High-Level CSP Model Compiler for FPGAs

Rohit M. Asthana

Thesis submitted to the Faculty of the
Virginia Polytechnic Institute and State University
in partial fulfillment of the requirements for the degree of

Master of Science
in
Computer Engineering

Peter M. Athanas, Chair
Paul E. Plassmann
Patrick R. Schaumont

December 14, 2010
Blacksburg, Virginia

Keywords: FPGAs, High-Level Synthesis, Communicating Sequential Processes (CSP), Models of Computation (MoC), Autocode Generation

High-Level CSP Model Compiler for FPGAs

Rohit M. Asthana

(ABSTRACT)

The ever-growing competition in current electronics industry has resulted in stringent time-to-market goals and reduced design time available to engineers. Lesser design time has subsequently raised a need for high-level synthesis design methodologies that raise the design to a higher level of abstraction. Higher level of abstraction helps in increasing the predictability and productivity of the design and reduce the number of bugs due to human-error. It also enables the designer to try out different optimization strategies early in the design stage. Inspite of all these advantages, high-level synthesis design methodologies have not gained much popularity in the mainstream design flow mainly because of the reasons like lack of readability and reliability of the generated register transfer level (RTL) code. The compiler framework presented in this thesis allows the user to draw high-level graphical models of the system. The compiler translates these models into synthesizeable RTL Verilog designs that exhibit their desired functionality following communicating sequential processes (CSP) model of computation. CSP model of computation introduces a good handshaking mechanism between different components in the design that makes designs less prone to timing violations during implementation and bottlenecks while in actual operation.

Acknowledgements

There are many people who have supported me directly or indirectly throughout the completion of my graduate studies. I feel very fortunate to have a loving family who has been a constant support for me throughout my life. I would like to express my sincere gratitude to my parents and my sister.

I would like to sincerely thank my advisor Dr. Peter M. Athanas, for giving me the opportunity to work in the CCM Lab. He has always been a constant support and motivation. His work ethics have always been an inspiration for me and will continue to be so. It has been a privilege working with him and a tremendously rewarding experience.

I would also like to thank Dr. Paul E. Plassmann and Dr. Patrick R. Schaumont for serving on my thesis committee.

I would like to thank my friends in the CCM Lab for all the fond memories, especially all the exciting foosball games. It was a real pleasure working with you all.

Thanks to Abhay, Aveek, Sriram, Aravind and Sushrutha for proofreading the thesis. Finally, thanks to all my friends in Blacksburg for all the fun.

Contents

List of Figures

Chapter 1

Introduction

The ever-growing consumer electronics market has resulted in a phenomenal density growth in today's silicon chips. This density growth has followed the Moore's law [1] for almost six decades, and it has become common to have billions of transistors packaged on a single system-on-chip (SoC). Today's advanced designs have become complex to design and verify using traditional design methodologies. Moreover, aggressive competition in current electronics industry has resulted in stringent time-to-market goals and reduced design time available to engineers.

In addition, large variety of discrete components are being integrated on current SoCs. A typical SoC consists of a processor core, high speed communication units, signal processing components and memory blocks. Design and testing of SoCs is challenging and involves considerable time and money. FPGAs serve as low-cost reprogrammable rapid prototyping platforms for SoCs, because of their soft and hard processor cores, configurable logic blocks, signal processing units and memory blocks.

In the context of SoCs, the traditional design methodology relies on various electronic design automation (EDA) tools with two major stages; a register transfer level (RTL) specification, followed by logical and physical synthesis. Because of the lesser design time available to the designer these days, there is a need for design methodologies that raise the design to a higher level of abstraction. High-level design descriptions make it possible for designers to focus on functionality and target performance, than spending their time on debugging RTL. A classic analogy can be drawn from the way by which C abstracted much of the complexities of programming in assembly.

Various high-level synthesis tools have been developed by different EDA companies. These tools either take a graphical block diagram description of a design or a description in computer programming language like C and SystemC. The design is then translated into a synthesizeable RTL description. The primary goal of these tools is to increase the predicability and productivity of the design by raising the level of abstraction, and to introduce more automation to the design flow and thereby, reduce the number of bugs introduced due to human-error.

High-level synthesis tools have been around for almost two decades now, yet they have been rather slow in being adopted to the mainstream design flow. Many reasons have been given for their slow adoption, such as bad readability of the generated RTL code, less reliability, and large and less efficient code for a desired cost-optimized solution [2]. However, the use of high-level synthesis tools increases the accessibility of hardware design to many more engineers, researchers and domain experts. Graphical block diagram design methodology has potential to add algorithm developers, researchers and system engineers to the current pool of design engineers, creating ten times more people to achieve challenging design time goals.

Multiple high-level synthesis tools were investigated in a study at Texas Instruments [3]. In the study, they found some tools to be focussed either on control logic or on data-path thus limiting their applicability. While others had custom constructs of languages, making their integration with the existing design flows inefficient. However, one advantage was very noticeable - a design implemented through these tools indeed reduced the design time, accelerating the implementation process and reducing the occurrence of bugs due to human-error that are routine in RTL code. It was also easier to try out the design with different operating clock frequencies and explore the tradeoffs.

This research work addresses the problem of the lack of wide acceptance of contemporary high-level synthesis tools. The compiler framework presented in this work allows the user to draw a high-level graphical block diagram of a design and it generates a structural RTL description of the design that exhibits the desired behavior following communicating sequential processes (CSP) model of computation (MoC).

The Ptolemy modeling environment developed by the University of California at Berkeley [4] is used as a front-end graphical user interface (GUI), through which the user can draw graphical block diagrams using various building blocks available in the actor library of the Ptolemy tool. The compiler framework has been developed in Java and it generates RTL models of the design in Verilog that can be synthesized using standard synthesis tools. The RTL generated by the compiler is a structural RTL Verilog netlist containing instances of various CSP Verilog models, a library of which has been developed as a part of this framework containing basic primitives necessary to design a digital hardware system.

Listed below are the main contribution of this research work:

1. A new design methodology for FPGA is presented that allows the user to draw graphical models of a design in the Ptolemy modeling environment. The compiler translates

these models into a structural RTL netlist in Verilog that can be synthesized using the standard synthesis tools.

2. A library containing CSP Verilog models of the primitives that operate under CSP model of computation is developed.

3. A compiler framework has been developed in Java that extracts the information out of the XML models saved from the Ptolemy tool. The compiler then interprets this extracted information to generate Verilog RTL models of the design that exhibit the desired behavior following the CSP model of computation.

1.1 Thesis Organization

This thesis is organized into five chapters. Chapter 1 presents the motivation of this work and gives a brief overview of the work. Chapter 2 discusses necessary background topics that would help the reader understand this work better. Some related work is also discussed in this chapter. Chapter 3 presents the design methodology for FPGAs using this compiler and an overall description of compiler design. Chapter 4 describes selected sample experiments performed using this compiler and presents a comparative evaluation of the compiler with other popular high-level synthesis tools. Lastly, Chapter 5 summarizes the work and discusses future work.

Chapter 2

Background

In this work, the Ptolemy II modeling environment developed by the University of California at Berkeley [4] is used as a design entry tool for FPGAs. The Communicating Sequential Processes (CSP) model of computation is used for autocode generation by the compiler developed in this work. This chapter covers the basic theory of background topics which would help the reader to understand this work better.

This chapter begins with an overview of the Ptolemy II modeling environment and how models are built in Ptolemy using Vergil, a GUI where models are built pictorially. The theory of Models of Computation is discussed in Section 2 with a subsection on choosing a model of computation. The next section discusses about the CSP model of computation, which is being used in this work and then, the concept of high-level synthesis is discussed. In the last section, some of the contemporary high-level synthesis tools are discussed.

2.1 Ptolemy II

Ptolemy II is an open-source heterogenous modeling and design environment, developed by the Center for Hybrid and Embedded Software Systems (CHESS) in the Department of Electrical Engineering and Computer Sciences of the University of California at Berkeley [4]. It provides a platform for modeling, simulation, and design of concurrent, real-time hardware systems. It is a modeling environment that lets the user systematically build the system using multiple models of computation, hierarchically combined. It uses eXtensible Markup Language (XML) format for its data representation. It provides a graphical user interface (GUI) called as Vergil to graphically construct models of a system.

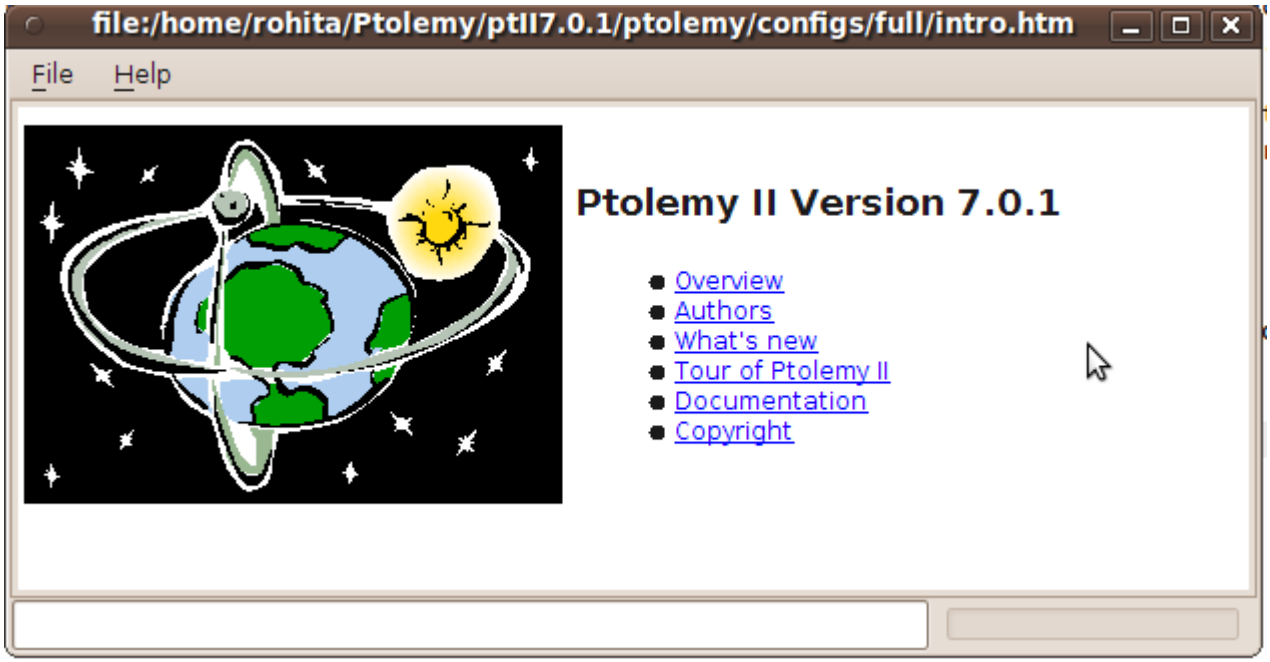

Figure 2.1 Ptolemy II (screenshot captured from Ptolemy GUI)

In this work, Ptolemy is used as a high-level modeling environment, because it also has the capability of doing CSP simulation, which is the model of computation of choice for this work, which will be later described in detail in Chapter 3.

2.1.1 Modeling and Design

The Ptolemy platform is used for heterogeneous modeling, simulation and design of concurrent systems. Modeling and Design can be defined as [5]:

Modeling is the act of formally describing a system. A model can be mathematical, wherein it can be viewed as a set of assertions about properties like functionality or physical dimensions of a system. There can also be a constructive model, which defines a computational functionality of the system, they are executable models that describe the behavior of a system in response to a stimulus from outside the system.

Design is the act of defining a system and refining the models of system until the desired functionality is obtained under the existing constraints.

Modeling and design are always closely interlinked with each other. Executable models can also be called *simulations*, where the executable model is distinct from the system it models. In hardware systems, an executable model eventually transforms into an implementation of a system [5].

2.1.2 Actor-Oriented Design

The basic principle behind modeling in Ptolemy environment is *actor-oriented design*. All the primitives present in the available library are called *actors* and they execute and interact with other actors in a model. All actors have a well-defined interface which abstracts the

internal structure and behavior of the actor and describes how that actor can interact with its environment. For instance, usually, an interface would include I/O ports of an actor and parameters that configure the behavior of that actor.

All actors have communication channels that transfer data from one port to another according to the scheme the actor is operating in. The use of channels for the communication implies that the actors communicate directly with the channels that they are connected to and not directly with other actors.

Like actors, a model can also define an interface, this interface is now referred to as a *hierarchical abstraction*. This interface consists of external ports and parameters which are different from the internal ports and parameters of the individual actors in the model. Now, these external ports of a model can be connected to other communication channels or to the ports of actors comprising the model. Hierarchical Abstraction of an actor is clearly shown in Figure 2.2:

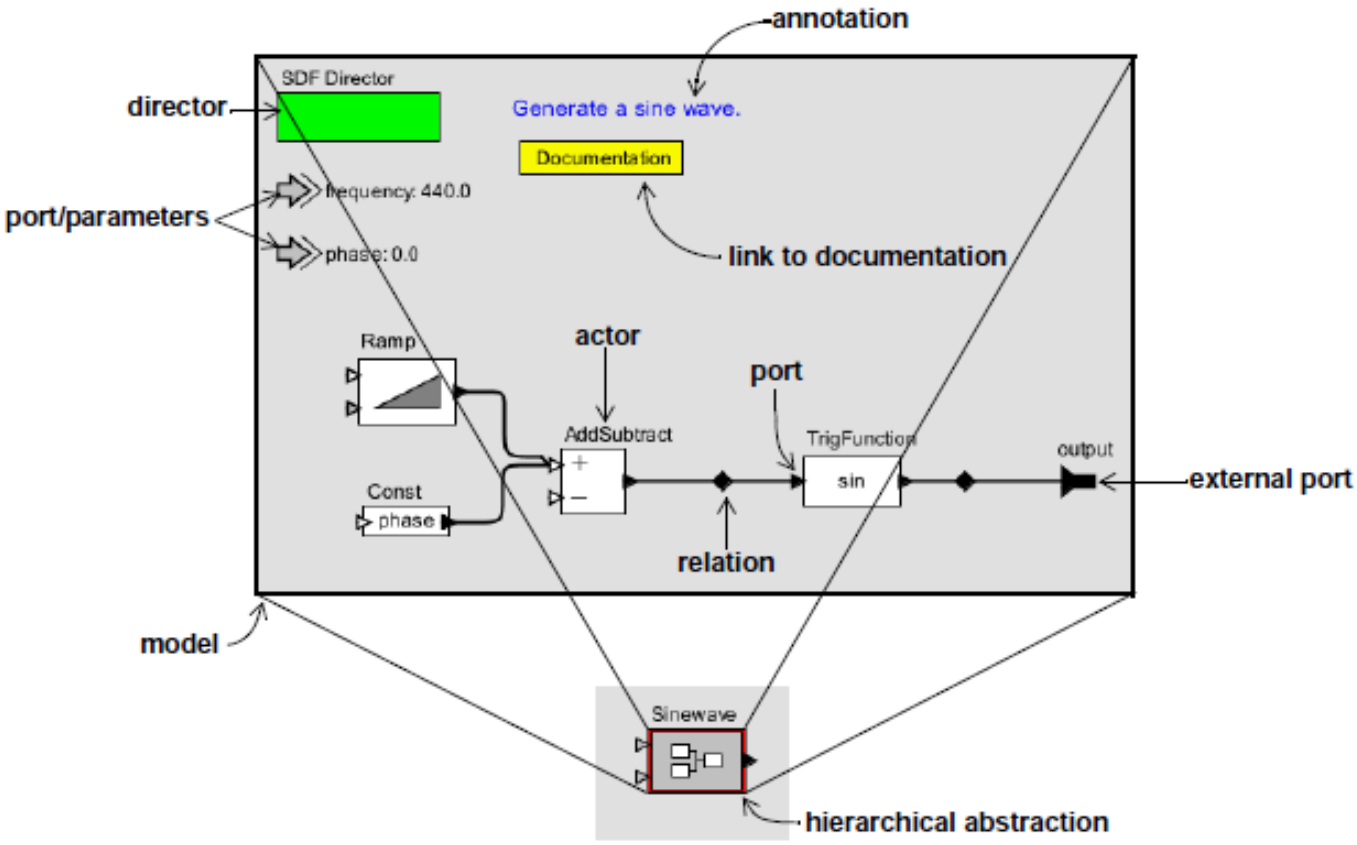

Figure 2.2 Hierarchical Abstraction of an Actor (figure from [5]) Used under Fair Use, 2010

The concepts of models, actors, ports, parameters and communication channels, taken all together constitute an actor-oriented design. Actor-oriented design has been popular for sometime now and is widely used in programs such as Simulink from Mathworks [6], LabVIEW from National Instruments [7] etc.

2.2 Models of Computation

There are various models of computation (MoC) that deal with concurrency and time in different ways. Each of them describes a scheme that governs the interaction between various components. Models of computation are useful in that they provide modeling properties that are applicable to all similar models. In an actor-oriented design, the model of computation explains how the behavior of the whole system is the result of the behavior of each of its actors. For any application, an appropriate model of computation should not impose unnecessary constraints and at the same time, it should be constrained enough to lead the system to desired results. For any specific application, selecting an appropriate model of computation is often challenging.

Models of computation like Communicating Sequential Processes - CSP, Rendezvous, Distributed Discrete Events - DDE and Process Networks - PN etc. are thread-oriented, so the actors in these MoCs in Ptolemy implement their own Java threads. While other MoCs like Continuous Time - CT, Discrete Events - DE and Synchronous DataFlow (SDF) implement their own scheduling between the actors. Refer to the Section 1.3 of "Volume 1: Introduction to Ptolemy II" for further insight into each of these MoCs [5].

2.2.1 Choosing Models of Computation

Selecting an appropriate model of computation from various existing MoCs for a particular application can be challenging to a designer. It is getting even harder as the level of abstraction and domain-specificity of a system both rise. Sophisticated and highly user-friendly graphical user interfaces (GUIs) are needed to enable designers to cope with this challenge.

Modeling of time is a distinct difference between different concurrent models of computation.

Continuous time MoCs take time to be a real number that advances continuously and place events on a time axis or other continuous signals on this axis. Popular examples of continuous time MoC usage are Simulink [6] and Modelica [8]. On the other hand, discrete time MoCs take time to be discrete and might even take time to be a constraint imposed by causality; thus, interpreting results in instances of time that are partially ordered. Partially ordered time provides a basis for analyzing and comparing different MoCs. Verilog and VHDL are examples of Discrete Time MoC.

The most basic approach would be to find a MoC that serves all the purposes. Thus, a mixture of all the available MoCs will be able to do the task. But, such a mixture of all the MoCs will be very complicated and difficult to use. Another approach perhaps would be to choose one MoC and show that the other MoCs are included as special cases, but this approach fails to acknowledge the strengths and weaknesses of each model of computation. Thus, to design elegant systems, one may need to use heterogeneous models having a mixture of multiple appropriate models of computation.

2.3 CSP

The Communicating Sequential Processes (CSP) is a model of computation that is used in this work for writing Verilog models of the actors currently supported. CSP is used for describing the scheme of interaction in highly concurrent systems. It was first proposed by Hoare in 1978 [9]. CSP models a system as a network of processes communicating with each other solely by passing tokens/data through unidirectional channels synchronously. If a process has a data token ready to be sent, it blocks until the receiving process is ready to accept that data token. This is generally referred to as *back pressure* in hardware systems which can be defined as a 'stall' signal in the backward direction that is used to prevent

overflow [10]. Likewise, if a process is ready to accept the token, it blocks until the sending process is ready to send the token. The following figure shows the communication protocol between the two processes:

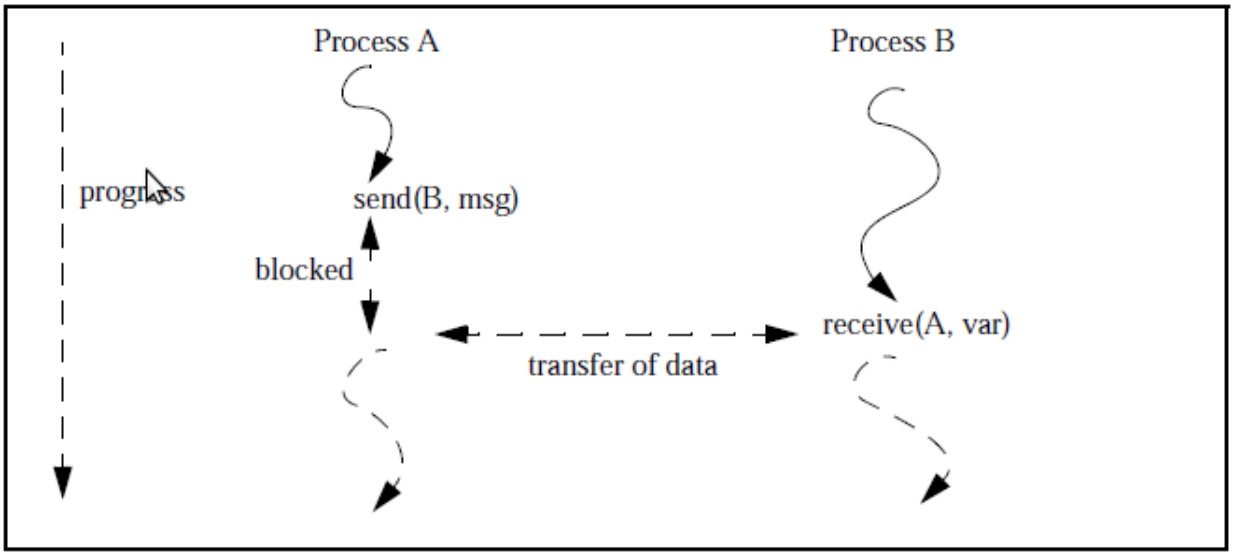

Figure 2.3 CSP (figure from [11]) Used under Fair Use, 2010

"Sequential" in the name CSP can be misleading at times, since CSP allows its processes to be defined both as sequential processes, and also as a concurrent mixture of more primitive processes. The interaction between different processes and that of a process with its environment, can be described using Process Algebraic operators. This approach makes complicated system descriptions to be easily constructed from a few primitive elements/actors. The construction of complex system descriptions through process algebraic operators can be better understood by knowing the basic terminology and principal operators of CSP, which are presented in the next section.

2.3.1 Basic Terms and Operators

Terms

There are two basic terms in CSP process algebra [9]:

Event: Events represent communications or interactions, and they are assumed to be instantaneous.

Process: Processes represent fundamental behaviors. For instance, *STOP* refers to a process that communicates nothing or a deadlock, and *SKIP* represents successful termination.

Operators

All the communication taking place between different processes due to the triggering of different events is represented by the use of algebraic operators. Some of the principal algebraic operators of CSP as described in [9, 12, 13] are:

Prefix: Let x be an event and let P be a process. Then

$$x \rightarrow P \tag{2.1}$$

represents a process which first engages in an event x and then, behaves exactly as described by P.

Deterministic Choice: This operator allows the future evolution of a process to be defined as a choice between the two component processes, and allows the environment to resolve the choice by triggering an initial event for one of the processes. For example,

$$(x \rightarrow P) \Box (y \rightarrow Q) \tag{2.2}$$

represents a process which can communicate events x and y, and then behaves as either P

or Q depending on which initial event is triggered by the environment. If both a and b were triggered at the same time, the choice would be resolved nondeterministically.

Nondeterministic Choice: This operator allows the future evolution of a process to be defined as a choice between two component processes, but does not allow the environment to have any control over the selection of the component processes. For example,

$$(x \rightarrow P) \sqcap (y \rightarrow Q) \tag{2.3}$$

represents a process which can refuse to accept x or y, and is only obliged to communicate if the environment offers both x and y. Nondeterminism can be inadvertently introduced into a normally deterministic choice if the initial events of both sides of the choice are identical.

Interleaving: The Interleaving operator represents completely independent concurrent activity. For example,

$$P \mid\mid\mid Q \tag{2.4}$$

is a process that behaves as P and Q simultaneously. The events from both processes are arbitrarily interleaved in time.

Interface Parallel: This operator represents a concurrent activity that requires synchronization between the component processes. Any event in the interface set can only occur when all the component processes engage in that event. For example,

$$P \mid [\{x\}] \mid Q \tag{2.5}$$

represents a process that requires P and Q to be able to perform event x before that event can occur.

Hiding: This operator is used to abstract processes from one another, by making some events unobservable. For example,

$$(x \rightarrow P) \setminus x \tag{2.6}$$

represents a process, which assuming event x does not occur in P, simply reduces to

$$P \tag{2.7}$$

2.3.2 Time and Deadlock

CSP is a highly concurrent model of computation and because of its nature, this model of computation is non-deterministic, as a process can be blocked waiting to send or receive tokens on any number of communication channels. High-level system modeling is one of the primary applications of CSP, which is the objective of this thesis. A notion of time may or may not be added to this model of computation, it is referred to as *Timed CSP* sometimes when a notion of time is added to it [11].

Before further discussing about the notion of time, the concept of a deadlock in CSP should be understood. Like its literal meaning, a deadlock in CSP refers to a situation when all the processes trying to communicate are either blocked or are delayed. Deadlocks can be classified as real deadlocks and time deadlocks.

Real Deadlock is a situation when all the active processes trying to communicate are blocked, while *Time Deadlock* on the other hand refers to a situation when all the active processes trying to communicate are either blocked or are delayed, and at least one of them is delayed.

The notion of time is centralized in CSP, in that all processes in a certain model share the same time known as the Current Model Time. A process is referred to as a *delayed process* if it has waited for some time relative to the current model time, or if it can wait for time

deadlock to occur at the current model time. To delay a process for some interval of time, it is suspended until that amount of time has advanced, at which stage it wakes up and continues.

Time may also be advanced when all the processes trying to communicate are either blocked or delayed. If it is a time deadlock, the delayed processes are woken up without advancing time. Alternatively, the current model time is advanced just enough to wake up at least one process.

2.4 High-level Synthesis

Hardware can be designed at various levels of abstraction. A Y-chart proposed by Gajski and Kuhn in 1983 (see Figure 3.4) gives a detailed model of the abstraction levels, elegantly depicting the relationship between description domains and levels of abstraction. The three radial lines represent the three description domains: the Behavioral, Structural and Geometrical domains. And, the circles represent the levels of abstraction in the three domains.

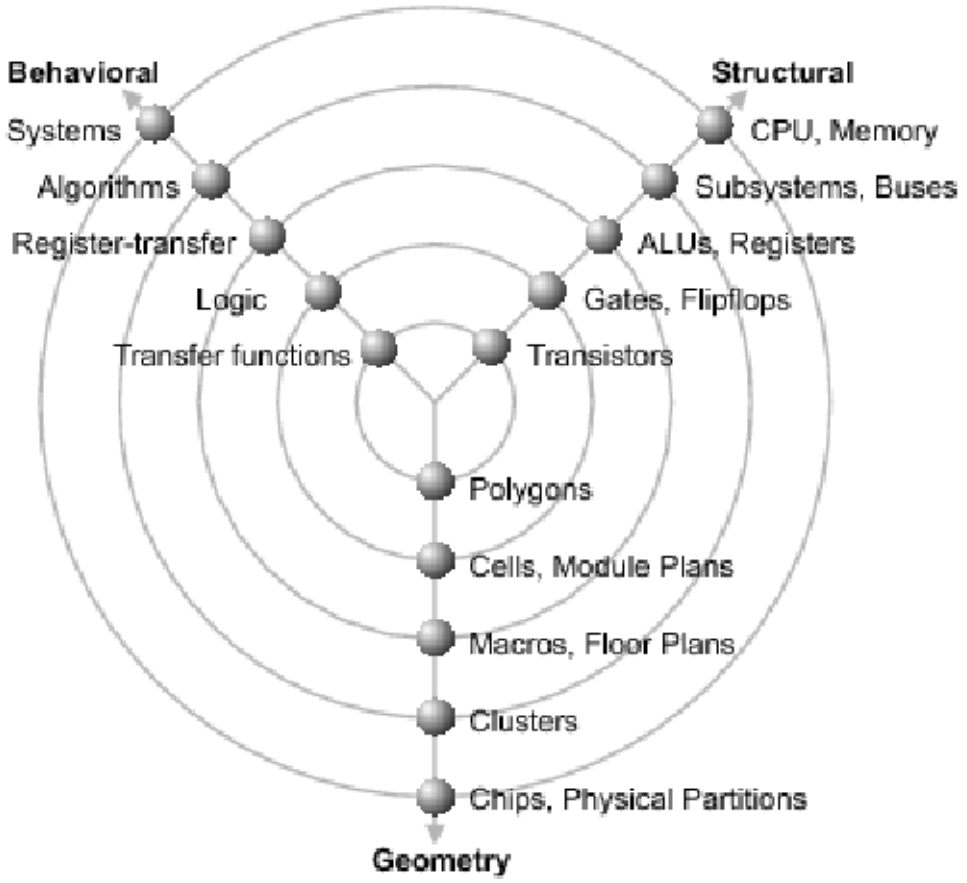

Figure 2.4 Y-Chart (figure from [14]) Used under Fair Use, 2010

When we are describing the behavior of a hardware, the most commonly used levels of abstraction are gate level, register transfer language (RTL) level and algorithmic/behavioral level. *High-level Synthesis* refers to the design process of describing behavior of a hardware at a higher level of abstraction. The "higher level of abstraction" here can be a C/SystemC code or a graphical block diagram of the hardware with basic building blocks like adders, multipliers, comparators etc. The main goal of high-level synthesis is to give hardware designers better control over optimization of their design, since higher the abstraction level is, lesser it is constrained. In the next section, some of the popular contemporary design tools that are platforms for high-level synthesis are briefly discussed.

2.5 Related Work

High-level synthesis tools have become increasingly popular in recent times as platforms for rapid prototyping. In this section, the design methodologies of some of the popular high-level synthesis tools are discussed.

2.5.1 Simulink

Simulink is a popular commercial tool developed by The Mathworks for modeling, simulation and analysis of hardware systems [6]. It provides an interactive GUI environment and a customizable set of block libraries. A designer can create a high-level diagram of a complex system using the building blocks from the available libraries. It is tightly integrated with MatLab and lets the designer create custom blocks by using well-defined MatLab functions.

The Simulink HDL Coder feature from Simulink allows the designer to generate bit-true and cycle-accurate synthesizeable HDL code from the high-level Simulink models. The model can also contain embedded MatLab code and Stateflow diagrams. The generated code is synthesizeable but it should go through functional simulation before the actual implementation. A testbench is also generated by the Simulink HDL Coder for this purpose. Other EDA tools like ModelSim, Xilinx ISim, Synopsys VCS etc. can be used for simulation. Once the code is verified to be functionally correct, it can go through the synthesis tool-chain for actual implementation, like Xilinx ISE [15] for implementation on FPGAs.

This design methodology works fine for simple systems, but it suffers from a problem of no back pressure mechanism to prevent bottlenecks while modeling complicated systems because of the lack of a well-defined model of computation in its design methodology. The model would look to be correct but may not function correctly after actual implementation,

because actual hardware is a lot more constrained than high-level graphical block diagram modeling and there are many datapath contention issues that should be addressed well in advance. This work tries to address this problem and how it does that is discussed in Chapter 3.

2.5.2 Impulse C

Impulse C is a high-level synthesis tool developed by Impulse Accelerated Technologies [16]. Similar to this research work, Impulse C is based on the CSP model of computation. It provides a development environment for a programming language similar to C and has C-compatible function libraries which are built for parallel programming, particularly for applications targeting FPGAs. The CoDeveloper tool allows you to compile C-language directly into optimized digital logic, which can be downloaded onto FPGA targets.

The basic principle behind Impulse C is to divide the high-level code into several blocks or processes. These blocks/processes communicate with each other through data-streams, carrying critical information from one process to another. These processes can be classified as a Producer process, a Consumer process or a Hardware process. The Producer is like an input to a software program, it generates input datastream to the Hardware process. The Consumer process is similar to the output of a software program, it receives output datastream from the Hardware process and transmits this information to the user. The Hardware process is the one by description of which Impulse C generates hardware, all the modification and processing of data takes place here. The following figure depicts the design methodology of this tool as compared to a standard C application:

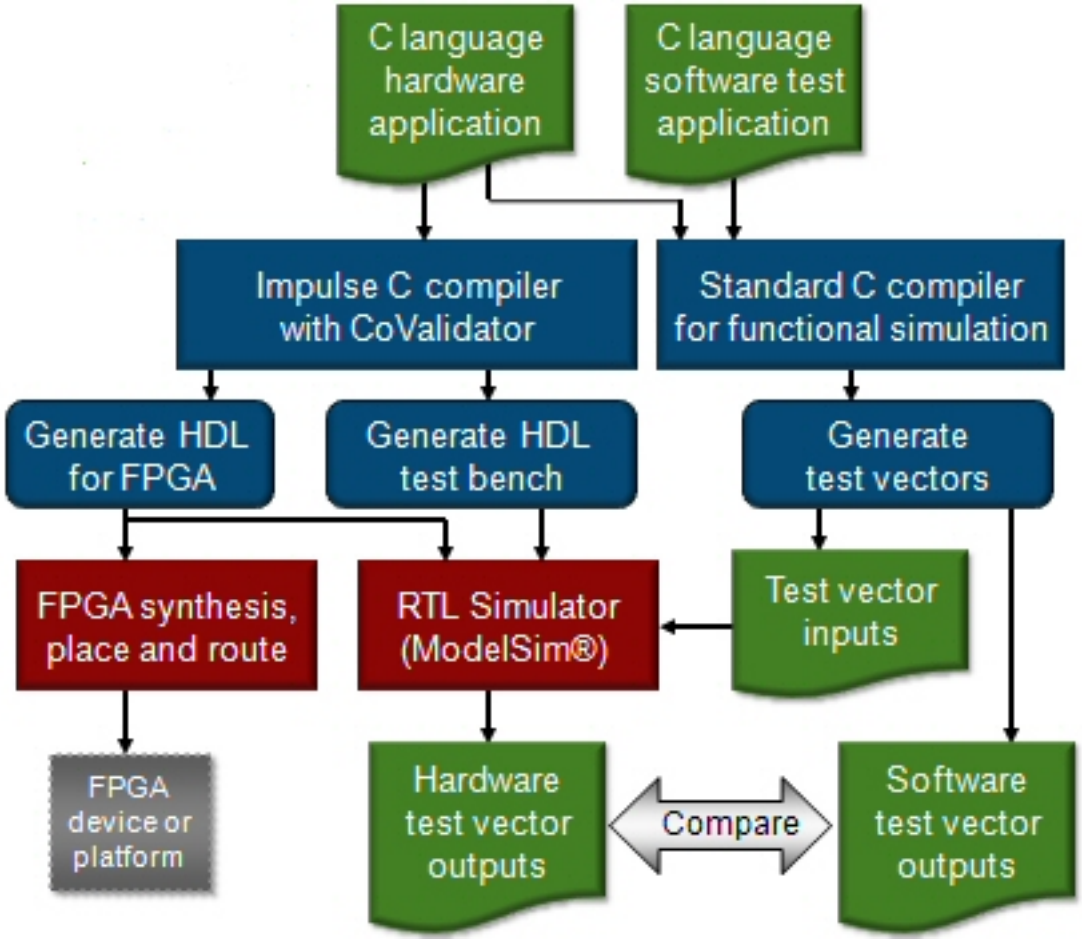

Figure 2.5 Impulse C Design Methodology (figure from [17]) Used under Fair Use, 2010

As shown in Figure 2.5, the Impulse C compiler takes input in the form of an application behavior description in the C programming language, that contains function libraries oriented towards parallel execution. It transforms the C description of the application to an HDL description of a digital hardware. It also generates an HDL testbench that can be used for functional verification. The generated HDL design can then be fed to standard synthesis toolchains, like Xilinx ISE [15] or Altera Quartus [18] for FPGAs. Functional verification can be done by various simulation tools like ModelSim, Xilinx ISim, Synopsys VCS etc.

2.5.3 Element CXI

Elemental Computing Array (ECA) is an architecture developed by Element CXI [19]. It is a highly scalable architecture consisting of heterogeneous functional Elements that can perform a variety of "Elemental" operations. These Elements communicate hierarchically via a network on chip and are capable of being assembled into more complex functions under hardware control in real-time. Tasks involving a large amount of number crunching are distributed across Elements for maximum speed and parallelism, while simple tasks time-share elements. ECA architecture claims to be more power efficient than ASICs and exhibits faster reconfiguration times as compared to FPGAs.

This architecture is based on Clusters, containing configurable Elements and a High-Speed Peripheral Interface (HSPI) interface to the host processor. The basic functional unit in ECA Architecture is an Element. 16 of these Elements and a Message Manager comprise a Cluster, which is the smallest unit of hardware scaling. Any type of logic can be built from a Cluster and it is divided into four zones of 4 Elements each. Everything in this architecture from Clusters on up is scalable, even across multiple ECA devices.

There are 7 types of programmable Elements present in this architecture and each Cluster contains the same set of Element types. Their operations range from 16-bit MACs, to 32-bit arithmetic and logical functions, to sorting, shuffling, and extraction of bits. A set of up to 3 operations can be performed per clock cycle by each Element. These Elements are interconnected by flexible, high-speed network. Each of these Elements are configured to behave in a certain manner. Memory used to store their configuration is called Context Memory, the configuration of the Context determines the operating mode, interpretation of the input data, knowledge of significant input and output destination. A collection of such configurations, called as task, can be loaded, started, halted or swapped out as a single unit.

The communication between the Elements in the Cluster as well as that between different Clusters is governed by the CSP model of computation.

Elemental Ptolemy is an open source tool suite built on top of Berkeley's Ptolemy II modeling environment (described earlier in this chapter). It can be used to create applications by drawing diagrams using ECA primitives. As an experiment, an 8-point radix-2 FFT butterfly structure was constructed with ECA primitives in Elemental Ptolemy, which is shown in Figure 2.6

The main drawbacks with this tool and architecture are size and no floating point capability. The biggest ECA device currently available is ECA256, which has 256 elements, and can be used for designing compact benchmarks for research but its not sufficient to design large complex systems. Also, its a purely fixed-point device and does not currently support floating-point implementations which makes it very difficult to use for signal processing and similar applications.

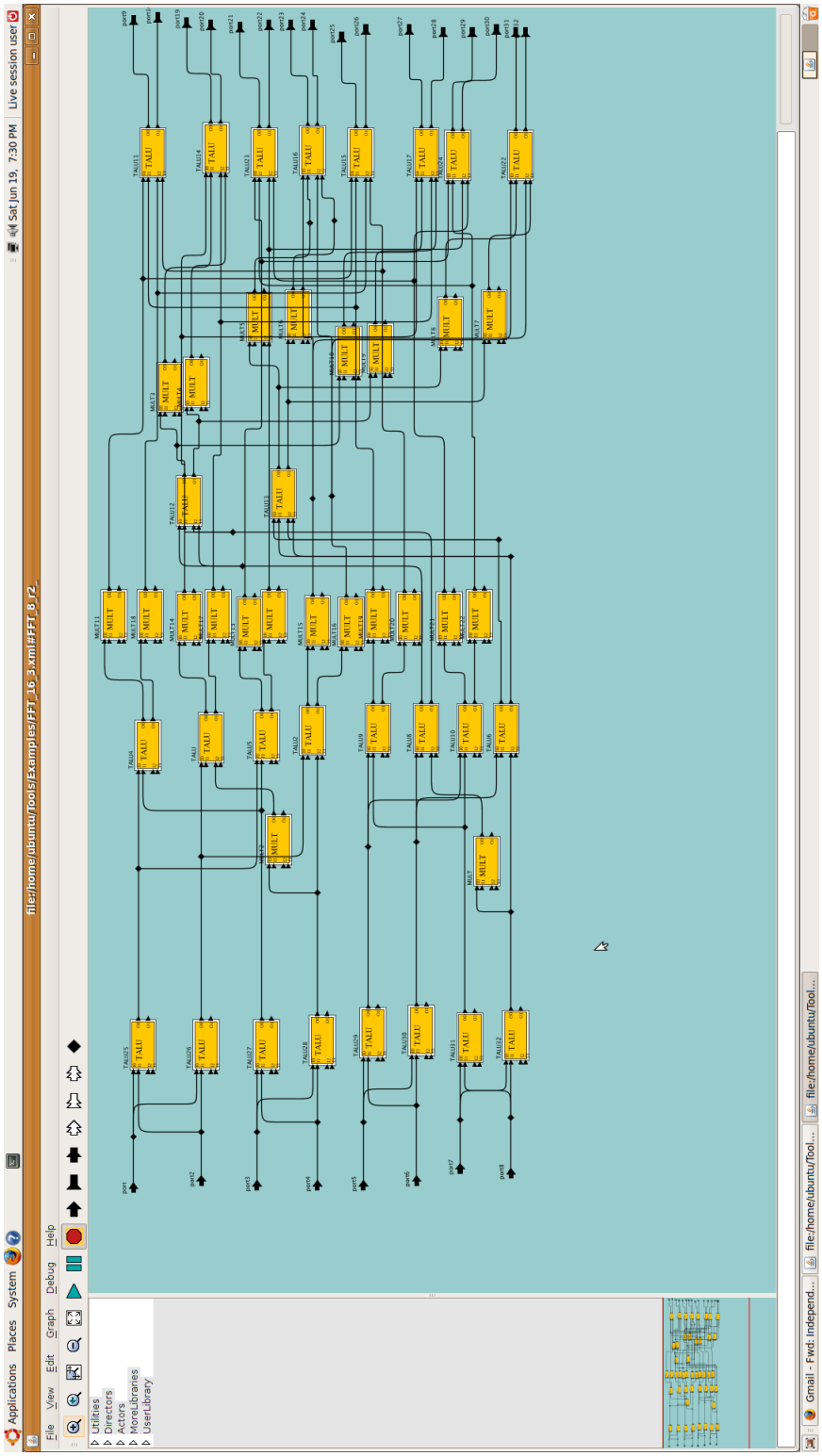

Figure 2.6 8-point FFT on ECA (screenshot captured from Elemental Ptolemy)

2.5.4 Summary

As described in this section, there are tools and architectures available for high-level synthesis that differ from one another in their design methodologies. Simulink and Element CXI use a system block diagram approach for their design entry, whereas Impulse C provides a development environment for C customized for parallel programming. The model of computation these tools use is a key factor in determining how the design is actually implemented. This thesis work, described in detail in next chapter, aims to develop a tool that can target a variety of hardware architectures and use the CSP model of computation that works well in resolving bottlenecks in massively parallel architectures like FPGAs.

Chapter 3

Design of the Compiler

This chapter describes the details of the high-level CSP model compiler developed in this work and the design methodology for FPGAs using this compiler. In the beginning, the typical FPGA design flow is discussed followed by a design methodology that uses this compiler for high-level synthesis (refer to Section 2.4) on FPGAs. In the next section, CSP Verilog models developed in this work are discussed. In the last section, the methodology of Verilog HDL generation by the compiler is presented.

3.1 Typical FPGA Design Flow

FPGAs are reconfigurable semiconductor devices with programmable logic and interconnects. The primary components of most moderns FPGAs are Configurable Logic Blocks (CLBs), Block Random Access Memories (BRAMs), Input/Output Blocks (IOBs), Digital Signal Processing unit (DSPs) and programmable interconnects. Reconfigurability, partial reconfiguration of portion of a design and low non-recurring engineering costs of FPGAs make them popular as rapid prototyping platforms for high-level synthesis.

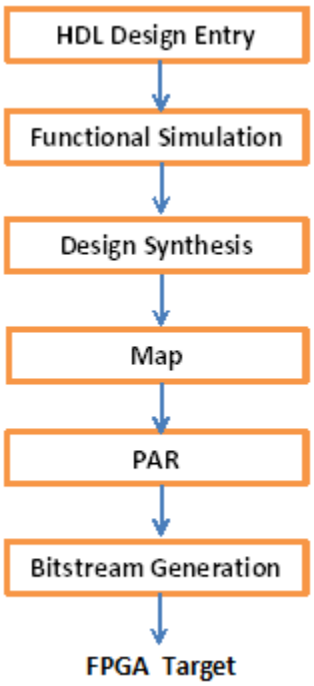

Figure 3.1 Typical FPGA Design Flow

Figure 3.1 depicts the major stages in a typical FPGA design flow. It starts with a behavioral description of a design in Hardware Description Languages (HDLs) like Verilog and VHDL. The HDL description of the design is verified for functionality by simulation in tools like ModelSim [20] etc. The verified code is synthesized into gate-level netlists by synthesis tools. The synthesized netlist should then go through *Map* for technology mapping of gate-level netlists into resources available on the targeted architecture. It should then go through *PAR* or *Place and Route* for placement of the mapped logic onto the available resources and routing to make connections between the placed logic. Finally, the routed design goes through *bitstream generation* to generate a binary bitstream that can be downloaded onto a FPGA target.

3.2 Design Flow using CSP model Compiler

In this work, the Ptolemy II modeling environment developed by the University of California at Berkeley [4] is used as a design entry tool. The high-level model of a design can be drawn in Ptolemy tool using the currently supported primitives by this work (refer to Section 3.3). Ptolemy tool has the capability to do CSP simulation, so a high-level model of a design can be simulated for functionality by adding a CSP director for the model. A director is a component added in the model to simulate a model as per a particular MoC. There are various kinds of such directors available in Ptolemy. After CSP simulation, the design can be saved in form of an XML file.

The saved XML file from the Ptolemy tool should then be given as input to the compiler developed in this work that would generate an equivalent Verilog netlist for the high-level model. In this work, CSP Verilog models have been developed for most of the basic arithmetic and logical primitives present in Ptolemy. The compiler generates a top-level Verilog netlist that contains the mapping of all the primitives in the design by instantiating and netlisting all corresponding CSP Verilog models from the library.

This top-level netlist not only contains all the connections between the primitives in the model, but this tool also generates CSP handshaking signals by itself along all the previously existing wires to let the primitives communicate to each other by CSP model of computation that would result in a more reliable design, which is less prone to issues like bus contention and timing violations.

The HDL generated by the compiler can then be given to standard synthesis tool-chains, like Xilinx ISE for Xilinx FPGAs or Altera Quartus for Altera FPGAs. The design flow using this compiler is shown in Figure 3.2:

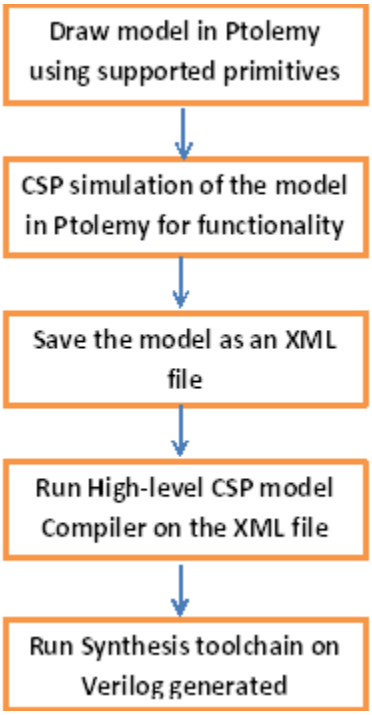

Figure 3.2 FPGA Design Flow using CSP Model Compiler

3.3 CSP Verilog Models

In this section, the methodology of creating CSP Verilog models equivalent to actors in Ptolemy is described. There is an exhaustive actor library present in Ptolemy, through which models can be drawn. Corresponding to most of the arithmetic and logical actors present in Ptolemy, Verilog models that operate as per CSP model of computation have been developed in this work. The basic principle behind these models is to exhibit the desired arithmetic/logical behavior while following the CSP handshaking protocol. Figure 3.3 shows an example of a CSP model with all the handshaking signals.

"Equals" is an actor available in the Ptolemy logical actor library. It produces a boolean output that is true or '1' if the inputs are equal, otherwise false or '0'. Figure 3.3 shows

a wrapper of CSP Verilog model equivalent of the Equals actor. 'a' and 'b' are two data inputs and 'out' is the boolean output. In a CSP Verilog model, there is a FIFO attached to every input port and there are handshaking signals that make the Verilog model to operate under CSP model of computation that helps in resolving bottlenecks in complex systems. The FIFO attached to the input ports with all its control signals is shown in Figure 3.4.

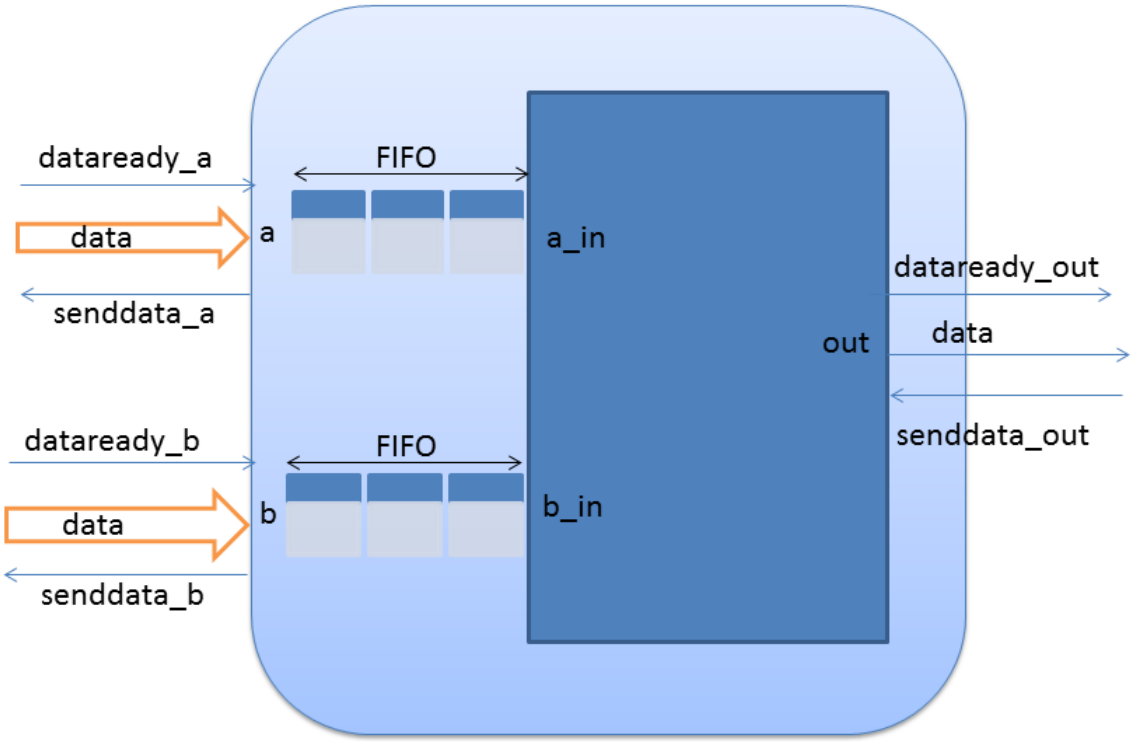

Figure 3.3 Equals CSP model

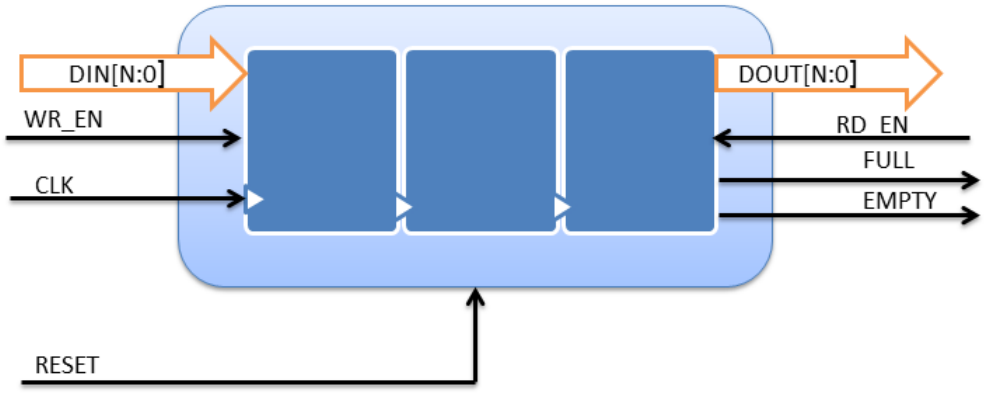

Figure 3.4 FIFO

When the sender of data 'a' is ready with the data token, it asserts the 'dataready_a' signal, same is the case with the sender of 'b'. Upon the assertion of these dataready signals, the CSP protocol mechanism in the model checks to see if there is space available in the attached FIFO by checking the 'full' output signal from the FIFO (see Figure 3.4). If the 'full' signal is not asserted, the model asserts the senddata signal to the sender. The 'wr_en', i.e. write enable signal of the FIFO is asserted when the outgoing senddata signal and the incoming dataready signal are both asserted. Thus, when the model asserts the senddata signal, the sender sends the data token and at this time, the 'wr_en' signal of the FIFO also gets asserted as both senddata and dataready signals are asserted, and thus the data token gets written to the FIFO.

For the output part, this model always monitors the 'empty' output signal from the FIFO. The 'empty' signal is asserted only when there are no data tokens present in the FIFO and it is de-asserted otherwise. In the case of the 'empty' output signals of FIFOs attached to 'a' and 'b' both being de-asserted, it asserts the 'dataready_out' signal going out of the model. The receiver connected to this model follows the same handshaking mechanism to assert or de-assert the 'senddata_out' signal. Only when the 'senddata_out' signal is asserted, the

model asserts the 'rd_en' input signal (read enable) to the FIFO and reads the data token from the FIFO output and sends the produced output data token to the receiver. The gist of the explained handshaking protocol can be expressed by following statements (with the conventional meaning of the logical operators) for this model :

$$senddata\_a = (!full\_a)\ \&\ (dataready\_a) \tag{3.1}$$

$$wr\_en\_a = (senddata\_a)\ \&\ (dataready\_a) \tag{3.2}$$

$$dataready\_out = (!empty\_a)\ \&\ (!empty\_b) \tag{3.3}$$

$$rd\_en\_a = (senddata\_out)\ \&\ (!empty\_a) \tag{3.4}$$

An example of the communication of CSP handshaking signals from a model's output port(s) to another model's input port(s) and vice-versa is shown in Figure 3.5. This kind of arrangement also makes the system *latency tolerant*, as the output tokens are always produced in one clock cycle once the 'senddata' signal is asserted and tokens are present in the input FIFOs. For the sake of simplicity, the 'dataready' signals at the input ports and the 'senddata' signals at the output ports of the system are tied to '1' by the compiler, that would make the system to start the flow of the data tokens.

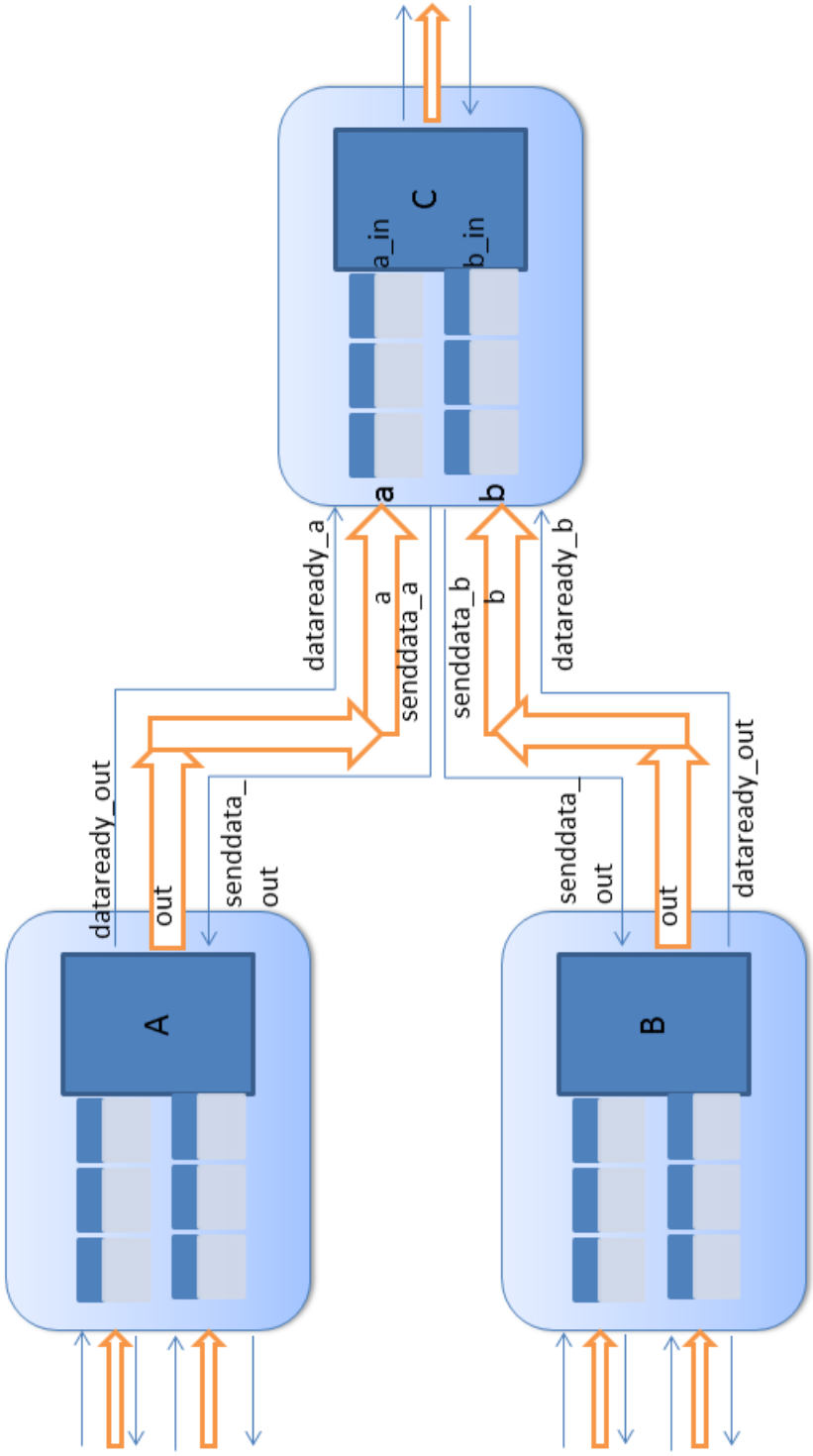

Figure 3.5 Communication of CSP handshaking signals

A library of Verilog models is created that incorporates the CSP handshaking mechanism described. The statements shown above are implemented as combinational logic by continuous assignment statements in each of the Verilog models. All the models have been verified to be functionally correct by simulation through testbenches that cover all the possible cases. Figure 3.5 summarizes the library of the Verilog models developed with their primary ports and operation. Note that each of the models have additional CSP ports and logic for the handshaking mechanism described earlier.

S.No.	Model Name	Primary Inputs	Primary output	Operation
1.	2-input Adder	plus1, plus2	out	out = plus1 +plus2
2.	3-input Adder	Plus1, plus2, plus3	out	out = plus1 + plus2 + plus3
3.	Subtractor	plus, minus	out	out = plus - minus
4.	Accumulator	in, reset	out	If(reset), out = 0 Else, out = out + in
5.	Multiplier	in1, in2	out	out = in1 * in 2
6.	Divider	in1, in2	out	out = in1/in2
7.	Equals	in1, in2	out	If(in1==in2), out = 1 Else, out = 0
8.	Logic Function	In1, in2	out	Can be configured as any Boolean gate by setting up a parameter
9.	BooleanSelect	in, control	out	If(control), out = in Else, out = out
10.	Remainder	in1, in2	out	out = in1%in2
11.	Counter	in	out	If(in), out = out + 1

Figure 3.6 CSP Verilog model library

3.4 HDL Generation

In this section, the methodology of generating a top-level Verilog file through the compiler, which is written in Java, is presented. The high-level models drawn in Ptolemy can be saved in form of an eXtensible Markup Language (XML) file format. The saved file contains collection of markup information of all the components used in the design like actors, wires and ports.

A snippet of an example XML file is illustrated with the information implied by various markups in Figure 3.6:

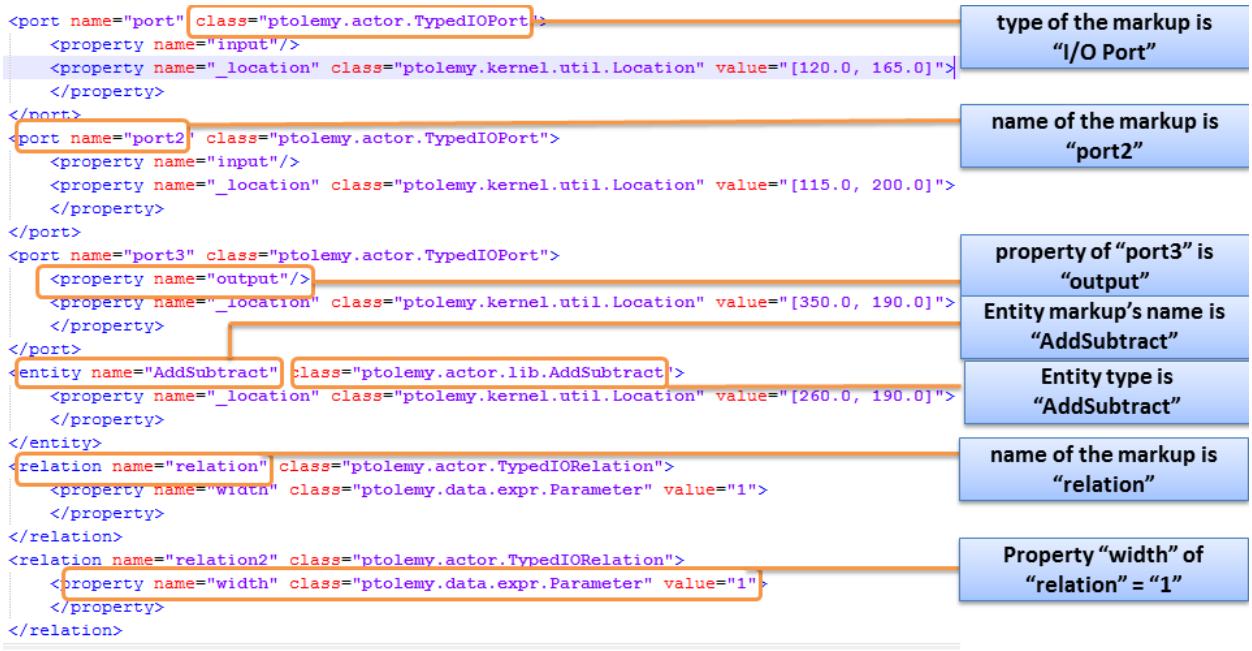

Figure 3.7 Format of an XML file

To extract useful data from the saved XML file, the *Javax XML parser* is used from the package "javax.xml.parsers" [21]. The *DocumentBuilderFactory* class present in the package defines an Application Programming Interface (API) through which the compiler can obtain a parser that produces Document Object Model (DOM) object trees [22] from the XML file.

Then, the *DocumentBuilder* class present in the same package defines an API through which the compiler obtains a DOM document instance from the XML file. This DOM instance contains all the information from the XML file in form of a tree that contains information of the markups (entity/port/relation) as nodes containing attributes.

Once the DOM object tree is created in such a fashion, all the data stored in the XML file is now represented by a data structure that can be accessed as and when any particular kind of information is required. The compiler then creates a new .v or Verilog file with the same name as the input XML file. Synthesizeable code is then written to this new Verilog file by extracting useful data out of the DOM object tree. The scheme of extracting a particular information and writing it out in an output Verilog file is shown in Figure 3.7:

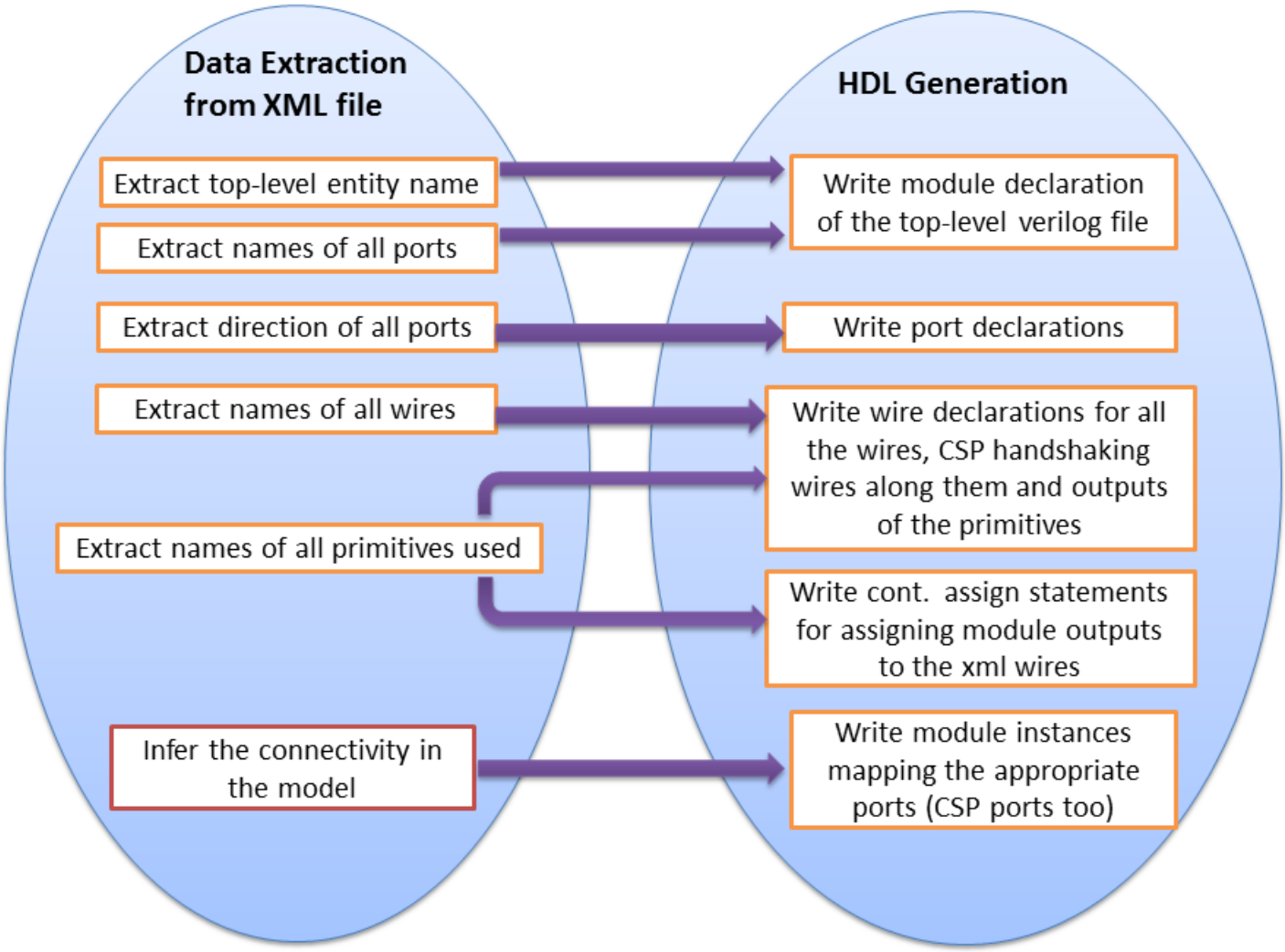

Figure 3.8 FPGA Design Flow using CSP Model Compiler

The main challenge while generating top-level netlist in this way is to make sure that the connectivity in the design is inferred correctly, because the XML file is not a netlist by itself, it is just a collection of all the components used in the design as markups. The only source for inferring the netlist here is the "link" markup in the design that gives the name of a wire connected to every port in the model. Figure 3.8 shows an XML file snippet depicting the

link markups.

```
<link port="port" relation="relation2"/>
<link port="port2" relation="relation"/>
<link port="port3" relation="relation3"/>
<link port="AddSubtract.plus" relation="relation"/>
<link port="AddSubtract.plus" relation="relation2"/>
<link port="AddSubtract.output" relation="relation3"/>
```

Figure 3.9 "Link" markups in the XML model

All the wires in Ptolemy tool are referred to as *relations*. The compiler has a "Relation" Class that defines any relation used in the model in form of a "source" field, a "width" field and an array "sinks" of length equal to the width. The compiler determines the source and the sinks of any relation searching through the "link" nodes in the DOM object tree. As it finds out the source and the sinks of every relation, it also updates a hashtable that maps every relation name to a corresponding object of "Relation" Class that will now have information of source and sinks of the wire. And, this hashtable is then looked up while writing the continuous assignment statements and module instances in the Verilog file for correct mapping of the ports to the wires.

In the next chapter, some of the experiments performed using this compiler are presented and it is compared with other high-level synthesis tools.

Chapter 4

Experiments and Results

The primary objective of this work is to develop a high-level synthesis tool that generates HDLs that are reliable and are less prone to problems like timing violations during synthesis and bottlenecks while in actual operation. A number of designs were drawn in the Ptolemy tool as test circuits for the compiler, and the generated HDLs were run through Xilinx XST synthesis tool [23] for synthesis and verified for functionality using Xilinx ISim simulation tool [24]. Some of the key experiments are presented in the first section of this chapter.

In the next section, the compiler is compared with the other popular high-level synthesis tools (refer to Section 2.5). All these different tools are based on various design methodologies, so they greatly differ in their design entry method, the model of computation they use etc. Thus, a qualitative comparison of these tools is presented with respect to their various attributes.

4.1 Experiment 1. Tree Adder Design

In this experiment, a 3-stage tree adder was drawn in the Ptolemy tool as shown in Figure 4.1. This test circuit was created by just using the "AddSubtract" actors from the Ptolemy actor library.

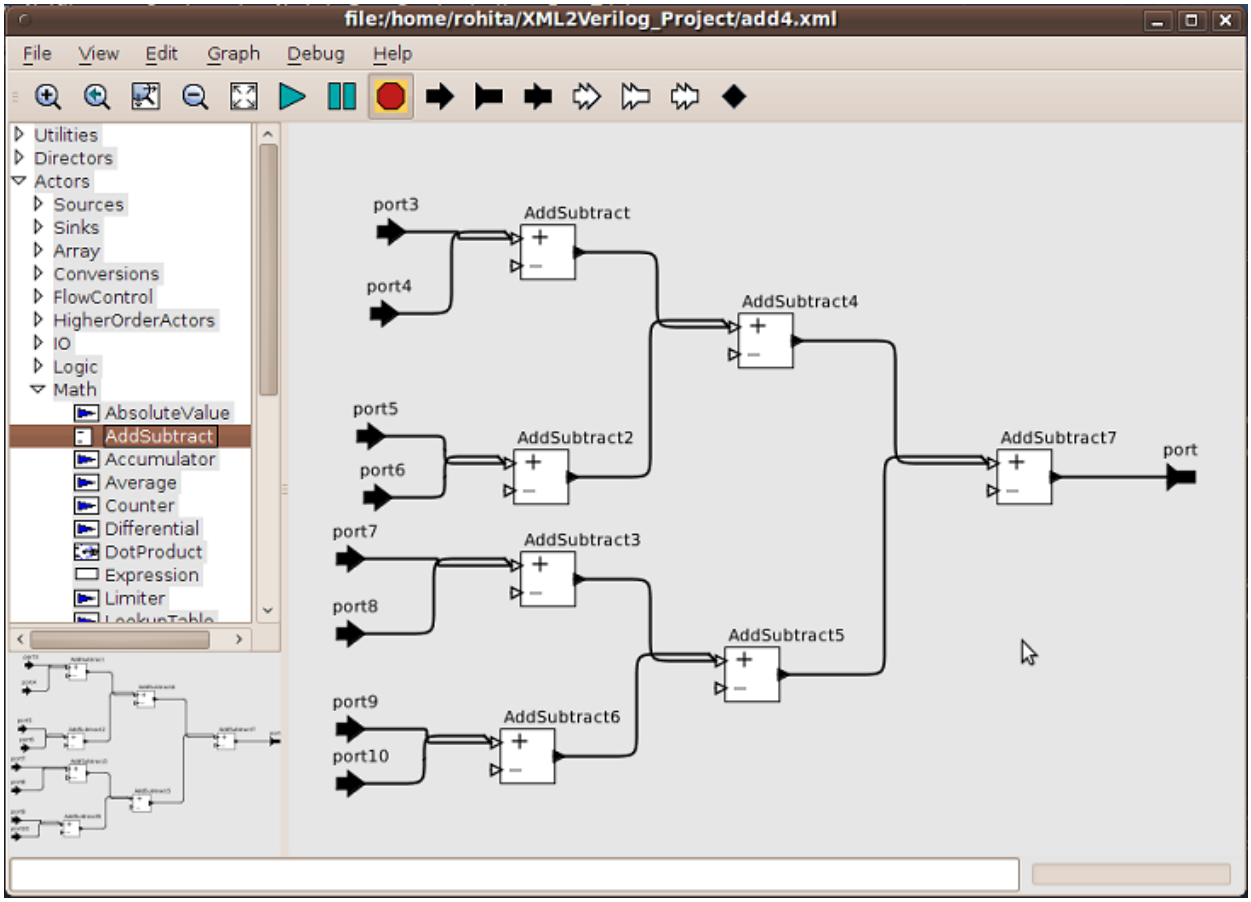

Figure 4.1 3-stage Tree Adder drawn in Ptolemy(screenshot captured from the Ptolemy GUI)

The primary purpose of this experiment was to test the resolution of multi-port input ports by the compiler. Most of the actors present in the Ptolemy actor library have multi-port input ports, so in order to get the "AddSubtract" actor to behave as an adder, multiple wires are connected to the same port, which is shown in the figure as "+", leaving the "-"

port unconnected. The compiler resolves this kind of multi-port input ports internally and generates HDL design that would actually have distinct ports for each of the wires connected to the multi-port input port. The input XML file of the design (generated from Ptolemy) and the synthesized HDL generated are shown in Listing A.1 and Listing A.2 of Appendix A respectively.

This model was simulated for functionality in the Ptolemy tool itself by adding a CSP director, and replacing the input and output ports by the source and the sink actors available in the Ptolemy library for simulation. After simulation, the model saved as an XML file was run through the compiler for HDL generation. The compiler displayed on the shell; the name of the top-level file generated and the CSP Verilog models used (shown in Figure 4.2). All these files were then fed to the XST synthesis tool for synthesis. The design completed the XST synthesis successfully without any errors. It was also verified to be functionally correct and exhibiting the CSP handshaking mechanism as desired, by simulation in the Xilinx ISim simulation tool in the Xilinx ISE Design Suite. Figure 4.2 shows a screenshot of the RTL schematic of the synthesized design.

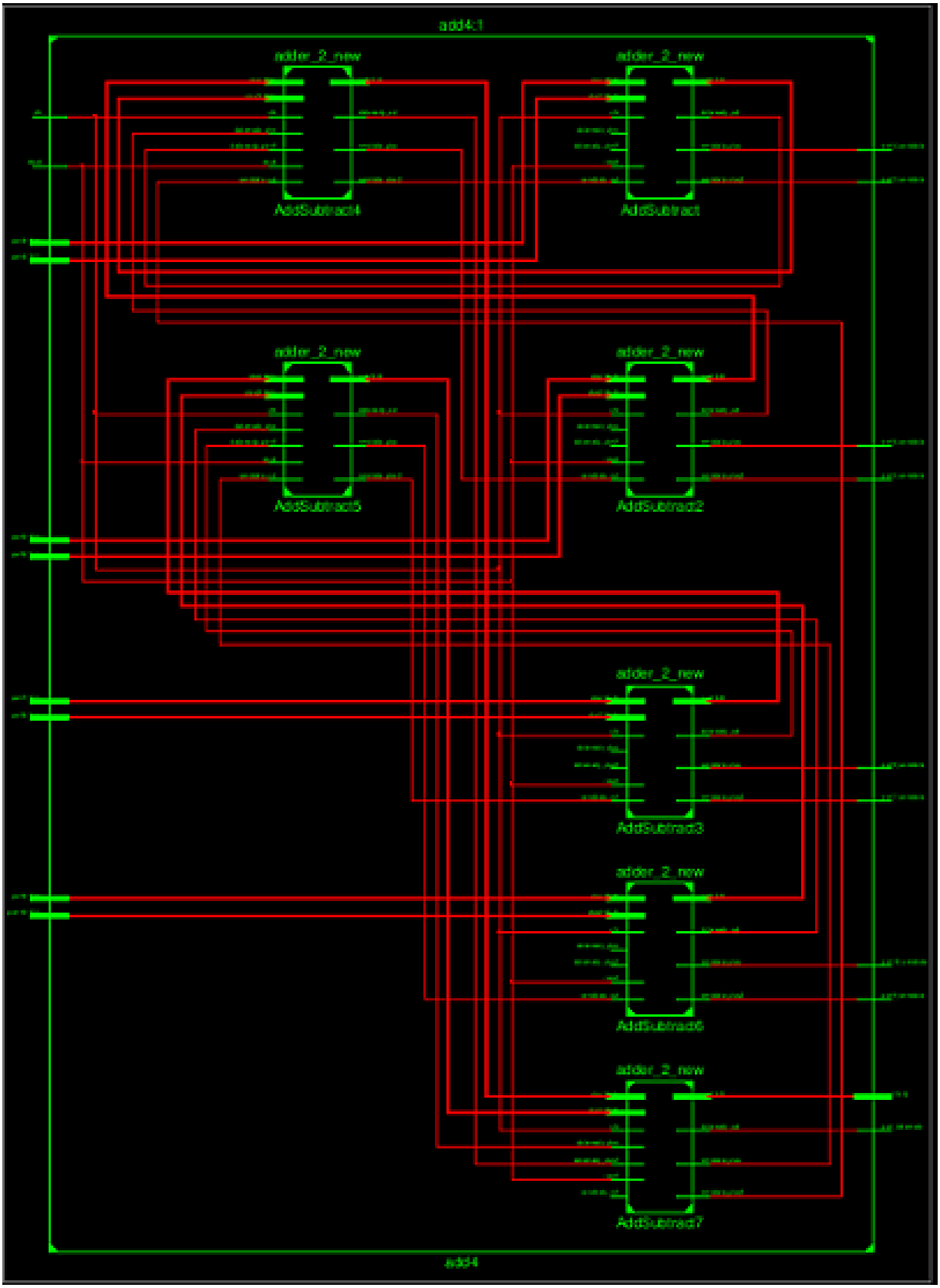

Figure 4.2 RTL Schematic of the 3-stage Tree Adder(screenshot captured from the Xilinx ISE GUI)

4.2 Experiment 2. Moving Average FIR Filter Design

Figure 4.3 shows a basic moving average FIR filter that was drawn in Ptolemy using various actors. The main purpose of this experiment was to test the compiler with a design built from many different components.

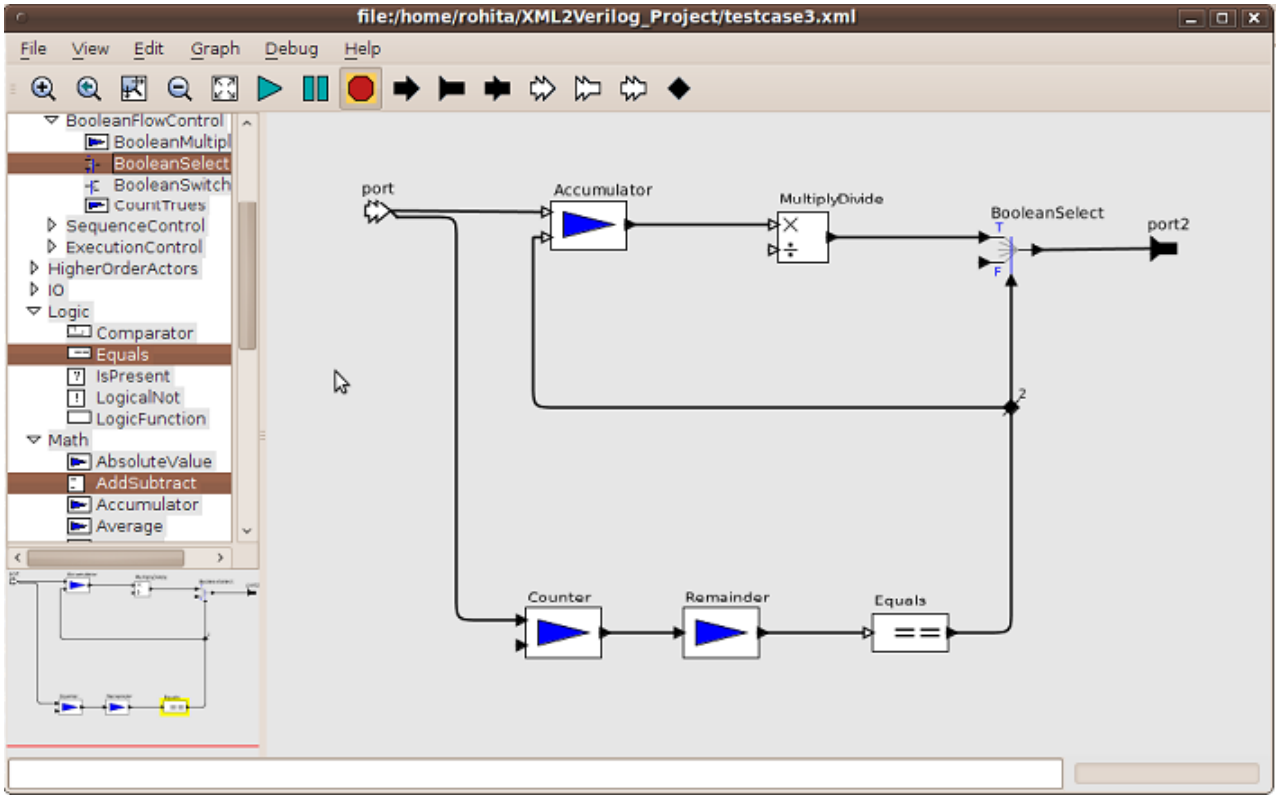

Figure 4.3 Moving Average FIR filter drawn in Ptolemy(screenshot captured from the Ptolemy GUI)

This filter produces one output sample for every four input samples, and the output sample is the average of the last four input samples. The actor shown in the figure as "Multiply-Divide" was configured to be a divide-by-4 divider, the divisor in the "Remainder" actor was set to four, and the wire from the "Equals" actor is connected to the 'reset' port of the "Accumulator" and the 'control' port of the "BooleanSelect" actor. "Equals" actor was

configured such that it would produce a '1' when the input is equal to 4. Thus for every four samples, the output from "Equals" actor would be '1' that would make the "BooleanSelect" actor to pass the output from the "MultiplyDivide" actor and also reset the "Accumulator". The result is a circuit that produces one output average sample for every four input samples, which can be also classified as a moving average FIR filter, whose output is downsampled by four.

The HDLs were created for this circuit from the input XML file using the compiler. The generated HDLs completed synthesis in the XST tool successfully and were verified to be functionally correct and exhibiting the CSP handshaking mechanism as desired. Figure 4.4 shows a screenshot of the RTL schematic of the synthesized design. The input XML file of the design (generated from Ptolemy) and the synthesized HDLs generated are shown in Listing B.1 and Listing B.2 of Appendix B respectively.

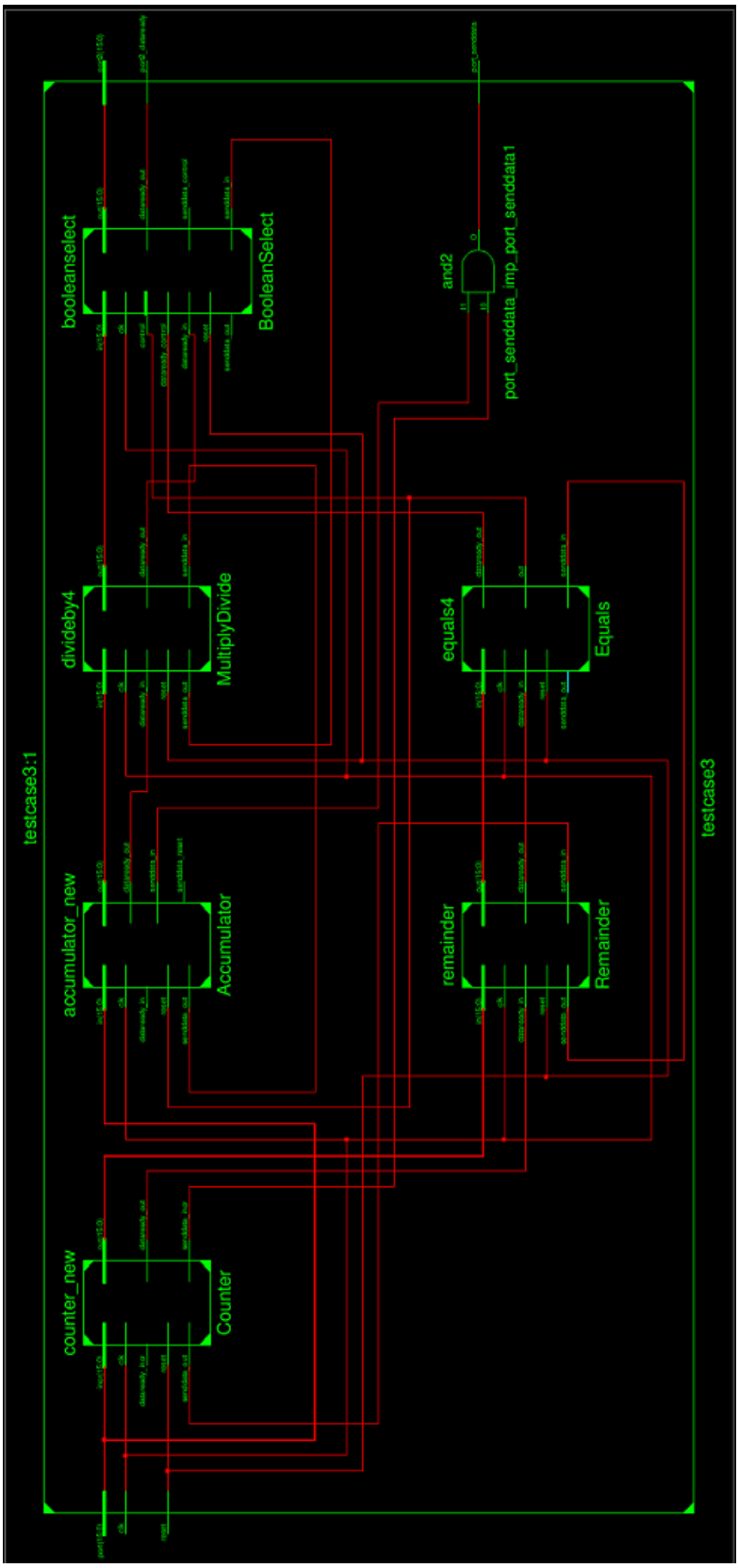

Figure 4.4 RTL Schematic of the FIR filter(screenshot captured from the Xilinx ISE GUI)

4.3 Tools Comparison

Figure 4.5 shows a qualitative comparison of the CSP model compiler with other popular high-level synthesis tools (refer to Section 2.5). The model of computation these tools use is a key factor in determining how the design is actually implemented and how reliable it is while in actual operation. Most of the other tools that have graphical block diagram design entry do not employ CSP as the MoC, and hence the designs generated by them are more susceptible to timing violations during implementation and bottleneck problems while in actual operation. Element CXI does employ CSP MoC, but it is limited to their own chips and the size of these chips is also not enough for building complex designs.

	Simulink HDL Coder	LabVIEW FPGA	Impulse C	Element CXI	CSP Model Compiler
Design Entry	Graphical Block Diagram	Graphical Block Diagram	C model	Graphical Block Diagram	Graphical Block Diagram
Back Pressure Mechanism	No	No	Yes (CSP)	Yes (CSP)	Yes (CSP)
Library Support	Good	Good	Good	Limited	Work in Progress
Supports Hierarchy	Yes	Yes	Yes	Yes	Work in Progress
Platform Independent	Yes	No	Yes	No	Yes
Open-Source	No	No	No	No	Yes

Figure 4.5 Comparison with other popular high-level synthesis tools

The mode of design entry is important from the design time point of view. Although algorithms described in C or SystemC take lesser time than describing hardware behavior in a hardware description language, a graphical block diagram mode of design entry raises the level of abstraction even more. Consequently, system engineers can be added to the pool of design engineers, making it easier to meet stringent design time goals. Impulse C has plenty of good things associated with its design methodology, yet the tool could have been even better if it had a graphical block diagram mode of design entry.

Being platform-independent, the tool enables the designer to try out different optimization strategies on different platforms and explore tradeoffs associated with them. The Verilog RTL generated by the CSP model compiler is very generic in nature and can be targeted to a variety of platforms including all kinds of FPGAs.

A big advantage of this compiler is that it would be open-source, very few of the high-level synthesis tools are open-source and among the existing ones, focus has been directed more towards parallelizing the tool [25] or towards DSP applications [26], rather than creating reliable designs that behave according to a certain MoC.

A good number of basic primitives have already been built in the CSP model library. While building the library for current work, the focus has been more on building fine-grained primitives to test the compiler for rather simple circuits as a proof-of-concept. A more exhaustive library containing coarse-grained primitives is currently being developed that would make it possible to design complex systems through this tool. The support for hierarchical designs is also under development (refer Section 5.2).

Chapter 5

Conclusions and Future Work

This thesis addressed the issue of the lack of wide acceptance of high-level synthesis tools in the traditional design methodology. High-level synthesis tools aim to increase the predictability and productivity of the design by raising the level of abstraction and reduce the number of bugs introduced due to human-error by creating more automation in the design flow. Because of the higher level of abstraction, these tools also facilitate the designer to try out different optimization strategies easily and explore the various trade-offs very early in the design stage. However, these tools have still not gained much popularity in the mainstream design flow, because of the reasons like bad readability and less reliability of the generated RTL code [2].

The research work presented in this thesis tries to address these issues. A compiler framework has been presented that can translate graphical models of a design into Verilog RTL models. The generated Verilog models exhibit their desired behaviors following CSP model of computation. CSP model of computation introduces a good handshaking mechanism between the different components in the design, which also acts as back pressure mechanism in preventing overflow. Such designs are less prone to have timing violations during im-

plementation and bottlenecks while in actual operation. Moreover, the design descriptions generated in Verilog are very generic and can be targeted to a variety of platforms including all kinds of FPGAs.

The compiler was tested for various circuits built from fine-grained primitives. A more exhaustive library consisting of coarse-grained primitives like arithmetic and logic unit (ALU), signal processing filters etc. would make it easier to design complex systems through this tool. Support for hierarchical designs would also help in designing complex systems, which is currently a work in progress and the directions for future work are presented in Section 5.2.

5.1 Contributions

Listed below are the contributions of this thesis:

1. A new design methodology for FPGA is presented that allows the user to draw graphical models of a design in the Ptolemy modeling environment. The compiler translates these models into a structural RTL netlist in Verilog that can be synthesized using the standard synthesis tools.

2. A library containing CSP Verilog models of the primitives that operate under CSP model of computation is developed.

3. A compiler framework has been developed in Java that extracts the information out of the XML models saved from the Ptolemy tool. The compiler then interprets this extracted information to generate Verilog RTL models of the design that exhibit the desired behavior following the CSP model of computation.

5.2 Future Work

Results presented in this thesis show that the compiler generates Verilog models of designs that operate as per CSP model of computation. The compiler was tested for relatively simple circuits to demonstrate a proof-of-concept. The tool framework is capable of incorporating following extensions to make it possible to design complex systems through this compiler:

1. More exhaustive library: The current CSP Verilog model library contains a good number of fine-grained primitives. Addition of coarse-grained primitives like ALUs, signal processing filters etc. to the library would facilitate building of more complex systems through this tool.

2. Support for hierarchical designs: Currently, the compiler maps the actors used in the Ptolemy models to the CSP Verilog models. It does so by mapping every port of an actor one-by-one to the ports in an existing CSP Verilog model in the library. The ports in a Verilog model are known beforehand in this case. In case of hierarchical designs, the ports of a Verilog file generated in a lower hierarchy are not known. In order to address this issue, the compiler would need to map the ports in a following manner:

   (a) Parse the Verilog file generated at a lower hierarchy level.

   (b) Extract the ports out of the parsed file and map them to the appropriate ports at the upper hierarchy level.

   (c) Repeat (a) and (b) recursively.

Appendix A

Sample Tree Adder

A.1 Input XML file (saved from Ptolemy)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="add4" class="ptolemy.actor.TypedCompositeActor">
    <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="7.0.1">
    </property>
    <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute" value="{bounds
        ={369, 192, 913, 655}, maximized=false}">
    </property>
    <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[701, 539]">
    </property>
    <property name="_vergilZoomFactor" class="ptolemy.data.expr.ExpertParameter" value="1.0">
    </property>
    <property name="_vergilCenter" class="ptolemy.data.expr.ExpertParameter" value="{350.5, 269.5}">
    </property>
    <port name="port" class="ptolemy.actor.TypedIOPort">
        <property name="output"/>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[655.0, 260.0]">
        </property>
    </port>
    <port name="port3" class="ptolemy.actor.TypedIOPort">
        <property name="input"/>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[75.0, 80.0]">
        </property>
    </port>
    <port name="port4" class="ptolemy.actor.TypedIOPort">
        <property name="input"/>
```

```
    <property name="_location" class="ptolemy.kernel.util.Location" value="[70.0, 140.0]">
    </property>
</port>
<port name="port5" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[60.0, 230.0]">
    </property>
</port>
<port name="port6" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[65.0, 275.0]">
    </property>
</port>
<port name="port7" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[45.0, 320.0]">
    </property>
</port>
<port name="port8" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[45.0, 375.0]">
    </property>
</port>
<port name="port9" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[45.0, 445.0]">
    </property>
</port>
<port name="port10" class="ptolemy.actor.TypedIOPort">
    <property name="input"/>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[45.0, 485.0]">
    </property>
</port>
<entity name="AddSubtract" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[190.0, 95.0]">
    </property>
</entity>
<entity name="AddSubtract2" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[185.0, 260.0]">
    </property>
</entity>
<entity name="AddSubtract3" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[190.0, 335.0]">
    </property>
</entity>
<entity name="AddSubtract4" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[350.0, 160.0]">
    </property>
</entity>
<entity name="AddSubtract5" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[340.0, 405.0]">
    </property>
```

```
</entity>
<entity name="AddSubtract6" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[175.0, 465.0]">
    </property>
</entity>
<entity name="AddSubtract7" class="ptolemy.actor.lib.AddSubtract">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[540.0, 260.0]">
    </property>
</entity>
<relation name="relation6" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation7" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation8" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation9" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation2" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation3" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation4" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation5" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation10" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation11" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
```

```
    </relation>
    <relation name="relation12" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation13" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation14" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation15" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <link port="port" relation="relation15"/>
    <link port="port3" relation="relation7"/>
    <link port="port4" relation="relation6"/>
    <link port="port5" relation="relation9"/>
    <link port="port6" relation="relation8"/>
    <link port="port7" relation="relation12"/>
    <link port="port8" relation="relation11"/>
    <link port="port9" relation="relation14"/>
    <link port="port10" relation="relation13"/>
    <link port="AddSubtract.plus" relation="relation6"/>
    <link port="AddSubtract.plus" relation="relation7"/>
    <link port="AddSubtract.output" relation="relation10"/>
    <link port="AddSubtract2.plus" relation="relation8"/>
    <link port="AddSubtract2.plus" relation="relation9"/>
    <link port="AddSubtract2.output" relation="relation5"/>
    <link port="AddSubtract3.plus" relation="relation11"/>
    <link port="AddSubtract3.plus" relation="relation12"/>
    <link port="AddSubtract3.output" relation="relation4"/>
    <link port="AddSubtract4.plus" relation="relation5"/>
    <link port="AddSubtract4.plus" relation="relation10"/>
    <link port="AddSubtract4.output" relation="relation"/>
    <link port="AddSubtract5.plus" relation="relation3"/>
    <link port="AddSubtract5.plus" relation="relation4"/>
    <link port="AddSubtract5.output" relation="relation2"/>
    <link port="AddSubtract6.plus" relation="relation13"/>
    <link port="AddSubtract6.plus" relation="relation14"/>
    <link port="AddSubtract6.output" relation="relation3"/>
    <link port="AddSubtract7.plus" relation="relation2"/>
    <link port="AddSubtract7.plus" relation="relation"/>
    <link port="AddSubtract7.output" relation="relation15"/>
</entity>
```

treeadder.xml

A.2 Generated Verilog files

```
////////////////////////////////////////////////////////////////////////
//Generated by High-Level CSP Model Compiler
//
//Top-level netlist for input file add4.xml containing instances of
// the models instantiated from the CSP model library.
//
////////////////////////////////////////////////////////////////////////


//declaration of top-level module
module add4 (port, port_dataready, port3_senddata, port4_senddata, port5_senddata, port6_senddata,
     port7_senddata, port8_senddata, port9_senddata, port10_senddata, port3, port4, port5, port6, port7,
     port8, port9, port10, clk, reset);


//declaration of input-output ports
output [15:0] port;
output port_dataready;
output port3_senddata;
output port4_senddata;
output port5_senddata;
output port6_senddata;
output port7_senddata;
output port8_senddata;
output port9_senddata;
output port10_senddata;
input [15:0] port3;
input [15:0] port4;
input [15:0] port5;
input [15:0] port6;
input [15:0] port7;
input [15:0] port8;
input [15:0] port9;
input [15:0] port10;
input clk;
input reset;


//declaration of intermediate wires
wire [15:0] relation6;
wire [15:0] relation7;
wire [15:0] relation8;
wire [15:0] relation9;
wire [15:0] relation2;
wire [15:0] relation;
wire [15:0] relation3;
wire [15:0] relation4;
wire [15:0] relation5;
wire [15:0] relation10;
```

```
wire [15:0] relation11;
wire [15:0] relation12;
wire [15:0] relation13;
wire [15:0] relation14;
wire [15:0] relation15;
wire [15:0] AddSubtract4_output;
wire AddSubtract4_output_dataready;
wire [15:0] AddSubtract2_output;
wire AddSubtract2_output_dataready;
wire [15:0] AddSubtract3_output;
wire AddSubtract3_output_dataready;
wire [15:0] AddSubtract6_output;
wire AddSubtract6_output_dataready;
wire [15:0] AddSubtract5_output;
wire AddSubtract5_output_dataready;
wire [15:0] AddSubtract7_output;
wire AddSubtract7_output_dataready;
wire [15:0] AddSubtract_output;
wire AddSubtract_output_dataready;
wire relation3_senddata;
wire AddSubtract5_plus_senddata;
wire relation15_dataready;
wire relation14_senddata;
wire AddSubtract6_plus2_senddata;
wire relation7_senddata;
wire AddSubtract_plus2_senddata;
wire relation12_senddata;
wire AddSubtract3_plus2_senddata;
wire relation5_senddata;
wire AddSubtract4_plus_senddata;
wire relation4_dataready;
wire relation10_senddata;
wire AddSubtract4_plus2_senddata;
wire relation8_senddata;
wire AddSubtract2_plus_senddata;
wire relation2_dataready;
wire relation2_senddata;
wire AddSubtract7_plus_senddata;
wire relation5_dataready;
wire relation_dataready;
wire relation13_senddata;
wire AddSubtract6_plus_senddata;
wire relation10_dataready;
wire relation6_senddata;
wire AddSubtract_plus_senddata;
wire relation3_dataready;
wire relation_senddata;
wire AddSubtract7_plus2_senddata;
wire relation11_senddata;
wire AddSubtract3_plus_senddata;
wire relation4_senddata;
wire AddSubtract5_plus2_senddata;
```

```
wire relation9_senddata;
wire AddSubtract2_plus2_senddata;


//continuous assignment statements
assign relation = AddSubtract4_output;
assign relation9 = port5;
assign relation8 = port6;
assign relation7 = port3;
assign relation6 = port4;
assign relation5 = AddSubtract2_output;
assign relation4 = AddSubtract3_output;
assign relation3 = AddSubtract6_output;
assign relation2 = AddSubtract5_output;
assign relation15 = AddSubtract7_output;
assign port = relation15;
assign relation14 = port9;
assign relation13 = port10;
assign relation12 = port7;
assign relation11 = port8;
assign relation10 = AddSubtract_output;
assign relation3_senddata = AddSubtract5_plus_senddata;
assign relation15_dataready = AddSubtract7_output_dataready;
assign port_dataready = relation15_dataready;
assign relation14_senddata = AddSubtract6_plus2_senddata;
assign port9_senddata = relation14_senddata;
assign relation7_senddata = AddSubtract_plus2_senddata;
assign port3_senddata = relation7_senddata;
assign relation12_senddata = AddSubtract3_plus2_senddata;
assign port7_senddata = relation12_senddata;
assign relation5_senddata = AddSubtract4_plus_senddata;
assign relation4_dataready = AddSubtract3_output_dataready;
assign relation10_senddata = AddSubtract4_plus2_senddata;
assign relation8_senddata = AddSubtract2_plus_senddata;
assign port6_senddata = relation8_senddata;
assign relation2_dataready = AddSubtract5_output_dataready;
assign relation2_senddata = AddSubtract7_plus_senddata;
assign relation5_dataready = AddSubtract2_output_dataready;
assign relation_dataready = AddSubtract4_output_dataready;
assign relation13_senddata = AddSubtract6_plus_senddata;
assign port10_senddata = relation13_senddata;
assign relation10_dataready = AddSubtract_output_dataready;
assign relation6_senddata = AddSubtract_plus_senddata;
assign port4_senddata = relation6_senddata;
assign relation3_dataready = AddSubtract6_output_dataready;
assign relation_senddata = AddSubtract7_plus2_senddata;
assign relation11_senddata = AddSubtract3_plus_senddata;
assign port8_senddata = relation11_senddata;
assign relation4_senddata = AddSubtract5_plus2_senddata;
assign relation9_senddata = AddSubtract2_plus2_senddata;
assign port5_senddata = relation9_senddata;
```

```
//instantiation of models from CSP model library
adder_2_new AddSubtract(.out(AddSubtract_output), .dataready_out(AddSubtract_output_dataready), .
    senddata_plus(AddSubtract_plus_senddata), .senddata_plus2(AddSubtract_plus2_senddata), .plus(relation7
    ), .plus2(relation6), .senddata_out(relation10_senddata), .dataready_plus(1'b1), .dataready_plus2(1'b1
    ), .clk(clk), .reset(reset));
adder_2_new AddSubtract2(.out(AddSubtract2_output), .dataready_out(AddSubtract2_output_dataready), .
    senddata_plus(AddSubtract2_plus_senddata), .senddata_plus2(AddSubtract2_plus2_senddata), .plus(
    relation9), .plus2(relation8), .senddata_out(relation5_senddata), .dataready_plus(1'b1), .
    dataready_plus2(1'b1), .clk(clk), .reset(reset));
adder_2_new AddSubtract3(.out(AddSubtract3_output), .dataready_out(AddSubtract3_output_dataready), .
    senddata_plus(AddSubtract3_plus_senddata), .senddata_plus2(AddSubtract3_plus2_senddata), .plus(
    relation12), .plus2(relation11), .senddata_out(relation4_senddata), .dataready_plus(1'b1), .
    dataready_plus2(1'b1), .clk(clk), .reset(reset));
adder_2_new AddSubtract4(.out(AddSubtract4_output), .dataready_out(AddSubtract4_output_dataready), .
    senddata_plus(AddSubtract4_plus_senddata), .senddata_plus2(AddSubtract4_plus2_senddata), .plus(
    relation5), .plus2(relation10), .senddata_out(relation_senddata), .dataready_plus(relation5_dataready)
    , .dataready_plus2(relation10_dataready), .clk(clk), .reset(reset));
adder_2_new AddSubtract5(.out(AddSubtract5_output), .dataready_out(AddSubtract5_output_dataready), .
    senddata_plus(AddSubtract5_plus_senddata), .senddata_plus2(AddSubtract5_plus2_senddata), .plus(
    relation4), .plus2(relation3), .senddata_out(relation2_senddata), .dataready_plus(relation3_dataready)
    , .dataready_plus2(relation4_dataready), .clk(clk), .reset(reset));
adder_2_new AddSubtract6(.out(AddSubtract6_output), .dataready_out(AddSubtract6_output_dataready), .
    senddata_plus(AddSubtract6_plus_senddata), .senddata_plus2(AddSubtract6_plus2_senddata), .plus(
    relation14), .plus2(relation13), .senddata_out(relation3_senddata), .dataready_plus(1'b1), .
    dataready_plus2(1'b1), .clk(clk), .reset(reset));
adder_2_new AddSubtract7(.out(AddSubtract7_output), .dataready_out(AddSubtract7_output_dataready), .
    senddata_plus(AddSubtract7_plus_senddata), .senddata_plus2(AddSubtract7_plus2_senddata), .plus(
    relation), .plus2(relation2), .senddata_out(1'b1), .dataready_plus(relation2_dataready), .
    dataready_plus2(relation_dataready), .clk(clk), .reset(reset));

endmodule
```

treeadder.v

```
////////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Adder CSP Model
//
////////////////////////////////////////////////////////////////////////


module adder_2 (out, dataready_out, senddata_plus, senddata_plus2, plus, plus2, senddata_out,
    dataready_plus, dataready_plus2, clk, reset);

  output reg [15:0] out;
  output dataready_out;
  output senddata_plus;
  output senddata_plus2;
  input senddata_out;
  input [15:0] plus;
  input dataready_plus;
  input [15:0] plus2;
  input dataready_plus2;
  input clk;
  input reset;

  wire rd_en_plus, rd_en_plus2;
  wire [15:0] plusin;
  wire [15:0] plus2in;
  wire empty_plus, empty_plus2;
  wire full_plus, full_plus2;
  wire wr_en_plus, wr_en_plus2;


  //CSP Handshaking protocol
  assign senddata_plus = ((~full_plus) & dataready_plus) | empty_plus;
  assign senddata_plus2 = ((~full_plus2) & dataready_plus2) | empty_plus2;
  assign dataready_out = (~empty_plus) & (~empty_plus2);
  assign rd_en_plus = senddata_out & (~empty_plus);
  assign rd_en_plus2 = senddata_out & (~empty_plus2);
  assign wr_en_plus = senddata_plus & dataready_plus;
  assign wr_en_plus2 = senddata_plus2 & dataready_plus2;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 16'd0;
    else
      if(rd_en_plus & rd_en_plus2)
         out <= plusin + plus2in;
  end


fifo_generator_v5_3 fifo_plus(.clk(clk), .rst(reset), .din(plus), .wr_en(wr_en_plus), .rd_en(rd_en_plus),
    .dout(plusin), .full(full_plus), .empty(empty_plus));
```

```
fifo_generator_v5_3 fifo_plus2(.clk(clk), .rst(reset), .din(plus2), .wr_en(wr_en_plus2), .rd_en(
    rd_en_plus2), .dout(plus2in), .full(full_plus2), .empty(empty_plus2));

endmodule
```

adder_2.v

Appendix B

Sample Moving Average FIR Filter

B.1 Input XML file (saved from Ptolemy)

```
<?xml version="1.0" standalone="no"?>
<!DOCTYPE entity PUBLIC "-//UC Berkeley//DTD MoML 1//EN"
    "http://ptolemy.eecs.berkeley.edu/xml/dtd/MoML_1.dtd">
<entity name="testcase3" class="ptolemy.actor.TypedCompositeActor">
    <property name="_createdBy" class="ptolemy.kernel.attributes.VersionAttribute" value="7.0.1">
    </property>
    <property name="_windowProperties" class="ptolemy.actor.gui.WindowPropertiesAttribute" value="{bounds
        ={720, 44, 1008, 630}, maximized=false}">
    </property>
    <property name="_vergilSize" class="ptolemy.actor.gui.SizeAttribute" value="[796, 514]">
    </property>
    <property name="_vergilZoomFactor" class="ptolemy.data.expr.ExpertParameter" value="1.0">
    </property>
    <property name="_vergilCenter" class="ptolemy.data.expr.ExpertParameter" value="{398.0, 257.0}">
    </property>
    <port name="port" class="ptolemy.actor.TypedIOPort">
        <property name="input"/>
        <property name="multiport"/>
        <property name="_location" class="ptolemy.kernel.util.Location" value="[88.6666145324707,
            79.99953842163086]">
        </property>
        <property name="width" class="ptolemy.data.expr.Parameter" value="2">
        </property>
    </port>
    <port name="port2" class="ptolemy.actor.TypedIOPort">
        <property name="output"/>
```

```
    <property name="_location" class="ptolemy.kernel.util.Location" value="[711.6015625,
        109.853515625]">
    </property>
</port>
<entity name="Accumulator" class="ptolemy.actor.lib.Accumulator">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[255.0, 90.0]">
    </property>
</entity>
<entity name="MultiplyDivide" class="ptolemy.actor.lib.MultiplyDivide">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[425.0, 100.0]">
    </property>
    <property name="divideby" class="ptolemy.data.expr.Parameter" value="4">
    </property>
</entity>
<entity name="Counter" class="ptolemy.actor.lib.Counter">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[235.0, 415.0]">
    </property>
</entity>
<entity name="Equals" class="ptolemy.actor.lib.logic.Equals">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[510.0, 415.0]">
    </property>
    <property name="equalto" class="ptolemy.data.expr.Parameter" value="4">
    </property>
</entity>
<entity name="BooleanSelect" class="ptolemy.actor.lib.BooleanSelect">
    <property name="_location" class="ptolemy.kernel.util.Location" value="[590.0, 110.0]">
    </property>
</entity>
<entity name="Remainder" class="ptolemy.actor.lib.Remainder">
    <property name="divisor" class="ptolemy.data.expr.Parameter" value="4.0">
    </property>
    <property name="_location" class="ptolemy.kernel.util.Location" value="[360.0, 415.0]">
    </property>
</entity>
<relation name="relation" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation5" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="2">
    </property>
    <vertex name="vertex1" value="[590.0, 235.0]">
    </vertex>
</relation>
<relation name="relation6" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
<relation name="relation3" class="ptolemy.actor.TypedIORelation">
    <property name="width" class="ptolemy.data.expr.Parameter" value="1">
    </property>
</relation>
```

```
    <relation name="relation7" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation2" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation9" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <relation name="relation10" class="ptolemy.actor.TypedIORelation">
        <property name="width" class="ptolemy.data.expr.Parameter" value="1">
        </property>
    </relation>
    <link port="port" relation="relation2"/>
    <link port="port" relation="relation9"/>
    <link port="port2" relation="relation10"/>
    <link port="Accumulator.input" relation="relation2"/>
    <link port="Accumulator.output" relation="relation"/>
    <link port="Accumulator.reset" relation="relation5"/>
    <link port="MultiplyDivide.multiply" relation="relation"/>
    <link port="MultiplyDivide.output" relation="relation6"/>
    <link port="Counter.increment" relation="relation9"/>
    <link port="Counter.output" relation="relation3"/>
    <link port="Equals.input" relation="relation7"/>
    <link port="Equals.output" relation="relation5"/>
    <link port="BooleanSelect.trueInput" relation="relation6"/>
    <link port="BooleanSelect.control" relation="relation5"/>
    <link port="BooleanSelect.output" relation="relation10"/>
    <link port="Remainder.input" relation="relation3"/>
    <link port="Remainder.output" relation="relation7"/>
</entity>
```

movavg.xml

B.2 Generated Verilog files

```
////////////////////////////////////////////////////////////////////////
//Generated by High-Level CSP Model Compiler
//
//Top-level netlist for input file movavg.xml containing instances of
// the models instantiated from the CSP model library.
//
////////////////////////////////////////////////////////////////////////


//declaration of top-level module
module testcase3 (port2, port2_dataready, port_senddata, port, clk, reset);


//declaration of input-output ports
output [15:0] port2;
output port2_dataready;
output port_senddata;
input [15:0] port;
input clk;
input reset;


//declaration of intermediate wires
wire [15:0] relation;
wire [15:0] relation5;
wire [15:0] relation6;
wire [15:0] relation3;
wire [15:0] relation7;
wire [15:0] relation2;
wire [15:0] relation9;
wire [15:0] relation10;
wire [15:0] Counter_output;
wire Counter_output_dataready;
wire [15:0] BooleanSelect_output;
wire BooleanSelect_output_dataready;
wire [15:0] Accumulator_output;
wire Accumulator_output_dataready;
wire [15:0] Remainder_output;
wire Remainder_output_dataready;
wire [15:0] MultiplyDivide_output;
wire MultiplyDivide_output_dataready;
wire [15:0] Equals_output;
wire Equals_output_dataready;
wire relation6_dataready;
wire relation_senddata;
wire MultiplyDivide_multiply_senddata;
wire relation10_dataready;
wire relation9_senddata;
wire Counter_increment_senddata;
```

```
wire relation7_senddata;
wire Equals_input_senddata;
wire relation5_dataready;
wire relation2_senddata;
wire Accumulator_input_senddata;
wire relation5_senddata;
wire Accumulator_reset_senddata;
wire BooleanSelect_control_senddata;
wire relation_dataready;
wire relation3_dataready;
wire relation7_dataready;
wire relation3_senddata;
wire Remainder_input_senddata;
wire relation6_senddata;
wire BooleanSelect_trueInput_senddata;


//continuous assignment statements
assign relation3 = Counter_output;
assign relation2 = port;
assign relation10 = BooleanSelect_output;
assign port2 = relation10;
assign relation9 = port;
assign relation = Accumulator_output;
assign relation7 = Remainder_output;
assign relation6 = MultiplyDivide_output;
assign relation5 = Equals_output;
assign relation6_dataready = MultiplyDivide_output_dataready;
assign relation_senddata = MultiplyDivide_multiply_senddata;
assign relation10_dataready = BooleanSelect_output_dataready;
assign port2_dataready = relation10_dataready;
assign relation9_senddata = Counter_increment_senddata;
assign relation7_senddata = Equals_input_senddata;
assign relation5_dataready = Equals_output_dataready;
assign relation2_senddata = Accumulator_input_senddata;
assign relation5_senddata = Accumulator_reset_senddata & BooleanSelect_control_senddata;
assign relation_dataready = Accumulator_output_dataready;
assign relation3_dataready = Counter_output_dataready;
assign relation7_dataready = Remainder_output_dataready;
assign relation3_senddata = Remainder_input_senddata;
assign relation6_senddata = BooleanSelect_trueInput_senddata;
assign port_senddata = relation9_senddata & relation2_senddata;


//instantiation of models from CSP model library
accumulator_new Accumulator(.out(Accumulator_output), .dataready_out(Accumulator_output_dataready), .
    senddata_in(Accumulator_input_senddata), .senddata_reset(Accumulator_reset_senddata), .in(relation2),
    .senddata_out(relation_senddata), .dataready_in(1'b1), .reset(relation5), .clk(clk));
divideby4 MultiplyDivide(.out(MultiplyDivide_output), .dataready_out(MultiplyDivide_output_dataready), .
    senddata_in(MultiplyDivide_multiply_senddata), .in(relation), .senddata_out(relation6_senddata), .
    dataready_in(relation_dataready), .reset(reset), .clk(clk));
```

```
counter_new Counter(.out(Counter_output), .dataready_out(Counter_output_dataready), .senddata_incr(
    Counter_increment_senddata), .incr(relation9), .senddata_out(relation3_senddata), .dataready_incr(1'b1
    ), .reset(reset), .clk(clk));
equals4 Equals(.out(Equals_output), .dataready_out(Equals_output_dataready), .senddata_in(
    Equals_input_senddata), .in(relation7), .senddata_out(relation5_senddata), .dataready_in(
    relation7_dataready), .reset(reset), .clk(clk));
booleanselect BooleanSelect(.out(BooleanSelect_output), .dataready_out(BooleanSelect_output_dataready), .
    senddata_in(BooleanSelect_trueInput_senddata), .senddata_control(BooleanSelect_control_senddata), .in(
    relation6), .control(relation5), .senddata_out(1'b1), .dataready_in(relation6_dataready), .
    dataready_control(relation5_dataready), .clk(clk), .reset(reset));
remainder4 Remainder(.out(Remainder_output), .dataready_out(Remainder_output_dataready), .senddata_in(
    Remainder_input_senddata), .in(relation3), .senddata_out(relation7_senddata), .dataready_in(
    relation3_dataready), .reset(reset), .clk(clk));

endmodule
```

movavg.v

```
////////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Accumulator CSP Model
//
////////////////////////////////////////////////////////////////////////


module accumulator_new (out, dataready_out, senddata_in, senddata_reset, in, senddata_out, dataready_in,
    reset, clk);

  output reg [15:0] out;
  output dataready_out;
  output senddata_in;
  output senddata_reset;
  input senddata_out;
  input [15:0] in;
  input dataready_in;
  input clk;
  input reset;

  wire rd_en_in;
  wire [15:0] fifo_out;
  wire empty_in;
  wire full_in;
  wire wr_en_in;


  //CSP Handshaking protocol
  assign senddata_in = ((~full_in) & dataready_in) | empty_in;
  assign dataready_out = (~empty_in);
  assign rd_en_in = senddata_out & (~empty_in);
  assign wr_en_in = senddata_in & dataready_in;


  always @(posedge clk, posedge reset) begin
    if (reset)
      out <= 16'd0;
    else
      if(rd_en_in)
         out <= out + fifo_out;
  end


fifo_generator_v5_3 fifo_in(.clk(clk), .rst(reset), .din(in), .wr_en(wr_en_in), .rd_en(rd_en_in), .dout(
    fifo_out), .full(full_in), .empty(empty_in));

endmodule
```

accumulator_new.v

```
////////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Counter CSP Model
//
////////////////////////////////////////////////////////////////////////

module counter_new (out, dataready_out, senddata_incr, incr, senddata_out, dataready_incr, reset, clk);

  output reg [15:0] out;
  output dataready_out;
  output senddata_incr;
  input senddata_out;
  input [15:0] incr;
  input dataready_incr;
  input clk;
  input reset;

  wire rd_en_incr;
  wire [15:0] fifo_out;
  wire empty_incr;
  wire full_incr;
  wire wr_en_incr;


  //CSP Handshaking protocol
  assign senddata_incr = ((~full_incr) & dataready_incr) | empty_incr;
  assign dataready_out = (~empty_incr);
  assign rd_en_incr = senddata_out & (~empty_incr);
  assign wr_en_incr = senddata_incr & dataready_incr;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 16'd0;
    else
      if(rd_en_incr)
         out <= out + 1'b1;
  end


fifo_generator_v5_3 fifo_incr(.clk(clk), .rst(reset), .din(incr), .wr_en(wr_en_incr), .rd_en(rd_en_incr),
    .dout(fifo_out), .full(full_incr), .empty(empty_incr));

endmodule
```

counter_new.v

```
////////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Divideby4 CSP Model
//
// Funtionality : Divides the input by 16'4, operating under CSP MoC
//
////////////////////////////////////////////////////////////////////////

module divideby4 (out, dataready_out, senddata_in, in, senddata_out, dataready_in, reset, clk);

  output reg [15:0] out;
  output dataready_out;
  output senddata_in;
  input senddata_out;
  input [15:0] in;
  input dataready_in;
  input clk;
  input reset;

  wire rd_en_in;
  wire [15:0] fifo_out;
  wire empty_in;
  wire full_in;
  wire wr_en_in;


  //CSP Handshaking protocol
  assign senddata_in = ((~full_in) & dataready_in) | empty_in;
  assign dataready_out = (~empty_in);
  assign rd_en_in = senddata_out & (~empty_in);
  assign wr_en_in = senddata_in & dataready_in;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 16'd0;
    else
      if(rd_en_in)
        out <= fifo_out/4;
  end


fifo_generator_v5_3 fifo_in(.clk(clk), .rst(reset), .din(in), .wr_en(wr_en_in), .rd_en(rd_en_in), .dout(
    fifo_out), .full(full_in), .empty(empty_in));

endmodule
```

divideby4.v

```
//////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Eqauls4 CSP Model
//
// Funtionality : Produces an output '1' when input value is equal
//                to 16'4, operating under CSP MoC
//
//////////////////////////////////////////////////////////////////////

module equals4 (out, dataready_out, senddata_in, in, senddata_out, dataready_in, reset, clk);

  output reg out;
  output dataready_out;
  output senddata_in;
  input senddata_out;
  input [15:0] in;
  input dataready_in;
  input clk;
  input reset;

  wire rd_en_in;
  wire [15:0] fifo_out;
  wire empty_in;
  wire full_in;
  wire wr_en_in;


  //CSP Handshaking protocol
  assign senddata_in = ((~full_in) & dataready_in) | empty_in;
  assign dataready_out = (~empty_in);
  assign rd_en_in = senddata_out & (~empty_in);
  assign wr_en_in = senddata_in & dataready_in;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 1'b0;
    else
      if(rd_en_in)
         if(fifo_out==4)
           out <= 1'b1;
         else
           out <= 1'b0;
  end


fifo_generator_v5_3 fifo_in(.clk(clk), .rst(reset), .din(in), .wr_en(wr_en_in), .rd_en(rd_en_in), .dout(
    fifo_out), .full(full_in), .empty(empty_in));

endmodule
```

equals4.v

```
//////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit Remainder4 CSP Model
//
// Funtionality : Produces the remainder with a divisor = 16'd4 being applied
//                to the input, operating under CSP MoC
//
//////////////////////////////////////////////////////////////////////


module remainder4 (out, dataready_out, senddata_in, in, senddata_out, dataready_in, reset, clk);

  output reg [15:0] out;
  output dataready_out;
  output senddata_in;
  input senddata_out;
  input [15:0] in;
  input dataready_in;
  input clk;
  input reset;

  wire rd_en_in;
  wire [15:0] fifo_out;
  wire empty_in;
  wire full_in;
  wire wr_en_in;


  //CSP Handshaking protocol
  assign senddata_in = ((~full_in) & dataready_in) | empty_in;
  assign dataready_out = (~empty_in);
  assign rd_en_in = senddata_out & (~empty_in);
  assign wr_en_in = senddata_in & dataready_in;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 16'd0;
    else
      if(rd_en_in)
        out <= fifo_out%4;
  end


fifo_generator_v5_3 fifo_in(.clk(clk), .rst(reset), .din(in), .wr_en(wr_en_in), .rd_en(rd_en_in), .dout(
    fifo_out), .full(full_in), .empty(empty_in));

endmodule
```

remainder4.v

```
////////////////////////////////////////////////////////////////////
//
// 2-input 16-bit BooleanSelect CSP Model
//
////////////////////////////////////////////////////////////////////

module booleanselect (out, dataready_out, senddata_in, senddata_control, in, control, senddata_out,
    dataready_in, dataready_control, reset, clk);

  output reg [15:0] out;
  output dataready_out;
  output senddata_in;
  output senddata_control;
  input senddata_out;
  input [15:0] in;
  input control;
  input dataready_in;
  input dataready_control;
  input clk;
  input reset;

  wire rd_en_in;
  wire rd_en_control;
  wire [15:0] fifo_out_in;
  wire fifo_out_control;
  wire empty_in;
  wire empty_control;
  wire full_in;
  wire full_control;
  wire wr_en_in;
  wire wr_en_control;


  //CSP Handshaking protocol
  assign senddata_in = ((~full_in) & dataready_in) | empty_in;
  assign senddata_control = ((~full_control) & dataready_control) | empty_control;
  assign dataready_out =  (~empty_in) & (~empty_control);
  assign rd_en_in = senddata_out & (~empty_in);
  assign rd_en_control = senddata_out & (~empty_control);
  assign wr_en_in = senddata_in & dataready_in;
  assign wr_en_control = senddata_control & dataready_control;


  always @(posedge clk, negedge reset) begin
    if (~reset)
      out <= 16'd0;
    else
      if(rd_en_in & rd_en_control)
        if(fifo_out_control)
           out <= fifo_out_in;
        else
```

```
            out <= out;
  end


fifo_generator_v5_3 fifo_in(.clk(clk), .rst(reset), .din(in), .wr_en(wr_en_in), .rd_en(rd_en_in), .dout(
    fifo_out_in), .full(full_in), .empty(empty_in));
fifo_generator_v5_3 fifo_control(.clk(clk),.rst(reset), .din(control), .wr_en(wr_en_control), .rd_en(
    rd_en_control), .dout(fifo_out_control), .full(full_control), .empty(empty_control));

endmodule
```

booleanselect.v

Bibliography

[1] Gordon E. Moore. Cramming more components onto integrated circuits. *Electronics, Volume 38, Number 8, April 19, 1965*, 38, 1965.

[2] Why is autocode generation slow to adoption?, http://www.eetimes.com/design/embedded/4007112/why-is-autocode-generation-slow-to-adoption-.

[3] Case study: High-level synthesis - ready for prime-time?, http://new.eetimes.com/design/eda-design/4210994/case-study–high-level-synthesis—ready-for-prime-time-?pagenumber=0.

[4] Joseph Buck, Soonhoi Ha, Edward A. Lee, and David G. Messerschmitt. Ptolemy: A framework for simulating and prototyping heterogeneous systems, 1992.

[5] John Davis, II, John Davis Ii, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. Ptolemy ii: Heterogeneous concurrent modeling and design in java(volume 1:introduction to ptolemy ii), 1999.

[6] *Mastering SIMULINK 2*. Prentice Hall, 1998.

[7] *LabVIEW User Manual*. National Instruments Corporation, 2003.

[8] Hilding Elmqvist, Sven Erik Mattsson, and Martin Otter. Modelica — the new object-oriented modeling language. In *in proceedings of the 12th European Simulation Multi-conference*, pages 127–131, 1998.

[9] C.A.R. Hoare. Communicating sequential processes. *Communications of the ACM*, 1978.

[10] Bin Xue, S. K. Shukla, and S. S. Ravi. Minimizing back pressure for latency insensitive system synthesis. In *Formal Methods and Models for Codesign (MEMOCODE), 2010 8th IEEE/ACM International Conference*, pages 189–198, 2010.

[11] John Davis, II, John Davis Ii, Mudit Goel, Christopher Hylands, Bart Kienhuis, Edward A. Lee, Jie Liu, Xiaojun Liu, Lukito Muliadi, Steve Neuendorffer, John Reekie, Neil Smyth, Jeff Tsay, and Yuhong Xiong. Ptolemy ii: Heterogeneous concurrent modeling and design in java(volume 3: Ptolemy ii domains), 1999.

[12] C. A. R. Hoare. Communicating sequential processes, 2004.

[13] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *Journal of the ACM (JACM)*, 1984.

[14] Gajski-kuhn chart, http://alt.schlosser.info/diplomathesis/thesissu7.html.

[15] *ISE 11.1 Design Suite Software Manuals and Help - PDF Collection*. Xilinx, Inc., 2009.

[16] Impulse accelerated technologies - c-to-fpga tools, http://www.impulseaccelerated.com.

[17] Impulse accelerated technologies - c-to-fpga tools, http://www.impulseaccelerated.com/products_covalidator.htm.

[18] *Introduction to the Quartus II Software*. Altera Corporation.

[19] Element cxi - home, http://www.elementcxi.com/.

[20] *ModelSim Users Manual*. Mentor Graphics Corporation, 2008.

[21] "javax xml parser", http://download.oracle.com/javase/1.4.2/docs/api/javax/xml/parsers/packagesummary.html.

[22] "java document object model(dom) package", http://download.oracle.com/javase/1.4.2/docs/api/org/w3c/dom/packagesummary.html.

[23] *XST User Guide*. Xilinx, Inc., 2009.

[24] *ISim User Guide*. Xilinx, Inc., 2009.

[25] Sumit Gupta, Nikil Dutt, Rajesh Gupta, and Alex Nicolau. Coordinated parallelizing compiler optimizations and high-level synthesis. *ACM Trans. Des. Autom. Electron. Syst*, 9:2004, 2002.

[26] Philippe Coussy, Ghizlane Lhairech-Lebreton, Dominique Heller, and Eric Martin. Gaut a free and open source high-level synthesis tool.