Degree Project in Technology
Second cycle, 30 credits

Code Synthesis for Heterogeneous Platforms

FU ZHOUXIANG
Code Synthesis for Heterogeneous Platforms

FU ZHOUXIANG

Master's Programme, Embedded Systems, 120 credits
Date: September 28, 2023

Supervisors: George Ungureanu, Rui Chen
Examiner: Ingo Sander
School of Electrical Engineering and Computer Science
Swedish title: Kodsyntes för heterogena plattformar
Abstract

Heterogeneous platforms, systems with both general-purpose processors and task-specific hardware, are largely used in industry to increase efficiency, but the heterogeneity also increases the difficulty of design and verification. We often need to wait for the completion of all the modules to know whether the functionality of the design is correct or not, which can cause costly and tedious design iteration cycles. Correctness by construction is a methodology that proposes tackling this problem by separating specification and implementation and bridging them through formal methods. Code synthesis is one of these methods which refers to the process of generating low-level codes implementing the desired system from high-level modelling languages. Formulating a generic synthesis method can, in principle, decrease error rates and shorten development cycles since target-specific usage and mechanisms can be systematically taken care of in an automatic manner, whereas the designer needs only to ensure the functional correctness of the high-level specification model. In this respect, this thesis aims to use the open-source Zero Overhead Topology Infrastructure (ZOTI) methodology to formulate a synthesis process from a denotational graph-based representation of an application, tailored towards heterogeneous hardware/software implementations. The case study presents the partially automated synthesis of an open-source streaming processing subsystem on a Xilinx Zynq-based system-on-chip architecture consisting of a software part and a custom hardware accelerator part, where both C software and VHDL hardware are generated from the input model. While the initial results demonstrate a promising path for systemizing the code generation process, certain aspects of the synthesis, such as generating glue code for complex data types (e.g., multi-arrays), are left out of the scope of this thesis and will be explored in future work.

Keywords

Code Synthesis, Heterogeneous Platform, Zero-Overhead Topology Infrastructure
ii | Abstract
Sammanfattning


Nyckelord

Kodsyntes, Heterogen plattform, Zero-Overhead Topologi Infrastruktur
Acknowledgments

I would like to thank George Ungureanu for being my industrial supervisor in Ericsson. He gave a lot of technical help for all kinds of questions and gave me a lot of advice on my thesis. I would like to thank Rui Chen for being my academic supervisor, who gave some advice on my thesis. I would like to thank Ingo Sander for being my examiner.

Stockholm, September 2023
Fu Zhouxiang
# Contents

1 **Introduction**

1.1 Background .................................................. 1

1.2 Problem ...................................................... 5
    1.2.1 Original Problem and Definition ..................... 5
    1.2.2 Engineering Issues .................................. 6

1.3 Purpose ..................................................... 6

1.4 Goals ....................................................... 6

1.5 Research Methodology ..................................... 7

1.6 Delimitations .............................................. 7

1.7 Structure of the thesis ................................. 8

2 **Theoretical Background** ................................. 9

2.1 Hardware Software Co-design ............................ 9

2.2 System Synthesis .......................................... 12

2.3 Component-Based Design ................................. 14

2.4 Software Synthesis ...................................... 16

2.5 Hardware Synthesis ...................................... 18

2.6 Code Synthesis ........................................... 19

2.7 Design Pattern ............................................ 20

2.8 Template-Based Code Generation ....................... 21

2.9 ZOTI .......................................................... 22
    2.9.1 ZOTI-YAML ........................................... 24
    2.9.2 ZOTI-Graph .......................................... 25
    2.9.3 ZOTI-FTN ............................................ 26
    2.9.4 ZOTI-Gen ............................................ 26

2.10 Related Work ............................................. 27
    2.10.1 Synthesis Flows .................................... 27
        2.10.1.1 ForSyDe Deep ................................. 27
        2.10.1.2 Chisel ......................................... 29
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>2.10.1.3 SkePU</td>
<td>30</td>
</tr>
<tr>
<td>2.10.1.4 FeldSpar</td>
<td>31</td>
</tr>
<tr>
<td>2.10.1.5 Bluespec</td>
<td>32</td>
</tr>
<tr>
<td>2.10.2 Model-to-Text Approaches</td>
<td>34</td>
</tr>
<tr>
<td>2.10.2.1 Papyrus</td>
<td>34</td>
</tr>
<tr>
<td>2.10.2.2 Acceleo</td>
<td>35</td>
</tr>
<tr>
<td>2.10.2.3 Xpand</td>
<td>35</td>
</tr>
<tr>
<td>2.10.2.4 Xtend</td>
<td>35</td>
</tr>
<tr>
<td>2.10.2.5 Velocity</td>
<td>36</td>
</tr>
<tr>
<td>2.10.3 Relation with This Thesis</td>
<td>36</td>
</tr>
<tr>
<td>3 Methods</td>
<td>39</td>
</tr>
<tr>
<td>3.1 Research Process</td>
<td>39</td>
</tr>
<tr>
<td>3.2 Experimental Design</td>
<td>41</td>
</tr>
<tr>
<td>3.2.1 Platform</td>
<td>41</td>
</tr>
<tr>
<td>3.2.2 Previous Work</td>
<td>42</td>
</tr>
<tr>
<td>3.2.3 Expected Outcomes</td>
<td>42</td>
</tr>
<tr>
<td>3.2.4 Experiment 1</td>
<td>42</td>
</tr>
<tr>
<td>3.2.5 Experiment 2</td>
<td>43</td>
</tr>
<tr>
<td>3.3 Data Collection</td>
<td>44</td>
</tr>
<tr>
<td>3.4 Validity of Methods</td>
<td>44</td>
</tr>
<tr>
<td>3.5 Hardware/Software Used</td>
<td>45</td>
</tr>
<tr>
<td>3.5.1 Vivado</td>
<td>45</td>
</tr>
<tr>
<td>3.5.2 Vitis</td>
<td>45</td>
</tr>
<tr>
<td>3.5.3 Tel</td>
<td>45</td>
</tr>
<tr>
<td>4 Implementation</td>
<td>47</td>
</tr>
<tr>
<td>4.1 General Structure</td>
<td>47</td>
</tr>
<tr>
<td>4.2 Node</td>
<td>51</td>
</tr>
<tr>
<td>4.3 Port</td>
<td>52</td>
</tr>
<tr>
<td>4.4 Schedule</td>
<td>52</td>
</tr>
<tr>
<td>4.5 Binding</td>
<td>54</td>
</tr>
<tr>
<td>4.6 Type Handling</td>
<td>56</td>
</tr>
<tr>
<td>5 Result</td>
<td>59</td>
</tr>
<tr>
<td>5.1 Evaluation Method</td>
<td>59</td>
</tr>
<tr>
<td>5.2 Model</td>
<td>59</td>
</tr>
<tr>
<td>5.3 Platform</td>
<td>60</td>
</tr>
<tr>
<td>5.4 Result</td>
<td>61</td>
</tr>
</tbody>
</table>
6 Discussion and Conclusion

6.1 Discussion ........................................ 67
   6.1.1 Input Model Format ....................... 67
   6.1.2 Order .................................. 68
   6.1.3 Expand .................................. 68
   6.1.4 Dimension .............................. 69
   6.1.5 Template ................................. 69
   6.1.6 Skeletons ............................... 71
   6.1.7 Post Process .................. ........... 72
   6.1.8 Fault Detection ......................... 73
   6.1.9 Code Pattern ............................ 73

6.2 Conclusion ..................................... 74

References ........................................ 79

A Codes ............................................. 83
   A.1 Files in Synthesis Flows ................. 83
   A.2 Hand-written Code ....................... 93
List of Figures

1.1 A typical reactive real-time embedded system architecture. . . 3
1.2 Design process. ...................................................... 4

2.1 A traditional design flow for an electronic system. .......... 10
2.2 Design flow based on HW/SW co-design .................... 11
2.3 Kienhuis’ Y-chart. .................................................. 12
2.4 Transaction Level Model (TLM) Design flow. ............... 13
2.5 Embedding language $L$ into the host language $H$. ....... 16
2.6 Software synthesis flow. ........................................... 17
2.7 Block diagram. ....................................................... 19
2.8 Code synthesis. ....................................................... 19
2.9 Components of Template-Based Code Generation (TBCG). .. 22
2.10 Zero Overhead Topology Infrastructure (ZOTI) synthesis flow. 24
2.11 Feldspar compiler pipeline. ..................................... 31
2.12 Bluespec compiler. .................................................. 34

3.1 Workflow. .......................................................... 40
3.2 PYNQ-z2 board internal-communication. ..................... 42
3.3 Manual design flow. ................................................ 43
3.4 Autonomous design flow. ......................................... 44

4.1 ZOTI-Graph(left) and ZOTI-Gen(right) example. .......... 48
4.2 Node projection of a single layer. .............................. 51
4.3 Intermediate label. .................................................. 55

5.1 A simplified version of ZOTI-Graph specification as a input. 60
5.2 Hardware platform layout. ........................................ 61
5.3 ZOTI-Graph for geomMean visualization. .................... 62
# List of acronyms and abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>AESA</td>
<td>Active Electronically Scanned Array</td>
</tr>
<tr>
<td>AOP</td>
<td>Aspect-Oriented Programming</td>
</tr>
<tr>
<td>APU</td>
<td>Application Processor Unit</td>
</tr>
<tr>
<td>AXI</td>
<td>Advanced eXtensible Interface</td>
</tr>
<tr>
<td>BSV</td>
<td>Bluespec SystemVerilog</td>
</tr>
<tr>
<td>CDFG</td>
<td>Control Data Flow Graph</td>
</tr>
<tr>
<td>CFAR</td>
<td>Constant False Alarm Rate</td>
</tr>
<tr>
<td>CPU</td>
<td>Central Processing Unit</td>
</tr>
<tr>
<td>CT</td>
<td>Corner Turn</td>
</tr>
<tr>
<td>DFB</td>
<td>Doppler Filter Bank</td>
</tr>
<tr>
<td>DMA</td>
<td>Direct Memory Access</td>
</tr>
<tr>
<td>DSE</td>
<td>Design Space Exploration</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>EMF</td>
<td>Eclipse Modeling Framework</td>
</tr>
<tr>
<td>FFT</td>
<td>fast Fourier transform</td>
</tr>
<tr>
<td>FPGA</td>
<td>Field Programmable Gate Array</td>
</tr>
<tr>
<td>FSMD</td>
<td>Finite State Machine with Data</td>
</tr>
<tr>
<td>GPIO</td>
<td>General Purpose Input/Output</td>
</tr>
<tr>
<td>GPU</td>
<td>Graphics Processing Unit</td>
</tr>
<tr>
<td>HLS</td>
<td>High Level Synthesis</td>
</tr>
<tr>
<td>IOP</td>
<td>I/O peripherals</td>
</tr>
<tr>
<td>IP</td>
<td>Intellectual Property</td>
</tr>
<tr>
<td>M2T</td>
<td>Model-to-Text</td>
</tr>
<tr>
<td>PL</td>
<td>Programmable Logic</td>
</tr>
<tr>
<td>PS</td>
<td>Processing System</td>
</tr>
<tr>
<td>Acronym</td>
<td>Description</td>
</tr>
<tr>
<td>---------</td>
<td>------------------------------------</td>
</tr>
<tr>
<td>RTL</td>
<td>Register Transfer Level</td>
</tr>
<tr>
<td>SDF</td>
<td>Synchronous Data Flow</td>
</tr>
<tr>
<td>SIMT</td>
<td>Single Instruction Multiple Thread</td>
</tr>
<tr>
<td>SysML</td>
<td>Systems Modelling Language</td>
</tr>
<tr>
<td>TBCG</td>
<td>Template-Based Code Generation</td>
</tr>
<tr>
<td>TLM</td>
<td>Transaction Level Model</td>
</tr>
<tr>
<td>UML</td>
<td>Unified Modelling Language</td>
</tr>
<tr>
<td>VTL</td>
<td>Velocity Template Language</td>
</tr>
<tr>
<td>ZOTI</td>
<td>Zero Overhead Topology Infrastructure</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

In this thesis, we will examine the usage of Zero Overhead Topology Infrastructure (ZOTI), a methodology provided by Ericsson for code synthesis. We assume that the input of ZOTI is the output of Design Space Exploration (DSE).

1.1 Background

Embedded systems are everywhere around us, bringing us a lot of convenience. After decades of development, embedded systems become more and more powerful but complicated. When enjoying the convenience brought by powerful embedded systems, their complicated design becomes a challenge for developers. It can be very difficult to make specific designs for each application. Some modularization of components to separate their functionalities is needed, so that the design process can be simpler. To design embedded systems with modularization, we need correctness by construction horizontally, which is to achieve some specific property from a composite of components glued together [1].

To increase the performance of embedded systems, we can increase the density of transistors, parallelize the operation, and introduce heterogeneity. The increasing density of transistors on a chip increases the efficiency steadily. But now Moore’s law comes to a limit on transistors, which means we cannot increase efficiency by decreasing the size of transistors. Parallelism is another approach to increasing the efficiency. Superscalar processors, Single Instruction Multiple Thread (SIMT), and Graphics Processing Unit (GPU) make it possible to process large chunks of data at the same time. But the parallel computation can also reach its limit according to Amdahl’s law, which
is limited by the amount of parallel part of a task. The speed of the overall system can be further increased if different processes are appropriately mapped onto different parts of the platform according to their characteristics, which means heterogeneity, i.e., different applications deployed on different parts of a platform, of platforms required for efficiency improvement.

People now expect a higher performance of a system, and efficiency is one of the metrics. Based on the discussion above, the last way of increasing efficiency, heterogeneity, has a high potential. Thus, we need systems consisting of both general-purpose processors and task-specific hardware so that the efficiency of the system can be optimized. An embedded system is one such system. An embedded system is a combination of software and hardware, which shows heterogeneity. Embedded systems are often used in safety-critical situations. Thus, some timing constraints need to be satisfied. An embedded system is composed of some sequential computing units for software, accelerators, and a bus for communication, which forms the general architecture or platform for computation. Figure 1.1 shows the general architecture of an embedded system. Figure 1.1 contains a memory, a micro-processor, a hardware accelerator, a Digital Signal Processing (DSP) processor, and I/Os, with interconnect fabric connecting with each other. We can see how complex it is. Different parts require different design methods and programming languages. The system contains not only the hardware and software heterogeneity but also the heterogeneity inside the hardware or software categories. For example, we have both DSP processors and microprocessors as software platforms, which means we need to write the programs according to their characteristics. The heterogeneity can make it harder to coordinate the relationship between different components and to test the functionality of the system. This means we have to wait for the completion of all the modules to know whether the design or algorithm is correct. If it is not correct, we need to redo everything from scratch, which is quite time-consuming.
A new design methodology is needed. It should fulfil two requirements: 1) separating “function (what the system is supposed to do) from architecture,” and 2) “separating communication from computation.” [2] Those two requirements can help to “promote the use of formal models and transformations in system design so that verification and synthesis can be applied to advantage in the design methodology” and ensure that “the most important point for functional specification is the underlying mathematical model of computation.” [2]

One way of solving the difficulty of designing a heterogeneous system is “correctness by construction”, which means before implementing the low-level codes, we know it is feasible, and then deploy a design process which preserves the correctness of the original model. The design process can be generally separated into three stages, modelling, validation, and synthesis [3].

In the modelling stage, a formal model, which contains all essential information for functionality, is built in a high-level modelling language. High-level modelling languages are target-agnostic languages, which means they are independent of platforms and have no bias towards either software or hardware. High-level programming languages can make users focus more on the functional aspect instead of the implementation details of a system. The validation stage refers to “the process of determining that the design is correct.” [3] Validation can be done through co-simulation or formal verification. Synthesis means a stage in the design refinement where a more abstract specification is translated into a less abstract specification [3]. For an embedded system, synthesis can be further divided into 3 sub-stages, including mapping to architecture, in which the general structure of implementation is chosen, partitioning, in which sections of the specification model are bound to the architectural units, and code synthesis, in which the details of units are filled out [3]. The process is shown in Figure 1.2. We first do the

---

Figure 1.1: A typical reactive real-time embedded system architecture.
modelling to get a high-level specification model. Then this model needs to be verified in the validation stage. After the verification of correctness, we need to map the model to architecture, partition the software and hardware parts, and synthesize the code into executable files. During this stage, DSE can be used to find the optimal decision of choosing the structure of the platform and partitioning the hardware and software.

Figure 1.2: Design process.

This thesis mainly focuses the last step, code synthesis, which includes both hardware and software synthesis.
1.2 Problem

Although the design methodology shown in Figure 1.2 can achieve correctness by construction, it is time and energy-consuming. Developers usually need to specify the entire system twice, one with the high-level modelling language to describe the specification, and the other with the low-level implementation languages, which are hardware and software codes to run on the platform. Thus, in industry, the most common method is still to directly implement the system with low-level codes and do the testing and verification after everything is done.

However, since each step follows a specific manner, it is not that meaningful that we execute every step manually. Thus, a question is raised. Is it possible to automate the entire design process with a kind of methodology? This problem is too big for a master thesis, so it can be narrowed down to only the code synthesis part: is it possible to formulate a methodology to synthesize high-level codes to low-level implementation codes?

1.2.1 Original Problem and Definition

Ericsson proposed a model-driven method, ZOTI, to solve the last step, code synthesis. ZOTI is a generic methodology for code synthesis associated with a tool ecosystem. ZOTI can generate the corresponding code files which can be deployed on the heterogeneous platforms from a fully-explicit declarative specification model.

The research question is whether the ZOTI framework is sufficient to formulate a generic synthesis flow from a fully-explicit declarative specification model to low-level languages, which is the final stage of a formal design methodology. In this project, the low-level languages are C and VHDL.

The hypothesis is that by using a methodology based on language embedding and templates, we can formulate the synthesis flow mentioned above. Language embedding is a semantics-preserving and structure-preserving mapping between two languages [1]. A template is an abstract and generalized representation written in the target language. To demonstrate the hypothesis, the plan is to synthesize the code on one PYNQ board, implementing two systems obtained via two fundamentally different paradigms: digital hardware design using VHDL and sequential software using C.

\* with all required information
\dagger describes what a system does instead of how it does
1.2.2 Engineering Issues

Engineering issues are: 1) how to format the semantics and structures of the input design in the process of synthesis and 2) how to avoid the template explosion problems. The first issue refers to how to define the format of the input system model to be processed for the system, including the semantics of the model. There should be some regulations that the model must obey. The second issue refers to how to control the number of templates so that we will not have a huge amount of templates. Considering there are infinite cases in the real world, if we create one template for each case, then the number of templates can explode.

1.3 Purpose

The purpose of this master thesis is to formulate a generic synthesis method from high-level specification to low-level codes, which should demonstrate the feasibility to be integrated into a formal system. System engineers and software, hardware and integration developers would benefit from the outcome of the master thesis. The benefit of ZOTI is for a complex system design, modelling can first be done in general, and then formal refinement can be done in particular. The design process can be decomposed into different clearly defined design steps, which can be taken care of by different groups. The outcome can also be used in the ongoing research on automated code synthesis in KTH and Ericsson.

This degree thesis can address questions of ethics and sustainability and avoid ethical issues. The tools used in the project, mainly Vitis and Vivado, are public software tools. The tool developed in Ericsson, ZOTI, is now open-source. The data collected for evaluation also come from an open project called ForSyDe AESA-Radar [4], so they will not contain any sensitive or personal information. The data themselves can also not be falsified or fabricated.

1.4 Goals

The goal of this project is to formulate a generic synthesis method from high-level specification to heterogeneous low-level code, VHDL and C, using the ZOTI framework. This has been divided into the following five sub-goals:
1. (Mandatory) Studying the engineering field, the theoretical concepts to be employed, and a brief overview of related work.

2. (Mandatory) Design of the synthesis flow from high-level languages to VHDL and C with ZOTI.

3. (Mandatory) Implementation, documentation and evaluation of the design flow.

4. (Optional) Analyzing the caveats and proposing and describing an extension for the methodology or the tool suite.

5. (Optional) Exploring the integration and usage of existing synthesis solutions (e.g. Vitis HLS).

The deliverables of the project are a set of code files to implement the synthesis from a fully-explicit model to C and VHDL code files which can be deployed on the PYNQ-z2 board.

### 1.5 Research Methodology

The first step is pre-study, including software and hardware resources, related code synthesis methods, and system design methods. The software resource includes Vivado, Vitis, and ZOTI. The hardware resource includes a PYNQ-z2 board. This step is the preparation for the required knowledge to start.

The second step is to build the C code files with Vitis and VHDL code files with Vivado by hand, to achieve communication within a heterogeneous platform with one PYNQ board. The hand-written code is the target and reference of the synthesis flow.

The third step is to formulate a synthesis flow with ZOTI, which maps high-level code to C or VHDL based on the requirements. During this time, ZOTI may need to be improved by modifying the source code. This is the main work of this thesis project.

The fourth step is to demonstrate the synthesis flow with the ForSyDe AESA-Radar [4]. This step is for the evaluation of the feasibility of the method with respect to the hypothesis.

### 1.6 Delimitations

The implementation part of the thesis project is mainly divided into two parts: 1) implementation of C code files (in Vitis) and VHDL code files (in Vivado)
by hand, and 2) implementation of the automated synthesis workflow from high-level codes to C and VHDL code files. The computation will be deployed on one PYNQ board, but two processing units, Central Processing Unit (CPU) and Field Programmable Gate Array (FPGA). After that, the documentation of the design flow will be written.

The DSE part and the modelling part are not part of the master project. Other low-level programming languages are also not part of the targets. Some deeper engineering problems, like complex data types and sizes of arrays, are also skipped. Thus, the outcome is a well-structured and documented semi-automatic synthesis flow with some of the final details to be added manually for demonstration purposes.

### 1.7 Structure of the thesis

Chapter 2 presents theoretical background information about the embedded system design process. Chapter 3 presents the methodology and method used to solve the problem. Chapter 4 presents the implementation of the algorithm. Chapter 5 presents the result. Chapter 6 presents the discussion and conclusion of this thesis.
Chapter 2
Theoretical Background

The purpose of this section is to give a general introduction to the background knowledge of the thesis, including the general design process, synthesis methods, and related works.

2.1 Hardware Software Co-design

Hardware/Software Co-design is a design methodology for electronic systems design, which aims at exploiting the trade-offs between the hardware and software. Some applications need to be executed on the hardware components, like FPGA, while others need to be run on the processor, like CPU.

The traditional design method is shown in Figure 2.1. The system design starts from a set of specifications and requirements on functional and non-functional properties. We first decide how functional properties can be met. Then, the hardware architecture and the partitioning of hardware and software functionalities should be decided. In traditional methods, this decision is totally based on the experience of the developers. After the decision, the design process is divided into two groups. One is for hardware development, and the other is for software development. The hardware group works first on implementing a prototype for the architecture because the software development is usually dependent on the hardware platform. The software group doesn’t start to work until a hardware prototype is finished. After the completion of both hardware and software, we need to port the software to the hardware to form an entire system. We also need to do some testing and debugging on the prototype. Based on the result, the design can either finish and come to the implementation phase or come back to the manual decision part. Since it is quite rare for a design to succeed the first time, we usually
need to iterate the design steps several times until the outcome is accepted.

![Diagram of a traditional design flow for an electronic system.](image)

**Figure 2.1:** A traditional design flow for an electronic system.

Since the software group needs to wait for the hardware group. The design cycle can be quite long. With the increase in complexity of target systems, developers’ past experience in partitioning cannot be that reliable.

Hardware/software co-design solves this problem. By running the software on a virtual prototype of hardware instead of a physical one for performance estimation, software and hardware can be designed at the same time. Figure 2.2 shows the general step of this method. Firstly we still need the specifications and requirements of the target, which is the same as the traditional method. Then we need to specify the behavior model of the system. The language for the behavior specification should be a high-level modelling language with formal semantics. Meanwhile, a hardware library of components to be used is provided. Both of them are the inputs to the design loop. In the design loop, a wide design space defined with several axes like candidate hardware architectures and hardware/software partitioning solutions is ready to be explored [5]. A performance estimation can be conducted through the hardware software co-simulation and give some numerical results about the metrics, like power, throughput, and so on. Based on the metrics, the design can be improved. This process, which is called DSE, can be iterated several times to find the optimal design solutions. Then the architecture can be fine-tuned to determine the details.
of the architecture system. Hardware/software co-verification can be done to verify the correctness of the functions of the entire system. After the verification, the last step is the synthesis of hardware and software codes for the implementation of the prototype.

![Design flow based on HW/SW co-design](image)

Figure 2.2: Design flow based on HW/SW co-design

This thesis focuses on the synthesis part. Instead of writing the code according to the specification models by hand, the project aims at generating the hardware and software codes automatically from a high-level code with all the required information.
2.2 System Synthesis

The core issues of embedded system design are shown in Figure 2.3, which is called Y-chart, or more specifically, Kienhuis’ Y-chart [6]. There are three important issues in the process of embedded systems design, architecture, mapping, and application. The architecture refers to the hardware platform on which the codes can run. It contains the computing units, memories, and other Intellectual Property (IP)s, as well as how they are connected with each other. Applications refer to a set of processes or tasks, which should be run on the architecture. Applications are the codes after synthesis. Mapping refers to the relationship between applications and computing units. Mapping is the process of assigning different applications to the computing units on the architecture, i.e. task 1 executes on CPU, and task 2 executes on FPGA. We can see that in Figure 2.3, the top three boxes are exactly architecture, applications, and mapping.

![Figure 2.3: Kienhuis’ Y-chart.](image)

Once architecture, applications, and mapping are determined, the model can be put into the performance analysis step to evaluate the performance model. A performance model reflects the performance in terms of some
metrics, like delay, power consumption, utilization, and throughput. After the performance analysis, some quantitative data will be given as a result, which is called performance numbers. Based on the performance numbers, we can decide whether the model fulfills our requirements or whether it is good enough.

Apart from the boxes, there are also some arrows in Figure 2.3. The black arrows refer to the normal workflow beside the box, which is forward. The dashed arrows mean that we can do the backtracking to the input of the performance analysis step, like reducing the number of applications, adding more components to the architecture, or changing the mapping of applications to computing units of the architecture.

Here is an example of system synthesis called Transaction Level Model (TLM) [7], which is shown in Figure 2.4. TLM is a methodology to model complicated electronic systems. We have the application, mapping, platform, and constraints, with which we can generate the TLM model. The component models also need to be provided because the parameters and configurations of components are needed for estimation. The specification model can be evaluated by estimating the metrics. Based on the evaluation results and constraints, some optimization can be applied to modify the mapping, application, or platform. The iteration can proceed again and again until the model is accepted.

![Figure 2.4: TLM Design flow.](image)

System synthesis is the work to be done prior to the code synthesis to get all the required information. The output of the system synthesis can be regarded as an intermediate representation, which is the input to the synthesis flow of this project.
2.3 Component-Based Design

There is an important trend that we need to translate domain-specific languages to target languages, but most of the translation is in an ad hoc manner. In this section, we will discuss how to transform the model in a coherent manner, which means preserving the structure and semantics. One of the advantages is the vertical correctness [1]. We do the programming in C instead of some machine codes, because we believe that the compiler will not make mistakes. The compiler translates the C code to assembly code in a specific method instead of randomly. Here we want to do the same thing, which is to formulate the translation step.

Component-based design means that when doing the design, we need to divide the entire system into components and use the design of the components respectively. Each component is similar to a behavior, which contains the observable states and output reactions corresponding to the input stimuli [8]. The framework of component-based contains two basic units:

- atomic components, \( C = \{ C_i \}_{i \in I} \) characterized by their behaviors,
- glue operators, \( GL = \{ gl_k \}_{k \in K} \) which are memoryless composition operators.

where \( I \) and \( K \) are sets of identifiers [8].

Translation between two component-based designs can be formulated in language embedding [8]. Embedding contains two characteristics: structuring preserving and semantics preserving. Supposing we need to translate one language \( H \) to another language \( L \). Here we assume that \( L \) is more expressive than \( H \), but \( H \) is more abstract than \( L \). \( H \) represents high-level languages, while \( L \) represents low-level languages. We need to first do the structure-preserving mapping. Structure-preserving mapping means that all the components in the original language \( L \), including atomic components and glue operators, should be contained in the target language \( H \). This can be formulated by

\[
\chi(t) = \begin{cases} 
\chi(X) \in C_L, t = X \in C_H \\
\chi(gl)(\chi(C_1), \ldots, \chi(C_n)) \in GL_L, t = gl(C_1, \ldots, C_n) \in GL_H.
\end{cases}
\]

Here \( \chi \) means the transformation from \( H \) to \( L \). The first line means for each atomic component \( C_i \) in \( H \), its transformed component in \( L \) is \( \chi(C_i) \), which is a one-to-one relation. The second line means for a combination of atomic
components glued together by $gl$ in $H$, its transformation result in $L$ is a combination of transformed atomic components glued by transformed glue.

Then we need to add some additional glues and ports to orchestrate the behaviors of each component, respecting the semantics of $H$, which can be formulated by

$$\sigma(t) = \sigma_1(t)(\chi(t), \sigma_2(t)) \approx t,$$

where $t \in H$ and $\sigma(t) \in L$. $\sigma$ is a semantic-preserving function from $H$ to $L$. $\sigma_1$ and $\sigma_2$ are two auxiliary functions, where $\sigma_1(t)$ refers to the semantic glue, and $\sigma_2(t)$ refers to the execution engine. The approximating sign $\approx$ means the two units on both sides are strongly bisimilar. This means the observable behaviors of both sides are the same. With the same inputs, their outputs would be the same. This expression shows we need to add additional glue and engine to orchestrate the translated components.

Figure 2.5 shows the principle which we have already discussed. The left model is written in a high-level language $H$, while the right model is written in a low-level language $L$. On one hand, those two figures are very similar. Everything in the left model, including components and edges, is in the right model. On the other hand, we can see that the right model contains more elements than the left model, like an extra engine, some ports, and more edges.
Figure 2.5: Embedding language $L$ into the host language $H$.

In this thesis, a synthesis flow from a high-level language to low-level languages is similar to the translation from $H$ to $L$, so the basic idea, which refers to structural preserving and semantics preserving, needs to be fulfilled in the process of transformation. The generated code should have the same structure as the high-level model. More components, like auxiliary functions, and glues, like intermediate variables, will be added to the generated code to ensure the generated code can have the same behavior as the high-level model.

2.4 Software Synthesis

Software synthesis refers to the synthesis process from a high-level specification to a machine code that can be run on the target platform. Because of the complexity of the machine code programming, or said assembly code programming, software synthesis is not a one-step process. We usually need to first synthesize the high-level specification to low-level codes, like C/C++, and then transform the low-level codes into machine codes through a compiler.

Figure 2.6 shows the software synthesis process. For the first step C/C++ code generation, it can be divided into two parts, code generation, and hardware-dependent software generation. Code generation produces the flat
C code from the model. Hardware-dependent software generation produces all the drives and support codes which are necessary to run the generated C code on the given platform, regarding multi-tasking, internal communication, external communication, and binary generation [7]. Hardware-dependent software generation also creates build and configuration files. Then the cross compiler and linker will take the C codes and support codes to generate the binary codes to be run on either the hardware platform or the virtual platform. Since hardware-software co-design also requires the design of the platform, the compiler should be a specific compiler towards the platform instead of a general one.

![Software synthesis flow diagram]

Figure 2.6: Software synthesis flow.

In this thesis, software synthesis only includes the process from a specification model to C code, which contains the normal part and the hardware-dependent part. The rest compilation and linking part is left to the specific tool developed by companies, which is Vitis in this project.
2.5 Hardware Synthesis

Hardware synthesis refers to the synthesis process from a high-level specification to Register Transfer Level (RTL) code written in a hardware description language, like Verilog and VHDL. An RTL component library is needed for component selection. The synthesis process starts from a given specification and compiles it into some intermediate representations of different tool models. The tool model needs to be used to estimate the performance on different metrics and to perform allocation, binding, and scheduling tasks [7]. The allocation task selects components from the RTL library and defines their connectivity. The binding task binds variables to registers and operations to functional units. The scheduling task assigns operations to clock cycles. After the finishing of the three tasks, the metrics can be estimated. Based on the evaluation results, some optimization can be performed. The optimized model can then generate the corresponding hardware description language.

The high-level specification model can contain no or few design decisions. C code, which is widely used in High Level Synthesis (HLS), doesn't contain any design decisions. Control Data Flow Graph (CDFG) shows control and data dependency. It exploits the concurrency in the sequential C codes. Finite State Machine with Data (FSMD) gives more information because its basic structure is a finite state machine. It also provides an accurate estimation of the performance. The generated RTL netlist codes contain all the decisions. The RTL is usually composed of two parts, a controller and a datapath, as shown in Figure 2.7. Both of them have inputs and outputs from outside and each other. The controller coordinates the execution of components in the platform, while the datapath performs some operation on the data.
In this thesis, the hardware synthesis aims at generating hardware code in VHDL at RTL level with both combinational and sequential parts, with the same structure as Figure 2.7, from a high-level specification.

### 2.6 Code Synthesis

Lu et al. proposed a method for code synthesis [9], which is shown in Figure 2.8.
From Figure 2.8, we can see that there are three layers for software synthesis, process layer, subsystem layer, and system layer. The synthesis method is a bottom-up structure, which starts from the process layer, then the subsystem layer, and finally the system layer. Although in Figure 2.8 the process layer is at the top and the system is at the bottom, we often consider system at the top as the root, subsystem in the middle as the middle nodes, and processes as leaves at the bottom, which is a tree structure.

The process layer corresponds to a single process with a skeleton and a function inside. To deal with the process layer, we first need to map data types from specification to implementation language. Some of the data types are the same, but complicated data types like `struct` are not. Then we need to select the corresponding skeleton for template programming and determine the parameters. After that, we need to translate the inner function. Combining everything together, we can get the process description. At the subsystem level, it is a combination of processes and other subsystems. For software, a subsystem is a hierarchy of functions. Here the schedule is needed for different processes by considering the order. For hardware, a subsystem is a netlist of components. The system layer is the top layer of everything, which can be a single top-level component for hardware or a main function for software.

In this thesis, the input model and the synthesis flow are similar to what is shown in Figure 2.8. The input model also has a hierarchy structure of functions or components. But the synthesis flow is top-down from root to leaves instead of bottom-up.

### 2.7 Design Pattern

Design Patterns are typical solutions to commonly occurring problems in software design. A design pattern is like a blueprint which can be customized to fulfil specific problems. A design pattern is not a piece of code which can be run directly. It is a template where we need to fill in some information. Different from an algorithm, which is a step-by-step method, a design pattern is more like a framework from which we can see the results and features. Design patterns are described very formally so that people can reuse them. There are mainly 3 kinds of design patterns, creational patterns, structural patterns, and behavior patterns.

Design patterns are “a toolkit of tried and tested solutions to common problems.” [10] Knowing one pattern means that we know the method of solving all problems of the same type. Design patterns can also “define a common language for programmers to communicate more efficiently.” [10]
A complicated solution can be difficult to explain, but if it follows a specific design pattern, then other programmers can easily understand it because they don’t need to explore the algorithm from scratch.

In this thesis, the design pattern for the input model is a tree structure, written in ZOTI-Graph specification. For more details, please check Section 2.9.2 to have an intuitive understanding.

2.8 Template-Based Code Generation

Automatic programming is to transform specifications into codes written in target programming languages [11]. A user can define what he expects and then the code can be generated automatically by the automatic programming machine without the assistance of humans [11]. Two conceptual tools, abstraction of codes and reusability of models, are raised to achieve automatic programming [12]. Template-Based Code Generation (TBCG) is one of the methods satisfying the two requirements.

TBCG is a synthesis technique for code generation. The input is a platform-independent model, and the output is a piece of source code. Figure 2.9 shows the general concept of TBCG. A template is an “abstract and generalized of the textual output it describes.” [11] A template has a static part, which remains the same in the output, and a dynamic part, which is embedded with splices of meta-code. Design-time input is an input model we expect, like a class with some attributes. A template uses the design-time input as an argument. Run-time input is an instance of design-time input, which follows the structure of design-time input so that it can be combined with the template. Templates are executed by a template engine. The template engine uses the template as the basic structure of the generated code, asks for information from run-time input, and generates the output. It computes the dynamic parts and replaces them with static texts according to run-time inputs.
In this thesis, **TBCG** is one of the main ideas for code generation. A library for templates in target languages are written prior to code synthesis. All the codes generated are based on some specific templates with details added.

### 2.9 **ZOTI**

**ZOTI**, an initiative in R&D for high-performance application on heterogeneous architectures, is designed for synthesizing a declarative model in high-level to low-level executable codes [13]. The full name of **ZOTI** is Zero-Overhead Topology Infrastructure. Zero-Overhead means directly generating low-level codes from high-level abstractions instead of generating the codes at a higher abstraction level. Some intermediate abstraction levels are skipped so that the overhead can be reduced. Topology means when generating codes with **ZOTI**, the entire system design should be considered as a whole.
The input model and intermediate representations of ZOTI have no bias to either software or hardware programming languages, which achieves the HW/SW co-design principles [14]. Infrastructure means models are used as intermediate representations. ZOTI is one of the bridges to narrow the gap between functional models and vendors on different platforms. It can help develop high-performance applications in heterogeneous platforms through model-driven engineering.

Since synthesis is really a complicated step, ZOTI is divided into several different parts, including ZOTI-YAML, ZOTI-Graph, ZOTI-FTN, and ZOTI-Gen, with each part responsible for different functionalities. The method used for the synthesis flow is called Genny, or “the ZOTI code generator” [14]. Figure 2.10 shows the general synthesis flow of ZOTI. The synthesis starts from the output of a DSE system, which is an intermediate representation with all required information for code generation. ZOTI-Graph receives a system model and transforms it into an annotated graph structure. ZOTI-FTN receives types and transforms them into a type structure. The Graph-to-Blocks part, which is written as Python scripts, is the core part of ZOTI, which is now defined by users. The script receives the type structures and the annotated graph structure and transforms them into the code blocks structure and type glue. ZOTI-Gen receives the code block structures and type glue, and uses templates from the library for the implementation of code generation. The languages used in all stages of ZOTI are embedded in yaml, which means the model specifications follow the yaml format.
In this master thesis, ZOTI is the main tool to be used, especially for ZOTI-Graph and ZOTI-Gen, and the Genny methodology is the guidance to the framework of this project.

### 2.9.1 ZOTI-YAML

ZOTI-YAML is a yaml extension tool used in ZOTI-Graph specification to make the model specification readable. It contains a set of keywords which have some specific functionalities. Each keyword is an abbreviation for a complicated situation so that the yaml codes are human-friendly.

Since ZOTI is a methodology for synthesis, the last step of system design, the input of ZOTI should be the output of another tool, which means its input doesn’t need to be human-readable. A long piece of code is acceptable as long as it is machine-friendly. However, as an independent component of the entire
design flow, we need to test ZOTI individually. Now the input yaml files also need to be written by hand. But ZOTI-YAML is still not mandatory. In this master thesis, ZOTI-YAML is not used.

2.9.2 ZOTI-Graph

ZOTI-Graph is used to generate a graph structure of the input specification model. The input is a specification model written in yaml, which specifies all the necessary functional details. ZOTI-Graph has three outputs, including an intermediate representation written in yaml for future processing, a node graph showing the hierarchy and structure of the model, and a tree graph showing the dependency and relationship of different nodes. The intermediate representation yaml file is one of the inputs for the next step, the Python scripts. In general, ZOTI-Graph provides a generic, target-agnostic, and language-independent format to describe the application model, and acts as the bridge between the system model and its implementation.

The basic structure of the input specification is similar to a Synchronous Data Flow (SDF), which contains both edges and nodes. There are four kinds of nodes in the input specification, including platform nodes, actor nodes, kernel nodes, and skeleton nodes. A platform node refers to the computation platform, which is essential to determining the grammar and pattern, including C and VHDL in this project. An actor node is a behavior computation unit, which is similar to an actor in actor models, or a function in C. A skeleton node represents some commonly used and pre-defined functions. A kernel node is a leaf computation node, which contains the function in the target language.

The starting point of a ZOTI-Graph specification is a top node, which is a root for parsing. The format for the basic unit node is shown in the piece of code below. A node should first have basic information like name and kind. A node should also contain ports, the input-output interfaces connecting outside with the node, edges, to connect ports of different nodes for data communication, and nodes, representing the sub-nodes inside. Different kinds of nodes also have different characteristics, like extern in a kernel node.

```yaml
name: example
kind: ActorNode
ports:
- name: inin
  kind: input
- ...
edges:
- connect: [node_name0, port_name0, node_name1, port_name1]
```
2.9.3 ZOTI-FTN

ZOTI-FTN is used to describe common data types and generate glue codes in the target language. The output of ZOTI-FTN is the other input of the Python scripts. The current version of ZOTI-FTN is only in the early experimental stage and not well-defined, so it is not used in this master thesis. In this master thesis, only a simplified version of type system is implemented. Manual modification after code generation has to be used.

2.9.4 ZOTI-Gen

ZOTI-Gen is used to generate the executable code based on template programming. The input to it is a yaml file describing the structure of the system, called ZOTI-Gen specification, and some pre-defined template codes written in target languages. The general idea of ZOTI-Gen is template programming, which is a way of reusing code. Libraries of templates are introduced in ZOTI-Gen for some general operations. This is because some of the low-level codes are similar except for the changes of some parameters and variables, like for loops.

The general structure of ZOTI-Gen is a block list. The basic format is a block, shown in the piece of code below. Different from ZOTI-Graph specification, where we can simply include a node inside another node, in ZOTI-Graph specification, we can only use an instance of a block inside another block after initialization. This is similar to function calls in C. A block first contains name for distinguishment. type is used for the template of the block, which is the pre-defined codes. Inside param, we can decide some constant variables and schedule, which is the order of the inner function calls. label is used to define variables inside the function block, including inputs, outputs, and intermediates. prototype can define the general structure of the function. requirement defines the required header files. instance is a list referring to schedule. placeholder refers to the item in schedule. block refers to the original block defined elsewhere. bind defines the relationship between the variables inside the parent function block and the argument in the child block.
2.10 Related Work

There are generally 2 approaches as related work to solve the synthesis problem, including synthesis flows and Model-to-Text (M2T).

2.10.1 Synthesis Flows

The first approach is synthesis flows, which can transform high-level modelling languages to target low-level codes. The high-level modelling languages are usually embedded in some simpler languages compared with the target languages and generate some specific codes after compilation. The transformation workflow is fixed, which means we can not customize our own version.

2.10.1.1 ForSyDe Deep

ForSyDe is a design methodology for embedded systems based on formal principles developed by KTH [15]. Within this framework, a number of modelling languages have been developed for different paradigms. ForSyDe Deep is the deep embedding one for the hardware, which means it explicitly represents the grammar of a language in the host logic and provides a semantic function for interpreting the meaning [16]. ForSyDe Deep is a high-level modelling language embedded in Haskell, which can be synthesized into VHDL codes.
There are generally three layers of ForSyDe Deep, including process function, process, and system. The system is a package of a top-level process, while a top-level process is a combination of several processes. A process is specified from a process function. Like hardware description languages, every component, here referring to processes and systems, needs to have a unique name.

A process function is a function which expresses the functionality of a process. A process is the basic unit of the code. Processes can have hierarchies, which means a process can be composed of several other processes. No matter how many processes there are, there can only exist one top-level process, because a system can only be defined based on one process. Different processes communicate through signals. A system refers to an entity in VHDL. Here is a piece example code of ForSyDe Deep.

```haskell
mulProcFun :: ProcFun (Int32 -> Int32 -> Int32)
mulProcFun = $(newProcFun
  [d| mulProcFun :: Int32 -> Int32 -> Int32
    mulProcFun x y = x * y
  ])

mulProc :: Signal Int32 -> Signal Int32 -> Signal Int32
mulProc = zipWithSY "mulProc" mulProcFun

mulSysDef :: SysDef (Signal Int32 -> Signal Int32 -> Signal Int32)
mulSysDef = newSysDef mulProc "mulSys" ["i1", "i2"] ["out"]
```

ForSyDe Deep has some skeletons. Skeletons are built-in functions. With skeletons, it is convenient to deal with some common patterns, like the constructions of Moore and Mealy finite state machines. Other skeletons are variations of pre-existing functions in Haskell, mainly focusing on the vector or list operations. ForSyDe Deep is easy to test. Users can check whether the result is correct by running the system in the command line instead of writing a test bench.

There are also some cons about ForSyDe Deep. The construction of an entity is quite tedious. We need first to define the type of the element before using them. What’s more, different from Chisel, ForSyDe Deep is a research tool. There are not enough users for ForSyDe Deep. Thus, there are some inconvenient functions and internal bugs about it.
2.10.1.2 Chisel

Chisel is a hardware construction language developed by UC Berkeley, which is now put into industry [17]. It is a high-level programming language embedded in Scala. Chisel combines object-oriented programming and functional programming, which makes the coding process easier compared with normal hardware description languages. Different from high-level synthesis, which is synthesized from the C/C++ level, Chisel is still programmed at the register-transfer level and can be translated to Verilog.

The basic unit of Chisel is **Component**, which is similar to **module** in Verilog and **entity** in VHDL. The component is composed of two parts, the interface and the behavior. Interface is the input-output ports, while the behavior is the body of the component, which is to describe the function. Interface is represented by **Bundle**, which is a struct-like data structure. The component is a class, where ports are the attributes, and the functions are the methods. The general structure is also very similar to the hardware description language. For the sequential part, Chisel contains an implicit global clock, which doesn’t need to be specified manually. Below is an example of Chisel code.

```scala
class Mux2 extends Component {
  val io = new Bundle {
    val sel = Bits(1, INPUT)
    val in0 = Bits(1, INPUT)
    val in1 = Bits(1, INPUT)
    val out = Bits(1, OUTPUT)
  }
  io.out := (io.sel & io.in1) | (~io.sel & io.in0)
}
```

The biggest advantage of Chisel is its reusability. The reusability of Chisel is built upon inheritance and parameterization. The inheritance is mainly illustrated by the struct-like structure of **Bundle**. Different from Verilog or VHDL, whose ports need to be specified one by one, ports of Chisel can be specified by inheritance with the help of **extends**. This can save a lot of time in writing repetitive ports if some of the components originate from one prototype, and also decrease the possibility of making mistakes. Parameterization means if we have two very similar functions or processes with different data types, we don’t need to write them twice. Instead, we can write a father function with no data type specified.

For Chisel, a C++ simulator is used for RTL debugging. The speed of this simulator is much faster than the simulator of the conventional software
There are also some disadvantages of Chisel. The first is that in Scala we cannot define some new tokens, which means no intermediate variable is allowed. The second one is about the error message. Now no useful information on the exact line where the error occurs can be provided.

2.10.1.3 SkePU

SkePU is a high-level programming approach developed by Linköping University. It is a C++ template library which provides interfaces for data-parallel computations [19]. The interfaces are skeletons, which can be used to write the applications. Different from other related works discussed above, which are auxiliary in the entire process of programming, SkePU only focuses on the skeletons. For the rest of the code, it uses C++. A piece of SkePU code is shown below.

```c++
template<typename T>
T mvmult_f(skepu::Index1D row, const skepu::Mat<T> m, const skepu::Vec<T> v) {
    T res = 0;
    for (size_t j = 0; j < v.size; ++j)
        res += m(row.i, j) * v(j);
    return res;
}

skepu::Vector<float> y(height), x(width);
skepu::Matrix<float> A(height, width);
auto mvmult = skepu::Map(mvmult_f);
mvmult(y, A, x);
```

SkePU has multiple backends. Each skeleton can be synthesized to adapt to different architectures, including sequential C, openMP, openCL, and so on. SkePU can also select the fastest implementation towards each skeleton [20]. Element access in SkePU is also more flexible and smarter. SkePU uses proxy containers to achieve complex data access, like neighborhood access. Since SkePU is based on C++, it also inherits some of the advantages of C++, like template and standard data type. Templates can increase the code reusability, and standard data type can save us a lot of time.

Since SkePU is based on C++, we still need to pay some attention to the implementation details. The grammar is still not that easy. SkePU is more like another sdt container instead of a package fully used for modelling.
2.10.1.4 FeldSpar

FeldSpar is a domain-specific language based on Haskell. It can synthesize functional descriptions into imperative C codes [21].

The design of FeldSpar is to combine deep embedding and shallow embedding together. The goal is to combine the modularity and extensibility of shallow embedding and the high-performance code generation of deep embedding. Below is a piece of FeldSpar code.

```haskell
f :: Data Int32 -> Data Int32
f i = (testBit i 0) ? (2i, i)
```

Feldspar program is first written in a high-level algorithm, which can then be evaluated to Feldspar’s low-level core language. Users can optimize the original code by checking the core code. Then the C code generator can generate the C code of different vendors based on the core code. Figure 2.11 shows the general structure of the compilation process.

![Figure 2.11: Feldspar compiler pipeline.](image)

Since Feldspar is an embedded language in Haskell, the language constructs are given as ordinary Haskell functions. Those functions only result in the data structure of the core language program. Other functions can have some semantics and be compiled into the core program.
2.10.1.5 Bluespec

Bluespec is a high-level hardware description language [22]. The input specification model is written in Bluespec SystemVerilog (BSV), which is a .bsv file. The BSV compiler can generate C++ and SystemC code for simulation, and Verilog code for hardware synthesis. Different from other high-level synthesis approaches which hide the complexity of hardware, BSV exposes it to users “as an intuitive high-level metaphor.” [23] However, BSV tries to hide some tedious details of traditional hardware design, including synchronization signals and poor typing systems [23].

The computation model of BSV is defined by atomic guarded rules, which is successful in describing concurrent systems. The atomic guarded rules can be formulated as

\[ pat_{lhs} \text{ if } p \rightarrow exp_{rhs}, \]

which means if the current condition matches a pattern, a new expression will come to replace the terms with new ones. Here terms are similar to variables stored in registers, which are states of the system. Each clock cycle, the rules whose conditions are valid are executed to update the state of the hardware system [23].

The programming paradigm of BSV is similar to object-oriented programming. First, we need to define an interface, where methods to call are initialized here. The specification of interface is defined in module. Inside a module, we need to define the rules for updating, terms representing states, and specifications for methods defined in interface. There are two kinds of data types in BSV, primary data types and data structures. Programmers can also define their own data types and impose some characteristics on them, which can achieve polymorphism.

A piece of code for computing the greatest common divisor of two numbers is shown below, which is an example code from [23].

```plaintext
interface GCD;
  method Action start(int newa, int newb);
  method int result;
endinterface

module mkGCD (GCD);
  Reg#(int) a <- mkRegU;
  Reg#(int) b <- mkReg(0);
  rule gcd_subtract ( a >= b && b != 0 );
    a <= a - b;
endrule
```
The BSV compiler is shown in Figure 2.12. The input to the compiler is bsv program. The compiler first does pre-elaboration for parsing and type checking, and then does post-elaboration for code generation [24]. The compilation result is either linked into a simulation environment or processed by a synthesis tool [24].
2.10.2 Model-to-Text Approaches

The second approach is M2T, which can transform a model, like a Unified Modelling Language (UML) or Systems Modelling Language (SysML) to any text, including low-level codes. Compared with synthesis flows, M2T approaches are more flexible. Users can customize the templates so that the generation can be anything they want.

2.10.2.1 Papyrus

Papyrus is a general-purpose UML2 graphical modeller based on the Eclipse environment, which also provides code generation in C, C++, and Java, and “facilitates external tools connection to enable models to be the driving artefacts of the development process.” [25] Papyrus is also a powerful tool
for “designing Domain Specific Modelling Languages using the UML profile concept” and “comes with a set of pre-defined extension plug-ins devoted to real-time embedded applications.” [25] Papyrus can also be customized in some parts to solve all kinds of problems, including model explorer, diagram notation and style and so on [26]. Papyrus also supports model-based techniques, including model-based formal testing, safety analysis and so on [26].

2.10.2.2 Acceleo

Acceleo is a template-based technology to create custom code generators [27]. Acceleo receives a model and a template as inputs and generates codes as outputs. Acceleo can receive any kind of Eclipse Modeling Framework (EMF) models, including UML models, Ecore models, and so on [28]. The model can also be generated by any kind of tools based on EMF. Acceleo can also generate any output codes as long as we can write them. We can reuse an existing module and change its original behavior with static override, and change the behavior of an existing generator by defining templates which can replace existing templates with dynamic override [28]. Incremental generation can help to protect changes by defining a protected area with a default piece of code which can be customized by users [28].

2.10.2.3 Xpand

Xpand is a language specialized in code generation based on EMF models [29]. Xpand supports Pluggable Type System, Dynamic Dispatch of Functions and Built-in support for Aspect-Oriented Programming (AOP) Rich Expressions [29]. It can also generate all kinds of languages. The general step is to use Xpand to define a meta-model, which is the structure of the input model to be processed, define templates to tell the code generator how to process the model, create an input model, and start the code generator [30].

2.10.2.4 Xtend

Xtend is “a flexible and expressive dialect of Java, which compiles into readable Java 8 compatible source code.” [31] Users can “use any existing Java library seamlessly.” [31] The generated output is pretty-printed and can run as fast as the hand-written Java code. Xtend provides an IDE with full debugging support, full Java integration, readable syntax and powerful indentation
control [32]. Xtend also provides dispatch methods for overloading based on the dynamic type of the created model [32].

2.10.2.5 Velocity

Velocity is a Java-based template engine, which permits users to use a simple but powerful template language to reference objects defined in Java code [33]. Although Velocity is designed mainly for web applications, it can also be used in M2T domain to generate source codes. Its engine has the flexibility to receive any format of inputs and generate any format of outputs. Velocity is built around 3 structures, including a context, a template, and an engine [34]. The context contains the framework for well-formatted input data which can be processed. The template is written in Velocity Template Language (VTL), which can grab information from the context and generate the output. The velocity engine takes the information from the context and the template and generates the output texts, which is like a function. The job of Velocity engine can be formulated as:

\[ C \times V \rightarrow T, \]

where \( C \) refers to the set of Velocity texts, \( V \) refers to the set of Velocity templates, and \( T \) refers to a set of output texts generated from the Velocity engine.

2.10.3 Relation with This Thesis

Two kinds of related work are presented above, including synthesis flows and M2T approaches. Synthesis flows refer to the methods of transforming a high-level language into a low-level language, called backend. Most of them have only one backend. The programming languages of synthesis flows can generate either hardware or software programming languages, not both of them. M2T approaches refer to the methods of transforming an input model to any text based on templates. This method is more flexible. Both of the methods are end-to-end transformations. The input is a human-written model, and the output is low-level code files.

This project has similarities with the related work. It is a synthesis flow from a high-level language synthesizing to low-level languages. This synthesis flow receives a yaml file describing the model, and generates C and VHDL files as outputs. In principle, the synthesis flow can generate any text. Thus, the synthesis flow in this project is more similar to M2T approaches. But the input to this synthesis flow in the project is not a human-written code or model, but
an intermediate representation generated from the front end. Compared to the synthesis flows and M2T approaches, this project is like the second half of the work. In the first half, a parser scans the high-level modelling language and generates an intermediate representation in yaml. In the second half, which is the work of the master thesis, the intermediate representation is transformed into multiple back ends.
38 | Theoretical Background
Chapter 3

Methods

The purpose of this chapter is to provide an overview of the research method used in this thesis. Section 3.1 describes the research process. Section 3.2 describes the experimental design. Section 3.3 focuses on the data collection techniques used for this research. Section 3.4 explains the techniques used to evaluate the reliability and validity of the project. Section 3.5 describes the hardware and software resources used in this thesis.

3.1 Research Process

The entire workflow is to transform a target-agnostic model specification written in yaml into low-level codes, including C and VHDL in this master thesis, with ZOTI. Considering the complexity of the type system, type handling is omitted. ZOTI-FTN is also not used. The workflow is shown in Figure 3.1. The input file written in yaml is first processed in ZOTI-Graph, then Python scripts, and finally ZOTI-Gen to generate C/VHDL. The library storing templates and skeletons is not shown in Figure 3.1 for simplicity. This is a top-down workflow. Models become are processed from abstraction to specification. But in the real implementation, the bottom-up methodology is adopted, which starts from low-level codes, then ZOTI-Gen specifications, and finally ZOTI-Graph specifications. This methodology conforms to the general regularity that people tend to start from familiar and easy parts.
The first step is pre-study, including software and hardware resources, related code synthesis methods, and system design methods. The software resource includes Vivado, Vitis, and ZOTI. The hardware resource includes PYNQ boards. This step is the preparation for the required knowledge to start.

The second step is to build the system with 2 components of the ForSyDe AESA-Radar [4], including Doppler Filter Bank (DFB) and Constant False Alarm Rate (CFAR) by hand. C code files with Vitis and VHDL code files are written with Vivado, both by hand, to achieve communication between a heterogeneous platform on one PYNQ board. Those code files are the target of the expected generated result.

The third step is to formulate a synthesis flow with ZOTI based on Genny methodology [14], which maps high-level code to C or VHDL based on the requirements. During this time, ZOTI may need to be improved by modifying the source code. This is the main work of this thesis project.
The fourth step is to demonstrate the synthesis flow with a model specification describing the heterogeneous system with DFB and CFAR by transforming the yaml input into C and VHDL codes. This step is for the evaluation of the feasibility of the method with respect to the hypothesis.

3.2 Experimental Design

In this section, the platform used in this project, the expected outcomes and deliverables, and the two experiments for illustration are shown. Experiment 1 is manual work for low-level programming. Experiment 2 is the main work of this master thesis by automating the entire process. Apart from ZOTI, Vivado, and Vitis, the previous work [35] finished in the ForSyDe Radar project of the course Embedded Project Design, IL2232, is also used as an reference.

3.2.1 Platform

The platform used in this project is a PYNQ-z2 board provided by Ericsson. The PYNQ-z2 board contains a ZYNQ core for data processing.

The ZYNQ 7000 SoC family integrates the software programmability of an ARM®-based processor with the hardware programmability of an FPGA. This means on a single board, we can do both hardware programming and software programming and make them communicate with each other. The software part is called Processing System (PS), while the hardware part is called Programmable Logic (PL). PS contains several functional units, including Application Processor Unit (APU), Memory Interfaces, I/O peripherals (IOP) and Interconnect. PS and PL can communicate through Advanced eXtensible Interface (AXI) ports.

For data communication between PS and PL, there are a lot of methods. The simplest method is to use AXI General Purpose Input/Output (GPIO), which is suitable for a small amount of data. Considering the data used in this master thesis is quite huge, Direct Memory Access (DMA) is used for data transmission. Instead of data flow from memory to PS then to the AXI interface, DMA transmits data through stream directly from memory to AXI interface without going through PS and reminding the Arm core with interrupts. This can be faster and more efficient.
3.2.2 Previous Work

For experiment 1, manual coding, the previous work from [35] is used. This GitHub repository is the implementation of the parts of the Active Electronically Scanned Array (AESA) radar, including Corner Turn (CT) and CFAR. The software code is written in C. Instead of writing the code directly, the skeletons are also implemented for simplicity. The software codes are written with the help of skeletons. The hardware code is written in ForSyDe Deep which can generate the VHDL codes, but the generated codes still require some modification. Since the hardware code contains fixed-size vectors, it is much less flexible compared with the software part. The idea is to use as much as possible work from the previous work.

3.2.3 Expected Outcomes

The expected outcome for illustration is a kind of distributed system with both software and hardware. One PYNQ-z2 board is used as the basic platform. On this PYNQ board, the software code C in PS and the hardware code VHDL in PL are deployed. The two parts can communicate with each other through DMA. The logic is to first fetch the data in PS, then transmit to PL for hardware computation, and finally transmit it back to PS for software computation and printing on the screen. Figure 3.2 shows a paradigm to illustrate the design.

![Figure 3.2: PYNQ-z2 board internal-communication.](image)

3.2.4 Experiment 1

Experiment 1 is the manual coding in C for PS and VHDL for PL. Since the idea is to first do the computation on PL and then to do the computation on
PS, the output of the hardware code should be the input of the software code. The previous work is on CT and CFAR, which are not continuous. Thus, the plan is to implement DFB in VHDL and CFAR in C.

Figure 3.3 shows the design flow for manual experiment. The general process is to first construct the block design for internal communication with DMA between PS and PL through Vivado, which uses DMA as the AXI interface. The VHDL code for DFB is then written. The block design constructed in Vivado and the written VHDL files can be synthesized to hardware files. The C code for CFAR in the previous work is reused and compiled to software files. Finally, the hardware files and software files are deployed on the PYNQ-z2 board.

![Figure 3.3: Manual design flow.](image)

3.2.5 Experiment 2

Experiment 2 is to automate the entire process from high-level specification to low-level implementation based on template programming. To implement this, some templates are constructed to be used in the process and the Python script is written for transformation from ZOTI-Graph to ZOTI-Gen. For the generated code files, some manual modification are needed so that they are able to compile. Figure 3.4 shows the workflow.

The original plan also includes the generation of the Tcl scripts. Tcl scripts are used to generate the block design, run the synthesis, and transmit the executable files to the PYNQ board. But because of the time constraints, this part is discarded.

### 3.3 Data Collection

Synthetic data is used in this project. This is because the focus of this master thesis is the synthesis flow. The output of the implementation is the generated code files written in C and VHDL. The verification method is the comparison between the generated code and the hand-written code, which can be checked by eye. Data is used to verify the correctness of the hand-written code, which is to ensure that the hand-written code can be compiled and run on the PYNQ board. The only requirement for data is that their absolute values must be smaller than 1 because the fast Fourier transform (FFT) IP used in this project receives fixed point numbers with Q1.15 format, whose range is from -1 to 1. What’s more, the input matrix for CFAR should be positive for all its entries, so an absolute value operation for the output of DFB is needed.

### 3.4 Validity of Methods

The validity can be measured through correctness of the generated code. Since the outcome is a semi-automated code generation tool, running the codes directly on the PYNQ board is not possible. The way to verify the generated code is to compare it with the original hand-written code by eye.
3.5 Hardware/Software Used

In this master thesis, Vivado, Vitis, and tools from ZOTI suite (mainly ZOTI-Graph and ZOTI-Gen) are used for software resources, and a PYNQ-z2 board is used for the hardware resource.

3.5.1 Vivado

Vivado is an IDE for hardware design. It is used for generating the hardware platform and hardware code. The hardware code is written by ourselves, but the hardware platform requires us to click and drag the IPs to generate. Vivado can also be used to generate our own IPs for reusability. After the completion of the hardware platform and hardware code, Vivado can be used to do the synthesis, implementation, and writing of the bitstream. The outputs of the Vivado, including bitstream and an xsa file, are used in Vitis.

3.5.2 Vitis

Vitis is an IDE for software design. It is used for writing the software code running in APU of PS. To write on the APU, we need to import the platform, the xsa file, which is exported from Vivado. Instead of the true C/C++ codes, Vitis also needs to write the interface code to control the IPs as well as the interface ports. After running the bitstream to the FPGA and running the software code on the hardware, we can check the printed result on the screen.

3.5.3 Tcl

Vivado and Vitis can also be executed totally from the command line with Tcl files. Tcl file is a kind of executable file used for Vivado and Vitis. With the Tcl file, we can finish the hardware platform to run the code without clicking the mouse and connecting the lines and run the codes on the boards, like a makefile. Code generation for Tcl files is not part of this master thesis. It is left as future work.
Chapter 4

Implementation

In this chapter, the implementation methods of the Python scripts for transmitting ZOTI-Graph specification to ZOTI-Gen specification are presented. The general structure is introduced first and then the details about it are explained.

4.1 General Structure

The total workflow of this master thesis is to transform a graph structure of a heterogeneous system into low-level executable codes, here including C and VHDL. Since ZOTI-Gen is already able to transform the intermediate representation, a yaml file in ZOTI-Gen representation, into low-level codes, the focus is put on transformation from ZOTI-Graph representation to ZOTI-Gen representation.

The general structure of a ZOTI-Graph representation is a tree structure, which means one node contains some other nodes. There is no difference between each node. The general structure of a ZOTI-Gen representation is an array of function blocks. Each function block is a two-level structure. This is similar to functions in C. A function can call other functions which are initialized before. Different from ZOTI-Graph representations, whose nodes at each level all have the same formats and structures, the two levels of ZOTI-Gen representations are different. The top level is the function prototype to specify the contents, called block, while the bottom level is the usage of other function calls, called placeholder. The demonstration of the transformation is shown in 4.1.
Generally the transformation can be decomposed into 2 dimensions. In the horizontal dimension, the algorithm can be divided into C and VHDL parts. Both of the parts are similar to each other. In the vertical dimension, the algorithm traverses all the nodes in the ZOTI-Graph specification from top to bottom. The pseudo algorithm of the main structure of the Python script is shown in Algorithm 1, which shows the general structure of the workflow.
Algorithm 1 General structure.

```plaintext
determine the platform_node_list
for plat in platform_node_list do
    block_list ← []
    parent_node_list ← [plat]
    while parent_node_list is not empty do
        pp ← parent_node_list.front
        parent_node_list.pop()
        add child nodes of pp to parent_node_list
        initialize temp as an entry for block_list
        determine name, type, prototype, and requirement of temp
        according to its node type
        transform ports of pp into label
        ag ← node_projection(pp)
        traverse all the nodes of ag and add them into instance
        traverse all the edges of ag to determine binding and to add
        intermediate labels
        determine the schedule
        add temp to block_list
    end while
    dump the block_list to a yaml file
end for
```

The first step is to determine the platform node list. This step determines
the language for the low-level codes. Since platform nodes only appear on
the top of the node trees, this step can be solved by simply traversing the first
layer of the ZOTI-Graph specification. Then for each platform node, we do
the breadth-first search from the platform node, down to all the sub-nodes.
Each sub-node is transformed into a block in ZOTI-Gen. For each node, we
first construct a dictionary skeleton for it, which is shown in the piece of code
below. temp shows the two-level structure of each function block in the ZOTI-
Gen specification. name, type, label, prototype, and requirement show the
information of parent node, while schedule and instance show the information
of its child nodes. Or more specifically, temp is the block specification of a
particular function, while each item in instance is a function call or usage of
another function block.

```javascript
temp = {
    "name": parent_node_id.name(),
    "type": {
        "module": "",
```
For each node, we only need to take care of its own information, and its children exactly under its own level. Children’s children are not needed to take care of. Instead, looking deeper for one level is enough. For simplicity, we call the current node parent node and its sub-nodes child nodes. Based on the information of the parent node, its basic information like `name`, `type`, `prototype`, and `requirement` can be determined. Here `name` refers to the block name, which is also the function name in the generated code. `type` refers to the template to use. The most common template is the generic template, which is to execute all the instances by order. `type` is not mandatory. Skeleton nodes and kernel nodes don’t have `type`. `prototype` refers to the function specification, which is usually composed of function definition and function implementation. Function definition refers to the package of a function, including its name, inputs, and outputs. Function implementation refers to how to achieve the functionality, which is the body part. Different nodes need different strategies for `prototype`, which will be discussed in 4.2. `requirement` refers to the header files needed for the functions. There is another entry called `code`, which is not shown in `temp`, because it is only used in leaf nodes. After the initialization of `temp`, we can transform its ports as `labels`, which are usually one-to-one mapping.

To deal with children of the parent node, we need to use the API function `node-projection`, which can transform the child nodes and their edges to a multi-graph in `networkx` format, so that the Python package `networkx` can be used for graph transformation. A multi-graph is a graph with nodes which can be connected with multiple directed edges. Each edge is represented as a triple, containing source, destination, and index. The multi-graph is composed of the ports of the parent node and its child nodes, with edges connected with each other. As shown in 4.2, `node-projection` can transform the structure into a multi-graph which can be processed with `networkx`. The following sections use the multi-graph as the analysis object.
4.2 Node

ZOTI-Graph has 4 different node types, including platform nodes, actor nodes, skeleton nodes, and kernel nodes. Platform nodes only appear in the first layer. The other 3 nodes are children or sub-children of platform nodes. Actor nodes are normal specific functions, whose children can be actor nodes, skeleton nodes, or kernel nodes. Skeleton nodes are pre-defined programming constructs, which can be used directly except for \texttt{farm} operation. Kernel nodes are functions written in target programming languages. The codes are written in a dfc file to be imported as a string. The leaf nodes can only be skeletons nodes or kernel nodes.

Thus, there are 4 cases to deal with. The key difference between the 4 nodes is their prototypes. For platform nodes and actor nodes, they need to specify their prototypes according to a specific pattern. For C, the prototype of an actor node is a void function, while the prototype of a platform node is a main function. For VHDL, both of them are entity + architecture. An example prototype is shown below. For the skeleton nodes, \texttt{prototype} and \texttt{code} are both set to be an empty string. Considering the reusability of skeletons, we
should also avoid multiple definitions of a skeleton in ZOTI-Gen. For kernel nodes, prototype is set to be `{placeholder.code}`, while code is a string imported from a dfc file.

```c
int main(){
    int i = 0;
    {{placeholder.code}}
}
```

### 4.3 Port

In most cases, ports in ZOTI-Graph can be transformed into labels directly in ZOTI-Gen. There are 3 kinds of port types specified in ZOTI-Graph specification, including default, const, and buffer. For default and const, they are combinational variables, which means once their values are generated, they will not change as long as the inputs don’t change. There is no time element involved. For buffer, it is sequential, which means data come one by one according to the clock. Thus, the synchronization problem needs to be considered. buffer is mainly used in the VHDL part as a stream for data transmission. A clock label and a reset label are needed for synchronization and initialization. A master-slave interface is also used, which means some extra labels, like last, ready, and valid are added. For a normal node with ports of buffer type, although there are only two explicit ports in ZOTI-Graph specification, we need to initialize 10 labels in ZOTI-Gen specification, including aclk, aresetn, s_axis_data_tdata, s_axis_data_tvalid, s_axis_data_tready, s_axis_data_tlast, m_axis_data_tdata, m_axis_data_tvalid, m_axis_data_tready, and m_axis_data_tlast. Those three ports are discussed further in 4.6.

In addition to labels corresponding to input and output ports in ZOTI-Graph, some intermediate labels are also needed. These ports are not explicitly shown in the ZOTI-Graph specification, but they are necessary so that parent labels can be mapped to child labels. This step is explained in detail in 4.5.

### 4.4 Schedule

Schedule refers to the order of functions. For software codes like C, the order of function execution is important, while for VHDL, it is not. The general principle is that an actor, which refers to a code block in ZOTI-Gen or a node in ZOTI-Graph, can only be activated or triggered if all of its inputs are ready.
Thus, at the beginning, only the input ports of the parent node are ready. We need to traverse all the edges to find those which are connected to the ready ports. Once the source ports are ready, the destination ports are also ready. Ready ports are input ports of the parent node in the beginning. For those actors whose input ports are all ready, they can be triggered and generate the outputs. Their outputs are then regarded as ready, which is similar to the input ports in the first iteration. We keep doing this until all the output ports of the parent node are ready, which ensures every actor is triggered. Finally we can get a schedule for function calls inside a parent prototype.

The pseudo-algorithm can be shown in Algorithm 2.
Algorithm 2 Determine the schedule.

\begin{algorithm}
\begin{algorithmic}
\State $\text{schedule} \leftarrow []$
\State $\text{node\_dict} \leftarrow [\text{nodes}]$
\State $\text{start\_port} \leftarrow [\text{input\_ports}]$
\While {$\text{node\_dict}$ is not empty}
\State $\text{visited\_edges} \leftarrow []$
\For {$s$ in $\text{start\_port}$}
\For {$e$ in $\text{ag\_edges}$}
\If {source of $e$ is $s$}
\State add $e$ to $\text{visited\_edges}$
\EndIf
\EndFor
\EndFor
\State $\text{start\_port} \leftarrow []$
\For {$\text{ve}$ in $\text{visited\_edges}$}
\If {destination of $\text{ve}$ is not a port}
\State $\text{ve\_destination\_ports} \leftarrow \text{ready}$
\EndIf
\EndFor
\State $\text{for\_delete} \leftarrow []$
\For {$n$ in $\text{node\_dict}$}
\If {all the input ports of $\text{ve}$ are ready}
\State add $n$ to $\text{for\_delete}$
\State add $n$ to $\text{schedule}$
\State add $n\_inputs$ to $\text{start\_ports}$
\EndIf
\EndFor
\For {$n$ in $\text{node\_dict}$}
\If {all the input ports of $\text{ve}$ are ready}
\State delete $n$ from $\text{node\_dict}$
\State add $n$ to $\text{schedule}$
\State add $n\_outputs$ to $\text{start\_ports}$
\EndIf
\EndFor
\EndWhile
\end{algorithmic}
\end{algorithm}

4.5 Binding

Binding refers to the relationship between the labels of the parent and the labels of its children. As to ZOTI-Gen specification, the labels of children
should be mapped with labels of parents, which is similar to function calls in C. Some of the variables come from the input and output ports of the parent node, while others are not explicitly shown in the ZOTI-Graph specification. This means some intermediate labels need to be added to the parent node in the ZOTI-Gen specification. As shown in Figure 4.3, the top shows the ZOTI-Graph visualization, and the bottom shows the ZOTI-Gen visualization. This is a transformation from the ZOTI-Graph specification to the ZOTI-Gen specification. In the ZOTI-Graph representation, rounded rectangles refer to nodes, squares refer to ports, and arrows refer to edges. In the ZOTI-Gen representation, rectangles refer to function blocks or instances, squares refer to labels, and arrows refer to binding between parent labels and child labels. This example is a parent node with two child nodes inside. Each node has two ports. In the ZOTI-Graph visualization, we can see that the output of n1 is connected to the input of n2, but there is no port in the parent node for binding. Thus, as shown in the ZOTI-Gen visualization, we need to add an intermediate label in the parent block for binding. We can see that except for the child ports connected with parent ports, all the other output ports need intermediate labels.

![Figure 4.3: Intermediate label.](image)

We only need to care about the output ports of all the child nodes except those which are directly connected with the parent ports. The target in this part is to add binding between the parent and children and to add intermediate variables in the parent. The general idea is to traverse all the edges of the multi-graph generated by node-projection. If the edge is connected between any port of the parent node and the child node, then no labels need to be added.
We simply add the binding between the parent node port and the child node port to temp. Otherwise, whether the source port of the edge is accessed before needs to be checked. If the source port is accessed, then we bind the parent label of the source port with the child node port. If the source port is not accessed, then we add a new label in the parent node and bind the new label with the child node port. Algorithm 3 shows the general logic.

Algorithm 3 Determine the binding.

for e in ag.edges do
    if e.destination is port then
        child_node ← e.source
        binding ← {e.destination, e.source.port}
        add binding to parent.placeholder.child_node.bind
    else if e.source is port then
        child_node ← e.destination
        binding ← {e.source, e.destination.port}
        add binding to parent.placeholder.child_node.bind
    else
        if e.source.port is not accessed then
            add new_label to parent.label
            child_node0 ← e.source
            binding0 ← {new_label, e.source.port}
            add binding0 to parent.placeholder.child_node0.bind
            child_node1 ← e.destination
            binding1 ← {new_label, e.destination.port}
            add binding1 to parent.placeholder.child_node1.bind
        else
            parent_label ← the allocation label for e.source.port
            child_node1 ← e.destination.node
            binding1 ← {parent_label, e.destination.port}
            add binding1 to parent.placeholder.child_node1.bind
        end if
    end if
end for

4.6 Type Handling

Type handling is not a trivial task. Type handling includes 2 parts, variable type and size. Variable type is easy to handle, because it is defined in ZOTI-Graph specification as data_type.type. Size is more difficult to handle. In
most cases, the size of every variable depends on the size of the inputs. So it is hard to predetermine. As discussed before, type handling is not part of the master thesis project, but for the sake of integrity, a simplified version of the type handling system with variable type and dimension is implemented, while size is omitted. This means only codes like `float matrix[][][]` instead of `float matrix[n1][n2][n3]` can be generated.

To make things simple, types and dimensions are provided in the ZOTI-Graph specification. It is obvious that two ports connected together with an edge should have the same type and the same dimension, except for the `farm` operation. `farm` operation is to apply the inner function to all the elements of the vector, which means the dimension of the inner function should be one dimension lower than the `farm` operation. But since there ports of the inner function of `farm` are connected to the parent ports. No intermediate labels are needed to add or initialize.

There are two ways to transmit data. One is by value, and the other is by pointer. For C, all data are transmitted by pointers. Thus, whenever the dimension of a port is 0, which means it is a single variable instead of an array, an `&` symbol needs to be added in front of a variable name. This also means all functions, except for the main function, are void functions. For VHDL, all the data are transmitted by value, which is easier to handle, so there is no need for further processing.

There are three kinds of port types implemented in this master project, including `default`, `const`, and `buffer`. `default` is the most commonly used situation, which is like the normal function usage by pointer. `const` refers to the case where the inputs are constant. In this case, we need to first create a new variable to the constant value and then call its pointer. `buffer` is mainly used in the VHDL part, where a stream of data is transmitted into a function block. Whenever `buffer` appears, some auxiliary labels, like last, valid, and ready need to be added to construct a master-slave interface.
Chapter 5

Result

In this chapter, we will discuss the evaluation standard for this project and give a small example of a single function to show the workflow. The input to the entire system is a ZOTI-Graph specification, and the output is a C code file and a VHDL code file. The generated ZOTI-Gen specification is also shown in this section as an intermediate result. For the details of the synthesis flows, please refer to A.1 for code generation and A.2 for hand-written codes.

5.1 Evaluation Method

The target of this master thesis project is to formulate a method for code generation. Thus, the evaluation standard is to what extent the generated code is similar to the hand-written code.

As mentioned before, the first experiment is to write the code by hand. For experiment one, the evaluation standard is whether it can run on the PYNQ-z2 board. The second experiment is to write a Python script to automate the entire process. The evaluation method is to compare the generated code with the hand-written ones. Since the type system is out of the scope of the master thesis, the generated code cannot be compiled. Thus, we compare the generated code with the original one by eye instead of directly running it.

5.2 Model

The input to the entire system is a yaml file written in ZOTI-Graph, describing the model. In this master project, two components of the AESA radar, DFB and CFAR, are deployed on the PYNQ board as a heterogeneous platform.
DFB is deployed on FPGA, and CFAR is deployed on the Arm core. The data cube is first processed by DFB, and then processed by CFAR. Its simplified version of ZOTI-Graph visualization is shown in Figure 5.1. The outermost rectangle refers to the platform node. The inner rectangle refers to any of the other three nodes. The squares refer to the ports. The arrows refer to how data flows. \texttt{data\_in} is the input, whose size is $nFFT \times nB \times nb$. \texttt{data\_in} is stored in a header file in C. Then the data cube is transposed, which changes its size to $nB \times nb \times nFFT$. The transposed result is transmitted to the FPGA part through streams. The transmitter process totally continues $nB \times nb$ times, with each time transmitting an array of size $nFFT$. After the process of DFB, the FPGA will transmit back the result to the Arm core. CFAR will start to deal with the received result until all the data have been processed by DFB, and finally generate the output.

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{figure51.png}
\caption{A simplified version of ZOTI-Graph specification as a input.}
\end{figure}

5.3 Platform

The block design of the hardware platform is shown in Figure 5.2. This is designed in Vivado by hand. The key components in the block design are ZYNQ7 Processing System, AXI Direct Memory Access, and top\_wrapper.
ZYNQ7 Processing System is the software core to run the C code, which is PS, and top_wrapper is the FPGA part to run the VHDL code, which is PL. AXI Direct Memory Access is used to transmit data between PS and PL. Once the data transmission is finished, AXI Direct Memory Access will send an interrupt to ZYNQ7 Processing System.

![Hardware platform layout](image)

Figure 5.2: Hardware platform layout.

5.4 Result

Considering the memory size of the PYNQ board, $nFFT$ is set to be 8, $nb$ is set to be 16, and $nB$ is set to be 2.

For experiment one, the result is that the code can run successfully on the PYNQ board. Since there is no data for reference, we can assume it to be correct.

For experiment two, the input is a ZOTI-Graph specification. The written Python script can transform the ZOTI-Graph specification into the ZOTI-Gen specification. Since the entire example is too big, here only `geomMean` function is taken as an example to show the entire transformation process.

The visualization of the `geomMean` function is shown in Figure 5.3. `geomMean` is the parent node on the outside, which is an actor node. Inside `geomMean`, there are two child nodes, `reduce1` and `farm_calculation`. Both of them are skeleton nodes. `reduce1` is an instance of the skeleton `reduce`, which is a leaf node, while `farm_calculation` is an instance of the skeleton `farm`, which has one child inside, `calculation`. `calculation` is a kernel node stored in a dfc file outside, which is also a leaf node. Considering the length and complexity of the transformation process, only the two-level transformation is shown here, which includes `geomMean` and its two children.
Figure 5.3: ZOTI-Graph for geomMean visualization.

The ZOTI-Graph specification is shown below. This is a tree structure from the root node `geomMean`, which means `prototype` and `instance` are combined. In most programming languages, they are separated. We must first create a `prototype` of a function so that we can use it.

There are three important parts of ZOTI-Graph specification, `ports`, `edges`, and `nodes`. `ports` define the inputs and outputs. A port should include its `name`, `kind`, `port_type` and `data_type`. `nodes` define the child nodes. `edges` act as the glue for all the components inside by connecting ports of different nodes. In the transformation process from ZOTI-Graph to ZOTI-Gen, how those three elements are transformed is important.

In line 33, the parameter for `num` is `unknown`. This is because we only know the inner kernel function `calculation` should be applied to all the elements of the array, but we don’t know its length. It is also a type system problem, which is not in the scope of the master thesis.

```plaintext
name: geomMean
kind: ActorNode
ports:
  - name: geom_in
    kind: in
    port_type: {type: default}
```
The transformed ZOTI-Gen specification is shown below, which is an intermediate result. This is block of **geomMean**, with two instances **reduce2d** and **farm_calculation**, which preserves the node structure in ZOTI-Graph specification. The function blocks of **reduce2d** and **farm_calculation** are not shown here. The structure in ZOTI-Gen separates the **prototype** and **instance** of a function, but the general structure still preserves.

For the edges, we can check **bind** of each **instance**. If there is an edge connecting the parent port and the child port, then we simply add the **label_to_label** into **bind**. If there is an edge connecting two child ports, then
a new label needs to be added first.

As said before, most of the ports can be transformed into labels directly, but we also need to determine the intermediate labels. As shown in Figure 5.3, there is no port in the parent node geom\text{Mean} corresponding reduce1\text{\_out} and cal\text{\_in}\_i, because they are connected directly. Thus, a new label needs to be created, which is intermediate1 in this example.

```
1 name: geom\text{Mean}
2 type: {module: C.Generic, name: Composite}
3 param: {schedule: [f1, f2]}
4 label:
5  - {name: geom\_in, usage: '{{label[p].name}}'}
6  - {name: geom\_out, usage: '{{label[p].name}}'}
7  - {name: intermediate1, usage: '{{label[p].name}}'}
8 prototype: void (\{name\}\{float geom\_in [\[]], float geom\_out
9 \}[]\{float intermediate1\}[]; \{placeholder.code\} )
10 instance:
11  - placeholder: f1
12  block: {module: Test, name: reduce2d}
13  bind:
14  - usage\_to\_label: {usage: addV, child: function}
15  - label\_to\_label: {parent: intermediate1, child:
16    reduce1\_out, usage: '{{label[p].name}}'}
17  - label\_to\_label: {parent: geom\_in, child: reduce1\_in,
18    usage: '{{label[p].name}}'}
19  - label\_to\_label: {parent: geom\_out, child: cal\_in\_i,
20    usage: '{{label[p].name}}'}
21  usage: '{{\{name\}}}{{\{label.function.name\}}, {{label.
22    reduce1\_in.name\}}, {{label.reduce1\_out.name\}}}'}
23  - placeholder: f2
24  block: {module: Test, name: farm\_calculation}
25  bind:
26  - label\_to\_label: {parent: intermediate1, child: cal\_in\_i,
27    usage: '{{label[p].name}}'}
28  - label\_to\_label: {parent: geom\_out, child: cal\_out\_i,
29    usage: '{{label[p].name}}'}
30  directive: [expand]
31  usage: '{{$\{placeholder.code\}$}}'
```

The generated C code is shown below. The generated code separate the geom\text{Mean} function into several different parts with comments. geom\text{Mean} itself is a function block. Inside this block, except for the skeleton function reduce, there is one function block called farm\_calculation, which is consistent with the ZOTI-Gen and ZOTI-Graph specifications. But there are still some missing parts. For the functions geom\text{Mean} and reduce2d, the sizes of variables are not provided. The length of the for loop is still unknown. The variable intermediate1 also lacks its length information.
void geomMean(float geom_in[][], float geom_out[]) {
    float intermediate1[];
    // v geomMean
    reduce2d(addV, geom_in, intermediate1);
    // v farm_calculation
    for (int _it = 0; _it < unknown; _it++) {
        calculation(&intermediate1[_it], &geom_out[_it]);
    }
    // ^ farm_calculation
    // ^ geomMean
}

The original hand-written code is shown as follows. Some modifications are made to the variable names to keep the consistency with the generated C code. Compared with the generated C code, the hand-written code includes the size information, d1 and d2, so the code is complete.

// v geommean block
void geommean(int d1, int d2, float geom_in[][d2], float geom_out[]){
    float intermediate1[d2];
    // v geommean template: Generic.Composite of various sub-blocks
    reduce2d(d1, d2, addV, geom_in, intermediate1);
    for (int _it = 0; _it < d2; _it++){
        calculation(&intermediate1[_it], &geom_out[_it]);
    }
    // ^ geommean template
}
// ^ geommean

Both of the C codes are very similar except for the dimension part. The dimension part includes a) the prototype of the function specification, b) the dimension of the array initialization, c) the function call, and d) the length of a for loop.

One important thing is that for schedule part, which is the order of function calls, there can be multiple choices. This is the same for the variable names and the order of intermediate variables. The orders and names cannot affect the functionality, so for simplicity, the order and names of the original hand-written codes are changed to make them easier for comparison. All the modifications of the hand-written codes are based on the success of compilation and functionality.
Chapter 6

Discussion and Conclusion

6.1 Discussion

The discussion part points out some important aspects of this master thesis project and mentions what can be done in the future for improvement.

6.1.1 Input Model Format

As a language, The ZOTI-Graph specification should follow some specific rules. Until now there is no formal documentation about the rules. Some experimental formats are listed as follows:

1. A port can only have one kind, either input or output.

2. If a port of a node is an input port, then this port should be the source of the edge inside the node or the destination of the edge outside the node. If a port of a node is an output port, then this port should be the destination of the edge inside the node or the source of the edge outside the node.

3. An edge can only connect ports at the same level.

4. The source and destination of an edge should belong to different nodes.

Those rules for the input model are far from enough. More rules can be added in the future.
6.1.2 Order

For sequential programming languages like C, a function can only be used after it is specified. ZOTI-Gen is clever enough to arrange the order of function specifications according to the instance. For example, if a function block A has an instance B, then in the generated code, the specification of B is put in front of that of A, even if the function block of A is put in front of the function block of B in the ZOTI-Gen specification. However, in C, functions can also be transmitted as an argument through a pointer. In this case, the transmitted function will not be part of the instance. Instead, the function name is a string in the parameter of the ZOTI-Gen specification, which is transmitted by usage_to_label. Thus, in this case, the order of the function specification can have some problems. Usually, this happens to kernel nodes.

One way to solve this problem is to separate the function definition and the function implementation. We can put the definition of a function in the beginning of the C file and put the implementation after the definition of all functions. The following examples show how to do that. This can be regarded as future work for ZOTI-Gen.

```c
void take1d(int take_n, float input_array[], float result[]);
void fanout1d(int d1, int n_times, float input_array[], float result[][d1]);
... ...
void take1d(int take_n, float input_array[], float result[]){
  ...
}
void fanout1d(int d1, int n_times, float input_array[], float result[][d1]){ ... }
```

6.1.3 Expand

There is an attribute which is not often used in ZOTI-Gen, called expand. There are two ways to execute a piece of code. One is to write the code directly in the main function and change the corresponding variable names. The other is to write it into a function and call the function when needed. In most programming languages, calling a function is preferred. For ZOTI-Gen specification, if we want to show an entire piece of code without using a function call, we need to set directive and usage like the first two lines of the code shown below. In this case, the content of a function can be expanded.
If we want to use a function call, then only usage needs to be used, which is shown in line 4.

```plaintext
directive: [expand]
usage: '{{placeholder.code}}'
usage: '{{name}}({{label.input.name}}, {{label.output.name}});
```

In this master thesis project, function calls are used in most cases except for farm operation, which corresponds to the for loop. Whenever we meet with a farm operation, this block needs to be expanded. The reason for expanding farm is that farm takes a function as an argument. It is difficult to deal with the binding relationship between labels of the inner function block with labels of the farm block.

### 6.1.4 Dimension

For many programming languages, including C and VHDL, it is important to know the length of each dimension of all input and output ports. But it is not always possible. One way to solve this problem is to include all the needed dimension information in the model specification part, which is the ZOTI-Graph specification in this project. But this can be not that easy and consumes a lot of energy for users because they need to calculate the size by themselves. Another way to solve this is to first leave the length unknown and then traverse all the nodes from bottom to top to calculate the size. The main difficult point for this method is to express the length of each port with variables, which uses the port names of the parent instead of the exact number.

### 6.1.5 Template

A template is “an abstract and generalized representation of the texture output it describes.” [11] It is a piece of code which are not complete, which means some additional parameters need to be filled in, like the length of a for loop and the inner contents. The following piece of code shows the two common examples of templates. The first one shows a for loop. We can see that the length of the for loop needs to be specified, and the operation in the for loop is a placeholder called “f1”. The second one shows the arrangement for all inner function blocks, which is to define all the placeholders, which are children in the ZOTI-Graph specification.

```plaintext
for (int {{ label._it.name }} = 0; {{ label._it.name }} < {{ param.itrange }}; {{ label._it.name }}++)
```
Template is helpful for code reusing. For similar scenarios, the same templates can be reused by replacing some variable names, which can save time for coding and decrease error rates. However, considering the huge number of programming languages and different cases, it is easy to write a huge number of templates, which causes template explosion problem.

In this master thesis project, there is no template explosion problem because this is a small project with a limited number of operations. But it can be a problem if template-based programming is used on large scales. Take farm operation as an example, which should be translated into for-loop. In this master project, only templates for one-dimension for-loop and two-dimension for-loop are written, so that each case corresponds to one category of the function block used in ZOTI-Gen, array farm and matrix farm. There can be infinite levels of iteration, so one template for one case is not a good idea. A possible solution is to construct a matrix farm block by putting an array farm block into another array farm block as an instance, which can avoid too many repetitive templates.

Another problem is how to deal with the trade-offs between fine-grained specific templates and large-grained generic templates. If the former is largely used, it can cause the template explosion problem. If the latter is commonly used, then not all cases can be covered. Users may need to add details themselves.

Considering so many programming languages, it can be quite annoying if we need to write a set of templates for each programming language. Different programming languages have similarities between them. For example, the generic template which performs showing all the instructions according to the schedule is the same for both C and VHDL in this master project, which is shown in the second piece of code written above.

Hardware and software programming languages are different. Whenever we want to call a function, we need to add a name for the function in hardware description language, while in software, there is no need to do that. Software and hardware programming languages can also have some similarities. The syntax for C and Verilog are more similar than the similarity between Verilog and VHDL. Thus, modularization may be a good idea so that different programming language can compose their needed characteristics. We
can first write some rules for templates, like different delimiters and whether
to add a name for a function. Whenever we need to do the code generation,
we can simply compose the necessary rules.

6.1.6 Skeletons

Skeletons refer to a “pre-defined, generic programming construct that
implements a common specific pattern of computation” implemented with
the target language [20]. Different from templates, which are incomplete,
skeletons are complete functions which can be used directly.

We can use a header file defining all the functions which can be directly
used, including reduce, stencil, and so on. Considering the format of C
languages, the dimension needs to be considered. The function names follow
the format like this: skeleton name + dimension. The arguments follow the
convention of dimensions, parameters, inputs, and outputs. The following
code shows an example of templates written in C.

```c
void print_array(int d1, float array[]);  
void print_matrix(int d1, int d2, float matrix[][d2]);  
void print_cube(int d1, int d2, int d3, float cube[][d2][d3]);  
void concate2d_mat(int d1_mat1, int d1_mat2, int d2, float
   input_matrix1[][d2], float input_matrix2[][d2], float
   result[][d2]);  
void concate3d_cube(int d1_cube1, int d1_cube2, int d2, int
   d3, float array1[][d2][d3], float array2[][d2][d3], float
   result[][d2][d3]);  
void take1d(int take_n, float input_array[], float result[]);  
void take3d(int d2, int d3, int take_n, float input_cube[][d2
   ][d3], float result[][d2][d3]);  
void drop1d(int d1, int drop_n, float input_array[], float
   result[]);  
void drop2d(int d1, int d2, int drop_n, float input_matrix[][d2],
   float result[][d2]);  
void fanout0d(int n_times, float * input_value, float result[]);  
void fanout1d(int d1, int n_times, float input_array[], float
   result[][d1]);  
void group2d(int d1, int d2, int num, float input_matrix[][d2],
   float result[][num][d2]);  
void reduce1d(int d1, void (*operation)(float *, float *,
   float *), float input_array[], float * result);  
void reduce2d(int d1, int d2, void (*operation)(int, float *,
   float *, float *), float input_matrix[][d2], float result[]);  
```
void stencil2d(int in_d1, int in_d2, int stencil_length,
               float a[][in_d2], float result[][stencil_length][in_d2]);

void complex2real3d(int d1, int d2, int d3, float complex
                     input[][d2][d3], float output[][d2][d3]);

void transpose3d(int d1, int d2, int d3, float complex input
                 [](d2)[d3], float complex output[](d3)[d1]);

For the VHDL part, since the code generation is relatively simpler, we only have one skeleton, called **fft**. **fft** refers to FFT, which is an operation for a list of data. To operate **FFT** in FPGA, we need to use the IP provided by Xilinx and set some parameters by hand. In this master thesis, the **FFT IP** is set to calculate 16 bits fixed point number with natural order. The initialization of an IP can be automated by tcl scripts, which can be left as future work.

To use a skeleton, a function block with the same name of the function needs to be defined, with labels of parameters, empty prototype, and empty code. Since those complete functions need to be reusable, multiple definitions of a function block in the ZOTI-Gen specification need to be avoided. This can be easily achieved by recording all the used skeletons in a dictionary.

The skeleton part is a one-to-many relationship between the ZOTI-Graph specification and ZOTI-Gen specification because we need to consider the dimension for low-level codes like C or VHDL. For example, **reduce** in the ZOTI-Graph specification doesn’t care about the dimension of the inputs, while **reduce** in C or VHDL needs to care about this. Thus, **reduce2d** or **reduce3d** are needed. In this project, every time we meet with a skeleton a C function block named skeleton name + dimension can be constructed.

For simplicity, there are also some rules for ZOTI-Graph specifications. The ports for all the skeletons with one input and one output are fixedly named by input and output. This means the function block of the skeleton in ZOTI-Gen has fixed labels for ports.

### 6.1.7 Post Process

Post process refers to the step after the generation of codes, including adding the header file and arranging the location of function specification. However, different programming languages have different formats. C only needs to add header files once, while VHDL requires header files in front of each entity. What’s more, functions in VHDL should also be put into a package. Now the post process is enough for C, but is not enough for VHDL. This can also be regarded as the future work of ZOTI.
6.1.8 Fault Detection

In this master project, we regard everything to be well-defined. Considering the complexity, only the ideal case is considered, which means all the information is given as expected. But in the real world, there can be something missing. For example, the data type of a port is not given. In this situation, some inference needs to be taken care of. There can also be something wrong, like the inconsistency between the type of two ports connected with an edge. Thus, some error messages for debugging need to be provided in the future.

6.1.9 Code Pattern

The most important part of a function block in ZOTI-Gen is prototype, which describes the structure of the function. The following two pieces of code show the prototypes of C and VHDL examples. The contents inside {{}} are dynamic, which can be changed, while others are static.

```c
void {{name}}(float md_block_input [], float md_block_output [][]){
    float intermediate1[];
    float intermediate2[][];
    {{placeholder.code}}
}
```

```vhdl
ENTITY {{name}} IS PORT :
    input.name : in std_logic_vector(31 downto 0);
    output.name : out std_logic_vector(31 downto 0);
END {{name}};

ARCHITECTURE {{name}}_arch OF {{name}} IS
    signal intermediate1 : std_logic_vector(31 downto 0);
    begin
    {{placeholder.code}}
END {{name}}_arch;
```

There are only two dynamic parts, which are name and placeholder.code. name refers to the name of the function, and placeholder.code refers to the content. Others are static parts, which will not change when compiling.

If we only consider the string itself rather than the compiling process, then the string can be classified into 2 parts. One part is general, which is the same for all function blocks, like “ENTITY name IS PORT (” and “ARCHITECTURE name_arch OF name IS”, and the other part is specific, which is different from one another, like “input.name : in std_logic_vector(31 downto 0); “. The problem is that those two parts interleaved with each other,
which makes it difficult to generate the code. We need to care about some details, like the parenthesis and semicolon, which makes the Python script complicated and error-prone.

One way to solve this is to write them as templates and upgrade ZOTI-Gen correspondingly. The target is to separate the general parts and separate parts. One proposed solution is like this.

```plaintext
1 void {{name}}({{port}}){
2   {{intermediate}}
3   {{placeholder.code}}
4 }

ENTITY {{name}} IS PORT (  
2   {{ports}}
3  END {{name}};

ARCHITECTURE {{name}}_arch OF {{name}} IS
  {{intermediate}}
begin
  {{placeholder.code}}
END {{name}}_arch;
```

By doing this, we successfully make templates for C and VHDL by separating the general and specific parts, which can simplify the efforts of making prototypes. Although this step can also be done in Python, it is better to put it in ZOTI-Gen so that the Python script only needs to care about how to extract information from the input graph to the texts to be filled.

6.2 Conclusion

In this thesis, the code synthesis for heterogeneous platforms with ZOTI has been explored. A semi-automatic synthesis method for transforming a high-level specification into low-level codes, which are C and VHDL in this project, has been successfully formulated. The entire system receives a file describing the functionality and structure written in yaml, and produces C and VHDL code files as outputs. Thus, it shows the feasibility of using the ZOTI framework to formulate a generic synthesis flow from a fully-explicit declarative specification model to low-level languages with language embedding and templates. The outcome of this master thesis can help developers to design a complex system with heterogeneity without making too many mistakes and increase efficiency.

For demonstration, two components of ForSyDe AESA Radar [4], DFB and CFAR, are used as a use case, and a PYNQ-z2 board is used as a
heterogeneous platform. The work is demonstrated by synthesizing C and 
VHDL files from a high-level specification model written in ZOTI-Graph, 
which describes DFB, CFAR and the platform structure. The synthesized 
result is a VHDL file describing DFB and a C file describing CFAR with 
hardware-dependent code for communication. After the manual modification 
of the generated codes, the VHDL code can run on PL and the C code can run 
on PS. PS and PL can communicate with each other.

As shown in 1.4, there are 5 goals in the master thesis project. The first 3 
are mandatory, and the last 2 are optional. All the 3 mandatory goals and the 
first optional one are finished, which are listed below.

1. Studying the engineering field, the theoretical concepts to be employed, 
and a brief overview of related work.

2. Designing the synthesis flow from high-level languages to VHDL and 
C with ZOTI.

3. Implementation, documentation and evaluation of the design flow.

4. Analyzing the caveats and proposing and describing an extension for the 
methodology or the tool suite.

As shown in Section 1.2.2, there are 2 engineering issues raised in the 
beginning, which are listed below:

1. How to format the semantics and structures of the input design in the 
process of synthesis.

2. How to avoid the template explosion problems.

Since this master thesis is only a small project, those 2 engineering issues are 
not met, but some intuitive answers are still given for them in Section 6.1. The 
first issue is discussed in Section 6.1.1, and the second issue is discussed in 
Section 6.1.5.

There are still a lot of things which are not implemented in this master 
thesis. They can be left as future work, which is listed below:

1. A complex type system handler is not part of the master thesis. Now 
this project only contains a simple version of the type system, which only 
considers dimensions of multi-arrays instead of sizes of multi-arrays. To 
implement a type system handler, a bottom-up traverse from the leaves 
of the input ZOTI-Graph specification is needed, while in this project, 
only a top-down traverse is implemented. ZOTI-FTN may also help in
the future. A complete type system dealing with the size of multi-arrays is left as future work.

2. ZOTI itself needs to be upgraded. Now the functionality of ZOTI is inclined to C code generation. For other programming languages, like VHDL, it doesn’t support that much. Those are mainly done in post-process. More support for other programming languages can also be done in the future.

3. In this master thesis, templates and skeletons are enough for the use case. But a lot more needs to be added for future work. This can cause the template explosion problem. So how to create templates and how to deal with the trade-offs between fine-grained templates and generic templates can be investigated. Some customer templates can also be provided for users to make some modifications to fulfil their requirements.

4. In this master thesis, only codes written in C and VHDL are generated. But for the VHDL part, FFT, an IP provided by Vivado, is used. FFT needs some manual configuration, like setting the input data types and the input array length, so that it can be created to use. This is not part of VHDL. The configuration of IP can be done with Tcl scripts, which is left as future work.
Discussion and Conclusion
References


fourth international workshop on High-level parallel programming and applications, 2010, pp. 5–14. [Page 30.]


Appendix A

Codes

There are 2 sections in this chapter, including files for synthesis flows and hand-written files. Considering the length of the code files in this project, only the important parts are presented.

A.1 Files in Synthesis Flows

Files in synthesis flows are shown in this section, including the ZOTI-Graph specification for the heterogeneous platform with CFAR and DFB, the ZOTI-Gen specifications for C and VHDL and the generated codes for C and VHDL. These can be regarded as a complementary material for Chapter 5.

The ZOTI-Graph specification of the heterogeneous platform is shown below, which is the input of the code generator system.

```yaml
# * Copyright Ericsson AB 2023 - All Rights Reserved. No part of this
# * software may be reproduced in any form without the written permission
# * of the copyright owner.

module: main
path: inputs/graph2.yaml
main-is: /nodes[Tst]
---

nodes:
  - name: Tst
    edges:
      - connect: [source, null, main, main_input]
      - connect: [main, main_output, drain, null]
      - connect: [main, transmitter, fsm, input]
      - connect: [fsm, output, main, receiver]

nodes:
  - name: main
    kind: PlatformNode
    target: {language: C, platform: zynq-arm1}
    ports:
      - name: main_input
        kind: in
        port_type: {type: default}
        data_type: {type: float complex, dimension: 3}
      - name: main_output
        kind: out
        port_type: {type: default}
        data_type: {type: float, dimension: 3}
```
```python
- name: transmitter
  kind: out
  port_type: {type: buffer}
  data_type: {type: short complex, dimension: 1}
- name: receiver
  kind: in
  port_type: {type: buffer}
  data_type: {type: short complex, dimension: 1}
  edges:
   - connect: [., main_input, transpose1, input]
   - connect: [transpose1, output, cube_transmission1, input]
   - connect: [cube_transmission1, output, complex2real1, input]
   - connect: [complex2real1, output, farm_fCFAR, CFAR_in_i]
   - connect: [farm_fCFAR, CFAR_out_i, ., main_output]
   - connect: [cube_transmission1, transmitter, ., transmitter]
   - connect: [., receiver, cube_transmission1, receiver]
  nodes:
   - name: cube_transmission1
     kind: SkeletonNode
     type: cube_transmission
     parameters: {form: expand, num: [NoB, Nob, NoFFT]}
     ports:
      - name: input
        kind: in
        port_type: {type: default}
        data_type: {type: float complex, dimension: 3}
      - name: output
        kind: out
        port_type: {type: default}
        data_type: {type: float complex, dimension: 3}
      - name: transmitter
        kind: out
        port_type: {type: buffer}
        data_type: {type: short complex, dimension: 1}
      - name: receiver
        kind: in
        port_type: {type: buffer}
        data_type: {type: short complex, dimension: 1}
      - name: complex2real1
        kind: SkeletonNode
        type: complex2real
        parameters: {dimension: 3}
        ports:
         - name: input
           kind: in
           port_type: {type: default}
           data_type: {type: float complex, dimension: 3}
         - name: output
           kind: out
           port_type: {type: default}
           data_type: {type: float complex, dimension: 3}
      - name: transpose1
        kind: SkeletonNode
        type: transpose
        parameters: {dimension: 3}
        ports:
         - name: input
           kind: in
           port_type: {type: default}
           data_type: {type: float complex, dimension: 3}
         - name: output
           kind: out
           port_type: {type: default}
           data_type: {type: float complex, dimension: 3}
      - name: farm_fCFAR
        kind: SkeletonNode
        type: farm
        parameters: {form: expand, num: NoB}
        ports:
         - name: CFAR_in_i
           kind: in
           port_type: {type: default}
           data_type: {type: float, dimension: 3}
         - name: CFAR_out_i
           kind: out
           port_type: {type: default}
           data_type: {type: float, dimension: 3}
  edges:
   - connect: [., CFAR_in_i, fCFAR, fCFAR_in]
   - connect: [fCFAR, fCFAR_out_i, ., main_output]
```

The generated ZOTI-Gen specification for C from the ZOTI-Graph specification above is shown below. Some formats are changed so that it is more readable.
```c
int {{name}}() {
  float complex main_input [][][];
  float main_output [][][];
  short complex transmitter[][];
  short complex receiver[];  
  float complex intermediate[][][];
  float complex intermediate2[][][];
  float complex intermediate3[][][];
  int {{name}}();
  return 0;
}
```

```c
{{name}}({{label.input.name}}, {{label.output.name}});
```

```c
{{placeholder.code}}
```
Appendix A: Codes

87

requirement:
  include: ["skeletons.h"]
code:
  name: transpose3d

label:
  - name: input, usage: "{{label[p].name}}"
  - name: output, usage: "{{label[p].name}}"

prototype:
requirement:
  include: ["skeletons.h"]
code:
  - name: farm_CCFAR
    type: {module: C.Skeleton, name: Farm}
    param:
      itrange: NoB
    label:
      - {name: _it, usage: "{{label[p].name}}"}
      - {name: CFAR_in_i, usage: "{{label[p].name}}"}
      - {name: CFAR_out_i, usage: "{{label[p].name}}"}
  requirement: {include: ["skeletons.h"]}
instance:
  - placeholder: f1
    block: {module: Test, name: fCFAR}
  bind:
    - label_to_label: {parent: CFAR_out_i, child: fCFAR_out, usage: "{{label[p].name}}[{{label._it.name}}]"}
    - label_to_label: {parent: CFAR_in_i, child: fCFAR_in, usage: "{{label[p].name}}[{{label._it.name}}]"}
  usage:
    {{name}}({{label.fCFAR_in.name}}, {{label.fCFAR_out.name}});
  - name: fCFAR
    type: {module: C.Generic, name: Composite}
    param:
      schedule: [f1, f3, f4, f2, f5, f6, f7, f8, f9, f10]
    label:
      - {name: fCFAR_in, usage: "{{label[p].name]}"}
      - {name: fCFAR_out, usage: "{{label[p].name]}"}
      - {name: dummy1, usage: "{{label[p].name]}"}
      - {name: intermediate1, usage: "{{label[p].name]}"}
      - {name: intermediate2, usage: "{{label[p].name]}"}
      - {name: intermediate3, usage: "{{label[p].name]}"}
      - {name: intermediate4, usage: "{{label[p].name]}"}
      - {name: intermediate5, usage: "{{label[p].name]}"}
      - {name: intermediate6, usage: "{{label[p].name]}"}
      - {name: intermediate7, usage: "{{label[p].name]}"}
      - {name: intermediate8, usage: "{{label[p].name]}"}
      - {name: intermediate9, usage: "{{label[p].name]}"}
    prototype:
      void {{name}}(float fCFAR_in [], float fCFAR_out []);
      float dummy1 = -1.3292279e36;
      float intermediate1[][][];
      float intermediate2[][][];
      float intermediate3[][][];
      float intermediate4[][];
      float intermediate5[][][];
      float intermediate6[][][];
      float intermediate7[][][];
      float intermediate8[][][];
      float intermediate9[][][];
      {
        (placeholder.code)
      }
instance:
  - placeholder: f1
    block: {module: Test, name: stencil2d}
...
The generated ZOTI-Gen specification for VHDL from the ZOTI-Graph specification above is shown below. Some formats are changed so that it is more readable.
function complex_mul (
  code: |
  prototype: |
  - name: complex_mul |
  code: |
  prototype: |
  - {name: aresetn, usage: |
  - {name: aclk, usage: |
  - {name: m_axis_data_tlast, usage: |
  - {name: m_axis_data_tready, usage: |
  - {name: m_axis_data_tvalid, usage: |
  - {name: m_axis_data_tdata, usage: |
  - {name: s_axis_data_tlast, usage: |
  - {name: s_axis_data_tready, usage: |
  - {name: s_axis_data_tvalid, usage: |
  - {name: s_axis_data_tdata, usage: |

END {{name}}_arch;

END {{name}};
The generated C code is shown below.

```c
void fCFAR(float fCFAR_in[8], float fCFAR_out[8]) {
    float dummy1 = -1.3292279e36;
    float intermediate1[8][8];
    float intermediate2[8];
    float intermediate3[8];
    float intermediate4;
    float intermediate5[8][8];
    float intermediate6[8][8];
    float intermediate7[8];
    float intermediate8[8];
    float intermediate9[8];
    // v fCFAR
...
    stencil2d(NoFFT, fCFAR_in, intermediate1);
    md_block(fCFAR_in, intermediate3);
    fanout0d(NoFFT, &dummy1, intermediate4);
    // v farm_aritMean
    for (int _it = 0; _it < unknown; _it++) {
        aritMean(intermediate1[_it], intermediate2[_it]);
    }
    // ^ farm_aritMean
    fanout1d(NoFFT + 1, intermediate4, intermediate5);
    fanout1d(2 * NoFFT, intermediate4, intermediate6);
    drop2d(2, intermediate2, intermediate7);
    concatenate2d(intermediate5, intermediate2, intermediate8);
    concatenate2d(intermediate7, intermediate6, intermediate9);
    // v matrix_farm_normCfa
    for (int _i = 0; _i < unknown; _i++) {
        for (int _j = 0; _j < unknown; _j++) {
            normCfa(&intermediate3[_i][_j], &fCFAR_in[_i][_j], &intermediate9[_i][_j],
                    &intermediate8[_i][_j], &fCFAR_out[_i][_j]);
        }
    }
    // ^ matrix_farm_normCfa
}
```
int main() {
  float complex main_input[1][1][1];
  float main_output[1][1][1];
  short complex transmitter[1];
  short complex receiver[1];
  float complex intermediate1[1][1][1];
  float intermediate2[1][1][1];
  float complex intermediate3[1][1][1];
  // v main
  transpose3d(main_input, intermediate3);
  // v cube_transmission1
  DMA_Init_Init(&AxiDma, 0);
  Init_Intr_System(&Intc);
  DMA_Setup_Intr_System(&AxiDma, TX_INTR_ID, RX_INTR_ID);
  DMA_Init_Init(&AxiDma);
  Init_Status;
  TxDone = 0;
  RxDone = 0;
  Error = 0;
  for (int _i = 0; _i < NoB; _i++) {
    for (int _j = 0; _j < Nob; _j++) {
      for (int _k = 0; _k < NoFFT; _k++) {
        transmitter[_k] = (short complex)(intermediate3[_i][_j][_k] * pow(2, 15));
        Xil_DCacheFlushRange((short complex *)transmitter, NoFFT * sizeof(short complex));
        Status = XAxiDma_SimpleTransfer(&AxiDma, (short complex *)receiver, NoFFT * sizeof(short complex), XAXIDMA_DEVICE_TO_DMA);
        if (Status != XST_SUCCESS) {
          return XST_FAILURE;
        }
        Status = XAxiDma_SimpleTransfer(&AxiDma, (short complex *)transmitter, NoFFT * sizeof(short complex), XAXIDMA_DMA_TO_DEVICE);
        if (Status != XST_SUCCESS) {
          return XST_FAILURE;
        }
        while (!TxDone || !RxDone) {
          }; TxDone = 0;
          RxDone = 0;
          if (Error) {
            return XST_FAILURE;
          }
          Xil_DCacheInvalidateRange((short complex *)receiver, NoFFT * sizeof(short complex));
          for (int _k = 0; _k < NoFFT; _k++) {
            intermediate1[_i][_j][_k] = (float)creal(receiver[_k]) / pow(2, 15) + (float)cimag(receiver[_k]) / pow(2, 15) * I;
          }
        }
      }
    }
    DMA_DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);
    // ^ cube_transmission1
    complex2real3d(intermediate1, intermediate2);
    // v farm_fCFAR
    for (int _it = 0; _it < NoB; _it++) {
      fCFAR(intermediate2[_it], main_output[_it]);
    }
    return 0;
  }
}

The generated VHDL code is shown below.
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE ieee.std_logic_unsigned.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
  PORT (aclk, aresetn, s_axis_data_tdata, s_axis_data_tvalid, s_axis_data_tready, s_axis_data_tlast, m_axis_data_tdata, m_axis_data_tvalid, m_axis_data_tready, m_axis_data_tlast);
END stream_vector_operation_32b;

ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
  SIGNAL index : INTEGER := 0;
  BEGIN
    PROCESS (aclk) BEGIN
      IF (aclk event AND aclk = '1') THEN
        m_axis_data_tvalid <= s_axis_data_tvalid;
        m_axis_data_tlast <= s_axis_data_tlast;
        s_axis_data_tready <= m_axis_data_tready;
      END IF;
    END PROCESS;

    PROCESS (aclk, aresetn) BEGIN
      IF (aresetn = '0' OR s_axis_data_tvalid = '0') THEN
        index <= -1;
      ELSIF (aclk event AND aclk = '1' AND s_axis_data_tvalid = '1') THEN
        index <= index + 1;
      ELSE
        NULL;
      END IF;
    END PROCESS;

    PROCESS (index) BEGIN
      IF index < 0 THEN
        m_axis_data_tdata <= "00000000000000000000000000000000";
      ELSIF index > 8 - 1 THEN
        m_axis_data_tdata <= "00000000000000000000000000000000";
      ELSE
        m_axis_data_tdata <= complex_mul(input1 => s_axis_data_tdata, input2 => mk_WeightCoefs(index));
      END IF;
    END PROCESS;
  END stream_vector_operation_32b_arch;

FUNCTION complex_mul( input1 : STD_LOGIC_VECTOR(31 DOWNTO 0); input2 : STD_LOGIC_VECTOR(31 DOWNTO 0)) RETURN STD_LOGIC_VECTOR IS VARIABLE result : STD_LOGIC_VECTOR(31 DOWNTO 0);
  VARIABLE m1_im : sfixed(0 DOWNTO -15) := to_sfixed(input1(31 DOWNTO 16), 0, -15);
  VARIABLE m1_re : sfixed(0 DOWNTO -15) := to_sfixed(input1(15 DOWNTO 0), 0, -15);
  VARIABLE m2_im : sfixed(0 DOWNTO -15) := to_sfixed(input2(31 DOWNTO 16), 0, -15);
  VARIABLE m2_re : sfixed(0 DOWNTO -15) := to_sfixed(input2(15 DOWNTO 0), 0, -15);
  VARIABLE real_result : sfixed(0 DOWNTO -15);
  VARIABLE imag_result : sfixed(0 DOWNTO -15);
  BEGIN
    real_result := resize(m1_re * m2_re, 0, -15);
    imag_result := resize(m1_im * m2_im, 0, -15);
    result := to_slv(imag_result) & to_slv(real_result);
    RETURN result;
  END FUNCTION;

ENTITY fsm IS PORT (aclk, aresetn, s_axis_data_tdata, s_axis_data_tvalid, s_axis_data_tready, s_axis_data_tlast);
  END fsm;
  END ARCHITECTURE;
A.2 Hand-written Code

Handwritten code refers to the C and VHDL files written by hand. These can be regarded as a complementary material for Chapter 5.

The Hand-written definitions of skeletons for C are shown below, stored in skeletons.h.

1 /*
2 * Copyright Ericsson AB 2023 - All Rights Reserved. No part of this
3 * software may be reproduced in any form without the written permission
4 * of the copyright owner.
5 */
6
7 #include <stdio.h>
8 #include <math.h>
9 #include <complex.h>
10
11 #ifndef skeletons_h
12 #define skeletons_h
13
14 void print_array(int d1, float array[]);
15 void print_matrix(int d1, int d2, float matrix[][d2]);
16 void print_cube(int d1, int d2, int d3, float cube[][d2][]);
17 void concate2d_mat(int d1_mat1, int d1_mat2, int d2, float input_matrix1[][d2], float input_matrix2[][d2], float result[][d2]);
18 void concate3d_cube(int d1_cube1, int d1_cube2, int d2, int d3, float array1[][d2][][], float array2[][d2][][], float result[][d2][][]);
19 void take1d(int take_n, float input_array[], float result[]);
20 void take3d(int d2, int d3, int take_n, float input_cube[][d2][][], float result[][d2][][]);
21 void drop1d(int d1, int drop_n, float input_array[], float result[]);
22 void drop2d(int d1, int d2, int drop_n, float input_matrix[][d2], float result[][d2]);
23 void fanout0d(int n_times, float * input_value, float result[]);
24 void fanout1d(int d1, int n_times, float input_array[], float result[]);
25 void group2d(int d1, int d2, int num, float input_matrix[][d2], float result[][num][d2]);
26 void reduce1d(int d1, void (*operation)(float *, float *, float *), float input_array[], float * result);
27 void reduce2d(int d1, int d2, void (*operation)(int, float *, float *, float *), float input_matrix[][d2], float result[]);
28 void stencil2d(int in_d1, int in_d2, int stencil_length, float a[][], float result[][stencil_length][in_d2]);
29 void complex2real3d(int d1, int d2, int d3, float complex input[][d2][][], float output[][d2][][]);
30 void transpose3d(int d1, int d2, int d3, float complex input[][d2][][], float complex output[][d3][][d1]);

*/
The hand-written code in C is shown below.

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <string.h>
#include <time.h>

#define NoFFT 8 // 256 //window
#define Nob 16// 1024 //range
#define NoB 2
static XScuGic Intc;
static XAxiDma AxiDma;
...

void fCFAR(int d1, int d2, float fCFAR_in[][d2], float fCFAR_out[][d2]){
    float dummy1 = -1.3292279e36;
    float intermediate1[d1 - NoFFT + 1][NoFFT][d2];
    float intermediate2[d1 - NoFFT + 1][d2]; // md
    float intermediate4[d2];
    float intermediate5[NoFFT + 1][d2];
    float intermediate6[NoFFT + 1][d2];
    float intermediate7[d1 - NoFFT + 1 - 2][d2];
    float intermediate8[NoFFT - 1 + d1 - NoFFT + 1][d2]; // env
    float intermediate9[d1 - NoFFT + 1 - 2 + NoFFT * 2][d2]; // lev
    ...
    // v fCFAR template: Generic.Composite
    stencil2d(d1, d2, NoFFT, fCFAR_in, intermediate1);
    md_block(d1, d2, fCFAR_in, intermediate3);
    fanout0d(NoFFT, &dummy1, intermediate4);
    for(int _it = 0; _it < d1 - NoFFT + 1; _it++){ // this is an expanded farm
        aritmean(NoFFT, d2, intermediate1[_it], intermediate2[_it]);
    }
    ...
    // v fCFAR template
    fanout1d(d2, NoFFT + 1, intermediate4, intermediate5);
    fanout1d(d2, NoFFT * 2, intermediate4, intermediate6);
    concat2d_mat(NoFFT + 1, d1 - NoFFT + 1, d2, intermediate4, intermediate5);
    concat2d_mat(d1 - NoFFT + 1 - 2, NoFFT * 2, d2, intermediate6, intermediate7);
    for (int _i = 0; _i < d1; _i++) {
        for (int _j = 0; _j < d2; _j++) {
            normCfa(&intermediate3[_i][_j], &fCFAR_in[_i][_j], &intermediate9[_i][_j], &intermediate8[_i][_j], &fCFAR_out[_i][_j]);
        }
    }
    ...
    // fCFAR
    int main(void){
        ...
    }
}
```

for (int j = 0; j < NoB; j++) {
    for (int k = 0; k < Nob; k++) {
        main_input[i][j][k] = dfb_input[array_index] + dfb_input[array_index + 1] * I;
        array_index = (array_index + 2) % 240;
    }
}
// ^ read in DFB data

// v transpose block
transpose3d(NoFFT, NoB, Nob, main_input, intermediate3);
// ^ transpose

DMA_Init(&AxiDma, 0);
Init_Intr_System(&Intc);
Setup_Intr_Exception(&Intc);
DMA_Setup_Intr_System(&Intc, &AxiDma);
int Status;
TxDone = 0;
RxDone = 0;
Error = 0;
for (int _i = 0; _i < NoB; _i++) {
    for (int _j = 0; _j < Nob; _j++) {
        for (int _k = 0; _k < NoFFT; _k++) {
            transmitter[_k] = (short complex)(intermediate3[_i][_j][_k] * pow(2, 15));
        }
        Xil_DCacheFlushRange((short complex *) transmitter, NoFFT * sizeof(short complex));
    }
    Status = XAxiDma_SimpleTransfer(&AxiDma, (short complex *) transmitter, NoFFT * sizeof(short complex), XAXIDMA_DEVICE_TO_DMA);
    if (Status != XST_SUCCESS) { return XST_FAILURE; }
    Status = XAxiDma_SimpleTransfer(&AxiDma, (short complex *) transmitter, NoFFT * sizeof(short complex), XAXIDMA_DMA_TO_DEVICE);
    if (Status != XST_SUCCESS) { return XST_FAILURE; }
    while (!TxDone || !RxDone) {}
    TxDone = 0;
    RxDone = 0;
    if (Error) { return XST_FAILURE; }
    Xil_DCacheInvalidateRange((short complex *) transmitter, NoFFT * sizeof(short complex));
}
// ^ cube_transmission1

DMA_DisableIntrSystem(&Intc, TX_INTR_ID, RX_INTR_ID);
// ^ cube_transmission1

// v complex2real

complex2real3d(NoB, Nob, NoFFT, intermediate1, intermediate2);
// ^ complex2real

// v change negative to positive, added manually
for (int i = 0; i < NoB; i++) {
    for (int j = 0; j < Nob; j++) {
        for (int k = 0; k < NoFFT; k++) {
            if (intermediate2[i][j][k] < 0) {
                intermediate2[i][j][k] = -intermediate2[i][j][k];
            }
        }
    }
}
// ^ change negative to positive

// v farm_fCFAR

for (int _it = 0; _it < NoB; _it++) {
    for (int k = 0; k < Nob; k++) {
        if (intermediate2[i][j][k] < 0) {
            intermediate2[i][j][k] = -intermediate2[i][j][k];
        }
    }
}
// ^ farm_fCFAR

for (int i = 0; i < NoB; i++) {
    for (int j = 0; j < Nob; j++) {
        for (int k = 0; k < NoFFT; k++) {
            printf("%.16lf \n", main_output[i][j][k]);
        }
    }
}
// ^ farm_fCFAR
The hand-written code for **package** is shown below.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
PORT (
    aclk : IN STD_LOGIC;
    arsetn : IN STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : IN STD_LOGIC;
    m_axis_data_tready : OUT STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC
);
END stream_vector_operation_32b;
```

```vhdl
ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
```

The hand-written code for **VHDL** is shown below.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
PORT (
    aclk : IN STD_LOGIC;
    arsetn : IN STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : IN STD_LOGIC;
    m_axis_data_tready : OUT STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC
);
END stream_vector_operation_32b;
```

```vhdl
ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
```

The hand-written code for **VHDL** is shown below.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
PORT (
    aclk : IN STD_LOGIC;
    arsetn : IN STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : IN STD_LOGIC;
    m_axis_data_tready : OUT STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC
);
END stream_vector_operation_32b;
```

```vhdl
ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
```

The hand-written code for **VHDL** is shown below.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
PORT (
    aclk : IN STD_LOGIC;
    arsetn : IN STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : IN STD_LOGIC;
    m_axis_data_tready : OUT STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC
);
END stream_vector_operation_32b;
```

```vhdl
ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
```

The hand-written code for **VHDL** is shown below.

```vhdl
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE work.myPackage.ALL;

ENTITY stream_vector_operation_32b IS
PORT (
    aclk : IN STD_LOGIC;
    arsetn : IN STD_LOGIC;
    s_axis_data_tdata : IN STD_LOGIC_VECTOR(31 DOWNTO 0);
    s_axis_data_tvalid : IN STD_LOGIC;
    s_axis_data_tready : OUT STD_LOGIC;
    s_axis_data_tlast : IN STD_LOGIC;
    m_axis_data_tdata : OUT STD_LOGIC_VECTOR(31 DOWNTO 0);
    m_axis_data_tvalid : IN STD_LOGIC;
    m_axis_data_tready : OUT STD_LOGIC;
    m_axis_data_tlast : OUT STD_LOGIC
);
END stream_vector_operation_32b;
```

```vhdl
ARCHITECTURE stream_vector_operation_32b_arch OF stream_vector_operation_32b IS
```
```vhdl
SIGNAL index : INTEGER := 0;
BEGIN
  PROCESS (aclk) BEGIN
    IF (aclk'event AND aclk = '1') THEN
      m_axis_data_tvalid <= s_axis_data_tvalid;
      m_axis_data_tlast <= s_axis_data_tlast;
      s_axis_data_tready <= m_axis_data_tready;
    END IF;
  END PROCESS;
  PROCESS (aclk, arestn) BEGIN
    IF (arestn = '0' OR s_axis_data_tvalid = '0') THEN
      index <= - 1;
    ELSIF (aclk'event AND aclk = '1' AND s_axis_data_tvalid = '1') THEN
      index <= index + 1;
    ELSE
      NULL;
    END IF;
  END PROCESS;
  PROCESS (index) BEGIN
    IF index < 0 THEN
      m_axis_data_tdata <= "00000000000000000000000000000000";
    ELSIF index > 8 - 1 THEN
      m_axis_data_tdata <= "00000000000000000000000000000000";
    ELSE
      m_axis_data_tdata <= complex_mul(input1 => s_axis_data_tdata, input2 => mk_WeightCoefs(index));
    END IF;
  END PROCESS;
END stream_vector_operation_32b_arch;
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;
USE ieee.numeric_std.ALL;
USE ieee.fixed_pkg.ALL;
USE ieee.fixed_float_types.ALL;
USE work.myPackage.ALL;
ENTITY fsm IS PORT (
  aclk: in std_logic;
  arestn : in std_logic;
  s_axis_data_tdata : in std_logic_vector(31 DOWNTO 0);
  s_axis_data_tvalid : IN STD_LOGIC;
  s_axis_data_tready : OUT STD_LOGIC;
  s_axis_data_tlast : IN STD_LOGIC;
  m_axis_data_tdata : out std_logic_vector(31 DOWNTO 0);
  m_axis_data_tvalid : OUT STD_LOGIC;
  m_axis_data_tready : IN STD_LOGIC;
  m_axis_data_tlast : OUT STD_LOGIC);
END fsm;
ARCHITECTURE fsm_arch OF fsm IS
  SIGNAL event_tlast_unexpected : STD_LOGIC;
  SIGNAL event_tlast_missing : STD_LOGIC;
  SIGNAL event_status_channel_halt : STD_LOGIC;
  SIGNAL event_data_in_channel_halt : STD_LOGIC;
  SIGNAL event_data_out_channel_halt : STD_LOGIC;
  SIGNAL event_frame_started : STD_LOGIC;
  SIGNAL s_axis_config_tdata : STD_LOGIC_VECTOR(7 DOWNTO 0) := (OTHERS => '0');
  SIGNAL s_axis_config_tvalid : STD_LOGIC := '0';
  SIGNAL s_axis_config_tready : STD_LOGIC := '1';
  COMPONENT xfft_0 IS PORT (...);
  END COMPONENT;
  COMPONENT stream_vector_operation_32b IS PORT (...);
  END COMPONENT;
  SIGNAL intermediate_axis_data_tdata : STD_LOGIC_VECTOR(31 DOWNTO 0);
  SIGNAL intermediate_axis_data_tvalid : STD_LOGIC;
  SIGNAL intermediate_axis_data_tready : STD_LOGIC;
BEGIN
  -- v fsm
  f2 : entity work.stream_vector_operation_32b PORT MAP(...);
  f1 : xfft_0 PORT MAP(...);
END fsm_arch;
```