Binary Instruction Format Specification for NoGap

Examensarbete utfört i Datorteknik
vid Tekniska högskolan vid Linköpings universitet
av

Yaochuan Chen

LiTH-ISY-EX--12/4609--SE

Linköping 2012
Binary Instruction Format Specification for NoGap

Examensarbete utfört i Datorteknik
vid Tekniska högskolan i Linköping
av

Yaochuan Chen

LiTH-ISY-EX--12/4609--SE

Handledare: Per Karlström
            ISY, Linköpings universitet

Examinator: Andreas Ehliar
            ISY, Linköpings universitet

Linköping, 29 June, 2012
Nyckelord  ASIP, NoGap, Parser, generator
Abstract

Nowadays, hardware designers want to get a powerful and friendly tool to speed up the design flow and design quality. The new development suit NoGap is proposed to meet those requirements. NoGap is a design automation tool for ASIP, it helps users to focus on the design stage, free them from module connection and signal assignment, or integration. Different from the normal ADL tools which limit users’ design ideas to some template frameworks, NoGap allow designers to implement what they want with NoGapCL. However, NoGap is still not perfect, some important functionalities are lacking, but with the flexible generator component structure, NoGap and NoGapCL can easily be extended.

This thesis will firstly investigate the structure of NoGap from software prospective view, and then present a new NoGap generator, OpAssignGen, which allows users to assign operation code values, exclude operation codes and customize the operation code size or instruction size.

A simple example based on the MIPS instructions sets will be mentioned to give users a brief view of how to use OpAssignGen. After that, the implementation of the new generator will be explained in detail.

What’s more, some of NoGap’s flaws will be exposed, but more suggestions and improvements for NoGap will be given.

At last, a successful synthesis result based on the simple MIPS hardware implementation will be shown to prove the new generator is well implemented. More results and the final conclusion will be given at the end of the thesis.
Finally, I finished my thesis work. The completion of my thesis project is really a good lesson for me. Although the work is tough, I definitely get lots of experience about hardware design, programming and academic writing.

Firstly I would like to give my gratitude to my supervisor Per Karlström and examiner Andreas Ehliar. During my thesis project, I got lots of helps from them. Without their helps, I cannot complete my project successfully. They give me lots of suggestions and advises which help me to improve my thesis significantly.

I deeply appreciate that Linköping University gave me a good study environment. The appreciation also goes to my family, my girlfriend and all of my friends. Thank you for your supports.
## Contents

1 Introduction .......................................................... 3  
  1.1 Background ......................................................... 3  
  1.2 Operation Code Generator ....................................... 4  
  1.3 Thesis and Chapter Overview ................................... 4  

2 Related Work .......................................................... 7  
  2.1 OpCode assignments in Other Architecture Description Language (ADL)s ................................. 7  
    2.1.1 Instruction Definition in LISA ........................... 7  
    2.1.2 Instruction Definition in ArchC ......................... 8  

3 NoGap Architecture .................................................. 11  
  3.1 NoGap System Overview ......................................... 11  
  3.2 NoGap Scanner .................................................... 14  
  3.3 NoGap Parser ...................................................... 14  
  3.4 Construction of Abstract Syntax Tree (AST) in NoGap .......... 17  
  3.5 Symbol Table in NoGap .......................................... 19  
  3.6 Parser Units ....................................................... 19  
  3.7 Generator and Generator Registration ........................ 19  
  3.8 Flow Control and Error Handling ............................... 23  

4 Operation Code Assignment Introduction ......................... 27  
  4.1 Operation Code and Instruction Format in NoGap ............... 27  
    4.1.1 Character OpCode .......................................... 28  
    4.1.2 Instruction Format ......................................... 28  
  4.2 Purpose of Operation Code Assignment .......................... 29  
  4.3 A Tutorial for OpCode Assignment .............................. 29  
    4.3.1 Define the Instruction and OpCode Size ................. 32  
    4.3.2 How to assign OpCode ...................................... 32  
    4.3.3 Auto-completion and Range Assignment .................. 32
List of Figures

3.2 NoGap system architecture from Dr Karlström’s thesis [4] . 13
3.3 Small binary trees generated with the functions in Listing 3.3 17
3.4 Graph vertexs (V’s) inheritance diagram . . . . . . . . . . 18
3.5 Symbol table architecture . . . . . . . . . . . . . . . . . 19
3.6 Generator Flowchart . . . . . . . . . . . . . . . . . . . . . 22

4.1 Instruction format example from Dr Karlström’s thesis [4] . 29

6.1 OpCode representation UML . . . . . . . . . . . . . . . . . . . 45
6.2 OpCode Assignment Flowchart . . . . . . . . . . . . . . . . . 48
6.3 AssignOpHandler and AssignOpCodes Unified Modeling Lan-
guage (UML) . . . . . . . . . . . . . . . . . . . . . . . . . . 49
6.4 A Simple AST generated from the Assignment in Listing 6.5 52
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>Instructions’ \texttt{OpCodes} in MIPS Architecture</td>
<td>28</td>
</tr>
<tr>
<td>4.2</td>
<td>All possible \texttt{OpCodes} generated by the incomplete \texttt{OpCode} definition</td>
<td>33</td>
</tr>
<tr>
<td>6.1</td>
<td>Vertex type</td>
<td>44</td>
</tr>
<tr>
<td>8.1</td>
<td>Synthesizing Results</td>
<td>62</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Background

There is a dream of designing hardware quickly and implementing designs easily just like software development. In the past several years, some development tools and hardware languages have tried to realize this dream. However, those tools have problems to balance the different requirements: rich/powerful design possibilities and a simple design procedure. The classic examples are some ADL tools. Predefined templates are used to simplify the design flow, which give designers an abstract HW view and free them from connection details. However, the price of such convenience is high sometimes. Designers cannot touch the architecture detail and their creativity are restricted by templates. This limitation is not good for novel HW designs.

On the other hand, Hardware Description Language (HDL)s such as Verilog allow designers completely control what they want. Then designers have a relative open space to turn their ideas to reality, but sometimes it is not convenient and efficient either, especially for the instruction driven architectures. Designers have to take care all details, assign every control signal bit, check every instruction and merge them into the architecture. What’s more, HDLs are really not friendly for the beginner or people who haven’t so much hardware knowledge.

The new hardware design tool NoGap is proposed to solve above problem. NoGap tries to give designers possibilities to manage micro architecture details and convenient templates to simplify the design procedure at the same time.

Like the Transaction-level modeling [5], NoGap separates the details of communication from the details of functional implementation. All function
units in the system contains two different part, Micro Architecture Generation Essentials (Mage) which describes the component functionality, and another part Micro Architecture Structure Expression (Mase) describing the connection details between the function units.

From the software view, NoGap is similar to a compiler, which contains three levels: front-end, intermediate representation and back-end. In NoGap, the back-end is made up of generators and spawner’s. This back-end interface is fairly flexible. NoGap’s functionality can be extended easily by adding new generators.

1.2 Operation Code Generator

In this thesis, I will mainly focus on the back-end and introduce, the OpCode Assignment Generator, a new NoGap generator which allows designers to customize operation code value, exclude unused operation units and define instruction or opcode length.

After the operation code customization, the processor modeled by NoGap may recognize and execute multiple instruction sets. In other words, the executable binary file compiled in other instruction sets can be executed by that processor. This new generator improves designs’ customization ability and extends the compatibility at the same time.

However, during the design of OpCode assignment generator, we have to face these questions: how to extend NoGap CL, how to design and simplify operation code assignment, what new features we should achieve for these customization, and what the structure of new generator we should use from software view. Next chapters in this these will show our implementation in detail and explain reason inside.

1.3 Thesis and Chapter Overview

This thesis will explain the architecture of NoGap from a programmer’s perspective, introduce the new OpCode Assignment Generator, and illustrate how to use the new functionality with some use cases.

The goals I want to achieve in this thesis are listed below:

1. Explain the architecture of NoGap
2. Illustrate how to assign OpCode by some examples
3. Introduce the OpAssignGen
4. Explain the implementation of the new generator in detail
5. Give some possible software improvement suggestions for NoGap

6. Show system synthesis results

The following list shows every chapter’s content in brief:

- **Chapter 1** gives introduction about the thesis.
- **Chapter 2** shows the comparison among different hardware design tools.
- **Chapter 3** explains the NoGap architecture from a software perspective.
- **Chapter 4** introduces the OpCode assignment and shows some examples.
- **Chapter 5** describes the implementation of OpCode Assignment Generator in detail.
- **Chapter 6** offers improvement suggestions about NoGap.
- **Chapter 7** gives a result about the synthesist hardware report.
- **Chapter 8** makes a conclusion of the OpCode Assignment Generator.
- **Chapter 9** shows the future work.
Chapter 2

Related Work

When we design new processors, a good instruction sets is very important. In this chapter, I will discuss issues about instruction size and OpCode size. After that, OpCode assignments in other ADL will be shown as a comparison.

2.1 OpCode assignments in Other ADLs

Currently, there are lots of ADLs developed to simplify the hardware design. The ADLs can help designers to generate multiple outputs including compiler, simulator, assembler and debugger. All these outputs are based on the abstract hardware description. These ADLs require designers to specify the architecture of processor, memory and the connection of components [7]. I will not compare these ADLs with NoGap here, because such comparisons have been shown in Dr Karlström’s thesis [4]. Because of the importance of instruction description for hardware design, I want to firstly talk about the instruction definition in other ADLs. The following examples will show how to define OpCode in LISA [10] and ArchC [11].

2.1.1 Instruction Definition in LISA

Lisa is an ADL mixing with the features of structural and behavioral ADLs. This means Lisa may touch the lower to RTL abstraction level and hide the gate-implementation, but it is also possible to abstract behavioral information to simplify the instruction sets extraction [6].

The operation definition in LISA uses an operation tree to show the relation among different operation declarations. Different with NoGap, LISA focuses on an Instruction Set Architecture (ISA) which captures all
of information of instructions such as syntax, behavior and encoding [6]. Then, designers can add or map structural descriptions based on these instruction description. Listing 2.1 shows how to define the ADDI instruction in LISA. We may notice the operation code value and instruction format are defined in CODING block. Furthermore, with the keywords SYNTAX and BEHAVIOR the instruction name and behavior are also defined at the same time.

There is no doubt that the LISA’s instruction definition grammar is more powerful than NoGap’s. Currently, we can not set instruction format in NoGap. Operation code definition grammar can only assign the numerical value to operation code, and instruction names and behaviour have to be assigned in different place. Actually, NoGap has a different view with LISA, which describes and abstracts hardware through their functions and operations. In other words, designers define instruction sets after hardware structures and functionalities have been defined.

So, the instruction definition is not the most important part in NoGap. On the contrary, all structural or further descriptions in LISA are based on ISA. The instruction definition grammar must be powerful to make sure instruction sets can be described in detail.

### Listing 2.1. Instruction Definition in LISA from the paper [9]

```plaintext
/*
 * The operation ADDI is defined
 * Its instruction format is clearly shown in list.
 * The pipeline or the execution phase has shown.
 * Its OpCode is 01100 in binary format.
 */
OPERATION ADDI IN pipe.DC {
    DECLARE {
        GROUP rs1, rd = fix_register;
        LABEL immediate;
    }
    CODING { 0b001000 rs1 rd immediate=0bx[16] }
    SYNTAX { "ADDI" rd "," rs1 "," immediate=%d }
    BEHAVIOR { rd = rs1 + immediate; }
}
```

### 2.1.2 Instruction Definition in ArchC

ArchC is an open-source ADL based on SystemC. It can be used to describe a large range of processors. Mostly ArchC is used to verify and explore a new architecture instead of the real hardware implementation. ArchC
can also automatically generated tools such as simulator, compiler and assembler to help designers experiment and evaluate new ISAs [1].

The final outputs of ArchC is SystemC programs. We can find that the syntax style of ArchC are heavily borrowed from C++/SystemC. This inputs relies both structural and instruction-level information [7]. Normally, the ArchC inputs are composed with two parts: Architecture Resources starting with AC_ARCH and Instruction Set Architecture starting with AC_ISA.

Listing 2.2 shows how to define the instruction format and declare instruction in ArchC. In the block AC_ISA, all information are provided to construct a decoder. We can find that from line 2 to 3, two different instruction formats are defined. After that, two instructions, add and load, are declared. On line 11, the add instruction’s operation code value is assigned. The notable thing is designers may define multiple OpCode fields and assign them easily. The grammar and functionality for instruction definition is powerful also and well designed. Although instruction size and operation code can be customized with new generator, the adjuration of instruction format is still not flexible in NoGap. Compared with ArchC, the operation code definition in NoGap is more clear and simple to use, but it is necessary to improve current grammar and add more functionalities, and ArchC is a good reference for the improvement of NoGap.

Listing 2.2. Instruction Definition in ArchC from the manual [1]

```c
1 AC_ISA(mips) {
2    ac_format Type_R = "%op1:6␣%rs:5␣%rt:5␣%rd:5␣0x00:5␣%op2:6";
3    ac_format Type_I = "%op1:6␣%rs:5␣%rt:5␣%imm:16";
4
5    DECLARE the instruction
6    ac_inst<Type_R> add;
7    ac_inst<Type_I> load;
8
9    ISACTOR(mips) {
10       add.set_asm("add␣%reg,␣%reg,␣%reg", rd, rs,(rt);
11       add.set_decoder(op1=0x00, op2=0x20);
12       load.set_asm("lw␣%reg,␣%imm(%reg)", rt, imm, rs);
13       load.set_decoder(op1=0x23);
14    }
15 }
```
Chapter 3

NoGap Architecture

Because the implementation of the operation code generator (OpAssignGen), need to touch NoGap deeply, it is better to firstly understand what NoGap is and how NoGap works. In this Chapter, NoGap’s architecture will be explained in brief. I will mainly focus on the parser, symbol table, AST, parser unit, generator, control flow and error handling in NoGap.

3.1 NoGap System Overview

NoGap is ADL tool for hardware designers. “The underlying design principle in NoGap is based on the assumption that designing individual modules, e.g. adders, multipliers, or Arithmetic Logic Unit (ALU)s of a pipelined architecture is generally a fairly simple task for a human, even for relatively inexperienced designer.” – – from Dr Karlström’s thesis [4]

NoGap simplifies designers’ work, improves the design quality and generates multiple outputs, such as assemblers, simulators, and synthesizable Register Transfer Language (RTL) code. The working flow of NoGap is shown in Figure 3.1. [4]

The idea behind NoGap is to let computer finish the repeated and complex works such as instructions merging for a pipelined Application Specific Instruction-set Processor (ASIP) architecture, assigning control signals and managing delays. Designers just need to finish simple or creative works like instructions definition, specific the modules and the relations between these modules. This strategy helps designers focus on innovation and novelty. To realize such idea, NoGap use a input which has the descriptions of hardware functionality, detailed data path structure and definitions of instruction sets.

The architecture of NoGap is split into three parts. They are facets,
**Figure 3.1. NoGap**

NoGap-core, and spawners. Figure 3.2 shows an overview of NoGap. From this figure, we may notice that NoGap is really like a compiler. A normal compiler is composed by three parts: front-end, intermediate representation and back-end. Then, the parser in the facet corresponds to the front end which scans, parses the input, and generates the intermediate representation. The input files for NoGap are normally some HW descriptions written in NoGap\(^{CL}\). NoGap\(^{CL}\) is used to describe the construction in NoGap. The usage of the language is not important in my thesis, and the detail introduction about NoGap\(^{CL}\) can be found in [4]. NoGap Common Description (NoGap\(^{CD}\)) is the intermediate representation in NoGap, which contains a collection of Parse Units (PUs). In NoGap, a PU is represented by the class ParseUnit. This class has not only the AST but also some other important members, such as the annotated data flow graph Mase, information needed by instruction decoders construction or certain data needed by various generators.

A spawner is arranged to complete a task. The idea of spawner is just a design concept. Normally, a spawner is made up of multiple generators which will process the intermediate representation data and finally generate some outputs, such as Verilog files, simulator or instruction table. Generators are components in NoGap and they are independent with each other. They can be configured and registered separately in NoGap system, which means generators can be added or removed easily. In other words, the functionality in NoGap can be extended easily also.

There is no doubt NoGap is similar with compiler, but it is not just a compiler. In the following sections, I will dive into NoGap’s processing flow and explain why NoGap is different from a compiler. At the same time, the generator architecture, NoGap’s data structure and some important
Figure 3.2. NoGap system architecture from Dr Karlström’s thesis [4]
classes, functions in the source codes will be illustrated as well.

### 3.2 NoGap Scanner

Same with the normal compiler, NoGap front-end is composed by two parts: scanner and parser. First of all, NoGap will break down the input into a list of tokens. The NoGap\(^{CL}\) grammar is borrowed from C++ and Verilog heavily, so the syntax elements in NoGap\(^{CL}\) are similar with the languages mentioned above, which includes identifiers, integer numbers, operators and so on. NoGap uses flex to generate the scanner. Flex \([8]\), the fast lexical analyzer, is a tool for generating scanners, which uses pairs of regular expressions and C/C++ code to describe lexical patterns. The NoGap scanner descriptions are defined in one file. Listing 3.1 shows part of scanner definition. On line 2 to 5, they defines how the comment types are handled. From line 9, some return types, or in other words returned tokens are defined for digital and certain NoGap\(^{CL}\) keywords. When the scanner gets such keywords from the inputs, the tokens will be returned and passed to the parser for further processing.

```c
1 // comment type definitions
2 "/*" BEGIN(comment);
3 <comment>"\n" {ncd1loc.first_line++;}
4 <comment>"*/" BEGIN(INITIAL);
5 <comment>. {}
6 // comment type definitions end
7
8 // keyword definitions
9 "fu" { return TT_FU; }
10 "template" { return TT_TEMPLATE; }
11 "main" { return TT_MAIN; }
12 ...
```

### 3.3 NoGap Parser

The parser is most important part of the front-end in NoGap. In this chapter, I will explain how the parser is generated in detail and illustrate the generation of AST, because my new generator highly relays on the parser and AST. To implement the OpCode customization functionality, the scanner and parser in NoGap have to be extended, so some illustrations
such as how to define new rules in NoGap parser, and how to insert nodes or edges in the AST are necessary.

The parser in NoGap is generated by Bison [2], which is a LR parser. The definition is a BNF grammar which declares the makeup of some key concepts in NoGapCL such as statements, expressions and clauses. Bison uses rules to express the grammar. Normally, a rule is composed by several terminal or non terminal symbols. The terminal symbol in Bison is token type which stand for keywords in the language. In Bison, there are two types of terminal representation. One is identifier name and another is C string constant characters. The rule will follow a set of of actions which are executed when the rule matches the input. All rules defined for the grammar will execute and merge recursively. Normally, the parser will generate a tree in the end. In our case, we will get an abstract syntax tree represented with boost graph [12].

Listing 3.2 defines if and elseif statements. I will utilize some examples to explain how nodes and edges are inserted into the AST. Line 1 defines how to get the if_stmt node. When the parser gets a key word if, the parser will try to match the remaining terminal or non terminal symbols with the inputs. If they are matched, the following action will be executed. The function insertNode will be invoked to insert a node of type If into the AST. We might notice that there are some special variables like $$ and $3 in the action. These variables represent the corresponding positions in the rule. The $$ means the start symbol in the rule. In our case, the start symbol is if_stmt, and $1 stands for TT_IF where is the first place after the colon in the rule. $3 to $5 represent the corresponding symbols on that places. $$ is given a value which is returned by function insertNode. I will explain that function and the AST in detail later. The elseif_stmt is more complex than if_stmt, because the elseif_stmt has more possibilities. In NoGapCL, we may use multiple elseif blocks or just end with one else block. From line 12 to line 27 in Listing 3.2, all possible elseif_if statements are listed. Their actions are similar with the action in the if statement.

The if_stmt and elseif_stmt are assigned to a value returned by function insertNode. The type returned by insertNode is a vertex descriptor. I will illustrate the descriptor later in NoGap AST. In the Listing 3.3, two versions of insertNode function are defined. The first version accepts two nodes and edges as input arguments and finally generate a small binary tree. Figure 3.3 shows two types of trees generated by that two version of insertNode functions. The second argument, EdgeOrder, is an index for edges. It is used to traverse the tree from one node to an-
Listing 3.2. parser definition

```c
// defines a rule about how to handle the if statement
if_stmt: TT_IF '(' expr ')' statement
{
  $$ = insertNode(parse_data.ast(), info::If(),
          info::EdgeOrder(), $3, $5, @$.first_line);
}
/

/* defines all possible elseif statements */
*/
elseif_stmt: TT_ELSEIF '(' expr ')' statement
{
  $$ = insertNode(parse_data.ast(),
                 info::ElseIf(),
                 info::EdgeOrder(),
                 $3, $5, @$.first_line);
}
| TT_ELSEIF '(' expr ')' statement elseif_stmt
|
| TT_ELSE statement
{|}
$$ = insertNode ...

```

Listing 3.3. Function insertNode Definition

```c
/*
 * this function generated a small binary tree
 * which the root node has two child nodes.
 */

vdesc_t parser::ast::insertNode(Graph &g, const info::Statement &ninf,
    const info::EdgeOrder &einf1,
    const vdesc_t &child1,
    const info::EdgeOrder &einf2,
    const vdesc_t &child2,
    const unsigned int &line)
{
    // this function get a simple tree which has only two nodes at all.
    vdesc_t parser::ast::insertNode(Graph &g, const info::Statement &ninf,
        const info::EdgeOrder &einf,
        const vdesc_t &child,
        const unsigned int &line)
```
other one. The argument \texttt{ninf} in the function designate the current root node of the tree. In the case I mentioned above, the root node is the class \texttt{parser::ast::info::If} which inherits from the class \texttt{Statement}. That class is the base class for all node types. The last function argument, \texttt{line}, means the line number of root node in the input file. We may use this line number later to tell users where the problem is when an error is caught.

3.4 Construcion of AST in \texttt{NoGap}

In \texttt{NoGap}, the AST normally represents as \texttt{Mage}. We use symbol \texttt{Mage Vertex} (\(V_{\text{Mage}}\)) and \texttt{Mage Edge} (\(E_{\text{Mage}}\)) to stand for the node type and edge type in the \texttt{Mage}. The AST in \texttt{NoGap} is represented with boost adjacency graph. The types of edge and node in the AST are represented with \texttt{info::Edge} and \texttt{info::Statement}.

However, as the base class of \(V_{\text{Mage}}\), class \texttt{info::Statement} has many sub types. Figure 3.4 shows the major of \(V_{\text{s}}\) derived from the \texttt{Statement}. This figure is relative small, but it shows that most of statements and keywords in \texttt{NoGap}^{CL} have corresponding node type in the AST. Although these nodes have different names, they are almost the same except some of the nodes have extra attributes such as the node name or number of statements.

What’s more, the class \texttt{Statement} inherits some methods from interface \texttt{OutStreamable} and \texttt{PolyNewClonable}. The interface \texttt{OutStreamable} contains a print method. The object which implements this interface may return a string representation of the object. The \texttt{PolyNewClonable} interface is used to clone an instance. There are multiple methods implemented in the class \texttt{Statement}. More detailed information about \texttt{Statement} can be found in [4].
Figure 3.4. Vs inheritance diagram
3.5 Symbol Table in NoGap

The symbol table representation in NoGap is the class `parser::SymbolTable`. The core element in this table is class `parser::Symbol`, which records the essential symbol attributes such as type, name, and values. All clauses in NoGapCL locate in the same level without hierarchical structure. Figure 3.5 shows the architecture of symbol table with an UML diagram, the symbol object is basic element in the symbol table. Most of statements gotten by the parser will be represented with the type `Symbol` and kept by the symbol table.

![Symbol Table Architecture](image)

Figure 3.5. Symbol table architecture

3.6 Parser Units

The PU is a core element in NoGap, for every Functional Unit (FU) in the input file. It must exist a corresponding PU which contains all information for that FU. The PU is represented as class `parser::ParseUnit`. Notable members in the PU are AST graph `ast_m`, `Main` graph, operation info, and the Mage dependency graph. Most of these members in the PU are wrapped by the container `generator::GeneratedData<Data>`. This container will keep an uninitialized boost shared pointer which point to the type `DATA` when the `ParseUnit` is generated. The memory used by the `DATA` type will be allocated by a generator when it handles the PU. Some important methods in this container are shown in Listing 3.4. The method `is_initialized` is used to check whether the data is initialized. Normally, this method will be called by some generators to verify whether the pre-dependency data type is initialized. The `initialize` function, on line 29, is used to allocate memory for the data and initialize the shared pointer.

3.7 Generator and Generator Registration

A generator is a software component that processes PUs [4]. Generators are responsible to perform most of tasks in NoGap, such as the symbol table
Listing 3.4. Definition of class GeneratedData

```cpp
// This template is a container for the data required by generators.

template<typename DATA>
class GeneratedData
{
public:
  typedef DATA value_type;

private:
  typedef boost::shared_ptr<value_type> data_t;

  data_t data_m;

public:
  GeneratedData()
  : data_m(data_t())
  {}
  ~GeneratedData()
  {}

  bool is_initialized() const
  {
    return data_m.get() != 0;
  }

  /*
  This function shows how the memory is allocated for the real data.
  */

  initialize()
  {
    assert(data_m.get() == 0);
    data_m = data_t(new value_type());
  }

  ...
};
```
3.7 Generator and Generator Registration

generation, syntax validation check, or even drawing abstract syntax trees. Generators may be independent with each other. They are initialized, released at the beginning of \texttt{NoGap} processing stage, implemented with the generator interface and managed under the generator registration strategy. Based on this strategy, we may easily add new generator to \texttt{NoGap}. In the next chapter, I will introduce a new generator that extends \texttt{NoGap}'s functionality.

Listing 3.5. Generator interface

```cpp
class Generator
{
protected:
  virtual const bool /* The predication function is used to verify the existence of dependencies */
  generate_predicate(const parser::ParseUnit&) const;
  virtual void operator()(parser::ParseUnit& pu) = 0;
public:
  Generator();
  virtual ~Generator();
  bool generate(parser::ParseUnit& pu);
  virtual void initialize();
  virtual void finalize();
  virtual void reset();
};
```

All generators are derived from the base class \texttt{generator::Generator}. Listing 3.5 shows most methods in that base class. For every generator, they may override these methods to do their own tasks without taking care how these methods are invoked. The method \texttt{generate\_predicate} on line 9 is used to check whether the generator should process this PU. A generator will normally override the method on line 10 to implement the processing on the PUs. The method \texttt{generate} on line 14 will call the virtual method \texttt{operator()} to process PUs, and the \texttt{reset} function will reset the generator when the process is finished. Different with the methods mentioned above, the methods \texttt{initialize} and \texttt{finalize} will only run one time when the generator is initialized and released respectively. These two functions may do some extra works for the generator, such as initialize or clear global variables.

The flowchart shown in Figure 3.6 displays a normal generator processing flow.

Besides the universal generator interface, all generator are registered
Figure 3.6. Generator Flowchart
3.8 Flow Control and Error Handling

In NoGap, the configuration file is called flow control file or flow-configuration file. This file controls the generator processing flow and set generators’ options at the same time.

An XML based flow-configuration file is used to configure the generators in NoGap. Listing 3.7 shows what the eXtensible Markup Language (XML) configuration file looks like. We can use the generator element with type attribute to specify what generator we want to run (line 4). A more special case on line 7 uses the extra attribute enabled to specify whether the generator will be run. On line 10, we may use the element option to set some options in the generator, if that generator offers such options. The finish element on line 12 will halt the execution flow. The generators after the finish element will be ignored. For example, the generator gen_assembler and gen_simulator on line 13 and 14 are excluded from the flow. Because of the limited space, the more detailed guidelines about the XML configuration file can be found in Dr Karlström’s paper. [3]

To handle errors in NoGap, some exceptions have been defined in NoGap. Listing 3.8 shows the usage of macro NOGAP_NORMAL_ERROR. This macro will instantiate an exception error::Normal which contains an error message, file name and line number.

On the other hand, the class CompileIssueHandler is used to hold and handle issues. These issues will pass information to users directly during the NoGap compilation stage. The issue is represented as class
Listing 3.7. XML flow file example

```
1 <xml version="1.0" ?>
2 <!DOCTYPE flow SYSTEM "flow_format.dtd">
3 <flow>
4 <generator type="gen_set_file_name" />
5 <generator type="dot_parsed_ast" />
6 <generator type="gen_symbol_table" />
7 <generator enabled="no" type="validate_identifiers" />
8 <generator enabled="no" type="validate_identifiers" />
9 <generator type="gen_op_classes">
10 <option>verbose</option>
11 </generator>
12 <finish />
13 <generator type="gen_assembler" />
14 <generator type="gen_simulator" />
15 </flow>
```

Listing 3.8. Exception in NoGap

```
1 ...
2 throw NOGAP_NORMAL_ERROR("Vertex is neither input nor output port");
3 ...
```

*nogap_system::Issue*. There are three types of issue: error, warning and note. The error issue will stop the NoGap compilation after the current generator finishing its work.

The issue handler structure works like a stack, which issues are put and pop from the handler in order. The handler will reset itself when a generator starts work. The issues raised by that generator during processing are kept by the handler. When the generator completes its work, the handler would check whether there is an issue. If issues are found, they will be processed according the insertion order, which means the message and issue type will be printed one by one. If the handler found an error issue, it would call a method to stop the NoGap compilation. The definitions of *Issue* and *CompileIssueHandler* are shown in Listing 3.9. The class *CompileIssueHandler* is a special type. For the hole NoGap global space, only one issue handler instance would be instantiated. Although technically not correct, the *CompileIssueHandler* can be thought of a static class. Line 13 shows this issue handler is also non-copyable. To get the handler in a generator, we can use the static method *get()* on line 31 to get the handler pointer. The *get()* method allocates memory for issue handler if the static shared pointer defined on line 29 is not initialized, and
then returns the shared pointer directly.

**Listing 3.9. Definition of Issue and Handler**

```cpp
// Definition of Issue
struct Issue
{
    struct severity_t { enum enum_t {INFO=2, WARNING=1, ERROR=0};
    const std::string file_m;
    const unsigned int line_m;
    const severity_t::enum_t severity_m; // the level of Issue
    Issue(const std::string& file, const unsigned int& line,
          const severity_t::enum_t& severity);
};

class CompileIssueHandler: private boost::noncopyable,
    public policy::OutStreamable
{
private:
    // the container for holding all Issues
    typedef std::multimap<Issue,
        boost::shared_ptr<std::stringstream> > issues_t;
    ...
private:
    issues_t issues_m;
    boost::optional<issues_t::iterator> curr_issue_m;

    /* The static link is used to make sure that
    * there is only one issue_handler instance in NoGAP.
    */
    static boost::shared_ptr<CompileIssueHandler> compile_issue_handler_m;
public:
    static CompileIssueHandler& get();
    ...
};
```
Chapter 4

Operation Code Assignment

Introduction

In this chapter, I will explain the OpCode and introduce the purpose of operation code assignment. Then an implementation which is partly compatible with MIPS instructions will be shown to illustrate OpCode assignment.

4.1 Operation Code and Instruction Format in NoGap

The operation code, which is known as OpCode, is a part of instruction which specifies the operations to be executed. The operation code contains multiple bits, and the OpCode value is normally represented as a binary number. Figure 4.1 displays the position of operation code in MIPS, and Table 4.1 shows certain I-type MIPS instructions and their OpCode values. The I-type instruction in MIPS has an immediate operand and supportable instruction format by NoGap.

| opcode(6) | rs(5) | rt(5) | immediate(16) |

With my new generator in NoGap, the size/length of OpCode can be defined by users. On the other hand, the OpCode size decides the maximum value for OpCode. If users do not set the OpCode size manually, A default OpCode size will be set based on the total number of OpCode. For example, if there are 2 OpCode totally, then we only need one bit to represent these OpCode. For 3 OpCode, 2 bits are required.
### 4.1.1 Character OpCode

In \texttt{NoGap}, the character format \texttt{OpCode} is a string representation for the operation code. It is a human readable format to help users define and print \texttt{OpCodes}. The character format \texttt{OpCode} is also used in my \texttt{OpAssignGen} to define different \texttt{OpCodes}, and then the generator may distinguish what \texttt{OpCodes} are going to be assigned by users.

The character \texttt{OpCode} is composed with multiple sub operations, and one sub operation has a function unite name and usage name. For example, operation code \texttt{[t1.%AND, t2.%OR]} denotes that the instruction will perform AND operation in the FU \textit{t1} and OR operation in the FU \textit{t2}. Actually, every sub operation in the \texttt{OpCode} defines what operation should to be performed in the corresponding phase/pipeline. More information about the definition of FU and its usage are shown in Section 4.3. Some important guidelines should be mentioned: an \texttt{OpCode} may contain one or more sub operations, but every FU is only allowed to use one time in that \texttt{OpCode}. For instance, the \texttt{[t1.%AND, t1.%OR, t2.%SUB]} is not a correct \texttt{OpCode}. The operations’ order in \texttt{OpCode} is not an issue. \texttt{[t2.%SUB, t1.%AND]} and \texttt{[t1.%ADD, t2.%SUB]} is the same \texttt{OpCode}.

### 4.1.2 Instruction Format

The instruction in \texttt{NoGap} has a fixed format, Figure 4.1 shows that format, and an \texttt{OpCode} always places at the beginning of instruction. Except the padding bits, the remaining bits in an instruction are immediate data which could be constant numbers, operands or register addresses. Normally, the
4.2 Purpose of Operation Code Assignment

Instruction size is up to the longest instruction. The shorter instructions will be padded by padding bits, in order to make sure every instruction in NoGap has the same size.

![Instruction Format](image)

**Figure 4.1.** Instruction format example from Dr Karlström’s thesis [4]

With the OpAssignGen, users may customize the instruction size/length, but the position of OpCode and immediate data cannot be changed in current version of NoGap.

4.2 Purpose of Operation Code Assignment

My thesis project is to implement a NoGap generator that allows users to customize or exclude the operation code and adjust instruction as well as OpCode size.

Based on the operation code customization, a processor modeled by NoGap can extend its compatibility. A typical example is a new processor designed by NoGap can execute different instruction sets. However, this new generator is only a first step to realize such compatibility. Some supports from the decoder, compiler and assembler are necessary. In Section 4.3, I will use MIPS instruction sets as an example to guide users assigning operation codes.

4.3 A Tutorial for OpCode Assignment

Listing 4.1 shows the implementation for some MIPS instructions, the full list of this example can be found in Appendix A. The pipelined operations are defined in Listing 4.1. In that operation block, signals are assigned and FU usage lists are declared at every phase which starts with symbol @. The FU usage list defines which FU should be used and what possible operations are able to be executed in a pipelined operation. All sub operations mentioned in Section 4.1.1 are gotten from the FU usage list. Then NoGap combines those sub operations randomly and generates all possible combinations, the OpCodes.

The target I want to achieve is to make sure instructions in new designed processor have same operations and operation codes with those of instruc-
Listing 4.1. Operations for ADD unit in MIPS

```c
fu data_path {
  operation(pipe) alu_i(dec_unit.i_type)
  {
    // defines operations in pipeline p_ID
    @p_ID;
    dec_unit.instr_i = instr_i;
    rf;
    rf.addr_a_i = dec_unit.rf_addr_a_o;
    /* function se will use two units
       in this pipeline */
    se '%NOP,%EXTEND';
    se.dat_i = dec_unit.imm_o;
    extended = se.dat_o;
    // defines operations in pipeline p_EX
    @p_EX;
    // alu will use 4 units
    alu '%ADD,%AND,%OR,%XOR';
    alu.dat_a_i = rf.dat_a_o;
    alu.dat_b_i = extended;
    // ov_detect will use 2 units
    ov_detect '%DETECT_OVERFLOW,%NO_OVERFLOW';
    ov_detect.dat_i = alu.res_o;
    @p_WB;
    // rf only has 1 unit.
    rf '%WRITE';
    rf.addr_wr_i = dec_unit.rf_wr_addr_o;
    rf.dat_i = ov_detect.dat_o;
    dat_o = ov_detect.dat_o;
  }
} ...
```
4.3 A Tutorial for OpCode Assignment

In other words, the processor modeled by NoGap may get the compatibility with MIPS instructions sets. Since NoGap is not able to accept all formats of MIPS instructions, I just implemented part of MIPS instructions sets.

The instructions in this table are only I-type instructions, because the other types of instructions in MIPS have different instruction format which can not be supported by the current version of NoGap. For instance, the format of R-type instruction is not supported by NoGap, because there is no immediate value in that type of instructions.

According the inputs in Listing 4.1, NoGap will finally generated 16 instructions. Listing 4.2 shows these instructions with default OpCode values without OpCode customization. The immediate data, port numbers and timing have been ignored for simplification.

**Listing 4.2. Operations for ADD unit**

```c
1 /*
2 * The first field before the colon is opcode value,
3 * for example 5<16>u00 shows the opcode value is 00 in
4 * hexadecimal format and its length is 5.
5 * The next part is the character opcode
6 * showing what the instruction is.
7 */

9 /*opcode value**************character opcode*************/
10 5<16>u00: alu.%ADD,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP
11 5<16>u01: alu.%ADD,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%EXTEND
12 5<16>u02: alu.%ADD,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%NOP
13 5<16>u03: alu.%ADD,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%EXTEND
14 5<16>u04: alu.%AND,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP
15 5<16>u05: alu.%AND,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%EXTEND
16 5<16>u06: alu.%AND,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%NOP
17 5<16>u07: alu.%AND,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%EXTEND
18 5<16>u08: alu.%OR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP
19 5<16>u09: alu.%OR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%EXTEND
20 5<16>u0a: alu.%OR,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%NOP
21 5<16>u0b: alu.%OR,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%EXTEND
22 5<16>u0c: alu.%XOR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP
23 5<16>u0d: alu.%XOR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%EXTEND
24 5<16>u0e: alu.%XOR,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%NOP
25 5<16>u0f: alu.%XOR,ov_detect.%NO_OVERFLOW,rf.%WRITE,se.%EXTEND
```

This group instructions have no immediate data field and their OpCodes have the same values. A function code at the end of instruction is used to differentiate this type of instructions.
4.3.1 Define the Instruction and OpCode Size

The OpCode size in Listing 4.2 is 5 which is not we expect of. Table 4.1 shows OpCodes in MIPS have size 6, so we should define the size in the fu data_path block as following:opcode_size 6;

I always suggest users to firstly consider and configure the size of OpCode and instruction before any operation code assignments. Because The OpCode size limits the maximum number you may use on OpCode assignment, and the instruction size must be greater than the OpCode size. For example, if OpCode size is 20, then the maximum value of the OpCode is \(2^{20}-1\). Furthermore, the generator has a default OpCode size calculated with the total amount of OpCodes in the current Masu, so the OpCode size must be not less than the default size also.

4.3.2 How to assign OpCode

Next, we need to assign the OpCodes in Table 4.1 manually. These assignments have been shown in Listing 4.3. An operation code assignment should always be in the operation block and start with the keyword opcode. The character OpCode is used to define what OpCodes are going to be assigned. In Listing 4.3, values on the right hand side of the expression are decimal numbers, but the binary or hexadecimal number can be used as assigning value also. For example, 6<2>000001 defines a value with 6 bit length in binary format.

4.3.3 Auto-completion and Range Assignment

The operation code assignment generator also support auto-completion and range assignment. The auto-completion functionality allows users to ignore some sub operations in an OpCode assignment. For example, the sub operation rf.%WRITE can be ignored in the assignment on line 13, because the FU rf under the phase p_WB only has one operation WRITE and the generator can detect there is a missing sub operation and complete the definition automatically.

However, if we ignore a sub operation which has more than one possible usages, the generator with auto-completion will finally generates all possible OpCodes. For instance, if the assignment on line 15 in Listing 4.3 ignores sub operations in the alu, we will get all possible candidate OpCodes which are combined

\[\text{se.%NOP, ov_detect.%NO_OVERFLOW, rf.%WRITE}\]
Listing 4.3. Operation Code Assignment for instructions in the ADD unit

```c
{fu data_path {
    operation(pipe) alu_i(dec_unit.i_type) {
        ...
        /*
        * The following FU names and their possible operations
        * have been declared in the FU usage lists under phases
        * These declarations can be found
        * in Listing 4.1
        */
        opcode{
            //ADDI (with overflow)
            {se.%EXTEND,alu.%ADD,ov_detect.%DETECT_OVERFLOW,rf.%WRITE} = 8;
            //ADDIU (no overflow)
            {se.%NOP,alu.%ADD,ov_detect.%NO_OVERFLOW,rf.%WRITE} = 9;
            //ANDI (no overflow)
            {se.%NOP,alu.%AND,ov_detect.%NO_OVERFLOW,rf.%WRITE} = 12;
            //ORI (no overflow)
            {se.%NOP,alu.%OR,ov_detect.%NO_OVERFLOW,rf.%WRITE} = 13;
            //XORI (no overflow)
            {se.%NOP,alu.%XOR,ov_detect.%NO_OVERFLOW,rf.%WRITE} = 14;
        }
    }
}
```

with operations in the FU alu. All possible completed OpCodees are shown in Table 4.3.3, it is notable that the only difference for these completed OpCodees is the sub operations from FU alu.

<table>
<thead>
<tr>
<th>se.%NOP,ov_detect.%NO_OVERFLOW,rf.%WRITE</th>
</tr>
</thead>
</table>

Table 4.2. All possible OpCodees generated by the incomplete OpCode definition

To finish the incomplete assignment se.%NOP,ov_detect.%NO_OVERFLOW, rf.%WRITE, users have to use a range of values as the right hand side operator, because there are four OpCodees waiting for assigning. Listing 4.4 shows the range assignments for the incomplete OpCode. There are total five available values in the range [14:10]. The range assignment can be completed successfully, except a warning message will be arise to inform
users that only four values in the range have been used.

The size of range should be always greater than the amount of assigning OpCodes. For example, if the range $[14:12]$ \(^2\) is used to assign OpCodes listed in Table 4.3.3, an critical error will terminate the processing, because the amount of values in the range $[14:12]$ are not enough for all OpCodes. The values assigned to OpCodes in range assignment are not in order and may be any numbers in the range. The range assignment only helps users to specify certain OpCodes in a range space. If users want to assign specific values to OpCodes precisely, they should assign values to OpCodes one by one.

Some important guidelines should be followed for the assignment. Users may overwrite operation codes in OpCode assignment, but they should check the assigned value carefully, guarantee assigned values are unique and valid. When users define a range for the assignment, the range upper limit should always be greater than the lower limit, if not an error will be generated. What’s more, the range size must be large enough to make sure all OpCodes can get a value from the range.

After the operation code customization, OpCode’s values in these instructions are same as values in MIPS architecture. The results are listed in Listing 4.5. We can find that OpCodes on line 3, 5, 10, 15, 20 have been assigned with correct values. The size of OpCode has been 6 now, and the instruction size can be set with keyword instruction_size as well. But the default instruction size has been 32 bits, it is no necessary to set instruction size again.

4.3.4 OpCode Exclusion

In Listing 4.5, there are still some unnecessary instructions. OpAssignGen offers exclusion functionality to eliminate the OpCodes we don’t want to

\(^2\)the range size is 3, the range has number 12, 13 and 14.
keep. For example, MIPS instruction sets have no instructions XOR and OR with overflow detection, so OpCodes on line 17 and 12 are useless. We can exclude these OpCodes easily with the symbol '!'. Listing 4.6 shows how to exclude OpCodes. When we exclude an OpCode, we should check the

**Listing 4.6. Operation Code Exclusion**

```plaintext
... 
opcode{
    !{alu.%XOR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP};
    !{alu.%OR,ov_detect.%DETECT_OVERFLOW,rf.%WRITE,se.%NOP};
}
```

existence of OpCode firstly and make sure the OpCode has not been assigned previously. Excluding OpCodes which have been excluded or assigned are valid, but certain warnings messages will prompt to inform the abnormal behaviors. One important thing is we cannot use incomplete OpCode to exclude multiple OpCodes in one statement, because excluding OpCodes is dangerous action for the generator. To make sure all OpCodes are excluded correctly, all exclusions should be defined clearly.

With the Verilog generator in NoGap, the example in Listing 4.1 will finally get the hardware implementation. In Chapter 8, the output will be
synthesized to verify the hardware implementation.

4.4 OpAssignGen Configuration

Finally, I want to introduce how to configure the OpAssignGen. As mentioned in Section 3.8, the generator can be configured with an XML control flow file. In this file we may set what generator we want to use and which options will be used on the generator. OpAssignGen has two options: verbose and autocomplete. Listing 4.7 shows how to configure the OpAssignGen.

Listing 4.7. OpAssignGen Configuration

```xml
1 <generator type="gen_assign_opcodes" >
2   <option>verbose</option>
3   <option>autocomplete</option>
4 </generator>
```

OpAssignGen is specified with name: “gen_assign_opcodes” in the flow file. The <option> label marks what options are enabled. When the verbose option is set, all level of information will be printed by the generator. The autocomplete option will turn on the auto-complete functionality, an incomplete OpCode declaration will be completed automatically.
Chapter 5

OpCode Assignment Syntax

5.1 Introduction

To customize the operation code value, exclude operations, and adjust instruction and OpCode size in NoGap, current NoGap<sup>CL</sup> must be extended. This chapter aims to introduce the new OpCode assignment grammar, and describe the idea behind the grammar.

5.2 Design of Assignment Grammars

When we extend grammar for some languages, the most important guideline is to follow the syntax style in the old language. The new OpCode assignment grammars should be familiar by users easily, so I need to make sure the new grammar is integrated well.

An OpCode assignment is made up of two parts: OpCode definition and value or range. The OpCode definition is left hand side operator, and the value or range is right hand side operator. These two operators are connected with the symbol ‘=’, so the operation code assignment looks like a variable assignment.

The format of OpCode definition borrows from the character OpCode format in the instruction manual generated by NoGap. The character OpCode format is used by NoGap globally to help users distinguish different OpCodes. It is readable well for human, although this format is not convenient when assigning lots of OpCodes. The detailed explanation about character OpCode can be found in Section 4.1.1.

A typical OpCode definition is composed by one or more sub operations, and a sub operation contains a FU name and usage name, e.g. t1.%ADD. FU and usage name are connected with operator ‘. ’, the dot operator is a good
connector, because it shows the relation that the usage is one of elements in the FU. A pair of curly brackets around the OpCode definition helps compiler to find the beginning and end of definition. The original character OpCode format use square brackets to mark the definition, e.g. \([t1.%add, t2.%sub]\), but the symbol, square bracket, has been used to declare a value range in \(\text{NoGap}^{CL}\). If the square bracket was used in OpCode definition, a shift reduce error would be arise by the compiler, since the syntax is ambiguous for the compiler.

When users eliminates OpCode, an OpCode definition starts with symbol ‘!’ tells the generator what OpCode should be excluded from operations. This symbol clearly shows the intention of the statement, and it is also simple to use for those who are familiar with C-like languages.

Anyhow, all OpCode assignments and exclusion should be in a block starts with the keyword opcode. This keyword tells the compiler and generator that the following statements are OpCode assignment statements. Theopcode block is placed in the operation block, and different operation block has their own opcode block also. This strategy helps user to handle the right OpCode in the right place.

On the other hand, the instruction size and OpCode size are different with the OpCode assignment. I consider both of two sizes as constant variables in \(\text{NoGap}\). These two variables are important attributes for Masc unit. In other words, the instruction size and operation code size can only be initialized at the beginning and overwritten is not allowed. Like what a typical programming languages does for the initializing, the two keywords, instruction_size and opcode_size, are also constant variable names and follow a numerical number as initial value. This two variables are global variables, in other words, they can only be initialized one time. What’s more, all FUs have the same operation code and instruction size.

5.3 Grammar for Assigning OpCode

As mentioned before, the grammar for OpCode assignment is clear and simple to use. The following sections will show the assignment and exclusion grammar in Backus-Naur Form (BNF) format.

5.3.1 Opcode Block

**Grammar 5.1** opcode statement

\[
\langle \text{opcode} \_\text{cons} \rangle \rightarrow \text{opcode}\{ \langle \text{opdef} \_\text{stmts} \rangle \} \]


Grammar 5.1 shows how to start **OpCode** assignment with the **opcode** keyword. All **OpCode** assignments or exclusions in Mase*CL take place within an **opcode** block and the **opcode** block is in an **operation** block which defines the pipelined operation. An example of **operation** block can be found in Listing 4.1 (And more examples of completed operation blocks can be found in [4].)

### 5.3.2 Operation Code Definition

**Grammar 5.2** opcode assignment grammar

\[
\text{⟨opdef_stmts⟩} \rightarrow \text{⟨opdef_stmts⟩ } \text{⟨opdef_stmt⟩} \\
| \text{⟨opdef_stmt⟩} | \emptyset \\
\text{⟨opdef_stmt⟩} \rightarrow \text{⟨opdef_list⟩} = \text{⟨const_num⟩} \\
| \text{⟨opdef_list⟩} = \text{⟨range⟩} \\
| \text{⟨opdef_list⟩} = \text{⟨op_const_list⟩} \\
| !\text{⟨opdef_list⟩}
\]

The operation code assignment within **opcode** block assigns one or more values to the **OpCode**s. This assignment grammar is shown in Grammar 5.2. All operation code assignments and exclusions in the **opcode** block are named as **opdef_stmts**. One or more **opdef_stmt** construct the **opdef_stmts**, and an **opdef_stmt** stands for one operation code assignment or exclusion.

Moreover, the last line on Grammar 5.2 shows how to exclude the **OpCode** which users do not want to use. All excluded **OpCodes** will be deleted from the Mase and never be assigned.

**Grammar 5.3** OpCode definition grammar

\[
\text{⟨opdef_list⟩} \rightarrow \{ \text{⟨opdef_list⟩} , \text{⟨opdef⟩} \} \\
\text{⟨opdef⟩} \rightarrow \text{⟨fu_id⟩} . \text{⟨clause_name⟩}
\]

The **opdef_list** in Grammar 5.3 declares which **OpCode** will be used in the assignment. The **opdef_list** is composed by multiple **opdef** elements. These elements defines what FU and which usage will be used in this **OpCode**.

More **OpCode** assignment examples and some notices are shown in Section 4.3.
5.3.3 Grammar for define OpCode and instruction size

The size of OpCode and instruction is the length of bits in OpCode value and instruction respectively. This size definition is placed in the FU block. Grammar 5.4 shows how to use the two keywords instruction_size and opcode_size to set the size of instruction and OpCode.

Grammar 5.4 OpCode and instruction size definition grammar

\[
\langle \text{instruction\_size} \rangle \rightarrow \text{instrucion\_size} \ (\text{const\_num})
\]

\[
\langle \text{opcode\_size} \rangle \rightarrow \text{opcode\_size} \ (\text{const\_num})
\]
Chapter 6

OpCode Assignment Generator

The OpAssignGen is implemented to complete three tasks:

- Instruction and OpCode size customization
- OpCode assignment
- OpCode exclusion

The first task is to allow users to adjust the instruction or operation code size. For example, setting the instruction size to 64 will get a 64bit length instruction sets at last.

The OpCode assignment is core business for my generator, which users can define OpCodes’ value according their own requirements.

The last task, OpCode exclusion, is to allow users to eliminate OpCodes from Masc. Normally, we can exclude unused OpCodes or any OpCodes. Excluded OpCodes will never appear in the instruction sets. This exclusion action will also decrease the number of final instruction in the processors.

6.1 Procedures of OpCode assignment

As mentioned in Chapter 4, The declaration of FU usage list defines the corresponding operations for every phase/pipeline in the operation block. OpCodes are combination of these operations and represent all of possible instructions can be executed by the processor.

OpCodes are assigned automatically by NoGap. If we want to customize certain OpCode values, the first important thing is let NoGap knows what OpCodes are going to be assigned and what their values are. Because we
haven’t such grammars to allow users assign OpCodes, the first step is to extend current grammars. The extended grammars have been explained in Chapter 5. At next step, we need to extend NoGap scanner and parser, then NoGap can understand the new assignment grammar and add new nodes to ASTs. The extension of scanner and parser are shown in Section 6.2. Next, a new generator should be implemented to process corresponding new assignment nodes in every PU’s AST and finally complete the assignments or report errors.

The process of operation code assignment can be split into three steps: get OpCode assignments, verify assignments and finish assignments.

Getting OpCode assignments from AST is relative simple, the new generator only needs to traverse the AST in all PUs to detect if there is OpCode assignment and fetch corresponding assignment statement nodes to get necessary data such as what the type of OpCode assignment is, which OpCodes are going to be assigned and what the values are. At this point, the new generator will also handle the semantic error due to incorrect OpCode assignments.

When the assignment data have been known, verifying these assignments is the next important step for the generator. The verification of assignment can also be split into two steps: verify the definitions of OpCode in assignments and check if assigned values are valid. Before the process of operation code assignments, all OpCodes in every Mast have been generated. A container at PU keeps all OpCode objects represented by the class OpCode. Then, the verification of OpCodes is just to search the container and check the existence of OpCodes. For the validity of assigning values, the generator should make sure these values are unique and unused. On the other hand, if the OpCode assignment is range assignment, the amount of available values will be check also. Actually, there are more verification procedures during OpCode assignments. For example, the generator needs to check if the OpCode has been assigned. The assignment processing flow and most of verification procedures will be explained in Section 6.5.

The new generator will finally assign values to corresponding OpCode objects mentioned above, if the assignment passes all checks. Then, other generators in NoGap continue to process PUs and combine OpCode and immediate data to an instruction at last.

6.2 Scanner and Parser Extension for OpAssignGen

Because of the new elements and grammars in NoGap$^{CL}$, the scanner and parser should be extended. Firstly, the keywords used in the new grammar
are added to the lex file, then these keywords can be recognized as tokens in the parser later. The three new keywords and their returned tokens have been shown in the Listing 6.1. The keyword instruction_size and opcode_size have been introduced in Section 5.3.3. The keyword opcode on line 3 marks the beginning of operation code assignment.

The next step is to configure the parser. Then the parser can recognize the new grammar and insert corresponding nodes into the AST. As mentioned in Section 3.3, Bison is used to construct the NoGap parser. Listing 6.2 shows a part of definitions for the new OpCode assignment grammar.

The insertNode function on line 5 will insert a node named OpAssignList into the AST. The information about node insertion and parser rule definition have been listed in Section 3.3. All statements, identifiers and keywords for the OpCode assignment will be inserted into AST as nodes. The more detailed list about the definition for the parser extension can be found in Appendix B. Table 6.1 shows all node types for the operation code assignment.

### 6.3 Representation of OpCode

The representation of OpCode is kept in the OperationInfo class. Before the OpAssignGen starts its process, all possible OpCode combinations have been saved in the data structure arch_data::v2::OperationCodes. As
Table 6.1. Vertex type

<table>
<thead>
<tr>
<th>Node Type</th>
<th>Explanation</th>
</tr>
</thead>
<tbody>
<tr>
<td>InstructionSize</td>
<td>The root node for instruction size definition</td>
</tr>
<tr>
<td>OpCodeSize</td>
<td>The root node for OpCode size definition</td>
</tr>
<tr>
<td>OpCode</td>
<td>The root node for OpCode assignment block</td>
</tr>
<tr>
<td>OpAssignStmts</td>
<td>The root node for all OpCode assignments and exclusions</td>
</tr>
<tr>
<td>OpAssign</td>
<td>The root node for one value OpCode assignment</td>
</tr>
<tr>
<td>OpAssignList</td>
<td>The root node for list OpCode assignment</td>
</tr>
<tr>
<td>OpAssignRange</td>
<td>The root node for range OpCode assignment</td>
</tr>
<tr>
<td>OpExclude</td>
<td>The root node for OpCode exclusion</td>
</tr>
<tr>
<td>OpCodeDec</td>
<td>The node for OpCode declaration</td>
</tr>
<tr>
<td>OpConstList</td>
<td>The root node for a list of constant number</td>
</tr>
</tbody>
</table>

mentioned in Section 6.1, the OperationCodes will be searched by my generator to find the matched OpCode objects.

From Figure 6.1, we know the vector container operation_codes_m in OperationCodes holds the operation code objects. And the operation code is represented with OpCode class. This class is composed by multiple OpCodeSlices. The slice looks like t1.%ADD which is represented with the FuUsage class. On the other hand, the numerical OpCode is represented with the class NumericOpCode.

The friend class AssignOpHandler is instantiated by my generator, which will be introduced in Section 6.6. This handler is responsible for processing OpCode in the OperationCodes.

6.4 Data Structure in OpAssignGen

As mentioned in Section 3.6, the data and tables containing assigned and excluded OpCode are kept by another class separated with the generator action part. A simple type, arch_data::AssignOpCodes, wrapped by Generated<DATA>\(^1\) is used to hold all data for OpAssignGen. Except the operation codes table, some methods are implemented in the AssignOpCodes class as the operation interface. Line 5 in Listing 6.3 shows the most important table, the operation code table, which keeps all OpCode customized by users. This table uses the bimap in boost library [12] as OpCode container. A bimap is a data structure that represents bidirec-

\(^1\)This wrapper has been explained in Section 3.6.
Figure 6.1. OpCode representation UML

Listing 6.3. AssignOpCodes Definition

```c++
1  class AssignOpCodes: public policy::OutStreamable
2  {
3    ...
4    typedef boost::bimaps::bimap<boost::bimaps::set_of<utils::BitNum>,
5                                   boost::bimaps::unordered_set_of<table_entry_t> > optable_t;
6    ...
7  };
```
tional relations between elements of two collections. The left collection, \textit{BitNum}, in the container represents a big int. The value of \textit{BitNum} can be printed in hexadecimal or decimal easily with fixed or unfixed length. The main reason I use \textit{bimap} is due to the bidirectional constraints between the \textit{OpCode} and its value. Both \textit{OpCode} and its value are key elements, which means they must be unique in the map. The mapping relation between the two mapped types are same. The mapping from value to \textit{OpCode} is injective and vice versa.

There are some important assignment rules I need to mention again. The \textit{OpCode} value can be overwritten multiple times, but only the last assigned value is valid and kept in the table. If the assigned value has been in the table, the generator would raise an error issue and terminate processing. Listing 6.4 shows some \textit{OpCode} overwriting examples. The overwriting on line 3 to line 7 are correct. However, the value 3 used in the assignment on line 13 has been assigned to the \textit{OpCode} \texttt{t1.%ADD,t2.%ADD,jump_cond.%F0}, which leads to an error issue finally.

\begin{verbatim}
1 ... 2 opcode { 3 {t1.%ADD,t2.%ADD,jump_cond.%F0} = 0; 4 {t1.%ADD,t2.%ADD,jump_cond.%F0} = 2; 5 // the OpCode:[t1.%ADD, t2.%ADD, jump_cond.%F0] finally has value 3 6 {t1.%ADD,t2.%ADD,jump_cond.%F0} = 3; 7 {t1.%SUB,t2.%OR,jump_cond.%F0} = 2; 8 */ 9 * Value 3 is not valid number. 10 * Overwriting OpCode is allowed, 11 * but OpCodes should have a unique value. 12 */ 13 {t1.%SUB,t2.%OR,jump_cond.%F0} = 3; 14 } 15 ... 16 }
\end{verbatim}

\section{OpAssignGen Process Flow}

The OpAssignGen is made up of two classes, \texttt{Generator::AssignOpCodes} and \texttt{Generator::AssignOpHandler}. The \texttt{AssignOpCodes} represents the operation code assignment generator. The \texttt{AssignOpHandler} is a help class which helps the generator to perform the \textit{OpCode} assignment and verification. In a normal input file, one \texttt{Mase} has one or more operation blocks.
To save the resource, the `AssignOpHandler` will only be initialized, when a valid operation code assignment is found in the `operation` block. After the handler finishes its work, it will be released immediately.

Figure 6.2 shows the generator processing flow. All generators registered in the `default_registration` will be initialized when `NoGap` starts. Then, in the flowchart, it assumes the operation code assignment generator has been initialized. As mentioned in Section 3.7, every generator will perform initializing as well as finalizing work after generator initialization and before generator release. The flowchart ignores these two stages, because there is nothing to do for the `OpAssignGen` in the initialization and finalization stage. When the generator starts and prepares to process PUs, it will firstly detect whether the data member `MaseOperation` in the PU has been initialized. The instance of type `MaseOperation` contains all possible operation codes combinations and operation information. The more detailed introduction about `MaseOperation` has been shown in Section 6.3.

If everything is fine, the `operator()` function in the generator will be invoked, which marks the processing stage starts. The first operation in this stage for the `OpAssignGen` is to calculate how many bits we need to represent a numerical `OpCode`. The calculation method has been shown in Section 4.3.1. Next, the generator checks whether there is a valid user customized `OpCode` length. If the user defined length is less than the default value, the generator will terminate immediately and report an error.

Next, the generator will try to search a symbol type `info::Operation` in the symbol table. If no `Operation` symbol is found, it means there is no `operation` and `opcode` block in the current PU, then the generator goes to END step directly. Assuming the generator finds the `Operation` symbol, it will get the corresponding `operation` vertex descriptors from the symbol table, and then save all `operation` vertex descriptors into a list for further processing. After that, the generator goes to check the generator options setting in the XML based control file. There are two options for the `OpAssignGen`: `verbose` and `autocomplete`. Option `verbose` asks the generator to print the info messages in all levels. When `autocomplete` turns on, the generator can handle incomplete `OpCode` assignments.

After checking these options, the generator will pick one `operation` vertex from the list and do further checks. The first check is to find whether the `operation` vertex has one or more sub vertices named `OpAssignStmts` and then check further if there is a sub vertex under the `OpAssignStmts`. These checks make sure that current PU has operation code assignment. Then the `AssignOpCodes` initializes an `AssignOpHandler` to process the operation code assignments one by one. For example, the handler needs
Figure 6.2. OpCode Assignment Flowchart
to process 3 OpCode assignments and one exclusion in List A.2. The detailed assignment validation and processing will show in Section 6.6. After processing the OpCode assignments in the current operation block, the generator will pick a new operation vertex and repeat the checks mentioned above. When all operation vertices have been handled, the generator will assign values to the remaining OpCodes which are not assigned by users.

Finally, the generator will print all user customized operation codes and their values if the option verbose has been set, and reset the generator for processing the next PU.

### 6.6 Structure of AssignOpHandler

Figure 6.3 shows the UML diagram of AssignOpCodes and AssignOpHandler. The most important attributes, methods and the relationship between these classes have been listed in the figure. As mentioned in the Section 6.5, after
some initialization works, the OpAssignGen will try to traverse the AST to find a vertex named operation. The member method find_operation in class AssignOpCodes is used to find the corresponding operation vertex. When we get the operation vertex and verify the operation vertex has sub vertices named OpAssignStmts, the AssignOpHandler will be initialized by the generator. The UML figure shows the associate relationship between the handler and generator class.

The class AssignOpHandler is the core for the OpAssignGen. It verifies the user defined OpCode as well as their assigned values, handles errors and warnings and saves the assigned and excluded OpCode to the corresponding containers. The constructor of class AssignOpHandler can be found in Listing C.2 at Appendix C.

In Figure 6.3, the two assign_opcode functions in AssignOpHandler are both used to assign the user customized OpCode value. The difference is the first version of assign_opcode only accepts one value and assigns one OpCode, while the second version accepts a list of values and also handles multiple OpCode. Actually, the process flow of this two version of functions are similar.

6.6.1 Processing Flow for AssignOpCodeHandler

```
Listing 6.5. Simple OpCode Assignment Example

1 if (data_path
2  
3  ...
4  operation(pipe) op3(dec_unit.type_b)
5  %OP3 {
6  
7  @P0;
8  @P1;
9  t1'%(SUB';
10  @P2;
11  t2'%(AND';
12  @P3;
13  
14  opcode
15  {  
16   {t1.%SUB,t2.%AND} = 20;
17  }
18 )
19  ...
20 }
```

Listing 6.5 shows a simple operation code assignment. The AST gener-
ated from this input is shown in Figure 6.4. I will use this simple example to illustrate how the operation code assignment is processed by the handler. To simplify the example, other nodes in the AST have been ignored. Before illustrating the processing flow in the handler, I hope to repeat the **OpCode** assignment flow. When users want to do **OpCode** assignment, they should designate what **OpCode** they want to assign. Next, they should give one or more values to corresponding **OpCodes**. In Section 6.1, we have known that two key points for **OpCode** assignment. They are **OpCode** and value verification respectively. The following check list shows what will be verified by the handler.

1. The customized **OpCodes** exist.
2. **OpCodes** are not assigned previously (if **OpCode** have been assigned, the generator will return a warning)
3. The values assigned to **OpCodes** are correct.
4. The assigned **OpCodes** value are unique.

To finish the **OpCode** assignment shown in Listing 6.5, the handler needs the following steps to verify and assign value.

1. check whether the value assigned to **OpCode** is valid. For example, in the Listing 6.5, the handler will search the opcode table to verify that value 20 is not in the table and less than the maximum number.

2. check the amount of sub operation for the **OpCode** is correct.

   The format of sub operation has been introduced in Section 6.3. Normally, **OpCode** is composed of multiple sub operation. And the number of sub operation in the **OpCode** is decided by the amount of operation units declared in advance under the phases block. We also call sub operation as **OpCode** slice.

   In Listing 6.5, there are only two units declared in the phase P1 and P2. So a valid **OpCode** declaration in the assignment should only contain two sub operations.

   The handler will scan the AST to get the number of sub operations used in the assignment statement. Figure 6.4 shows of there are two sub operations under the **OpCodeDec** nodes, which is \texttt{t1.\%SUB} and \texttt{t2.\%AND} respectively.

3. if the number of sub operation in the assignment statement is greater than the correct number, the **warning_handler** is invoked to terminate processing and report error.
Figure 6.4. A Simple AST generated from the Assignment in Listing 6.5
If the number is less than we expect, the generator will assume there is an incomplete assignment, and then the generator starts to generate all possible OpCode combinations.

4. In this step, the handler generates the character format OpCode based on the AST traverse result. This character format, table_entry_t, has been introduced in Section 6.4.

5. The handler checks the validation of OpCode in character format. If the handler doesn’t find any matching OpCode in the OperationCodes, an error is reported. If the corresponding OpCode in the OperationCodes is found, the reference to that entry will be passed to the handler.

6. check whether the OpCode has been assigned previously.
   
   If the OpCode has been assigned, raise the OVERWRITING warning

7. if the statement is a range or constant list assignment, the handler will check that the available amount of values in that range or list is no less than the number of OpCode in the assignment.

   For example, if we use the range [1:0] to assign 3 OpCode in a range OpCode assignment, this assignment will cause the RANGE_SMALL error.

8. check that the assigned value is not used previously, and raise an error if the value is not unique. The handler will check the used_op_code table in the arch_data::AssignOpCodes.

9. if every step above is OK, a class NumericOpCode will be instantiated with the assigned value in the corresponding OpCode object.

   In List 6.5, the handler will finally insert the OpCode, t1.%SUB,t2.%AND, and its value into the optable locating at arch_data::AssignOpCodes.

   Now, the OpCode assignment in the current operation block Op3 is finished, the handler will be released, and a new handler instance will be initialized by the generator if there are more operation blocks in the PU.

   For OpCode exclusion, it is similar with assignment. The same checks are used on the exclusion to make sure the OpCode trying to be excluded are valid. When everything is fine, the corresponding OpCode entries in the OperationCodes are removed. At the same time, the excluded opcode table in arch_data::AssignOpCodes will keep the deleted OpCode as a backup. The notable thing for exclusion is we cannot use auto-complete in OpCode exclusion, since deleting an OpCode from Mase is a dangerous behavior, an explicit definition is necessary.
6.7 Instruction and OpCode Size Definition

As I introduced previously, the instruction and OpCode size can be customized by the user. With the size definitions, the length of instruction and numerical OpCode can be customized by user. This customization is very important, especially for the OpCode assignment, because the length of numerical OpCode determines the greatest number of OpCode we may assigned. For example, if we set the OpCode size equals to 3, then the maximum numerical OpCode is 0x7 and the total amount of OpCodes should be less than or equal to 8. These two definitions are defined out of operation block. Users can only define these two sizes at one time. Overwriting or redefining them will result an error and terminate generator immediately.

The OpAssignGen processes the length assignment before it processing the OpCode assignment. The instruction and OpCode length are handled by two methods: get_and_assign_custom_instruction_size and get_and_assign_opcode_size in AssignOpCodes. The declaration of these two methods have been shown in Figure 6.3.

Before handling the OpCode size defined by users, the generator will invoke the function determine_opcode_size to calculate the default OpCode size. After that, the two methods in the generator will search the AST to get the two nodes named OpCodeSize and InstructionSize respectively. After analyzing these two nodes, the generator will make sure there is no redefinition and that OpCode length is shorter than the instruction length. What’s more, the operation code size should be greater than the default operation code size calculated previously. Finally, these two values will be passed to current PU and kept as PU attributes. The default OpCode size will be replaced by the customizing value, but the instruction size will be handled later by the instruction format generator.

6.8 OpAssignGen Registration

Listing 6.6. OpAssignGen Registration

```c
void default_registration(Register& reg) {
  ...
  reg.register_generator<generator::AssignOpCodes>
  ("gen_assign_opcodes");
  ...
}
```
If we want to use the OpAssignGen, it should be registered in \texttt{NoGap} firstly. The generator register has been introduced in Section 3.7. Listing 6.6 shows the generator registration for OpAssignGen. The function \texttt{register\_generator} on line 5 instantiates the generator, and then inserts the pointer of generator as well as the generator name into a map. The generator registration is finished at the beginning of \texttt{NoGap} processing flow. Users can utilize the generator name \texttt{gen\_assign\_opcodes} to configure the OpAssignGen in the control flow file.
Chapter 7

NoGap Improvements and suggestions

In this chapter, some improvements and suggestions will be given. I will focus on the improvements about software functionality, such as the interface, API and error handling.

7.1 NoGap Error Detection and Handling

The error I mention here is caused by user inputs. The user inputs can cause two types of error. They are syntax error and semantic error respectively. A syntax error is normally due to the spelling mistake or missing symbols. In NoGap, the syntax error is detected during the compile-time. Listing 7.1 shows an input with the syntax error. On line 3, the semicolon is missed. NoGap can detect the syntax error successfully.

Listing 7.1. Input with Syntax Error

```
1   image_test
2 {
3   input [31:0] dat_a_i
4   input [31:0] dat_b_i;
5   input [1:0] op_i;
6   output [31:0] res_o;
7 }
```

Listing 7.2 shows the prompting message returned by NoGap for the syntax error. The message on line 1 indicates there is a syntax error and its line number in the input file, but the line number is incorrect. What’s more,
the message on line 2 is not meaningful. An improvement for the NoGap front-end should be made to give users more friendly information about what and where the error is. This improvement can be implemented simply with certain Bison options. For instance, the macro YYERROR_VERBOSE in Bison can generate more meaningful message except the plain Syntax error message. After using this macro, the error message in list 7.1 will become Error: 'syntax error, unexpected IDENTIFIER, expecting SEMICOLON'.

Listing 7.2. Syntax Error Information

```
1 4: Syntax error
2 nogap: parser.y:106: int ncdparse(): Assertion 'false' failed.
3 Aborted
```

On the other hand, the semantic error detection is more complex than the syntax error detection. The semantic error may be detected by NoGap parser during the compile-time, or generators during their parsing PUs. In NoGap, the ValidateIdentifiers generator is specified to validate identifiers. According the predefined rules or actions, the validation generator search identifier nodes in ASTs and validate nodes with recordings in the symbol table. Currently, these validations are relative simple. Errors among different FU, and the instantiating a non-existing FU are not detectable. The semantic error detection for various statements is huge work, one possible method to simplify the implementation is to require the generators to detect errors during their performance. Since generators may developed by different developers, the error detection work can be distributed easily. Actually, the identifier validations in the operation code assignment are performed by the OpAssignGen.

Another possible improvement is the error handler CompileIssueHandler. Sometimes, the generator want to terminate its work immediately when there is a critical error. However, the generator can not terminate processing in real time, because all issues are handled until the generator finishing its work. A critical error issue may be added for the emergency termination. When the critical error is raised, the handler may throw an exception to terminate processing right away.

### 7.2 Better API and Interface for NoGap

I think a good platform must have a good API. NoGap has offered a serious of API to traverse ASTs, visit PUs and do some conversions, most of
containers in NoGap have exposed the iterators for read and write accesses. But APIs in NoGap are not categorized well. All help functions are put into the same namespace without hierarchies, some assessors are missed or hidden in the implementation. For developers, a clean and multi-functional APIs are expected.

The generator structure gives NoGap a flexible layout to extend functionality. Generators are implemented as static libraries and configured by the flow configuration file. This method also guarantees that unused generators will not be initialized, which saves the system resource, such as memory on the heap. But this layout may be not a good method for commercial development. The NoGap system have to recompiled when a new generator is added. In addition, installing a new generator and making its work is also not simple for normal users. I think the dynamical loading is good alternative for current structure. With the dynamical loading, generators may be complied to a single object file and loaded at run time. This mechanism guarantees the installation of generators is easy, the development of generator is independent and the loading of generators is controllable.

Furthermore, the interface of generator in NoGap should be extended also. Developers may easily implement a generator to extend NoGap functionality based on the simple interface. However, the generator interface is too simple to fulfill the further demands from developers. Developers have to implement their own data structures or methods to improve the generator. Normally, these behaviors will destroy the software encapsulation. For instance, current interface does not define how to share data among different generators. If a generator want to pass data to another generator, developers have to hack other generator and the PU structures by themselves.
Chapter 8

Results

To introduce the new NoGap generator OpAssignGen for operation code assignment, we firstly explained NoGap structure (Chapter 3) and the purpose of operation code assignment (Section 5.1). Secondly, we illustrated new grammars in Section 5.3 for operation code assignment and some modifications in NoGap scanner and parser (Chapter 6). What’s more, we also explained the idea of operation code assignment syntax format and what its advantage is in Section 4.1. The implementation of OpAssignGen is introduced in Chapter 6. To assistant users to use new OpAssignGen, a simple tutorial presented in Section 4.3. After that we will show some synthesizing results which prove the customized operation code assignment by OpAssignGen has only tiny effect for the logic HW design in next Section 8.1. Finally, we will mention some NoGap drawbacks and possible improvements in Chapter 7.
8.1 Results of Synthesizing

After the operation code assignment for the input in Listing 4.1, we can finally get some hardware implementation with the Verilog generator. To verify the design, I synthesized the hardware implementation and decoder implementation. Table 8.1 shows the two results which are implementation with operation code customization and without customization respectively. The target device for synthesizing is Field Programmable Gate Array (FPGA) xc2v4000. We can find that the synthesized result for the decoder implementation with customization has a tiny increase, compared with the result without operation code customization. This change shows the operation code customization almost has no effect for the logic utilize in the decoder implementation.

<table>
<thead>
<tr>
<th>Name</th>
<th>Number of 4 input LUTs</th>
</tr>
</thead>
<tbody>
<tr>
<td>HW Implementation (customized)</td>
<td>32 out of 46,080</td>
</tr>
<tr>
<td>Decoder Implementation (customized)</td>
<td>29 out of 46,080</td>
</tr>
<tr>
<td>HW Implementation</td>
<td>32 out of 46,080</td>
</tr>
<tr>
<td>Decoder Implementation</td>
<td>28 out of 46,080</td>
</tr>
</tbody>
</table>

Table 8.1. Synthesizing Results
Chapter 9

Conclusions

Chapter 8 shows a successful result that the amount of used LUTs has a reasonable value compared with the result without operation code customization. Based on this result, I can get the conclusion that my OpAssignGen is implemented successfully. Due to the time limitation, I cannot simulate the hardware implementation and verify the result further. What’s more, the compatibility of MIPS instruction sets in my hardware implementation is not verified also, since there is no possible simulator solution now. Anyhow, the operation code customization is a good extension for Novel Generator of Accelerators and Processors (NoGap). It gives users more possibilities to create a novel Hardware (HW) design. There is no doubt current customization capability cannot fulfill the designers’ requirements. Besides the operation code customization, NoGap needs a more flexible instruction format. Designers can not only adjust the size of instruction but also alter the structure of instruction. For example, designer can add or remove bit fields in an instruction. NoGap has shown a good future for hardware design. Although this tool is still not maturity, it has a good potential to be improved.
Chapter 10

Future Work

Currently, the OpAssignGen lacks an important functionality: changing the position of OpCode in the instruction. The instruction format in NoGap is not adjustable now. This format has been explained in Section 4.1. Users are not allowed to change the position of OpCode. This limitation effects the flexibility of NoGap and weaken compatibility. So, I will try to implement this functionality in the future.

The initial idea is using the range assignment to display the position during the OpCode size customization. In Listing 10.1, I firstly defines the

Listing 10.1. Define the Position of OpCode in the Instructions

```c
1  // the number used in the range is decimal number.
2  instruction_size 32;
3  opcode_size [31:28];
```

instruction size, so the length of instructions is 32 bit. Then, an operation code size definition with a range is used on line 3 The range [31:28] in this OpCode size definition has different meaning. This size definition finish two things at the same time. It defines the OpCode is 4 bit length, and tell NoGap that OpCode should placed at bit 31 to bit 28, the beginning of the instruction. This change of grammar allows users to set the position of OpCode. The range upper limit and lower limit defines the beginning bit position and end bit position respectively, while the size of range defines the OpCode length.

Because instruction generation are not processed by OpAssignGen, certain generators need to be modified to accept new OpCode position option and generate corresponding instructions.
Appendix A

Implementaion for Part of MIPS Instructions

Listing A.2 shows a simple example which implements certain MIPS instructions.  

Listing A.1. Implementation of ADD unit in MIPS

```c
fu alu
{
  input [31:0] dat_a_i;
  input [31:0] dat_b_i;

  output [32:0] res_o;

  input[2:0] op_i;

  input foo_or_bar_i;
  output foo_bar_o;

  signal [32:0] tmp_res;

  comb
  {
    switch(op_i)
    {
      0: %ADD.default {
        tmp_res = dat_a_i + dat_b_i;
      }
      1: %SUB {
        tmp_res = dat_a_i - dat_b_i;
      }
      2: %AND {
        tmp_res = dat_a_i & dat_b_i;
      }
    }
  }
}
```

1This implementation is mainly written by Dr Karlström.
30  
31 3: \textbf{OR} \{  
32  \text{tmp}_\text{res} = \text{dat}_a_i ~|~ \text{dat}_b_i;  
33  \}  
34 4: \textbf{XOR} \{  
35  \text{tmp}_\text{res} = \text{dat}_a_i ~^\wedge~ \text{dat}_b_i;  
36  \}  
37 5: \textbf{NOR} \{  
38  \text{tmp}_\text{res} = !(\text{dat}_a_i ~|~ \text{dat}_b_i);  
39  \}  
40  \}  
41  \}  
42  \}  
43  \}  
44  \textbf{comb}  
45  \{  
46  \text{res}_o = \text{tmp}_\text{res}[31:0];  
47  \}  
48  \}  
49  \}  
50  \textbf{fu} \text{ detect\_overflow}  
51  \{  
52  \text{input} \text{ signal\_overflow}_i;  
53  \text{input }[32:0] \text{ dat}_i;  
54  \text{output }[31:0] \text{ dat}_o;  
55  \text{output} \text{ overflow}_o;  
56  \}  
57  \textbf{comb}  
58  \{  
59  \textbf{switch}(\text{signal\_overflow}_i)  
60  \{  
61  0: \textbf{DETECT\_OVERFLOW}  
62  \{  
63  \text{overflow}_o = \text{dat}_i[32];  
64  \}  
65  1: \textbf{NO\_OVERFLOW}  
66  \{  
67  \text{overflow}_o = 0;  
68  \}  
69  \}  
70  \text{dat}_o = \text{dat}_i[31:0];  
71  \}  
72  \}  
73  \}  
74  \textbf{fu} \text{ register\_file}  
75  \{  
76  \text{input }[4:0] \text{ addr\_a}_i;  
77  \text{output }[31:0] \text{ dat\_a}_o;  
78  \}  
79  \text{input }[4:0] \text{ addr\_b}_i;  
80  \text{output }[31:0] \text{ dat\_b}_o;  
81  \text{input }[4:0] \text{ addr\_wr}_i;  
82  \text{input }[31:0] \text{ dat}_i;  
83  \text{input} \text{ wr\_en}_i;  
84  \text{signal auto}("\text{dat}_i"):[31:0] \text{regs};
cycle
{
    switch (wr_en_i)
    {
        0: %NOP{}
        1: %WRITE
        {
            regs[addr_wr_i] = dat_i;
        }
    }
}

comb
{
    if (addr_a_i == 0)
    {
        dat_a_o = 0;
    }
    else
    {
        dat_a_o = regs[addr_a_i];
    }
    if (addr_b_i == 0)
    {
        dat_b_o = 0;
    }
    else
    {
        dat_b_o = regs[addr_b_i];
    }
}

fu: template<INSTRUCTION> mips_decoder
{
    input auto("#") INSTRUCTION;
    output [4:0] rf_wr_addr_o;
    output [4:0] rf_addr_a_o;
    output [15:0] imm_o;
    immediate [4:0] rf_wr_addr;
    immediate [4:0] rf_addr_a;
    immediate [4:0] rf_addr_b;
    immediate [15:0] imm_data;

    instruction r_type
    {
        source{
            rf_addr_a_o = rf_addr_a;
        }
        destination{
            rf_wr_addr_o = rf_wr_addr;
        }
    }
instruction i_type
  {
    source{
      rf_addr_a_o = rf_addr_a;
      imm_o = imm_data;
    }
    destination
    {
      rf_wr_addr_o = rf_wr_addr;
    }
  }

instruction type_nop
  {}

fu sign_extend
  {
    input op_i;
    input [15:0] dat_i;
    output [31:0] dat_o;
    comb
    {
      switch(op_i)
      {
        0: %NOP {
          dat_o = &{16<2>0,dat_i}&;
        }
        1: %EXTEND{
          dat_o = &{R{16,d[15]}R,dat_i}&;
        }
      }
    }
  }

fu data_path
  {
    input auto(internal) instr_i;
    output [31:0] dat_o;
    phase p_ID;
    phase p_EX;
    phase p_MEM;
    phase p_WB;
    stage ff(){cycle{ffo = ffi;}}
    pipeline pipe
    {
      p_ID -> ff -> p_EX -> ff -> p_MEM -> ff -> p_WB;
    }
    fu::alu(%ADD) alu;
    fu::mips_decoder<instr_i>() dec_unit;
    fu::register_file(%NOP) rf;
    fu::sign_extend(%NOP) se;
Listing A.2 shows another simple example for Opcode assignment.
Listing A.2. operation code assignment examples

```plaintext
operation (pipe) ex_op (dec.ex_op)
{
    ...
    // signal assignments are ignored.
    @P1;
    t1'%.ADD,.SUB';
    ...
    @P2;
    t2'%.ADD';
    ...
    @P3;
    jump_cond'%.F0';
    ...
    opcode {
        // A typical OpCode specification
        {t1%.ADD,t2%.ADD,jump_cond%.F0} = 1;
        // Range Assignment with incomplete OpCode definition
        {t1.any,jump_cond%.F0} = [4:2];
        // OpCode exclusion
        !{t1%.SUB,t2%.ADD,jump_cond%.F0};
    }
    ...
}
```
Appendix B

OpCode Assignment Definition for the Parser

Listing B.1 shows a part of definitions for the new OpCode assignment grammar.

Listing B.1. OpCode Assignment Definition for the Parser

```plaintext
1 opdef_stmt: opdef_simple_stmt '=' const_num ';'
2 {
3     $$ = insertNode(parse_data.ast(),info::OpAssign(),
4           info::EdgeOrder(),$1,$3,@$.first_line);
5 }
6 | opdef_simple_stmt '=' op_const_stmt ';'
7 {
8     $$ = insertNode(parse_data.ast(),info::OpAssignList(),
9           info::EdgeOrder(),$1,$3,@$.first_line);
10 }
11 | opdef_simple_stmt '=' range ';'
12 {
13     $$ = insertNode(parse_data.ast(),info::OpAssignRange(),
14           info::EdgeOrder(),$1,$3,@$.first_line);
15 }
16 | '!' opdef_simple_stmt ';;'
17 {
18     $$ = insertNode(parse_data.ast(),info::OpExclude(),
19           info::EdgeOrder(),$2,$$.first_line);
20 }
```

The `opdef_simple_stmt` in Listing B.1 is OpCode definition. Line 3 shows that a node named OpAssign is inserted into the AST, when a constant value is assigned to an OpCode. The `opdef_simple_stmt` in the definition is a declaration which announces what OpCode is assigned for. The operation code range and list assignment are shown on line 6 to line 11.
Line 16 shows the definition of `OpCode` exclusion. The function `insertNode` in every action block is used to insert one or more nodes into the AST. These nodes correspond to the assignment type, keyword, identifier and `OpCode` definition.
Appendix C

The Implementation of OpAssignGen

The Listing C.1 shows the structure of class AssignOpCodes. The table_entry_t,

```
1 class AssignOpCodes: public policy::OutStreamable
2 {
3 ...
4 public:
5     typedef std::map<std::string, std::string> table_entry_t;
6     typedef boost::bimaps::bimap<boost::bimaps::set_of<utils::BitNum>,
7         boost::bimaps::unordered_set_of<table_entry_t> > optable_t;
8     typedef std::map<table_entry_t, v2::OpCode> exclude_table_t;
9     typedef std::set<utils::BitNum> used_op_codes_t;
10 ...
11 public:
12     enum Status {OK=0, EXCLUDED, OVERWRITE, OVERWRITE_FAILURE,
13         WRONG_NUM, ERR_FU, WRONG_FU_UNIT, NO_RANGE,
14         REDEFINITION, MULTI_DEF, INCOMPLETE_DEF,
15         NONE_INCOMPLETE, RANGE_ERR,
16         RANGE_OVERLAPPING, UNITS_GEN_ERR,
17         RANGE_LARGE, RANGE_SMALL, ERASED};
18 public:
19     bool excluded(const table_entry_t& item) const;
20     enum Status exclude_item(const exclude_table_t::value_type& item);
21     enum Status insert_item(const optable_t::value_type& item);
22     bool insert_used_num(const utils::BitNum& num);
23 ...
24 }
```

on line 5, represents a character format OpCode. All OpCode definitions for assignments in the ASTs are converted to the table_entry_t. Then, the OpAssignGen uses these representations to get the corresponding OpCode
objects in the `OperationCodes`.

Line 8 in Listing C.1 shows the structure of table containing all excluded `OpCodes`. The type of mapped value, class `OpCode`, is a standard representation of `OpCode`. This class keeps both numerical and character operation codes.

If the operation in `AssignOpCodes` succeeds, it returns a zero value. When an operation fails, the value in the enumeration `Status` on line 12 is returned to indicate the cause of the failure.

The set, `used_op_t`, on line 9 contains all used `OpCode` values. It does not only include the values assigned by users, but also the default values assigned automatically by the generator. Furthermore, if a range of values are assigned to some `OpCodes`, all values in that range are kept by the set, even not every value in that range is used. The main purpose for this set is to control the value space, when users want to limit the possible values used by `OpCodes`.

Listing C.2 shows the constructor of the handler. All necessary data, such as the list of `operation` vertices, the reference to the `OperationInfo` and the current PU are passed to the handler from `AssignOpCodes`.

**Listing C.2. AssignOpHandler Constructor**

```cpp
AssignOpHandler(const parser::ast::vdesc_t& operation,
                 arch_data::v2::OperationInfo& operation_info,
                 parser::ParseUnit& pu,
                 bool verbose);
```
Bibliography


