Investigation of ΝοGap - SIMD Datapath Implementation

Examensarbete utfört i Reglerteknik vid Tekniska högskolan vid Linköpings universitet av Chun-Jung Chan

LiTH-ISY-EX--11/4454--SE

Linköping 2011
Investigation of NoGap - SIMD Datapath Implementation

Examensarbete utfört i Reglerteknik vid Tekniska högskolan i Linköping av

Chun-Jung Chan

LiTH-ISY-EX--11/4454--SE

Handledare: Per Karlström
            ISY, Linköpings universitet

Examinator: Andreas Ehliar
            ISY, Linköpings universitet

Linköping, 18 November, 2011
Investigation of NoGap - SIMD Datapath Implementation

Nowadays, many ASIP systems with high computational capabilities are designed in order to fulfill the increasing demands of technical applications. However, the design of ASIP system usually takes many man hours. Therefore, a number of EDA tools are developed to ease the design effort, but they limit the design freedom due to their predefined design templates. Consequently, designers are forced to use lower level HDLs which offer high design flexibility but require substantial design hours. A novel design automation tool called NoGap was proposed to balance such issues. The NoGap system, which is especially used in ASIPs and accelerator design, effectively provides high design flexibility and saves design effort for designers.

The efficiency and design ability of NoGap were investigated in this thesis work. NoGap was used to implement an eight-way SIMD datapath of an ASIP called Sleipnir, which was devised by the Division of Computer Engineering at Linköping University. For contrast, the manually crafted HDL implementation of the Sleipnir was taken. The critical path implementations, done by both design approaches, were synthesized to the Altera Stratix IV FPGA. The synthesize results showed that the NoGap design although used 1.358 times as many hardware units as the original HDL design. Their timing performance is comparable (HDL/NoGap-60.042/58.156Mhz).

In this thesis, based on the design experience of SIMD datapath, valuable aspects were suggested to benefit the future users who will use NoGap to implement SIMD structures. In addition, the hidden bugs and insufficient features of NoGap were discovered, and the referable suggestions were provided in order to help the developers to improve the NoGap system.

Keywords ASIP, NoGap, ePUMA, Sleipnir, SIMD
Abstract

Nowadays, many ASIP systems with high computational capabilities are designed in order to fulfill the increasing demands of technical applications. However, the design of ASIP system usually takes many man hours. Therefore, a number of EDA tools are developed to ease the design effort, but they limit the design freedom due to their predefined design templates. Consequently, designers are forced to use lower level HDLs which offer high design flexibility but require substantial design hours. A novel design automation tool called NoGap was proposed to balance such issues. The NoGap system, which is especially used in ASIPs and accelerator design, effectively provides high design flexibility and saves design effort for designers.

The efficiency and design ability of NoGap were investigated in this thesis work. NoGap was used to implement an eight-way SIMD datapath of an ASIP called Sleipnir, which was devised by the Division of Computer Engineering at Linköping University. For contrast, the manually crafted HDL implementation of the Sleipnir was taken. The critical path implementations, done by both design approaches, were synthesized to the Altera Stratix IV FPGA. The synthesize results showed that the NoGap design although used 1.358 times as many hardware units as the original HDL design. Their timing performance is comparable (HDL/NoGap-60.042/58.156Mhz).

In this thesis, based on the design experience of SIMD datapath, valuable aspects were suggested to benefit the future users who will use NoGap to implement SIMD structures. In addition, the hidden bugs and insufficient features of NoGap were discovered, and the referable suggestions were provided in order to help the developers to improve the NoGap system.
Acknowledgments

I never thought that one day I could have the opportunity to study in Sweden such a nice, beautiful, friendly country. It has been two years since I studied in Linköping University. This thesis consists of the results of my hard working and unforgettable memories at the university. It is my pleasure to thank those who made all my dreams come true.

First of all, I want to give my sincere appreciation to Linköping University for giving me a perfect education environment. Also, I would like thank to all the professors and staff who ever given me helps during my studies.

Secondly, I would like to give my deepest gratitude to professor Dake Liu, who is a passionate professor, provided me an excellent chance to work with the NoGap research team. The same gratitude also gives to my supervisor Per Karlström. Per supervised and supported me in this thesis work. He always patiently discussed any issues with me. In addition, I want to express my appreciation to Wenbiao Zhou who often gave me suggestions and encouragement. The same appreciation also goes to Andréas Karlsson. He provided me many design skills as well as sufficient knowledge of designing DSP system.

Finally, to my dear family and friends, thank you for all your supporting, consideration and company. This thesis not only belongs to me but also to every person who ever influenced me in my life.
## Contents

1 Introduction .......................................................... 3
   1.1 Background ....................................................... 3
   1.2 Selected System ................................................ 4
   1.3 Purposes and Goals ............................................. 4
   1.4 Chapter Overview ............................................... 5

2 NoGap Introduction .................................................. 7
   2.1 System Composition ............................................ 7
      2.1.1 Facet ....................................................... 7
      2.1.2 $\text{NoGap}^{\text{CD}}$ ................................... 8
      2.1.3 Spawners ................................................... 9
   2.2 Introduction to $\text{NoGap}^{\text{CL}}$ ......................... 9
      2.2.1 Main Advantages of $\text{NoGap}^{\text{CL}}$ .............. 10
      2.2.2 FU Overview ............................................... 10
      2.2.3 Mage FU Descriptions ................................... 10
      2.2.4 Mase Descriptions ....................................... 12
      2.2.5 Decoder Template & Declaration of Instruction Formats . 16
      2.2.6 More Features ............................................... 17

3 Introduction to SIMD Sleipnir Processor .......................... 19
   3.1 Sleipnir Overview .............................................. 19
   3.2 Vector Formats .................................................. 20
   3.3 8-Way SIMD Datapath of Sleipnir .............................. 22
   3.4 Sleipnir Pipeline ............................................... 23

4 Preview Of My Work .................................................. 27
   4.1 Chapter 5 ......................................................... 27
   4.2 Chapter 6 ......................................................... 27
   4.3 Chapter 7 ......................................................... 28
Contents

8.1.3  NoGap\textsuperscript{CL} - Unsigned and Signed Value Comparison 67
8.1.4  NoGap\textsuperscript{CL} - Set Parameters .............................. 68
8.1.5  NoGap\textsuperscript{CL} - Instantiate Mase in Mase .................. 70
8.2  NoGap - Issues of Generators ........................................ 72
  8.2.1  Issues of Graphs Size ............................................. 74
  8.2.2  Issues of Speed ................................................... 76
  8.2.3  Issues of Reported Error Messages ............................. 78

9  Conclusions .............................................................. 81
  9.1  Accomplishment ..................................................... 81

10  Future Work .......................................................... 83
  10.1  Advanced Implementation of Sleipnir Datapath ................. 83
  10.2  Advanced Implementation of NoGap .............................. 84

A  Mage FU - Single Structure ........................................... 85
B  Mage FU - Multiple Structure .......................................... 87
C  Mase FU - Single Structure ............................................ 91
D  Mase FU - Multiple Structure .......................................... 95

Bibliography .............................................................. 99
List Of Abbreviations
<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AGU</td>
<td>Address Generation Unit</td>
</tr>
<tr>
<td>API</td>
<td>Application Programmer Interface</td>
</tr>
<tr>
<td>AR</td>
<td>Address Register</td>
</tr>
<tr>
<td>ASIP</td>
<td>Application Specific Instruction-set Processor</td>
</tr>
<tr>
<td>ASIC</td>
<td>Application Specific Integrated Circuit</td>
</tr>
<tr>
<td>AST</td>
<td>Abstract Syntax Tree</td>
</tr>
<tr>
<td>CAR</td>
<td>Constant Address Register</td>
</tr>
<tr>
<td>Castle</td>
<td>Control Architecture STructure Language</td>
</tr>
<tr>
<td>CM</td>
<td>Constant Memory</td>
</tr>
<tr>
<td>DCT</td>
<td>Discrete Cosine Transform</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processor</td>
</tr>
<tr>
<td>EDA</td>
<td>Electronic Design Automation</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FU</td>
<td>Functional Unit</td>
</tr>
<tr>
<td>ISA</td>
<td>Instruction Set Architecture</td>
</tr>
<tr>
<td>LSB</td>
<td>Least Significant Bit</td>
</tr>
<tr>
<td>LUT</td>
<td>Look-Up Table</td>
</tr>
<tr>
<td>LVMs</td>
<td>Local Vector Memories</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply And Accumulate</td>
</tr>
<tr>
<td>Mage</td>
<td>Micro Architecture Generation Essentials</td>
</tr>
<tr>
<td>Mase</td>
<td>Micro Architecture Structure Expression</td>
</tr>
<tr>
<td>MSB</td>
<td>Most Significant Bit</td>
</tr>
<tr>
<td>NoGap</td>
<td>Novel Generator of Accelerators and Processors</td>
</tr>
<tr>
<td>NoGap&lt;sub&gt;CD&lt;/sub&gt;</td>
<td>NoGap Common Description</td>
</tr>
<tr>
<td>NoGap&lt;sub&gt;CL&lt;/sub&gt;</td>
<td>NoGap Common Language</td>
</tr>
<tr>
<td>PC</td>
<td>Program Counter</td>
</tr>
<tr>
<td>PC-FSM</td>
<td>Program Counter-Finite State Machine</td>
</tr>
<tr>
<td>PM</td>
<td>Program Memory</td>
</tr>
<tr>
<td>PPU</td>
<td>Physical Processing Unit</td>
</tr>
<tr>
<td>PU</td>
<td>Parse Unit</td>
</tr>
<tr>
<td>RISC</td>
<td>Reduced Instruction Set Computing</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Language</td>
</tr>
<tr>
<td>SIMD</td>
<td>Single Instruction, Multiple Data</td>
</tr>
<tr>
<td>SISD</td>
<td>Single Instruction, Single Data</td>
</tr>
<tr>
<td>SRF</td>
<td>Special Register File</td>
</tr>
<tr>
<td>TTM</td>
<td>Time To Market</td>
</tr>
<tr>
<td>VACR</td>
<td>Vector Accumulate Register</td>
</tr>
<tr>
<td>VFLAG</td>
<td>Vector Flag Register</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VLIW</td>
<td>Very Large Instruction Word</td>
</tr>
<tr>
<td>VRF</td>
<td>Vector Register File</td>
</tr>
</tbody>
</table>

**Table 1. Abbreviations**
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.1</td>
<td>NoGap System Architecture [16]</td>
<td>8</td>
</tr>
<tr>
<td>2.2</td>
<td>NoGap Design Flow [16]</td>
<td>9</td>
</tr>
<tr>
<td>2.3</td>
<td>NoGap vs. HDL</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>ePUMA Overview [5]</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>Sleipnir Processor Overview</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>Data Vector Formats</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>Sleipnir Datapath Stages</td>
<td>24</td>
</tr>
<tr>
<td>4.1</td>
<td>Project Working Flow</td>
<td>28</td>
</tr>
<tr>
<td>5.1</td>
<td>Sleipnir Datapath Architecture [6]</td>
<td>31</td>
</tr>
<tr>
<td>5.2</td>
<td>Short Quick Instruction [6]</td>
<td>33</td>
</tr>
<tr>
<td>5.3</td>
<td>Short Bypass &amp; Long Instruction [6]</td>
<td>35</td>
</tr>
<tr>
<td>5.4</td>
<td>Datapath Mage FU - MUL [6]</td>
<td>38</td>
</tr>
<tr>
<td>5.5</td>
<td>Datapath - ADDER_TREE Mage FUs [6]</td>
<td>39</td>
</tr>
<tr>
<td>5.6</td>
<td>Datapath Mage FU - Datswitch [6]</td>
<td>40</td>
</tr>
<tr>
<td>6.1</td>
<td>Single Lane of SIMD Datapath [6]</td>
<td>47</td>
</tr>
<tr>
<td>7.1</td>
<td>Single Mage FU Simulation</td>
<td>54</td>
</tr>
<tr>
<td>7.2</td>
<td>Overall SIMD Datapath Simulation</td>
<td>55</td>
</tr>
<tr>
<td>7.3</td>
<td>Synthesized Big ADDER_TREE Stage [6]</td>
<td>56</td>
</tr>
<tr>
<td>7.4</td>
<td>Equalization of ADDER_TREE [6]</td>
<td>58</td>
</tr>
<tr>
<td>7.5</td>
<td>Different Adder Structure</td>
<td>63</td>
</tr>
<tr>
<td>7.6</td>
<td>Subword Operation In NoGap</td>
<td>64</td>
</tr>
<tr>
<td>8.1</td>
<td>Long Arithmetic &amp; Logic Instruction Data flows [6]</td>
<td>73</td>
</tr>
<tr>
<td>8.2</td>
<td>Hierarchical Graph Representation</td>
<td>75</td>
</tr>
<tr>
<td>8.3</td>
<td>Partitioned Design Generation</td>
<td>77</td>
</tr>
</tbody>
</table>
## List of Tables

1. Abbreviations .......................................................... ii
2. Instruction stages of datapath ........................................ 32
3. Small excerpt of Sleipnir short type instructions ............. 34
4. Partial excerpt of Sleipnir long type instructions ............. 36
5. Data size of each ADDER_TREE Mage ............................... 37
6. Datswitch input control signals ....................................... 40
7. Other Mage FU inside the datapath .................................. 41
8. Resources usage report - HDL ADDER_TREE ..................... 59
9. Resources usage report - NoGap ADDER_TREE ................... 60
10. FPGA - Timing performance ........................................... 60
11. ASIC - Area ............................................................. 60
12. Arithmetic comparison format ....................................... 68
13. Need Recompiled and regenerated parts ........................... 78
Chapter 1

Introduction

1.1 Background

Application Specific Instruction-set Processor (ASIP), which has features of high performance, low power consumption, and low silicon cost, is getting more preferable to current systems. Comparing with Application Specific Integrated Circuit (ASIC), ASIP strongly offers programmable flexibility. However, ASIP though gives powerful solutions to the system designers, requires great human design effort. Designers still have to find solutions to lower their design effort and reduce Time To Market (TTM) issues. If readers have interests about ASIP design, more information can be found in [17].

To ease the design effort of ASIP, many Electronic Design Automation (EDA) tools are developed. When designing ASIP, designers can use e.g. LISA [19], EXPRESSION [10], nML [9], SimpleScalar [1], MIMOLA [18], ASIP Meister [20]. All the existing EDA tools help, but not completely. Since some of them only provide sub-functions, most of them, due to their predefined design templates, limit the designers. When using such tools, designers neither receive complete supports nor have full design flexibility. Therefore, the constructed designs usually differ from designers’ original ideas. In order to obtain full design flexibility, designers have to use lower level Hardware Description Languages (HDLs) e.g. VHSIC Hardware Description Language (VHDL), Verilog, and SystemVerilog. However, the use of HDLs takes time and substantial design effort.

To balance such issues, Novel Generator of Accelerators and Processors (NoGap) system, which is especially used in ASIPs and accelerator design, was developed. The novel NoGap system was devised by the Division
of Computer Engineering at Linköping university. NoGap provides high design flexibility and saves design effort for designers. Also, NoGap has no predefined design template and can help designers to handle detailed and error prone tasks during the design time. After designers specifying the complete instruction descriptions of their expected system, NoGap is able to generate the target structure for the designers. By using NoGap, without many working hours, designers can truly implement their novel system.

1.2 Selected System

ePUMA platform is the current research at the Division of Computer Engineering at Linköping University. It has features of high computational capabilities and low power consumption. The main target of ePUMA is to create a highly parallel Digital Signal Processor (DSP) platform for real-time embedded applications. The ePUMA platform is an attempt of Master-multi-SIMD DSP processor which combines one Reduced Instruction Set Computing (RISC) Master processor with eight Single Instruction, Multiple Data (SIMD) Sleipnir co-processor in a single chip.

Previously, NoGap had been used to implement certain systems e.g. RISC PIONEER and SENIOR processors [21], floating point datapath [14]. All previous implementation showed many advantages by using NoGap. However, NoGap has not yet been used to implement any advanced SIMD architectures. Therefore, in this thesis work, the SIMD datapath of the Sleipnir co-processors as a experiment was chosen to be implemented by NoGap.

1.3 Purposes and Goals

There were two main purposes of this thesis work. One was to evaluate the efficiency of the SIMD implementation done by NoGap. Another was to discover the potential insufficiency of the NoGap system.

The goals of the work are listed below.

1. Investigated the feasibility of SIMD architectures implementation done by NoGap.
2. Compared the NoGap SIMD implementation with the original HDL design.
3. Gave useful suggestions and design techniques to future users.

4. Discovered the insufficient features and potential bugs of the NoGap system.

5. Provided possible solutions and user feedback to the NoGap developers.

1.4 Chapter Overview

- **Chapter 1** provides the general information of the thesis.
- **Chapter 2** gives the brief introduction of NoGap and the syntax descriptions of NoGap Common Language ($NoGap^{CL}$).
- **Chapter 3** introduces the SIMD Sleipnir processor.
- **Chapter 4** outlines the main implementation flow of the work.
- **Chapter 5** introduces the detailed NoGap implementation of SIMD datapath.
- **Chapter 6** offers useful suggestions of SIMD implementation in NoGap.
- **Chapter 7** reports the SIMD datapath simulation and synthesize results. Also discusses the differences between NoGap and HDL design.
- **Chapter 8** lists the design limitations, suggestions, and improvements of the NoGap system.
- **Chapter 9** summarizes the thesis’s achievements and conclusions.
- **Chapter 10** schedules the future advanced work.
Chapter 2

NoGap Introduction

This chapter aims to give brief introductions of the NoGap system. More detailed information about NoGap could be found in [12, 13, 15, 16]. The general NoGap system can be divided into three main components that are facets, NoGap Common Description (NoGap\textsuperscript{CD}), and spawners. In the following sections, how each of such three components works and where they stand inside the system will be introduced.

2.1 System Composition

The overall NoGap system is depicted in Figure 2.1 (copied from [16]), which clearly shows the operating flow of the NoGap system.

2.1.1 Facet

When using NoGap, designers have to use NoGap\textsuperscript{CL}, which is the default facet of NoGap, constructs the design.

Suppose designers are going to implement a simple processor. A processor might have many functional units. Let’s take the most basic function unit “Adder” for example, an “Adder” unit is used to perform addition or other similar arithmetic operations. In NoGap system, such small function unit is called Micro Architecture Generation Essentials (Mage) Functional Unit (FU). After designers designing all the Mage FUs, to construct this processor, designers need to find a way to make connections between all the Mage FUs. In NoGap system, there is a particular FU called Micro Architecture Structure Expression (Mase) FU, which is the top FU contains all the information about the spatial and temporal relation between each
From the previous example, designers basically have finished the structure construction of the simple processor. In fact, one more thing has to be done, which is to construct the instruction decoder of this processor. The decoder is used to control the behaviors of every Mage FU, and the operation flow of every instruction inside this processor. In NoGap system, one thing is called Control Architecture STructure Language (Castle) which contains all the information needed to construct a decoder.

To sum up, all such Mage, Mase FUs, and Castle are constructed by NoGap default facet. Since NoGap$^{CL}$ is the default facet, designers have to use it to implement such three units. The detailed introductions of NoGap$^{CL}$ and its syntax examples will be further introduced in Section 2.2.

2.1.2 NoGap$^{CD}$

NoGap$^{CD}$ is a main component in NoGap. As shown in Figure 2.2 (copied from [16]), when using NoGap, designers first use NoGap$^{CL}$ to construct
NoGap parser takes such NoGap designs, and use C++ Application Programmer Interface (API) to construct Abstract Syntax Trees (ASTs), which are another type of descriptions. At the same time, parser writes such ASTs into Parse Units (PUs). Here, NoGap is a composition of many PUs.

Figure 2.2. NoGap Design Flow [16]

### 2.1.3 Spawners

In NoGap system, spawners e.g. Verilog Generator, Assembler Generator, and Cycle-accurate Simulator Generator are implemented to help the designers. As shown in Figure 2.2, the spawners which are implemented read the information inside the NoGapCD, and generate useful outputs such as SystemVerilog design, Assembler, Cycle-accurate Simulator, etc.

The spawners are reusable. Even if a facet has been redesigned, the new facet still can use the old spawners. Since the design of spawner is independent to the facet, by using old spawner, the design time can be saved.

### 2.2 Introduction to NoGapCL

As it was mentioned before, NoGapCL is the default facet of the NoGap system. It is used to construct the Mage, Mase, and Castle. Therefore, this section aims to introduce the syntax and semantics of NoGapCL. Some simple examples were also provided to clearly show how NoGapCL is used.
to construct the design.

The syntax of NoGap\textsuperscript{CL} is quite similar to VHDL, Verilog, and C language. People who already have basic knowledge among those languages won’t have the problems to read or use it. For example, the unit which is called module in Verilog is called Mage FU in NoGap\textsuperscript{CL}. With only the differences in their syntax, both the module in Verilog and the Mage in NoGap can be implemented to perform the same functions, and instantiated into another module/Mage. All other features about NoGap\textsuperscript{CL} will be specified in the following sections.

\subsection{Main Advantages of NoGap\textsuperscript{CL}}

The main advantages of NoGap\textsuperscript{CL} as my supervisor Per Karlström points out in his thesis \cite{16} are specified below.

1. Less micromanagement needed for control path construction.

2. No processor template restriction, providing more freedom for the designer.

3. Support of dynamic port sizes.


5. Pipeline stages can be adjusted easily, different pipelines can be defined for different operations.

\subsection{FU Overview}

The design of a FU is to accomplish specific tasks. There are two kinds of FUs in NoGap. As it was mentioned, they are Mage FU and Mase FU. The design examples and their syntax definition are given below.

\subsection{Mage FU Descriptions}

One Mage FU specification is illustrated in Listing 2.1. Let’s take a first look, it is quiet similar to HDL design. In fact, there are some differences between them. The clock signal and the reset signal in HDL are removed in NoGap\textsuperscript{CL} since both signals are automatically detected and assigned by NoGap. In addition, the operation is divided into two types which are cycle and comb. The cycle operation block performs the timing operation
2.2 Introduction to NoGap<sup>CL</sup>

which is triggered by clock signal, and the <code>comb</code> block performs the combinational operation which has nothing to do with the clock signal. Further to say, the <code>switch</code> statement is frequently used in Mage FU construction, and will be particularly introduced in Section 2.2.3.1.

**Listing 2.1. Mage FU description**

```plaintext
fu alu_example  //Fu declaration
{
  //Signal declaration
  input [3:0] a_in;
  input [3:0] b_in;
  input [2:0] op;  //Operation selection
  input enable;
  output [3:0] c_out;
  signal [3:0] c_tmp;

  //Comb construct logic
  //Arithmetic & Logic operations
  comb
  {
    switch(op)
    {
      0 : %ADD { c_tmp = a_in + b_in; }
      1 : %SUB { c_tmp = a_in - b_in; }
      2 : %AND { c_tmp = a_in & b_in; }
      3 : %OR  { c_tmp = a_in | b_in; }
      .................; //Abridgement of the code
      default : { c_tmp = 0; }
    }
  }
  //Cycle Construct Logic
  //Register Write
  cycle
  {
    switch(enable)
    {
      0 : %NOP {}
      1 : %WRITE { c_out = c_tmp; }
    }
  }
}
```

2.2.3.1 Switch & Clause Selection

The <code>switch</code> statement in NoGap<sup>CL</sup> is quite similar to the syntax in other C like languages. The <code>switch</code> statement is used to select clause operations. For instance, on line 15–23 and line 29–33 in Listing 2.1, these are two typical clause selection examples. The input signals <code>op</code> and <code>enable</code> are the selection signals, and used to indicate a particular clause. If a clause is selected, the operation specified inside it will be performed.
As shown in Listing 2.1, the constant number in front of every clause e.g. “0, 1, 2, and 3” are used to number the possible clause operations that a comb/cycle block can perform. As it can be seen on line 15, the input selection signal op is defined as a three bit wide signal, and can be used to indicate 8 possible operations. However, if the total sub-operations of this block are lower than 8, then default clause statement has to be added. The default clause statement is used to replace the rest of the clause conditions which do not exist in this switch block.

In fact, the most important information for designers are especially those %ADD, %SUB, %AND, and %OR clause names. NoGap uses those clause names to generate the control signals and the control path for a corresponding Mage FU. For example during operating, If a %ADD clause is intended to be performed, the decoder, which automatically generated by NoGap, will assign “1” to the input signal op.

This function of NoGap is implemented to replaced the work which traditionally designers manually construct the instruction decoder as well as the micro code table. Here in NoGap, such tedious and error prone work are automatically handled.

2.2.4 Mase Descriptions

The idea of NoGap design mainly centers around instruction descriptions. The difference between NoGap and HDL is shown in Figure 2.3. The target architecture of NoGap implementation is constructed by many single instruction descriptions which are independent to each other (See Figure 2.3). When using NoGap, designers only think and describe one instruction flow at a time. However, in HDL approach, at a time, designers need to think all the instruction flows, hardware multiplexing and control signal assignment issues. In NoGap design, such issues are automatically handled by the tool.

The NoGap Mase FU description consists of all the possible instruction which are supported in the target architecture. Listing 2.2 shows a simplified example of a Mase FU for a particular architecture. Inside a NoGap Mase, any operation block can construct one or multiple instructions. This depends on how architecture designers describe the operation inside the Mase. By using NoGap, designers have freedoms to specify the Mase FU operations.

As shown on line 11–15 in Listing 2.2, all the Mage FUs used to construct the instructions should be instantiated inside the Mase FU first.
Listing 2.2 is only a small example. In real NoGap implementation, there are usually lots of Mage FUs which are used to construct the target architecture.

As it was mentioned, in NoGap design, every instruction description is independent to each other. Therefore, designers can construct an operation which only corresponds to the expected instructions. Later, NoGap generator go through all the operations specified in the Mas, and automatically generate the target architecture, multiplexers and control signals needed. This feature of NoGap helps to save great human effort and design time. Also, since all the instruction description are independent to each other, it is very easy to add any new instructions supported to an existing design. More detail about Mas description will be discussed in the following subsections.

**Listing 2.2. Mas FU description**

```plaintext
fu data_path
{
    // Signals & Ports declaration
    input [23:0] op_i;
    input [34:0] dat_a_in;
    input [34:0] dat_b_in;
    output [35:0] dat_o;

    //FU Usage List
    fu::decoder_spec<instr_i>() dec_unit;//Decoder
    fu::alu_example(%ADD) alu;
    fu::adder_example(%ADD) adder;
    fu::round(%RND) rnd;
    .................;

    //Phase description
    phase DE;
    phase EX;
    phase WB;

    //Stage description
    stage ff() { cycle { ffo=ffi; } }

    //Pipeline description
    pipeline the_pipe
    {
        DE -> ff -> EX -> ff -> WB;
    }
    .................;

    //Multiple pipeline operations
    operation(the_pipe) alu_1 (dec_unit.alu_type1)
    {
        @DE;
```

Listing 2.2. Mas FU description

1 fu data_path
2 {
3     // Signals & Ports declaration
4     input [23:0] op_i;
5     input [34:0] dat_a_in;
6     input [34:0] dat_b_in;
7     output [35:0] dat_o;
8         ..........; // Abridgement of the code
9
10     // FU Usage List
11     fu::decoder_spec<instr_i>() dec_unit; // Decoder
12     fu::alu_example(%ADD) alu;
13     fu::adder_example(%ADD) adder;
14     fu::round(%RND) rnd;
15     ..........;

16     // Phase description
17     phase DE;
18     phase EX;
19     phase WB;

20     // Stage description
21     stage ff() { cycle { ffo=ffi; } }

22     // Pipeline description
23     pipeline the_pipe
24     {
25         DE -> ff -> EX -> ff -> WB;
26     }
27     ..........;

28     // Multiple pipeline operations
29     operation(the_pipe) alu_1 (dec_unit.alu_type1)
30     {
31         @DE;
```
2.2.4.1 Pipeline Descriptions & Pipeline Operations

A pipeline structure is usually divided into multiple phases (Note that pipeline phases are separated by pipeline registers). In the \texttt{Mase} description, designers have freedoms to decide how many phases are inside a particular architecture. As shown on line 18–20 in Listing 2.2, where clearly shows how to specify all the phases of a pipeline architecture. Designers need to construct the pipeline register \texttt{ff}, which in \texttt{NoGap} is called \texttt{stage}. Another kind of \texttt{stage} is \texttt{wire}, which is a combinational \texttt{stage}. To construct a pipeline, a syntax example shows how to specify a pipeline is shown on line 26–29 in Listing 2.2 (Note that it is possible to have multiple pipeline specifications in one \texttt{Mase} FU, but their names have to be different).

The pipeline datapath is an instruction driven datapath. Based on the input instruction, a datapath performs corresponding operations. As shown on line 33–59 in Listing 2.2, two operations are inside this block. These two operations totally construct 10 instructions. The reason why two operations can construct more than 2 instructions is that each of the clause name e.g. line 39, line 40, and line 54 all represents an individual clause specification. By such clause specification, \texttt{NoGap} can generate all
2.2 Introduction to \texttt{NoGap}^{CL}

Figure 2.3. \texttt{NoGap} vs. HDL
the clause combinations which equals many instructions. As shown in operation_1 block, all the 8 possible combinations are (“%ADD/%RND”), (“%SUB/%RND”), (“%AND/%RND”), etc. After NoGap generation, every such combination forms an individual instruction, and gets an unique instruction opcode assigned by NoGap. The same generation also goes to the operation_2 block.

Worth noting here is that different operations are allowed to use different pipeline formats. The formats used should be specified in the pipeline description. In addition, a particular Mage FU can be placed in different pipeline phase in every Masc operation. NoGap will handle the hardware multiplexing issues.

2.2.5 Decoder Template & Declaration of Instruction Formats

A decoder is used to provided the control signals for a system. The input to decoder are instructions. After resolving an instruction, the decoder sends out all the control signals of such instruction to manipulate the corresponding micro-operations inside the system.

Traditionally, when using HDL approach to design a system, the decoder has to be manually constructed by designers. The manual construction of the decoder is a time consuming, tedious, and error prone task. Besides, if any future designer wants to add new instructions to a finished HDL old design, the new designer has to go through all the previous HDL code, and check all the hardware multiplexing issues. Moreover, they need to modify the previous old decoder and the micro-code table.

In the NoGap system, the decoder construction is totally different. NoGap automatically generate a decoder by using Decoder Template. Designers only need to specify how many instruction formats are supported in a decoder. For example, the “(dec_unit.alu_type1)” on line 33 and the “(dec_unit.alu_type2)” on line 48 in Listing 2.2 use different instruction formats. Such instruction formats should be specified in the decoder.

Listing 2.3 is a typical decoder description example which shows how to define a decoder and specify instruction formats in it. The main concept of the decoder specification are the immediate field assignments which are used to construct the immediate fields for instructions. A typical instruction format declaration is shown on line 17–40. It is possible to have
multiple instruction formats specified inside a decoder.

Listing 2.3. Decoder & Instruction Declaration

```c
fu:template<INSTRUCTION,FLUSH> decoder_spec
def {  
  //Input of instruction opcode
  input auto("#") instruction;//Dynamic port sizing
  output [4:0] rf_a;
  output [4:0] rf_b;
  output [5:0] rf_c;
  output [4:0] rf_w;
  ................;//Abridgement of the code

  //Immediate field declaration
  immediate [4:0] imm_rf_a;
  immediate [4:0] imm_rf_b;
  immediate [5:0] imm_rf_c;
  immediate [4:0] imm_rf_w;

  instruction alu_type1
  {
    source
    {
      rf_a = imm_rf_a;
      rf_b = imm_rf_b;
    }
    destination
    {
      rf_w = imm_rf_w;
    }
  } //End of alu_type1

  instruction alu_type2//Another instruction type
  {
    source
    {
      ................;//Abridgement of the code
    }
    destination
    {
      ................;//Abridgement of the code
    }
  } //End of alu_type2

  .......................;/More types
  } //End of decoder template
```

2.2.6 More Features

There are variety of features and functions which are not introduced in this chapter. If readers or any future NoGap users want to know more about NoGap, we strongly recommend you to look into [16] which is written by the inventor of the NoGap system.
Chapter 3

Introduction to SIMD Sleipnir Processor

The SIMD Sleipnir processor is one of the co-processor in the ePUMA platform. The architecture of ePUMA platform is illustrated in Figure 3.1 (copied from [5]). The main work of the thesis is to use NoGap to implement the SIMD datapath of the Sleipnir processor. One of the purposes was to evaluate the design efficiency of NoGap by comparing the NoGap datapath design with the manually crafted HDL design. Another purpose was to discover any insufficient functions especially in designing SIMD architectures or hidden bugs in NoGap. Below in this chapter, the SIMD Sleipnir processor will be briefly introduced.

3.1 Sleipnir Overview

The Sleipnir’s internal structure is shown in Figure 3.2. Inside the processor core, there is an eight-way SIMD datapath, which supports long vector operations. There are also different memory units inside Sleipnir, where the Program Memory (PM) contains the program for executing, the three Local Vector Memories (LVMs) are the main data memories which store all the input, intermediate, and output data. Besides, the Constant Memory (CM), which can not be written, is the memory which is used to store the coefficient and constant data of the current computations.

There are also different registers such as Vector Register File (VRF), Vector Flag Register (VFLAG), Vector Accumulate Register (VACR), and Special Register File (SRF). The VRF is used to store the vector data, and the maximum number of it are eight vectors. The VFLAG is used to save
the flags which are set by the current computations. The VACR, which is used by Multiply And Accumulate (MAC) or other MAC-like operations, is used to store intermediate computation data. The SRF contains three registers, which are Address Register (AR), Constant Address Register (CAR), and top bottom registers. These three registers are used to address the LVMs, CM, and module addressing respectively. In Sleipnir, there are some hardware units which support advanced data access. there are also some hardware units which are used to communicate with the other seven Sleipnir co-processors and the rest parts of the ePUMA system.

![Figure 3.1. ePUMA Overview [5]](image)

### 3.2 Vector Formats

The computing power of Sleipnir is highly related to its data vector format. If several scalar data can be packed inside one data vector, then several scalar operations are able to be executed at the same time. In Sleipnir, there are three basic scalar data formats which are 8-bit(byte), 16-bit(word), and 32-bit(double). A long vector data is set to be 128 bit wide, so multiple scalar data can be packed inside, which produces different vector types. In addition, if complex numbers need to be packed, then both of the real part and the imaginary part should be stored, which also produces extra vector types. All the possible compositive vector formats are shown in Figure 3.3.
Figure 3.2. Sleipnir Processor Overview
3.3 8-Way SIMD Datapath of Sleipnir

The Sleipnir 8-way SIMD datapath is the target structure, which will be implemented by NoGap in this thesis work. This 8-way SIMD datapath consists of eight identical 16-bit wide data computation lanes. All the operation types supported in this datapath are specified below.

1. Scalar-Scalar operations e.g. adding two scalars and produce a scalar.
2. Vector-Scalar operations e.g. adding a scalar value on every small scalar value in a vector.
3. Vector-Vector operations e.g. adding two vector values.
4. Triangular operations e.g. consecutive MAC operations, multiply two vectors then perform several accumulation by adders.
5. Special operations e.g. Butterflies, Taylor series, DCT.
6. Custom micro-coded operations e.g. Custom operations controlled by micro code program.

As shown in Figure 3.4, the original Sleipnir datapath is divided into three Arithmetic Logic Units (ALUs) stage which are separated by pipeline
3.4 Sleipnir Pipeline

The first ALU is the multiplier stage (MUL), which consists of 16 multipliers that are capable of multiplying two 17 bit numbers (Note that 16 bit with one bit extension two’s complement signed or unsigned numbers). When computing a 32 bit multiplication, it will be performed by several 17 bit multiplications with the additions done in the later ALU stages. The second ALU (ADDER_TREE), which contains multiple small adder units like a tree structure, are mostly used to perform triangular and complex multiplications. The last ALU (ACCR) stage also contains adders. Such adders support accumulate functions which are MAC and MAC-like operations. The logic unit inside ALU (ACCR) is especially used to perform logical operations. There are also many typical DSP functions in this stage such as shifter, rounding, and saturation. The flag signals are also set here.

In Sleipnir, not all the instructions use the whole datapath. In fact, Some of the instructions only go through part of the datapath. Therefore, the instruction operations can be classified into two types, which are long and short datapath operations. They will be introduced in Section 5.4.

In Figure 3.4, one important thing has to be mentioned. During my thesis work, the ePUMA research group is considering to add one more pipeline register into the second ALU “ADDER_TREE” stage, which forms a four stage pipeline datapath. The reason of this consideration is that the previous datapath synthesize results, which was done by the ePUMA group, showed that the critical path is the ALU (ADDER_TREE) stage. For meeting their next new Sleipnir structure, the Sleipnir SIMD datapath which NoGap implement was based on four stage datapath implementation. All the implementation details will be discussed in Chapter 5.

As a matter of fact, one good advantage of NoGap design is that it is very easy to adjust the number of pipeline stages e.g. add or delete an “ff” on the pipeline declaration in Mase description and rearrange the port and signal connections.

3.4 Sleipnir Pipeline

Figure 3.5 (copied and modified from [11]) shows the whole pipeline architecture of the Sleipnir processor. To achieve high parallelism, the Sleipnir
processor has a long pipeline structure. Therefore, many instructions can be executed simultaneously in all the pipeline stages.

As it was mentioned in Section 3.3, not all the instructions pass all the datapath pipeline stages so the number of pipeline stages of every instruction varies a lot. Since the aim of this thesis work is to implement only the datapath stage “D2,D3,D4” (See Figure 3.5) but not the whole Sleipnir processor, the detailed introductions about this long Sleipnir pipeline will not be given in the thesis. More information about Sleipnir and ePUMA platform can be found in [3, 5, 7].

One thing worth saying is that the current Sleipnir micro-architecture has very few hazard detection techniques. The ePUMA platform is still a research project. Also, the various Sleipnir pipelines length of every instruction makes it tricky to avoid hazards. At the moment, no stall will be performed when a hazard occurs. It is up to the programmers themselves to write the code that avoid the possible hazards. There are also methods to solve the hazards e.g. rearrange the code or simply insert No Operation (NOP) instruction while programming.
Figure 3.5. Sleipnir Pipeline Overview [11]
Chapter 4

Preview Of My Work

The following Chapter 5, Chapter 6 and Chapter 7 specify the main implementations and achievements of the works. A short introduction of each chapter are given here for readers to have brief understandings about the working flow and what have been done in this thesis (See Figure 4.1).

4.1 Chapter 5

Chapter 5 illustrates the main implementation of Sleipnir SIMD datapath. First, the design strategy about how to divide the target Sleipnir SIMD datapath into multiple Mage FUs are discussed. Such Mage FUs divided are implemented by NoGap$^{CL}$. Second, the decoder template which was introduced in Section 2.2.5 is used to construct the decoder of the SIMD datapath. Finally, the Mase FU is designed to construct the whole SIMD datapath.

4.2 Chapter 6

The datapath implementation done in Chapter 5 is based on “multiple units” structure, where eight identical units are packed together in one Mage FU. As a matter of fact, both “single unit” and “multiple units” structures had been tried in the SIMD datapath implementation during designing. Chapter 6 mainly focus on comparing both design structures. All the discussions in Chapter 6 clearly states the reasons why we choose to use “multiple units” structure on the final SIMD datapath implementation.
4.3 Chapter 7

Chapter 7 consists of verification and comparison results. All the Mage FUs, Mage FU, and decoder which have been implement in Chapter 5 are simulated and verified by Modelsim tool. The critical path, which is the ADDER_TREE stage, of both NoGap and manually crafted HDL design are synthesized to the Field Programmable Gate Array (FPGA). Both their area and timing synthesize results are reported and discussed.

Figure 4.1. Project Working Flow
Chapter 5

NoGap – Sleipnir SIMD Datapath Implementation

5.1 Overview
The NoGap design framework had been used by previous master student Ching-Han Wang [21] to implement RISC PIONEER and SENIOR processors. There is one more NoGap design case which is introduced in [14]. All such previous works showed the advantages of using NoGap. Since NoGap have not yet been used to implement an advance SIMD architecture, the SIMD datapath of Sleipnir was chosen to be implemented. The detailed introductions about the SIMD datapath implementation will be demonstrated in this chapter.

5.2 SISD vs. SIMD
NoGap had been used to implement two RISC type processors. Both of them are Single Instruction, Single Data (SISD) type architecture. As for the SISD structure, only “one data” is fed into the datapath for every execution. In contrast, for every instruction execution in SIMD structure, there are “multiple data” packed in one data vector and is fed into the datapath.

Suppose we want to perform an “ADD” instruction in both structure. The SISD performs an addition operation only with one adder unit. On the other hand, an 8-way SIMD architecture performs an addition with eight adder units, which eight input data are executed at the same time. Therefore in SIMD, high parallelism and throughput are obtained.
In theory, because of the higher parallelism feature of SIMD, for the low power purpose, the supply voltage of SIMD can be reduced to get a lower clock frequency. Although the clock is lower, the high parallelism and throughput of SIMD, which makes its performance still comparable with the SISD. However, the drawbacks of using SIMD architecture are its high hardware cost and programming complexity. For example, in an 8-way SIMD structure, not only the adder units but also the other functional units and the approximate design areas are eight times greater than a SISD. Besides, the controllability of every functional unit and long data access are also big issues.

5.3 Datapath Under Design

Figure 5.1 (copied and modified from [6]) clearly illustrates the main tasks of the this thesis work. This figures shows the detailed datapath architecture which will be implemented by NoGap. During the thesis work, the ePUMA group did not make many changes in the structure.

The NoGap Sleipnir datapath implementation is according to the behavioral descriptions of ePUMA simulator [2]. No matter how the structure of the datapath (See Figure 5.1) are changed, it should still perform the same behaviors as the ePUMA simulator. The NoGap system is developed to stand at a higher level than the traditional HDL approach, so we only need to use NoGapCL (Section 2.2) to construct the target SIMD datapath. Later, the synthesizable SystemVerilog SIMD datapath will be automatically generated by NoGap.

5.4 Sleipnir Datapath Instructions

There are variety of instructions supported by Sleipnir processor. For example, load-store, jump, call-return, and arithmetic instructions. They could either reference the data from memory and registers or the other parts of processor.

As shown in Figure 5.1, the target SIMD datapath is divided into four stages by five pipeline registers. The first pipeline registers, which are (a0,b0...a7,b7), contain the input data needed for accessing. The second pipeline registers, which are (mul_out[0],[1].....[14],mul_out[15]), contain all the input data to the ADDER_TREE stage. The third pipeline registers, which are extra added in this datapath, are used to separate the orig-
Figure 5.1. Sleipnir Datapath Architecture [6]
inal ADDER_TREE stage into two stages. The fourth pipeline registers, which are \(\text{atree}_\text{out}[0], \ldots, \text{atree}_\text{out}[7]\), contain the output data from the ADDER_TREE stage. Besides, the registers before the Datswitch, which are \((a0,b0,\ldots,a7,b7)\), all have the same data as the first pipeline registers. The last pipeline registers, which are \((\text{res}[0], \ldots, \text{res}[7])\), are used to save output data.

In Sleipnir, the instruction executed in the datapath can be divided into two groups. This depends on how the instructions flow passing through the Sleipnir pipeline. A classification of instruction types and their corresponding datapath pipeline stages are listed in Table 5.1.

*AT=ADDER_TREE; MULT=Multiplication; ACCU:Accumulation.

<table>
<thead>
<tr>
<th>Stage</th>
<th>Short quick</th>
<th>Short bypass</th>
<th>long</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>MUL: xx</td>
<td>MUL: MULT</td>
<td>MUL: MULT</td>
</tr>
<tr>
<td>P2</td>
<td>ALU_1: xx</td>
<td>ALU_1: AT_1</td>
<td>ALU_1: AT_1</td>
</tr>
<tr>
<td>P3</td>
<td>ALU_2: xx</td>
<td>ALU_2: AT_2&amp;3</td>
<td>ALU_2: AT_2&amp;3</td>
</tr>
<tr>
<td>P4</td>
<td>ACCR: ACCU</td>
<td>ACCR: ACCU</td>
<td>ACCR: ACCU</td>
</tr>
</tbody>
</table>

5.4.1 Short Instruction

In Sleipnir datapath, short instructions can be further divided into two types. As shown in Table 5.1, one type is called “short quick” and another is “short bypass”. A small excerpt of Short instructions from Sleipnir instruction set manual [8] are listed in Table 5.2.

5.4.1.1 Short Quick

As shown in Figure 5.2 (copied and modified from [6]) the instruction flow is : (P4).

A “short quick” instruction, which is a special instruction type, can directly enters the datapath and takes the data directly from the pipeline register\((a0,b0,\ldots,a7,b7)\) before the Datswitch. This instruction does not go through MUL (P1) and ADDER_TREE (P2&P3) stages, and only pass through the ACCR (P4) stage.
Figure 5.2. Short Quick Instruction [6]
Table 5.2. Small excerpt of Sleipnir short type instructions

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description (data formats support)</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADD</td>
<td>Addition (byte, word, double)</td>
</tr>
<tr>
<td>SUB</td>
<td>Subtraction (byte, word, double)</td>
</tr>
<tr>
<td>ABS</td>
<td>Absolute (byte, word, double)</td>
</tr>
<tr>
<td>MIN</td>
<td>Minimum (byte, word, double)</td>
</tr>
<tr>
<td>MAX</td>
<td>Maximum (byte, word, double)</td>
</tr>
<tr>
<td>ACCR</td>
<td>Accumulate (word, double)</td>
</tr>
<tr>
<td>CMP</td>
<td>Compare (word, double)</td>
</tr>
<tr>
<td>LOGIC</td>
<td>Logical operation (word, double)</td>
</tr>
<tr>
<td>....</td>
<td>.... others</td>
</tr>
</tbody>
</table>

5.4.1.2 Short Bypass

As shown in Figure 5.3 (copied and modified from [6]), the instruction flow is: (P1) \rightarrow (P2) \rightarrow (P3) \rightarrow (P4).

A “short bypass” instruction takes the data from the pipeline register (a0, b0....a7, b7). Such data taken will be bypassed from MUL (P1) and ADDER_TREE (P2&P3) stages. Finally, such data bypassed will be stored into the third pipeline registers (atree_out[0]....atree_out[7]) and waiting for the next computation in ACCR (P4) stage. Without any data computation, the bypass operations only performs sign extension to each data. The only change is the bit length of the data, and the actual value of data is still the same.

In fact, both of the two type instructions eventually will have the same value before going to the ACCR (P4) stage. The reason of why separate “short quick” and “short bypass” instructions is that sometimes an instruction or a particular data has to be delayed for a number of clock cycles, which depends on the purpose of the instruction used.

5.4.2 Long Instruction

As shown in Figure 5.3, the instruction flow is: (P1) \rightarrow (P2) \rightarrow (P3) \rightarrow (P4).

A “long” instruction passes through all the pipeline stages of the Sleipnir datapath. Long instruction takes the data from the first pipeline register, which are (a0, b0, coeff[0]....a7, b7, coeff[7]) and shown in Figure 5.1. Such
5.4 Sleipnir Datapath Instructions

Figure 5.3. Short Bypass & Long Instruction [6]
data taken then go through all the pipeline stages along with executions. A partial excerpt of long instructions from Sleipnir instruction set manual [8] are listed in Table 5.3.

<table>
<thead>
<tr>
<th>Instruction</th>
<th>Description (data formats support)</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL</td>
<td>Complex multiplication (word, double)</td>
</tr>
<tr>
<td>MAC</td>
<td>Multiply and Accumulate (word, double)</td>
</tr>
<tr>
<td>ABSD</td>
<td>Absolute difference (word, double)</td>
</tr>
<tr>
<td>BF</td>
<td>Butterfly operation (Radix-2, Radix-4)</td>
</tr>
<tr>
<td>ABS</td>
<td>Triangular absolute difference (byte)</td>
</tr>
<tr>
<td>ADD</td>
<td>Triangular addition (byte, word, double)</td>
</tr>
<tr>
<td>MAX</td>
<td>Triangular complex maximum square (word, double)</td>
</tr>
<tr>
<td>MIN</td>
<td>Triangular complex minimum square (word, double)</td>
</tr>
<tr>
<td>....</td>
<td>.... others</td>
</tr>
</tbody>
</table>

5.5 Datapath Implementation

In the NoGap datapath design, all the memory units, registers, and special registers which are mainly used to load or store data in Sleipnir processor, will not be implemented. The main target of the work is only to implement the SIMD datapath part but all the Sleipnir processor. Therefore, those memory and registers units are assumed to have already been implemented in advanced, and all the correct input data will be loaded into our pipeline registers before execution. Below are all the components implemented by NoGapCL and used to construct the SIMD datapath.

5.5.1 Mages in Datapath

A Mage FU of NoGap corresponds to a function unit of the datapath. From flowing subsections, the strategies of how we divide the Sleipnir datapath into multiple Mage FUs will be explained. The multiple units structure is finally chosen to be used to implement all the datapath Mage FU, and the reasons of why using multiple units structure will be discussed in Chapter 6.

5.5.1.1 Mage – MUL

As shown in the Figure 5.4 (copied and modified from [6]), which is directly cut from Sleipnir datapath, this Mage MUL is the multiplication FU, which
totally contains 16 multipliers. Every of such multiplier supports $17 \times 17$ bit multiplications.

The MUL stage also formats the input data into different data types e.g. WORD, DBLE, SUBW. There are totally sixteen 34 bit output data come out for from this stage, and are selected by the “mul_bp” signal. Hence, “mul_bp” is chosen to be the clause condition signal of this Mage.

5.5.1.2 Mage – ADDER_TREE

Inside the datapath, the second (P2) and the third (P3) stages shown in Figure 5.5 (copied and modified from [6]) are composed of three horizontal ADDER_TREE Mage FU (Note that one ADDER_TREE Mage FU is composed of 8 single adder units).

As it can be seen in Figure 5.1, there are totally four ADDER_TREE Mage FUs which are ADDER_TREE_1 (P2), ADDER_TREE_2 and ADDER_TREE_3 (P3), and ADDER_TREE_AB (P4) in this SIMD datapath. In theory, each of the ADDER_TREE unit performs the same functions, and only have differences in their input and output port size, which are listed in Table 5.4. The ADDER_TREE_AB in ACCR (P4) stage supports accumulate operation, and each of the accumulation register is 40-bit wide.

A simplified ADDER_TREE Mage FU example is given in Listing B.1. Since all the single adder cell are working in the same data format e.g. %WORD, %DBLE, %SUBW, and %EXTD, the datw signal, shown on line 67, is chosen to be the clause condition signal to indicate a particular data format for all the 8 single adder in one ADDER_TREE Mage. All the 34, 35, 36, and 40 bit ADDER_TREE Mage FUs are implemented in the similar formats.

<table>
<thead>
<tr>
<th>Name</th>
<th>1 input size</th>
<th>1 output size</th>
<th>#of in/out ports</th>
</tr>
</thead>
<tbody>
<tr>
<td>ADDER_TREE_1</td>
<td>34 bit</td>
<td>35 bit</td>
<td>16/8</td>
</tr>
<tr>
<td>ADDER_TREE_2</td>
<td>35 bit</td>
<td>36 bit</td>
<td>16/8</td>
</tr>
<tr>
<td>ADDER_TREE_3</td>
<td>36 bit</td>
<td>36 bit</td>
<td>16/8</td>
</tr>
<tr>
<td>ADDER_TREE_AB</td>
<td>40 bit</td>
<td>40 bit</td>
<td>16/8</td>
</tr>
</tbody>
</table>

Table 5.4. Data size of each ADDER_TREE Mage
Figure 5.4 Datapath Mage FU - MUL [6]
Figure 5.5. Datapath - ADDER_TREE Mage FUs [6]
5.5.1.3 *Mage* – Datswitch

A Datswitch *Mage* FU on top of the ACCR(P4) stage is shown in Figure 5.6 (copied and modified from [6]). As it is also shown in Figure 5.1, the inputs to Datswitch are either eight 36-bit data from the previous ADDER_TREE stage or sixteen 16-bit data from the first top registers (See Section 5.4).

In the datapath, the Datswitch is mainly used to shape the data, and the formats shaped are decided by the input control signals, which are listed in Table 5.5. As for the Datswitch implementation, the control signals “alu_sign_i, alu_datfsel_i, and alu_insntype_i” are combined together to form a long clause condition signal which can indicate 12 clause conditions ($2 \times 3 \times 2$). One things has to be noted which is that a normal or quick instruction is issued by the programmer during programming.

Since all the horizontal Datswitch units are operating in the same data format, by specifying one of the twelve clause condition names, the corresponding data formation will be performed.

![Figure 5.6. Datapath *Mage* FU - Datswitch [6]](image)

<table>
<thead>
<tr>
<th>Table 5.5. Datswitch input control signals</th>
</tr>
</thead>
<tbody>
<tr>
<td>signal name</td>
</tr>
<tr>
<td>-------------</td>
</tr>
<tr>
<td>alu_sign_i</td>
</tr>
<tr>
<td>alu_datfsel_i</td>
</tr>
<tr>
<td>alu_insntype_i</td>
</tr>
<tr>
<td>alu_quick_i</td>
</tr>
</tbody>
</table>

5.5.1.4 *Mage* – Other Small FU

There are many *Mage* FUs which are also implemented to construct this SIMD datapath. Those *Mage* FUs are listed in the Table 5.6. The locations
of those Mage FUs can be found from Figure 5.1. Here, a logic Mage FU excerpts from our NoGap datapath design is given in Listing 5.1.

**Table 5.6. Other Mage FU inside the datapath**

<table>
<thead>
<tr>
<th>FU name</th>
<th>pipeline stage</th>
<th>function description</th>
</tr>
</thead>
<tbody>
<tr>
<td>sign</td>
<td>P1</td>
<td>signed/unsigned extension</td>
</tr>
<tr>
<td>block</td>
<td>P1</td>
<td>data blocking mask</td>
</tr>
<tr>
<td>logic</td>
<td>P4</td>
<td>8-logical functional block</td>
</tr>
<tr>
<td>accru</td>
<td>P4</td>
<td>8*40 bit accumulate registers</td>
</tr>
<tr>
<td>scale</td>
<td>P4</td>
<td>scale input value</td>
</tr>
<tr>
<td>rnd</td>
<td>P4</td>
<td>rounding</td>
</tr>
<tr>
<td>saturate</td>
<td>P4</td>
<td>saturation</td>
</tr>
<tr>
<td>flag</td>
<td>P4</td>
<td>set flag signal</td>
</tr>
<tr>
<td>outf</td>
<td>P4</td>
<td>shape right output data</td>
</tr>
</tbody>
</table>

**Listing 5.1. Mage FU–Logic**

```python
1 fu logic_block
2 {
3   input [39:0] dat_a_0;
4   .................;//Abridgement of the code
5   input [39:0] dat_a_7;
6   ................;
7   input [39:0] dat_b_7;
8   //Clause condition, four different operation
9   input [1:0] logic_op;
10  output [39:0] logic_res_0;
11  ....................;
12  output [39:0] logic_res_7;
13
14  comb
15 {
16    switch(logic_op)//clause condition
17    {
18      0: %AND //Clause selection_0
19      {
20        logic_res_0 = dat_a_0 & dat_b_0;
21        logic_res_1 = dat_a_1 & dat_b_1;
22        .........................;
23        logic_res_6 = dat_a_6 & dat_b_6;
24        logic_res_7 = dat_a_7 & dat_b_7;
25      }
26      1: %OR //Clause selection_1
27      {
28        logic_res_0 = dat_a_0 | dat_b_0;
29        logic_res_1 = dat_a_1 | dat_b_1;
```
5.5.2 Decoder of Datapath

Since \texttt{NoGap} provides a decoder template for designer, the design of datapath decoder is much similar to the decoder example shown in Listing 2.3. The difference between them is only the number of instruction formats.

5.5.3 Datapath \texttt{Mase}

The \texttt{Mase} FU of SIMD datapath is also constructed in the same similar as we introduced in Section 2.2.4. The Sleipnir datapath \texttt{Mase} is composed of many operation descriptions which all correspond to the assembly instructions of Sleipnir. Every operation specifies how each of the \texttt{Mage} FU that we mentioned in Section 5.5.1 are connected and work together to construct the SIMD datapath. A simplified datapath \texttt{Mase} excerpts from the \texttt{NoGap} design is shown in Listing D.1.

5.6 Results

During the thesis work, the ePUMA group had made some modifications on the behavioral descriptions of ePUMA simulator, there were also bugs in the \texttt{NoGap} system which had to be fixed before proceeding with the datapath implementation, and different design approaches of implementing the SIMD datapath also had been tested (See Chapter 6). Because of such
issues and the work time limited of this thesis work, not all the Sleipnir assembly instructions were implemented in the NoGap datapath Mase.

The first verified version of the NoGap partial datapath design, which all the Mage FUs, datapath decoder, and Mase operations work correctly, took around 10 man weeks. Such 10 man weeks include the time of facing changes on the ePUMA simulator, waiting the bugs fixed of NoGap, and testing different NoGap design approaches. If there were no bugs in NoGap and no changes on the simulator, the datapath implementation time can be lowered.
Chapter 6

NoGap - Comparisons of SIMD Implementation

Because of the high design flexibility of NoGap, there are many approaches which can be used to implement a SIMD architecture. In this chapter, two different design aspects, single unit and multiple units structures, are discussed. In addition, useful design suggestions and techniques are provided for future users who will use NoGap to implement SIMD type architectures. All the design skills and techniques presented below are based on the design experience when implementing the Sleipnir SIMD datapath.

6.1 Motivation

Traditionally, the structure which system designers usually implement is the single lane type SISD. In SISD design, the system designers only concern the behavior of one function unit. On the contrast, in SIMD design, since multiple function units are working in parallel at a time, the system designers need to concern all of them. The SIMD structure is more huge and complicated than the SISD, some of the issues which never happen in SISD will give great design challenges to system designers.

For NoGap SIMD structure implementation, if designers simply consider a SIMD structure e.g. Figure 5.1 as multiple duplicates of a SISD structure e.g. Figure 6.1 (copied and modified from [6]), then designers only need to implement one SISD, and copy it multiple times to compose the SIMD structure. Such design approach, which is called single unit, sounds easy and simple. However, the truth is that it totally makes the SIMD implementation becoming more complicated in NoGap. Therefore,
another design approach, which is called multiple units, is proposed for SIMD implementation in NoGap. The multiple units approach can effectively utilize the superior features of NoGap, and the differences between them are discussed in the following sections.

### 6.2 Size of Mage FU

Figure 6.1, which is cut from Figure 5.1, is a single lane datapath. For a better comparison between the multiple SISD (single unit) and the multiple units approaches, the datapath shown in Figure 6.1 is assumed to be an essential SISD datapath, and the design target is to construct the 8-way SIMD datapath by such two approaches.

The main components inside a datapath are functional units which might be the multipliers, adders, registers, DSP units, and etc. Since the target structure is the 8-way SIMD datapath, we then choose the most commonly used function unit “adder” as a reference.

Listing A.1 shows a `adder_single Mage` done by single unit approach. Listing B.1 shows a 8-way `adder_block Mage` done by multiple units approach. The `adder_block Mage` can represent any of the horizontal `ADDER_TREE Mage` shown in Figure 5.5. The detailed comparisons are discussed below.

#### 6.2.1 Coding Space

Regarding the coding size of the Mage, the 8-way `adder_block Mage` is obviously larger than the `adder_single Mage`. However, there are totally 32 adder units inside the datapath (See Figure 5.1), where the `ATREE_ONE (P 2)` and the `ATREE_TWO_THREE (P3)` stages consist of 24 single adder unit, and the `ACCR (P4)` stage consists of 8 single AB adder unit.

By using NoGap, every single Mage FU used should be specified inside the Mage FU description. If the `adder_single Mage` is used to construct the 8-way SIMD datapath, the `adder_single Mage` has to be instantiated 32 times, which is shown on line 11–18 in Listing C.1. By contrast, if the `adder_block Mage` is used to construct the 8-way SIMD datapath, the `adder_block Mage` only need to be instantiated 4 times, which is shown on line 11–14 in Listing D.1.
Figure 6.1. Single Lane of SIMD Datapath [6]
6.2.2 Signal and Data Permutation

The data format of the SIMD datapath is a vector which is composed of multiple scalar operands. If the adder_single Mage is used, then only one scalar data is executed at a time. In fact, in SIMD datapath, the interconnections between each in/out data of every single adder are more complex. Therefore by using adder_single Mage, the data permutation tasks between each Mage should be carefully manipulated. In addition, the control signal assignment of each adder_single Mage also has the same problem. Hence, If the number of Mage increased, then more time is needed to identify the right signals to each Mage.

By contrast, if the adder_block Mage is used, the situation is changed. When using adder_block Mage, the in/out data of the adder_block Mage is kept in a vector format, so that the data permutation is easy to control. As for the control signal assignment issue, the multiple control signal of the adder_block Mage can be easily assigned by the design techniques introduced in Section 6.3.1.

6.2.3 Clause Selection

The data type of the SIMD datapath is a vector. Since a vector is computed in the 8-way datapath, the executed data of every eight horizontal units are forced to be in the same data formats which can be %WORD, %DBLE, %SUBW, and %EXTD. In the datapath ADDER_TREE stages (See Figure 5.5), for every execution, there are totally 24 adder units work in one of such data formats.

Based on such feature, a good advantage used in the adder_block Mage is that the identical control signal among all the adder units can be chosen as the clause selection signal. Therefore, one clause selection signal can control multiple adder units at the same time. For example, the datw signal on line 67 shown in Listing B.1 is the clause selection signal that controls 8 adder units in the adder_block Mage. To construct the ADDER_TREE stage, we have to specify the clause selection 3 times which is shown on line 50, line 63, and line 73 in Listing D.1.

In contrast with the adder_block Mage, if the adder_single Mage is used, the datw clause selection signal on line 18 shown in Listing A.1 only controls the adder unit itself. To construct the ADDER_TREE stage, we then have to specify the clause selection 24 times which are shown on
6.3 Size of Mase FU

As for the NoGap SIMD structure design, the size of the Mase FU is highly associated with its Mage FU design. Two different design approaches and the possible issues of them have already been discussed in Section 6.2. For the purpose of giving a better understandings for thesis readers, two datapath Mase implemented by different approaches will be shown in this section. (Note that we only illustrate how the ADDER_TREE stage is constructed inside the datapath Mase, the rest of the stages are constructed in a similar format).

Listing C.1 shows a SIMD datapath Mase FU constructed by the single unit adder_single Mages. Listing D.1 is an excerpts from our NoGap datapath design (Note that with few modifications), and it is constructed by multiple units type adder_block Mages FU. Readers can go through these two datapath Mase design and find the issues which we have discussed before.

6.3.1 Special Technique - Control Signal Assignment

For SIMD datapath, as we have said in Section 6.2.3, every horizontal 8 units are forced to work in the same data format e.g. %WORD, %DBLE, %SUBW, and %EXTD.

However, the same data format does not mean that 8 units also work in the same operations. In fact, for some instructions, not all the horizontal adder units are working in the same arithmetic operations e.g. the
even adders perform addition and the odd adders perform subtraction, so that there should have a signal to indicate the operation for each adder unit.

By using `adder_single Mage`, line line 64–68, line 83–87, line 94–98, and line 118–122 in Listing C.1 show how the operation of all the single adder are manipulated. Since each of the `adder_single Mage` is only working itself, the operation signal assignments have to be done separately.

However by using `adder_block Mage`, a useful technique in NoGap can be used. We can implement an extra Mage FU used to control the operations of 8 adder units inside the `adder_block` at the same time. Such control signal Mage FU example is shown in Listing 6.1, and it is used in Listing D.1 shown on line 58.

By using such extra Mage, we only need to specify the clause name “%OP0” in Mage operation description, then the operations of eight adder units can be controlled at the same time. Besides, there is an extra advantages which is that the design time can be sped up by this technique. When every time we want to construct a new instruction, we only need to specify that instruction’s corresponding clause description in Listing 6.1 e.g. line 27, and then specify the clause name e.g.”%OPN” in the corresponding Mage operation.

**Listing 6.1. Single adder Mage FU**

```plaintext
fu_op_assignment_0
{
  input [2:0] op_select;
  // Control signal of an 8-way adder_block
  output [23:0] op_sig_o;
  
  comb
  {
    switch(op_select) // Based on the number of operation.
    {
      // The combinations of all control signal
      // from single unit approach
      0 : %OP0 // operation op0 assignment
      { op_sig_o = 24b0000001......100101; }
      1 : %OP1 // operation op1 assignment
      { op_sig_o = .........................; }
      2 : %OP2 // operation op2 assignment
      { op_sig_o = .........................; }
      // The rest of the combinations
      .........................; // Abridgement of the code
    }
  }
}
```
6.4 Suggestions

“We construct the Mage one time, but we construct the Mase multiple times”

The NoGap system is developed to have high design flexibility. By using NoGap, designers have freedoms to choose their own way to implement their target system. As for the SIMD structure, we have discussed and listed all the possible issues when using different approaches. Based on the experience in Sleipnir datapath implementation and the comparisons shown in this chapter, we highly recommend the future SIMD architecture designers to choose the multiple units design approach when constructing the SIMD architectures by NoGap.

The reasons is that we have to specify a corresponding new operation in Mase FU for every new instruction. Comparing to the multiple units approach, the single unit approach need more complicated and tedious works. In addition to this, the use of the multiple units approach can obtain better hardware controllabilities that multiple functional units can be controlled simultaneously by the technique introduced in Section 6.3.1.

Note: We are not going to say our suggestions presented in this chapter are the best way to implement SIMD architectures. At least the design results have proved that these solutions save design effort, and are more easy to manipulate the SIMD structure. However, not all the SIMD system have the same structure. The final decision depends on the target architecture, the cost factors, and the designers themselves.
Chapter 7

Design Verification And Result

7.1 Single Mage FU Simulation

The behavior simulation of all the Mage FUs has to be done by Modelsim tool before using them to construct the Mase datapath. As shown in Figure 7.1, the first step of the simulation is to connect all the SystemVerilog Mage FUs to the Modelsim tool. All such Mage FUs were originally designed in NoGap\textsuperscript{CL} (See Section 5.5.1) and then generated by the NoGap SystemVerilog generator. After compilation of the Modelsim tool, the next step is to analyse the behavior of such FUs, this is done by manually comparing the behavioral results of the simulation with the behavioral descriptions of the ePUMA simulator [2] (Note that the NoGap SIMD datapath design was based on the descriptions of the ePUMA simulator).

7.2 SIMD Datapath Simulation

The simulation of the SIMD datapath, which was implemented by NoGap, is also done by Modelsim tool. Figure 7.2 illustrates ideas of the datapath simulation. The details are explained below.

As shown in the left side of Figure 7.2. The assembly instruction simulator, which was built by ePUMA research group, is used to model the assembly instructions of Sleipnir processor. This simulator can reports the detailed intermediate data computed by each pipeline stage. Besides, in this simulator, users can issue any of the supported assembly instruction e.g. addw. Users can also set the source and the destination location which
are either memory or register units, and the data stored inside such units can be directly read and set by users.

As shown in the right side of Figure 7.2. After specifying the \texttt{Mage} FU instruction descriptions of the Sleipnir, \texttt{NoGap} is able to generate the corresponding datapath. Besides, each of the instruction generated gets its unique opcode assigned by the \texttt{NoGap} instruction generator.

In Modelsim datapath simulation, since every instruction generated has its unique opcode, if a particular instruction opcode is specified, the datapath decoder generated by \texttt{NoGap} will resolve such particular opcode, and performs the same instruction operations as the assembly instruction simulator. Also in the assembly instruction simulator, since all the values stored in the memory and register units can be manually set by users, the same input values which have been set in the assembly simulator can be manually specified to the datapath simulated by Modelsim.

By the above settings, both initial simulation environments are fixed in the same condition. The datapath co-simulation are done by comparing the datapath computation results shown in Modelsim with the computation results shown in the assembly instruction simulator.
7.3 Synthesize Result

The synthesize flow is done by Precision RTL Synthesize tool. Some important information have to be mentioned. Firstly, only the Sleipnir critical path, which consists of the ATREE_ONE (P2) and ATREE_TWO_THREE (P3) stages in NoGap design, is synthesized. Secondly, the ADDER_TREE, which was synthesized and shown in Figure 7.3 (copied and modified from [6]), is only “one stage” structure which differs from the original “two stages” (P2) & (P3) design (See Figure 5.5). The reasons are specified below.

7.3.1 Synthesize Critical Path

The reason of choosing ADDER_TREE stage is:

The main target of the work is to compare the results of NoGap design with the manually crafted HDL datapath design. In this case, both designs structures compared should be as fair as possible. The design of NoGap is constructed by multiple single instruction descriptions (See Section 2.2.4), which is totally different with the HDL approach (See Figure 2.3). Therefore in NoGap, if not all the instructions of the SIMD datapath are implemented, then both design structures are incomparable. In fact, because of some limitations in this project (See Section 5.6), not all the instructions
Figure 7.3. Synthesized Big ADDER_TREE Stage [6]
were implemented.

Since the pre-synthesize results done by ePUMA group showed that the ADDER_TREE stage is the critical path among all the processor stages. Therefore, the idea is to synthesize the NoGap ADDER_TREE stage design and compare the synthesize results with the same HDL based ADDER_TREE stage design. Such comparison results in a way can be a significant reference to evaluate the overall datapath.

### 7.3.2 One Stage Based Design

The reason of synthesizing “one stage” type ADDER_TREE but not my original “two stages” NoGap ADDER_TREE is:

As it was mentioned in Section 3.3, the ePUMA research group are thinking to further separate the ADDER_TREE stage. However, the HDL ADDER_TREE design [4] designed from the ePUMA group is only an one stage ADDER_TREE structure (See Figure 7.3). Their idea is to use register retiming function of the synthesize tool to separate their one stage ADDER_TREE.

In that case, for making an equivalent one stage structure, one of the extra pipeline stage from the NoGap ADDER_TREE design has to be removed. Then, by manually connecting three ADDER_TREE acMAGE FUs, “34, 35, and 36 bit” (See Table 5.4) and adding all the multiplexers needed, the corresponding comparable structure is done and shown in Figure 7.4 (copied and modified from [6]).

### 7.3.3 Report - FPGA Area

The synthesize results of both design’s ADDER_TREE stage are shown in Table 7.2 and Table 7.1.

Both designs are targeted to the EP4SE530H40C2ES family device on Altera Stratix IV FPGA board, Table 7.1 shows the results of the manually crafted HDL ADDER_TREE design, and Table 7.2 shows the results of the NoGap ADDER_TREE design.
Figure 7.4. Equalization of ADDER TREE [6]
### Table 7.1. Resources usage report - HDL ADDER_TREE

<table>
<thead>
<tr>
<th>Functional Units</th>
<th>Input size</th>
<th>Look-Up Tables (LUTs)/Registers</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Level_0</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ablock_0</td>
<td>34 bit</td>
<td>161/0</td>
</tr>
<tr>
<td>ablock_1</td>
<td>34 bit</td>
<td>171/0</td>
</tr>
<tr>
<td>ablock_2</td>
<td>34 bit</td>
<td>171/0</td>
</tr>
<tr>
<td>ablock_3</td>
<td>34 bit</td>
<td>171/0</td>
</tr>
<tr>
<td>ablock_4</td>
<td>34 bit</td>
<td>164/0</td>
</tr>
<tr>
<td>ablock_5</td>
<td>34 bit</td>
<td>164/0</td>
</tr>
<tr>
<td>ablock_6</td>
<td>34 bit</td>
<td>164/0</td>
</tr>
<tr>
<td>ablock_7</td>
<td>34 bit</td>
<td>164/0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1330/0</strong></td>
</tr>
<tr>
<td><strong>Level_1</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ablock_0</td>
<td>35 bit</td>
<td>167/0</td>
</tr>
<tr>
<td>ablock_1</td>
<td>35 bit</td>
<td>179/0</td>
</tr>
<tr>
<td>ablock_2</td>
<td>35 bit</td>
<td>179/0</td>
</tr>
<tr>
<td>ablock_3</td>
<td>35 bit</td>
<td>179/0</td>
</tr>
<tr>
<td>ablock_4</td>
<td>35 bit</td>
<td>137/0</td>
</tr>
<tr>
<td>ablock_5</td>
<td>35 bit</td>
<td>137/0</td>
</tr>
<tr>
<td>ablock_6</td>
<td>35 bit</td>
<td>137/0</td>
</tr>
<tr>
<td>ablock_7</td>
<td>35 bit</td>
<td>137/0</td>
</tr>
<tr>
<td>cin_adder0</td>
<td>16bits</td>
<td>17/0</td>
</tr>
<tr>
<td>cin_adder1</td>
<td>16bits</td>
<td>17/0</td>
</tr>
<tr>
<td>cin_adder2</td>
<td>16bits</td>
<td>17/0</td>
</tr>
<tr>
<td>cin_adder3</td>
<td>16bits</td>
<td>17/0</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1320/0</strong></td>
</tr>
<tr>
<td><strong>Level_2</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ablock_0</td>
<td>36 bit</td>
<td>123/36</td>
</tr>
<tr>
<td>ablock_1</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_2</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_3</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_4</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_5</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_6</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td>ablock_7</td>
<td>36 bit</td>
<td>138/36</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>1089/288</strong></td>
</tr>
<tr>
<td>lcell_comb</td>
<td></td>
<td><strong>5952/0</strong></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td></td>
<td><strong>9691/288</strong></td>
</tr>
</tbody>
</table>
### 7.3.4 Report - FPGA Timing

The timing synthesize results of both manually crafted HDL and NoGap design are shown in Table 7.3. The critical path in both ADDER_TREE designs are also specified.

<table>
<thead>
<tr>
<th>Design</th>
<th>Maximum Freq</th>
<th>Critical path</th>
</tr>
</thead>
<tbody>
<tr>
<td>Manual HDL</td>
<td>60.042MHz</td>
<td>level_0(ablock_0) → level_1(cin_adder0) → level_1(ablock_0) → level_2(ablock_0)</td>
</tr>
<tr>
<td>NoGap</td>
<td>58.156MHz</td>
<td>ablock_0 → ablock_1 → ablock_2</td>
</tr>
</tbody>
</table>

### 7.3.5 Report - ASIC Area

The ASIC synthesize flow is done by using the synthesize transcript written by the department. The results shown in Table 7.4 are the preliminary synthesize results without setting advanced constraints.

<table>
<thead>
<tr>
<th>Frequency</th>
<th>HDL ADDER_TREE</th>
<th>NoGap ADDER_TREE</th>
</tr>
</thead>
<tbody>
<tr>
<td>100MHz</td>
<td>76440 µm²</td>
<td>117596 µm²</td>
</tr>
<tr>
<td>500MHz</td>
<td>92337 µm²</td>
<td>144669 µm²</td>
</tr>
</tbody>
</table>
7.4 Discussions

7.4.1 Area Comparison

As shown in Table 7.2 and Table 7.1, the total number of LUTs used in the NoGap design is 1.358 times greater than the HDL ADDER design ("NoGap:13160/HDL:9691"). In fact, such synthesis results are reasonable and acceptable. The reasons are explained below.

7.4.1.1 Optimization

First of all, the HDL ADDER_TREE design taken and done by the ePUMA group was actually optimized in advanced, but the first version NoGap ADDER_TREE has not yet made so many optimizations. Second, the efficient of the System Verilog design generated by NoGap is highly correlated with the NoGap System Verilog generator. If later the System Verilog generator can be optimized, the design generated will be more efficient.

7.4.1.2 Architecture Construct Strategy

The target structure of both designs is shown in Figure 7.3. As also shown in Table 7.2 and Table 7.1, there are big difference in the total number of function unit used. Such difference comes from the fact that the different design strategies were done in the HDL design. The HDL ADDER_TREE is constructed by many single adder units as well as the extra carry computation units.

However, in the NoGap ADDER_TREE design, 8 adder cells are constructed together in a ADDER_TREE Mage, which is the adder_block illustrated in Listing B.1. Also, the carry in computation units are also assembled inside the ADDER_TREE Mage. Although the ADDER_TREE design in NoGap is more simple and transparent to manipulate the data, the synthesize report shows that there are many large multiplexers as well as the wide wires used in every adder_block Mage. Such multiplexers and wires are mainly use for data multiplexing and signal transmission. One more reason is that because the System Verilog design generated by NoGap contains a great number of case statements, which may produce numbers of multiplexers when synthesizing.
7.4.1.3 Component Used

A special technique done in the HDL design is that every single adder unit used to construct the ADDER_TREE stage is separated to 3 sub-computation parts (shown in Figure 7.5). Because of such structure, every adder input data for computation is also divided into MSB(high), MID(middle), and LSB(low) parts. When doing computation, the same computations are done separately. The final output are produced by connecting three sub-computation results together.

Many special design techniques were used in the HDL design, which avoid using extra hardware resources. The HDL designers have to think many computation details when performing operations. In contrast to the HDL design, the first version NoGap design did not separate the single adder unit (Note that 8 single adder units construct an ADDER_TREE). Since every single adder unit is not divided, the data computed is longer than the HDL design.

In addition, an addition operation in NoGap design is simply implemented by using the “+” operator, for example line 110 in Listing B.1. Hence, The actual hardware resources used to perform the “+” are decided by the NoGap SystemVerilog generator as well as the synthesize tool.

From NoGap point of view, although hardware utilization are the drawbacks, many working effort and design complexity are saved. As a matter of fact, the same design techniques done in HDL can also be done by NoGap. However, the working time limited does not allow me to finish it. The next version of the optimized NoGap datapath design will be implemented in the future.

7.4.1.4 Extra Adder Units

The used of the extra hardware resources makes the NoGap design larger than the HDL design. In fact, in Sleipnir datapath, there is a special data format called subword “%SUBW”. A word in Sleipnir is defined as 16-bit wide, and a subword is 8-bit wide. For NoGap ADDER_TREE implementation, the critical problem is how to handle such subword operation by a long adder unit. Since only the partial computation results of every output data are needed.
Figure 7.6 illustrates the ideas of how to perform a subword operation in a longer adder cell. The input to every single adder unit is always a long data (34, 35, 36 bit). As it can be seen from the left part of the Figure 7.6, the significant subword data usually are located in two separate parts. In such case, when we directly perform the arithmetic computation on these two long data, the computational errors might happen. As shown in Figure 7.6, the lower part subword data is correctly computed. However, a possible carry in might propagate to the higher subword part. Thus the computational error occurs.

To overcome such issue, the solution are provided and shown in the right part of the Figure 7.6. In the NoGap design, especially for subword operation, every input data is separated into two parts. The same operation are performed in these two part and their computation results are connected together to a long data. In such case, the possible carry propagate problem is solved. However after synthesizing, many extra adder units only used in the subword operation are produced inside the NoGap ADDER_TREE design. The reason of such extra adder units is that because the input data of subword operation is separated, so the extra adder units are generated to compute them.

Worth noting is that every single adder unit in HDL is divided into three computation parts (See Figure 7.5). Besides, special design techniques are also used in HDL design to solve carry propagation issues, so that extra adder units for subword operations are eliminated.
7.4.2 Timing Comparison

As it can be seen in Table 7.3, the timing synthesize results are satisfied. The maximum working frequency of both design (HDL/\textsc{NoGap}-60.042/58.156Mhz) are comparable. Since these are only the synthesize report of the ADDER_TREE stage, so the critical path is all about the data computation path. In HDL design, the carry in unit is implemented separately, so it will be shown in the critical path. However, in the \textsc{NoGap} design, such carry in unit has already been assembled into the ADDER_TREE \textsc{Mage} design (adder\_block), so it is not shown in the critical path.
Chapter 8

NoGap - Suggestions And Improvement

The issues and limitations of NoGap in SIMD architecture design are classified in this chapter. The NoGap development group are currently working to optimize the entire system to be more user friendly and efficient. This chapter expounds the user feedback of NoGap, which helps the NoGap developers to improve the system.

8.1 NoGap - Syntax Issues

A number of insufficient features of NoGap\textsuperscript{CL} are found during the work. Those features have been pointed out here together with the solutions suggested.

8.1.1 NoGap\textsuperscript{CL} - Sign Extension

The sign extension operations are commonly performed during computation. Especially in arithmetic operation, a shorter data is usually sign extended before feeding into a longer size unit.

As a matter of, there are numbers of arithmetic and data shift operations performed in the Sleipnir datapath. The syntax example to achieve the sign extended operation is shown in Listing 8.1. To perform a sign extension, the current approach of the NoGap\textsuperscript{CL} is copy the sign bit Most Significant Bit (MSB) multiple times and connect them together. However, if a long bit arithmetic right shift operation is performed, then the MSB
has to be copied and connected multiple times.

Such inconvenient feature of \texttt{NoGap} should be improved in the next version of \texttt{NoGap}. A syntax suggestion is also shown in Listing 8.1.

\textbf{Listing 8.1. NoGap\textsuperscript{CL} Sign Extension Example}

```c
1 fu sign_extend
2 {
3 input [19:0] a;
4 output [19:0] b;
5 output [24:0] c;
6
7 // Issues of the sign extension
8 comb
9 {
10 // Perform \texttt{a \gg 6}, arithmetic right shift
11 b = \&\{\{a[19],a[19],a[19],a[19],a[19],a[19],a[19:6]\}\};
12 // Assign \texttt{c = a}, need sign extension
13 c = \&\{\{a[19],a[19],a[19],a[19],a[19],a[19],a[19:0]\}\};
14 }
15
16 // Suggestions of the sign extension
17 comb
18 {
19 // Perform \texttt{a \gg 6};
20 b = \&\{6\{a[19]\},a[19:6]\}\};
21 // Assign \texttt{c = a};
22 c = \&\{6\{a[19]\},a[19:0]\}\};
23 }
24 }
```

\textbf{8.1.2 \texttt{NoGap}\textsuperscript{CL} - Loop Statements}

The current \texttt{NoGap} does not support loop function. In fact, loop function have already implemented in HDL e.g. Verilog and VHDL. Especially in SIMD architecture design, the loop function is very powerful because it can easily handle any repeated executions or control multiple identity functional units.

Since there is no loop function in \texttt{NoGap}, all the repeated tasks should be solved separately. An example which shows how to perform repeated tasks by \texttt{NoGap} is illustrated in Listing 8.2. In fact, especially in SIMD design, since there are always many similar operations and function units working at the same time, so it is inconvenient to decompose such operations and units into multiple similar parts. A feasible loop syntax for \texttt{NoGap} have also been suggested in Listing 8.2.
8.1 NoGap - Syntax Issues

Listing 8.2. NoGap<sup>CL</sup> Need Loop Statement Example

```plaintext
1 fu bit_reverse
2 {
3    input [15:0] a;
4    output [15:0] b;
5
6    // Without loop function
7    comb
8    {
9        // Reverse a 16-bit vector
10       b = &{a[0],a[1],a[2],a[3],a[4],a[5],a[6],a[7],a[8],a[9],a[10],a[11],a[12],a[13],a[14],a[15]}&;
12    }
13
14    // New loop function
15    comb
16    {
17        // Support multiple statements
18        for(i=0;i<15;i++)
19        {
20            b[15-i] = a[i];
21        }
22    }
23 }
```

8.1.3 NoGap<sup>CL</sup> - Unsigned and Signed Value Comparison

It is very powerful to have arithmetic comparison for unsigned and signed value. The current NoGap<sup>CL</sup> only supports unsigned number comparison. However, for signed number comparison, the current NoGap<sup>CL</sup>’s solution is to check the MSB between two values and then judge the result. If both MSB are different, then it is easy to judge their magnitude. However, if the MSB of both values are the same (either 0 or 1), then designers have to look into the rest bit of both number to make the judgement.

Besides, designers often care about the overflow issues on DSP application. An overflow happens when a particular result that has been computed exceeds the upper or lower bound range. It is very powerful if designers can directly compare the results been computed with the bound value only by one statement in NoGap<sup>CL</sup>.

The suggested formats to achieve positive and negative values comparison are given in Table 8.2. Besides, a syntax example with the new comparison format is shown in Listing 8.3.
### Table 8.1. Arithmetic comparison format

<table>
<thead>
<tr>
<th>Type</th>
<th>Unsigned value</th>
<th>Signed value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>S'4b1111</td>
<td>U'4b1111</td>
</tr>
<tr>
<td>Value</td>
<td>-1</td>
<td>15</td>
</tr>
</tbody>
</table>

#### Listing 8.3. Unsigned and Signed Value Comparison

```plaintext
1  fu compare_example
2  {
3    input [8:0] a;
4    input [8:0] b;
5    input [19:0] c;
6    output [1:0] d;
7    output [15:0] e;
8  
9   // Simple comparison
10  comb
11  {
12    if( S'{a} > S'{b} )
13        d = 00;
14    else if( S'{a} < S'{8b10101010} )
15        d=01;
16    else        d = 11;
17  }
18
19  // Overflow comparison
20  comb
21  {
22    // Check positive overflow
23    if( S'{c} > S'{16b0111111111111111} )
24        e = 16b0111111111111111;
25    
26    // Check negative overflow
27    else if( S'{c} < S'{16b1000000000000000} )
28        e = 16b1000000000000000;
29    
30    // No overflow
31    else        e = c[15:0];
32  }
33}
```

### 8.1.4 \(NoGap^{CL}\) - Set Parameters

In the Sleipnir SIMD datapath design, there are a number of similar functional units which only have differences in their in/out port sizes. For instance, the Sleipnir ADDER_TREE Mage (1, 2, 3, and AB) discussed in Section 5.5.1.2 are actually four similar Mage FUs with different in/out port sizes (See Table 5.4). However, \(NoGap^{CL}\) has not yet supported a
complete function to define parameters, the parameter setting function of \( \text{NoGap}^{CL} \) is experimenting.

Therefore, to implement such four similar ADDER_TREE \( \text{Mage} \) FUs, an ADDER_TREE \( \text{Mage} \) FU (ex: \text{adder\_block} \( \text{Mage} \)) has to be redesigned four times with the accurate port sizes assigned to each of them.

In reality, the \( \text{Mages} \) design are always large and complex in SIMD structure. If the parameter setting function of \( \text{NoGap}^{CL} \) is supported, then a better \( \text{Mage} \) utilization is obtained, which saves great design effort.

One possible syntax to set parameters in \( \text{NoGap}^{CL} \) is suggested and shown in Listing 8.4. From this solution, if designers want to utilize a particular \( \text{Mage} \) FU multiple times, then the designers have to specify different "PARAMETER" values "w" inside the \( \text{Mase} \) description. Such syntax example is shown in Listing 8.5. By setting different "w" values, an \text{adder\_parameter} FU shown in Listing 8.4 can be reused four times.

**Listing 8.4.** Local Parameter Setting Method

```
1 fu adder_parameter
2 {
3   PARAMETER {w = 34}; // Set local parameter
4   input [w:0] a;
5   input [w:0] b;
6   input [1:0] op;
7   signal [w+1:0] tmp;
8   output [w+1:0] d;
9
10  // Simple adder operation
11  comb
12  {
13    switch (op)
14    {
15      0:%ADD { tmp = a + b; }
16      1:%SUBA { tmp = -a + b; }
17      2:%SUBB { tmp = -b + a; }
18      default: {}
19    }
20  }
21  comb
22  {
23    d = tmp;
24  }
25 }
```
Listing 8.5. Global Parameter Setting

```c
1 fu data_path
2 {
3 ..............................................; // Abridgement of the code
4 // Global parameter {w} setting
5 fu::adder_parameter(%ADD){w = 34} atree1;
6 fu::adder_parameter(%ADD){w = 35} atree2;
7 fu::adder_parameter(%ADD){w = 36} atree3;
8 fu::adder_parameter(%ADD){w = 40} ab;
9 ..............................................;
10
11 operation(pipe) op1(dec_unit.type_1)
12 {
13     @DE;
14     @NUL;
15 ..............................................; // Abridgement of the code
16     @ATREE_ONE;
17     atree0’%ADD,%SUA,%SUBB’;
18 ..............................................;
19     @ATREE_TWO_THREE;
20     atree1’%ADD,%SUA,%SUBB’;
21     atree2’%ADD,%SUA,%SUBB’;
22 ..............................................;
23     @ACCR;
24     ab’%ADD,%SUA,%SUBB’;
25 ..............................................;
26     @OUTPUT;
27 ..............................................;
28 }
29 }
```

### 8.1.5 NoGapCL - Instantiate Mase in Mase

For NoGap Mase implementation, the Mase in Mase function can offer great helps when there are similar instructions which only use few different parts inside a particular structure.

For instance in Sleipnir SIMD datapath Mase implementation, the Figure 8.1 (copied and modified from [6]) illustrates two assembly instruction operation flows which are long arithmetic and logical instructions. Both of such instructions use the same units and the control signals before the P4(ACCR) stage. The only difference is in ACCR(P4) stage where the arithmetic instruction go through the ADDER_TREE_AB Mage and the logical instruction go through the LOGIC Mage. When it comes to the Mase description of both instructions, both of them should have the same statements from P1-P3 and different statements in P4 pipeline stages.

In Sleipnir processor, there are more than two instructions working in
such similar data flows. If the identical parts of each instruction could be packed into a smaller \texttt{Mase} in advance, then later when constructing another similar instruction, such small \texttt{Mase} been packed can be reused and instantiated. Therefore, only the different architecture part of the instruction should be specified. The \texttt{Mase} in \texttt{Mase} approach makes the \texttt{Mase} design become easier and save design effort.

A reusable and instantiable \texttt{Mase} example is shown in Listing 8.6. It is instantiated into the \texttt{data_path} on line 40, line 59 and line 71 to construct the two similar instructions illustrated in Figure 8.1.

Worth noting is that the \texttt{Mase} in \texttt{Mase} function is experimenting in \texttt{NoGap} during the thesis work. After the functional verification, the latest version of \texttt{NoGap} which supports \texttt{Mase} in \texttt{Mase} function will be released.

\textbf{Listing 8.6. Instantiate \texttt{Mase} in \texttt{Mase}}

```c
1 fu mase_small
2 {
3  // Instantiate MAGE FUs
4  ............;
5  fu::decoder_spec<instr_i>() dec_unit_small;
6  // Pipeline construction
7  phase DE;
8  phase P1;
9  phase P2;
10 phase P3;
11
12 stage ff()
13 { cycle{ffo=ffi;} }
14
15 pipeline pipe
16 { DE->ff->P1->ff->P2->ff->P3 }
17
18 // Construction of the identical part
19 operation(pipe) op0(dec_unit_small.type_null)
20 {
21    @DE;
22    @P1;
23    ............; // Abridgement of the code
24    @P2;
25    ............; // Abridgement of the code
26    @P3;
27    ............; // Abridgement of the code
28 } // End of op0
29 }
30
31 fu data_path
32 {
33  // Instantiate MAGE FUs
34  fu::adder_block(\%WORD) ab;
```
8.2 ΝοGap - Issues of Generators

There are many powerful generators implemented inside the ΝοGap system. After the ΝοGapCL design phase, the whole ΝοGapCL designs are fed into such ΝοGap generators. Later, after the generation process, all the corresponding outputs such as SystemVerilog design, instruction opcode file, AST file, etc. are generated. Although the generators implemented
Figure 8.1. Long Arithmetic & Logic Instruction Data flows [6]
in \textit{NoGap} are powerful, there are still some issues which are going to be discussed in the following sections.

### 8.2.1 Issues of Graphs Size

After the generation process, the \textit{NoGap} generators produce two main graphical outputs which are \texttt{mase\_graph} and \texttt{parsed\_graphs}.

The \texttt{mase\_graph} is the graphical representation of the \texttt{Mase} FU and the \texttt{parsed\_graphs} are for the \texttt{Mage} FUs. Such two figures not only describe the detail relations, timing, and internal connections information of the designs but also help the designers to find the potential errors within the designs. For example, the errors might occur when there are signals which are wrong specified or ports which are unconnected in the designs.

In SIMD datapath implementation, some new problems of the generated figures which we did not notice in the previous \textit{NoGap} design were found. In SIMD structure design, the size of the \texttt{Mage} for are usually multiple times than the normal structure. Besides, the relative size and complexity of the \texttt{Mase} are also increased. Under such circumstances, the transformed figures are enlarged a lot than the figures generated from previous simple structure implementations. The figures generated for SIMD structure are harder to inspect and difficult to find the possible error parts of the designs.

The current graph representation strategy is to clearly specify all the information on the generated graphs. The graphical information within a parsed\_graph are interconnection, clause specification, clause statement, ports, and signals of a \texttt{Mage}. The information within \texttt{mase\_graph} are the interconnections of the instruction decoder, control signals, \texttt{Mage} FUs used, and their in/out ports. It is especially difficult to specify the detailed information of a large SIMD design all in one figure.

A possible solution called “Hierarchical Graph Representation” is shown in Figure 8.2, which only shows the top level first layer design. By clicking the box of the graph, designers are able to enter in to the second layer to see more detailed information specified inside. There is also a good advantage by using “Hierarchical Graph Representation”. If there is an error existing in a particular unit, the generator can also highlight the graph box with “Hierarchical Error Marks”. Therefore, designers are easy to find the existing design mistakes, e.g. wrong specifications and port connections in the designs.
Figure 8.2. Hierarchical Graph Representation
8.2.2 Issues of Speed

Some of the previous implementation reports (See [21] Chapter7) pointed out the low generation speed issue of NoGap generators. However, such issue become more serious in SIMD structure design. Because of the even large size of the Mage FUs, Mase FU, instruction decoder, and AST graphs, the time used to compile and generate the significant outputs increases.

Here, one possible solution combines “Two Mode Generator” with “Partitioned Design Generation” is proposed and can be implemented in the NoGap system. This combined solution can help to improve the speed of the compilation and generation process.

In the “Two Mode Generator” NoGap system, users can choose to perform either a normal generation or an advanced generation. When the normal generation is selected, NoGap system compiles all the Mage FUs, Mase FU, and the decoder. After the compilation, NoGap generate all the significant outputs e.g. assembly instructions, SystemVerilog design, AST graphs. However, when designers later modify only part of the design, the normal generation still recompiles and regenerates all the units, which is time consuming. Therefore, an advanced generation mode called “Partitioned Design Generation” can be implemented in NoGap. In this mode, NoGap system only compiles the new parts added or recompiles the old parts modified. The detailed concepts of “Partitioned Design Generation” are introduced below.

The “Partitioned Design Generation” method centers around the Mase FU construction. The Mase FU is constructed by multiple operation descriptions partitioned, and any single Mage FU used inside the operation descriptions should be specified inside the Mase in advance. As shown in Figure 8.3, after the first normal generation of NoGap, all the generated units are in new state. Later, designers modify the MAGE FU_5 and Operation_2 and also add the new Operation_4 inside the Mase.

Since all the Mage FUs and operation descriptions are partitioned and independent to each others, during run time, according to the modification records which can be saved when every new changes were made, NoGap can only recompiles and regenerates such modified parts as well as the instruction decoder.

Table 8.2 list the all the parts which should be recompiled and regener-
Figure 8.3. Partitioned Design Generation
ated. Note that although the MAGE FU_1 has been modified by designers, since it is not utilized in the Mase FU, so when the designers use “Partitioned Design Generation” mode, the MAGE FU_1 will not be recompiled and regenerated.

### Table 8.2. Need Recompiled and regenerated parts

<table>
<thead>
<tr>
<th>parts</th>
<th>Explanations</th>
</tr>
</thead>
<tbody>
<tr>
<td>MAGE FU_5</td>
<td>modified and used Mage FU</td>
</tr>
<tr>
<td>Operation_2</td>
<td>modified Mase operation description</td>
</tr>
<tr>
<td>Operation_3</td>
<td>modified MAGE FU_5 is inside</td>
</tr>
<tr>
<td>Operation_4</td>
<td>new added Mase operation description</td>
</tr>
<tr>
<td>Decoder</td>
<td>new Mase modification</td>
</tr>
</tbody>
</table>

In practice, there are numbers of operation descriptions inside the Mase. For those old and fixed operation descriptions, NoGap do not need to recompile and regenerate them after every small changes. The same concepts also goes to Mage FUs, NoGap only checks the Mage FUs which have been used and modified, and only recompiled and regenerated them.

The “Partitioned Design Generation” mode saves quite much generation time. This concept also can be introduced into the AST mase_graphs construction. In that case, NoGap only need to redraw the modified parts in each graph.

### 8.2.3 Issues of Reported Error Messages

Although NoGap system is a complete tool which helps the designers to implement a system, there are still some features which should be refined to become more user friendly.

As a matter of fact, the current error reporting functions of NoGap are not well implemented. Most of the error messages reported by the NoGap are too blurred to indicate the actual error issues, so that designers have to find and fix the errors by their previous design experiences.

As for SIMD implementation, the whole design is huge. It is getting harder to find the particular mistakes within the design. From my personal point of view, it would be very helpful and user friendly if the next version’s error reporting functions could point out the possible error statements or the errors parts within the design. Therefore, designers can easily fix the
design mistakes with the help from the “Hierarchical Error Marks” function introduced in the Section 8.2.1.
Chapter 9

Conclusions

9.1 Accomplishment

Below are the achievement of my entire work.

1. The estimated implementation time of SIMD datapath was investigated (See Section 5.5.1).

2. The comparisons between the single unit and multiple units design approach were made (See Chapter 6).

3. A better design aspects for implementing SIMD architectures using NoGap were suggested (See Chapter 6).

4. All the individual Sleipnir datapath’s Mage FUs were verified respectively (See Section 7.1).

5. The co-simulation of the implemented datapath with the Sleipnir’s assembly instruction simulator was done (See Section 7.2).

6. The critical path the ADDER_TREE stage of SIMD datapath was synthesized, the synthesize results of NoGap and HDL was compared (See Chapter 7).

7. The potential drawbacks and design limitations of NoGap were discovered, and the suggestions were given to improve the NoGap system (See Chapter 8).
The previous implementations which had been done by NoGap, all indicated that NoGap saved design effort and maintained high design flexibility. Such advantages were also shown in the SIMD implementation, NoGap effectively helped to ease design effort. The functional units used in the SIMD structure are especially in a great number. By using NoGap, all the hardware multiplexing issues were automatically handled. Besides, the decoder of the SIMD datapath was easily constructed by NoGap.

The first verified NoGap implementation of the partial SIMD datapath took around 10 man weeks (note that the time of facing SIMD structure changes, fixing the bugs of NoGap, and using different NoGap design approaches were also included inside).

The critical path of the NoGap design and the original HDL design were synthesized to the Altera Stratix IV FPGA. Comparing the synthesis results of both designs, although the NoGap design used 1.358 times as many hardware resources as the original HDL design, their timing performance is comparable (HDL/NoGap-60.042/58.156Mhz).

Although the implementation of SIMD datapath was a huge task, it is an encouragement that NoGap was proved to support the SIMD implementation. Moreover, another encouragement is that few bugs and insufficient features of NoGap were discovered during my work, and most of the bugs were fixed by the developer. Hopefully more powerful functions of NoGap can also be improved and implemented in the future. NoGap is supposed to be more assistant and powerful before it is released to be an official design framework.
Chapter 10

Future Work

The development group of NoGap is currently working to refine and improve the system. The NoGap system is expected to have more features and become more efficient and user friendly. At the same time, the modifications and improvements of the Sleipnir processor and the ePUMA platform are also proceeding. Therefore, much advanced work about Sleipnir datapath implementation could be done in the future.

10.1 Advanced Implementation of Sleipnir Data-path

The NoGap implementation of Sleipnir datapath is done by describing all the supported instructions of the datapath. Since the number of the instructions and the final architecture of the Sleipnir processor have not yet been settled. Also, since there were bugs in NoGap and timing limitation of my work, I did not implement all the instructions of the Sleipnir. Therefore, the possible future tasks are listed below.

1. Optimize the current NoGap implementation of the Sleipnir datapath.
   - From the results shown in Section 7.3, there are still some room to optimize my NoGap design.

2. Implement all the final settled instructions of the Sleipnir processor.
   - I implemented only few instructions of the Sleipnir in my work, so the rest of the instructions have to be implemented in the future.

3. Synthesize the whole Sleipnir datapath.
   - I synthesized only the ADDER_TREE stage, which is the critical
path of the datapath. It is more accurate if I can synthesize the whole datapath.

4. Implement the Sleipnir control path and the internal sub-memory units.
   - Since my thesis’s target was to implement only the Sleipnir datapath, I did not implement the control path and the memory units. Such things are required if I need to verify the whole Sleipnir processor.

5. Synthesize the whole Sleipnir processor on FPGA board.
   - The NoGap implementation of the Sleipnir processor should finally be implemented and it also has to be synthesized and verified on the FPGA board.

10.2 Advanced Implementation of NoGap

The NoGap system had been used and proved to support the implementation of SISD and SIMD architectures. After the next improvements and optimizations of NoGap, NoGap can be used to implement other architectures e.g. Very Large Instruction Word (VLIW). The possible issues of NoGap VLIW implementation might appear, such as the instruction opcode assignment of the VLIW instruction. Also, the decoder and the Mase construction of VLIW might produce potential issues which have to be discussed.
Listing A.1 shows the single unit type Mage adder of the SIMD datapath.

**Listing A.1. Single adder Mage FU**

```plaintext
fu adder_single
{
  input [33:0] opa_i;  // Input of single adder
  input [33:0] opb_i;
  input [1:0] datw;
  input [2:0] op;     // Operation selection of single adder
  input ext;          // Sign extend signal
  input bp;           // Bypass data signal
  output [34:0] adder_o; // Output of single adder
  signal [33:0] exta;  // Datw_extd shifting signal
  signal [33:0] extb;  // Datw_extd shifting signal
  signal [33:0] tmp;   // Adder result before bypass and sign extension
  signal [33:0] ext;   // Saved data for 32-bit multiplication

  comb
  {
    switch (datw) // 0:WORD, 1:DBLE, 2:SUBW, 3:EXTD
      { // Operation data_type WORD "adder top clause selection"
        0: %WORD
        1: Single adder unit
          {
            0: { tmp = opa_i + opb_i; }
            1: { tmp = -opa_i + opb_i; }
            2: { tmp = -opb_i + opa_i; }
            // MIN
            3: { if (opa_i[33] == opb_i[33])
                 { if (opa_i[32:0] >= opb_i[32:0]) { tmp = opb_i; } else { tmp = opa_i; } }
                 else { if (opa_i[33] == 1) { tmp = opa_i; } else { tmp = opb_i; } }
                 } //MIN
            4: { if (opa_i[33] == opb_i[33])
```
Mage FU - Single Structure

86

//ABS
5: {if (opa_i[33] == 1) {tmp = -opa_i;} else {tmp = opa_i;}}
default: {} 
}

//Operation data_type DBLE "adder top clause selection"
1: %DBLE 
{
  //Single adder unit
switch(op){//0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF 
  { 
    0:{tmp = opa_i + opb_i;} 
    1:{tmp = -opa_i + opb_i;} 
    2:{tmp = -opb_i + opa_i;} 
    //MIN 
    3:{if (opa_i[33] == opb_i[33])
      {if (opa_i[32:0] > opb_i[32:0]) {tmp = opb_i;} else {tmp = opa_i;}}}
    else {if (opa_i[33] == 1){tmp = opa_i;} else {tmp = opb_i;}}}
    //MAX 
    4:{if (opa_i[33] == opb_i[33])
      {if (opa_i[32:0] > opb_i[32:0]) {tmp = opb_i;} else {tmp = opa_i;}}}
    else {if (opa_i[33] == 1){tmp = opb_i;} else {tmp = opa_i;}}}
    //ABS 
    5:{if (opa_i[33] == 1) {tmp = -opa_i;} else {tmp = opa_i;}}
    default: {} }
}

2: %SUBW{ }

//Operation data_type EXTD "adder top clause selection"
3: %EXTD 
{
  //Opa shifting opb maintain
  if (ext == 0) {exta = &{opa_i, 16b0}&; extb = opb_i;} 
  //Opb shifting opa maintain 
  if (ext == 1) {extb = &{opb_i, 16b0}&; exta = opa_i;}
  //EXTD only can perform "ADD" operation 
switch(op){//0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF 
  { 
    0:{tmp = exta + extb; }
    default: {} }
})}))} /End of comb

//Adder output, either bypass input or arithmetic result, 
//and both do sign extension
comb 
{
  if (bp == 1) {adder_o = &{opa_i[33], opa_i} &; }
else {adder_o = &{tmp[33], tmp} &;}
} /End of comb

//End of adder_single MAGE FU
Appendix B

Mage FU - Multiple Structure

Listing B.1 shows the multiple units type Mage adder of the SIMD datapath.

**Listing B.1. ADDER_TREE Mage FU**

```plaintext
fu adder_block
{
  // Input for 8-way adder block
  input [33:0] opa_0_i;
  input [33:0] opa_1_i;
  ......................;
  input [33:0] opa_6_i;
  input [33:0] opa_7_i;
  input [33:0] opb_0_i;
  input [33:0] opb_1_i;
  ......................;
  input [33:0] opb_6_i;
  input [33:0] opb_7_i;

  input [1:0] datw; // Data type clause condition
  input [23:0] op; // Operation selection of each single adder
  input [7:0] ext; // Sign extend signal
  input [7:0] bp; // Bypass data signal

  output [34:0] ablock_0_o; // Output of 8-way adder block
  output [34:0] ablock_1_o;
  ......................;
  output [34:0] ablock_6_o;
  output [34:0] ablock_7_o;

  signal [33:0] exta_0; // Datw_extd shifting signal
  signal [33:0] exta_1;
  ......................;
  signal [33:0] exta_6;
```
Mage FU - Multiple Structure

30  signal [33:0] exta_7;
31  signal [33:0] extb_0;
32  signal [33:0] extb_1;
33  .....................;
34  signal [33:0] extb_6;
35  signal [33:0] extb_7;
36
37  // Adder results before bypass and sign extension
38  signal [33:0] tmp_0;
39  signal [33:0] tmp_1;
40  ......................;
41  signal [33:0] tmp_6;
42  signal [33:0] tmp_7;
43
44  // Save intermediate data for 32-bits multiplication
45  signal [33:0] ext1;
46  signal [33:0] ext3;
47  signal [33:0] ext5;
48  signal [33:0] ext7;
49  signal [1:0] op0;// Operation selection assignment
50  signal [1:0] op1;
51  ......................;
52  signal [1:0] op6;
53  signal [1:0] op7;
54
55  comb // Operation op assignment to each small adder
56  {
57    op0 = op[23:21];
58    op1 = op[20:18];
59    ......................;
60    op6 = op[5:3];
61    op7 = op[2:0];
62  }
63
64  comb
65  {
66    switch (datw) // 0: WORD, 1: DBLE, 2: SUBW, 3: EXTD
67      {
68        // Operation data_type WORD "adder top clause selection"
69            0: %WORD
70            {
71                // Adder unit_0
72                switch (op0) // 0: ADD 1: SUBA 2: SUBB 3: MIN 4: MAX 5: ABS X: DEF
73                  {
74                    0:{tmp_0 = opa_0_i + opb_0_i; }
75                    1:{tmp_0 = -opa_0_i + opb_0_i; }
76                    2:{tmp_0 = -opb_0_i + opa_0_i; }
77                  }
78                  // MIN
79            }if (opa_0_i[33]==opb_0_i[33])
80            {if (opa_0_i[32:0]>=opb_0_i[32:0]) {tmp_0=opb_0_i;}
81                  else{tmp_0=opa_0_i;}}
82            else{if (opa_0_i[33]==1){tmp_0=opa_0_i; else{tmp_0=opb_0_i;}}}
83                  // MAX
84            4:{if (opa_0_i[33]==opb_0_i[33])
85                  {if (opa_0_i[32:0]>=opb_0_i[32:0]) {tmp_0=opa_0_i;}
86                      else {tmp_0=opb_0_i;}}
87                  else{if (opa_0_i[33]==1){tmp_0=opb_0_i; else{tmp_0=opa_0_i;}}}
//ABS
5:{if(opa_0_i[33]==1){tmp_0=-opa_0_i;}else{tmp_0=opa_0_i;}}
default:{}

//Adder unit_1
switch(opi)// 0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF
	{ ...same but use 1 instead of 0; }
//The same for adder unit_2, 3, 4, 5, 6
..............................;

//Adder unit_7
switch(op7)// 0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF
	{ ...same but use 7 instead of 0; }
}

//Operation data_type DBLE 'adder top clause selection'
1 : %DBLE
{
//Adder unit_0
switch(op0) //0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF
{
0:{tmp_0 = opa_0_i + opb_0_i; }
1:{tmp_0 = -opa_0_i + opb_0_i; }
2:{tmp_0 = -opb_0_i + opa_0_i; }
//MIN
3:{if(opa_0_i[33]==opb_0_i[33])
	{if(opa_0_i[32:0]>=opb_0_i[32:0]) {tmp_0=opa_0_i;}
	else{tmp_0=opb_0_i;}}
	else{if(opa_0_i[33]==1) {tmp_0=opa_0_i;} else{tmp_0=opb_0_i;}}
//MAX
4:{if(opa_0_i[33]==opb_0_i[33])
	{if(opa_0_i[32:0]>=opb_0_i[32:0]) {tmp_0=opa_0_i;}
	else{tmp_0=opb_0_i;}}
	else{if(opa_0_i[33]==1) {tmp_0=opb_0_i;} else{tmp_0=opa_0_i;}}
//ABS
5:{if(opa_0_i[33]==1) {tmp_0=-opa_0_i;} else{tmp_0=opa_0_i;}}
default:{}
}
//Adder unit_1
switch(op1)// 0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF
	{ ...same but use 1 instead of 0; }
//The same for adder unit_2, 3, 4, 5, 6
..............................;

//Adder unit_7
switch(op7)// 0:ADD 1:SUBA 2:SUBB 3:MIN 4:MAX 5:ABS X:DEF
	{ ...same but use 7 instead of 0; }
}

2 : %SUBW{}

//Operation data_type EXTD 'adder top clause selection'
3 : %EXTD
{
//Opa shifting opb maintain
if(ext[7]==0){exa_0_0_i=&{opa0_0_i_16b0}; extb_0=opb0_0_i;}
if (ext[0] == 0) { exta_7 = &{opa0_7_i, 16b0}; extb_7 = opb0_7_i; }

// Opb shifting opa maintain
if (ext[7] == 1) { extb_0 = &{opb0_0_i, 16b0}; exta_0 = opa0_0_i; }

if (ext[0] == 0) { extb_7 = &{opb0_7_i, 16b0}; exta_7 = opa0_7_i; }

// EXTD only can perform 'ADD' operation
switch (op0) { 0: { tmp_0 = exta_0 + extb_0; }
    default: {}  }

switch (op7) { 0: { tmp_7 = exta_7 + extb_7; }
    default: {}  }

if (bp[0] == 1) { ablock_0_o = &{opa0_0_i[33], opa0_0_i}; }
else { ablock_0_o = &{tmp_0[33], tmp_0}; }

if (bp[0] == 1) { ablock_7_o = &{opa7_7_i[33], opa7_7_i}; }
else { ablock_7_o = &{tmp_7[33], tmp_7}; }

} // End of adder_block MAGE FU
Appendix C

Mase FU - Single Structure

Listing C.1 shows the Mase construction of the SIMD datapath by the single unit type Mage FU.

Listing C.1. Single unit type Mase datapath

```c
fu : datapath
{
  input [127:0] data_in_a; // Vector data_a input
  input [127:0] data_in_b; // Vector data_b input
  input [127:0] op_i; // Input instruction opcode
  output [127:0] data_out; // Vector data output
  output [127:0] accr_dat; // Data of accumulation register

  fu::decoder_spec<instr_i>() dec_unit; // Decoder instantiation
  // Instantiate all the "32" used single adder units
  fu::adder_single (%WORD) atree_0;
  fu::adder_single (%WORD) atree_1;
  ..............;
  fu::adder_single (%WORD) atree_22;
  fu::adder_single (%WORD) atree_23;
  fu::adder_single (%WORD) ab_0;
  ..............;
  fu::adder_single (%WORD) ab_7;
  ..............; // Other MAGE FUs used in the datapath
  ..............; // Eight times amount than multiple type

  // Pipeline phase declaration
  phase DE;
  phase MUL; // P1
  phase ATREE_ONE; // P2
  phase ATREE_TWO_THREE; // P3
  phase ACCR; // P4
  phase OUTPUT;

  // Register declaration
  stage ff() {
    cycle{ffo=ffi;}
  }
```

// Datapath pipeline declaration
pipeline pipe
{ DE -> ff -> MUL -> ff -> ATREE_ONE -> ff ->
  ATREE_TWO_TREE -> ff -> ACCR -> ff -> OUTPUT; }

// Assembly instruction declaration
// Instruction description _ 0
operation(pipe) op0(dec_unit.type_1)
{
  @DE;
  dec_unit;
  dec_unit.instr_i = op_i;
  @MUL;
  // Eight times fu here, eight times clause selection
  ......................;
  @ATREE_ONE;
  // Other FUs
  ......................;
  // Adder tree, '8' times adder units
  atree_0 '%DBLE' ;
  atree_1 '%DBLE' ;
  ......................;
  atree_6 '%DBLE' ;
  atree_7 '%DBLE' ;
  // Control signal & data permutation
  atree_0.opa_i = mul.ext_mul_res_0;
  atree_1.opa_i = mul.ext_mul_res_2;
  ......................;
  atree_6.opa_i = mul.ext_mul_res_12;
  atree_7.opa_i = mul.ext_mul_res_14;
  atree_0.op = 3b000 ;//Op assignment
  atree_1.op = 3b001;
  ......................;
  atree_6.op = 3b100;
  atree_7.op = 3b101;
  @ATREE_TWO_THREE;
  atree_8 '%DBLE' ;
  atree_9 '%DBLE' ;
  atree_10 '%DBLE' ;
  ......................;
  atree_21 '%DBLE' ;
  atree_22 '%DBLE' ;
  atree_23 '%DBLE' ;
  // Adder tree _ 1 & data permutation
  atree_8.opa_i = atree_0.adder_o;
  atree_9.opa_i = atree_1.adder_o;
  ......................;
  atree_14.opa_i = atree_6.adder_o;
  atree_15.opa_i = atree_7.adder_o;
  atree_8.op = 3b101 ;//Op assignment
  atree_9.op = 3b101;
  ......................;
  atree_14.op = 3b001;
  atree_15.op = 3b001;
  // Adder tree _ 2 & data permutation
  atree_16.opa_i = atree_8.adder_o;
  atree_17.opa_i = atree_9.adder_o;
a tree_22.opa_i = atree_14.adder_o;
a tree_23.opa_i = atree_15.adder_o;
atree_16.op = 3b011;  //Op assignment
atree_17.op = 3b011;
..................;
atree_22.op = 3b001;
atree_23.op = 3b001;
//Other FUs

@ACCR;
ab_0,'%DBLE';
ab_1,'%DBLE';
..............................;
ab_6,'%DBLE';
ab_7,'%DBLE';
//Control signal & data permutation
ab_0.opa_i = sw_0.sw_data_o;
ab_1.opa_i = sw_1.sw_data_o;
..............................;
ab_6.opa_i = sw_6.sw_data_o;
ab_7.opa_i = sw_7.sw_data_o;
ab_0.opb_i = sw_0.sw_datb_o;
ab_1.opb_i = sw_1.sw_datb_o;
..............................;
ab_6.opb_i = sw_6.sw_datb_o;
ab_7.opb_i = sw_7.sw_datb_o;
ab_0.op = 3b010;  //Op assignment
ab_0.op = 3b010;
..............................;
ab_6.op = 3b100;
ab_7.op = 3b100;
//Other FUs

@OUTPUT;
//Final output vector data
data_out =
&{out.d_out_0,out.d_out_1,out.d_out_2,out.d_out_3,
out.d_out_4,out.d_out_5,out.d_out_6,out.d_out_7}&;
}

//Instruction description_1
operation (pipe) op1(dec_unit.type_2)
{  //Mostly the same as op0
  ......................;
}

}  //End of op1

}  //End of MASE datapath
Appendix D

Mase FU - Multiple Structure

Listing D.1 shows the Mase construction of the SIMD datapath by the multiple units type Mage FU.

Listing D.1. Multiple unit type Mase datapath

```cpp
1 fu datathp
2 {
3     input [127:0] data_in_a; // Vector data_a input
4     input [127:0] data_in_b; // Vector data_b input
5     input [127:0] op_i; // Input instruction opcode
6     output [127:0] data_out; // Vector data output
7     output [127:0] accr_dat; // Data of accumulation register
8     fu::decoder_spec<inst_r_i>() dec_unit; // Decoder
9     // Instantiate all the '4' used adder_block units
10    fu::adder_block(WORD) atree1;
11    fu::adder_block(WORD) atree2;
12    fu::adder_block(WORD) atree3;
13    fu::adder_block(WORD) ab;
14    fu::op_assignment_0(%DEF) op_block_1;
15    fu::op_assignment_1(%DEF) op_block_2;
16    fu::op_assignment_2(%DEF) op_block_3;
17    fu::op_assignment_ab(%DEF) op_block_ab;
18    // Other MAGE FUs used in the datapath
19  }
20  // Pipeline phase declaration
21  phase DE;
22  phase MUL; // P1
23  phase ATREE_ONE; // P2
24  phase ATREE_TWO_THREE; // P3
25  phase ACCR; // P4
26  phase OUTPUT;
27  // Register declaration
```
stage ff()
{
  cycle{ff0=ffi;}
}

// Datapath pipeline declaration
pipeline pipe
{
  DE -> ff -> MUL -> ff -> ATREE_ONE -> ff ->
  ATREE_TWO_THREE -> ff -> ACCR -> ff -> OUTPUT;
}

// Assembly instruction declaration
// Instruction description_0
operation(pipe) op0(dec_unit.type_1)
{
  @DE;
  dec_unit;
  dec_unit.instr_i = op_i;
  @MUL;
  .......................;
  @ATREE_ONE;
  // Other FUs
  .......................;
  atree1 '%DBLE';
  op_block_1 '%OP0';
  // Control signal & data permutation
  atree1.opa_o_i = mul.ext_mul_res_o_0;
  atree1.opa_1_i = mul.ext_mul_res_o_2;
  .......................;
  atree1.opa_6_i = mul.ext_mul_res_o_12;
  atree1.opa_7_i = mul.ext_mul_res_o_14;
  atree1.op = op_block_1.op.sig_o;
  .......................;
  @ATREE_TWO_THREE;
  // Other FUs
  .......................;
  atree2 '%DBLE';
  op_block_2 '%OP0';
  // Adder tree_1 & data permutation
  atree2.opa_o_i = atree1.ablock_0_o;
  atree2.opa_1_i = atree1.ablock_1_o;
  .......................;
  atree2.opa_6_i = atree1.ablock_6_o;
  atree2.opa_7_i = atree1.ablock_7_o;
  atree2.op = op_block_2.op.sig_o;
  .......................;
  atree3 '%DBLE';
  op_block_3 '%OP0';
  // Adder tree_2 & data permutation
  atree3.opa_o_i = atree2.ablock1_0_o;
  atree3.opa_1_i = atree2.ablock1_1_o;
  .......................;
  atree3.opa_6_i = atree2.ablock1_6_o;
  atree3.opa_7_i = atree2.ablock1_7_o;
  atree3.op = op_block_3.op.sig_o;
  .......................;
  @ACCR;
  ab '%DBLE';
  op_block_ab '%OP0';
  // Adder tree_AB & data permutation
  ab.opa_o_i = sw.sw_data0_o;
ab.opa_7_i = sw.sw_data7_o;
ab.opb_0_i = sw.sw_data0_o;

ab.opb_7_i = sw.sw_data7_o;
atree.op = op_block_ab.op_sig_o;

// Other FUs

@OUTPUT;

// Final output vector data

data_out =
&out.d_out_0, out.d_out_1, out.d_out_2, out.d_out_3,
out.d_out_4, out.d_out_5, out.d_out_6, out.d_out_7 &;

} // End of op0

 operation (pipe) op1 (dec_unit.type_2)

{ // mostly the same as op0

} // End of op1

} // End of MASE datapath
Bibliography


