Bluenose II: Towards Faster Design and Verification of Pipelined Circuits

by

Ca Bol Chan

A thesis
presented to the University of Waterloo
in fulfillment of the
thesis requirement for the degree of
Master of Applied Science
in
Electrical and Computer Engineering

Waterloo, Ontario, Canada, 2008

Abstract

The huge demand for electronic devices has driven semiconductor companies to create better products in terms of area, speed, power *etc.* and to deliver them to market faster. Delay to market can result in lost opportunities. The length of the design cycle directly affects the time to market. However, inadequate time for design and verification can cause bugs that will cause further delays to market and correcting the error after manufacturing is very expensive. A bug in an ASIC found after fabrication requires respinning the mask at a cost of several million dollars. Even as the pressure to reduce the length of the design cycles grows, the size and complexity of digital hardware circuits have increased, which puts even greater pressure on design and verification productivity. Pipelining is one optimization technique which has contributed to the increased complexity in hardware design. Pipeline increases throughput by overlapping the execution of instructions. It is a challenge to design and verify pipelines because the specification is written to describe how instructions are executed in sequence while there can be multiple instructions being executed in a pipeline at one time. The overlapping of instructions adds further complexity to the hardware in the form of hazards which arise from resource conflicts, data dependencies or speculation of parcels due to branch instructions.

To address these issues, we present *PipeNet*, a metamodel for describing hardware design at a higher level of abstraction and Bluenose II, a graphical tool for manipulating a *PipeNet* model. *PipeNet* is based on a pipeline model in a formal pipeline verification framework. The pipeline model contains arbiters, flow-control state machines, datapath and data-routing. The designer describes the pipeline design using *PipeNet*. Based on the *PipeNet* model, Bluenose II generates synthesizable VHDL code and a HOL verification script. Bluenose II's ability to generate HOL scripts turns the HOL theorem prover into Bluenose II's external verification environment. A direct connection to HOL is implemented in the form of a console to display results from HOL directly in Bluenose II. The data structures that represent *PipeNet* are evaluated for their extensibility to accommodate future changes. Finally, a case study based on an implementation of a two-wide superscalar 32-bit RISC integer pipeline is conducted to examine the quality of the generated codes and the entire design process in Bluenose II. The generation of VHDL code is improved over that provided in Bluenose I, Bluenose II's predecessor.

Acknowledgements

A heartfelt thanks to Professor Mark Aagaard who not only gave me the opportunity to experience firsthand what research is all about but also the opportunity to work with a great supervisor. He is always full of encouragement (when I did not do a good job), patience (when I did not know how to do the job), and motivation (for his students to strive to do the best possible job!)

I would also like to thank Vlad Ciubotariou for helping me out with his Eclipse expertise.

Thanks to all the friends who have helped me along the way. Non-technical help is equally important in the completion of this thesis.

I thank my parents and my brother for their unconditional love and support which have allowed me to reach this point in my life.

To my mom, dad and brother

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Without looking at the statistics, it is not hard to tell that the use of electronic devices has greatly increased. Headlines such as “Worldwide PC processor market hits record levels of unit shipments again” [1], “Cell phone sales hit 1 billion mark” [2] and “1 million iPhone 3Gs sold in 3 days” [3] are not uncommon these days. The huge consumer appetite for electronic devices was made more evident when an article described them as physical nourishment to college students [4]. Semiconductor companies are competing to fulfill this increasing demand by creating better products in terms of area, speed, power *etc.* and delivering them to the market faster.

Delay to market can result in lost opportunities. The length of the design cycle directly affects the time to market. However, inadequate time for design and verification can cause bugs that will cause further delays to market and correcting the error after manufacturing is very expensive. A bug in an ASIC found after fabrication requires respinning the mask at a cost of several million dollars. Even as the pressure to reduce the length of the design cycles grows, the size and complexity of digital hardware circuits have increased, which puts even greater pressure on design and verification productivity.

Driven by the demand for better (smaller, faster *etc.*) products, highly advanced architectural ideas are developed which result in more complex designs. An example is pipelining which increases the throughput at the price of increased design and verification complexity. In the last 15 years, designs have grown in size and complexity by 1 to 2 orders of magnitude. Many hardware intellectual property (IP) blocks today are larger than an entire chip back then [5]. Engineers were reported working on multiple projects at once although the average size of a design team has grown [6]. An engineer can write only so many lines of code a day; therefore, in the face of increasing size and complexity in hardware design, each line has to be worth “more” than before. More “value” can be added to the code in several ways. One way is to raise the level of abstraction so a line in a high-level hardware description language can be synthesized into more lines of code at the register transfer level (RTL). Another way is to incorporate verification into the design process to achieve "first-time-right" functional correctness in a shorter time. Such are the motivations for the Bluenose II project. Bluenose II aims to increase designer productivity and reduce verification effort by raising design abstraction, focusing on pipelined circuits in particular.

With these goals in mind, we take a more detailed look at pipelines and explain why it is a challenge to design and verify them. Following that, we present the research overview, thesis statement and research contributions. The chapter is ended with an outline of the major sections in this thesis.

1.1 Overview of Pipelines

Pipelining is an optimization technique in hardware design that overlaps the execution of instructions. A hardware pipeline can be viewed as an assembly line for instructions. Each pipeline stage processes an instruction and passes the instruction to the next stage. This process occurs in all pipeline stages simultaneously, thus allowing multiple instructions to be processed concurrently. An advantage of a pipelined design is the increase of throughput since each instruction gets processed sooner compared to a non-pipelined design in which each instruction has to wait for the previous one to finish. However, the overlapping of instructions adds complexity to the hardware in the form of hazards. Three types of hazards are present in pipelined designs. Structural hazards arise from resource conflicts among overlapped execution of instructions. Data hazards are caused by data dependencies between overlapping instructions. Finally, control hazards deal with speculation of parcels due to branch instructions. These concepts are explained in depth by Hennessy and Patterson in [7]. Many design complexities arise from trying to resolve these hazards properly. And to verify that these complex pipelined designs are correct further adds to the overall challenge.

Even without these complexities, a pipelined implementation by itself is hard to verify formally. The challenge stems from the fact that the specification is written to describe how instructions are executed in sequence while there can be multiple instructions being executed in a pipeline at one time. Many techniques have been developed to verify a pipelined implementation against its non pipelined specification. A well-known correctness statement known as Burch-Dill flushing was proposed in [8]. For small scale designs, *e.g.*, in-order pipelines with a small depth, Burch-Dill flushing can be fully automated. For realistic designs like Intel's Core microarchitecture which has 14 stages with out-of-order execution [9], Burch-Dill flushing suffers from the state explosion problem as do other model checking techniques. Many other formal verification techniques have been developed since then and applied in the industry. Although companies reported that formal verification has led to the discovery of subtle design bugs [10], it is still not widely adopted since formal verification requires specialized expertise which most designers do not have.

In view of these challenges, Aagaard developed a pipeline verification framework which structures and simplifies the proof of the pipelining circuitry for structural hazards [11]. The framework provides a stage template which consists of a set of components and techniques for how to verify these components. By mapping a pipeline design into these components, the verification is in effect divided into smaller tasks. It was recognized that in addition to easing the verification effort, a designer can also use these components to design a pipeline. From this realization, Higgins created Bluenose I [12]. Bluenose I demonstrated that the stage template from Aagaard's verification framework could be applied to pipeline design [12]. It showed that the stage template used in combination with a graphical user interface simplified the design and verification of pipelines. Building on the work of Bluenose I, Bluenose II is created, which is the topic of this thesis.

1.2 Research Overview

Bluenose II continues in the direction taken by its predecessor to increase automation in pipeline design and verification. One particular area that we explore in this thesis is the model driven development methodology. We examine the applicability of the methodology not only to the development of Bluenose II but also to the pipeline design process. For this, we created a pipeline metamodel called *PipeNet* based on Aagaard's stage template. A metamodel is literally a model that describes another model. Since a model represents concepts from a domain, the metamodel for that model provides a modeling language for that domain. Consequently, a metamodel can also be considered a domain specific language [13].

We use Eclipse and its modeling plugins to build Bluenose II based on *PipeNet*. Bluenose II can be viewed as a domain specific tool for instantiating and configuring the *PipeNet* metamodel. In turn, the designer creates a pipeline model using the *PipeNet* "language". The *PipeNet* language should be expressive enough to capture all the relevant characteristics of a pipeline design. The effectiveness can be evaluated by the ease to generate codes. If the metamodel is not expressive enough, it will be difficult to retrieve or derive information needed for code generation. Bluenose II generates synthesizable VHDL code and a HOL verification script based on the pipeline model entered by the designer. The data structures that represent *PipeNet* are then evaluated for their extensibility to accommodate future changes. Finally, a case study based on the OpenRISC architecture [14, 15] is conducted to examine the quality of the generated codes and the entire design process in Bluenose II. Ultimately, Bluenose II should pave the road to faster design and verification of pipelined circuit that was started in Bluenose I.

1.3 Thesis Statement and Contributions

In this thesis, we propose that not only can model driven development be applied to the development of a tool for pipeline design, it can also be applied to the pipeline design process. The pipeline metamodel, which forms the basis of Bluenose II, can capture all the relevant characteristics of a pipeline design. This is demonstrated through the ease to generate codes from the pipeline model instantiated from the metamodel. Harnessing the power of Eclipse's modeling framework and its code generation facility, Bluenose II is a full featured environment for pipeline design and verification. Altogether, Bluenose II improves upon the work started by Bluenose I and paves the road to faster design and verification of pipelined circuits.

This work makes the following contributions to the area of hardware design and verification.

- We create *PipeNet*, a pipeline metamodel, which is suitable for capturing pipeline designs.
- Extensible data structures that implement *PipeNet* are developed.
- We develop Bluenose II which implements the *PipeNet* data structures in Eclipse with full-featured user interface. Bluenose II is developed using the model driven engineering approach. We show how a designer can apply the same approach to pipeline design using Bluenose II.
- The generation of VHDL code is improved over that provided in Bluenose I in terms of conciseness.
- We enhance the Bluenose tool with the HOL generation feature for verification. A direct connection from Eclipse to HOL in the form of a console in Eclipse is implemented. The HOL console displays HOL's output directly in Bluenose II.

1.4 Thesis Outline

The remainder of the thesis is organized as follows.

- Chapter 2 introduces the pipeline model and provides details on the technologies used in Bluenose II. Related work is also described in this chapter.
- Chapter 3 outlines the design flow in Bluenose II and the implementation of design related features.
- Chapter 4 describes HOL related modules which are used in pipeline verification.

- Chapter 5 presents the data structures and discusses some implementation issues that arose in the development of Bluenose II.
- Chapter 6 provides the results of a case study based on an implementation of a two-wide superscalar 32-bit RISC integer pipeline in Bluenose II.
- Chapter 7 contains the conclusions of the thesis and directions for future work.

Chapter 2

Background

In the previous chapter, we discussed the challenges in the design and verification of pipelined circuits; we also gave an overview of Bluenose II, a design and verification tool for the pipelines. This chapter gives the background that will help in understanding the inner working and philosophy of Bluenose II.

The pipeline model which Bluenose II is based on is presented in Section 2.1. The model driven development approach that we took to develop Bluenose II is discussed in Section 2.2. The Eclipse Graphical Modeling Framework and the HOL theorem proving system, which were used in the development of Bluenose II, are described in Section 2.3 and Section 2.4 respectively. Bluenose I, the predecessor to Bluenose II, is discussed in Section 2.5. Finally, we relate and contrast our work to others in Section 2.6.

2.1 A Pipeline Model

Aagaard's solution (modified by Higgins in [12]) to the design complexity is to view a pipeline as a system of stages that follows a request/accept protocol [11]. Instructions are treated as parcels and are transfered from one stage to the next based on this protocol. In order for a pipeline stage to send a parcel, it first sends a request to the next stage. The stage will only send the parcel upon the receipt of an accept signal from the next stage. The receiving stage sends an accept signal when it has received a request and when it is ready to accept a parcel. Aagaard defined a variety of instantiations of this request/accept protocol that allows for efficient implementations in many pipelines.

The decision to request or accept is decentralized and distributed to the stages themselves. The transferring of parcels and the decision-making are coordinated by the components inside each stage. These components are shown in Figure 2.1. Table 2.1 describes the role and the number of instances allowed in a stage for each component. The first group of components (*Interface Prev* and *Interface Next*) make up the interface of the stage for transferring parcels. One instance of

Interface Prev (*Interface Next*) is needed per incoming (outgoing) signal to (from) a stage. The second group of components (*Register* and *Datapath*) form the datapath of the stage. One instance of *Register* is required per incoming signal that needs to be registered, while the datapath of a stage is encapsulated in one *Datapath*. The components in the third group together act as the control center for the stage. Since each stage needs only one "control center", only one instance of each component in this group is allowed. The interface and datapath components are placed on the left of Figure 2.1 and the control components on the right.

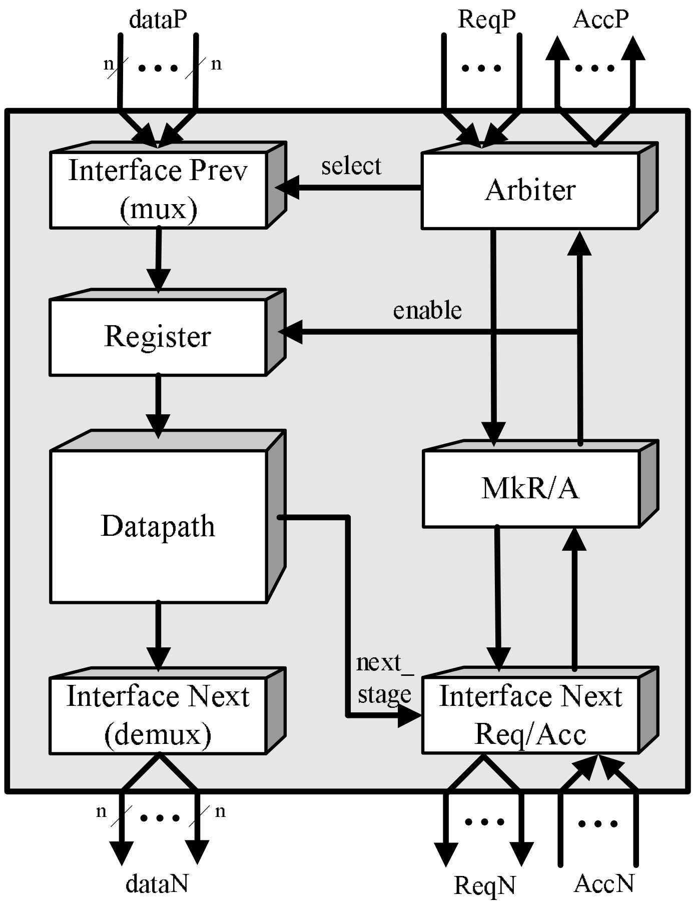


Figure 2.1: Aagaard's pipeline stage decomposition

There are different instantiations for *Arbiter* and *MkR/A* that a designer can choose from. The set of instantiations allows a designer to design pipelines with different behaviours. The choice for *Arbiter* depends on the number of upstream pipeline stages that are connected to the stage containing the *Arbiter* and the expected request behaviour. The choice for *MkR/A* is based on the behaviour of the *Datapath* of the containing stage, and the request and accept behavoiurs of the upstream and downstream stages respectively. Descriptions of each instantiation of *Arbiter* and *MkR/A* are given in Table 2.2 and Table 2.3.

Higgins provided mathematical descriptions of the instantiations in [12]. The definition of *General MkR/A* is reproduced in Definition 2.1 as an example. All

Table 2.1: Pipe stage components

Component	Description	Cardinality
Interface Prev	Receives datapath values from the neighboring stages	1 to many
Interface Next	Sends datapath values from the neighboring stages	1 to many
Register	Pipe stage register	0 to many
Datapath	Pipe stage datapath	1
Arbiter	Arbitrates requests from the upstream pipe stages	1
MkRA	Keeps track of the state of the pipeline stage	1
Interface Next Req/Acc	Monitors the request and accept signals from downstream pipe stages	1

Table 2.2: Description of instantiations of *Arbiter*

Degenerate :	There is only one input, so forwards the request directly.
Exclusive :	There is more than one input, but there will be at most one request at any time.
StaticPriority :	There is more than one input and there may be more than one request at a time. The request with the highest priority is forwarded.

input and output signals are prefixed with *i_* and *o_*; the rest are internal signals. The signals in parentheses are logical groupings of the input and output signals according to their usage. They also represent the interface of *General MkR/A*. The description is quantified over time and the time is indicated with the subscripts. The "$\wedge$" and "$\vee$" operators represent the logical `AND` and `OR` of two signals respectively. The conjunction of the signal assignments is indicated by the "&" symbol. Together these signal assignments make up the definition of *General MkR/A*.

Definition 2.1: General MkR/A

$$
\begin{array}{l}
mkReqAccGeneral(i\_reqP, o\_accP)(o\_reqN, i\_accN) \\
\quad (i\_maskP, i\_maskN, i\_abort, i\_loopMstrReq) \equiv \\
\ \exists\ reqP, accN, reqN, accP\ .\ \forall\ t\ . \\
\begin{array}{rl}
reqP_{(t)} & = i\_reqP_{(t)} \wedge i\_maskP_{(t)}\ \& \\
accN_{(t)} & = (i\_accN_{(t)} \vee i\_loopMstrReq_{(t)}) \wedge i\_maskN_{(t)}\ \& \\
reqN_{(t+1)} & = [reqP_{(t)} \vee (reqN_{(t)} \wedge \neg accN_{(t)} \wedge \neg i\_abort_{(t)})] \wedge \neg i\_reset_{(t)}\ \& \\
accP_{(t)} & = accN_{(t)} \vee \neg reqN_{(t)} \vee i\_abort_{(t)}\ \& \\
o\_reqN_{(t)} & = reqN_{(t)} \wedge i\_maskN_{(t)} \wedge i\_abort_{(t)}\ \& \\
o\_accP_{(t)} & = accP_{(t)} \wedge i\_maskP_{(t)}
\end{array}
\end{array}
$$

Table 2.3: Description of instantiations of *MkR/A*

Degenerate :	This instantiation of *MkR/A* assumes that there is a request from the previous stage in every clock cycle and the next stage always accepts. In other words, the *Degenerate MkR/A* sends an accept signal to the previous stage and a request to the next stage in every clock cycle.
General :	There is a unit delay through the current stage. There may or may not be a new request from a previous stage and the next stage may or may not accept.
MultiDelay :	The current stage may take multiple cycles to process a parcel. There may or may not be a new request from a previous stage and the next stage may or may not accept.
UnitDelay :	There is a unit delay through the current stage. There may or may not be a new request from a previous stage and the next stage always accepts.
ZeroDelay :	There is zero delay through the current stage. This instantiation of *MkR/A* merely connects the request (accept) signal from (to) the previous stage to (from) the next stage.

The request/accept protocol, components and their instantiations are derived from Aagaard's pipeline verification framework for structural hazards [11]. In Aagaard's framework, the instantiations of the components characterize the pipelining circuitry that is to be verified. The framework provides a verification strategy based on these components. Therefore, mapping the pipeline design into these components eases the task of verification.

The request/accept protocol, components and their instantiations together form a template of a pipeline stage. The designer instantiates and configures the template for each stage in the pipeline to describe the overall structure and behaviour of the pipeline. The benefits of using this stage template are as follows:

1. The behaviours of these components are in terms familiar to pipeline designers therefore the learning curve is less daunting.

2. The components encapsulate and abstract away the low level mechanisms used in hardware to stall stages and to route parcels from one stage to the next.

3. The encapsulation of details also means that the structure of the pipeline is available at the same time as the designer completes the design of the behaviour of the pipeline.

4. The components which implement the request/accept protocol allow the designer to create pipelines that handle structural hazards correctly.

5. The mapping between the pipeline model and Aagaard's verification framework allows the designer to design for verification.

This conceptual model is the foundation of Bluenose II. From it, a metamodel which we named *PipeNet*, is created to be used in Bluenose II. Using Bluenose II, the designer describes the pipeline design in *PipeNet* and the pipeline model created from *PipeNet* allows the design to be reasoned about. *PipeNet* is described further in Section 3.2. The model leads us to take advantage of the model driven development approach. Model driven development and metamodels are explained in Section 2.2.

2.2 Model Driven Development

There are different standards and methodologies for model driven development (MDD) [16, 17]. Each has its own terminology but in generic terms, model driven development is an approach to develop software or any other system by using a model or models which abstract some concepts of the problem domain. We focus our discussion of model driven development to the scope of Model Driven Architecture (MDA), a framework proposed by the Object Management Group (OMG) [17].

MDA supports model driven development by providing standards to define and transform models. An application is first specified using one or more Platform Independent Models (PIM). As its name implies, the model is independent from platform details such as a programming language. The model's independence allows it to be transformed into different Platform Specific Models (PSM). It may take multiple transformations to map a high level model to executable code for a given platform [13]. Therefore, the role of a model, whether it is a PIM or a PSM, is relative to the other models. For example, a C++ program is a PSM to the Unified Modeling Language (UML) model that it implements, but it is a PIM to the corresponding assembly code which is executed on specific processors. Model, be it PIM or PSM, is specified with the modeling constructs defined in its metamodel. A model is an instantiation of its metamodel. A metamodel is just like any other model that abstracts the concepts in a domain, except the domain of a metamodel is another model. Therefore, the concept of a metamodel is also relative to the model that it is being related to. A model can be a metamodel to another model, while it itself conforms to another metamodel. This "chain" of models is also referred to as meta-layers or metalevels [18]. In the previous example of UML, C++ and the assembly language, there are three layers. Since a model represents the concepts from a domain, the metamodel for that model provides a modeling language for that domain. In this way, a metamodel can also be considered a domain specific language (DSL).

A DSL provides the syntax to express ideas in a particular domain. A DSL offers the following benefits:

- The specialized notation provides domain specific abstraction. The layer of abstraction provides the ability to capture an idea in the domain more concisely.
- A DSL allows its user to identify and communicate ideas that are too abstract to express in a general purpose programming language.
- The domain information captured by the language coupled with domain knowledge enables domain specific error checking and optimization. This leads to the creation of domain intelligence tools that provide support to facilitate domain specific activities.

To illustrate the benefits of a DSL, consider the example of a database query language which is a DSL for a structured collection of data. A *table* in the database query language is known to refer to a set of data that is organized in rows and columns. The database query languages also provide the syntaxes to describe operations on a *database* such as creating and removing a *table*. Domain abstractions such as rows, columns, and table creation make up a language for describing the organization and manipulation of data. A DSL is not confined to text and to be standalone; it can also be visual or embedded in another programming language [13].

In MDA, models or metamodels have to be expressed in a Meta-Object Facility (MOF) based language [17]. MOF is an OMG standard for defining modeling languages. A benefit of expressing a model in a MOF-based modeling language is that the model can be transformed and manipulated by MOF-compliant tools [18]. The Eclipse Modeling Framework (EMF), part of the Eclipse Graphical Modeling Framework (GMF) that we used to develop Bluenose II, is compatible with a subset of the MOF metamodel called the Essential Meta-Object Facility (EMOF) metamodel. EMF can be thought of as a EMOF tool for defining metamodels. It has the generative functionality to produce a set of Java classes for the metamodel based on its specification. To distinguish from the EMOF metamodel, the core metamodel in EMF is called Ecore [19]. EMF is discussed further in Section 2.3.

We adopted the model driven development approach to harness the generative functionality of EMF and GMF. The generative functionality of these Eclipse plugins allows us to focus on implementing features for manipulating the pipeline model instead of basic editor functionalities like file management. GMF is discussed in Section 2.3.

2.3 Eclipse Graphical Modeling Framework

Eclipse is an open source integrated development environment. It was originally created by IBM and has evolved into a platform supported by an open source community [20]. Eclipse can be customized and extended through plug-ins. Bluenose II runs on the Eclipse platform as a plug-in. Bluenose II was implemented also using other Eclipse plug-ins; an important one is GMF.

GMF is a model driven tool set for developing graphical editors. It includes a generative component for generating the graphical editor and a runtime infrastructure which provides services to the graphical editor during runtime [21]. GMF is based on two other plug-ins: EMF and the Eclipse Graphical Editing Framework (GEF). GMF integrates the modeling functionalities of EMF into the model-view-controller architecture of GEF. The model-view-controller architecture is a design pattern that decouples the model from the view so that more than one view can be implemented to display the data of the model. An overview of GMF is shown in Figure 2.2.

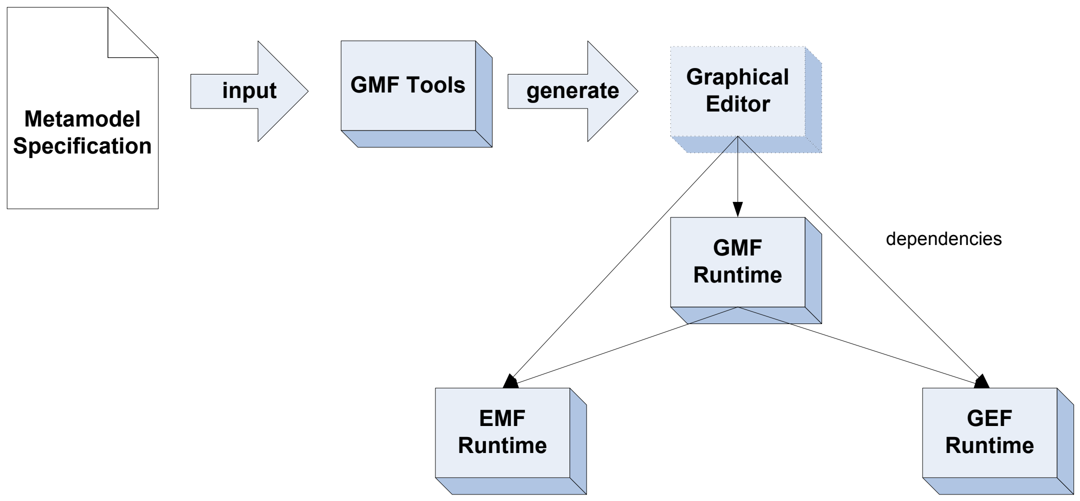


Figure 2.2: Overview of the Graphical Modeling Framework

The basic steps for developing a graphical editor with GMF are illustrated in Figure 2.3. The developer first develops the domain model, graphical definitions and tooling definitions. The domain model defines the abstract syntax of the modeling language that will be available in the graphical editor. It is expressed using the Ecore metamodel. The graphical definitions describe the graphical elements (*e.g.*, figures and connectors) that will appear in the editor. The tooling definitions specify tools (*e.g.*, palette and toolbars) that will be available in the editor. These models are fit together in the mapping model such that the graphical elements are linked to the concrete aspect of the domain model and tools in the editor. The mapping model is transformed into a generator model which contains modifiable code generation parameters (*e.g.*, plug-in name). Finally, the diagram code (Java code) for the editor plug-in is generated from the generator model. The generated plug-in reuses components from the GMF runtime infrastructure and runs on the Eclipse platform. The editor persists the data which represent the models created by the end user, in XML Metadata Interchange (XMI), a standard also defined by OMG [21].

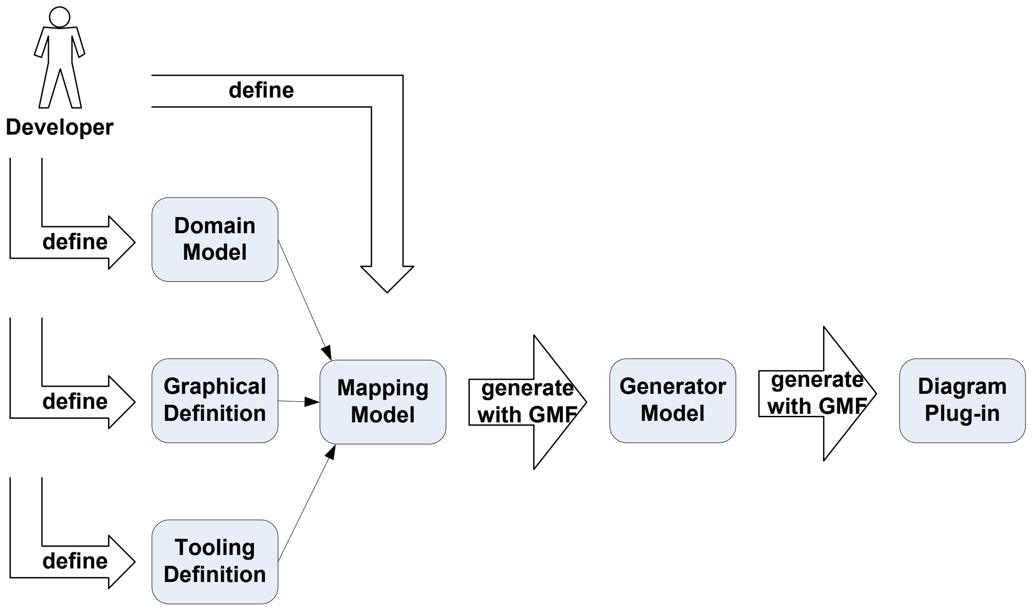


Figure 2.3: Basic steps for developing a graphical editor with the Graphical Modeling Framework

We are involved in the model driven development field both as a modeler and a model user. We used the Ecore metamodel to specify *PipeNet*, a pipeline metamodel. In turn, the pipeline designer uses *PipeNet* to describe the pipeline design. The relationship between these models is shown in Figure 2.4. We used GMF to develop Bluenose II, a graphical editor to manipulate the pipeline model. GMF's generative functionality and the reusability of its runtime components allow us to derive the benefit of adopting the model driven development approach.

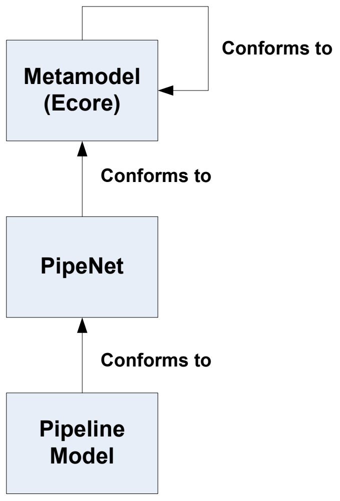


Figure 2.4: Relationship between (meta)models

2.4 The HOL Theorem Prover

Formal verification involves proving that the implementation satisfies the specification using mathematical reasoning. One formal verification approach is logic inference. The specification, implementation and their relation are expressed in some type of logic. The verification then becomes proving a theorem in that specific logic [22]. Theorem provers are used to keep track of this type of proofs. Other benefits of the theorem prover are explained as part of the description of HOL. Bluenose II generates a HOL script to enable the verification of pipeline designs.

The HOL system is a theorem prover for the theory that it is named after, higher order logic. In higher order logics, functions can take functions as arguments, and quantifiers can range over functions [23]. Therefore, higher order logic supports modeling a digital system as a relation between its input and output signals and models a signal as a time-to-value function. The signal function, when given a time, returns the state of the signal. With this in mind, HOL was originally created for hardware verification at the register transfer level, modified from the Cambridge LCF (Logic for Computable Functions) system. LCF-style theorem provers have terms from the typed λ-calculus and formulae from predicate calculus [24].

Some benefits of using theorem provers like HOL are that the system keeps track of the proof, ensures the proof's soundness, automates the proof when possible and allows extensibility. In LCF-style theorem provers, a small core of axioms and inference rules are encoded. New theorems can only be constructed by applying a sequence of inference rules on the axioms, thereby ensuring soundness of the proof — if the system proved something is true, then it really is true. Furthermore, since theorems are represented as an abstract data type, strict type checking prevents any violation of the soundness of the system. The HOL system supports both forward and goal oriented proofs. A step in a forward proof can only be derived with the rules of inference from results of other proofs, namely theorems, previous steps in the proof, or axioms. In a goal oriented proof, the proof goal is decomposed into subgoals with tactics or tacticals which are combinations of tactics. A tactic matches a goal or a subgoal with an axiom or a proven theorem. A goal is solved when there are no more subgoals to be proven; the goal itself or all of its subgoals are matched with axioms or theorems. Proven theorems can be stored in theories for future use [23].

The extensibility of HOL is enabled by the separation of the meta language and object language. HOL is written in the programming language ML (Meta Language) which is a functional language, designed by the same creator as the original LCF system. ML is strictly typed to implement type checking. The separation of the meta language (ML) and object language (HOL) allows the logic to be extended. New proof procedures can be developed through ML and theories can be added to HOL. HOL supports the development of application specific tools [23] — in this case, the application specific constructs become the object language and HOL becomes the meta language. There is research to embed hardware description languages into formal logics to give formal semantics to these languages

[24]. Lastly, HOL can be extended by dynamic loading and calling of external C functions, a feature provided by ML [25]. This allows HOL to harness the power of other verification tools.

HOL is used in hardware verification to model specifications and implementations. An implementation can then be reasoned about with HOL. For example, an implementation can be proven to conform to a specification, and if the specification can be proven to satisfy a property then we can say the implementation also satisfies the property. To facilitate pipeline verification in HOL, we have established a connection between Bluenose II and the HOL system, and from HOL to other verification tools. Bluenose II generates HOL script based on the pipeline design entered by the user. The user can then reason about the design with HOL and its extensions. In this thesis, we focus on the connection between Bluenose II and HOL because the Bluenose II connections from HOL to external tools are not fully mature.

2.5 Bluenose I

Bluenose I is the predecessor to Bluenose II. Bluenose II builds on the same mathematical model (described in Section 2.1) as Bluenose I. As the main purpose of both tools is to provide a means to manipulate the mathematical model, they have the components that a designer uses to build a pipeline. However, many differences exist between these two tools in how the designer uses and manipulates the components.

The language of implementation is one reason for the differences. Bluenose I was implemented in Moscow ML, which is a functional language [25]. Bluenose II was implemented in Java, which supports the object-oriented paradigm. Another reason for the differences was the different approach to the development of the two tools. The model driven development approach was adopted for the development of Bluenose II as described in Section 2.2. This has led to the definition of a more explicit pipeline model, which is described in Section 3.2. This explicit pipeline model did not exist in Bluenose I.

The verification task is delegated to third party tools in both Bluenose I and Bluenose II. Bluenose I generates queries for Cadence FormalCheck, a commercial model-checking tool. The queries are used to assert certain properties in the pipeline design. On the other hand, Bluenose II generates a HOL script that models the user's pipeline design. The designer can then reason about the pipeline design in HOL. We have chosen to integrate HOL and Bluenose II because HOL offers a general purpose verification environment which can be extended with other verification tools.

Some features in Bluenose I are improved in Bluenose II. More basic editor functionalities such as file management are enabled in Bluenose II due to the Eclipse platform and its plug-ins. The VHDL code generation is also improved in Bluenose II. The differences between Bluenose II and Bluenose I listed in this section are an overview. The specific changes between Bluenose II and Bluenose I are discussed in detail in corresponding sections.

2.6 Related Work

Due to the enormous amount of detail in the register transfer level (RTL) of hardware design, many languages and tools have been created to raise the level of abstraction to different degrees. The abstraction increases the designer's productivity by showing only the details of interest. The abstraction level where system design and verification are carried out is also known as the electronic system level (ESL) [5]. These languages and tools model different aspects of the hardware architecture ranging from the behaviour to the structure. They also represent the models in different ways, from visual to textual. Although they all serve the purpose of capturing certain aspects of the architecture, some are meant only for simulating functionalities, some for design space exploration and some generate RTL that produces hardware with comparable performance as the hand-written RTL. In the latter case, these tools are also called high-level synthesis (HLS) tools. In this section, we discuss a subset of this work most related to Bluenose II and *PipeNet*.

The block based approach taken by Bluenose II is employed by many commercial and academic tools, be they graphical or textual. They include Mentor Graphics' HDL Designer [26], Altera's Quartus II [27] and the *X* language [28], to name a few. This is not surprising as hardware designers often used blocks as a high level representation to describe and visualize different entities and their relationship to each other. In Bluenose II, the purpose of the blocks is more than just to hide the details, the blocks represent elements in the *PipeNet* model and their identities are used for different types of analysis. In this model driven development aspect, MCF [29] and MMV [30] are very similar to Bluenose II.

The Metamodeling-driven Component Composition Framework (MCF) is developed by Mathailkutty *et al.* to harness the power of a metamodel based framework to create system-level models [29]. In MCF, the user creates a design by instantiating, configuring and connecting components which represent SystemC intellectual property (IP) cores. MCF then checks for inconsistencies in the model and generates the appropriate glue-logic for the components. The Metamodeling Based Microprocessor Validation Environment (MMV) is also developed by Mathailkutty *et al.* [30]. MMV is a validation environment based on metamodeling. In MMV, the user models the design at the system level and continually refines it to a lower abstraction. The user specifies the models in each abstraction level using the respective MMV metamodels. MMV then generates code for specific validation targets based on the translators customized by the user.

A metamodel is characterized by the domain concepts that it can express. The MCF metamodel allows the modeling of system-level components for design space exploration. On the other hand, the MMV metamodel is intended for microprocessor validation and provides specialized elements such as *Pipeline*, *Stages* and instruction registers. While *PipeNet* focuses on the modeling of pipelines, it can also be used to describe system-level components. Furthermore, *PipeNet* is intended to be used in both design and verification. There appears to be no direct mapping between the MCF and MMV metamodels.

Bluenose II, MCF and MMV are similar in a number of ways. The tools promote model driven development. MCF and MMV are both built on a metamodeling framework called the Generic Modeling Environment (GME) [31] which is much like GMF to Bluenose II. GME, like GMF, is a toolkit for creating visual domain specific languages. Instead of Ecore, the metamodel is specified in UML [32]. The component model of GME is built upon Microsoft's technology; therefore, although Java access is available, its primary languages of integration are C++ and Visual Basic. The metamodels in these tools impose structural constraints on the construction of the model. MCF and MMV enforce additional constraints on the static semantics of the model by checking the model against rules specified in the Object Constraint Language (OCL) [33]. As GMF has built-in support for OCL and other validation facilities, our future work includes providing additional validation in Bluenose II to guide the designer in the modeling process.

The differences between Bluenose II, MCF and MMV are as follows. The influence of SystemC can be seen in the metamodel used by MCF. For example, the type *Argument* is used in MCF for representing arguments in a function call at transaction level in SystemC. Although the design of Bluenose II's *PipeNet* is inspired by some constructs in VHDL, we are confident that the concepts in *PipeNet* are universal to hardware design languages. The models specified in *PipeNet* are independent of implementation language. As a result, we can easily implement a generator that walks through the model and outputs a different language. The HOL generator in Bluenose II is an example of one such generator. It is not clear whether MCF can easily be extended to support other languages or not. In this regard, MMV is more similar to Bluenose II, since it stresses a language-independent representation to enable multi-target code generation.

Type mismatch between ports in a connection is handled differently between Bluenose II and MCF. Bluenose II generates a conversion function template for the designer to implement when a type mismatch is encountered. MCF labels any type mismatch as a type conflict that needs to be fixed. Furthermore, MCF only supports a fixed set of C++ and SystemC types and does not support composite and user defined types. On the contrary, composite and user defined types are used extensively in Bluenose II. Type handling in Bluenose II is discussed in detail in Section 3.6.

Besides its use in the specification of other domain specific languages such as the ones used in MCF and MMV, the direct application of UML in ESL design has also been studied by Mueller *et al.* [34]. The UML notation defines a set of diagrams that are used to describe the structure or behaviour of a system [32].

Not all of UML's modeling capabilities are useful in hardware design. Most tools support a subset of UML that is applicable to the respective domains that are being modeled. A tool may further customize UML for its domain through profiling. From the perspective of a metamodel designer, using a DSL design tool such as GMF may be more straightforward than creating UML profiles and defining constraints on them.

Architecture description languages (ADL) have been used for fast prototyping, design verification as well as hardware synthesis [35, 36]. An ADL models the structure, behaviour or both of a hardware architecture. Mishra *et al.* used the EXPRESSION ADL to capture the structure and behaviour of processor architectures [35]. The EXPRESSION ADL allows the user to model processor specific elements such as a pipeline connection and datapath which transfer instruction and data respectively. EXPRESSION has a Lisp like syntax. The structure is specified as a net-list while the behaviour is captured through the description of the instruction set. The mapping between the structure and the behaviour is also captured in EXPRESSION. From the description in EXPRESSION, its associated tools generate a software toolkit which includes architecture-sensitive compiler and simulator. The software toolkit facilitates the design space exploration. However, there is no support for RTL generation from EXPRESSION. Although the pipeline modeling capabilities between *PipeNet* and EXPRESSION are similar, EXPRESSION focuses on processor design while *PipeNet* and Bluenose II have a broader scope that covers general hardware design. *PipeNet* has a tighter link with the actual implementation (each block in the model is associated with an implementation) than EXPRESSION. In this regard, the nML ADL which has been extended by Target Compiler Technologies [36] is more similar to *PipeNet*. This version of nML can be used for hardware synthesis. It is in our future work to provide simulation support for *PipeNet* similar to the one for EXPRESSION by connecting Bluenose II to third party tools.

The *X* language describes the structure of the system and is used in the Auto-Pipe framework. The framework facilitates the exploration of a design space which may include heterogeneous devices such as processors and FPGAs [28]. *X* supports a hierarchical description of blocks. In *X*, a block may be directly tied to an implementation which may be specified in C and/or VHDL. The Auto-Pipe framework generates code for communication between blocks which are written in either C or VHDL. On the other hand, Bluenose II provides the designer a library of specialized blocks to describe the control circuitry of the hardware and generates VHDL code based on the structure described with *PipeNet*. Although the Auto-Pipe framework supports the description of the pipeline topology, this support is limited to the description of the connectivity of the components.

Bluespec is a high level language based on guarded actions [37]. Its approach is referred to as operation-centric which contrasts the RTL description which is state-centric. The behaviour of the hardware is specified through a set of rules which change the states of the hardware when their conditions are met. The application of each rule is assumed to be atomic. The operation-centric model underlying Bluespec nondeterministically selects one of these enabled rules to execute in one step. As a result, a sequence of atomic application of state transition rules is produced. This atomic and sequential execution semantics allows the designer to specify each rule as if the rest of the system were frozen. While this model eases the verification of the design, its literal translation will create an extremely slow hardware. A special compiler is used to synthesize an RTL description which allows non-conflicting rules to execute in parallel. In other words, it generates the arbitration and control logic to coordinate access to shared resources. Bluespec has Haskell-like syntax and Haskell's functional language features are leveraged. Bluespec requires the user to design hardware with a different mindset [38]. In the initial design, the Bluespec can be translated directly from the instruction set architecture manual. However, the time gained may be offset by the time spent to optimize the design through manipulating the rules to generate RTL with competitive performance. *PipeNet* is different from Bluespec in a number of ways. *PipeNet* offers the designer a library of components to describe the arbitration and control logic. The structures and behaviours of these components are well understood.

Ptolemy II is a modeling tool that has a long history [39]. It supports a rich set of computation models. It is especially designed to handle the composition of heterogeneous models. Embedded system is an example where multiple models are often needed to describe its total behaviour. Although these models of computation can be used to specify hardware and pipeline design and RTL can be generated from these models, typical hardware engineers may not be familiar with their syntaxes and semantics. On the other hand, the *PipeNet* model is intuitive to designers because it is expressed in terms of hardware and pipeline concepts.

The frameworks described in this section automate different aspects of the design and verification processes. At one end of the spectrum, the glue logic between the structural components are generated automatically (MCF and the *X* language); at the other end of the spectrum, almost the entire design is automatically generated from the instruction set architecture (Bluespec). Bluenose II strikes a middle ground between the glue logic auto-generation and full design automation. In Bluenose II, the datapath implementation has to be provided by the designer and the control circuitry is built with the components selected from a library provided by Bluenose II, called *PipeLib*. These components correspond to the ones described in Section 2.1. These components not only describe the structure of the pipeline, they also describe its behaviour. The ability to characterize a pipeline with the components is not present in the frameworks that automate the generation of glue logic. The components provided by these frameworks generally represent hardware components such as the splitter in MCF which is synthesized into a multiplexer in RTL. Another advantage of Bluenose II's "middle" approach is that it is easier for the designer to predict the hardware that will get generated from the model. Reusable hardware components from *PipeLib* provide tight links between the model and the implementation (but the model remains implementation independent). This knowledge helps the designer to optimize the performance if needed. Our case study (Section 6.3.1) indicated that the performance of the code generated by Bluenose II is comparable to that of the handcrafted code.

2.7 Summary

In this chapter, we presented the pipeline model used in Bluenose II which is based on Aagaard's verification framework. We introduced the Model Driven Architecture (MDA), a model driven development framework proposed by the Object Management Group (OMG). In the process, we explained related concepts such as metamodel and domain specific language. The model driven development approach was adopted through the use of Eclipse Graphical Modeling Framework (GMF). Following the description of GMF, we described HOL, a high order logic theorem prover that was connected to Bluenose II to be used as a verification environment. Lastly, we related and contrasted Bluenose II to related work after a comparison with its predecessor, Bluenose I.

In the rest of this thesis, we describe how pipeline design and verification are carried out in Bluenose II and how their respective features were implemented (Chapter 3 and Chapter 4). These features were implemented using data structures that represent the pipeline model. These data structures and their implementation are detailed in Chapter 5. Finally, a case study based on the implementation of an OpenRISC processor is presented in Chapter 6 to demonstrate the value of Bluenose II.

Chapter 3

Pipeline Design with Bluenose II

Traditionally, pipeline designs are sketched out in block diagrams with blocks representing pipeline stages and arrows representing different types of connections. The design is then refined and implemented using hardware description languages (HDLs) such as VHDL and Verilog. The design described in a hardware description language is then simulated, synthesized, placed and routed, and analyzed for timing. Depending on the methodology, verification on the design may happen at different points of the design flow. Under time-to-market pressure, it is important to not only get the designs out quickly but also to get the correct designs. An incorrect design will not only cost millions of dollars to respin or recall, it will also damage the company's reputation. This chapter will show how Bluenose II helps designer to quickly produce pipeline designs that are functionally correct.

The chapter is organized as follows: An overview of Bluenose II is given in Section 3.1. Section 3.2 describes *PipeNet*, the pipeline metamodel used in Bluenose II. Section 3.3 puts the different modules of Bluenose II in the context of the design flow. The rest of the chapter is devoted to the details of the individual modules in Bluenose II. Section 3.4 describes the graphical user interface of Bluenose II. *PipeLib*, a cell library used in VHDL code, is presented in Section 3.5 which leads to the discussion of the process of generating VHDL code in Section 3.6.

3.1 Overview

The mathematical model of pipeline described in Section 2.1 guides the pipeline designers in their designs by providing them a template for control circuitry (how the parcels are requested and accepted in each pipeline stage); thereby allowing the designers to focus on the datapath and reducing the design time overall. Bluenose II takes a step further by taking the design and generates the corresponding VHDL code and verification scripts.

Continuing Bluenose I's effort to simplify the creation of pipelined circuits, Bluenose II aims to improve upon the subgoals set by Bluenose I [12].

1. The tool should provide a pipeline designer with a quick, easy and intuitive approach to manipulate the mathematical model described in Section 2.1. This translates to a tool that allows a pipeline designer to quickly and easily create and modify the design of a pipeline.

2. The tool should be able to generate synthesizable VHDL code from the user's pipeline design. The user should not need to edit the generated code before using the next standard tool in the design flow process.

3. The generated VHDL code should not suffer any significant decrease in performance (or increase in area) when compared to a custom pipelined circuit.

4. The tool should allow the user to describe a wide range of pipelined circuits.

5. The tool should provide the user with a convenient, logical, and efficient manner to resolve pipeline hazards.

We will see how these goals are met in Bluenose II as we describe the different modules in the following sections. Figure 3.1 shows the modules that make up Bluenose II. Table 3.1 describes the specific functionality performed by each module. Since HOL scripts are used in the pipeline verification, the HOL generation module is described in Section 4.2. As we present the modules, we will also discuss the improvement in Bluenose II from Bluenose I. To put the modules in the context of the design flow, Section 3.3 describes the typical design flow in Bluenose II.

3.2 PipeNet

PipeNet is a pipeline metamodel for structural hazards based on Aagaard's verification framework [11]. Figure 3.2 shows a simplified version of *PipeNet* in class diagram notation. The operations of the classes are not shown. The numbers on the two ends of each relationship (composition or association) indicate the numbers of instances of each class that are allowed to participate in the relationship. For example, a *PipeStage* object must have only one *Arbiter* but it can have zero or more instances of *Register*. The multiplicities of instances of each class in all the relationships are based on the pipeline model from Aagaard's verification framework as explained in Section 2.1.

A pipeline model in *PipeNet* starts with a *Pipeline* element. A *Pipeline* consists of *PipeStage*s, *MemoryArray*s and *Custom Block*s. Each *PipeStage* is composed of components that describe its datapath and control circuitry. The *Datapath* of a *PipeStage* may include a *Pipeline* element to describe a hierarchical pipeline design. All the elements in *PipeNet* contain *Port*s and *Generic*s. Each connection between two *Port*s is represented by a *Connection* element. Connections are not explicit in Aagaard's pipeline model. The decision to model connection this way is discussed in Section 5.1.

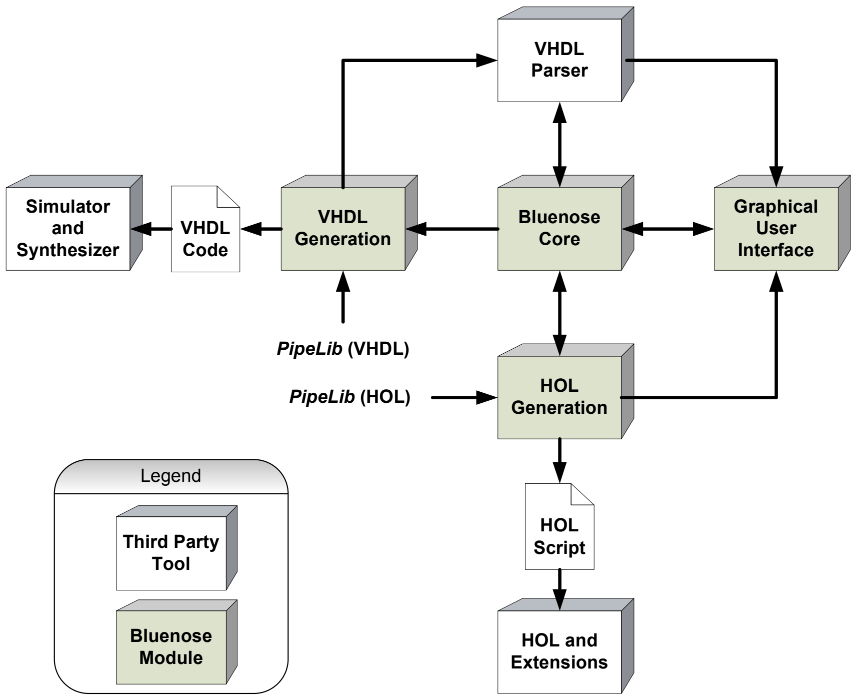


Figure 3.1: The main modules of Bluenose II

Table 3.1: Description of main modules in Bluenose II

Bluenose Core	Contains the main data structures and the functions for manipulating those data structures. Interacts with all other modules.
VHDL Parser	Third party parser included in the Signs plug-in [40]. It is used by Bluenose II to parse a VHDL entity and convert it into the internal data structures used by the core. The Signs module also provides an VHDL editor used by the graphical user interface to display VHDL code. Discussion on the Signs module will be limited to where it is due.
VHDL Generation	Converts a Bluenose pipeline into synthesizable VHDL code.
Simulator and Synthesizer	External tools for simulating and synthesizing the VHDL code.
PipeLib (VHDL and HOL)	A set of libraries written in VHDL and HOL, based on the representation of Section 2.1, used during the VHDL code and HOL generation process.
HOL Generation	Converts a Bluenose pipeline into HOL script.
HOL and Extensions	The high order logic theorem prover and connections to other verification tools through HOL.
Graphical User Interface	The graphical interface through which the user interacts with Bluenose II. Pipeline design is saved from and loaded into the graphical interface.

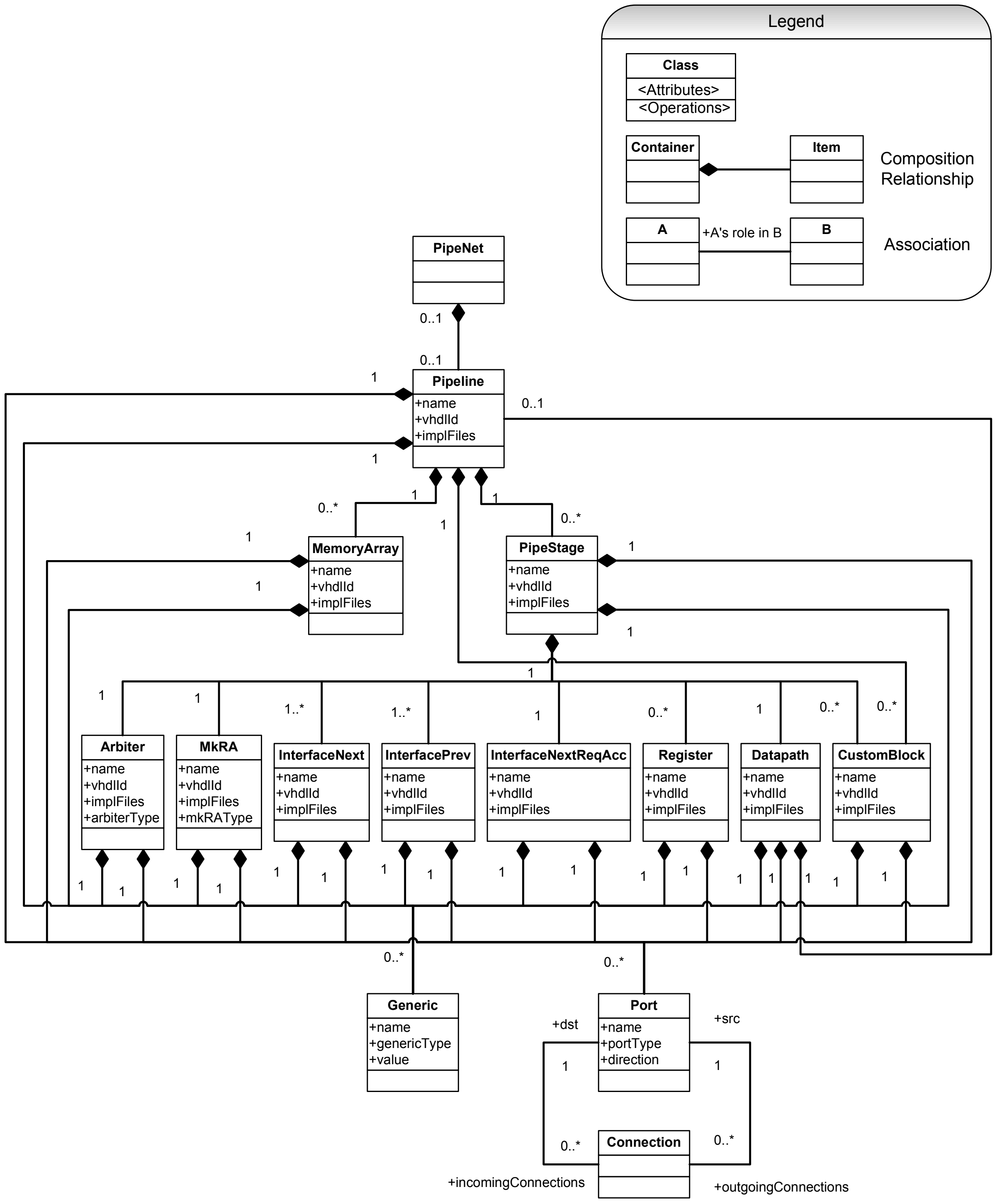


Figure 3.2: PipeNet

A well defined pipeline metamodel provides enough structure and constraints that there is benefit and enough freedom for designers to create the pipelines that they want. The pipeline metamodel was implicit in Bluenose I. For example, it was implied that a composition of Aagaard's framework parameters such as *Arbiter* and *MkR/A* represents a pipeline stage and the composition of the stages represents a pipeline. The implicit pipeline metamodel in Bluenose I can be considered a prototype of *PipeNet*. Higgins showed that the implicit pipeline metamodel used in combination with a graphical user interface simplified the design and verification of pipelines [12].

By identifying and making explicit the important elements in the pipeline metamodel, we capture information that we can use to reason about the design. For example, *PipeNet* shows that a *PipeStage* contains only one *Arbiter* or conversely, a *PipeStage* with more than one *Arbiter* is not allowed in the pipeline model. Constraints beyond the ones captured by the metamodel can be added through languages like OCL [33] if the metamodel is expressed in a MOF-based language. An example is the constraint of not allowing more than one connection to the *PipeStage* when a *Degenerate Arbiter* is selected. This is possible because we can identify *PipeStage* and *Arbiter* in the model. In addition to making *Pipeline* and *PipeStage* explicit, we added *MemoryArray* and *Custom Block*, which did not exist in Bluenose I, to further increase the expressiveness of *PipeNet*. Although there is no mathematical model or standard implementation file associated with these blocks, we want to enforce the differentiation between the different types of blocks (memory, pipeline stage and components) as a guide towards a correct pipeline design.

While it is important for the pipeline metamodel to capture the constraints that are present in a pipeline, it should also be expressive enough for the designers to create the design that they want. Aagaard was able to characterize a number of commercial microprocessors with his verification framework which *PipeNet* is based on [11]. By transitivity, this indicates *PipeNet*'s capability to express new designs. Furthermore, Higgins created a reasonably complex pipeline [12] with the "*PipeNet* prototype" so we expect that *PipeNet* can be used to express a wide range of pipelines. In addition to pipelined circuits, *PipeNet*'s hierarchical block design should be able to support circuits of other topologies, since any circuit can be described with blocks and connections between blocks. This hierarchical block design is inspired by the modular nature of the mathematical model.

As explained in Section 2.2, since a model represents the concepts from a domain, therefore the metamodel for that model provides a modeling language for that domain. In this way, a metamodel can also be considered a domain specific language (DSL). In the case of *PipeNet*, it is a language for describing the structure and behaviour of a pipelined circuit. Benefits that are associated with domain specific languages such as domain specific error checking, can be gained with a tool designed for *PipeNet*, that is Bluenose II. Through *PipeNet* and Bluenose II, we offer a means to manipulate the mathematical model described in Section 2.1. In the rest of this chapter, we will describe the pipeline design process with *PipeNet* in Bluenose II.

3.3 Design Flow in Bluenose II

This section outlines the design flow of Bluenose II. As we list the major steps, we also name the associated modules to show how they fit in the broader scheme of things.

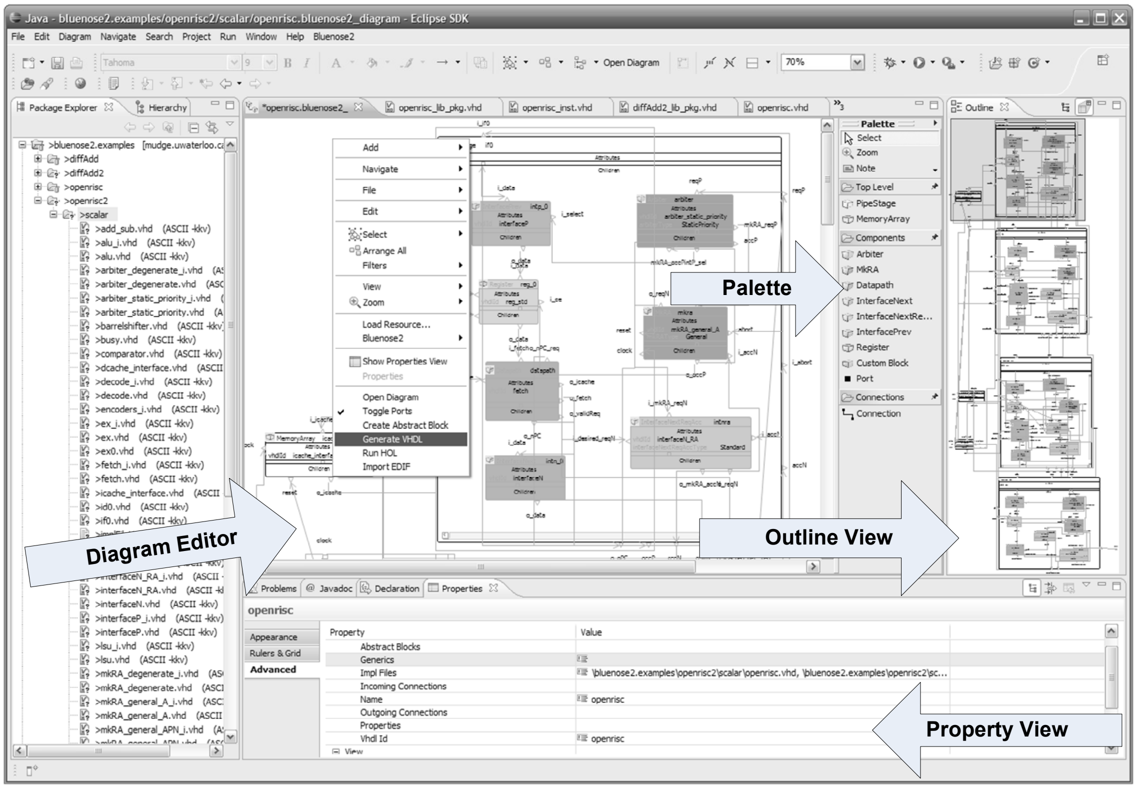
Before the pipeline designer uses Bluenose II, we expect her to have already partitioned the work into pipeline stages and to have an understanding of the behaviour of each pipeline stage, *e.g.*, an input signal from one stage has priority over one from another stage.

Graphical User Interface: Through the palette on the graphical interface (shown in Figure 3.3), Bluenose II provides a library of components from which the designer selects, drags and drops onto the diagram editor. A stage template with standard components is also provided to the designer through the palette. After the designer has placed the pipeline stages and components on the diagram editor, she then connects them through their ports. The behaviours of the elements (*e.g.*, type of *Arbiter*) in the pipeline model are configured through the Property View of the graphical interface. The designer can save the diagram at any time during the process of the design capture. For the *Datapath* component, the designer has to associate an implementation written in VHDL with the component by setting its *Impl Files* property. This can be done in two ways. The designer can set the *Impl Files* property of the *Datapath* component with the VHDL file in the Property View. Or, the VHDL file can be dragged from the package explorer (shown on the left of Figure 3.3) and dropped onto the *Datapath* component directly.

VHDL Parser: When a VHDL file is dragged and dropped onto a component, the VHDL entity is automatically parsed by the *VHDL Parser* to extract the generics and ports for the component. The port type information is also extracted in this step while the port types for the standard components (*MkR/A*, *Arbiter etc.*) are pre-defined. The *VHDL Parser* comes from a third party plug-in called Signs [40], Bluenose II extracts information from the resulting abstract syntax tree.

VHDL Generation and *HOL Generation*: The last steps are to generate VHDL and verification scripts from the pipeline model. During the generation processes, standard components are instantiated from the respective *PipeLib*'s. The graphical user interface displays VHDL code in the VHDL editor provided by the Signs plug-in, which has features like syntax highlighting [40].

In the following sections, we will discuss the individual modules in details.



Property	Value
Abstract Blocks	
Generics	
Impl Files	\bluenose2.examples\openrisc2\scalar\openrisc.vhd, \bluenose2.examples\openrisc2\sc...
Incoming Connections	
Name	openrisc
Outgoing Connections	
Properties	
Vhdl Id	openrisc

Figure 3.3: Bluenose II screenshot with labels

3.4 User Interface

In this section, we give an overview of the user interface and discuss some of the strategies that we employed to make it user friendly. Refer to Figure 3.3 for the different graphical interface parts mentioned in this section.

Essentially, we want to make Bluenose II a tool that allows the designer to visualize and manipulate graphically the mathematical model described in Section 2.1. *PipeNet* provides the syntax to describe the mathematical model and Bluenose II implemented this syntax. In Bluenose II, graphical representations of the elements in *PipeNet* are selected from a palette. The pipeline designer creates her pipeline design by creating pipeline stages, dragging and dropping components from a palette, configuring the components and finally connecting the individual components and stages through the ports.

User friendly features are implemented in Bluenose II's user interface to speed up the design process. In addition to the components mentioned in Section 3.2, the palette also provides a pipeline stage template which contains standard components and connections. The graphical interface also provides a means for the designer to edit component properties, for example, to select the type of an *Arbiter*. The behaviour of a pipeline stage can be easily modified through changing the types of its components. The ease of modifying the behaviour of a pipeline stage facilitates design exploration.

We aim to make Bluenose II's graphical interface as intuitive as possible to make it easier to capture the designer's intent. The use of block and arrow figures to represent pipeline stages and various kinds of connection is natural to pipeline designers. The block representation of the components also abstracts away low level mechanisms. For example, at the design entry phase, it is enough to understand the purpose of a *StaticPriority Arbiter* instead of its detailed implementation. Furthermore, we made an effort to create a graphical representation that resembles a hardware schematic so a Bluenose model can double as a design document.

We also strive to develop Bluenose II into a "full-featured" graphical editor with features such as saving and loading designs, file management, save to image, zoom in/out *etc.* that designers expect from commercial visual development tools. We have harnessed third party tools (the Eclipse platform, GMF's graphical editor code generation and Signs) to provide and implement these functionalities while we focus on designing the pipeline metamodel and implementing the pipeline logic and Bluenose core. Development with these tools and the implementation of Bluenose core are discussed in Chapter 5.

3.4.1 Reducing Clutter

The design area (diagram editor) of Bluenose II by default shows a flattened view of the pipeline with all the stages and all the components inside them. This approach has both an advantage and a disadvantage. The advantage is that the designer can view the entire design without traversing the different hierarchies, the disadvantage is that the design area becomes cluttered, especially with all the ports and connections between components and stages. We improved the clutter by reducing connections with the use of VHDL record types, which led to changes in *PipeLib* (Section 3.5) and the VHDL generation (Section 3.6).

Essentially, each connection in the Bluenose II model represents a signal assignment. By grouping stage inputs/outputs into record types, they are now only represented as one connection in the Bluenose II model. The use of VHDL record types has led to some challenges in the VHDL generation which is described in Section 3.6. The use of record type has led to some desirable side-effects such has the ease in updating stage inputs and outputs. Instead of creating a new port and a new connection (or removing a port and a connection) in the Bluenose II model, the designer only has to add the new input/output as an element in the record type definition and modify the conversion function (described in Section 3.6), which involves only a few lines of VHDL. We have also built in various features to allow graphical abstraction to "hide" the clutter when it is not needed. Designer can collapse the compartment of a block to abstract away the children blocks when they want to only see the high level view of the pipeline. Designer can also see the high level view of the pipeline in the outline view.

3.4.2 Changes from Bluenose I

The graphical user interface has undergone significant changes from Bluenose I. Although both Bluenose I and Bluenose II use hierarchical blocks to represent pipelines, Bluenose II has a flattened view of the pipeline whereas in Bluenose I, hardware designer has to traverse through the hierarchies to view the different levels of abstraction, from the view of an entire pipeline to the content of individual pipeline stages. Similarly, the connection in Bluenose II represents individual signal/port assignment whereas in Bluenose I, a connection between two blocks (component to component, pipeline stage to component, stage to stage) represents all the signal/port assignments between the two blocks. These changes are applied with the reasoning that if no information is lost in the representation, we opt for the simpler implementation. The "flattened" view in Bluenose II can be cluttered in more complex design so we have implemented different ways for user to abstract lower level details and make use of VHDL record types as described in previous section.

In our effort to make Bluenose II into a "full-featured" graphical editor, we have added more graphical user interface features that did not exist in Bluenose I, such as file management, save to image (for printing), zoom in/out, auto-placement of blocks, auto-routing of connections *etc.* We have also incorporated the VHDL editor from Signs into Bluenose II with user friendly features such as syntax highlighting for viewing VHDL codes.

3.5 VHDL PipeLib

In this section, we present *PipeLib* (VHDL) and discuss the changes that we made to the original version for it to work with the updated user interface.

PipeLib is a library of the components described in Section 2.1 written in VHDL. It was created by Higgins to be used with Bluenose I [12]. These components are instantiated in the VHDL code generated by Bluenose I according to the user's design. Different types of the same component have the same VHDL entity declaration (hence the same port interface list) but different VHDL architecture bodies. This allows the hardware designer to make changes to the stage's behaviour by simply changing the type of the component without having to change the wiring/connections. This is made possible by keeping the interfaces of the components of the same type (*e.g.*, a *Degenerate Arbiter* is the same type as a *StaticPriority Arbiter*) the same. Bluenose II has reused the implementation of most of the components in the library except for a few changes described in the following section.

3.5.1 Changes from Bluenose I

The original *PipeLib* uses generics to make the components more customizable. Generics act as placeholders in the entity declaration and architecture body to allow the specification of values such as the widths of the array signals and ports to be delayed until the instantiation of the entity. While it is relatively straightforward to calculate and specify these values with the original components, it requires some thought to be used with VHDL record types (Section 3.4.1). In the rest of this section, we describe the original components and explain why it is more difficult to calculate and specify values for generic constants when VHDL record types are involved.

In Bluenose I and the original *PipeLib*, all the signals and port are of the *standard-logic vector* type (*std_logic_vector*), an array type for a vector of *standard-logic* values. Therefore, using the values of generic constants to calculate the size of any array is straightforward. For each pipeline stage, the designer only has to specify the number of inputs, the number of outputs and the data width. These values are then passed to the generics of the components. Therefore during the VHDL generation, the instantiation of the components inside a pipeline stage is fully automated.

The original *Interface Prev* component shown in Listing 3.1 is an example of how generics are used to specify the sizes of array ports and the values for generate parameters. The number of inputs to the stage containing the *Interface Prev* and the data width (line 3 and 4) are passed to the component and the signals of appropriate sizes and appropriate structure are generated (lines 6-8, 15 *etc.*). Generics allow the components in *PipeLib* to be re-used for different data widths and numbers of inputs and outputs.

Listing 3.1: VHDL code for *Interface Prev* component

```
 1 entity interfaceP is
 2   generic (
 3     numInputs : Integer := 1;
 4     wordSize : Integer := 1);
 5   port (
 6     i_data : in std_logic_vector(((numInputs*wordSize)-1) downto 0);
 7     i_select: in std_logic_vector((numInputs-1) downto 0);
 8     o_data : out std_logic_vector((wordSize-1) downto 0));
 9 end interfaceP;
10
11 architecture main of interfaceP is
12   signal data_and : std_logic_vector(((numInputs*wordSize)-1) downto 0);
13   signal data_or : std_logic_vector(((numInputs*wordSize)-1) downto 0);
14 begin -- main
15   single: if numInputs = 1 generate
16     o_data <= i_data;
17   end generate single;
18
19   dual: if numInputs = 2 generate
20     o_data <= i_data((wordSize-1) downto 0)
21                      when i_select(0) = '1' else
```

```
            i_data(((2*wordSize)-1) downto wordSize);
  end generate dual;

  multiple: if numInputs > 2 generate
    gen1: for i in 0 to (numInputs-1) generate
      gen2: for j in 0 to (wordSize-1) generate
        data_and(i*wordSize + j) <= i_data(i*wordSize + j) and i_select(i)
            ;
      end generate gen2;
    end generate gen1;
    gen3: for i in 0 to (numInputs-1) generate
      gen4: if i /= 0 generate
        data_or((((i+1)*wordSize)-1) downto (i*wordSize)) <=
          getArrayElement(data_or, i-1, wordSize) or
          getArrayElement(data_and, i, wordSize);
      end generate gen4;
      gen5: if i = 0 generate
        data_or(((wordSize)-1) downto 0)<=getArrayElement(data_and,i,
            wordSize);
      end generate gen5;
    end generate gen3;
    o_data <= getArrayElement(data_or, numInputs-1, wordSize);
  end generate multiple;
end main;
```

With the use of user defined record types in Bluenose II, we are faced with two problems:

1. We can no longer directly connect the stage inputs/outputs (in user defined record types) to the standard *PipeLib* components (ports in *std_logic_vector*). Connecting these ports together will cause type conflicts. The original *PipeLib* uses *std_logic_vector* because the VHDL generation module of Bluenose I transforms all the ports and connections into *std_logic_vector*s.

2. Each stage input/output is now an element in a record type as opposed to individual signals as in Bluenose I. *PipeLib* components such as *Interface Prev*, *Register* and *Interface Next* used to be instantiated per stage input or output; the sizes of their array ports and signals could be specified easily using the values of generic constants. For Bluenose II, we have to consider whether a modification to *PipeLib* is required.

Our solution is to auto-generate type conversion function templates for the designer to implement and we modified the *PipeLib* components to use constrained multi-dimensional arrays. The conversion functions are for converting between the user defined record types and the types used in the *PipeLib* components. The generation of conversion function templates is discussed in Section 3.6. As described below, we arrived at the decision to use constrained multi-dimensional arrays from a series of discoveries in the hardware description language.

Multi-dimensional Array vs. Array of Arrays

The use of VHDL record types has prompted a modification of *PipeLib*. Upon a closer look, it was observed that the logics of most of the components could remain the same. It was the entity declarations of the components that required modification in order to be connected to signals of user defined record types without conflict. To be more specific, the port types in the entity declarations had to be made compatible with the user defined record types. When deciding on the right port type, there were two points that we kept in mind:

1. The re-design of the *PipeLib* components should simplify their implementation.
2. With the original design of *PipeLib*, the designer has to enter only three values for each pipeline stage (data width, number of inputs and number of outputs). The new design should not require the designer to enter more values than three or do any extra work.

A reason that it is more difficult to calculate values such as the data width when user defined VHDL record types are involved is simply that, we cannot predict how the user is going to define the record type. Therefore, we began by trying unconstrained array ports since this type of port takes on the size of the signal assigned to it. In a VHDL entity declaration, the sizes of array ports do not necessarily have to be constrained. It is legal to have an unconstrained array port in the entity declaration; the size is to be determined by the signal associated with the port during the instantiation. Listing 3.2 shows an example where unconstrained array ports (line 3 and 4) are associated with constrained array signals (line 14 and 15). The *standard-logic vector* type (*std_logic_vector*) is an array type for a vector of *standard-logic* elements.

Listing 3.2: Unconstrained array ports in the entity declaration and instantiation

```
1  entity unconstrainedEntity is
2    port (
3      i_data : in std_logic_vector;
4      o_data : out std_logic_vector;
5  end unconstrainedEntity;
6  ...
7  -- within another architecture
8  signal data_in : std_logic_vector(31 downto 0);
9  signal data_out : std_logic_vector(7 downto 0);
10 begin
11   ...
```

```
  -- Unconstrained ports take on the sizes of the array signals
  ue : entity work.unconstrainedEntity
    port map( i_data => data_in,
              o_data => data_out );
  ...
```

Originally we thought the record types would automatically be flattened into *std_logic_vector*s during the elaboration (much like the struct data type in the C programming language). Therefore, we made the array ports in the *PipeLib* components unconstrained and expected that they would simply take on the sizes of the flattened record types. It turned out that record types are not automatically flattened even if all the elements inside them are *std_logic_vector*s. We solved this problem by generating the conversion function templates, as a result we can assume that signals associated with the inputs (outputs) of the *PipeLib* components will be converted from (to) the user defined record types into (from) the types used by the *PipeLib* components. A type conversion function is a subprogram in VHDL that converts its input of a certain type into another type. We generate a conversion function template complete with the parameter and return type so that the designer only has to fill out the body of the function. The generation of conversion function template is detailed in Section 3.6.

In the original *PipeLib*, all the components were implemented using *std_logic_vector*s which is an one-dimensional array of *standard-logic* values(*std_logic*). For certain components in *PipeLib*, using two-dimensional arrays might simplify their designs. *Interface Prev* is an example of a component which may benefit from two-dimensional arrays. *Interface Prev* is essentially a multiplexer for selecting a stage input when there is more than one upstream stage connected to the stage containing the *Interface Prev* (see Figure 2.1). In *Interface Prev*, a two-dimensional array can be used to access each stage input as an element instead of calculating the indexes of each bit in the one-dimensional array as shown in Listing 3.1. A reason that all the components were implemented with one-dimensional arrays was simply that, multi-dimensional arrays were not supported by the synthesizer used during the development of Bluenose I. During this re-design of *PipeLib* components, we wanted to exploit the features of the synthesizer that were available to us. This led us to investigate the similarities and differences between multi-dimensional arrays and arrays of arrays.

Although conceptually, the structures of a multi-dimensional array and an array of arrays are similar, declarations and the ways elements are accessed are very different for the two array types. A multi-dimensional array type is declared by specifying a list of index ranges. One or more indexes in a multi-dimensional array type can remain unconstrained during declaration, whereas the type of the element of an (constrained or unconstrained) array type cannot be an unconstrained array type [41]. However, elements cannot be sliced in multi-dimensional arrays as in arrays of arrays. The elements in a multi-dimensional array can only be accessed element by element, with all indexes specified. Listing 3.3 and Listing 3.4 illustrate the differences between the two array types. In both listings, the VHDL syntax for type declaration is shown, so is the syntax to access elements 6 to 2 in row 1 in the arbitrary arrays.

Listing 3.3: VHDL code for multi-dimensional array

```
-- Array type declared by specifying a list of index ranges
-- One or more indexes can remain unconstrained during declaration
type matrix_type is array (natural range <>, natural range <>) of
   std_logic;
...
signal matrix : matrix_type(255 downto 0, 7 downto 0);
signal temp : std_logic_vector(4 downto 0);
...
-- Slicing not supported
-- Elements can only be accessed one by one, with all indexes specified
temp(0) <= matrix(1,2);
temp(1) <= matrix(1,3);
...
temp(4) <= matrix(1,6);
```

Listing 3.4: VHDL code for array of arrays

```
-- The element of an array type cannot be an unconstrained array type
-- Shown here, although the outer array type can be unconstrained, its
   element type must be constrained
type two_d_array_type is array (natural range <>) of std_logic_vector(7
   downto 0);
...
signal two_d_array : two_d_array_type(255 downto 0);
signal temp : std_logic_vector(4 downto 0);
...
-- Slicing supported
temp <= two_d_array(1)(6 downto 2);
```

In summary, there were three choices for the port types of the *PipeLib* components: unconstrained multi-dimensional array, constrained multi-dimensional array and constrained array of arrays. The advantages and disadvantages of each type are further analyzed below.

A disadvantage with multi-dimensional array is that we cannot access consecutive elements as a slice. An advantage is that we do not have to specify the index ranges if the multi-dimensional array is unconstrained. As a result, the designer does not have to enter any values for the generic constants. The unconstrained array ports in the *PipeLib* components can simply take on the sizes of the signals associated with the ports during the components' instantiation. This further automates the pipeline design process. However with the array ports unconstrained, if the ports were not assigned properly, these errors might not be detected during elaboration. These errors may go undetected because the sizes of the signal and the port are not checked and the port simply takes on the size of the signal. These errors can be caused by the misunderstanding of how the ports should be assigned. It was our experience that an error was only caught during simulation when the hardware did not behave as expected.

With the use of array of arrays, if the signals associated with the ports in the instantiation were of different sizes than the formal ports, the analyzer will signal the error early in the design process (*i.e.*, during elaboration), making it easier to correct. As suggested by Ashenden, whenever the sizes of different array ports of an entity are related, generic constants should be considered to enforce the constraint [42].

Based on our experience with unconstrained types, we conclude that constrained types provide more error checking than unconstrained types. The errors also make it easier to locate the sources of some problems (*e.g.*, a signal being assigned to the port was not instantiated with the correct size) while these problems may go unnoticed with an unconstrained type. However, we still need to decide which constrained type to use, constrained multi-dimensional array or array of arrays. We return to the second objective of the re-design of *PipeLib* components: the new design should not require the designer to enter more values than three or do any extra work.

An implementation that uses the *constrained* multi-dimensional array type and one that uses the array of arrays type will most likely require the same generic constants (number of inputs, number of outputs and data width of one element). However, the multi-dimensional array type provides more flexibility. We can leave the index ranges of the multi-dimensional array type unconstrained during the type *definition* and constrain these ranges accordingly for each port that uses the array type in the entity declarations of the components. For the array of arrays type, the range of the element type must be specified during the type definition.

Based on the analysis, we decided to use constrained multi-dimensional array types for the ports of the *PipeLib* components. Listing 3.5 shows a code fragment of the new implementation of *Interface Prev* (in contrast to Listing 3.1). *std_logic_matrix* in line 6 is our implementation of a two-dimensional array type based on the constrained multi-dimensional array type. We implemented a utility function, *to_std_logic_vector* (lines 16 and 20), to be used with the custom two-dimensional array type to access a "slice" of the array. The function helps us to achieve the same simplicity as using the array of arrays type.

Listing 3.5: Revised VHDL code for *Interface Prev* component

```
1 entity interfaceP is
2   generic (
3     numInputs : integer := 1;
4     dataSize : integer := 8);
5   port (
    i_data : in std_logic_matrix((numInputs-1) downto 0, (dataSize-1)
        downto 0);
    i_select: in std_logic_vector((numInputs-1) downto 0);
    o_data : out std_logic_vector((dataSize-1) downto 0));
end interfaceP;

architecture main of interfaceP is
  signal data_and : std_logic_vector(((numInputs*dataSize)-1) downto 0);
  signal data_or : std_logic_vector(((numInputs*dataSize)-1) downto 0);
begin -- main
  single: if numInputs = 1 generate
    o_data <= to_std_logic_vector(i_data, 0);
  end generate single;

  dual: if numInputs = 2 generate
    o_data <= to_std_logic_vector(i_data, 0) when i_select(0) = '1' else
              to_std_logic_vector(i_data, 1);
  end generate dual;
...
```

In this section, we presented *PipeLib*, a library of components written in VHDL based on the ones in Aagaard's pipeline model. The types used in the original *PipeLib* components are likely incompatible with the record types defined by the user in Bluenose II. As part of the effort to resolve the type conflicts, we rewrote the *PipeLib* components to use constrained multi-dimensional arrays. Although we did not eliminate the use of generics through our choice of array types as we had hoped, we eliminated the need for the designer to manually specify the values for the generics by using VHDL attributes. The technique is described in Section 3.6.

3.6 VHDL Generation

Bluenose II generates VHDL code based on the pipeline design entered by the designer. The VHDL generation functionality aims to integrate Bluenose II into existing design flows. The user first designs the pipeline and generates the corresponding VHDL code in Bluenose II, then uses any simulation and synthesis tools that she is familiar with to simulate and synthesize the design. This section outlines the VHDL code generation process and highlights the improvement in the generated VHDL code.

A pipeline model created in Bluenose II conforms to *PipeNet*. The model can be viewed as a tree with each node representing a block (pipeline, stage, memory array or component). Children of a parent node represent blocks that are instantiated in the block represented by the parent node. In other words, a parent block is implemented by its children blocks. A "leaf block" is a block that does not have any children because it has already been associated with an implementation (VHDL files). A tree view of a Bluenose II pipeline model is illustrated in Figure 3.4.

The VHDL generation module performs a depth-first traversal of the tree. It starts at the root of the pipeline model which is the pipeline block, then it keeps expanding the first child block in the list of children blocks that it encounters (in the case of the pipeline block, the children blocks can be pipeline stages, memory arrays and custom blocks) until it reaches a leaf block. Since a leaf block is associated with an implementation, there is no need to generate entity declarations or architecture bodies for these blocks. All there is to do with these blocks is to instantiate their associated entities in the enclosing architecture bodies when the processing backtracks to their parent blocks. In turn, when the processing returns to a parent block, entity declaration and architecture body are generated for the block in addition to its instantiation in its enclosing architecture bodies. The pseudo-code in Listing 3.6 shows the logic of the VHDL generation module. The VHDL generation module performs a depth-first traversal of the hierarchical blocks in the design recursively.

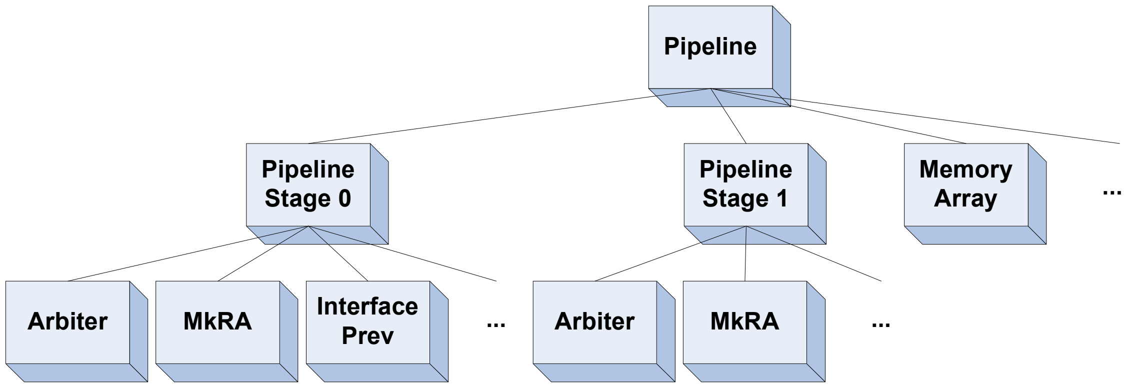


Figure 3.4: A tree view of a Bluenose II pipeline model

Listing 3.6: Pseudo-code for VHDL generation

```
generateVHDL( currentBlock )
{
  // getChildrenBlk returns a list of the children blocks in currentBlock
  // childrenBlockList is empty if currentBlock is a leaf block
  childrenBlockList = getChildrenBlk( currentBlock );

  if( childrenBlockList is not empty )
  {
    for each childBlock in childrenBlockList
    {
      generateVHDL( childBlock ); //recursive function
    }
    // Generate entity declaration for currentBlock
    traverse all the ports of currentBlock to generate entity declaration
         for currentBlock;

    // Generate architecture body for currentBlock
    Instantiate all the signals based on the connections between children
         blocks;
    Instantiate all the children blocks and assign signals to their ports
        ;
  }
}
```

The choice of the traversal algorithm is inherited from the previous version of the VHDL generation module. The earlier version of the module generated only one file that included all the VHDL code. For the VHDL file to be synthesizable, entities have to be defined (with entity declaration and architecture body) before they can be instantiated in another enclosing architecture body. In other words, the children blocks have to be declared before they can be instantiated in the parent block. The current version of the VHDL generation module generates one file per block (except "leaf block"). Therefore, it can be implemented with either breadth-first or depth-first traversal. For the rest of the discussion, we will only refer to the implementation with depth-first traversal.

3.6.1 Conversion Function Templates

During the generation of component instantiations and signal assignments, if the port types of the source and destination ports are different, a conversion function template is added to the generated VHDL library file for the designer to implement. Interfacing the stage input which is a user defined record type with the *PipeLib* components whose ports have pre-defined types, is an example where the types of the source and destination ports are different (Section 3.5). The conversion function templates allow us to auto-generate the VHDL code without knowing the content of the user defined record types. The designer does not have to edit the generated VHDL which can be error prone. The designer should already be familiar with the record types that she defined so implementing the conversion functions should not pose any problems. Figure 3.5 shows a stage with the components inside. The port names and their types (shown as *component name.port name: port type*) "along" the chain of components enclosed in the box on the left of the stage are shown on the leftmost of Figure 3.5. An arrow between two port indicates that the port types of the source and destination ports are different and require a conversion function. On the other hand, an equal sign indicates that the two ports types are the same and conversion function is not necessary. It can be seen that type clashes between the source and destination ports are common in a typical stage.

Listing 3.7 illustrates an example of a conversion function for converting from a record array type to a multi-dimensional array type. Bluenose II users will usually encounter this type of conversion functions in their designs. To transform an array of record type elements into a constrained two-dimensional array, the designer would generally need the following:

1. another conversion function to convert the array of record type elements into a *std_logic_vector* (which constitutes a row in the two-dimensional array) (line 5 and 10 in Listing 3.7)

2. nested loops to assign values from the *std_logic_vector* to elements in the two-dimensional array (lines 7 to 12) since slicing of a multi-dimensional array is not supported as explained in Section 3.5

3. VHDL attributes to specify the widths of the signals in the function so they are scalable and do not have to be hard-coded (lines 5, 7 and 8)

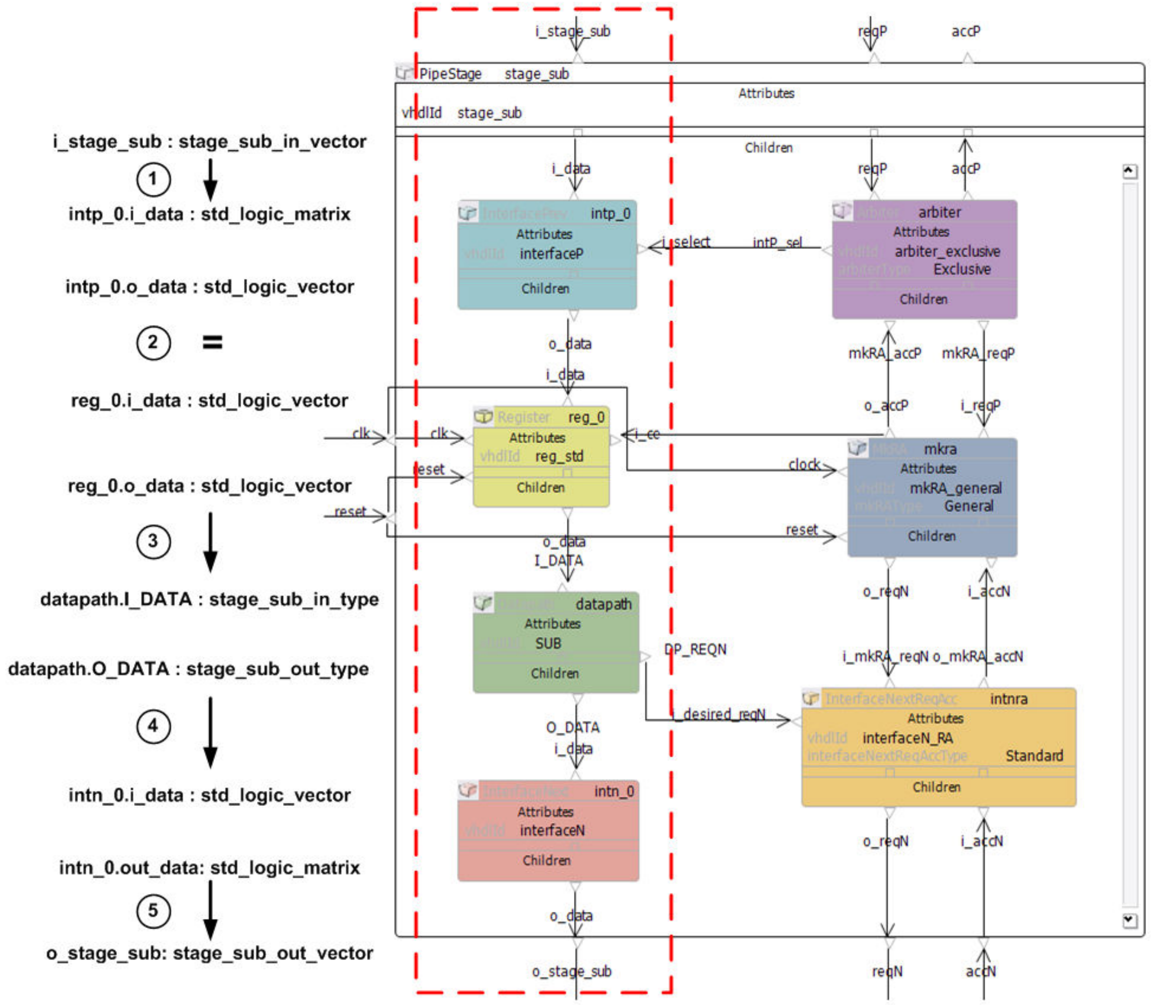


Figure 3.5: Conversion functions for a stage

Listing 3.7: Function for converting from record array type to multi-dimensional array type

```
function to_std_logic_matrix (
  input_vec : fetch_input_vector)
  return std_logic_matrix
is
  variable temp_matrix : std_logic_matrix(input_vec'range,
      to_std_logic_vector(input_vec(0))'range);
begin --to_std_logic_matrix
  for vec_slice_index in temp_matrix'range(1) loop
    for vec_bit_index in temp_matrix'range(2) loop
      temp_matrix(vec_slice_index, vec_bit_index) :=
        to_std_logic_vector(input_vec(vec_slice_index))(vec_bit_index);
    end loop; -- vec_bit_index
  end loop; -- vec_slice_index
  return temp_matrix;
end to_std_logic_matrix;
```

Conversion functions are used to automate the specification of values for the generics in the VHDL generation. To eliminate the need for designers to manually specify values for the generics, the VHDL generation module automatically generates VHDL attributes. As mentioned in Section 3.5, some of the *PipeLib* components use the values associated with the generic constants to constrain the sizes of their array ports and internal array signals. We assign the correct values by passing attributes such as array length as arguments to the generics in the generic association lists in the component instantiation statements. If the signals to which the attributes are applied are not *std_logic_vector*s, the signal is first converted into a *std_logic_vector* using the user defined conversion function. An example of generic values computed using conversion functions and attributes is shown in Listing 3.8 where *if0_o_nPC* and *if0_o_if0* are a record and an array of record respectively.

Listing 3.8: Generic map in entity declaration

```
if0_inst : entity work.if0
  generic map (
    numInputs => 2,
    numOutputs => 1,
    dataSize_in => to_std_logic_vector(if0_o_nPC)'length,
    dataSize_out => to_std_logic_vector(if0_o_if0(0))'length
  )
  ...
```

3.6.2 Changes from Bluenose I

Besides the additional use of various VHDL features (record type, conversion functions and attributes), the VHDL code generated by Bluenose II is made more concise than the code generated by Bluenose I in a number of ways. To illustrate the difference in the generated codes between Bluenose I and Bluenose II, code fragments from the same design are excerpted in Listing A.1 (in Appendix A) and Listing 3.9 respectively. The design is a three-stage pipeline which takes the absolute difference between two inputs and adds it to the third input, that is $|i1-i2|+i3$. The design results in an entity named *diffAdd*. A simplified block diagram of *diffAdd* is shown in Figure 3.6.

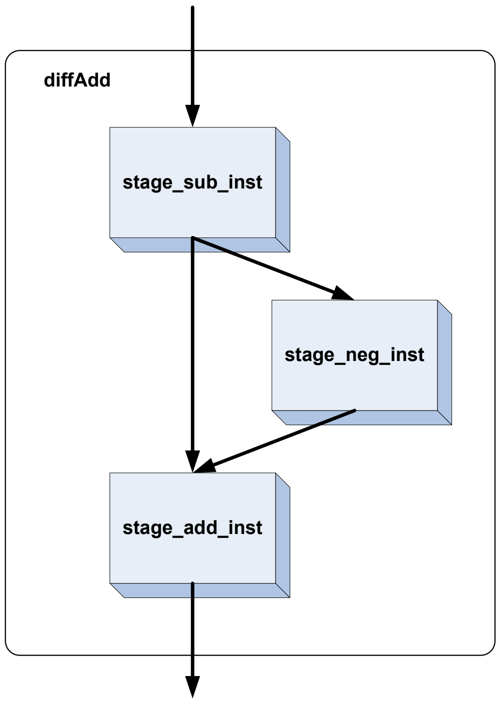


Figure 3.6: The diffAdd entity

The codes generated by Bluenose I and Bluenose II differ in the following ways:

1. *Bluenose II generates VHDL in the VHDL-93 syntax while Bluenose I in VHDL-87.*

   The VHDL-93 standard does not require components to be declared before their instantiation. Component declaration like the one in lines 3 to 19 in Listing A.1 is omitted. The VHDL-93 syntax of component instantiation is slightly different from the VHDL-87 syntax (contrast line 34 in Listing A.1 with line 9 in Listing 3.9).

2. *Indirect signal assignments are eliminated in the VHDL code generated by Bluenose II.*

   In Bluenose I, a signal is created for and assigned to each port (both input and output) of the instantiated component (lines 21 to 30 and lines 37 to 46 in Listing A.1). Ports are connected by assigning their respective signals to each other (lines 51 to 62 in Listing A.1). In Bluenose II, signals are created only for the output ports of the components (lines 3 to 5 in Listing 3.9). Ports are connected by associating these signals to the input ports of the components in the port maps of the corresponding component instantiations (lines 16 to 23 in Listing 3.9) In contrast to Bluenose I, Bluenose II generates signal assignments only for connections between the component ports and the ports of the enclosing architecture (line 26 in Listing 3.9).

Listing 3.9: DiffAdd code generated by Bluenose II

```
 1 architecture main of diffAdd is
 2 ...
 3   signal stage_neg_inst_accP: std_logic_vector((1-1) downto 0);
 4   signal stage_neg_inst_reqN: std_logic_vector((1-1) downto 0);
 5   signal stage_neg_inst_stage_neg_out: stage_sub_out_vector(0 downto 0);
 6 ...
 7 begin
 8 ...
 9 stage_neg_inst : entity work.stage_neg(main)
10   generic map ( numInputs => 1,
11   numOutputs => 1,
12   dataSize_in => to_std_logic_vector(stage_sub_inst_stage_sub_out(0))'
         length,
13   dataSize_out => to_std_logic_vector(stage_neg_inst_stage_neg_out(0))'
         length
14  )
15   port map (
16   clk => clk,
17   reset => reset,
18   reqP(0) => stage_sub_inst_reqN(0),
19   accP => stage_neg_inst_accP,
20   reqN => stage_neg_inst_reqN,
21   accN(0) => stage_add_inst_accP(0),
22   stage_neg_in(0) => stage_sub_inst_stage_sub_out(0),
23   stage_neg_out => stage_neg_inst_stage_neg_out
24   );
25 ...
26 accP <= stage_sub_inst_accP;
27 ...
28 end main;
```

The overall result of these changes from Bluenose I is that the VHDL code generated by Bluenose II is more concise and readable. Table 3.2 compares the numbers of lines of code for *diffAdd* generated in Bluenose I and Bluenose II. As seen in Table 3.2, the amount of newly generated code is reduced by more than half.

Table 3.2: Lines of code for the DiffAdd project

Category	Bluenose I	Bluenose II
Generated	1031	457
PipeLib components	423	427
Datapath in all stages	65	68
User defined library	N/A	164 (108 of which were generated)

We presented the VHDL generation module in this section. Continuing the theme of reducing clutter from Section 3.4.1, Bluenose II generates VHDL code that is more concise and therefore easier to "trace". Bluenose II's VHDL generation module supports VHDL record types by generating conversion function templates. By using an updated VHDL standard and removing indirect signal assignments, Bluenose II is able to generate concise VHDL code from the design.

3.7 Summary

In this chapter, we presented *PipeNet*, the pipeline metamodel that Bluenose II is based on. The different modules in Bluenose II that contribute to the pipeline design are discussed. The *Graphical User Interface* provides user an interface to interact with Bluenose II and a visual presentation of the pipeline model. To reduce the clutter in the graphical interface, we reduce the number of connections that need to be shown by using VHDL record types. A signal of VHDL record type is used to group individual signals together so they can be shown as one connection. Users only define record types for datapath signals and not the control signals. We also presented *PipeLib* in this chapter. *PipeLib* is a library of components based on Aagaard's pipeline model written in VHDL. The components are reused in the VHDL code generated by Bluenose II. Bluenose II's *VHDL generation* module supports VHDL record types by generating conversion function templates. The readability of the VHDL code generated by Bluenose II is significantly improved from the one generated by Bluenose I. By using an updated VHDL standard and removing indirect signal assignments, Bluenose II is able to generate concise VHDL code from the design.

In the following chapter, we discuss pipeline verification with Bluenose II. In particular, we will describe the HOL script generation module.

Chapter 4

Pipeline Verification with Bluenose II

The HOL theorem prover is used in hardware verification to model specifications and implementations so they can be reasoned about. In HOL, a digital system is modeled as a relation between its input and output signals and a signal is modeled as a time-to-value function. The goal of Bluenose II's HOL generation module is to generate a script in HOL syntax that describes the user's pipeline design. The generated script declares stage and pipeline definitions which are essentially compositions of functions from the HOL libraries and the HOL version of *PipeLib*. The user can then run the script in HOL and reason about the design in HOL and its extension. As mentioned in Section 2.4, HOL can be extended with other verification tools by calling the functions provided by these tools. In essence, Bluenose II's ability to generate HOL scripts turns HOL into Bluenose II's external verification environment.

Although formal verification with the use of tools such as HOL has led to the discovery of subtle design bugs, it is still not widely adopted because it requires specialized expertise which most designers do not have. Our ultimate goal is to fully automate the pipeline verification process with Bluenose II. The goal is for Bluenose II to generate a HOL script complete with all the functions and commands needed for verification in addition to the pipeline model which we describe below. Before we achieve full verification automation, we also strive to make the interaction with HOL from Bluenose II as intuitive to designers as possible. As a first step to that end, we develop the HOL console, a direction connection between HOL and Bluenose II.

In this chapter, we first describe how to model stages and pipelines using functions from *PipeLib* in Section 4.1. We then outline the generation of the HOL script in Section 4.2. Section 4.3 discusses the HOL console, a direct connection from Bluenose II to HOL.

4.1 HOL PipeLib

The HOL version of *PipeLib* was written by Aagaard. Similar to the VHDL version of *PipeLib*, the purpose of the HOL version is to create a library of reusable definitions of the *PipeNet* components. The user as well as the HOL generation module reuse these definitions in the construction of the pipeline definition. In addition, *PipeLib* provides a `stage` function which takes component instantiations, datapath, stage inputs and outputs as arguments. The function saves the user the trouble of "connecting" the internal signals of a pipeline stage.

Listing 4.1 shows the definition of the `stage` function. As shown in line 2 in Listing 4.1, the `stage` function takes the options for *Arbiter* and *MkR/A* (`arb_opt` and `mkra_opts`), the *Datapath* definition (`dp`), the `reset` signal, and stage input and output signals (`xPclP` and `xPclN`). The options for *Arbiter* and *MkR/A* are provided by *PipeLib* while the *Datapath* definition is specified by the designer. The result of HOL's evaluation of the `stage` function is shown in Listing 4.2.

Listing 4.1: Stage definition in HOL

```
 1  Define
 2  ‘ stage arb_opt mkra_opts dp reset xPclP xPclN =
 3      intern pclP selP pclN selN zDataP .
 4      begin_ckt
 5        ARB arb_opt (vec_map pcl_to_req xPclP) selP;
 6        INTP xPclP selP pclP;
 7        MKRA_GENL mkra_opts reset (pclP.req) (pclP.acc) (pclN.req) (pclN.acc
            );
 8        REG [] (pclP.data) zDataP;
 9        dp zDataP (pclN.data) selN;
10        INTN pclN selN xPclN
11      end_ckt
12  ‘;
```

Listing 4.2: HOL's evaluation of stage definition

```
Definition has been stored under "stage_def".
> val it =
    |- !(arb_opt :arb_options)
          (mkra_opts :(mkReqAcc_option # bool signal) list)
          (dp :’a signal -> ’b signal -> bool vector signal -> ckt)
          (reset :bool signal) (xPclP :(’a, bool, bool) pcl vector signal)
          (xPclN :(’b, bool, bool) pcl vector signal).
         stage arb_opt mkra_opts dp reset xPclP xPclN =
         ...
           begin_ckt
             ARB arb_opt
               (vec_map (pcl_to_req :(’a, bool, bool) pcl -> bool) xPclP :
                  bool vector signal) selP;
             INTP xPclP selP pclP;
             MKRA_GENL mkra_opts reset pclP.req pclP.acc pclN.req pclN.acc
                 ;
             REG ([] :(reg_opt # 'a reg_opt_ty) list) pclP.data zDataP;
             dp zDataP pclN.data selN;
             INTN pclN selN xPclN
           end_ckt : thm
```

Listing 4.2 shows that each variable is associated with a type. These types are inferred by the HOL type checker from their contexts. For example, the type checker might have deduced the type of `xPclP` (`('a, bool, bool) pcl vector signal`) from `INTP` (line 6 in Listing 4.1). `INTP` is a function defined in *PipeLib* for modeling *Interface Prev* which takes a parcel vector and a select signal, and returns the selected parcel. `INTP` has the type of

```
('a, bool, bool) pcl vector signal -> bool vector signal -> ('a, bool,
    bool) pcl signal -> ckt
```

where `('a, bool, bool) pcl vector signal` is the type of the parcel vector, `bool vector signal` is the type of the select signal, `('a, bool, bool) pcl signal` is the type of the selected parcel and `ckt` is the type of the expression.

If there is not enough information for the type checker to resolve the types of all variables, the type checker then assigns the variable with a type variable which has a polymorphic type. Variables and functions that contain the same type variables in the *same* definition have the same types. All the type checking rules apply. The definition shown in Listing 4.2 has two type variables, `'a` and `'b`.

When used in a pipeline definition, the stage input and output signals in each stage definition allow us to specify how the stages are connected to each other. Any external signal can be specified as a parameter to the function and this applies to the datapath function, stage function and pipeline function. The order of the parameters has to be respected when the function is used. Line 2 in Listing 4.3 is an example of how the `stage` function is used. `stage_sub` is the name of the function, `exclusive_arb` is the instantiation for *Arbiter*, `[]` indicates that the default options are to be used for *MkR/A* and `SUB` is the *Datapath* that is defined as an uninterpreted function in line 1 of Listing 4.3. Uninterpreted functions are discussed below.

To discuss how the *Datapath* function is specified, we need to first describe uninterpreted functions and uninterpreted types. Uninterpreted functions and types can be thought of as placeholders for the actual functions and types where they will be used. An uninterpreted function is declared as a constant with a name and type and an uninterpreted type is declared with a new type constructor. They constrain and help the type checker to determine the types of other variables in the same expression. Since the implementation of the *Datapath* component is provided by the user in Bluenose II, the *Datapath* function will also have to be defined by the user in HOL. Before the actual *Datapath* function is implemented, an uninterpreted function is declared and used in the definition of the stage. Similarly, the content or the HOL type of a stage parcel is analogous to the record type used in the VHDL generation. The type is defined by the user and unknown to Bluenose II. In this case, an uninterpreted type is used in place of the actual parcel type until it is defined. The declarations of uninterpreted functions and uninterpreted types are explained below with an example.

Listing 4.3 shows a HOL script that describes the pipeline shown in Figure 3.6. The HOL script defines a pipeline function and the associated stage functions. An uninterpreted function is declared with the `new_constant` function. The `new_constant` function installs a constant with the specified name and type in the current theory so the constant can be used later [43]. The type of the uninterpreted function shown in line 1, 3 and 5 in Listing 4.3 is chosen so that it conforms to the type of the *Datapath* function as inferred from the `stage` definition. The inferred type of the *Datapath* function is shown in line 17 in Listing 4.2. Similarly, an uninterpreted type is is declared with the `new_type` function. The `new_type` function declares a type constructor in the current theory with the specified name and number of parameters. An uninterpreted type is declared in line 7 in Listing 4.3.

Listing 4.3: Definition of the diffAdd pipeline and its stages

```
 1 new_constant("SUB", Type‘:’a signal -> ’b signal -> bool vector signal ->
       ckt‘);
 2 Define ‘stage_sub = stage exclusive_arb [] (SUB )‘;
 3 new_constant("NEG", Type‘:’a signal -> ’b signal -> bool vector signal ->
       ckt‘);
 4 Define ‘stage_neg = stage exclusive_arb [] (NEG )‘;
 5 new_constant("ADD", Type‘:’a signal -> ’b signal -> bool vector signal ->
       ckt‘);
 6 Define ‘stage_add = stage static_arb [] (ADD )‘;
 7 new_type("uninterpreted_intern_type_1",0);
 8 Define
 9 ‘ diffAdd reset pclP pclN =
10   intern
11     pcl_sub_neg:(uninterpreted_intern_type_1, bool, bool) pcl signal
12     pcl_sub_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
13     pcl_neg_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
14   .
15   begin_ckt
16     stage_sub reset [& pclP &] [& pcl_sub_neg; pcl_sub_add &];
17     stage_neg reset [& pcl_sub_neg &] [& pcl_neg_add &];
18     stage_add reset [& pcl_sub_add; pcl_neg_add &] [& pclN &]
19   end_ckt
20 ‘;
```

The job of the HOL generation module is to generate a script with the definitions of stage and pipeline functions and the declarations of the necessary uninterpreted functions and uninterpreted types. The definitions together describe the designer's pipeline model in higher order logic. The HOL generation module is detailed in Section 4.2.

4.2 HOL Generation

As mentioned in the previous section, uninterpreted functions and uninterpreted types are used as placeholders in the definitions of the `stage` functions. Therefore, they need to be declared before they are used. The HOL scripts generated by Bluenose II have the structure shown in Listing 4.4.

Listing 4.4: HOL script structure

```
Declaration of datapath (uninterpreted) function for datapath0
Declaration of stage function for stage0 which contains datapath0

Declaration of datapath (uninterpreted) function for datapath1
Declaration of stage function for stage1 which contains datapath1
...

Declarations of all the uninterpreted types that will be used in the
   pipeline definition
Declarations of all the conversion functions that will be used in the
   pipeline definition

Declaration of the pipeline definition using the functions declared
   previously
```

Special care has to be taken when generating a *Datapath* function — declaring a constant with a name that has already been declared previously, even with the same type, will cause the "out-of-date" exception in HOL. This may happen if two stages use the same *Datapath* in the pipeline model since the VHDL ID property (name of VHDL entity) of the block is used as the name of the constant. The uninterpreted function of a *Datapath* is only declared if it has not been declared previously. Listing 4.5 shows a high-level view of the logic in the HOL generation module.

Listing 4.5: Pseudo-code for HOL generation

```
generateHOL()
{
  for each pipeline stage
  {
    generate datapath definition if it is not already generated
    generate stage definition
  }
  generate pipeline definition
}
```

Uninterpreted Types

One of the challenges in implementing the HOL generation module is the generation and assignment of the uninterpreted types. The uninterpreted types in the generated HOL script are used to guide the HOL type checker in inferring the types of the input and output signals. In particular, the types of all the input signals to a stage have to be identical and the types of the output signals must be identical to each other. Where the types of the signals conflict, we need to convert the type using a conversion function. Unlike the VHDL generation where port types are supplied by the user, the HOL generation module needs to deduce the uninterpreted types from the structure of the pipeline. An overview of the logic to assign uninterpreted types is shown in the pseudo-code in Listing 4.6. The details of each step are described below.

Listing 4.6: Pseudo-code for generating and assigning uninterpreted types

```
/* 1. Generate a stage map that keeps track of the order of the stages in
     the pipeline. */
/* 2. For each connection, determine its type number and wrap it in a
    conversion function if needed */
for each connection between stages {
  if (source stage placement number < destination stage placement number)
       {
    type number = source stage placement number;
  } else if (source stage placement number == destination stage placement
       number) {
    if (the source stage has connection to stages that have bigger
        placement number) {
      /* This is a connection between stages that are placed at the same
          position in the pipeline(s). The source stage also outputs to
          other downstream stages. */
      type number = source stage placement number;
      signal needs to be wrapped in a conversion function that converts
          the output type of the source stage to the type specified by the
           type number before passing it to destination stage;
    } else {
      /* Output of source stage either loops back as an input to the same
          stage. Or, the source stage ONLY outputs to destination stages
          that are placed at the same position as the source stage. */
      type number = source stage placement number - 1;
    }
  } else {
    // source stage placement number > destination stage placement number
    type number = source stage placement number;
    signal needs to be wrapped in a conversion function that converts the
         output type of the source stage to the type specified by the type
         number before passing it to destination stage;
  }

  Save any new type that is assigned;
}
```

The first step is to generate a stage map that keeps track of the order of the stages in the pipeline. Although the code generated by EMF provides a method to get all the stages in a pipeline, the list of stages is not sorted in any order. A stage map is implemented as a hash map that associates the stages with their placements in the pipeline. Given a stage, the stage map returns the placement of the stage in the pipeline. The hash map data structure is used because it provides efficient lookup operation. This Java implementation of hash map provides constant-time performance for the lookup operation [44]. Other data structures such as lists involve traversing the whole data structure to find the stage of interest.

The stage map is created by first finding the stages that do not have any previous stages in the pipeline (these are assumed to be the first stages in the pipeline) then traversing along the stage connections. During the traversal, "placement numbers" are assigned to the stages. Figure 4.1 shows the order of the stages (the number after the colon on the block) of a two-wide superscalar pipeline after the traversal. The smaller the number, the earlier the corresponding stage is in the pipeline.

After the order of the stages is determined, we generate the uninterpreted types and assign them to signals that will be passed to the stage inputs and outputs. Since the types of all the input (output) signals to (from) one stage have to agree with each other, the type of a signal may agree with the types of the output signals of the source stage and not with the types of the input signals of the destination stage. In that case, we also need to generate an uninterpreted function that converts one type to another and wrap it around the signal before passing it to the input of the destination stage. To automate the generation and assignment of uninterpreted types, the names of all the types take the form `UNINTERPRETED_INTERN_TYPE_`*number* where the number distinguishes the type. The task of generating and assigning the uninterpreted type then becomes determining the right type number.

In a trivial pipeline, all the stages output to their downstream stages (*e.g.*, no self loop). The type of the input signals of a stage is one smaller than its placement number and the type of its output signals is the same as its placement number. Since the pipeline is trivial in the sense of parcel flow, the type of the output signals of the source stage is the same as the type of the input signals of the destination stage in a connection. For the non-trivial cases, we have to determine the type of the signal connecting the two stages and whether a conversion function is needed.

To determine the number for an uninterpreted type for a signal, we look at the source stage and destination stage that are connected by the signal. There are three scenarios, the source stage placement number can be smaller than, equal to or larger than the destination stage placement number. The first scenario where the source stage placement number is smaller than the destination stage placement number is the trivial case mentioned above. In this case, the source stage output signals have the same type as the destination stage input signals. The type of the signal is assigned the placement number of the source stage (line 4 and 5 in Listing 4.6). An example of the first scenario is illustrated in Figure 4.1 where stage *Fetch 1* is connected to *Decode 1*. All the outputs of the *Fetch 1* stage have type `1` and so do all the inputs of the *Decode 1* stage. The “type” of a signal is shown beside the signal in Figure 4.1.

The second scenario, where the source stage and the destination stage have the same placement number, may happen for two reasons. One reason is a self loop where the source and destination are the same stage; the other reason is a connection between two stages that are placed at the same position in the pipeline (or pipelines in a superscalar pipeline). The type of the signal depends on which of these reasons is the cause of the condition. If the condition is caused by a self loop or by the connection of a source stage that outputs *only* to destination stages that are placed at the same position as the source stage, then the type of the signal is one smaller that the placement number of the source stage (line 11 to 14 in Listing 4.6). The reason for this is, if the source stage does not output to other downstream stages, then there is no reason to create a separate type for its output. On the other hand, if the condition is caused by a connection between two stages that are placed at the same position in the pipeline (or pipelines) and the source stage outputs to other downstream stages, then the type of the signal is the same as the placement number of the source stage. The signal is “wrapped” in a conversion function when passed to the input of the destination stage (line 7 to 10 in Listing 4.6). The two possible causes for the second scenario are shown in Figure 4.1 where one of the outputs of the *ALU* stage loops back as an input to the same stage and *Decode 1* is connected to *Decode 0*.

Finally, a possible cause for the third scenario where the source stage placement number is larger than the destination stage is a bypass path from a stage that is placed later in the pipeline to a stage that is placed earlier in the pipeline. In this case, the type of the signal is the same as the placement number of the source stage and the signal is “wrapped” in a conversion function when passed to the input of the destination stage (line 15 to 18 in Listing 4.6).

In the last step of the uninterpreted type assignment, if a type is created in the assignment, it is saved into a list. We traverse this list to generate type declarations before the types are used in the pipeline definition. The generated HOL script based on the structure illustrated in Figure 4.1 is shown in Listing C.1 in Appendix C.

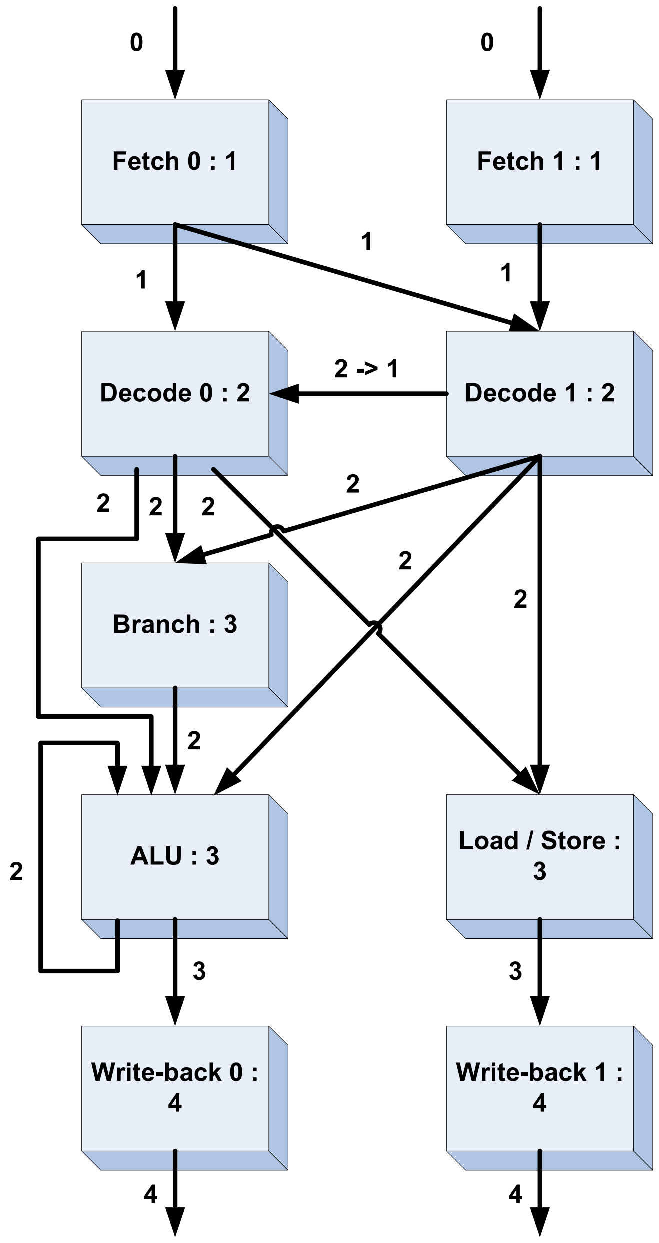
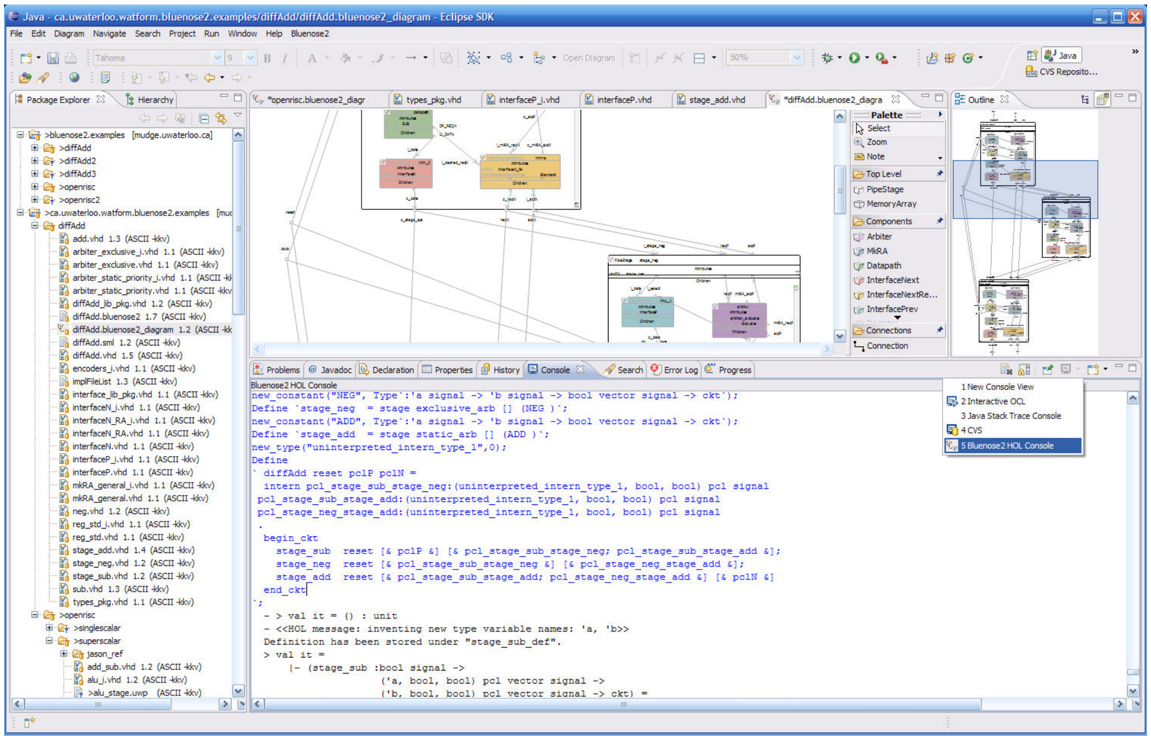


Figure 4.1: Two-issue superscalar pipeline with numbers denoting stage order and signal types

4.3 HOL Console

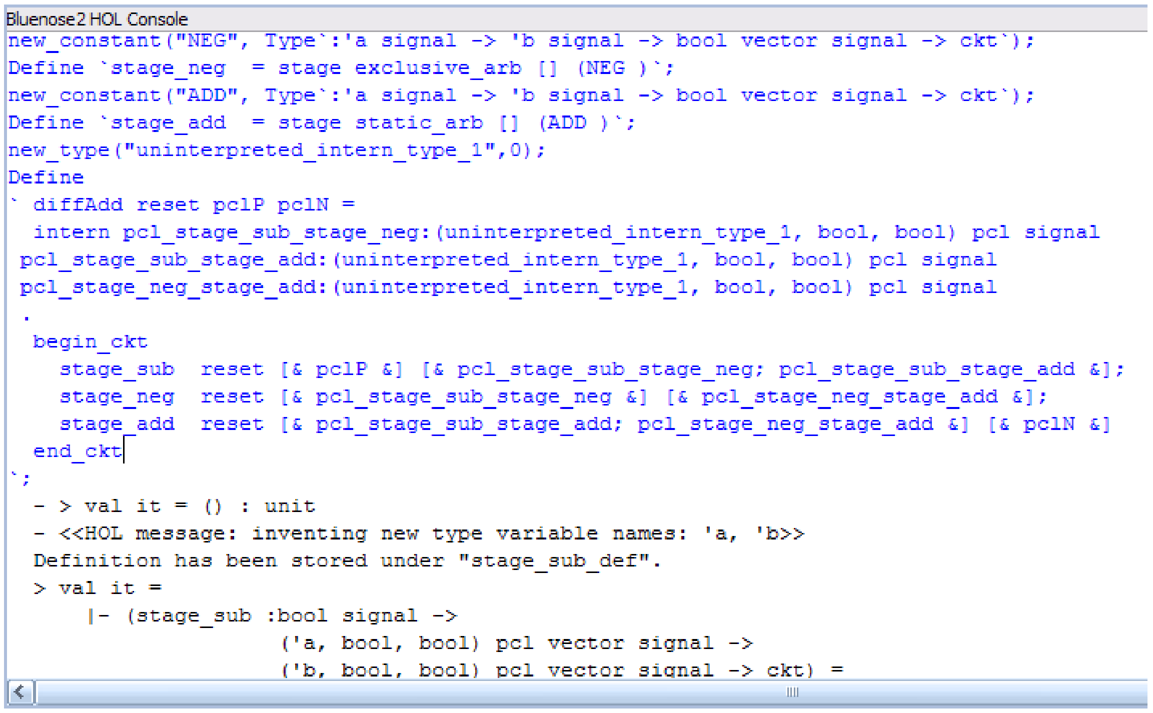
As a step towards a full-featured integrated development environment for pipeline design and verification, we implemented a direct connection to HOL. We are able to display results from HOL as a console in Eclipse. Figure 4.2 is a screen capture of Bluenose II with the HOL console at the bottom. Figure 4.3 zooms in on the HOL console of the screen capture. A brief description of the implementation is provided below.


Bluenose2 HOL Console

```
new_constant("NEG", Type`:'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `stage_neg  = stage exclusive_arb [] (NEG )`;
new_constant("ADD", Type`:'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `stage_add  = stage static_arb [] (ADD )`;
new_type("uninterpreted_intern_type_1",0);
Define
` diffAdd reset pclP pclN =
  intern pcl_stage_sub_stage_neg:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_stage_sub_stage_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_stage_neg_stage_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
 .
  begin_ckt
    stage_sub  reset [& pclP &] [& pcl_stage_sub_stage_neg; pcl_stage_sub_stage_add &];
    stage_neg  reset [& pcl_stage_sub_stage_neg &] [& pcl_stage_neg_stage_add &];
    stage_add  reset [& pcl_stage_sub_stage_add; pcl_stage_neg_stage_add &] [& pclN &]
  end_ckt
`;
  - > val it = () : unit
  - <<HOL message: inventing new type variable names: 'a, 'b>>
  Definition has been stored under "stage_sub_def".
  > val it =
      |- (stage_sub :bool signal ->
                     ('a, bool, bool) pcl vector signal ->
                     ('b, bool, bool) pcl vector signal -> ckt) =
```

Figure 4.2: Bluenose II with HOL console

The HOL console is implemented by running HOL using Java's `Runtime` class's `exec()` method [44]. The output and error streams of the HOL process are passed to the HOL console in separate threads so they can be handled simultaneously. The HOL console is implemented as an extension to Eclipse's console factories (`org.eclipse.ui.console.consoleFactories`). Although Java and Eclipse are cross-platform, since HOL is run as a process on the operating system, the HOL console is subjected to HOL's availability on an operating system. HOL is currently available on Unix systems and Windows. However, there are differences running HOL between the two systems. For example, dynamic linking in HOL is unavailable on Windows. It is in our future work to further investigate these issues so pipeline verification with HOL in Bluenose II will be more seamless to the users.



```
new_constant("NEG", Type`:'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `stage_neg  = stage exclusive_arb [] (NEG )`;
new_constant("ADD", Type`:'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `stage_add  = stage static_arb [] (ADD )`;
new_type("uninterpreted_intern_type_1",0);
Define
` diffAdd reset pclP pclN =
  intern pcl_stage_sub_stage_neg:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_stage_sub_stage_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_stage_neg_stage_add:(uninterpreted_intern_type_1, bool, bool) pcl signal
 .
  begin_ckt
    stage_sub  reset [& pclP &] [& pcl_stage_sub_stage_neg; pcl_stage_sub_stage_add &];
    stage_neg  reset [& pcl_stage_sub_stage_neg &] [& pcl_stage_neg_stage_add &];
    stage_add  reset [& pcl_stage_sub_stage_add; pcl_stage_neg_stage_add &] [& pclN &]
  end_ckt
`;
  - > val it = () : unit
  - <<HOL message: inventing new type variable names: 'a, 'b>>
  Definition has been stored under "stage_sub_def".
  > val it =
      |- (stage_sub :bool signal ->
                     ('a, bool, bool) pcl vector signal ->
                     ('b, bool, bool) pcl vector signal -> ckt) =
```

Figure 4.3: HOL console

4.4 Summary

In this chapter, we described how the HOL version of *PipeLib* is used to describe a pipeline design. The HOL version of *PipeLib* provides a library of reusable definitions of the *PipeNet* components. Bluenose II's HOL generation module generates script with HOL commands and functions from *PipeLib* to describe the user's pipeline design. The user can then run the script in HOL to reason about the design. A script is made up of four parts: declarations of uninterpreted functions that act as placeholders for *Datapath*, stage definitions that contain these *Datapath* functions, declarations of uninterpreted types that represent the parcels, and lastly, the composition of the stage definitions and the uninterpreted types to form the pipeline definition. The generation and assignment of uninterpreted types is one of the challenges in the implementation of the HOL generation module. The algorithm was described in this chapter. Finally, the HOL console, a direct connection between Bluenose II and HOL, was presented. It is in our future work to further incorporate HOL as Bluenose II's external verification environment.

Chapter 5

Bluenose II Core

The pipeline metamodel, *PipeNet*, is appropriate for describing pipelined circuits. However, it needs to be represented by appropriate data structures in the core of Bluenose II in order to be efficiently manipulated. The Bluenose II core contains the main data structures and the methods for manipulating these data structures. The core interacts with all the other modules as shown in Figure 3.1. These data structures are generated by GMF from the developer-defined domain model.

As mentioned in Section 2.3, the developer defines a domain model that describes the abstract syntax of the modeling language that will be available in the graphical editor. In our case, *PipeNet* is the modeling language. The abstract syntax for *PipeNet* is a metamodel that *PipeNet* conforms to. For each class in the domain model, GMF generates a Java interface and a corresponding implementation class. The generated code also contains methods that are needed to manipulate these model classes. These model classes make up the data structures that are used in the core of Bluenose II. In this chapter, we describe these data structures and how they are implemented and used.

Section 5.1 describes the data structures in Bluenose II. Section 5.2 outlines the implementation of the Bluenose II core.

5.1 Data Structures

While we can enter *PipeNet* directly as the domain model to GMF or other model-driven development tools, there will be a lot of redundancies in the generated data structures. For example, all the components in *PipeNet* share a few properties (*e.g.*, `name`, see Figure 3.2). If a new component is added to *PipeNet*, it would be tedious to add these properties to the new component. Taking this into consideration, the overall *PipeNet* metamodel is represented as hierarchical blocks and directed connections in the domain model definition. The objects in *PipeNet*, from the atomic components to the pipeline itself, are specializations of the generic *Block* class. The simplified class diagram in Figure 5.1 illustrates the concept. The block-based approach promotes reuse and extensibility and the traversal algorithm described in Section 3.6 can be easily implemented using this structure.

As seen in Figure 5.1, *Block* and *Connection* are fundamental data structures in Bluenose II. All the other pipeline model elements derive from them. For the rest of this chapter, the class identifiers (*e.g.*, *Block* and *Connection*) can refer to either the actual class or an object which is an instantiation of a class. The concept that a class identifier represents depends on the context.

5.1.1 Block

In addition to the influence from Bluenose I, the modularity of Aagaard's pipeline model and the structural hierarchy of VHDL have inspired the design of the data structures used in the Bluenose II core. Upon examining the VHDL code of a pipeline, we observed that the pipeline is a component and it contains pipeline stages which are themselves components and they in turn contain other components. The structural hierarchy is achieved through a series of component instantiations; a component is instantiated in the enclosing architecture body, which is instantiated in another enclosing architecture body. This observation led us to the creation of a generic *Block* class which in modeling terms, has a composition relationship with itself. The self-composition relationship is represented by the arrow looping back to itself in Figure 5.1. This indicates that a *Block* object can contain other *Block* objects. All the components in the pipeline model derive from the *Block* class. The composition relationship also means that the items are destroyed along with their container (see legend in Figure 5.1) when their container is destroyed. It makes sense to model the pipeline this way because most of the elements have no meaning when they are considered on their own or outside their container. An example is a *PipeStage* which alone serves no purpose without its *Pipeline* container. Like VHDL, the components are connected to other components through their ports. The lists of *Port* and *Generic* objects are contained in the parent *Block* object's *BlkInterface*, which is analogous to the entity declaration statement in VHDL. Although these parts of the *PipeNet* metamodel were inspired by VHDL, we are confident that *PipeNet* is generic enough to capture a wide range of hardware designs. So far the classes that we have discussed are structurally similar to their corresponding VHDL constructs, but the *Connection* class is unlike its VHDL counterpart.

5.1.2 Connection

Connections are not explicit in Aagaard's pipeline model and they are implied by signal assignments in VHDL. We model the connection between two ports as the *Connection* class with source (*src*) and destination (*dst*) attributes. Therefore, the *src* and *dst* objects can be reached with the accessor functions (*i.e.*, get and set methods) of the *Connection* object. Although the point-to-point connection

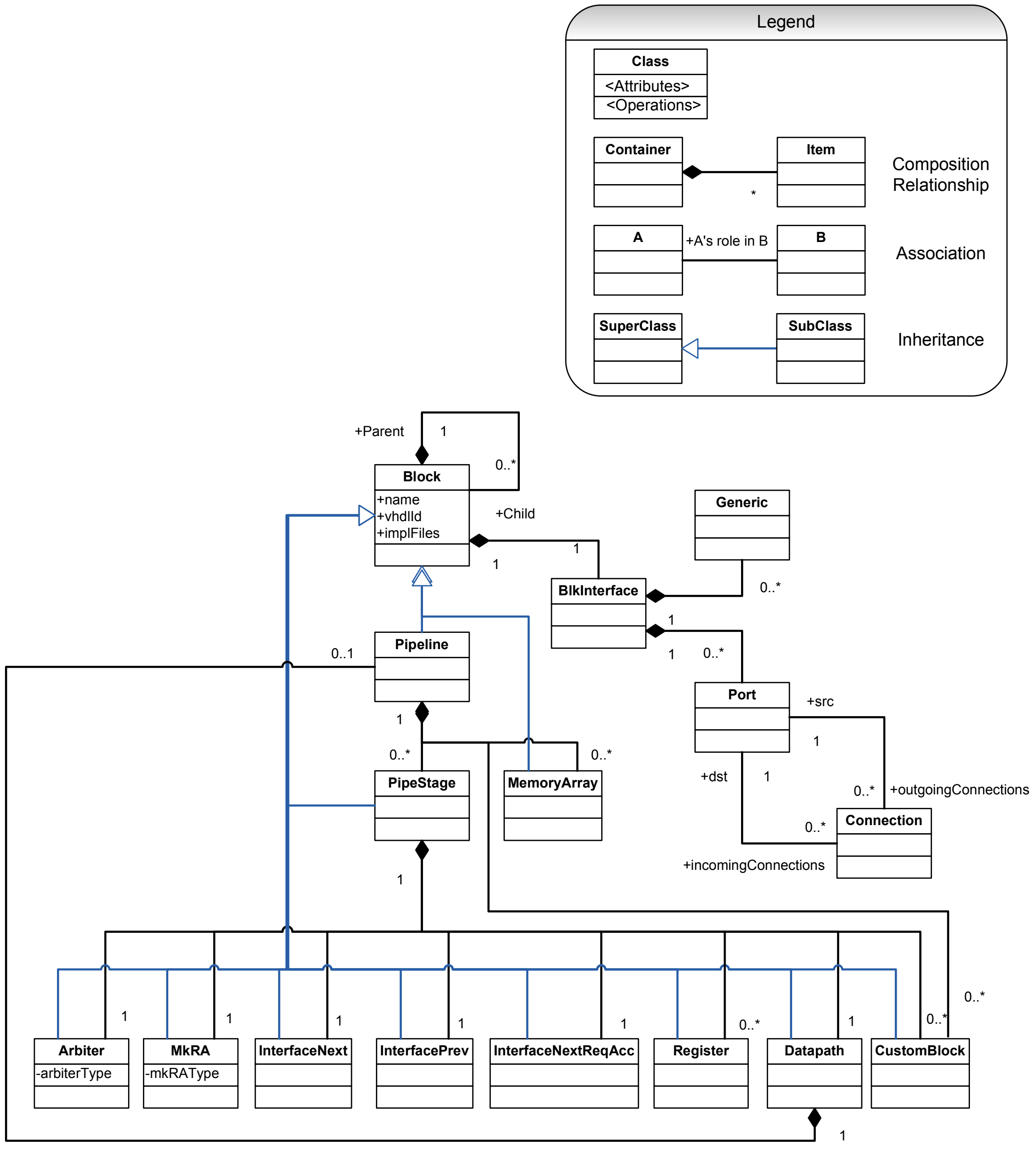


Figure 5.1: Simplified class diagram for the pipeline metamodel

allows us to capture the same wiring information as a signal in VHDL, there is a subtle difference between the models represented by the *Connection* class and a signal. A signal can drive multiple receivers in VHDL. From the signal identifier alone (without doing a search in the VHDL source code), we cannot tell what its receivers (ports or other signals) are. On the other hand, there is always a source and destination associated with a *Connection*. Figure 5.2 illustrates the difference between the two models. On the right hand side of Figure 5.2, the signal *port_A_sig* which serves as an intermediate value from *port_A*, drives *port_B*, *port_C*, and *port_D*. On the left hand side of Figure 5.2, each directed edge shows a *Connection* (*connA_B*, *connA_C* and *connA_D*) with a source and destination.

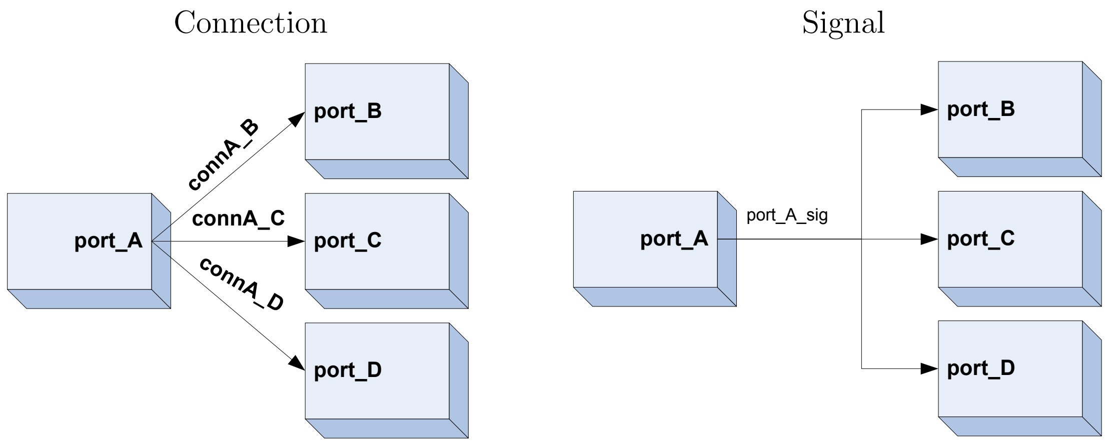


Figure 5.2: Comparison between connection and signal

The question is why not simply model the structure and/or behaviour of VHDL signal more closely like *Block* and *BlkInterface* to VHDL entity and entity declaration? There were a few factors that led us to the decision to model the wiring between two components as *Connection*:

- We were influenced by the way connection was drawn in Bluenose I. All the connections, from stage connections to port connections, were point-to-point.
- The point-to-point connection model was used in many examples in GMF literature. This is an assurance that there will be a lot of references for its implementation if we run into problems.
- We predicted that the graphical implementation of the routing style shown on the right hand side of Figure 5.2 would be more difficult.

These factors are from the developer's point of view. From the user's perspective, we predict that the level of ease to express in either model will be similar. However, Bluenose II's intention is not only to capture user's design but to also guide the user in the design process. It is possible that the different connection models may influence the designer in how she perceives the wiring between multiple blocks and lead to different designs. Future exploration of different connection models is needed to evaluate and compare the expressiveness of the models.

Based on the domain analysis, we identified the common properties among the different objects in the *PipeNet* metamodel. We assigned these properties to the *Block* and *Connection* classes so other classes that specialize them will automatically inherit these properties as well. The properties for *Block* and *Connection* are shown in Table 5.1 and Table 5.2 respectively.

Table 5.1: Attributes of the *Block* class

Name :	Equivalent to instance label of an entity in VHDL
VhdlID :	Equivalent to entity name in VHDL
BlkInterface :	Equivalent to entity declaration in VHDL, contains a list of ports and generics (objects)
ImplFiles :	Contains the list of files that are associated with this block
Properties :	Contains a list of custom properties and their values (for future expansion)

Table 5.2: Attributes of the *Connection* class

Name :	Optional attribute, used for identifying the connection
Src :	Points to a port on the source block
SrcSlice :	Equivalent to the range on the array port in VHDL, used to indicate the slice of the array port on the source block that is connected
Dst :	Points to a port on the destination block
DstSlice :	Used to indicate the slice of the array port on the destination block that is connected

In this section, we presented the data structure used in the core of Bluenose II. The two fundamental classes are the *Block* and *Connection* classes. We embedded this information in our domain model along with *PipeNet* so that GMF will generate the desired data structures. Details on how the domain model is specified and other implementation issues are documented in Section 5.2.

5.2 Implementation

Bluenose II makes use of GMF, an Eclipse plug-in, to harness the benefits of model driven development and more specifically the framework's generative functionality.

This section outlines the implementation with Eclipse and GMF.

5.2.1 Java Generation

As mentioned in Sectcion 2.3, part of the first step in developing a graphical editor with GMF is to define the domain model. The Ecore metamodel is used to specify the domain model. There are a few ways to define an Ecore model that are accepted by the EMF plugin. EMF generates the Java code for the domain model used by GMF. We have chosen annotated Java to input our domain model as its syntax is the most familiar to us [19]. As its name implies, the domain model is specified by annotating Java with model information. To be specific, the developer provides partially complete Java interfaces that are annotated with model information. Normally, the EMF generator generates full Java interfaces based on the model definition (defined in other ways) and annotates the code with model information. By providing the partially complete Java interface, we are effectively guiding the EMF code generation process.

Listing 5.1 shows an excerpt of the model definition of the *Block* class in annotated Java. As the EMF generator parses the Java interface, the `@model` tag (line 1, 4, 9 *etc.*) signifies to the generator that the construct that follows corresponds to a model element and requires generation. The `@model` tag can also be used to carry additional model information for the element. Line 14 is an example of a containment specification which describes the composition relationship as explained in Section 5.1. Attributes and references for a model are specified as `get` methods in the corresponding model class annotated with the `@model` tags (line 6, 11, 16 *etc.*). The EMF generator completes the interface with the model information and generates the corresponding implementation code for the model.

Listing 5.1: Model definition of the *Block* class in annotated Java

```
 1  @model
 2  public interface Block extends End {
 3    /**
 4     * @model
 5     */
 6    String getName();
 7
 8    /**
 9     * @model
10     */
11    String getVhdlId();
12
13    /**
14     * @model containment="true"
15     */
16    BlkInterface getBlkInterface();
17
```

```
  /**
   * @model dataType="ca.uwaterloo.watform.bluenose2.File"
   */
  EList getImplFiles();

  /**
   * @model mapType="ca.uwaterloo.watform.bluenose2.
       EStringToStringMapEntry" keyType="java.lang.String" valueType="java
       .lang.String"
   */
  EMap getProperties();
...
```

As mentioned above, the EMF generator also annotates the generated code with model information. Specifically, each generated portion is annotated with the `@generated` tag. The annotation allows the EMF generator to distinguish between the code that it generated and the user's changes. As a result, the developer can modify the domain model as explained above or edit the generated code directly. The EMF generator will not modify user's *addition* to the code if the `@generated` tag is removed or its value is changed (*e.g.*, `@generated NOT`). Otherwise, the EMF generator will overwrite the change.

The code generated from the domain model is only responsible for monitoring the states of the model objects while their graphical representation are being manipulated in the graphical editor. The rest of the code for the graphical editor is based on the graphical definition, tooling definition, and the mapping model as explained in Section 2.3. These definitions all have to be provided by the developer.

Since the time when the graphical and tooling models were first defined, significant customization has been made to the generated diagram code and the models have not been updated at the same time as these changes were applied. For example, out of convenience, some graphical definitions were added to the code manually instead of being added to the definition and re-generated. Therefore, these accumulated changes are in conflict with the old graphical and tooling definitions. Since then, we have stopped generating diagram code from these definitions. We maintain and modify the diagram code directly. Although we have not encountered any problems where we needed up-to-date graphical and tooling models, it may be worth spending the time on aligning these models with the code as they provide clear pictures of the graphical definitions and tools implemented in Bluenose II. These models may also serve as documentation.

5.2.2 Customization

GMF is made up of two components, the toolset for generating code of the graphical editor and the runtime infrastructure that the graphical editor uses. In the previous section, we described the use of the GMF tools. In this section, we will discuss how the runtime infrastructure affects the development of Bluenose II.

Although a GMF generated graphical editor is intended to be customized, it is not as easy as plug and play. There is a steep learning curve of the internal working of the runtime infrastructure before functionally correct customizations can be made. As an example of this process, the implementation of a feature to update a *Generic* object automatically after the creation of a port connection is discussed below.

As mentioned in Section 3.6, we wanted to eliminate the need for designer to enter values for generics. Besides using some of the features in VHDL as mentioned in Section 3.6, the values of some of these generics can be deduced from the pipeline model, two of these are the *numInputs* and *numOutputs* generics. *numInputs* and *numOutputs* should be equal respectively to the numbers of upstream pipeline stages and downstream stages that are connected to the stage to which these generics belong. The values of these generics are used in the calculations of the widths of array ports and signals inside the block. A pipeline stage receives requests from its upstream pipeline stages and accept signals from its downstream stages through the ports, *reqP* and *accN* respectively. Therefore, with our point-to-point connection model, the numbers of incoming connections to the *reqP* and *accN* ports can be used as the values for the *numInputs* and *numOutputs* generics.

The issue of generating the right values for *numInputs* and *numOutputs* arose originally in the implementation of the VHDL generation module. We wanted to generate VHDL with the right values for the generics without user input. Therefore, our first attempt was to address the issue in the VHDL generation module. Since the VHDL generation module traverses all the blocks and connections, we can update *numInputs* and *numOutputs* in all the *PipeStage* objects during the traversal. The values of *numInputs* and *numOutputs* in a *PipeStage* can be set to the number of connections which have the corresponding *reqP* and *accN* ports as destinations. This approach is a simple solution because we are familiar with the implementation of the VHDL generation module and its logic; therefore, it will not be difficult for us to modify the traversal to include the code to get the numbers of incoming connections and update the generics. However, this approach also has the following problems:

1. Implementing this feature as part of the VHDL generation means that the generics will only be updated when the VHDL generation module is run. The user may be confused when she sees the old values of the generics.

2. Coupling this feature with the VHDL generation functionality may make it difficult to modify the VHDL generation module in the future.

This is an example of the issue that we are faced with very often during the development of Bluenose II — a trade-off between elegance and ease of implementation. In this case, the first problem affects the ease of use in Bluenose II significantly so we opted for a more complex solution. To illustrate the intricacy of GMF as well as the general steps involved in working with the core of Bluenose II, an explanation of the implementation is given below.

Our goal is to implement the feature to automatically update the value of a *Generic* object after a port connection is created. When an object in the model is created, it is configured by its corresponding *EditHelper*. While an *EditHelper* class specifies the base editing commands for a corresponding object, an *EditHelperAdvice* class specifies the commands to execute before and after the commands contributed by the corresponding *EditHelper*. These commands can be viewed as the pre- and post-processing of the base editing commands provided by the *EditHelper*. Knowing these, we implement the *Generic* update feature by specifying a "post-processing" command in an *EditHelperAdvice* class that acts on the *Connection* class. The "post-processing" command gets executed after a *Connection* object is created and edited by its *EditHelper*. Specifically, the "post-processing" command updates the *numInputs* or *numOutputs* generics of the *Block* containing the destination port of the *Connection* if the destination port is a *reqP* or *accN* port respectively. This is possible because we can access the object being edited, in this case, a *Connection* object. The destination of a *Connection* object can be accessed through a `getter` method since the destination is an attribute of the *Connection* class as explained in Section 5.1.2. Through the destination *Port* object, the containing *Block* and consequently its *Generic* objects can be accessed through methods provided by GMF to access the container of a model object. GMF is an intricate infrastructure. This example is only one among many features that we implemented in Bluenose II.

5.3 Summary

In this chapter, we described the data structures used in Bluenose II. The *Block* and *Connection* classes are the building blocks for the pipeline. All the other pipeline elements inherit from *Block* or *Connection*. We then outlined the procedure to generate the Java code for Bluenose II with GMF. In particular, we explained how we specified the domain model with annotated Java. An example of the customization of the generated diagram code was given to illustrate the intricacy of the GMF runtime infrastructure.

Chapter 6

Case Study

To evaluate Bluenose II's functionalities, we have chosen to construct in Bluenose II the OpenRISC 1200 (OR1200) [15] implementation that is based on the OpenRISC 1000 (OR1000) instruction architecture [14]. In particular, we replicated Higgins' design, OR1200-BNv2 (OR1200, Bluenose design, version 2). Following the naming convention, the new version of the design is named OR1200-BNv3. A major reason we have chosen to implement OR1200, specifically Higgins' design, is that we can use OR1200-BNv2 as a reference to evaluate different aspects of Bluenose II. Furthermore, as noted by Higgins, the level of complexity and the amount of work required to construct OR1200 are suitable for the size of the case study [12]. The design is complex enough to illustrate pipelining issues and we have access to the datapath components.

We first discuss the design of OR1200-BNv2 and OR1200-BNv3 in Section 6.1. Observations from the construction of OR1200-BNv3 are noted in Section 6.2. In Section 6.3, we compare the performance of OR1200-BNv3 to that of OR1200-BNv2.

6.1 The OpenRISC Design

OR1200 specifies an implementation of a subset of the OpenRISC 1000 (OR1000) instruction architecture [15]. OR1000 is an architecture for a family of open-source, synthesizable RISC microprocessor cores [14]. The OR1200 is a 32-bit RISC integer pipeline with a Harvard microarchitecture. The OR1200 has the following features:

1. Branch delay slot (one cycle) to keep pipeline as full as possible
2. Conditional branch and unconditional jump instructions
3. Separate execution unit for load and store operations so they do not stall the pipeline unless there is a data dependency

4. Arithmetic instructions, compare instructions, logical instructions, rotate and shift instructions

5. In-order completion

Higgins' implementation of OR1200, OR1200-BNv2, is slightly different from the specification in order to highlight Bluenose I's features with reasonable amount of effort. OR1200-BNv2 are different from the original specification at the following points:

1. OR1200-BNv2 is a two-wide superscalar integer pipeline instead of the 5 stage scalar pipeline as defined in the specification of OR1200

2. Exception handling is not implemented

3. Only the flag and carry supervision registers and four general-purpose 32-bit registers were implemented

4. Shifting requires two cycles instead of one

5. Multiplication requires five cycles instead of three

The purpose of the case study is to compare the generated codes by Bluenose I and Bluenose II, and the original and modified *PipeLib*; therefore, we made few changes to the design. This means that we reused entities from OR1200-BNv2 as much as possible and kept the overall structure of the pipeline. The only major difference between OR1200-BNv2 and OR1200-BNv3 is the use of record types in OR1200-BNv3. Stage inputs and outputs and some related signals are grouped together respectively instead of being passed from one component to another as individual signals. Figure 6.1 shows an overview of OR1200-BNv3. The configuration of each stage is shown in Table 6.1. The reasons for the choices of the instantiations will be discussed as we describe the implementation of some of the features of the design. Although the main focus of our case study is not the design of OR1200, the implementation of some of the features are worth mentioning and they are described next in Section 6.1.

OR1200-BNv3 Design Overview

A list of the features and their implementation are given below. The *MkR/A* instantiations and *Arbiter* instantiations used in the pipeline are summarized after the list.

- Alternate paths for parcel flow allow one or two new instructions to be fetched in every clock cycle. Instead of stalling both fetching stages when the target of one fetching stage is not ready, the *Dispatch + Control* unit increments the

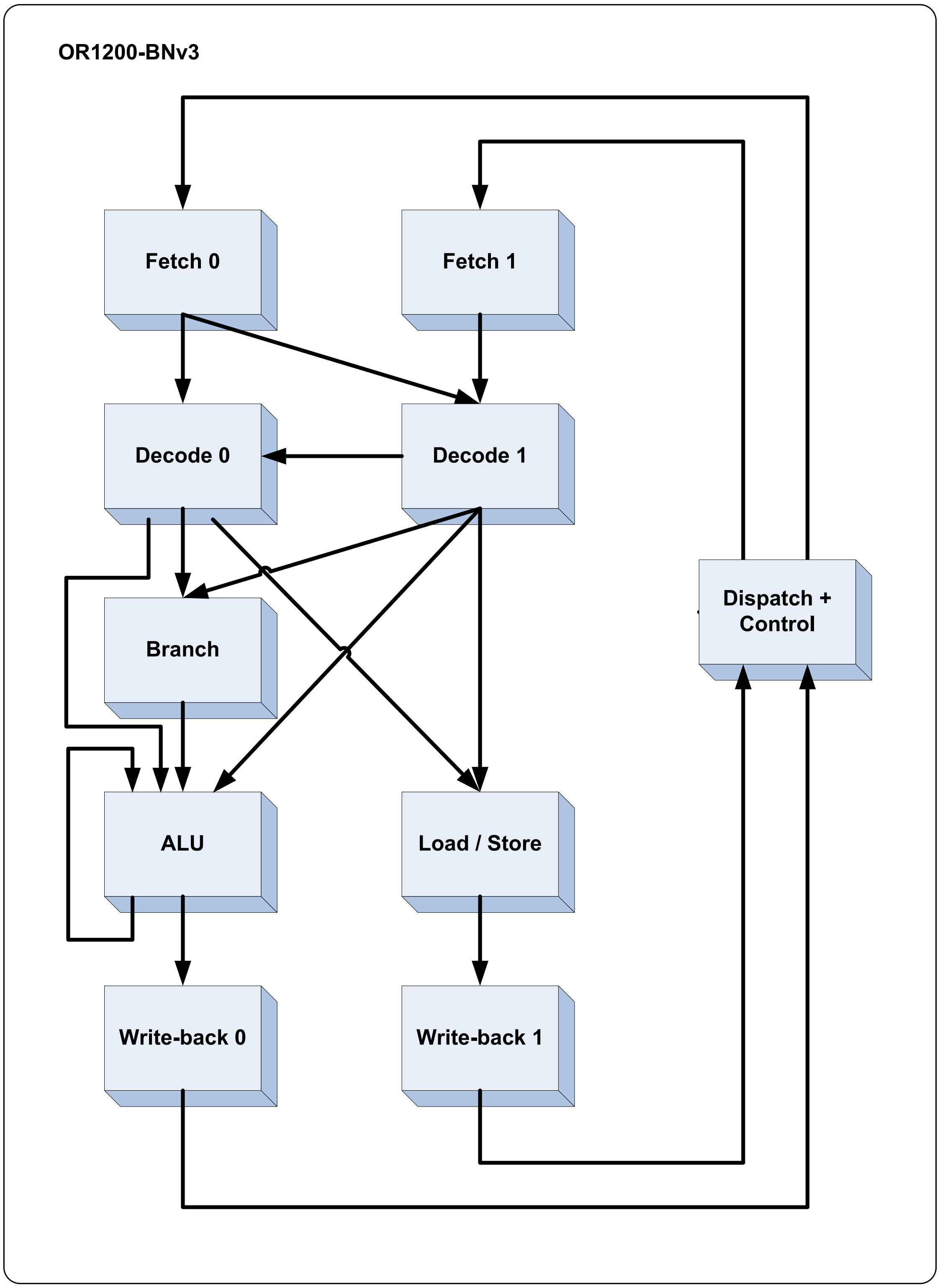


Figure 6.1: OR1200-BNv3

Table 6.1: Stage instantiation parameters for OR1200-BNv3

Stage	Arbiter Type	MkR/A Type	Interface Next Req/Acc Type
Fetch 0	Degenerate	ZeroDelay + abort	Priority
Fetch 1	Degenerate	ZeroDelay + abort	Standard
Decode 0	StaticPriority	General + maskN	Standard
Decode 1	StaticPriority	General + maskN	Priority
Branch	StaticPriority	ZeroDelay	Standard
ALU	StaticPriority	MultiDelay + maskN + loopMstrReq	Standard
Load/Store	StaticPriority	MultiDelay + maskN	Standard
Write-back 0	Degenerate	UnitDelay	Standard
Write-back 1	Degenerate	UnitDelay	Standard

program counter by one instead of two which results in the parcel in *Fetch 1* being dropped and re-fetched into *Fetch 0* and a new parcel being fetched into *Fetch 1*. Consequently, two consecutive instructions are fetched in every clock cycle. This is made possible by the alternate paths for parcel flow from *Fetch 0* and *Decode 1* as shown in Figure 6.1. In the event that *Decode 0* is stalled and *Decode 1* has a bubble, *Fetch 0* can send its parcel to *Decode 1*. Similarly, if the target of *Decode 1* is not available and *Decode 0* is free to accept a request, *Decode 1* can send its parcel to *Decode 0*.

This topology is implemented by selecting the *Priority Interface Next Req/Acc* in *Fetch 0* and *Decode 1* and the *StaticPriority Arbiter* in *Decode 0* and *Decode 1*. The *Priority Interface Next Req/Acc* allows the stage to send a request to a backup stage when the desired targets are not available; while the *StaticPriority Arbiter* allows the stage to accept a request with higher priority. To *Decode 1*, *Fetch 0* has a higher priority than *Fetch 1* and to *Decode 0*, *Decode 1* has a higher priority than *Fetch 0*.

- Special care needs to be taken in implementing the branch instruction in a superscalar pipeline. The branch instruction is handled as follows: In the event that a branch is taken, the instruction in *Fetch 1* is discarded while the instruction in *Fetch 0* may or may not be discarded. Discarding the instruction in *Fetch 0* or not depends on where the branch instruction was processed. If the branch instruction comes from *Decode 1*, then the instruction in *Fetch 0* is from the branch delay slot. An instruction from the branch delay slot needs to be executed and cannot be discarded. On the other hand if the branch instruction comes from *Decode 0* then the instruction from its branch delay slot is already in *Decode 1*; therefore, the instruction in *Fetch 0* is discarded. The discarding logic is implemented as a *Custom Block* external to the stages and it controls the *abort* signals to the *MkR/A*s in *Fetch 0* and *Fetch 1*.

- Read after write hazards are resolved by stalling the decoding stages when data dependency is detected. If the instruction in the ALU or LSU stage writes to the same register that the decoding stage reads from, then the instruction in the decoding stage is prevented to leave. This is achieved by de-asserting the signal to the *maskN* port on the *MkR/A* in the decoding stage until the source register and destination register are not the same. As a result, all the request and accept signals to and from the next stage will be de-asserted until the data dependency no longer exists.

- In-order completion is ensured by preventing a parcel to leave a stage by de-asserting the *maskN* signal to the stage's *MkR/A*. *maskN* to *MkR/A* in *Decode 1* is de-asserted when *Decode 0* is not accepting any request from its previous stages. This is to maintain in-order completion by preventing the instruction in *Decode 1* to leave the stage before the instruction in *Decode 0* leaves *Decode 0*. Similarly, the order of completion in the instructions that enter *ALU* and *Load/Store* are maintained by the assertion of the signals to the respective *maskN* ports. A state machine accounts for the variable latencies in the two stages and determines when a parcel is allowed to leave the stage to enter the write-back stage. The state machine controls the individual *maskN* signals to control the flow of parcels in the two stages. The state machine is implemented as a *Custom Block* external to the stages.

- To demonstrate Bluenose II's capability to handle loops, the shift operation which has a latency of two clock cycles is implemented as an external loop to the *ALU* stage. Upon a shift instruction, *ALU* sends a request to itself. The request also drives the *loopMstrReq* port on the *MkR/A*. The *loopMstrReq* signal tells the *MkR/A* to accept the request even though the "next" stage which is *ALU* itself, has not accepted the current request. The *loopMstrReq* signal in *MkR/A* is used to break the deadlock caused by a request loop. *ALU*'s request to itself has the highest priority over the other stage inputs, with the *StaticPriority Arbiter* selected, *ALU* executes the shift instruction properly.

Here we briefly summarize the selection of *MkR/A* instantiations in all the stages. The *ZeroDelay* instantiation of *MkR/A* was selected for *Fetch 0*, *Fetch 1*, and *Branch* since these stages are combinational (they do not have registers). As mentioned above, the interfaces of *MkR/A*s in *Fetch 0* and *Fetch 1* also include the *abort* ports as inputs. The *General* instantiation of *MkR/A* was selected for *Decode 0* and *Decode 1* as their next stages do not always accept their requests. The *maskN* ports on these *MkR/A*s are driven by logic that resolves read after write hazards. The *MkR/A*s in *ALU*and *Load/Store* are of the *MultiDelay* instantiation since these stages may take multiple clock cycles to process the instructions. Their *maskN* ports are driven by logic that maintains in-order completion between the two stages. In addition to the *maskN* port, the inputs to *MkR/A* in *ALU* also include the *loopMstrReq* signal to handle request that originates from *ALU* itself.

A summary of the selection of *Arbiter* instantiation in all the stages is presented here. Since *Fetch 0*, *Fetch 1*, *Write-back 0* and *Write-back 1* each has only one stage input, the *Degenerate Arbiter* is selected for these stages. On the other hand, *Decode 0*, *Decode 1*, *Branch*, *ALU* and *Load/Store* may receive multiple requests which have different priorities; therefore, the *StaticPriority Arbiter* is selected in these stages.

6.2 Observations

In this section, we discuss the observations made during the construction of OR1200-BNv3 in Bluenose II, both as a user and as a tool developer.

Building on the Eclipse platform, Bluenose II inherits some common editor features such files management, save to image, cut-and-paste, undo *etc.*. These features are accessed in the same ways as in popular editors (*e.g.*, a user can copy a selected block by pressing the control and c keys together). For example, a stage can be copied by selecting the stage then holding the control and C keys together, which is a common editor key sequence for copying. As a result, navigation around Bluenose II is intuitive. The cut-and-paste feature is particularly useful in constructing a superscalar pipeline where some stages are copies of or very similar to other stages. These stages can be configured just once then copied and pasted as a new stage in the pipeline. Connection routing is also built into Eclipse graphical editor but the functionality needs to be adjusted (in code) to return a look that is more like a schematic for the purpose of Bluenose II. In addition, the VHDL editor from the Signs plugin added the convenience of editing VHDL directly from Bluenose II. The ability to directly edit VHDL as well as some convenience features like connection routing did not exist in Bluenose I.

During the construction of OR1200-BNv3, we identified areas that can be improved. Although the use of record types has simplified connections, graphically connections are still in need of improvement. As mentioned in Section 6.1, there were some logics that were implemented and encapsulated in various *Custom Block*s. A few blocks appear a number of times in the pipeline. We are going to analyze these blocks and decide which of the following scenarios these blocks fit into:

1. the logic is related to some existing *PipeLib* components and it should be incorporated into them.

2. the logic occurs in many other pipelines and should be included as a new component in *PipeLib*.

3. the occurrences of the logic are infrequent or the logic is specific to only one pipeline so it should remain as a *Custom Block*.

One such block that we have identified is the *stage_valid* block. It has appeared in several stages in OR1200-BNv3. Its output is true if the request to next stage has been accepted. This logic is closely related to the functionality of and uses the same signals as *MkR/A*. Therefore, the logic of the *stage_valid* block will be incorporated into *MkR/A*.

As tool developers, we observed that the development in Java on Eclipse is also a strength of the tool. Although it takes a steep learning curve to learn to customize the GMF generated code and interact with the GMF and Eclipse runtime, once mastered, a lot of features can be easily inherited and extended. For example, in the event that a new block is added to *PipeNet*, the new block can inherit from the *Block* class to inherit its properties. As mentioned earlier, a lot of common editing features are already built into the Eclipse platform which saves the developers the tedious but non-trivial work of implementing these features.

6.3 Performance Comparison

6.3.1 Speed and Area

To compare the performance of the codes generated by Bluenose I and Bluenose II, both set of codes were synthesized with Mentor Graphics' Precision [41] with Altera's Stratix II set as the target technology. The speed and area information obtained from both sets of codes were then compared. A reference model style testbench was written to compare the behaviours of OR1200-BNv2 and OR1200-BNv3. The two designs were simulated and their program counters and write-back values were compared clock cycle by clock cycle. The code for the testbench is included in Appendix B.

Table 6.2 summarizes the speeds and areas of OR1200-BNv2 and OR1200-BNv3. OR1200-BNv3 is faster than OR1200-BNv2 by 11 MHz, uses 82 fewer look up tables (LUT) and 31 more registers. To understand the gain in speed, the critical paths of the two designs were examined. The timing reports of the two designs indicate that that both critical paths have the same source and destination nodes. Although the beginnings and the ends of the two critical paths are the same, the one in OR1200-BNv2 seems to involve more cells that the one from OR1200-BNv3. Figure 6.2 and Figure 6.3 show the critical paths of OR1200-BNv2 and OR1200-BNv3 respectively. The figures were created in the Gatevision PRO graphical netlist analyzer [45] by extracting and tracing the cells on the critical paths as reported by the timing analysis. The bold lines with arrows on the figures were drawn to show the paths. The numbers on Figure 6.2 indicate the order of the paths. There are two types of blocks in these figures and they are boolean gates and hierarchical blocks. Boolean gates are identified with the names prefixed with "ix" shown at the top of the blocks. Hierarchical blocks contain other gates. The timing analysis specifies which of the gates in the hierarchical blocks are on the critical path but some of them are abstracted away by the hierarchical blocks in the figures.

A possible cause for the gain in speed and LUT in OR1200-BNv3 may be the code and design clean-up in OR1200-BNv2 as we ported it into Bluenose II. For example, unread signals and their assignments were removed. In both OR1200-BNv2 and OR1200-BNv3, only the instructions in the fetching stages are discarded in the event that a branch is taken. In OR1200-BNv2, an abort signal is passed into the *Decode 1* stage even though it was not read inside the stage. In OR1200-BNv3, the abort signal is removed from the inputs of *Decode 1*. The probability that this kind of code clean-up is the cause for the gain may be unlikely since unread signals are "optimized away" by the synthesizer. More time is needed to understand the critical paths.

As a preliminary analysis, it is enough to know that the newly generated code performs as well as or better than the original code. In the future, the design of OR1200-BNv2 should be refactored the same way OR1200-BNv3 was and their performance re-evaluated. Finally, the increase in the number of registers in OR1200-BNv3 may be attributed to the use of record types. Synthesizer decomposes the elements in a record into individual signals. The elements that are not used in one stage are still passed to that stage if the whole record is passed. For example, the element for immediate data for load/store instruction (*lsu_imm*) in the record type for the output of the decoding stages (*decode_output_type*) is only relevant to *Load/Store*. However, the decoding stages also output to *ALU*. The *lsu_imm* is simply not accessed in *ALU*.

Table 6.2: Performance Comparison

	Speed	Area
OR1200-BNv2	60.62 MHz	2633 LUTs, 650 registers
OR1200-BNv3	71.63 MHz	2551 LUTs, 681 registers

Higgins compared the code generated by Bluenose I to code that was handcrafted, both sets of codes implemented OR1200-BNv2 [12]. Higgins' tests showed that there was no significant performance degradation in the code generated by Bluenose I. Since we do not have access to the handcrafted code, we are not able to directly compare that to the code generated by Bluenose II. However, since the performances of the codes generated by Bluenose II and Bluenose I are similar, we safely conclude that the performance of the code generated by Bluenose II is also comparable to the handcrafted code. Since OR1200-BNv2 and OR1200-BNv3 are reasonably complex, we expect that this result can be generalized for other pipeline designs as well. Future work may include implementation of other designs both manually and through Bluenose II and a more thorough analysis.

6.3.2 Code Quality

As mentioned in Section 3.6, the VHDL code generated by Bluenose II is made more concise than the code generated by Bluenose I through the use of record types and syntax from the VHDL-93 standard. Although the number of lines of generated code does not directly indicate the readability of the code, generally the fewer lines

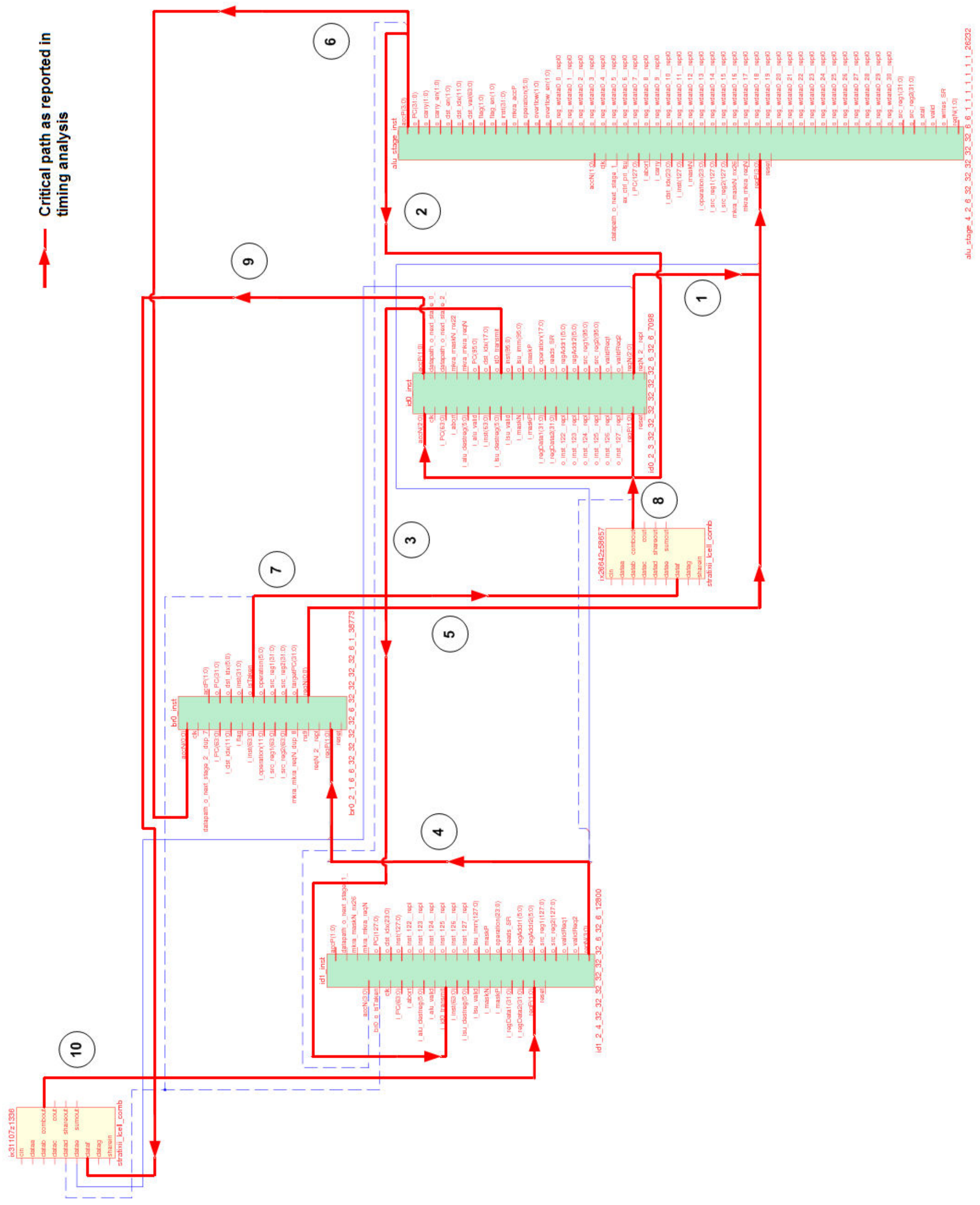


Figure 6.2: Critical path of OR1200-BNv2

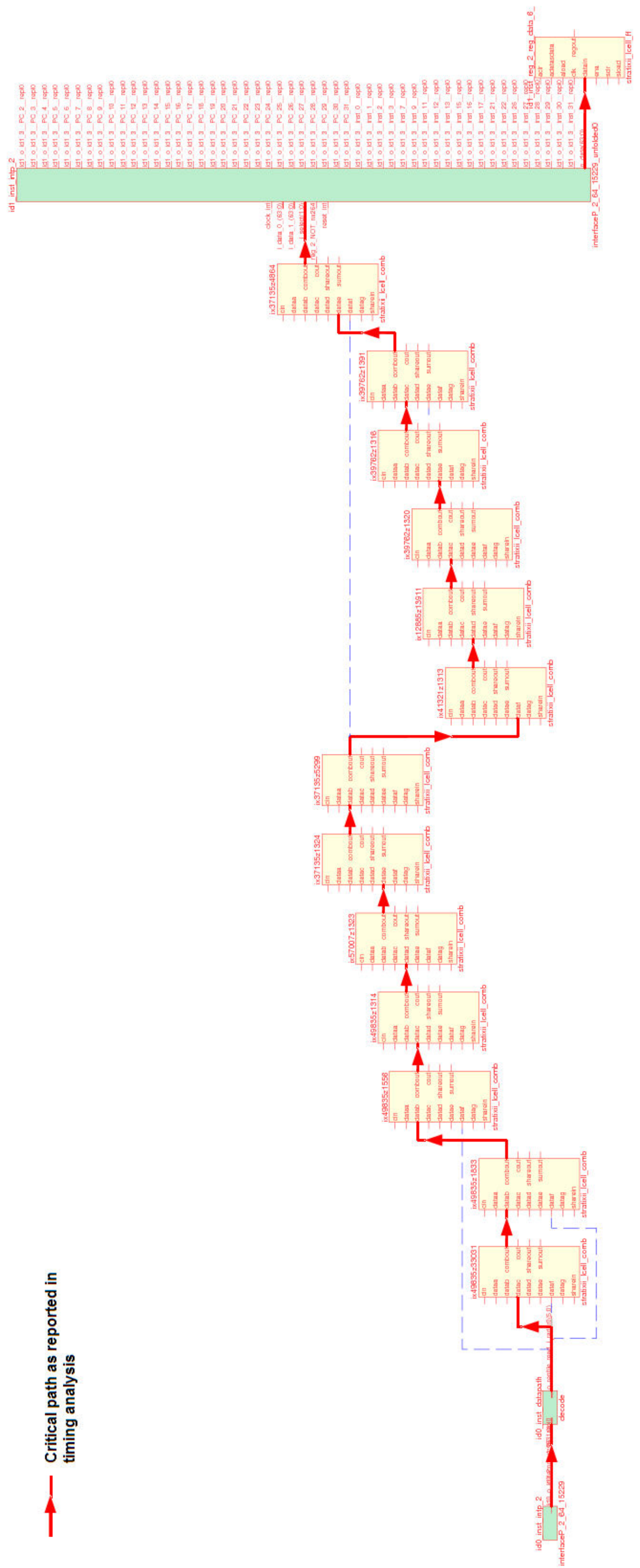


Figure 6.3: Critical path of OR1200-BNv3

the easier to peruse. Here we compared the numbers of lines of generated code as well as written codes to give a general idea of the conciseness of the codes.

In OR1200-BNv2, custom *MultiDelay MkR/A* blocks were used instead of the standard ones from *PipeLib*. In Bluenose II, it is important to identify the *MkR/A* component (*e.g.*, for HOL generation) in the model; therefore, we used the standard *MultiDelay MkR/A* and extract the extra logic from OR1200-BNv2's custom *Multi-Delay MkR/A* block into custom blocks. Consequently, the number of lines of code from *PipeLib* for OR1200-BNv3 is higher with the standard *MultiDelay MkR/A*. Secondly, Bluenose I does not have blocks that represent memory arrays; therefore, the generated code for the top level of the design had to be wrapped within another entity that included memory arrays for data and instruction caches. The number of lines of this extra code is included as custom code for OR1200-BNv3.

Table 6.3: Lines of code for the OpenRISC project

Category	Bluenose I	Bluenose II
Generated	4672	1589
PipeLib components	563	586
Datapath and other custom code	3445	3647 (228 of which were generated)

Finally, Bluenose II was able to successfully generate HOL script based on the OR1200-BNv3 design and the script is included in Appendix C.

6.4 Summary

A case study based on the implementation of Higgins' modification of OR1200 (OR1200-BNv2) was presented in this chapter. Higgins' design is a two-wide superscalar 32-bit RISC integer pipeline. Following the naming convention, the pipeline implemented in Bluenose II is named OR1200-BNv3. Features of the pipeline design were described. During the construction of OR1200-BNv3 in Bluenose II, we observed that the basic editor features were improved in Bluenose II from Bluenose I. We also identified some *Custom Block*s that appeared a few times in the pipeline which can be incorporated into some of the *PipeLib* components. As tool developer, we noted that the hierarchical block design of the *PipeNet* is highly extensible. A comparison of the performance of the codes generated by Bluenose I and Bluenose II was made. The speed of OR1200-BNv3 was 11 MHz faster than that of OR1200-BNv2 while it uses 82 fewer LUTs and 31 more registers. It is suspected that the design clean up such as the removal of unused signals from OR1200-BNv2 from OR1200-BNv3 has caused the slight improvement in speed and area. The increase in register is suspected to be caused by the use of record types. Record types force signals that are used in one stage but not another to be passed and registered. The qualities of the generated codes by Bluenose I and Bluenose II were also compared.

Bluenose II generated significantly fewer lines of codes and consequently improved the conciseness of the code.

Chapter 7

Conclusions and Future Work

7.1 Conclusions

In this thesis, we presented Bluenose II, a graphical tool for the design and verification of pipelined circuits. Bluenose II builds on the work of Bluenose I. Bluenose I demonstrated that the pipeline model from Aagaard's verification framework could be applied to pipeline design. Pipeline design and verification were also simplified by combining the pipeline model with a graphical user interface. Although Bluenose I is based on a pipeline model, the model was not explicit or clearly defined in Bluenose I. The pipeline model served as a guideline in the design of Bluenose I and did not play a fundamental role in its development. In this work, we gave form to this implicit pipeline model and developed *PipeNet*. *PipeNet* is not only used to drive the development of Bluenose II, it is also used to drive the pipeline design process. The hierarchical design of *PipeNet* matches the structure of the hardware and appeals to the intuition of the designer. Furthermore, *PipeNet*'s connection to Aagaard's verification framework means that a *PipeNet* model can take advantage of the framework's verification capabilities.

Bluenose II is developed as a plug-in that runs on the Eclipse platform. This allows Bluenose II to inherit basic editor functionalities from Eclipse and more importantly, it allows Bluenose II to be developed with Eclipse's GMF. GMF is a model driven development tool that generates graphical editors from user-defined models. We developed Bluenose II by specifying the *PipeNet* metamodel to GMF then customizing the graphical editor code generated by GMF. The *PipeNet* metamodel can be viewed as the abstract syntax of *PipeNet*. Its hierarchical block and connection design can accommodate future extensions to *PipeNet*. The VHDL and HOL generation modules were developed by extending the code generated by GMF. Bluenose II's VHDL generation module is modified to handle user-defined record types

The end product is a tool that allows the designer to use *PipeNet* to describe a pipeline design then generate VHDL code and HOL script from the pipeline model.

The HOL generation functionality allows HOL to be used as an external verification environment to Bluenose II. We also presented the HOL console which is a step towards integrating this external verification environment into Bluenose II.

We demonstrated the usefulness of Bluenose II and *PipeNet* in a case study based on OR1200-BNv3, an implementation of Higgins' design, OR1200-BNv2. The case study was chosen so that we could use OR1200-BNv2 as a reference to evaluate different aspects of Bluenose II. The performance of the generated VHDL code by Bluenose II is slightly better than that of Bluenose I, in terms of speed and area. In addition, the readability of the generated code by Bluenose II is also improved.

Based on the observations and analysis in the thesis, the following conclusions are drawn:

- The model driven approach can be applied to both the development of the pipeline design tool and pipeline design itself.
- *PipeNet*, the pipeline metamodel that was the driver of the Bluenose II development and the pipeline design process, captures the relevant characteristics of pipeline design. This is demonstrated by the ease to generate VHDL code and HOL script from the model described in *PipeNet*.
- Bluenose II is an improvement from Bluenose I in terms of the basic editor functionalities and the quality of the generated VHDL code.
- Bluenose II leads to faster design and verification of pipelined circuit by increasing the designer's productivity.

7.2 Future Work

Bluenose II is an ongoing project. During its development, we noted several areas of improvement which we left for future work due to time constraint and they are listed below.

- Provide additional validation

  Additional validation to a *PipeNet* model can be provided in two ways. One way is to specify more constraints on *PipeNet* to guide the designer in the modeling process. Currently, *PipeNet* only describes the structural constraints on a pipeline model (*e.g.*, number of *Arbiter* allowed in a *PipeStage*). Constraints to enforce the legal semantics of a pipeline model still need to be added to *PipeNet*. An example of such a constraint is "if a stage has more than one input, then the *Degenerate Arbiter* must not be selected for that stage". The second way to add validation to a *PipeNet* model is to let the designer write their own constraints on the model. The second way requires us to provide a feature in Bluenose II that not only allows the designer to enter additional constraints but to also check the pipeline model with the user defined constraints. This additional validation can be specified with OCL [33], which is supported by EMF.

- Connect to third party tools to harness their design and verification capabilities

  We envision Bluenose II to be an integrated development environment for the design and verification of pipelined circuits. The design and verification functionalities of Bluenose II can be extended by harnessing the design and verification capabilities of other tools. This can be done in two ways. One is to integrate these tools into Bluenose II either at source code level or as an external process as we demonstrated through the Signs plug-in and HOL. The second way is to generate code or script in formats that are supported by these tools so the code or script can be exported to these tools. An example of this is the HOL script generation. As described in the thesis, Signs' VHDL parser has been integrated into Bluenose II. Thanks to the Signs plug-in, the designer can also view and edit VHDL code directly in Bluenose II with VHDL specific support such as syntax highlighting. Signs' functionality is not limited to parsing VHDL, it also provides a VHDL simulator. Bluenose II may be able to provide simulation support using this simulator. The HOL console is a step towards a seamless connection between HOL and Bluenose II. HOL itself can be connected to other tools by dynamic loading and calling of external C functions. By integrating HOL into Bluenose II, we are also connecting to HOL's extensions. There are also many other tools that provide Java application programming interfaces (API) to allow access to their functionalities which may be useful in pipeline design and verification.

- Implement other generators

  Closely related to the previous point, by implementing generators to transform the *PipeNet* model to other languages, we can integrate Bluenose II into existing design flow or tool chain that supports the languages. Furthermore, a refactoring of the Java code may speed up the implementation of new generators. For example, the code for the traversal of the *PipeNet* model in the VHDL generation module may be reused in other generators.

- Extend *PipeNet* and explore its potentials

  There are three directions of the development of *PipeNet*. The first is to apply *PipeNet* outside Bluenose II. The popularity of a modeling language can be a validation of the language's quality. Similar to the first direction, the second direction is to show *PipeNet*'s capability to represent different hardware designs. So far, we have only shown the use of *PipeNet* in describing a microprocessor pipeline design. More case studies can be conducted to show *PipeNet*'s capability to express other designs. The third direction is to extend *PipeNet* to describe other aspects of pipeline design. *PipeNet* allows the designer to describe the structure of the design with hierarchical blocks. It may be extended or augmented by other models to describe other aspects such as the algorithm implemented in the hardware design, control hazards and data hazards. Using different models to represent different aspects of a system is not a new idea [13]. UML has different diagrams to represent different views of a system. Lessons may be drawn from other modeling languages such as UML.

- Develop better graphical representation

  A visually pleasing graphical user interface can potentially increase the designer's productivity. We made an effort to create a graphical representation that resembles a hardware schematic. A complex design involves many connections which can clutter the design area. A large design with all the components and connections can also be difficult to navigate. These issues need to be addressed otherwise Bluenose II can be rendered useless for a large and complex design. The use of VHDL record type and collapsible blocks are the first steps towards reducing "graphical clutter". We are currently investigating the idea of the abstract connection which is a graphical abstraction that "hides" other connections or present a group of related connections as one when the designer toggles on the feature. An abstract connection may be used to represent concepts such as the parcel flow in the pipeline.

- Conduct more case studies

  More case studies can be conducted to validate Bluenose II's usefulness. Comparison between the generated code and hand written code on the same design will be useful to evaluate *PipeNet*'s ease of use and the performance of the generated code. A designer's productivity is not increased if she can write code faster than create a pipeline model. On the other hand, it is not useful if the performance of the generated code is poor, even if the designer can quickly specify the pipeline model. It will also be worthwhile to implement some of the designs in related work (Section 2.6) to evaluate Bluenose II against other tools.

Appendix A

DiffAdd Code Generated by Bluenose I

Listing A.1: DiffAdd code generated by Bluenose I

```
architecture main of diffAdd is
...
component stage_neg
  generic (
    ...
  );
  port (
    clk: in std_logic;
    reset: in std_logic;
    reqP: in std_logic_vector((numInputs-1) downto 0);
    accP: out std_logic_vector((numInputs-1) downto 0);
    reqN: out std_logic_vector((numOutputs-1) downto 0);
    accN: in std_logic_vector((numOutputs-1) downto 0);
    i_d: in std_logic_vector((wordSize_i_d*numInputs-1) downto 0);
    o_n: out std_logic_vector((wordSize_o_n*numOutputs-1) downto 0);
    i3: in std_logic_vector((wordSize_i3*numInputs-1) downto 0);
    o3: out std_logic_vector((wordSize_o3*numOutputs-1) downto 0)
  );
end component;

  signal stage_neg_clk: std_logic;
  signal stage_neg_reset: std_logic;
  signal stage_neg_reqP: std_logic_vector((1-1) downto 0);
  signal stage_neg_accP: std_logic_vector((1-1) downto 0);
  signal stage_neg_reqN: std_logic_vector((1-1) downto 0);
  signal stage_neg_accN: std_logic_vector((1-1) downto 0);
  signal stage_neg_i_d: std_logic_vector((dpWidth*1-1) downto 0);
  signal stage_neg_o_n: std_logic_vector((dpWidth*1-1) downto 0);
  signal stage_neg_i3: std_logic_vector((dpWidth*1-1) downto 0);
  signal stage_neg_o3: std_logic_vector((dpWidth*1-1) downto 0);
...
begin
...
stage_neg_inst : stage_neg
  generic map (1, 1, dpWidth, dpWidth, dpWidth, dpWidth, dpWidth)
  port map (
  clk => stage_neg_clk,
  reset => stage_neg_reset,
  reqP => stage_neg_reqP,
  accP => stage_neg_accP,
  reqN => stage_neg_reqN,
  accN => stage_neg_accN,
  i_d => stage_neg_i_d,
  o_n => stage_neg_o_n,
  i3 => stage_neg_i3,
  o3 => stage_neg_o3
  );
...
accP(0) <= stage_sub_accP(0);
...
stage_neg_clk <= clk;
stage_neg_reset <= reset;
...
stage_add_reqP(0 downto 0) <= stage_neg_reqN(0 downto 0);
stage_neg_accN(0 downto 0) <= stage_add_accP(0 downto 0);
stage_add_i_dn(0 downto 0) <= stage_neg_o_n(0 downto 0);
stage_add_i3(0 downto 0) <= stage_neg_o3(0 downto 0);
...
stage_neg_reqP(0 downto 0) <= stage_sub_reqN(0 downto 0);
stage_sub_accN(0 downto 0) <= stage_neg_accP(0 downto 0);
stage_neg_i_d(0 downto 0) <= stage_sub_o_d(0 downto 0);
stage_neg_i3(0 downto 0) <= stage_sub_o3(0 downto 0);

end main;
```

Appendix B

Reference Model Testbench

```
library ieee;
use ieee.std_logic_1164.all;
use work.types_pkg.all;
library jason_lib;
use jason_lib.all;

entity openrisc_tb is
end openrisc_tb;

architecture main of openrisc_tb is

  signal clock, reset : std_logic :=
      '0';
  signal o_pc : fetch_input_vector(1
      downto 0);
  signal o_regfile_write0,
     o_regfile_write1 :
     regfile_write_input_type;
  signal ref_pc0, ref_pc1 :
     std_logic_vector(31 downto 0);
  signal o_reg_we0, o_reg_we1,
     o_flag : std_logic;
  signal o_reg_wdata0, o_reg_wdata1
     : std_logic_vector(31 downto
     0);
  signal ok_pc0, ok_pc1 : std_logic;
  signal ok_reg_wdata0,
     ok_reg_wdata1 : std_logic;

begin -- main

  uut : entity work.openrisc
port map(
  clock => clock,
  reset => reset,
  o_pc => o_pc,
  o_regfile_write0 =>
     o_regfile_write0,
  o_regfile_write1 =>
     o_regfile_write1);

ref : entity jason_lib.
   openrisc_top
port map (
  clock => clock,
  reset => reset,
  o_pc(63 downto 32) => ref_pc1,
  o_pc(31 downto 0) => ref_pc0,
  o_reg_wdata0 => o_reg_wdata0,
  o_reg_wdata1 => o_reg_wdata1,
  o_reg_we0 => o_reg_we0,
  o_reg_we1 => o_reg_we1,
  o_flag => o_flag);

clockProc: process
begin -- process
  wait for 10 ns;
  clock <= not clock;
end process;

resetProc: process
begin -- process
  reset <= '0';
  wait for 35 ns;
  reset <= '1';
  wait for 40 ns;
  reset <= '0';
  wait;
end process;

check_pc0 : process
begin
  wait until rising_edge(clock);
  if(ref_pc0 = o_pc(0).i_pc) then
    ok_pc0 <= '1';
  else
    ok_pc0 <= '0';
  end if;
end process;

check_pc1 : process
begin
  wait until rising_edge(clock);
  if (ref_pc1 = o_pc(1).i_pc) then
    ok_pc1 <= '1';
  else
    ok_pc1 <= '0';
  end if;
end process;
```

```
  check_reg_wdata0 : process
  begin
    wait until rising_edge(clock);
    if (o_reg_wdata0 =
        o_regfile_write0.i_wdata)
        then
      ok_reg_wdata0 <= '1';
    else
      ok_reg_wdata0 <= '0';
    end if;
  end process;

  check_reg_wdata1 : process
  begin
    wait until rising_edge(clock);
    if (o_reg_wdata1 =
        o_regfile_write1.i_wdata)
        then
      ok_reg_wdata1 <= '1';
    else
      ok_reg_wdata1 <= '0';
    end if;
  end process;

end main;
```

Appendix C

Generated HOL Script Based on OR1200-BNv3

Listing C.1: Generated HOL Script Based on OR1200-BNv3

```
new_constant("FETCH", Type`:'c signal -> 'd signal -> 'a signal -> 'b
    signal -> bool vector signal -> ckt`);
Define `if0 O_ICACHE I_ICACHE = stage exclusive_arb [(discard_opt,CONST T
    )] (FETCH O_ICACHE I_ICACHE)`;
Define `if1 O_ICACHE I_ICACHE = stage exclusive_arb [(discard_opt,CONST T
    )] (FETCH O_ICACHE I_ICACHE)`;
new_constant("DECODE", Type`:'c signal -> 'd signal -> 'a signal -> 'b
    signal -> bool vector signal -> ckt`);
Define `id0 O_REGFILE_READ I_REGFILE_READ = stage static_arb [(
    enableReqN_opt,CONST F)] (DECODE O_REGFILE_READ I_REGFILE_READ)`;
new_constant("BRANCH", Type`:'c signal -> 'd signal -> 'e signal -> 'f
    signal -> 'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `br0 I_VALID O_ISTAKEN O_TARGETPC I_FLAG = stage static_arb [] (
    BRANCH I_VALID O_ISTAKEN O_TARGETPC I_FLAG)`;
new_constant("ALU", Type`:'c signal -> 'd signal -> 'e signal -> 'f
    signal -> 'g signal -> 'h signal -> 'i signal -> 'j signal -> 'k
    signal -> 'a signal -> 'b signal -> bool vector signal -> ckt`);
Define `alu_stage I_CLK I_RESET I_VALID I_CARRY I_START O_FINISHED
    O_STALL O_ALU_LOOP O_REGFILE_SR = stage static_arb [(enableReqN_opt,
    CONST F)] (ALU I_CLK I_RESET I_VALID I_CARRY I_START O_FINISHED
    O_STALL O_ALU_LOOP O_REGFILE_SR)`;
new_constant("LSU", Type`:'c signal -> 'd signal -> 'e signal -> 'f
    signal -> 'g signal -> 'h signal -> 'i signal -> 'a signal -> 'b
    signal -> bool vector signal -> ckt`);
Define `lsu_stage I_CLOCK I_RESET I_DCACHE O_DCACHE I_START O_FINISHED
    O_STALL = stage static_arb [(enableReqN_opt,CONST F)] (LSU I_CLOCK
    I_RESET I_DCACHE O_DCACHE I_START O_FINISHED O_STALL)`;
new_constant("WB", Type`:'c signal -> 'a signal -> 'b signal -> bool
    vector signal -> ckt`);
```

```
Define ‘wb0 I_VALID = stage exclusive_arb [] (WB I_VALID)‘;
Define ‘id1 O_REGFILE_READ I_REGFILE_READ = stage static_arb [(enableReqN_opt,CONST F)] (DECODE O_REGFILE_READ I_REGFILE_READ)‘;
Define ‘wb1 I_VALID = stage exclusive_arb [] (WB I_VALID)‘;
new_type("uninterpreted_intern_type_1",0);
new_type("uninterpreted_intern_type_2",0);
new_type("uninterpreted_intern_type_3",0);
new_constant("two_to_one", Type‘:(uninterpreted_intern_type_2, bool, bool) pcl signal -> (uninterpreted_intern_type_1, bool, bool) pcl signal‘);
Define
‘ openrisc reset pclP pclN if0_O_ICACHE if0_I_ICACHE if1_O_ICACHE if1_I_ICACHE id0_O_REGFILE_READ id0_I_REGFILE_READ br0_I_VALID br0_O_ISTAKEN br0_O_TARGETPC br0_I_FLAG alu_stage_I_CLK alu_stage_I_RESET alu_stage_I_VALID alu_stage_I_CARRY alu_stage_I_START alu_stage_O_FINISHED alu_stage_O_STALL alu_stage_O_ALU_LOOP alu_stage_O_REGFILE_SR lsu_stage_I_CLOCK lsu_stage_I_RESET lsu_stage_I_DCACHE lsu_stage_O_DCACHE lsu_stage_I_START lsu_stage_O_FINISHED lsu_stage_O_STALL wb0_I_VALID id1_O_REGFILE_READ id1_I_REGFILE_READ wb1_I_VALID =
  intern pcl_if0_id0:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_if0_id1:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_if1_id1:(uninterpreted_intern_type_1, bool, bool) pcl signal
 pcl_id1_id0:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_id0_br0:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_id0_alu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_id0_lsu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_id1_br0:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_br0_alu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_alu_stage_alu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_id1_alu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_alu_stage_id0:(uninterpreted_intern_type_3, bool, bool) pcl signal
 pcl_alu_stage_id1:(uninterpreted_intern_type_3, bool, bool) pcl signal
 pcl_alu_stage_wb0:(uninterpreted_intern_type_3, bool, bool) pcl signal
 pcl_id1_lsu_stage:(uninterpreted_intern_type_2, bool, bool) pcl signal
 pcl_lsu_stage_id1:(uninterpreted_intern_type_3, bool, bool) pcl signal
 pcl_lsu_stage_id0:(uninterpreted_intern_type_3, bool, bool) pcl signal
 pcl_lsu_stage_wb1:(uninterpreted_intern_type_3, bool, bool) pcl signal
 .
  begin_ckt
    if0 if0_O_ICACHE if0_I_ICACHE reset [& pclP &] [& pcl_if0_id0; pcl_if0_id1 &];
    if1 if1_O_ICACHE if1_I_ICACHE reset [& pclP &] [& pcl_if1_id1 &];
    id0 id0_O_REGFILE_READ id0_I_REGFILE_READ reset [& pcl_if0_id0; (two_to_one pcl_id1_id0) &] [& pcl_id0_br0; pcl_id0_alu_stage; pcl_id0_lsu_stage &];
    br0 br0_I_VALID br0_O_ISTAKEN br0_O_TARGETPC br0_I_FLAG reset [&
       pcl_id0_br0; pcl_id1_br0 &] [& pcl_br0_alu_stage &];
    alu_stage alu_stage_I_CLK alu_stage_I_RESET alu_stage_I_VALID
       alu_stage_I_CARRY alu_stage_I_START alu_stage_O_FINISHED
       alu_stage_O_STALL alu_stage_O_ALU_LOOP alu_stage_O_REGFILE_SR
       reset [& pcl_alu_stage_alu_stage; pcl_id0_alu_stage;
       pcl_br0_alu_stage; pcl_id1_alu_stage &] [& pcl_alu_stage_id0;
       pcl_alu_stage_id1; pcl_alu_stage_wb0 &];
    lsu_stage lsu_stage_I_CLOCK lsu_stage_I_RESET lsu_stage_I_DCACHE
       lsu_stage_O_DCACHE lsu_stage_I_START lsu_stage_O_FINISHED
       lsu_stage_O_STALL reset [& pcl_id1_lsu_stage; pcl_id0_lsu_stage &]
        [& pcl_lsu_stage_id1; pcl_lsu_stage_id0; pcl_lsu_stage_wb1 &];
    wb0 wb0_I_VALID reset [& pcl_alu_stage_wb0 &] [& pclN &];
    id1 id1_O_REGFILE_READ id1_I_REGFILE_READ reset [& pcl_if0_id1;
       pcl_if1_id1 &] [& pcl_id1_alu_stage; pcl_id1_id0; pcl_id1_br0;
       pcl_id1_lsu_stage &];
    wb1 wb1_I_VALID reset [& pcl_lsu_stage_wb1 &] [& pclN &]
  end_ckt
‘;
```

Glossary of Terms

ASIC Application Specific Integrated Circuit

DSL Domain Specific Language

EMF Eclipse Modeling Framework

GEF Eclipse Graphical Editing Framework

GMF Eclipse Graphical Modeling Framework

HOL In this thesis, HOL refers to the HOL theorem prover which stands for Higher Order Logic.

MDA Model Driven Architecture, a framework proposed by the Object Management Group

OMG Object Management Group

PIM Platform Independent Model

PSM Platform Specific Model

RTL Register Transfer Level

References

[1] E. Mah, “Worldwide PC processor market hits record levels of unit shipments again in 4q07, says IDC,” *DIGITIMES*, January 22, 2008. `http://www.digitimes.com/mobos/a20080122PR202.html` (current as of July 15, 2008). 1

[2] M. Reardon, “Cell phone sales hit 1 billion mark,” *CNET News Blog*, February 2008. `http://news.cnet.com/8301-10784_3-9881022-7.html` (current as of July 15, 2008). 1

[3] C. Sorensen, “1 million iPhone 3Gs sold in 3 days,” *thestar.com*, July 15, 2008. `http://www.thestar.com/Business/article/460175` (current as of July 16, 2008). 1

[4] D. L. Walker, “The longest day,” *Washington Post Magazine*, p. W20, August 5, 2007. `http://www.washingtonpost.com/wp-dyn/content/article/2007/08/01/AR2007080101720_2.html` (current as of July 16, 2008). 1

[5] M. McNamara, “ESL handoff: closer than you think,” *Embedded.com*, July 8, 2008. `http://www.embedded.com/design/208803191` (current as of July 12, 2008). 1, 16

[6] L. Wirbel, “Embedded designers on tighter schedules, juggling multiple projects in 2008,” *EE Times*, July 3, 2008. `http://www.eetimes.com/miu/showArticle.jhtml?articleID=208802378&pgno=1` (current as of July 15, 2008). 1

[7] D. A. Patterson and J. L. Hennessy, *Computer Architecture: A Quantitative Approach.* Morgan Kaufmann Publishers, Inc., 1990. 2

[8] J. R. Burch and D. L. Dill, “Automatic verification of pipelined microprocessors control,” in *Proceedings of the sixth International Conference on Computer-Aided Verification CAV* (David L. Dill, ed.), vol. 818, (Stanford, California, USA), pp. 68–80, Springer-Verlag, 1994. 2

[9] I. Gavrichenkov, “Getting ready to meet Intel Core 2 Duo: Core microarchitecture unleashed,” *X-bit labs*, June 29, 2006. `http://www.xbitlabs.com/articles/cpu/display/core2duo-preview.html` (current as of July 16, 2008). 2

[10] N. Mokhoff, “Intel, Motorola report formal verification gains,” *EE Times*, June 21, 2001. http://www.eetimes.com/story/OEG20010621S0080 (current as of July 16, 2008). 2

[11] M. Aagaard and M. Leeser, “Reasoning about pipelines with structural hazards,” in *TPCD '94: Proceedings of the Second International Conference on Theorem Provers in Circuit Design - Theory, Practice and Experience*, (London, UK), pp. 13–32, Springer-Verlag, 1994. 3, 6, 9, 22, 25

[12] J. Higgins, “Simplifying the design and verification of pipelined circuits,” Master's thesis, University of Waterloo, 2004. 3, 6, 7, 21, 25, 29, 64, 71

[13] K. Czarnecki, “Overview of generative software development,” in *Unconventional Programming Paradigms, International Workshop UPP 2004, Le Mont Saint Michel, France, September 15-17, 2004, Revised Selected and Invited Papers*, pp. 326–341, 2004. 3, 10, 11, 79

[14] OpenCores, *OpenRISC 1000 Architecture Manual*, April 2006. Rev 1.3. 3, 64

[15] D. Lampret, *OpenRISC 1200 IP Core Specification.* OpenCores, Sep 2001. Rev. 0.7. 3, 64

[16] S. W. Ambler and R. Jeffries, *Agile modeling: effective practices for extreme programming and the unified process.* New York, NY, USA: John Wiley & Sons, Inc., 2002. 10

[17] J. Mukerji and J. Miller, *Technical Guide to Model Driven Architecture: The MDA Guide.* Object Management Group, Inc., v1.0.1 ed., 2003. http://www.omg.org/cgi-bin/doc?omg/03-06-01 (current as of Mar. 23, 2008). 10, 11

[18] Object Management Group, Inc., *Meta Object Facility (MOF) Core Specification*, v2.0 ed., January 2006. http://www.omg.org/spec/MOF/2.0/ (current as of July 13, 2008). 10, 11

[19] “The Eclipse Modeling Framework (EMF) overview,” 2005. http://help.eclipse.org/ganymede/index.jsp?topic=/org.eclipse.emf.doc/references/overview/EMF.html (current as of Mar. 9, 2008). 11, 60

[20] “The eclipse development platform.” http://www.eclipse.org (current as of Mar. 9, 2008). 11

[21] “Graphical modeling framework.” http://www.eclipse.org/gmf (current as of Mar. 9, 2008). 12

[22] C. Kern and M. R. Greenstreet, “Formal verification in hardware design: a survey,” *ACM Trans. Des. Autom. Electron. Syst.*, vol. 4, no. 2, pp. 123–193, 1999. 14

[23] M. J. C. Gordon and T. F. Melham, eds., *Introduction to HOL: a theorem proving environment for higher order logic.* New York, NY, USA: Cambridge University Press, 1993. HOL4 website: http://hol.sourceforge.net. 14

[24] M. Gordon, "From LCF to HOL: a short history," *Proof, language, and interaction: essays in honour of Robin Milner*, pp. 169–185, 2000. 14, 15

[25] P. Sestoft, *Moscow ML Library Manual.* Moscow ML, 2000. 15

[26] Mentor Graphics Corporation, "HDL designer." `http://www.mentor.com/products/fpga_pld/hdl_design/hdl_designer_series/` (current as of July 12, 2008). 16

[27] Altera, "Quartus II." `http://www.altera.com/products/software/products/quartus2/qts-index.html` (current as of July 12, 2008). 16

[28] M. Franklin, E. Tyson, J. Buckley, P. Crowley, and J. Maschmeyer, "Auto-pipe and the X language: a pipeline design tool and description language," *Parallel and Distributed Processing Symposium, 2006. IPDPS 2006. 20th International*, pp. 10 pp.–, April 2006. 16, 18

[29] D. A. Mathaikutty and S. K. Shukla, "MCF: A metamodeling-based component composition frameworkcomposing SystemC IPs for executable system models," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 792–805, July 2008. 16

[30] D. Mathaikutty, S. Kodakara, A. Dingankar, S. Shukla, and D. Lilja, "MMV: A metamodeling based microprocessor validation environment," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 16, pp. 339–352, April 2008. 16

[31] "The generic modeling environment." `http://www.isis.vanderbilt.edu/projects/gme/` (current as of July 5, 2008). 17

[32] Object Management Group, Inc., *UML Specification*, v2.1.2 ed., November 2007. `http://www.omg.org/spec/UML/Current` (current as of July 13, 2008). 17

[33] Object Management Group, Inc., *Object Constraint Language Specification*, v2.0 ed., May 2006. `http://www.omg.org/technology/documents/formal/ocl.htm` (current as of July 13, 2008). 17, 25, 78

[34] W. Mueller, A. Rosti, S. Bocchio, E. Riccobene, P. Scandurra, W. Dehaene, and Y. Vanderperren, "UML for ESL design - basic principles, tools, and applications," *IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*, pp. 73–80, Nov. 2006. 17

[35] P. Mishra, A. Shrivastava, and N. Dutt, "Architecture description language (ADL)-driven software toolkit generation for architectural exploration of programmable SOCs," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 11, no. 3, pp. 626–658, 2006. 18

[36] Target Compiler Technologies, "What is nML?." `http://www.retarget.com/products/whatisnml.php` (current as of July 14, 2008). 18

[37] J. Hoe and Arvind, "Operation-centric hardware description and synthesis," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 23, pp. 1277–1288, Sept. 2004. 18

[38] N. Dave, "Designing a reorder buffer in Bluespec," *Formal Methods and Models for Co-Design, 2004. MEMOCODE '04. Proceedings. Second ACM and IEEE International Conference on*, pp. 93–102, June 2004. 19

[39] C. Brooks, E. A. Lee, X. Liu, S. Neuendorffer, Y. Zhao, and H. Zheng, "Heterogeneous concurrent modeling and design in java (volume 1: Introduction to Ptolemy II)," Tech. Rep. UCB/EECS-2008-28, EECS Department, University of California, Berkeley, Apr 2008. 19

[40] G. Bartsch, *Signs - VHDL Hardware Developement.* Institut fuer Technische Informatik. `http://www.iti.uni-stuttgart.de/~bartscgr/signs/wiki/index.php/Main_Page` (current as of Mar. 6, 2008). 23, 26

[41] Mentor Graphics Corporation, *Precision RTL Synthesis Style Guide*, release 2004c update1 ed., February 2005. 33, 70

[42] P. J. Ashenden, *The Designer's Guide to VHDL.* San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2001. 35

[43] *HOL Reference Page.* `http://hol.sourceforge.net/kananaskis-4-helpdocs/help/HOLindex.html` (current as of July 23, 2008). 47

[44] Sun Microsystems, Inc., *JavaTM 2 Platform Standard Edition 5.0 API Specification*, 2004. `http://java.sun.com/j2se/1.5.0/docs/api/` (current July 23, 2008). 50, 53

[45] Concept Engineering, *Gate/SpiceVision Documentation*, 2.8.1 ed., 2005. 70