a_i = rf.dat_a_o;
    alu.b_i = rf.dat_b_o;
    alu_flag.dat_i = alu.res;

    @WB;
    rf\%WRITE;
    rf.addr_w_i = dec_unit.rf_w;
    rf.dat_i=alu.res[15:0];
    status_reg;
    status_reg.alu_flag_i=alu_flag.dat_o;

  }
}
```
Figure 5.2: NoGap Mage graph
to determine if loops are present in \textit{Mase} graph in the coding of cycle-accurate simulator generator.

![NoGap variable dependency graph](image)

\textbf{Figure 5.3: \textit{NoGap} variable dependency graph}

There are five input ports and two output ports in the \textit{Mage} FU. The top two nodes in the graph are that of output ports \texttt{dat\_a\_o} and \texttt{dat\_b\_o}. These two ports are assigned the values \texttt{register[addr\_a\_i]} and \texttt{register[addr\_b\_i]} within the comb, and hence the outedges of these ports are \texttt{co=}. The Operands of \texttt{register[addr\_a\_i]} are \texttt{register} and \texttt{addr\_a\_i} whose inedges are \texttt{Op, wr\_en\_i, dat\_i}} and \texttt{addr\_w\_i} are connected by the \texttt{cy=} in cycle clause. All the out ports are the roots nodes and all in ports are the leaf nodes of the variable dependency graph.

\subsection{Mase Graph}

\textit{Mase} graph is created from \textit{Mase} FU. Every element within the \textit{Mase} FU (like ports, instance of \textit{Mage} FUs, decoder, signals) are represented as nodes. The final graph is created by interconnecting \textit{Mage} FUs and ports, inserting multiplexer and flip-flops, combining edges that are equal, wire naming, wire and port sizing.

Every node and edge in the \textit{Mase} graph have attributes, and these attributes are represented in a class each for vertex and edge information. The class hierarchy of the nodes is shown in Figure 5.4.
Figure 5.4: Mase graph node class hierarchy

An example Mase graph is shown in Figure 5.5. The example is a simple Mase graph to illustrate the inter-connection between the Mage FUs, and doesn’t contain all types of node. Below are the possible types of nodes inserted in the Mase graph depending on the NoGap\(^{CL}\) code in Mase FU.

5.3.3.1 Global In Port
The input port of the Mase FU is represented by the Global In Port.

5.3.3.2 Global Out Port
Global Out ports are the output ports in Mase FU.

5.3.3.3 Signal Node
Signals in Mase FU is represented by the Signal node.

5.3.3.4 FU In Port
Every Input port in a Mage FU is modeled into a FU In Port.

5.3.3.5 FU Out Port
Each output port of the Mage FU is an FU Out Port in the Mase graph

5.3.3.6 FU
The instance of an FU, which is instantiated within Mase FU, is represented by FU node in Mase graph. This node contains the information of the Parse unit (in other words, the FU) from which it is instantiated.

5.3.3.7 Inline Expression
The Inline FUs are modeled as Inline Expression nodes.
5.3.3.8 Decoder

The Decoder within the processor is represented as decoder node.

5.3.3.9 Flip-Flop node

A flip-flop node is inserted in the \texttt{Mase} graph wherever a delay is necessary in the instruction pipeline.

5.3.3.10 Multiplexer Node

When there are more than one inputs to a port, the inputs go through a multiplexer node.

5.3.3.11 Multiplexer Control Port

Multiplexer control port is the control signal for the multiplexer node, and controls which input should be the output from the multiplexer.

5.3.3.12 PipelineClassControl Node

This node represents the class selector unit which selects the instruction to be executed in the pipeline.

More explanation about \texttt{NoGap} elements can be found in [17].

5.4 Uniqueness of \texttt{NoGap}

According to Per et al in [15], existing ADL tools are instruction based and does not know much about RTL (Register Transfer Level). On the other hand, \texttt{NoGap} will be based on RTL and also will know about instructions. In \texttt{NoGap} every unit of hardware including registers are defined at the register transfer level, as a Functional unit (FU). This enables the FUs to manage the hardware complexities of pipelining and multiplexing. The key principle of \texttt{NoGap} is compositional design. The \texttt{Mage} FU is made independent of the operations in \texttt{Mase} FU. This enables the designer to use the \texttt{Mage} FU as either instruction driven module or a normal hardware module. This is made possible by the use of dynamic clause selection, where the named clause defined in \texttt{Mage} FU can be accessed in an operation in \texttt{Mase} FU. \texttt{NoGap}’s design can be learned more through [15] and [18].
Figure 5.5: NoGap Mase graph
Chapter 6

NoGap Assembler Generator Implementation

This chapter explains the implementation of Assembler Generator developed for NoGap. The assembler developed, is given the name NoGap assembler as it works with all the processors developed through NoGap framework. The NoGap assembler is developed through a generator tool, and this chapter includes implementation details of the generator. General concept of the assembler is briefly described in Chapter 2. Reader can also refer to [16] for overview of Assembler Generator.

To avoid confusion of what is what when discussing the assembler generation process, the terminology in Table 6.1 will be used throughout this chapter.

<table>
<thead>
<tr>
<th>Term</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>assembly program</td>
<td>Assembly code written by a user for a processor to perform a task.</td>
</tr>
<tr>
<td>NoGapAsm</td>
<td>The generated executable, which controls the parser, that converts an assembly program to a binary file for the processor.</td>
</tr>
<tr>
<td>AsmGen</td>
<td>Part of NoGap that uses NoGap$^{CD}$ to generate NoGapAsm.</td>
</tr>
<tr>
<td>instruction</td>
<td>Binary instruction for the processor.</td>
</tr>
<tr>
<td>assembly instruction</td>
<td>Assembly code representing one binary instruction.</td>
</tr>
</tbody>
</table>

Table 6.1: Terminology
6.1 **NoGap Assembler**

This section explains the purpose and overview of the NoGap Assembler. NoGap Assembler Generator, discussed in Section 6.3, generates the NoGap Assembler. The purpose of NoGap Assembler is to convert an assembly program written by the user into a machine code for a processor generated with NoGap framework.

NoGap Assembler can handle assembly program written for any processor created with NoGap. This is due to the generic process used for assembler construction. A general purpose parser is used, which means that all assembly programs will have the same basic syntax. When a new processor is generated, the only thing that will be changed is the definition file, discussed in Section 6.3.1.

One of the problems of implementing the Assembler Generator is that NoGap Assembler has to be a stand alone executable. This means there must be a way for Assembler Generator to pass information to an as of yet non existing program. This problem was solved by storing the necessary information in intermediate files. This flow as well as the flow of running NoGap Assembler is shown in Figure 6.1, where the top gray area represents components that are part of the NoGap executable, and the bottom gray area represents the components that are part of the NoGap Assembler executable. Assembler Generator will extract instruction details from NoGap CD, and write this information to a definition file. The definition file is loaded by NoGapAsm into a data structure which is used by the parser to verify the assembly program. The parser and lexical analyzer are constructed using Bison [6] and Flex [26] respectively. The mnemonic alias, instruction table, and definition file are described in Section 6.3.

The process of assembling an assembly program is done in two passes. In the first pass, the syntax of the input program is verified by the parser against the instruction details provided in the definition file. If the first pass is successful, labels are handled in the second pass, finally generating the binary for the processor.

6.2 **Instruction Format**

Each new processor will probably have a unique instruction format and instruction set, although all instructions will have some commonalities. This section describes the general instruction format used for processors generated by NoGap.

Currently NoGap synthesizes the instruction format so that it always consists of an opcode, padding bits, and immediate data fields. By an immediate field we mean an instruction field that will be copied verbatim from the instruction word into the data path, e.g. register addresses will be considered immediate data as well as, what is traditionally considered immediate data in an instruction. So far NoGap can only handle fixed size instructions, for this reason padding bits are used to pad shorter instructions. An example of this general instruction format is
Figure 6.1: NoGap assembler flow
shown in Figure 6.2.

NoGap will automatically synthesize the needed instruction formats for a particular data path/processor.

![Instruction format example](image)

**Figure 6.2:** Instruction format example

### 6.3 NoGap Assembler Generator (AsmGen)

The name Assembler Generator for this module comes from the fact that it generates the information that eventually is used to generate the NoGap assembler. Assembler generator is a NoGap spawner used to create assemblers for generated processors.

![Assembler Generator](image)

**Figure 6.3:** Overview of NoGap Assembler Generator

Figure 6.3 shows overview of NoGap Assembler Generator. The NoGap\(^{CL}\) code created by the designer is parsed by NoGap parser which produces NoGap\(^{CD}\). Assembler Generator uses the information from NoGap\(^{CD}\) and extracts the needed information. For each instruction following information will be generated by AsmGen:

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

The information extracted from the NoGapCD is stored to a data-structure within the AsmGen. Figure 6.5 shows the container that stores the information. It is a C++ STL map. The map has the mnemonic name as the key value, and a C++ STL pair as the value. The pair is composed of a struct called mnemonicinfo, and a vector of another struct called registerinfo. This data structure is directly written to an output file through a boost library by name serialization. In serialization, a class is created which contains the complex STL map. An object is created for this class, and is filled with instruction details which are extracted from instruction table within NoGapCD. Through serialization the whole object is written directly to the output file. The benefit of writing the map to the file this way is that when the file is read by another program using serialization, the values get loaded into a similar STL map directly. This process is illustrated in Figure 6.6

Three different files are generated from the AsmGen containing different data and targeting different goals. Figure 6.7 shows the internal data structures, generated files and displays a method name through which the files are generated.

Figure 6.8 shows the executions flow and the operations performed by the AsmGen.

As shown by the Figure 6.4, following are the files generated by the AsmGen
• Definition file
• Instruction Table file
• Mnemonic Alias file

Files generated by the AsmGen will be discussed in brief in the later sections. The file description gives idea about what type of information is used by the NoGap assembler.
**Figure 6.5:** NoGap AsmGen serialization data structure

**Figure 6.6:** Writing definition file using serialization

**Figure 6.7:** NoGap AsmGen internal data structures and generated files
Figure 6.8: NoGap AsmGen Flow Chart
Listing 6.1: Instruction table contents

<table>
<thead>
<tr>
<th>Instruction (Mnemonic)</th>
<th>Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_imm(%WRITE)</td>
<td>rf[19:4]</td>
</tr>
<tr>
<td>ret_inst(%TRUE)(%INC)</td>
<td></td>
</tr>
<tr>
<td>store_mem(%WRITE)</td>
<td>rf[7:4]</td>
</tr>
</tbody>
</table>

6.3.1 Definition file

The information generated from ΝοGap is stored in the definition file. The definition file is written using the serialization package in Boost [5]. The definition file is used as input to Assembler as discussed in Section 6.5. The assembler reads the file through serialization package and makes use of the information during the parsing, lexical analysis and error handling.

6.3.2 Instruction table

The instruction table file contains all available assembly instructions, used as a reference for the user. Each assembly instruction is listed with its mnemonic and details about the operands. Listing 6.1 shows the contents of the definition file.

Each line in the Listing 6.1 represents one instruction. The part of the instruction before the first space in a line is the instruction name. The instruction name is followed by the register name. The numbers in the square brackets are the respective immediate operands bit positions in the generated instruction format. Each register is separated by space. Instructions can have different number of registers. An instruction may even have no register.

This file is made available to the user and it is useful in different ways, such as:

- Once the ΝοGap assembler, generates an error for an instruction, the user can refer to this file. After checking the instruction table file user will know the exact format for the instruction and can fix the bug easily.

- This file can also be used by the user to check all possible instruction that s/he can use. This is important in one way that ΝοGap framework is rapidly evolving and there will be more instructions added to the processor. Therefore the definition file is always updated by the AsmGen whenever it runs, providing the user with the latest instructions set.

6.3.3 Mnemonic aliases

Mnemonic aliases file, as the name depicts contains aliases for the automatically generated mnemonics. The mnemonics generated automatically are mostly lengthy
and the mnemonic alias file makes it possible for the users to choose on their own, short and precise alternative for the mnemonics. The mnemonic alias file is also made available to the user in order for the user to view and change its contents. Once the user adds or changes an existing alias then the AsmGen reads the alias file again while running, and updates the information in the other files accordingly. This is one way NoGap Assembler provides ease of use and customization. Listing 6.2 shows few mnemonics with their aliases. All the mnemonics may not have aliases.

The three files are interlinked. If data in one file is changed, other files also have to be updated by re-running Asmgen. If the user adds a new mnemonic to the Mnemonic aliases file, the definition file has to be updated to support the new alias and for the new alias to be considered as an instruction. Instruction table should also be updated in order to show the new alias among the instructions. The changing and customization is supported and handled by the Assembler Generator.

### 6.4 Assembler Program

Another part of the assembler generation process is to generate C++ code for the actual assembler program, i.e. the binary that translates an assembly program to a binary program for the generated processor. The assembler program consist of three static source programs:

1. C++ Program (Assembler Driver Program)
2. Lexical Analyser
3. Parser

Figure 6.9 shows the diagram of assembler program displaying the input and output files and the internal sub-programs. The purpose of the diagram is to give a quick view of the assembler program.

The programs making the assembler work, shown in the Figure 6.9, are described in Sections 6.5, 6.6, 6.7.

### Listing 6.2: Mnemonics with aliases

<table>
<thead>
<tr>
<th>Mnemonic</th>
<th>Alias</th>
</tr>
</thead>
<tbody>
<tr>
<td>load_imm(%WRITE)</td>
<td>load_immediate</td>
</tr>
<tr>
<td>move_rf_mac(%HIGH)(%WRITE)</td>
<td>move_register_mac</td>
</tr>
<tr>
<td>ret_inst(%TRUE)(%INC)</td>
<td></td>
</tr>
<tr>
<td>store_mem(%WRITE)</td>
<td></td>
</tr>
<tr>
<td>alu_inst(%DEC)(%UPDATE)(%WRITE)</td>
<td></td>
</tr>
<tr>
<td>add(%ADD)(%WRITE)</td>
<td>add</td>
</tr>
</tbody>
</table>
6.5 C++ Program - Assembler Driver

The C++ program works as a driver program for the NoGap Assembler. This is the reason for calling this program assembler driver. This program performs several actions for handling different sections of the Assembler execution. The C++ program can be described as having following main functionalities

- Getting input from user
- Reading the definition file
- Controlling the Flex and Bison programs
- Performing the second pass for the assembler
- Generating binary output files

Figure 6.10 shows the containers that are used within the driver program and the arrows pointing in and out to the parser show which containers get and provide data to parser program.

Figure 6.11 shows a flowchart of the assembler driver. It depicts the sequences of operations performed by the driver program.

6.5.1 Getting input from user

The C++ program gets the input assembly program that user has written. The user can also mention the output file along with the input file. The program checks if the user has provided an output file or not. The binary output generated through the NoGap assembler will be sent to the output file specified by the user. If the user has not mentioned any output file then one will be created automatically and the binary output will be written to the corresponding file.
6.5.2 Reading the definition file

Reading the definition file (explained in Section 6.3.1) is performed using the serialization package in boost. The information after loaded into a data-structure is used by the parser for syntax checking. Flex does not use the information from the definition file.

6.5.3 Controlling the Flex and Bison programs

The C++ program controls the lexical analysis and parsing in a way that it provides input to the corresponding program (lexical analyzer and parser) and receives their output. It also stops execution of the assembler in case of syntax errors detected during parsing.

6.5.4 Performing the second pass for the assembler

NoGap assembler is a two pass assembler and the second pass is carried out by the C++ driver program. The second pass is performed in order to resolve the labels (explained in Section 6.7.2). During the parsing of the assembly program the labels are saved and are not processed. Once the parsing is performed and there are no errors detected, then the binary values for every instruction and the label details are passed to the driver program. Then the labels are resolved and final binary is generated if there are no errors detected.

6.5.5 Generating binary output files

The driver program generates multiple output files. Description of the output files is provided in Section 6.9.
Figure 6.11: Assembler Driver Flow Chart
Figure 6.12: Assembler Generator Class diagram

AssemblerGenerator

- v_token_m : 
- mnemonic_alias_map : 
- path_m : 
- dp_pu_m : const parser::ParseUnit*
- seq_pu_m : const parser::ParseUnit*

- get_single_slice_ins_fmt : const arch_data::instructionFormat&, type : const arch_data::instructionField::Type::enum_t& : boost::optional< types :: slice_t >
- get_opcode_bits(pu : const arch_data::instructionFormat&) : generator::types::slice_t
- get_instruction_bits(pu : const parser::ParseUnit&) : generator::types::slice_t
- get_number_of_immediates(ins_fmt : const arch_data::instructionFormat&, type : slice_t )
- get_padding_bits(ins_fmt : const arch_data::instructionFormat&, type : slice_t )

# generate_predicate(pu : const parser::ParseUnit& ) : const bool
# operator ( ) (pu : parser::ParseUnit&
# process_mnemonic_alias() 
# extract_instruction_info(pu : const parser::ParseUnit&
# create_instr_serializationfile() 
# create_instr_lookupfile() 
# set_output_path(path : const boost::filesystem::path&) 

 mnemonic_info

+ mnemonic_name_m : string
+ opcode_m : int
+ opcode_size_m : int
+ opcode_upper_range_m : int
+ opcode_lower_range_m : int
+ padding_bits_size_m : int
+ padding_bits_upper_m : int
+ padding_bits_lower_m : int
+ serialize[ Archive&, : const unsigned int]
+ mnemonic_info() 
+ mnemonic_info(mnemonic_name : string, opcode_m : int, op_size : int, op_upper : int, op_lower : int, padding_bits : int, padding_upper : int, padding_lower : int)
6.6 Lexical Analyser

Flex is used to generate the lexical analyzer for the NoGap Assembler. The lexical analyzer stays the same for all NoGap generated processors. The user assembly program is first provided as input to the lexical analyzer. It analyzes the assembly program and produces the tokens for all the unique patterns found in the user’s assembly program. These tokens are then used by the Parser. The lexical analyzer generates generic tokens for:

- Mnemonics
- Registers
- Label Names

Thanks to the generic token that we are able to make a stable lexical analyzer for all kind of NoGap generated processors. The existing lexical analyzer program need not be changed for any new instruction or mnemonic.

Figure 6.13 shows the functionality of lexical analyzer. It takes the assembly program and matches the lexemes against patterns to generate tokens.

![Figure 6.13: Lexical Analyser generated using Flex](image)

6.7 Parser

Bison is used to generate the parser. Figure 6.14 shows the parser function. The parser verifies and validates the assembly program and individual instructions and generates the values of the instructions elements.

The parser defines the grammar rules for the assembly language program. Rules are composed of terminal and non-terminal symbols. Each line in user program can be either:

- Instruction
- Directive
Each of the above stated items should be in a new line. The grammar rules for the assembly language are always the same, independent of the processor generated. The general grammar for the parser is simple and is shown in BNF 1.

\[
\begin{align*}
\langle \text{statement} \rangle & : \Rightarrow \langle \text{label} \rangle \mid \langle \text{directive} \rangle \mid \langle \text{instruction} \rangle \\
& \quad \mid \langle \text{newline} \rangle \\
\langle \text{directive} \rangle & : \Rightarrow \langle \org \rangle \ \langle \text{number} \rangle \ \langle \text{newline} \rangle \\
\langle \text{label} \rangle & : \Rightarrow \langle \text{string} \rangle : \ \langle \text{newline} \rangle \\
\langle \text{instruction} \rangle & : \Rightarrow \langle \text{mnemonic} \rangle \ \langle \text{oplist} \rangle \ \langle \text{newline} \rangle \\
\langle \text{oplist} \rangle & : \Rightarrow \langle \text{oplist} \rangle \ \langle \text{operand} \rangle \mid \langle \text{operand} \rangle \\
\langle \text{operand} \rangle & : \Rightarrow \langle \text{string} \rangle : \ \langle \text{label} \rangle \\
& \quad \mid \langle \text{string} \rangle : \ \langle \text{number} \rangle \\
\langle \org \rangle & : \Rightarrow .\org \\
\langle \text{newline} \rangle & : \Rightarrow \text{end of line} \\
\langle \text{number} \rangle & : \Rightarrow \text{numeric literal} \\
\langle \text{string} \rangle & : \Rightarrow \text{alphanumeric string}
\end{align*}
\]

**BNF 1: NoGapAsm grammar**

An instruction can be either a mnemonic name, or a mnemonic name followed by list of operands. Operands contain the registers and the values. Register must contain register name followed by colon and memory address or label. Checks are made to ensure the parsed instructions have the format specified in the definition file. Operand values are checked against the permissible range.

In case label is used to represent value of the register, it is not processed in the first pass of Bison, but written as it is to a container (STL vector).

If all checks are OK, all values are written to a data structure. Another option would have been to write each parsed and assembled instruction directly to the
output file. The advantage, of writing the assembled results to an intermediate format first, is that it is easy to extend the functionality of NoGap Assembler with new output file generators. Currently supported output formats are ASCII-binary (the ASCII representation of the binary code), ASCII-binary with inlined assembly code, and pure binary.

6.7.1 Directive

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

6.7.2 Label

A label is an alphanumeric string which is followed by a colon(:). For NoGap Assembler, while defining the label, it must be written in a separate line. It can not be followed by an instruction or directive or another label in the same line. A label can also be used in place of a register value as well. The labels are not resolved by the parser. When a label is encountered in the assembly program, its name and the memory location is saved in a data structure. This data structure is later used by the C++ Driver Program to check and resolve the labels.

6.8 Mnemonic Generation

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

Mnemonics are automatically generated according to the following pattern:

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

where \text{opname} is the operation name and \text{%CA}_X is a clause access name. The \text{opname} and clause access name(s) are collected from the \text{Mase} description of the data path. To understand the meaning of a clause access name and what it has to do with an instruction, the reader is referred to [15].

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

\text{fname}:N

where \text{fname} is the name of the FU connected to the respective immediate output port from the decoder and \text{N} is a number. For example a resulting assembly instruction can look like this: add(%ADD)(%WRITE) rf:10 rf:4 rf:3
6.9 Output Files

\texttt{NoGap} Assembler generates output files in different formats. The format of the output file also depends upon the user which format s/he specifies. Following are the output file formats supported by the \texttt{NoGap} Assembler.

- Binary
- ASCII
- ASCII-Binary

The ASCII-Binary will always be generated by the \texttt{NoGap} Assembler whether the user asks for it or not. Through ASCII-Binary file the user is able to know the binary equivalent for each of his/her instructions. The ASCII-Binary output file gives the binary inlined with the instructions. The ASCII-Binary can also be helpful for generating the disassemblers.

An output file from the \texttt{NoGap} Assembler is shown in Listing 6.5. The Listing 6.5 shows the output as ASCII-binary with comments.

6.10 Error Handling

\texttt{NoGap} Assembler performs regular error checking and reporting at different steps during the binary generation process. The input assembly program is checked for errors. On detecting an error two types of messages can be generated, error or warning. These error messages are displayed on the terminal window. Error checking is based on the instruction rules (written to the definition file explained in section 6.3.1) that are generated by the AsmGen.

\begin{quote}
\textbf{Example 6.1: Error message format}
\end{quote}

Following is the format of messages generated when different errors are encountered:

\begin{verbatim}
FileName:LineNumber:warning:Warning Message Text
FileName:LineNumber:error:Error Message Text
eroerror:label LabelName at line: LineNumber: Error Message Text
\end{verbatim}

\textit{FileName} is the name of user input file and Error and Warning message text states the exact problem. \textit{LineNumber} is the line in the user program which contains the error. \textit{LabelName} will be the name of label that is in error. The error message related to the label will only be generated through the assembler driver, parser will never display such an error message.
Each instruction in the input program goes through a number of checks. Checks for errors are made by the C++ driver program and the Bison parser. Error handling by two different programs will be discussed in different sections.

### 6.10.1 Error Handling by Bison Parser

Error checking starts when a mnemonic is found in the user program. Parser checks if the mnemonic is available in the NoGap mnemonics. Once the mnemonic is found in the NoGap mnemonic list, the list of registers is checked. The parser checks if the instruction contains the necessary registers and whether their names are correct in the input program or not. Finally the size of data provided for each register is checked, whether it fits the register or not. The labels defined in the input program are only stored in a data structure and later passed to the C++ Driver program for error checking.

---

**Example 6.2: Errors generated from parser**

This example states some errors that are generated from the Parser program. Instead of generating some general errors, NoGap Assembler generates very specific errors. It is an effort to provide ease of use and making error correction easier for the user.

```
\it Test_file:2: error: cannot find the mnemonic 'add(%ADD)(%WRITE)'
Test_file:7: error: register value out of range: expected: less than '16' received: '40'
```

Above error messages are copied verbatim. [Test_file] is the user input file name, following it is the line number, then it shows if it is error or warning followed by the error message.

The above example shows that the errors generated through the parser program are very specific and informs the user about the exact problem. First error shows that the mnemonic is not found, therefore user can check in the instruction table file (explained in section 6.3.2) if it exists or not. The second error clearly shows the problem and can also be corrected or verified by looking into the instruction table file provided to the users.

Generating specific errors is a good feature of NoGap Assembler. Generating specific error takes some extra implementation time, strict error checking and a good understanding of the input assembly language.

### 6.10.2 Error Handling by Assembler Driver Program

Error checking starts in the assembler driver. In the beginning of the execution assembler driver checks for all the related file. The related files are input, output and the definition files. If any of these except (output file) is not found then the program terminates without proceeding any further and displays an error message.
Example 6.3: Errors generated from Driver Program

This example states few errors that are generated from the Driver program.

error: can not open the file 'instruction_definition.nogap'
error: Input file not mentioned

Two error are stated above which were obtained purposefully. The first error is caused by the fact the definition file was not present there. This error occurred because the definition file was deleted for testing purpose. The second error is due to the user not mentioning any input file.

If the first error check is passed then input is forwarded to the lexical analyzer and the parser. If any errors are found by the parser, no further processing of the input program is carried out and errors are reported to the user. If there are no errors detected by the parser the driver program checks all the labels. Checks are made to see if a label is defined somewhere in the assembly program or not. If the label is defined then whether it is defined within a specific memory address range. The driver program further check the labels that come in the registers. These labels coming in the registers as data are checked for their size if they are within the size limits of the registers or not.

Example 6.4: Errors generated from Driver Program

This example states an errors that is generated from the assembler driver program due to erroneous label.

error: label 'label2' at line: 6 is either not defined or in error

There was no label2 defined in the program. Therefore an error informing the user that it is in error is displayed as stated above.

Therefore it can be said that \texttt{NoGap} Assembler performs a complete syntax checking and error reporting of the input assembly program.

6.11 Results

To test the Assembler Generator and the resulting assembler we used a simple RISC processor developed with \texttt{NoGap}. Listing 6.3 shows the assembler instructions extracted from \texttt{NoGap} for this processor. For description of Listing 6.3, user is referred to Section 6.3.2
Listing 6.3: Instruction set for our test processor

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Function</th>
<th>RF Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td>add(ADD)</td>
<td>WRITE</td>
<td>[14:10] rf</td>
</tr>
<tr>
<td>add(SUB)</td>
<td>WRITE</td>
<td>[14:10] rf</td>
</tr>
<tr>
<td>add2(ADD)</td>
<td>WRITE</td>
<td>[14:10] rf</td>
</tr>
<tr>
<td>add2(SUB)</td>
<td>WRITE</td>
<td>[14:10] rf</td>
</tr>
<tr>
<td>io_out</td>
<td></td>
<td>rf [4:0]</td>
</tr>
<tr>
<td>jump_0(ALWAYS)</td>
<td></td>
<td>rf [4:0]</td>
</tr>
<tr>
<td>load(WRITE)</td>
<td></td>
<td>[36:5] rf, [4:0]</td>
</tr>
<tr>
<td>loop_0(WRITE)</td>
<td></td>
<td>loop_addr [9:5], loop_addr [4:0]</td>
</tr>
<tr>
<td>move(WRITE)</td>
<td></td>
<td>[9:5] rf, [4:0]</td>
</tr>
<tr>
<td>nop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

Listing 6.4: Program excerpt

load_imm(WRITE) rf:1 rf:1
load_imm(WRITE) rf:2 rf:2
mul_inst(WRITE) mac_reg:0 rf:1 rf:2
mac_inst(ADD)(UPDATE)(WRITE) mac_reg:0 mac_reg:0 rf:1 rf:2
mac_inst(ADD)(UPDATE)(WRITE) mac_reg:0 mac_reg:0 rf:1 rf:2
move_rf_mac(HIGH)(WRITE) mac_reg:0 rf:4
move_rf_mac(GUARD)(WRITE) mac_reg:0 rf:5
alu_inst(DEC)(UPDATE)(WRITE) rf:1 rf:1 rf:6
mac_inst(ADD)(UPDATE)(WRITE) mac_reg:1 mac_reg:1 rf:1 rf:1
load_imm(WRITE) rf:8 rf:8
load_imm(WRITE) rf:9 rf:9
load_imm(WRITE) rf:10 rf:10
load_imm(WRITE) rf:11 rf:11
load_imm(WRITE) rf:12 rf:12
load_imm(WRITE) rf:13 rf:13
load_imm(WRITE) rf:14 rf:14
load_imm(WRITE) rf:15 rf:15
move_rf_mac(HIGH)(WRITE) mac_reg:1 rf:7
alu_inst(SUB)(UPDATE)(WRITE) rf:6 rf:1 rf:3
store_mem(WRITE) rf:9 rf:1
store_mem(WRITE) rf:10 rf:2
store_mem(WRITE) rf:11 rf:3
store_mem(WRITE) rf:12 rf:4
store_mem(WRITE) rf:13 rf:5
store_mem(WRITE) rf:14 rf:6
store_mem(WRITE) rf:15 rf:7
6.12 Conclusion

A small excerpt from an assembly program is shown in Listing 6.4. From the excerpt it can be seen that labels can be used as general numbers, e.g. the memory address of [label1] and [ca] are used in place of memory address for one of the operands in line 5 and 9.

This program was successfully assembled and a correct binary file was generated using command ./nogapasm Assembly File.S BinaryOutput.dat. Listing 6.5 shows the output as ASCII-binary with comments.

6.12 Conclusion

By having a fixed grammar for all kinds of processor that can be generated by NoGap, it was possible to specify a fixed lexical analyzer, parser, and driver program. Also this means that a user will be familiar with the assembly language syntax independent of the generated processor, which makes it easier to implement assembly programs for new processors. The use of a fixed parser with a definition file makes it easy to extend NoGapAsm with new functionality in the future, which is important since the NoGap project is still evolving.
Listing 6.5: ASCII-binary with comments

```
// load_imm(%WRITE) rf:1 rf:1
011001_0000000000000001_0001
// load_imm(%WRITE) rf:2 rf:2
011001_0000000000000010_0010
// mul_inst(%WRITE) mac_reg:0 rf:1 rf:2
100011_0000000000000001_0010
// mac_inst(%ADD)(%UPDATE)(%WRITE) mac_reg:0 mac_reg:0 rf:1 rf:2
011011_00000000_00_00_0001_0010
// mac_inst(%ADD)(%UPDATE)(%WRITE) mac_reg:0 mac_reg:0 rf:1 rf:2
011011_00000000_00_00_0001_0010
// move_rf_mac(%HIGH)(%WRITE) mac_reg:0 rf:4
011110_0000000000000001_0100
// move_rf_mac(%GUARD)(%WRITE) mac_reg:0 rf:5
011101_0000000000000000_00_0101
// alu_inst(%DEC)(%UPDATE)(%WRITE) rf:1 rf:1 rf:6
000010_000000000001_0001_0110
// mac_inst(%ADD)(%UPDATE)(%WRITE) mac_reg:1 mac_reg:1 rf:1 rf:1
011011_00000000_01_01_0001_0001
// load_imm(%WRITE) rf:8 rf:8
011001_0000000000001000_1000
// load_imm(%WRITE) rf:9 rf:9
011001_0000000000001001_1001
// load_imm(%WRITE) rf:10 rf:10
011001_000000000001010_1010
// load_imm(%WRITE) rf:11 rf:11
011001_000000000001101_1111
// load_imm(%WRITE) rf:12 rf:12
011001_00000000001110_1110
// load_imm(%WRITE) rf:13 rf:13
011001_00000000001111_1111
// load_imm(%WRITE) rf:14 rf:14
011001_00000000011110_1110
// load_imm(%WRITE) rf:15 rf:15
011001_00000000111110_1111
// move_rf_mac(%HIGH)(%WRITE) mac_reg:1 rf:7
011110_0000000000000000_01_0111
// alu_inst(%SUB)(%UPDATE)(%WRITE) rf:6 rf:1 rf:3
000110_00000000_0110_0001_0011
// store_mem(%WRITE) rf:9 rf:1
100110_000000000000_1001_0001
// store_mem(%WRITE) rf:10 rf:2
100110_000000000000_1010_0010
// store_mem(%WRITE) rf:11 rf:3
100110_000000000000_1011_0011
// store_mem(%WRITE) rf:12 rf:4
100110_000000000000_1100_0100
```
Chapter 7

Overview of NoGap Cycle-Accurate Simulator Generator

Several hardware description languages are used to develop simulators for simulating hardware and processors. The aim of this simulation is to generate the accurate and useful hardware. Using hardware description languages such as Verilog and VHDL for generating instruction or cycle based simulation has a number of disadvantages. They cover a huge amount of hardware implementation details which are not needed for performance evaluation, cycle-based simulation and software verification [27]. Furthermore these hardware description languages are difficult to be used for describing pipelined operations.

We have developed a cycle accurate simulator for NoGap and the overview of the NoGap cycle-accurate simulator generator implementation is briefed in this Chapter. Chapter 8 explains detailed implementation of the generator. Only the core functionality of a simulator is implemented in this thesis as proof of concept that a cycle-accurate simulator can be developed for the processor created using NoGap. The possible extensions to the functionality of the simulator is presented in Chapter 9.

For ease we’ll mention our cycle-accurate simulator generator as simgen in this thesis report.

When multiple instructions are to be executed in a processor, a pipeline is used to execute more than one instruction simultaneously. When one instruction is being executed at a pipeline stage, other instructions get executed at different pipeline stages. If there is no pipeline, every instruction will be executed without any delay, and each instruction will have to wait for the previous instruction to be executed, which is undesirable. Common pipeline stages are Instruction Fetch(IF), Instruction Decode(ID), Operand Fetch(OF), Execute(EX), Memory
Access(MEM) and Write Back(WB). A pipeline example is shown in Figure 7.1, where $I_1$, $I_2$, $I_3$ and $I_4$ are four instructions being executed simultaneously and are at different pipeline stages at any instance.

Each stage in the pipeline is executed in one clock cycle. If an instruction is dependent on the result of another instruction in the pipeline, delays are introduced in the execution of the instruction, so that the former instruction waits until the execution of later instruction. The number of clock cycles needed for execution of one instruction is called clock cycles per instruction(CPI). In Cycle-accurate simulation, results can be obtained according to the actual clock cycles.

Knowledge of $\text{Mase}$ Graph and Variable Dependency Graph explained in Chapter 5 is necessary in order to understand the working of the simgen.

### 7.1 Major Steps in implementation

The working of simgen can be divided into four major steps as below

- Step 1: Sequentialization of $\text{Mase}$ graph
- Step 2: Initialization of processor elements
- Step 3: Write Execute methods
- Step 4: Execute cycle

For simplicity sake, these four steps are explained by the use of a simple $\text{Mase}$ graph as in Figure 7.2. This simple $\text{Mase}$ graph consists of two $\text{Mage}$ FUs namely $\text{Mage}$ FU1 and $\text{Mage}$ FU2. Each of the $\text{Mage}$ FUs have two input ports, and one output port, which are named as $\text{FuIn1}$, $\text{FuIn2}$, $\text{FuIn3}$, $\text{FuIn4}$, $\text{FuOut1}$, $\text{FuOut2}$. The input to FUs are directly fed through the global input ports $\text{GIn1}$, $\text{GIn2}$, $\text{GIn3}$ and the output of the FUs are obtained through the global output port $\text{GOut}$. 

![Figure 7.1: Example for pipeline](image-url)
**Figure 7.2**: Simple Mase graph
If a loop is introduced in the example \texttt{Mase} graph where outputs of the each \texttt{Mage} FU is used as input to the other one, the graph transforms into Figure 7.3. This figure also shows the multiplexers (\texttt{Mux1}, \texttt{Mux2}, \texttt{Mux3}) added whenever a port has more than one input. Multiplexers have control ports (\texttt{Mux ctl1}, \texttt{Mux ctl2}, \texttt{Mux ctl3}), which control the multiplexer in deciding the actual output of the multiplexer at any instance. Flip-Flops (\texttt{FF1}, \texttt{FF2}, \texttt{FF3}, \texttt{FF4}, \texttt{FF5}, \texttt{FF6}) are added within the \texttt{Mase} graph to introduce delay in execution of the instruction with respect to the different phases in the pipeline.

\begin{figure}[h]
\centering
\includegraphics[width=0.8\textwidth]{example_graph.png}
\caption{Simple \texttt{Mase} graph with Flip-flops and Multiplexers}
\end{figure}

### 7.1.1 Step 1: Sequentialization of \texttt{Mase} graph

As the first step in simulation, the whole \texttt{Mase} graph is sequentialized. By sequentialization, we mean removal of any combinational loop within the \texttt{Mase} graph. For example, a loop can be introduced in \texttt{Mase} graph when the output of one FU is the input to another FU, whose output is in turn the input to the first FU. The
loops are removed by modifying the nodes within the \textbf{Mase} graph. The current implementation do not remove all possible loops, and the possible improvement to this method is discussed in Chapter 9.

If any FU contains a register to which value is assigned during \textit{cycle} clause, the FU node in the \textbf{Mase} graph is split into source, sink and pass node as shown in Figure 7.4. For each instance of \textbf{Mage} FU, the paths between all input ports and output ports are examined in the corresponding variable dependency graph. If \textit{cy=} edge is present in a path, it indicates that the input port is not connected to the output port directly and values are stored in internal registers. The input port of the path is connected to a new node called \textit{sink}, and output port of that path is connected to new node called \textit{source}. If there is no \textit{cy=} edge in the path between an input and output port, both of them are connected to a new node called \textit{pass}. The Fu node is then deleted. Thus, a Fu node is replaced with source, sink and pass nodes (either all the three or less, depending on the variable dependency graph).

The nodes which are neither FUs, nor ports are replaced with a pass node. This is the case with inline expression, multiplexers, decoders, signal nodes and pipeline-control class.

All the flip-flops are replaced with a input flip-flop and output flip-flop as seen in Figure 7.4. The values in input flip-flop and output flip-flop are maintained as the value of the original flip-flop.

After applying these node replacements, the example \textbf{Mase} graph will be transformed into an SSP graph as shown in Figure 7.5. This transformation of the graph ensures removal of combinational loops in the \textbf{Mase} graph.

In this example, it is assumed that both the FUs have no source and sink nodes, but only pass nodes. The SSP graph is then topologically sorted to know the order of execution of the nodes. For the case where the output of \textit{FU1} is the input of \textit{FU2}, the final order of execution is shown in Figure 7.6.

There are five phases in the execution. Phase I transfers inputs from global input ports to the flip-flops. Phase II executes \textit{FU1} with the flip-flops as input (actual input from \textit{Gin1} and \textit{Gin2}) and writes the value to the next set of flip-flops. Phase III executes the \textit{FU3} with input from flip-flops (originally that of \textit{FU1} output and \textit{GIn3}). Phase IV decides the output of multiplexer, in this case it is the output from \textit{FU2}, and finally in phase V the output is available in global output port. This execution illustrates the overview of cycle-accurate simulation. In every cycle, one phase is executed in the given example.

Considering execution of only one instruction, these five phases are executed in order one by one continuously. When multiple instructions are executed simultaneously, these five phases will be executing different instructions at the same time.
7.1.2 Step 2: Initialization of processor elements

The source, sink and pass nodes explained in the previous steps are C++ classes derived from the FU class. Refer to Figure 5.4 in chapter 5 and Mase graph details to know about vertex attributes.

For every FU, an object is instantiated. Each of these classes will contain input ports, output ports and registers (if any) in STL containers which can be assigned corresponding values during execution. Methods are written in these classes for executing each of source, sink, pass, (called as execute_source(), execute_sink(), execute_pass()) and methods to fill the ports corresponding to each FU, read values from port, write values to port. A class diagram of an FU class is shown in Figure 7.7.

Classes are created for all types of nodes namely multiplexers, signal nodes, inline expressions, flip-flops and decoders. After sequentialization, the SSP graph is topologically sorted which stores the vertices of SSP in a C++ STL vector. This container will have the vertices stored according to the precedence of their execution. The container will be iterated, and for every node of SSP graph which has a C++ class an object is instantiated.

Complete details of classes and related operations are explained in detail in Chapter 8.
Figure 7.5: Mase graph transformed to SSP graph
Figure 7.6: SSP graph with order of execution

Figure 7.7: Class diagram of FU
7.1 Major Steps in implementation

7.1.3 Step 3: Write Execute methods

The Execute methods within the FU and all other classes needs to be implemented with C++ code corresponding to the functionality of the FU. This is accomplished by the usage of NoGap\textsuperscript{CD}. At present, this code is written manually after the classes are generated from simgen. Automatic generation of the code from NoGap\textsuperscript{CD} is a future work for the existing simgen.

7.1.4 Step 4: Execute cycle

According to the steps in simgen implementation, so far, the output of topologically sorted SSP graph is stored in a STL vector in step 1. For each node found in the vector (which is according to the order of execution of the nodes in graph) we had already instantiated an object in step 2.

A new C++ class is generated automatically in last step, which contains a method called execute_cycle(). This method will have all the steps to execute instructions within one clock cycle. The topologically sorted vector is iterated, and for every SSP graph vertex, type of the vertex is determined. If the vertex is used to get input, then a step is added in execute_cycle() method to read the value from the previous node and store within the object to which the input vertex belongs. For example, if the node is a FU input port, corresponding FU’s write method is called, which will store the value within the container of the FU’s object. If the vertex is a source or sink or pass, corresponding execute_source(), or execute_sink() or execute_pass() method is added to the execute_cycle() method. The execute methods will store the values to the corresponding out ports according to the functionality of the FU.

Execute_cycle() method is executed once for each clock cycle. Each execution is graphically similar to Figure 7.6. Detailed explanation of these steps are presented in Chapter 8 for clearer understanding.
Chapter 8

NoGap Cycle-Accurate Simulator Generator Implementation

This chapter describes the implementation details of the NoGap SimGen discussed in chapter 7. NoGap SimGen is developed in C++. With future improvements and enhancements in mind, implementation is carried out in a way that supports reuseability and future extensibility. The implementation is performed by separate modules with each carrying out different functions. Following are the modules used for the implementation of NoGap cycle-accurate Simulator Generator. Figure 8.1 shows the overview of the SimGen.

- Simulator Generator
- Simulator Mage Generator
- Simulation Runner
- Simulator Driver

Subsequent sections provide details of the listed modules.

8.1 Simulator Generator

Simulator Generator mainly works with the generation of classes. A class is generated for every functionality of the processor constructed using NoGap\(^{CL}\). As first step, Simulator Generator module generates classes from Mage graph. Parse units are provided as input to the Simulator Generator and depending on the graph contents, different classes are generated. The generated classes are later used by the Simulation Runner (explained in Section 8.3) and Simulator Driver (explained in Section 8.4) modules. Classes generated through Simulator Generator will be
compiled along with a driver C++ program to form the actual simulator at the end of the simulator generation process.

Following are the classes that are generated through the Simulator Generator.

- Base Class
- Parse Unit (PU) Classes
- Inline Expression Classes
- Multiplexer Class
- Flip-Flop Class
- Global IN and OUT Classes
- Pass Node Class
- Decoder Classes
- Pipeline Class Control Classes

Classes given in the above list are shown in the Figure 8.2.

Classes that need to be generated only once and do not depend on the input graph, will be generated the first time Simulator Generator executes. Following sections provide a detail of the generated classes.
8.1 Simulator Generator

8.1.1 Generating Base Class

This class serves as the base class for all classes (except Global IN, OUT and Flip-Flop classes) generated through Simulator Generator. The base class defines a container for storing information about the ports. It also contains the following methods.

- Read
- Write
- Initialize
- Execute Source
- Execute Sink
- Execute Pass

Above list does not provide exact names of the methods. Purpose of listing the methods is to give an idea to the reader about their function. None of the methods is implemented in the base class, it includes only the declaration of the functions. Base class is never instantiated.

8.1.2 Generating PU Classes

For each PU, Simulator Generator generates a separate class. Each FU is an instance of a PU. Each PU class is derived from the base class and contains the input, output ports and the registers. The value and sizes of the in and out ports are also saved. Size of the ports is used to check if the value written to the port is within the size of the port. Register values can be loaded by the user or a register will get a value due to the result of an operation execution.

Figure 8.2: Simulator Generator generated classes
8.1.2.1 PU Class Methods

Read and write methods are implemented to return the value of the ports and to write values of the ports. PU class provides a method to initialize the ports with their sizes. All the ports that are present within the PU are collected and stored within the PU class. Therefore PU classes provide the ports itself, only size and values are to be initialized. If a port that is not within the PU is tried to be accessed or initialized with a size or value, an error will be generated.

PU class also implements execute source, sink and pass methods. Currently these methods are written by hand using C++. These execute methods need to be written dynamically according to the required function that is performed by the hardware. Execute methods will be used by the corresponding FU types (Source, Sink, Pass).

8.1.2.2 Purpose of separate class for each PU

Each PU will have different FU instantiations. Each FU within the PU is an instance of the PU. Therefore when we create an instance of a PU class for a specific FU, this PU should have the FU input and output ports present in the PU class. An FU that is in one parse unit may not have the same input and output ports in the other parse unit. Therefore there should be a separate class for each PU to provide the right details for FU when an object is instantiated.

8.1.3 Generating Multiplexer Class

One Multiplexer Base class is generated through Simulator Generator. Multiplexer class is derived from Base class. Multiplexer class contains input and output ports. There is no register data stored in the Multiplexer class, as Multiplexer will not have any registers.

8.1.3.1 Multiplexer Class Methods

Read and write methods are implemented in the same way as for the PU class. Values are read and written to the corresponding ports. Init method is used to initialize port names with their size. Unlike the PU class, the Multiplexer class provides no input and output ports. This implies that the port name have to be provided after a Multiplexer class is instantiated. As a Multiplexer vertex will be converted to a pass node (discussed in section 8.2.1.6), only execute pass method is implemented.

8.1.3.2 Purpose of only one Multiplexer class

For each PU there is a separate class, but for Multiplexer there is just one class. As the ports are not collected and stored in the Multiplexer class, after instantiation the port name can be provided with their details including size. The reason for this is that when a Multiplexer is converted to a pass node, the port names will
also be changed to integer names excluding control port name (more explanation in Section 8.2.1.6). Therefore providing the port name when generating the class is not useful and may lead to conflicts. This leads to the option of generating only one class for Multiplexers without specifying the port names.

8.1.4 Generating Inline Expression Classes

Each Inline Expression encountered results in the creation of a new class. Therefore there is one class for each Inline Expression. The class name is not same as the Inline Expression name. This is due to the inline expressions have names containing symbols that are not allowed in C++ identifiers. This leads to a different name for each inline expression class. The name of the inline expression class is generated by a method and stored in a container along with the original name of the inline expression. Data in the container containing the inline expression names and respective class names is written to the disk and is later read by the Simulator Generator for invocation of objects. The read and write operations of the container is done using Boost serialization. Inline expression classes store the input and output ports. Inline Expressions are later converted to a pass node (explained in Section 8.2.1.6).

8.1.4.1 Inline Expression Class Methods

Each Inline Expression class implements read, write, init and execute pass method. These methods are used to return value of the port, write a value to the port, initialize port names and sizes and perform the execution of the operation respectively. Inline expression has only execute pass methods because it will be converted to a pass node, therefore there is no need for other execute methods.

8.1.5 Generating Flip-Flop Class

Simulator Generator generates a Flip-Flop class for the Flip-Flops. Flip-Flop class is not derived from the base class. The class contains a container for source and sink Flip-Flops and their values. A Flip-Flop found in the graph will be split into source and sink Flip-Flop (explained in section 8.2.1.8). Two pointers that are pointing to the source and sink containers are provided by the class. These pointers are used to point to the source and sink containers. At the end of each execution cycle the pointers are flipped and the one directing to the source will point to sink and vice-versa.

8.1.5.1 Flip-Flop Class Methods

Following are the methods implemented by the Flip-Flop class.

- Read
- Write
• Execute Source
• Execute Sink
• Finish Cycle
• Flip Pointers
• Fill Source and Sink Flip-Flops

The Flip-Flop class has read and write methods that are implemented in the same way as for the PU class. Values are read and written to the corresponding Flip-Flops. A Flip-Flop will be converted to a source and sink node (discussed in section 8.2.1.8), therefore execute source and sink methods are implemented by the Flip-Flop class. Flip-Flop class does not have an execute pass. Finish cycle will indicate end of one execution cycle of the simulator and will be used to flip the pointers. Filling the source and sink Flip-Flop containers is done by the fill source sink Flip-Flop method.

8.1.6 Generating Global Input Class

Simulator Generator generates one Global input (GI) base class. The GI class contains a read method. GI class serves as a base class for three class. The read method for GI class reads the input. This method has been implemented in three different ways in the derived classes. Following are the derived classes for GI class.

1. Constant Reader
2. Random Generator
3. Stream Reader

The above stated classes categorizes the Global Input ports in three different categories.

1. The Global Input ports that read a constant number
2. The Global Input ports that read a random number
3. The Global Input ports that read from a stream

Each of the three given classes have a read method and it is implemented in a different way for the classes. The constant reader classes implements its read method in a way that it reads a constant value to initialize the Global Input port. The random generator, generates a random number value for its global input port and the stream reader class reads the value of the port from a stream.

8.1.7 Generating Global Output Class

Simulator Generator generates one Global output (Gout) base class. The Gout class contains a write method. The write method for Gout class writes the output to a stream. An object of Gout class will be instantiated for every global output port found in the Mase graph.
8.2 Simulator \texttt{Mase} Generator

8.1.8 Generating Pass Node Class

There are vertices other than FU, Multiplexer, decoder and inline expression vertices that are also converted to a pass node. Signal Node is one such vertex. Simulator Generator generates one class for all of the above mentioned node. These nodes are later converted to a pass node by the Simulator \texttt{Mase} generator. Therefore this class implements an execute pass method. Vertices along with their size and values are also stored. If suitable, this class can also serve as a pass node generation class for a new node added to the \texttt{NoGap} graph later during implementation.

8.1.9 Generating Decoder Class

For each decoder found in the \texttt{Mase} graph, one class is generated. The class has the same name as the decoder name and it is derived from the base class. All the methods of the base class are implemented except execute source and sink methods. The reason for not implementing execute source and sink methods is that any decoder found in the graph is converted to a pass node. Therefore execute source and sink are not needed.

8.1.10 Generating Pipeline Class Control Classes

For each pipeline class control found in the \texttt{Mase} graph, one class is generated. The class has the same name as the node name and it is derived from the base class. All the methods of the base class are implemented except execute source and sink methods. Any pipeline class control found in the graph is converted to a pass node. Therefore execute source and sink are not needed.

8.1.11 Other methods of Simulator Generator

For initializing the sizes of the ports, Simulator Generator implements a method to get the port sizes. The method gets all the ports along with their respective sizes within a parse unit. Functions for generating inline expression class names and for performing serialization are also part of Simulator Generator.

8.2 Simulator \texttt{Mase} Generator

The Simulator \texttt{Mase} Generator is responsible for sequentialization (explained in 7.1.1) of the \texttt{Mase} graph and generating the Simulation Runner class. Sequentialization of the graph is done by generating a source-sink-pass partitioning of the graph. The source-sink-pass graph will be referred to as SSP graph. Generating the SSP is one of the most important process for the simulator, therefore, Simulator \texttt{Mase} Generator plays a vital role in generating the \texttt{NoGap} Simulator. Subsequent sections explain the process of generating SSP Graph and the Simulation Runner class.
Figure 8.3 shows a flowchart for Simulator Mase Generator.

![Flowchart Diagram](image)

**Figure 8.3: NoGap Simulator Mase Generator flowchart**

### 8.2.1 SSP Graph Generation

This section describes the implementation of SSP graph generation process. The SSP graph is generated by creating the source-sink-pass nodes and deleting few other vertices from the Mase graph. The Mase graph itself is never changed, but it is copied to another graph. The copied graph is changed and hence it becomes an SSP graph. Leaving the Mase graph without any change, therefore the Mase graph can be used later for further processing. Source-sink-pass nodes will be referred to as SSP nodes collectively. The phenomenon of inserting the source-sink-pass nodes and generating SSP graph is explained in the following sections.

#### 8.2.1.1 Source Sink Pass Partitioning

Source, sink and pass nodes are generated for an FU. Few other nodes in the graph will be changed as well but this section explains the SSP node generation for an FU. An FU is the only node that has the possibility of being split to either a source and sink or a pass node. All FUs in the Mase graph have to be split into Source, Sink or a Pass node. Meaning that the resulting SSP graph will have no
FU, but SSP nodes instead of an FU. Within the Simulator Mase Generator there is another generator known as SSP Generator, which is responsible for generating the SSP graph.

The graph which will undergo the changes is a copy of Mase graph as mentioned in Section 8.2.1. Therefore the term graph is referred to the copied graph in this section unless explicitly mentioned as Mase or any other graph.

Whenever an FU is found in the graph, its corresponding variable dependency graph is opened. All the vertices in the variable dependency graph are gathered by using a C++ Boost library function that returns the vertices of a graph. The gathered vertices are checked if they are input or output vertices. Once identified as input or output vertices, they are inserted into separate input and output containers. After identification of input and output vertices, the variable dependency graph is traversed by a Boost library visitor. The visitor visits the graphs and provides the following information:

- All the finished vertices
- Cycle assign edges within the variable dependency graph

The purpose of getting the finished vertices and the cycle assign edges is to perform a depth first search and to split the FU to a Pass, Source or Sink Node. A Depth first search is performed on the variable dependency graph. Depth first search is customized to provide the relevant data. The depth first visitor starts from one of the input vertices and reaches all the possible finished vertices. If a cycle assign edge is encountered it is deleted. A path from an input vertex to an output vertex that had any cycle assign edge in-between is split because of deletion of cycle assign edges. Therefore the input vertices that had cycle edges in the path to the output vertices are no longer connected. After cycle assign edge removal, the process of identification of SSP nodes starts.

Figure 8.4 shows the variable dependency graph for FU regfile. The phenomena described in this section is demonstrated in figure 8.5. Figure 8.5 shows the variable dependency graph of FU regfile after the removal of $cy=$ edges. Now the graph is split as compared to figure 8.4. Once again the visitor visits the graph after $cy=$ edge removal and it identifies it to be a source and sink node.

### 8.2.1.2 Identification of Pass Node

The variable dependency graph (of the respective FU that is being processed), contains no cycle edges (after $cy=$ edges are removed in the previous step 8.2.1.1). Now the variable dependency graph is visited once again by the visitor (after $cy=$ edge deletion), starting from the input vertex. If it reaches any of the output vertex then it means that there was no cycle edge between them. From this we assume that there is no combinatorial dependency between the input and output. In this condition the entire FU will be replaced by a pass node. The in and out
Figure 8.4: Variable Dependency Graph of regfile FU

Figure 8.5: Variable Dependency Graph of regfile FU after removing cy edges
vertex containers are also modified to reflect which input and output vertices are connected to a pass node.

Figure 8.6 shows the variable dependency graph of an FU "sub". There are no edges in the graph for sub. Therefore it will be converted to a pass node.

![Figure 8.6: Variable Dependency Graph of sub FU](image)

### 8.2.1.3 Identification of Source and Sink Nodes

As explained in the Section 8.2.1.2, if there was no cycle assign edge, the path between input to output vertices is not split. In such a condition when the visitor visits the same graph once again it ends in the finished vertices that it provided before. In the other case, if there was any edge that was removed then visitor will not finish in any of the output vertex. When the path between the input and output vertices is split due to the presence of any edge in this condition the FU is split generating a source and a sink node (the FU can also be split in source, sink and pass, all three vertices).

Figure 8.4 shows a variable dependency graph of FU "regfile". The graph shows four edges. After the removal of edges refile will be identified as an FU which will be split into source and a sink node.

### 8.2.1.4 Insertion of Pass Node in Maste Graph

Once the SSP nodes are identified by using the variable dependency graph, the new SSP vertices have to be inserted into the SSP graph replacing their corresponding FUs.

Process of inserting a pass vertex requires removing the in and out edges from the existing FU and attaching these edges to the inserted pass node. For this purpose the containers filled during the identification of the SSP vertices are used. The program checks if there is any pass node indicated in any of the containers.
If there is a pass node mentioned in any of the containers, following steps are undertaken:

- Input vertex from the variable dependency graph will be checked against all input vertices to the FU.
- Find if an input vertex to the FU in the SSP graph is also found among input vertices in the variable dependency graph.
- If same input vertex exist in SSP graph and variable dependency graph then the input edge to the FU from the input vertex is connected to the Pass node and is removed from the FU.
- Repeat the process (adding edge to Pass node and removing from FU) for all input vertices to the FU.
- Add edges to the output vertices of FU from Pass in the same way as for input vertices. The output vertices to FU are checked against output vertices in variable dependency graph.

This process connects the pass node to the input and output vertices of the FU and removes these edges from the FU, leaving the FU with no connections. The pass node has completely replaced the FU, thereafter the FU node is removed from the SSP graph.

Figure 8.7: Adder and Subber FUs and Flip-Flop 4 and 5
Figure 8.7 shows two FUs (Adder and Subber) and two Flip-Flops (FF4 and FF5). FUs and Flip-Flops in this figure are converted to pass node and source and sink Flip-Flops.

![Diagram of FUs and Flip-Flops](image)

**Figure 8.8**: Adder and Subber Pass node and Flip-Flop 4 and 5 Source nodes

Figure 8.8 shows the pass nodes inserted in place of Adder and Subber FU shown in Figure 8.7.

### 8.2.1.5 Insertion of Source and Sink Nodes

The process of inserting source and sink nodes is quite similar to inserting pass node with a minor difference. Input and output vertex containers are checked if there is any source and sink nodes. Upon finding a source node all the output vertices (output vertices for the FU which is being considered) are matched with the input vertices from the variable dependency graph. If there is a vertex with the same name then the source node is connected to that output vertex in the SSP graph and removing the connection from the FU node. All the output vertices from the FU will be connected to the source node and removed from the FU.

For sink node all the input vertices to an FU are matched with the output vertices from the variable dependency graph. If there is a vertex with the same name then the sink node is connected to that input vertex in the SSP graph and removing the connection from the FU node. All the input vertices of the FU will be connected to the sink node and removed from the FU.

Once all the input and output edges are removed from the FU and connected to the sink and source node, the FU vertex is deleted from the SSP graph.
8.2.1.6 Pass Node Generation for Multiplexer

Multiplexer is replaced by a pass node in the SSP Graph. Generation of Pass node for Multiplexer is simpler as compared to splitting an FU. SSP generator identifies all the Multiplexer and stores the names in a container.

For a Multiplexer in the graph, each in edge is connected to an input port and are given integer names as 1, 2, 3 and \( a_i \) is added after the integer up to the number of ports present. The edge coming from Multiplexer Control Port is also connected to an input port and given the name \( sel_i \). All the input ports are then connected to a Pass node. For the Multiplexer vertex all the in and out edges are removed and consequently the Multiplexer vertex is deleted after all the edges are removed from it. By doing so a Multiplexer is converted to a pass node.

Figure 8.9 shows the sequence of steps carried out to convert a Multiplexer node to a pass node.

Figure 8.10 shows a Multiplexer node. The Multiplexer node is connected to input, output and a control port. The figure shows that the Multiplexer node is not like an ordinary FU. Output ports are connected as input and input ports are connected to the output of the Multiplexer node. Therefore a Multiplexer node has to be converted completely to an FU pass node to make it similar to other node and work with it normally as with any other node.

Figure 8.11 shows a Multiplexer node (shown in figure 8.10) being converted to a pass node. The Multiplexer pass in figure 8.11 gets its input from the input ports and the output from the node goes to an output port.

8.2.1.7 Pass Node Generation for Decoder, Pipeline Class Control, Signal Node and Inline Expressions

For each vertex of type Decoder, Pipeline Class Control, Signal Node and inline expression, a pass node is inserted into the graph. In and out edges are removed from the vertex and connected to the Pass node. After removal of all edges the corresponding vertex is deleted. The pass nodes for inline expressions have the same name as the class names for the inline expressions (given by Simulator Generator 8.1.4). For vertices of types other then inline expression the pass nodes have the same name as the corresponding vertex.

8.2.1.8 Splitting Flip-Flops

Each Flip-Flop in the SSP graph is split into a source and a sink vertices. All the ports connected to a Flip-Flop are removed and connected to the new source and sink nodes. The source Flip-Flop is connected to the output ports and sink to the input ports. The original Flip-Flop is deleted when the ports are connected to the source and sink Flip-Flop. FUs are converted to either source-sink or pass node but the Flip-Flop are only split into source and sink. Flip-Flops are pipeline
Figure 8.9: Multiplexer to Pass conversion flowchart
Figure 8.10: Multiplexer node

Figure 8.11: Multiplexer Pass node
registers an input that comes to Flip-Flop will always be stored.

Figure 8.7 shows Flip-Flop 4 and 5. Source nodes for these two Flip-Flops are shown in figure 8.8. Figure 8.12 shows the sink nodes generated for the Flip-Flops 4 and 5.

Figure 8.12: Flip-Flop Sink nodes

When all the FUs, Multiplexers, Inline Expressions, Flip-Flops, Decoders, Pipeline class controls and signal nodes are converted to their respective nodes and the graph contains none of these node, this graph represents the final SSP Graph.

8.2.2 Topological Sorted List Generation

A topological sorted list of the graph vertices is generated for the SSP graph. The Simulator Mase Generator generates a topological sorted list of SSP graph vertices by using the topological_sort class of the C++ Boost Graph library. The sorted list is later used during the generation of cycle execute methods for the Simulation Runner Class 8.3.

8.3 Simulation Runner Class Generation

The Simulation Runner Class is generated through the Simulator Mase Generator. The contents and the operations performed by the Simulation Runner class may vary every time NoGap simulator is generated. The Simulation runner class
depends on the SSP graph for its structure and implementation. It performs the following tasks

- Instantiating Objects
- Initializing Port Sizes and Values
- Generating Execution Cycle

The list of tasks given above are explained briefly in the later sections. Later sections use the term SimRunner instead of the Simulation Runner Class.

### 8.3.1 Instantiating Objects

Classes generated through Simulator Generator are used by SimRunner. SimRunner instantiates objects for Flip-Flop, FU, Multiplexer, Decoder, Signal Node, Pipeline Class control and Inline Expressions. Simulator Mase Generator provides lists containing FU, Multiplexer, Decoder, Signal Node, Pipeline Class control and Inline Expressions. These lists are accessed, instantiating an object for every required vertex.

### 8.3.2 Initializing Port Sizes and Values

The instantiated objects have the containers for the ports. SimRunner has to initialize the ports with their respective sizes. Simulator Mase Generator collects the information regarding the ports and sizes in separate containers. These containers are accessed by the SimRunner, port and sizes for the objects are initialized.

### 8.3.3 Generating Execution Cycle

Cycle execute function is a C++ function that simulates the execution of one clock cycle. This sections explains how an execution cycle is generated for a NoGap Simulator.

#### 8.3.3.1 Execution Cycle Generation Process

the execution cycle method is generated using a list containing the vertices of the SSP graph and the SSP graph. The list of vertices of the SSP graph is sorted in topological order by the Simulator Mase Generator 8.3. This list corresponds to the execution order of the different vertices. Following steps are performed in order to generate the execution cycle.

- Traversing the topologically sorted list
- Identifying the type of each sorted element that is being traversed
- Performing specific operation on the traversed element according to its type

Operations are performed on the vertices according to their types. Following are the operations performed on different types of vertices.
8.4 Simulator Driver

- Output Port: No operation is to be performed if the traversed element is an output port
- Source Flip-Flop: No operation is to be performed on a source Flip-Flop
- Sink Flip-Flop: Read the value from the previous port and write it to the current sink Flip-Flop
- Input Port: Read the value from the previous port and write it to the current input port
- Source: perform execute source operation
- Sink: perform execute sink operation
- pass: perform execute pass operation

In this way the value is read by the subsequent nodes and operations are performed on the input by the simulator.

8.4 Simulator Driver

The Simulator Driver gets the number of execution cycles from the user and then executes the cycle for the required number of times. Simulator Driver has an object of SimRunner. Through this object read methods is called to the global input port which sets a file to be read. This file contains the binary generated by the NoGap Assembler. Then the initialize methods of the SimRunner is called to initialize all the ports with their sizes. Once the input to the simulator is set and all the ports have their sizes, the cycle execute method is called. The result of the operations performed during the cycle can be checked at the end of each cycle. Pointers to the source and sink Flip-Flops are also flipped at the end of each cycle. At the end of given number of cycles, the results are displayed.

8.5 Results

The SimGen implements the core functionality of a cycle-accurate simulator. To test the simulation, we used a simple test processor having the capability to load values into registers, perform addition and subtraction of two registers. There are three instructions that can be used to program the processor namely load, add and sub. The addition functionality, subtraction functionality and registers are defined as Mage FUs in the processor. Decoder is defined as a template FU. Data path of the processor is defined through a Mase FU. The data path Mase FU creates an instance of the register, add, sub, decoder FUs and define the pipeline process for the three possible instructions using the Mage FU instances. Figure 8.13 shows the Mase graph for the test processor. There is a GlobalIn port to which instructions are loaded through a file. The values in GlobalIn port are fed to the decoder which decodes each instruction. Decoder identifies the opcode, source
and destination operands of the instruction and generates control signals for the multiplexers and registers. Two multiplexers are inserted into the Mase graph to manage multiple inputs to the register, and few flip-flops to control the delays in pipeline for the three instructions. Three FU nodes are present in the graph one each for addition, subtraction and register. There are three control signal nodes for the two multiplexers and the register.

The Mase graph is transformed to the SSP graph shown in Figure 8.14. The add and sub FUs, Multiplexers, control signals and decoders are replaced with pass nodes and the register FU is split into source, sink and pass node. Every Flip-Flop is split into source and sink Flip-Flop. From SimGen, C++ classes were generated for GlobalIn port, decoder, add, sub, Flip-Flop, register, Multiplexer, one class each for the three control signals. Objects for these were instantiated through simulation runner class’s initialization method. Input ports and output ports for all the objects and registers for register FU were filled automatically through simulation runner method. The cycle execute method of simulation runner class was generated from the SimGen.

Figure 8.15 shows the classes automatically generated from SimGen for the test processor and the relationship between them. All the classes are derived from PU base class. Simulation runner class consists of composition of all objects of the generated classes. The execute methods within the classes are empty after running SimGen. These methods were filled with C++ code manually by looking into the NoGap CL code for the processor, and from the Verilog code generated for the processor through already existing verilog generator spawner. Finally an object of the Simulation runner class was instantiated through the simulation driver program and the initialize method was called to instantiate all objects and fill the ports within objects. The driver program asks the user for the input file, which should be the binary code generated from the assembler generator. The possible instructions generated by assembler is shown in Listing 8.1. As assembly program in NoGap at present have similar notation for registers and immediate values, we look into NoGap CD to identify immediate, source and destination operands in the instructions. The first two operands of add(%WRITE) and sub(%WRITE) are source registers and the third register is destination register. Only load(%WRITE) has immediate value which is the second operand in the instruction. A simple program was written as in Listing 8.2. The listing shows the output from assembler which shows assembly instructions along with the binary code generated for the user’s reference. The binary code of this program was loaded by the Simulation Driver program into the GlobalIn port that reads the instruction. The simulation
Figure 8.13: NoGap Mase graph of test processor
Figure 8.14: SSP graph of test processor
was run for the number of clock cycles specified by the user, and at the end of simulation, the result stored in the internal registers was displayed.

Values in all registers can be checked at the end of each execution cycle to verify the results. Listing 8.3 shows the register values after 14 cycle executions. Empty values means as of yet uninitialized values. In Listing 8.3 register 0, 1 and 2 were loaded with values 8, 9 and 7. The result of addition of register 1 and 2 was saved to register 3 which is 10 in hexadecimal representation. Register 5 and 4 were subtracted and result was stored into register 7. The processor consists of 16 registers, and those registers which are not used for any processing are displayed with blank values.
Figure 8.15: Class diagram for the C++ classes generated through SimGen
### Listing 8.3: Register Values

<table>
<thead>
<tr>
<th>Register</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>[0]</td>
<td>8&lt;16&gt;u08</td>
</tr>
<tr>
<td>[1]</td>
<td>8&lt;16&gt;u09</td>
</tr>
<tr>
<td>[2]</td>
<td>8&lt;16&gt;u07</td>
</tr>
<tr>
<td>[3]</td>
<td>8&lt;16&gt;u10</td>
</tr>
<tr>
<td>[4]</td>
<td>8&lt;16&gt;u03</td>
</tr>
<tr>
<td>[5]</td>
<td>8&lt;16&gt;u08</td>
</tr>
<tr>
<td>[6]</td>
<td>8&lt;16&gt;u07</td>
</tr>
<tr>
<td>[7]</td>
<td>8&lt;16&gt;u05</td>
</tr>
<tr>
<td>[8]</td>
<td></td>
</tr>
<tr>
<td>[9]</td>
<td></td>
</tr>
<tr>
<td>[10]</td>
<td></td>
</tr>
<tr>
<td>[11]</td>
<td></td>
</tr>
<tr>
<td>[12]</td>
<td></td>
</tr>
<tr>
<td>[13]</td>
<td></td>
</tr>
<tr>
<td>[14]</td>
<td></td>
</tr>
<tr>
<td>[15]</td>
<td></td>
</tr>
</tbody>
</table>
Chapter 9

Conclusion

ADL tools are valuable tools that can be used to speed up design times for processor designs. NoGap is a new ADL tool that allows for almost full design flexibility while supporting designers with tedious and erroneous tasks.

The thesis report has covered the implementation of the Assembler Generator and Cycle-Accurate Simulator for NoGap. Thus we have established proof of concept that assembler and cycle-accurate simulator can be generated from NoGap framework and can be used to aid the ASIP designer to verify the designed processor.

9.1 Limitations and Future Work

9.1.1 Assembler Generator

The major improvements necessary in Assembler are position independent code, linking and resource conflict checking.

Serialization definition is to be defined once in AsmGen, and to be used in Assembler. Current Assembler generator generates basic assembler and does not support defining constants. There are features that need to be enhanced. At present immediate values are written in association with register name due to the existing framework structure, which needs to be changed in future.

Help system should be implemented for the assembler. Right now there is no help system. If a user types help from the console for the assembler, it must show all the execution options.

Modules used for serialization of data by the AsmGen 6.3 are not reused. Assembler driver program 6.5 writes the module again to read the files contents. This has to be improved and the modules implemented by the AsmGen should be reused instead of writing the modules once again.
9.1.2 Cycle-Accurate Simulator Generator

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

We have not discussed the solutions for inter instruction loops. And this is not yet implemented in the simulator. However, it is an important future work. The solution to this problem is to find the minimum set of inserted Multiplexer that has to be removed to break all possible inter instruction loops. Then looking at all possible instructions we can create a minim set of removed multiplexer configuration vectors for the data path. Each configuration vector describes a different pipeline architecture to sequentialize. Therefore, for each configuration vector, we can generate the corresponding sequentialized code, i.e. one cycle execution function for each of the possible configuration vectors. Thus depending on the configuration vector at the start of a cycle, the corresponding cycle execution function is called. Future work also includes development of better GUI interface for the generated simulator.

The other improvements necessary for simulator are simulating multiple Mase graphs, checking size for values stored within ports and other processor elements and development of GUI. Though size is collected within SimGen, it is not used in simulator as the size is missing in Flip-Flop.
Bibliography


[27] Stefan Pees, Andreas Hoffmann, Vojin Zivojnovic, and Heinrich Meyr. Lisa—
machine description language for cycle-accurate models of programmable dsp

[28] PEAS Project. ASIP Meister User’s Manual. PEAS Project, 1.1.1 edition,
October 2003.

based architecture description language. In Computer Architecture and High


[31] B. Stroustrup. The C++ Programming Language. Addison-Wesley, 3rd edi-
tion, 1997.


[33] Chao Wang, Stefan Maier, Xiaoning Nie, Xuehai Zhou, and Jinsong Ji. An
accurate multi-processing simulator based on adl. In 2008 Fifth IEEE Interna-
tional Symposium on Embedded Computing, pages 266–271. IEEE, October
2008.

[34] Scott J. Weber, Matthew W. Moskewicz, Matthias Gries, Christian Sauer,
and Kurt Keutzer. Fast cycle-accurate simulation and instruction set gener-
ation for constraint-based descriptions of programmable architectures. In
CODES+ISSS ’04: Proceedings of the international conference on Hard-
ware/Software Codesign and System Synthesis, pages 18–23, Washington,


language for processor hardware description. In Design and Diagnostics of

[37] Vojin Zivojnovic, Stefan Pees, and Heinrich Meyr. Lisa - machine description
language and generic machine model for hw/sw co-design. In Proceedings of
Appendix A

Code Excerpt

**Listing A.1:** Generated Code 1 from Simulator Generator

class FlipFlop
{
  public:
  typedef std::map<std::string, boost::optional<utils::BitNum>> ff_map_t;

  public:
  ff_map_t source_ff_m;
  ff_map_t sink_ff_m;
  ff_map_t *source_ptr_m;
  ff_map_t *sink_ptr_m;

  public:
  ff_map_t::iterator iter_m;

  public:
  FlipFlop();
  void execute_sink();
  void execute_source();

  public:
  boost::optional<utils::BitNum> read(const std::string& ff_name);
  void write(const std::string& ff_name, boost::optional<utils::BitNum> value);
  void finish_cycle();
  void fill_source_ff_map(std::vector<std::string> source_ff_list);
  void fill_sink_ff_map(std::vector<std::string> sink_ff_list);
Listing A.2: Code Generated from Simulator Generator 2

```cpp
boost::optional<utils::BitNum>
FlipFlop::read(const std::string& ff_name)
{
    iter_m = source_ff_m.find(ff_name);
    if(iter_m != source_ff_m.end())
    {
        return boost::optional<utils::BitNum>(*iter_m->second);
    }
    else
    {
        iter_m = sink_ff_m.find(ff_name);
        if(iter_m != sink_ff_m.end())
        {
            return boost::optional<utils::BitNum>(*iter_m->second);
        }
    }
}

void FlipFlop::write(const std::string& ff_name, boost::optional<utils::BitNum> value)
{
    iter_m = source_ff_m.find(ff_name);
    if(iter_m != source_ff_m.end())
    {
        source_ff_m[ff_name] = boost::optional<utils::BitNum>(value);
    }
    else
    {
        iter_m = sink_ff_m.find(ff_name);
        if(iter_m != sink_ff_m.end())
        {
            sink_ff_m[ff_name] = boost::optional<utils::BitNum>(value);
        }
    }
    else
    {
        std::cout << "Flip-Flop not found";
    }
}
```

Listing A.3: Code from Assembler Generator 3

```cpp
class sub : public PuBase
{
```
public:
sub();
void init(const std::string& port_name, 
boost::optional<unsigned int> size);
void execute_sink();
void execute_source();
void execute_pass();

public:
boost::optional<utils::BitNum>
read(const std::string& port_name);
void write(const std::string& port_name, 
boost::optional<utils::BitNum> value);

public:
PuBase::port_map_t input_ports_m;
PuBase::port_map_t output_ports_m;
PuBase::port_map_t in_out_ports_m;
PuBase::port_map_t::iterator map_iter;
};
sub::sub()
{
input_ports_m["a_i"] =
std::make_pair(boost::optional<utils::BitNum>(),
boost::optional<unsigned int>());
inout_ports_m["b_i"] =
std::make_pair(boost::optional<utils::BitNum>(),
boost::optional<unsigned int>());
output_ports_m["res_o"] =
std::make_pair(boost::optional<utils::BitNum>(),
boost::optional<unsigned int>());
}

void sub::init(const std::string& port_name, 
boost::optional<unsigned int> size)
{
if(input_ports_m.find(port_name) != input_ports_m.end())
{
input_ports_m[port_name] =
std::make_pair(boost::optional<utils::BitNum>(),
boost::optional<unsigned int>(size));
} else if(output_ports_m.find(port_name) != 
output_ports_m.end())
{
output_ports_m[port_name] =
std::make_pair(boost::optional<utils::BitNum>(),
boost::optional<unsigned int>(size));
}
boost::optional<unsigned int>(size));
} else if (in_out_ports_m.find(port_name) !=
  in_out_ports_m.end())
{
  in_out_ports_m[port_name] =
  std::make_pair(boost::optional<utils::BitNum>(),
                  boost::optional<unsigned int>(size));
}
regfile_m.execute_sink();
regfile_m.write("res_i", __MUX_1___m.read( "dat_o_1"));
regfile_m.write("wr_en_i", CS_regfile_wr_en_i_m.read( "dat_o_1"));
flipflop_obj.write("__FF_12__InputFF", suber_m.read( "res_o"));
suber_m.execute_pass();
suber_m.write("a_i", flipflop_obj.read( "__FF_4__OutputFF"));
suber_m.write("b_i", flipflop_obj.read( "__FF_5__OutputFF"));
__MUX_0___m.execute_pass();
__MUX_1___m.execute_pass();
flipflop_obj.write("__FF_7__InputFF",
flipflop_obj.read("__FF_6__OutputFF"));
flipflop_obj.write("__FF_8__InputFF",
flipflop_obj.read("__FF_7__OutputFF"));
__MUX_0___m.write("dat_i_1",
flipflop_obj.read("__FF_8__OutputFF"));
__MUX_0___m.write("dat_i_2",
flipflop_obj.read("__FF_9__OutputFF"));
__MUX_1___m.write("dat_i_1",
flipflop_obj.read("__FF_10__OutputFF"));
__MUX_1___m.write("dat_i_2",
flipflop_obj.read("__FF_11__OutputFF"));
__MUX_1___m.write("dat_i_3",
flipflop_obj.read("__FF_12__OutputFF"));
flipflop_obj.write("__FF_14__InputFF",
flipflop_obj.read("__FF_16__OutputFF"));
flipflop_obj.write("__FF_15__InputFF",
flipflop_obj.read("__FF_14__OutputFF"));
CS_regfile_wr_en_i_m.execute_pass();
CS_regfile_wr_en_i_m.write("dat_i_1",
flipflop_obj.read("__FF_15__OutputFF"));
CS_regfile_wr_en_i_m.write("dat_i_2",
flipflop_obj.read("__FF_16__OutputFF"));
flipflop_obj.write("__FF_18__InputFF",
flipflop_obj.read("__FF_20__OutputFF"));
flipflop_obj.write("__FF_19__InputFF",
flipflop_obj.read("__FF_18__OutputFF"));
CS__MUX_0___m.execute_pass();
CS__MUX_0___m.write("dat_i_1",
flipflop_obj.read("__FF_19__OutputFF"));
CS__MUX_0___m.write("dat_i_2",
flipflop_obj.read("__FF_20__OutputFF"));
flipflop_obj.write("__FF_22__InputFF",
flipflop_obj.read("__FF_24__OutputFF"));
flipflop_obj.write("__FF_23__InputFF",
flipflop_obj.read("__FF_22__OutputFF"));
Listing A.5: Code from Assembler Generator 3

struct mnemonic_info
{
    std::string mnemonic_name_m;
    int opcode_m;
    int opcode_size_m;
    int opcode_upper_range_m;
    int opcode_lower_range_m;

    int padding_bits_size_m;
    int padding_bits_upper_m;
    int padding_bits_lower_m;

    template<class Archive>
    void serialize(Archive& ar, const unsigned int)
    {
        ar & opcode_m;
        ar & opcode_size_m;
        ar & opcode_upper_range_m;
        ar & opcode_lower_range_m;

        ar & padding_bits_size_m;
        ar & padding_bits_upper_m;
        ar & padding_bits_lower_m;
    }
}
Listing A.6: Code from Assembler Generator 2

namespace generator
{
namespace types
{
  typedef boost::numeric::interval<size_t> slice_t;
}
}

boost::optional<types::slice_t>
AssemblerGenerator::get_single_slice(const
arch_data::InstructionFormat& ins_fmt, const
arch_data::InstructionField::Type::enum_t& type) const
{
  std::vector<const arch_data::InstructionField*> fields =
    find_fields_of_type(ins_fmt, type);
  assert(fields.size() <= 1); // Only handles one or zero fields.
  if(fields.size())
  {
    const arch_data::InstructionField& field = *fields.at(0);
    return ins_fmt.field_slice(field);
  }
  return boost::optional<types::slice_t>();
}

void AssemblerGenerator::create_instr_serializationfile()
{
  fs::path
    serialization_file(path_m/"instruction_serialization.nogap");
  fs::ofstream instr_serialization_output(serialization_file,
    std::ios::out);
  {
    boost::archive::text_oarchive
      oa(instr_serialization_output);
    oa << instr_info;
void AssemblerGenerator::create_instr_lookupfile() //
{
    fs::path lookup_file(path_m/"instruction_lookup.nogap");
    fs::ofstream instr_lookup_output(lookup_file, std::ios::out);
    instruction_info::instruction_info_map_t::const_iterator
    instr_map_iter;

    for (instr_map_iter = instr_info.instruction_map_m.begin();
        instr_map_iter != instr_info.instruction_map_m.end();
        instr_map_iter++)
    {
        instr_lookup_output << instr_map_iter->first << " " ;
        std::vector<register_info> reg_vec =
        instr_map_iter->second.second;
        BOOST_FOREACH ( const register_info reg_test, reg_vec )
        {
            instr_lookup_output << reg_test.reg_name_m << "[ " <<
            reg_test.reg_upper_range_m << ": " <<
            reg_test.reg_lower_range_m << "] " << ""
            ;
        }
        instr_lookup_output << std::endl ;
    }

Listing A.7: Code from Assembler Generator 5

void AssemblerGenerator::extract_instruction_info(const
parser::ParseUnit& pu)
{
    std::string opname_tmp, opname_short;
    int opcode_assembler;
    std::string ins_elem_name;
    types::slice_t ins_elem_range, instr_size ;

    std::size_t found;

    instr_size = get_instruction_bits(pu);

    instr_info.instruction_size_m = instr_size.upper() -
    instr_size.lower() + 1 ;

    const ncg::InstructionTable& ins_tab =
    *pu.instruction_table_m;
    //! for each instruction in the table
    BOOST_FOREACH(const ncg::InstructionTable::value_type& e,
        ins_tab)
    {

const arch_data::InstructionFormat & ins_fmt = 
e.instruction_format_m;
opcode_assembler = e.op_code_m.get();
found = e.op_name_m.find(":");
if(found != std::string::npos)
{
    opname_tmp =
e.op_name_m.substr(found + 1, std::string::npos);
opname_short = e.op_name_m.substr(0, found);
}
else
{
    opname_short = e.op_name_m;
opname_tmp = e.op_name_m;
}

// Look through all opcodes for the operation
BOOST_FOREACH(const arch_data::OperationInfo::opcodes_t::value_type & oc,
              e.op_inf_m->opcodes()){
    // Find the right opcode
    if(boost::lexical_cast<std::string>(oc) == opname_tmp)
    {
        const arch_data::OpCode::op_code_t & oct =
          oc.op_code();

        BOOST_FOREACH(const
                      arch_data::OpCode::op_code_t::value_type & o, oct)
        {
            BOOST_FOREACH(const
                          arch_data::CanonicalOpName::use_names_t::
                          value_type & n, o.use_name()){
                opname_short = opname_short + "( " + n + ")";
            }
        }
    }
}

ins_elem_range = get_opcode_bits(ins_fmt);
int opcode_upper = ins_elem_range.upper();
int opcode_lower = ins_elem_range.lower();
int opcode_size = opcode_upper - opcode_lower + 1;

types::slice_t padding;
if(get_padding_bits(ins_fmt))
{
    ins_elem_range = get_padding_bits(ins_fmt).get();
}
else
{
ins_elem_range = 0;
}

ins_elem_name = "padding_bits";

int padding_upper = ins_elem_range.upper();
int padding_lower = ins_elem_range.lower();
int padding_size;
if(padding_upper == 0)
{
    padding_size = 0;
}
else
{
    padding_size = padding_upper - padding_lower + 1;
}

mnemonic_info mnemonic_inf( opname_short, 
opcode_assembler, 
opcode_size, opcode_upper, opcode_lower, padding_size, 
padding_upper, padding_lower);

v_token_m.push_back(mnemonic_inf);
const std::string& op_name = 
e.op_inf_m->operation_name();
typedef std::vector<const arch_data::InstructionField*> 
imms_t;
imms_t imm_fields = 
find_fields_of_type(ins_fmt, 
arch_data::InstructionField::Type::IMMEDIATE);

BOOST_FOREACH(const imms_t::value_type& i, imm_fields)
{
    for(std::size_t k=0;k < (*i).controlled_verts_size();k++)
    {
        const ncg::types::vdesc_t& ctrl_vert = 
(*i).controlled_vert(k);
        ncg::graph_vertex_t ctrl_gv(*(dpPu_m->mase_m), 
ctrl_vert);
        ncg::types::vdesc_t find_first = 
ncg::find_first_of_type<ncg::info::Fu>(ctrl_gv, 
op_name).get();
        ins_elem_name = ncg::name(ncg::graph_vertex_t 
(*(dpPu_m->mase_m),find_first));
    }

    ins_elem_range = ins_fmt.field_slice(*i);

    if (ins_elem_name != "padding_bits"")
Listing A.8: Code from Simulator Mase Generator 4

```cpp
template<typename FU>
ncg::types::vdesc_t
SimulatorMaseGenerator::insert_pass_no Pu(ncg::Graph& ssp_mase, ncg::graph_vertex_t& fu gv, const std::string& name)
{
    ncg::types::vdesc_t ssp_vertex =
        ssp_mase.add_vertex(ncg::info::PassWithoutPu(name));

    ncg::graph_vertex_t ssp gv(ssp_mase, ssp_vertex);

    ncg::edge_list_t fu_in_edges, fu_out_edges;

    ncg::types::ncg_t::in_edge_iterator iei, iei_end;
    tie(iei, iei_end) = in_edges(fu gv);
    for(; iei != iei_end; ++iei)
    {
        fu_in_edges.push_back(*iei);
    }

    ncg::types::ncg_t::out edge_iterator oei, oei_end;
    tie(oei, oei_end) = out_edges(fu gv);
    for(; oei != oei_end; ++oei)
    {
        fu_out_edges.push_back(*oei);
    }

    // in_edge_vector(fu gv, fu_in_edges);
```
Listing A.9: Code from Simulator Mase Generator 5

```cpp
SimulatorMaseGenerator::port_size_vec_t
SimulatorMaseGenerator::get_port_sizes(const
ncg::vertex_list_t& vertices, ncg::Graph& ssp)
{
    // std::cout << "inside Get Port Sizes method" << std::endl;
    port_size_vec_t port_size_vec;
    std::string name;
    BOOST_FOREACH(ncg::types::vdesc_t v, vertices)
    {
        const ncg::info::Port* port_info =
            &(ssp).getGraph()[v].getPortInfo();
        arch_data::SignalSize signalsize =
            port_info->getSize();
        boost::optional<unsiged int> numericSize =
            signalsize.getSizeNumeric();
        if(numericSize)
        {
            name = ncg::name(ncg::graph_vertex_t (ssp,v));
            port_size_vec.push_back(std::make_pair(name,
                numericSize.get()));
        }
        else
        {
            boost::optional<GiNaC::ex> ginac_size =
                signalsize.getSizeExpr();
        }
    }
    return port_size_vec;
}
```
```
name = ncg::name(ncg::graph_vertex_t (ssp,v));
if(ginac_size)
{
    port_size_vec.push_back(std::make_pair(name,
        signalsize.setSize(ginac_size.get())
            .getSizeNumeric()));
    std::cout << "GiNac Size Found : \n";
}
name = "";
}
return port_size_vec;
}````
## Appendix B

### Abbreviations

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NoGap</td>
<td>Novel Generator of Accelerator and Processor Framework</td>
</tr>
<tr>
<td>NoGap&lt;sub&gt;CD&lt;/sub&gt;</td>
<td>NoGap Common Description</td>
</tr>
<tr>
<td>NoGap&lt;sub&gt;CL&lt;/sub&gt;</td>
<td>NoGap Common Language</td>
</tr>
<tr>
<td>Mase</td>
<td>Microarchitecture Structure Expression</td>
</tr>
<tr>
<td>Castle</td>
<td>Control Architecture STructure Language</td>
</tr>
<tr>
<td>Mage</td>
<td>Microarchitecture Generation Essentials</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>NOP</td>
<td>No Operation</td>
</tr>
<tr>
<td>VLSI</td>
<td>Very Large Scale Integration</td>
</tr>
<tr>
<td>ADL</td>
<td>Architecture Description Language</td>
</tr>
<tr>
<td>HDL</td>
<td>Hardware Design Language</td>
</tr>
<tr>
<td>VHDL</td>
<td>Very High Speed Integrated Circuit Hardware Description Language</td>
</tr>
</tbody>
</table>

**Table B.1:** Abbreviations 1
<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction Set Processor</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>ISS</td>
<td>Instruction Set Simulator</td>
</tr>
<tr>
<td>I/O</td>
<td>Input / Output</td>
</tr>
<tr>
<td>SoC</td>
<td>System On Chip</td>
</tr>
<tr>
<td>GUI</td>
<td>Graphical User Interface</td>
</tr>
<tr>
<td>TID</td>
<td>Target Instruction Descriptor</td>
</tr>
<tr>
<td>DFS</td>
<td>Depth First Search</td>
</tr>
<tr>
<td>DAG</td>
<td>Directed Acyclic Graph</td>
</tr>
<tr>
<td>STL</td>
<td>Standard Template Library</td>
</tr>
<tr>
<td>BGL</td>
<td>Boost Graph Library</td>
</tr>
<tr>
<td>LALR</td>
<td>Look Adhead Left-to-right Rightmost</td>
</tr>
<tr>
<td>GLR</td>
<td>Generalized Left-to-right Rightmost</td>
</tr>
<tr>
<td>CFG</td>
<td>Context Free Grammar</td>
</tr>
<tr>
<td>cy</td>
<td>Cycle Assign</td>
</tr>
<tr>
<td>mux</td>
<td>Multiplexer</td>
</tr>
<tr>
<td>opname</td>
<td>Operand Name</td>
</tr>
<tr>
<td>ASCII</td>
<td>American Standard Code for Information Interchange</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>CPI</td>
<td>Cycle Per Instruction</td>
</tr>
<tr>
<td>SSP</td>
<td>Source Sink Pass</td>
</tr>
<tr>
<td>PU</td>
<td>Parse Unit</td>
</tr>
<tr>
<td>GI</td>
<td>Global Input</td>
</tr>
<tr>
<td>Gout</td>
<td>Global Output</td>
</tr>
</tbody>
</table>

**Table B.2: Abbreviations 2**