APPLICATION OF SYNCHRONOUS SYNTHESIS TOOLS FOR HIGH-LEVEL ASYNCHRONOUS DESIGN

by

Yanyi Zhao

A thesis submitted to the faculty of
The University of Utah
in partial fulfillment of the requirements for the degree of

Master of Science

in

Electrical Engineering

Electrical and Computer Engineering

The University of Utah

December 2004

Copyright © Yanyi Zhao 2004

All Rights Reserved

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Yanyi Zhao

This thesis has been read by each member of the following supervisory committee and by majority vote has been found to be satisfactory.

Chair: Chris J. Myers

Priyank Kalla

Eric Mercer

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Yanyi Zhao in its final form and have found that (1) its format, citations, and bibliographic style are consistent and acceptable; (2) its illustrative materials including figures, tables, and charts are in place; and (3) the final manuscript is satisfactory to the Supervisory Committee and is ready for submission to The Graduate School.

Date

Chris J. Myers
Chair, Supervisory Committee

Approved for the Major Department

Marc Bodson
Chair/Dean

Approved for the Graduate Council

David S. Chapman
Dean of The Graduate School

ABSTRACT

This thesis presents research work in the application of synchronous synthesis tools for asynchronous design, describing a method that starts from a channel-level VHDL code and automatically translates it into different types of VHDL code until it gets to structural-level VHDL code, which can be simulated and synthesized using existing synchronous synthesis tools. In particular, this method shows how to automatically transform a high-level specification in channel-level VHDL to a handshaking-level VHDL specification, and then partition it into a controller and datapath with bundling constraints. Next, the method transforms the handshaking-level datapath into a structural-level VHDL specification automatically. At this point, the bundling constraints are added into the datapath and also the controller. For the controller, our method uses ATACS [18] to synthesize it, and for all of the datapath components, our method uses Xilinx WebPack for synthesis. Finally, the results can be combined and physically designed on an FPGA. A testing methodology for this FPGA using a *linear feedback shift register* (LFSR) and signature analyzer has been developed. This thesis evaluates the usage of a synchronous synthesis tool, Xilinx WebPack for asynchronous design. Finally, this thesis describes the application of this method to several examples including an asynchronous dithery for an MPEG decoder.

To my parents for their great love and support

CONTENTS

LIST OF FIGURES

ACKNOWLEDGMENTS

My sincere thanks to my advisor, Dr. Chris Myers, for his great support and guidance over the years. I thank him for his help and direction in helping me understand research material, and I have learned tremendously from him. I would also thank my committee members Dr. Priyank Kalla and Dr. Eric Mercer, for reading my thesis, and for being additional source of advice and support during my study here.

I would like to thank the members of our research group – Scott Little, David Walter, Curt Nelson, Eric Peskin, and Kip Killpack – for discussions on various topics, and for their support and friendship over the years. I would like to express special appreciation to Scott Little and David Walter for assistance and advice on all kinds of questions, and to Curt Nelson for always bringing a laugh to the office. I also thank all the other people in the office for making my stay here enjoyable.

My friends have helped me in several ways, I offer great thanks to them. I would like to specially thank Sudheesh Madayi and Sandhya for their tremendous support and encouragement, and thank Sandhya for cooking delicious food for me.

The most special thanks goes to my parents, to whom this thesis is dedicated. I could not have done this without their support, encouragement and love. They have always been there for me no matter what. Thanks for believing in me, and for encouraging me to complete the degree.

This work was supported by the Semiconductor Research Corporation, Contract nos.2002-TJ-1024. I sincerely thank SRC for the research grant over the years.

CHAPTER 1

INTRODUCTION

Asynchronous (or self-timed) circuit design has been an active research area since the mid 1950s. Asynchronous circuits are circuits without a global clock. Instead, handshaking signals are used among all the processes in order to have proper communication, synchronization, and execution of the necessary operations. Although research in asynchronous design started a long time ago, it has seen only limited applications. The main advantages of asynchronous circuit design are:

1. Low power dissipation – due to the occurance of no unnecessary transitions.

2. Lower electromagnetic interference (EMI) – The high speed clock switching all over the chip at approximately the same time is the largest source of this type of interference. Asynchronous systems using local handshaking can have much lower EMI because the local switching of the handshake signals is not correlated in time [1].

3. High operating speed – In asynchronous circuits, the operating speed is determined by actual local latencies, not global worst-case latency, so it has the potential for higher speed than synchronous circuits.

4. No clock distribution and clock skew problems – In asynchronous circuits, there is no global clock signal to be distributed with minimal phase skew across the circuit.

5. Better composability and modularity – Asynchronous circuits use simple handshaking interfaces and local timing instead of global timing, which is much easier than trying to synchronize all the clocks in a synchronous system.

Despite all of these advantages of asynchronous circuits, there are also some drawbacks. In asynchronous circuits, the control logic that implements the handshaking usually limits circuit speed and can consume significant silicon area and power. Therefore, the question is whether asynchronous techniques can deliver on any of their promises. Another disadvantage of asynchronous circuit design is that there are not many CAD tools for synthesis and testing, which makes it difficult for both academia and industry to use. This thesis targets this deficiency by trying to determine whether the existing synchronous tools for FPGA design can be safely used.

1.1 Our Work

In particular, the goal of this research is to develop a design flow for an asynchronous model from a high-level description using communication channels and leveraging synchronous synthesis tools whenever possible. For asynchronous hardware systems, channel-level VHDL are used to communicate between processes. For example, one process may try to send data through a channel while another process receives data from the same channel. This project assumes the asynchronous design is specified using a channel-level VHDL model, which can be simulated by commercial simulators such as Mentor's ModelSim [2]. Our method then automatically translates this into a handshaking-level VHDL specification. Next, our tool separates it automatically into a controller and a datapath. First, our tool extracts the datapath from the handshaking-level VHDL code automatically. In order to automate this process, our tool needs to take the channel-level VHDL code for the datapath and extract all of the channels, sends and receives, and expands each of the channels into two handshaking signals, request and acknowledge. Then our tool needs to take out all of the matching delays from the VHDL code. After these modifications, our tool puts the handshaking-level VHDL code into the Xilinx WebPack tool [3] for synthesis. Using the WebPack tool to synthesize the datapath not only produces the synthesized circuit, but also derives the *bundling constraints* for users. *Bundling constraints* are timing assumptions required for the datapath. Our tool can use these data and add them back into the synthesized datapath circuit in order to make this asynchronous design work correctly. Our tool uses buffers to mimic the time delays to add bundling constraints back into the datapath. Different bundling constraints are generated by using different numbers of buffers and different routing schemes. We can also add a certain amount of percentage adjustment to the maximum bundling constraint in order to make sure that the data are applied to the data lines before the request is asserted. The controller is described using handshaking level VHDL code, and it is synthesized using ATACS, an asynchronous controller synthesis tool [18]. The output of this tool is structural-level VHDL code. Therefore, the controller and the datapath synthesis are fully automated.

This process is implemented using three Perl [19] scripts. The first script translates the channel-level VHDL specification into its corresponding handshaking-level VHDL specification. The second script separates the datapath and the controller from the handshaking-level VHDL specification and synthesizes the datapath into different components. The third script calculates the bundling constraints, then uses ATACS to synthesize the controller and uses Webpack to synthesize the datapath.

After both the structural level controller and datapath are ready, they can be combined, and the WebPack tool can be used to implement the entire design in an FPGA. At this point, CAD tools for synchronous physical design are leveraged to produce an FPGA circuit implementation. In addition, after the whole design is completed, it can be tested for correctness, which integrates this design with a testing circuit. We have developed a testing methodology to evaluate this process. Testing a circuit requires the application of an appropriate set of testing vectors and the comparison of response between the actual circuit and the correct response [21]. There are many ways to test it. Some of them can be complicated and expensive: for instance, using an external tester. Several techniques have been proposed that move some or all of the tester functions onto the chip itself in order to reduce the complexity of the tester, which is used in this project. Besides the cost, there are also some other problems, such as how to generate the test patterns without making the turnaround time too long and spending a huge amount of the computation cost; how to prevent the number of testing patterns from becoming too large to be efficiently handled by the testing hardware; determining how much time needs to be used in order to apply the testing patterns, etc. In this testing circuit, there are three components: a random test generator, the circuit that needs to be tested, and the signature analyzer. The complete design flow is shown in Figure 1.1.

1.2 Related Work

A great amount of research has been invested in exploring efficient methods for synthesizing asynchronous designs. Fortunately, many researchers have been working on this topic and have had varying success. By studying their work, some researchers developed particular circuit structures in order to modify channel-level descriptions to logical integrated circuits. For example, Burns and Martin developed a translation method from CSP [14, 15, 16]. Brunvand [12, 13] proposed using concurrent communication programs to design asynchronous circuits. Akella and Gopalakrishnan used a new channel language called hopCP [7, 8, 9] to develop a syntax-directed method. Berkel and Rem used a language called Tangram to produce a handshake circuit implementation [10, 11]. There are two major differences between our work and those previously described. First, we synthesize the circuit using conventional circuit structures instead of developing new ones. Second, we utilize synchronous tools not only for datapath and physical design, but also to map the controller circuit. Some researchers have attempted to use synchronous synthesis tools to synthesize asynchronous designs by developing new library cells and specializing the tools. For example, Ligthart and Smith proposed high level design for asynchronous logic in [20, 24]. However, in this project, we attempt to understand what the synchronous synthesis tools can do and cannot do instead of specializing them. We also begin from a channel-level model in our design flow.

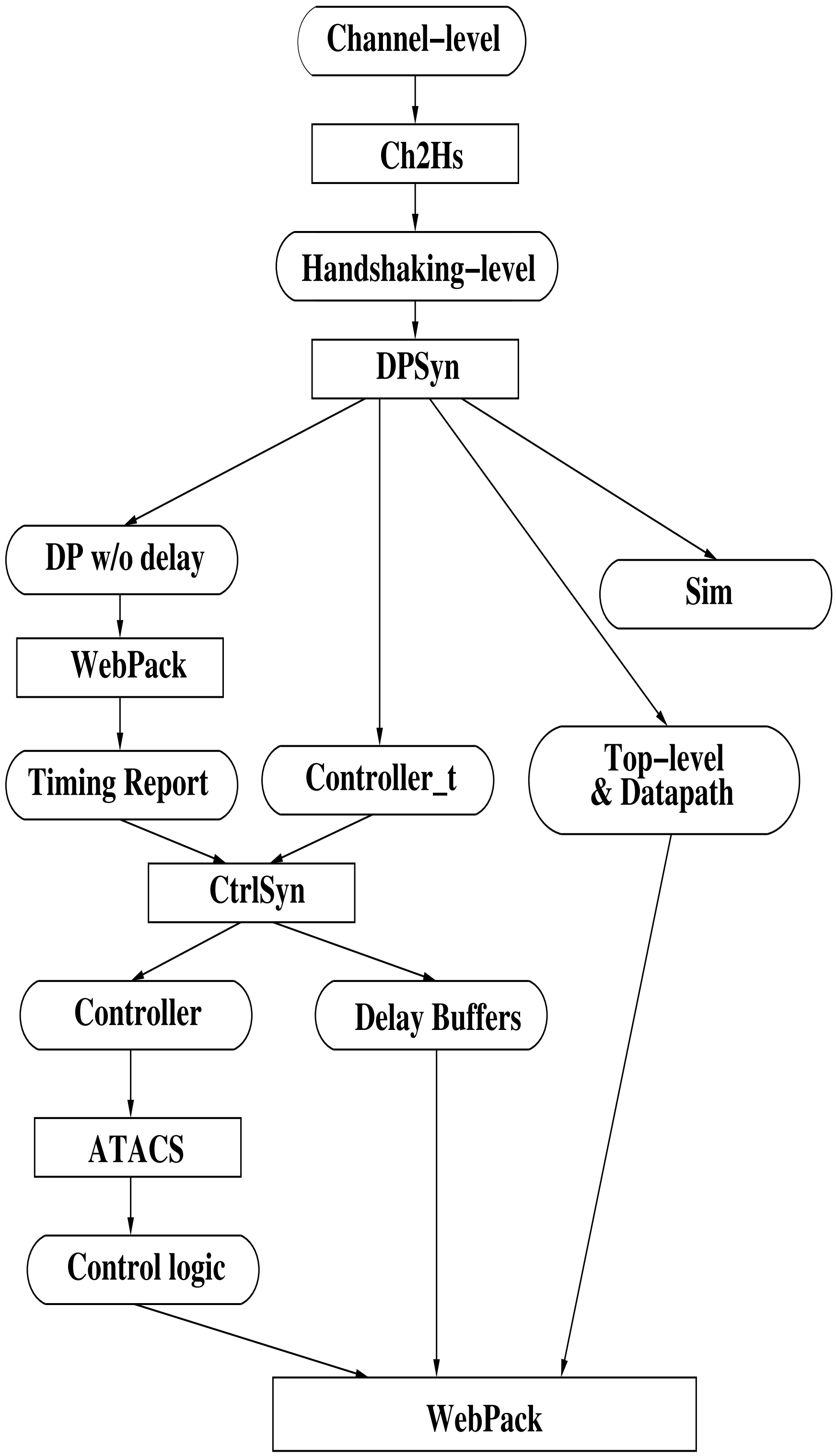

Figure 1.1. Design flow.

1.3 Comparison Between Synchronous and Asynchronous Circuits

Most digital circuits these days are designed and fabricated synchronously. There are two fundamental assumptions for synchronous circuit design that help to simplify the design. One is that all of the signals are binary, and the other one is that a global clock is employed, which defines that a clock signal is distributed throughout the whole circuit. Asynchronous circuits are essentially different. The first assumption is the same; it also uses binary signals, but there is no global clock signal. Instead of using a global clock to discretize time through the whole circuit, asynchronous circuits use handshaking between all of the components for communication and synchronization.

Synchronous circuits have been studied and used for a long time. There are many commercial simulation and synthesis tools for synchronous circuits. However, only academic tools exist for asynchronous circuits. These two types of circuits have many methods in common, except the global clock. Therefore, since there is a rich set of tools developed for synchronous designs, they can be used for asynchronous designs. This is also one of the motivations for this thesis.

In order to apply synchronous synthesis tools for asynchronous circuit design, the difference between these two types of circuit designs has to be studied. For synchronous circuits, the only aspect that matters is that the signals are ready and stable at the clock edge. For asynchronous circuits, since there is no clock, all of the signals have to be valid at all times. This means that all external signals must be hazard free. In order to achieve this, sometimes it is necessary to ignore the hazards on the internal signals. This is the main reason that it is difficult to synthesize asynchronous circuits, which also induces many researchers to propose different methods based on different assumptions.

If a circuit is in a stable state, it does not spontaneously produce a hazard. A hazard is an unwanted glitch on a signal for the circuit designers; it relates to the dynamic operation of a circuit. It also relates to the dynamics of the input signals just like the delays in the gates and the wires in the circuit. Therefore, it is very important to state clearly what kind of delay models have been used and what assumptions have been made between the circuit and its environment. Synchronous tools do not deal with this issue.

Another difference is *bundling constraints*, which is a timing assumption required for the datapath. This assumption basically needs the data to be ready on the data lines in advance before the controller is requested. We use wait statements in the model to simulate bundling constraints. However, synchronous synthesis tools cannot handle this. Therefore, the synthesis tool has to be convinced to solve the bundling constraint issue. We achieve this by calculating the different bundling constraints for each assignment, and we then add a certain number of buffers to mimic the bundling constraints. The number of buffers we need to use depends on the delay time.

1.4 Contributions

The major contribution of this master's thesis is the application of synchronous synthesis tools to high-level asynchronous design. There are many commercial tools for synchronous synthesis, but almost no tools for asynchronous synthesis. This thesis helps us know whether synchronous tools, the WebPack tool in particular, can be used to synthesize asynchronous designs. In particular, code is developed to translate channel-level VHDL code to a handshaking-level VHDL code for controller and datapath synthesis. Methods are also developed to extract the necessary bundling constraints for the datapath. To achieve this goal, we stitch together the design flow shown in Figure 1.1. Three Perl scripts are provided to automate the entire design process from a high-level channel description to an FPGA.

Another contribution of this thesis is to test the whole design using the idea of signature analysis. *Signature analysis* is a compression technique based on the concept of *cyclic redundancy checking* (CRC) and realized in hardware using *linear feedback shift registers* (LFSRs). The whole testing method can be separated into three stages: input generation, analysis for the output responses, and actual implementation. For the random testing generator, which is constructed of a simple circuit called a *linear feedback shift register* or *LFSR*, it randomly generates the testing patterns and sends them to our circuit. After going through our circuit, the outputs are sent into the signature analyzer. After all the input patterns have been applied, the final signature is compared with the precomputed correct value. This method is much simpler and also reduces memory space compared with checking the output values bit-by-bit. While constructing this testing circuit, we experienced several problems, such as how to connect the inputs and outputs and how to connect the handshaking signals, acknowledge and request. Each receive channel would connect with one LFSR, and each send channel would connect with one signature analyzer. This signature analyzer also need to be halted in order to exempt indefinite loops. During the course of this research, case studies are developed to evaluate this design flow. In particular, after synthesis, the design is evaluated for correctness. Through this research, a better understanding of asynchronous systems is achieved, and a better synthesis method for asynchronous design is developed.

1.5 Thesis Outline

Having given a short description of our work and the contributions of this thesis, we now delve into the asynchronous design background in Chapter 2. It begins with all of the basic definitions about asynchronous design, introduces channel-level modeling using the channel package, bundling constraints, handshaking-level modeling using the handshake package, and communication protocol (4-phase vs. 2-phase). Also the structural-level VHDL is produced by ATACS.

Chapter 3 introduces how to use synchronous tools for high-level asynchronous design. It starts with a short literature survey of previous asynchronous synthesis tools. Then the design flow for our work is described in detail. Next, we describe the algorithms used in the Perl scripts, how synthesis is done using ATACS and WebPack, and how we test the FPGA design. Finally, we describe all the difficulties we faced.

Chapter 4 focuses on the case studies during this research. The results of all the case studies have been discussed. It starts by introducing all of the examples and then describes both simulation and synthesis results by using Xilinx WebPack. Finally, it discusses what is working and what is not working, and the possible reasons.

Finally, Chapter 5 presents the conclusion and future work. It includes all of the concluding remarks, summarizes the results, and gives all the major contributions of the work. It also discusses future work in the area of asynchronous design using synchronous tools.

CHAPTER 2

ASYNCHRONOUS DESIGN BACKGROUND

This chapter gives an overview of all the terminologies and background materials related to this thesis. In particular, this chapter describes the different types of VHDL code utilized.

VHDL is a hardware description language. It is a standard language that is developed by the IEEE (Institute of Electrical and Electronics Engineers). It is widely used for describing everything from complete systems like personal computers to small logical gates on internal integrated circuits. At the same time, it also provides access to CAD (computer-aided design) tools in order to simulate and synthesize designs. There have been a few revisions for the VHDL language. This project uses the VHDL'93 standard.

This chapter is organized as follows. Section 2.1 presents channel-level modeling in VHDL using our channel package. Section 2.2 defines handshaking-level modeling using our handshake package. Section 2.3 describes structural VHDL models produced by the ATACS tool.

2.1 Channel-Level Modeling

This section describes channel-level modeling and the channel package we used in this thesis. A channel can be seen as a communication bridge between two processes. It connects two operating processes from one point to the other. One process uses the channel to send the data to the other process, and the other process uses the channel to receive this data.

This thesis uses the channel package developed by our research group [18]. It can be used as a standard VHDL package in simulation with any commercial tool that supports VHDL'93 syntax. It also can be synthesized by ATACS, a synthesis tool developed by our research group, which can recognize all the specifications in the channel package [23].

The channel package introduces the *channel* data type. The channel package also defines some operations on this data type namely, *send*, *receive*, and *probe*. The following VHDL is a variation on the wineshop example from [18]. The shop receives the data from winery through *wine_delivery* channel and stores it into signal *shelf*. After some calculation, the shop sends out the data stored in *shelf1* signal to patron through the *wine_selling* channel. The VHDL code below shows more details on how to use the channel package for the wineshop example.

```
-- shop.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity shop is
  port(wine_delivery:inout channel:=init_channel;
       wine_selling:inout channel:=init_channel);
end shop;
architecture behavior of shop is
  signal shelf:std_logic_vector(2 downto 0);
  signal shelf1:std_logic_vector(2 downto 0);
begin
  shop:process
  begin
    receive(wine_delivery,shelf);
    shelf1 <= shelf + 1;
    wait for 5 ns;
    send(wine_selling,shelf1);
  end process shop;
end behavior;
```

The code can be understood in advance. The first line is a comment line that states the name of the file. The next few lines show what libraries and packages have been used in this file. The first one shows that the library used is *IEEE*. The following three lines show the packages used. The *std_logic_1164* package in the IEEE library defines the std_logic data type that is used for all wires. The *std_logic_arith* in the IEEE library defines some types and basic arithmetic operations for representing integers in standard ways. The *std_logic_unsigned* library extends the *std_logic_arith* library to handle *std_logic_vector* values as unsigned integers. The *nondeterminism* and *channel* packages are ones designed for modeling asynchronous systems. The *nondeterminism* package includes functions to generate delays and random selections in the environment. It is useful for simulation and synthesis. Examples of their use are given later in the section. The *channel* package, as mentioned previously, defines the channel data type. The shop *entity* defines the interface to the shop. In this *shop* example, there are two channels, *wine_delivery* and *wine_selling*, which are declared as ports in the entity. Each signal has a direction, which can be *in*, *out*, or *inout*. The direction of a channel is always *inout* instead of *in* or *out* because even though the data flow in one way, the control for the communication flows in both directions. Next, the port type is given. In this example, the port types are channel. Channels are initialized using the *init_channel* function. After the entity declaration, the architecture of the shop is given, which includes the internal signal declaration and the shop process. The internal signal *shelf* is a 3-bit signal, which carries the data to be transmitted or copied. In the shop process, there are *receive* and *send* function calls. *Receive* is defined in the channel package. It takes two parameters, a communication channel and a *std_logic* or *std_logic_vector* to store the data received. For this example, the shop receives a bottle of wine over the wine_delivery channel by using the receive procedure call. During this receive procedure, it waits until the winery gets ready to deliver a bottle of wine, and then the shop accepts the delivery. After the shop receives the bottle of wine on the shelf, the winery adds 1 and then sends the wine to the patron, which is accomplished using the send statement. *Send* is also defined in the channel package, and takes two parameters, a communication channel and a *std_logic* or *std_logic_vector* for the data to transmit. In this example, the shop sends the bottle of wine to the patron over the *wine_selling* channel. During this send procedure, it waits until the Patron gets ready to receive the bottle of wine, and then delivers it.

At this point, the whole process can be simulated by any commercial VHDL simulator tools, and the controller can be synthesized using the ATACS tool developed in our research group. ATACS can not synthesize the datapath. In this project, we hope to use our tool to synthesize the control part, and use a commercial synchronous tool to synthesize the datapath. In theory, after having those two parts, we can combine them in a meaningful way.

Besides all of the basic syntax described above, there are also some special functions defined in the *nondeterminism package* and *channel package*. As an example, the following winery process shows the delay function and random selection function.

```
winery:process
begin
  bottle <= selection(8,3);
  wait for delay(5,10);
  send(WineryShop,bottle);
end process winery;
```

This process models the winery, starting by randomly selecting a type of wine by using the *selection(i1, i2)* function. It has two integer parameters *i1* and *i2*. *i1* indicates the number of choices we have available. In this case, there are eight types of wine; *i2* shows the size of the return value in bits, which in this case is the size of the *std_logic_vector bottle*.

The next line uses the *delay* function. The *delay* function is defined in the nondeterminism package and is used to make timing assumptions for the circuit. It is defined as *delay(l,u)*, in which l refers to a lower timing bound, and u refers to an upper timing bound. This function randomly selects a time between these two bounds. In this case, the circuit randomly selects the time between 5 and 10 *ns*. One nanosecond is the default time unit.

Another useful function from the channel package is *probe*, which is shown in the following patron2 process that includes both *OldShopPatron* and *NewShopPatron* channels.

```
patron2:process
begin
  if (probe(OldShopPatron)) then
    receive(OldShopPatron,bag);
```

```
    wine_drunk <= wine_list'val(conv_integer(bag));
  elsif (probe(NewShopPatron)) then
    receive(NewShopPatron,bag);
    wine_drunk <= wine_list'val(conv_integer(bag));
  end if;
  wait for delay(5,10);
end process patron2;
```

It checks whether there is a pending communication on a channel by using the channel as a parameter and returns a boolean value to indicate the presence of a pending communication. In this case, the patron probes on both the *OldShopPatron* and the *NewShopPatron* channels to check which Shop is requesting to send him wine. If the *OldShop* wants to send him wine, then probe (OldShopPatron) returns true, and he receives the wine. But if not, then he checks the *NewShopPatron* channel. If neither of them wants to sell him wine, then he waits for a random delay time, which is between $5ns$ to $10ns$, and then starts checking again.

2.2 Handshaking-Level Modeling

A channel communication must be implemented with one or more signal wires using a handshaking protocol. This section describes such a handshaking-level model. First of all, we describe different types of protocols and then focus on how to use a 4-phase bundled-data handshaking protocol to implement channel communication. This is described in VHDL using our handshaking package. Channel communication must handle data transfer. There are two usual ways to transfer data in an asynchronous system: *dual-rail* and *bundled-data*.

For a dual-rail protocol, each data bit is encoded by two wires. The usual encoding way is when there are no data, the values on wire0 and wire1 are 0s; when the data is 0, then the value on wire0 and wire1 are 1 and 0; when the data is 1, respectively, the values on wire0 and wire1 are 0 and 1, respectively. The wires wire0 and wire1 are never 1 at the same time. In the dual-rail protocol, it is not necessary to use a request signal since validity information has been encoded in the data. The acknowledge signal is still required to indicate the data have been received.

Bundled-data are also called single-rail in some other texts. By looking at these two names, we can see that bundled-data describe the timing relationship between the data signals and handshaking signals, but single-rail data describe using one wire to carry one bit of data. Also this type of data compares to the dual-rail described above. This thesis mostly uses bundled-data instead of single-rail because the timing issue is very critical in the project. Bundled-data presents the situation in which all the data signals are separated with the handshaking wires, and the data signals still use the normal boolean levels to pass the information. Each bit of data is represented by a single wire. Besides this, two more wires are needed: request and acknowledge. Request indicates when the data on the data wires are valid, and acknowledge indicates when the data have been received. Both of the wires are bundled with the data signals. Figure 2.1 shows what the bundled-data protocol looks like.

For control signaling, there are two choices, 4-phase and 2-phase. The 4-phase protocol uses boolean levels to encode information. The term 4-phase refers to the number of communication events in the protocol. It is completed by four events: request (req) goes high, which starts a handshake, then the acknowledge (ack) goes high, which shows that the data have been received; next, request(req) is set low, which starts the return-to-zero (RTZ) process; then the acknowledge (ack) goes low, which completes the handshake. The disadvantage of 4-phase handshaking protocol is the redundancy of return-to-zero transitions, which uses unnecessary

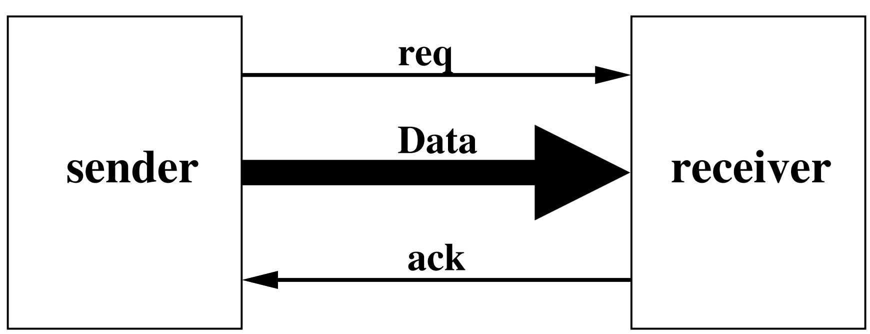

Figure 2.1. Bundled-data data transfer.

time and energy and can affect the performance.

There is also a 2-phase handshaking protocol. This protocol is very simple. The term 2-phase refers to the number of communication events. It is completed by two events: the sender sets request (req) high, which starts the handshaking, and the receiver sets acknowledge (ack) high, which completes the handshaking. If this is the first handshake, then the following handshake uses a request (req) low, and an acknowledge (ack) low to complete the handshaking. In other words, signal transitions are the events in this protocol. The disadvantages of 2-phase handshaking are after a complete handshake, the levels of wires are different from what they were before, which lead to a more complex circuit. And also, some of the datapath components may require level-sensitive control, requiring 2-phase to 4-phase converters.

Before getting into all the details, we first describe the 2-phase bundled-data protocol. For the 2-phase bundled-data protocol, which is illustrated in Figure 2.2, the request and acknowledge wires become like signal transitions, so these wires act the same between transitions 0→1, and 1→0. Both of them present the events for the signal. Therefore, we do not need to worry about the return-to-zero process. At this point, the 2-phase bundled-data protocol seems to avoid the disadvantages of 4-phase bundled-data protocol, which cuts off the unnecessary time and energy by not having the return-to-zero transition. To compare those two protocols, in

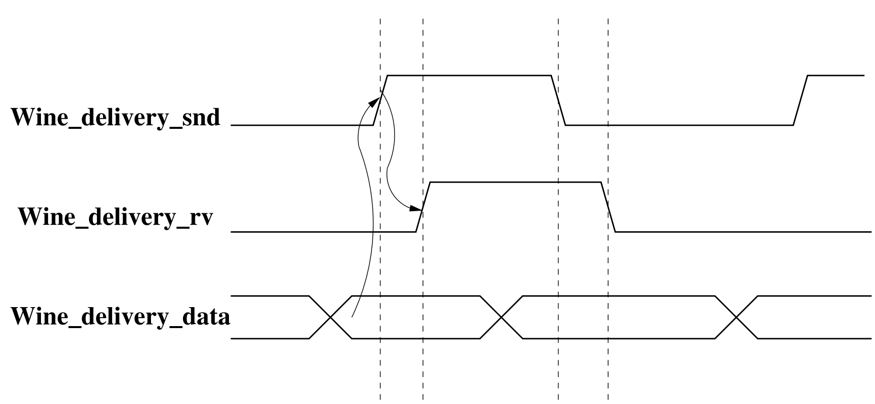

Figure 2.2. 2-phase bundled-data.

theory, the 2-phase bundled-data protocol should lead to faster circuits than the 4-phase bundled-data protocol; however, it is complicated to implement 2-phase protocol logic. Therefore, it is hard to say which protocol is better.

In this thesis, we choose to use the 4-phase bundled-data handshaking protocol. Figure 2.3 shows what it looks like.

In the following, we describe our protocol implementation using a handshaking VHDL model. The difference between handshaking-level VHDL and channel-level VHDL is that the circuit communicates using signals of type *std_logic*. Our handshake package includes the following operations on *std_logic* signals, *guard*, *guard_and*, *guard_or*, *assign*, and *vassign*. The *guard* operation is defined as *guard*(s, v), where s refers to a signal, and v refers to a value. The operation means to wait for the signal s to take the value v, if it is not already at that value. The *guard_and* operation is defined as *guard_and*(*s1*, *v1*, *s2*, *v2*), which means to wait for both signals *s1* and *s2* to take their values *v1* and *v2*. The *guard_or* operation is defined as *guard_or*(*s1*, *v1*, *s2*, *v2*), which means to wait until either *s1* or *s2* takes the appropriate value.

The next new operation is assign, which is defined as *assign*(s, v, l, u). It takes a signal s, and changes it to the value v at some point between the lower time bound, l, and the upper time bound, u. However, there is a precondition for assign to happen, which is that s is not equal to u. The *vassign* operation is defined as

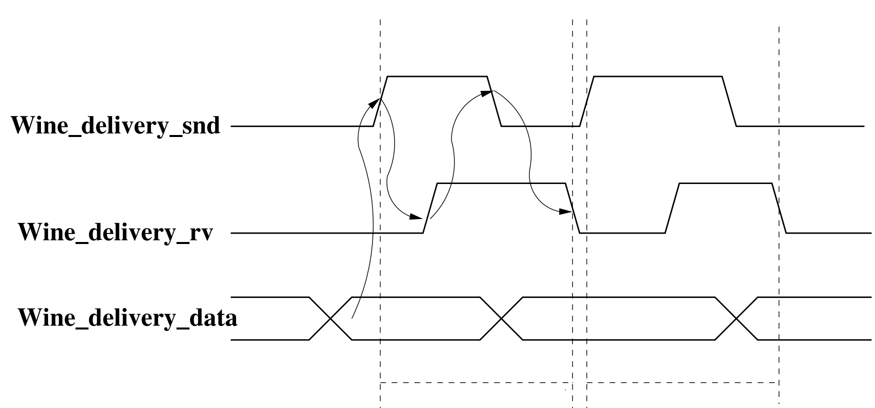

Figure 2.3. 4-phase bundled-data.

vassign(s, v, l, u), and takes a signal s, and changes it to a value v at some point between l and u time units without caring about what value s is originally. In other words, *vassign* allows *vacuous* assignments, where signal s does not change values. Both *assign* and *vassign* suspend operation until s equals to v.

Consider the following VHDL code, which is the same *shop* example used earlier, only this one is at the handshaking-level, and the previous one is at the channel-level.

```
-- shop.vhd
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.handshake.all;
entity shop is
  port(
    -- wine_delivery :  inout channel := init_channel;
    wine_delivery_rv :  inout std_logic;
    wine_delivery_snd :  in std_logic;
    wine_delivery_data :in std_logic_vector(2 downto 0);
    -- wine_selling :  inout channel := init_channel;
    wine_selling_rv :  in std_logic;
    wine_selling_snd :  inout std_logic;
    wine_selling_data :  out std_logic_vector(2 downto 0)
  );
end shop;
architecture behavior of shop is
  signal shelf:std_logic_vector(2 downto 0);
  signal shelf1:std_logic_vector(2 downto 0);
begin
  shop:process
  begin
    -- receive(wine_delivery,shelf);
    guard(wine_delivery_snd,'1');
    shelf <= wine_delivery_data;
    wait for 5 ns;
    assign(wine_delivery_rv,'1',1,3);
    guard(wine_delivery_snd,'0');
    assign(wine_delivery_rv,'0',1,3);
    shelf1 <= shelf + 1;
    wait for 5 ns;
    -- send(wine_selling,shelf1);
    wine_selling_data <= shelf1;
```

```
      wait for 5 ns;
      assign(wine_selling_snd,'1',1,3);
      guard(wine_selling_rv,'1');
      assign(wine_selling_snd,'0',1,3);
      guard(wine_selling_rv,'0');
    end process shop;
  end behavior;
```

This example uses a 4-phase bundled-data protocol. The first 4-phase protocol is translated from the receive statement. The shop waits for data to be sent. Therefore, this channel is *passive* with respect to the shop. It starts with a *guard* statement and waits for the signal *wine_delivery_data* to take the value *1*, and then *wine_delivery_data* delivers the data to the internal signal *shelf*. After waiting for an amount of time to make sure the delivered data have been received, it sets *wine_delivery_rv* to be high at some point between 1ns to 3ns; the sender sets the *wine_delivery_snd* to be low at this point to invalidate the data. The receiver responds by setting *wine_delivery_rv* low at some point between 1ns to 3ns. At this point, one whole handshake has been completed. Before sending the data out, constant 1 has been added to the *shelf*, and assigns this to *shelf1*. After waiting for 5 ns, *shelf1* is copied to *wine_selling_data*. The sender is *active*, that is it initiates the data transfer through the channel. It starts with an *assign* statement, which takes signal *wine_selling_snd* and changes it to value *1* in between 1*ns* and 3*ns*. After that, it waits for the signal *wine_selling_rv* to take value 1, then sets signal *wine_selling_snd* back to 0 again in between 1*ns* and 3*ns*. Finally, it waits for *wine_selling_rv* to change to 0 completing the second handshaking cycle.

2.3 Structual-Level Modeling

After getting the handshaking-level VHDL code, we can synthesize the control circuit using the ATACS tool [18]. ATACS supports different signal level specification formats, which includes handshake-level VHDL that can be simulated by commercial VHDL simulators. The following example shows how the structural-level VHDL looks after synthesized by ATACS.

First of all, we introduce generalized-C element (gC) circuits. A C-element has two inputs. If both of the inputs are the same, then the output is the same as the input; otherwise, the output stays the same. A generalized C-element is a gate that is composed of an arbitrary sum-of-products function to pull-up the signal as well as an arbitrary pulldown network [4]. When the condition by the pull-up network is satisfied, the output of the gC is set. If the condition of the pull-down network is satisfied, then the output is reset. Otherwise, the output holds its state. When both the pull-up and pull-down networks are satisfied, the synthesis tool would make sure that they are not happening at the same time, so a short circuit would not result.

ATACS generated VHDL files include many different *gC* circuits. The following VHDL code, *n1p1.vhd* is a simple example to describe how a *gC* circuit works. From the entity name, *n1p1*, it is not hard to know that the circuit is made of one nMOS and one pMOS transistor. There are two inputs *a1n* and *a1p*, two inouts *o* and *obar*, where *obar* is the inverse of *o*. In the architecture, there is an internal signal *o_i*, which is initialized to 0. From the main body of the architecture, we know that if input *a1n* is 1, then *o_i* becomes 1, else if *a1p* is 0, then *o_i* becomes 0, other than those two conditions, *o_i* keeps the current value. At the end, *o_i* assigns its value to output *o*, and the inverse value of *o_i* is assigned to output *obar*. Figure 2.4 shows the transistor schematic of the *n1p1 gC* circuit.

```
--n1p1.vhd
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
entity n1p1 is
    Port ( a1n :  in std_logic;
           a1p :  in std_logic;
           o :  inout std_logic :='0';
           obar :  inout std_logic :='1');
end n1p1;
architecture Behavioral of n1p1 is
signal o_i:  std_logic :='0';
begin
  o_i<='1' after 0 ns when (a1n ='1') else
        '0' after 0 ns when (a1p ='0') else
        o_i;
  o<=o_i;
  obar <= not o_i after 0 ns;
end Behavioral;
```

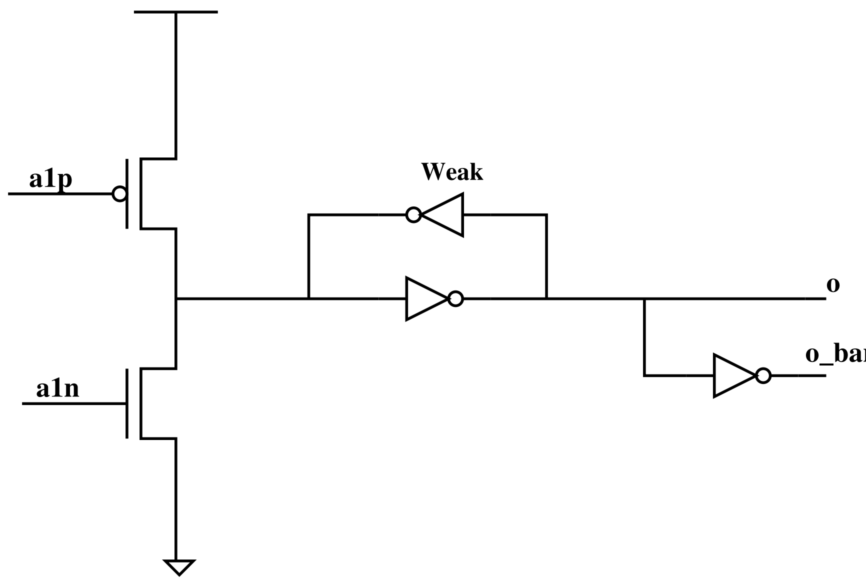

Figure 2.4. The n1p1 circuit.

The following shows a completely structural-level VHDL code generated by ATACS for THE_SHOP_shop_ctrl_instS.vhd. From the entity, we can see that the ports are the same as the datapath, which makes it easy to connect them. In the architecture, there are internal signals declarations, and components declarations. All the gC circuits are similar to the ones we just described earlier. The port map basically connects all the components together.

```
library ieee;
use ieee.std_logic_1164.all;
entity THE_SHOP_shop_ctrl_inst is
  port(   wine_delivery_rv :  inout std_logic;
        wine_delivery_snd_i :  in std_logic;
            wine_selling_rv :  in std_logic;
         wine_selling_snd_i :  inout std_logic
        );
end THE_SHOP_shop_ctrl_inst;
architecture structure of THE_SHOP_shop_ctrl_inst is
  -- internal signals declarations
  signal CSC0:  std_logic;
  -- input complements
  signal wine_delivery_snd_i_bar:  std_logic;
  signal wine_selling_rv_bar:  std_logic;
  -- output complements
  signal CSC0_bar:  std_logic;
  signal wine_delivery_rv_bar:  std_logic;
  signal wine_selling_snd_i_bar:  std_logic;
```

```
  component n1p1
  port(a1n :  in std_logic;
       a1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
  component n2p2
  port(a1n :  in std_logic;
       b1n :  in std_logic;
       a1p :  in std_logic;
       b1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
  component n3p2
  port(a1n :  in std_logic;
       b1n :  in std_logic;
       c1n :  in std_logic;
       a1p :  in std_logic;
       b1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
begin
  i_0 :  n3p2
    port map(a1n=>wine_delivery_snd_i, b1n=>wine_selling_rv_bar,
             c1n=>wine_selling_snd_i_bar,
             a1p=>wine_delivery_snd_i, b1p=>CSC0_bar,
             o=>wine_delivery_rv, obar=>wine_delivery_rv_bar);
  i_1 :  n2p2
    port map(a1n=>wine_delivery_rv_bar, b1n=>CSC0,
             a1p=>wine_selling_rv_bar, b1p=>CSC0,
             o=>wine_selling_snd_i, obar=>wine_selling_snd_i_bar);
  i_2 :  n1p1
    port map(a1n=>wine_delivery_rv,
             a1p=>wine_selling_snd_i_bar,
             o=>CSC0, obar=>CSC0_bar);
  -- complements of inputs
  wine_delivery_snd_i_bar <= not wine_delivery_snd_i after
0 ns;
  wine_selling_rv_bar <= not wine_selling_rv after 0 ns;
end structure;
```

After getting this structural-level VHDL code, it can be simulated by using any commercial tools. A goal of this thesis is to determine whether synchronous tools can be used to synthesize this type of circuit. The simulation result of the controller is shown in Figure 2.5.

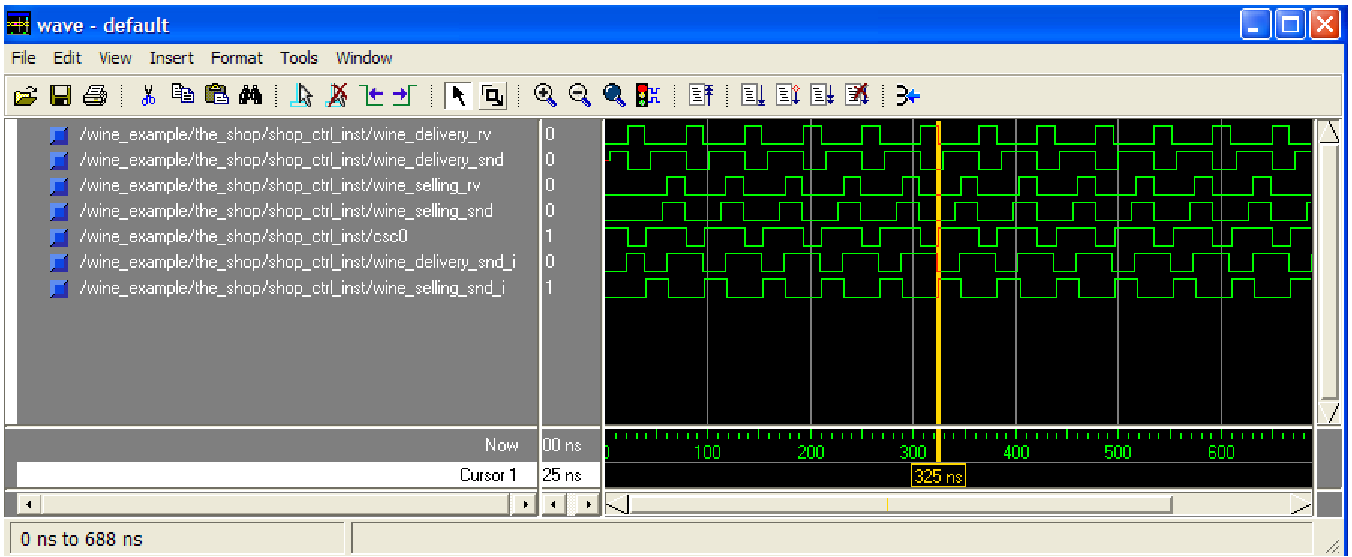

Figure 2.5. Shop controller simulation.

CHAPTER 3

DESIGN FLOW FOR SYNTHESIZING ASYNCHRONOUS CIRCUITS

This chapter presents the details of how to automate the translation procedures from one modeling method to another. This chapter also describes how to use synchronous synthesis tools to synthesize asynchronous circuit designs.

Simulation and synthesis are always very important procedures in circuit design. There are simulation tools that can be used for both synchronous and asynchronous circuit designs. There is also a rich set of synthesis tools developed for synchronous circuits, but only academic tools for asynchronous circuit synthesis. Many researchers have tried to create new synthesis tools for asynchronous circuit design, but the tools can be difficult to use by non-experts. Therefore, this thesis chooses to apply synchronous synthesis tools to asynchronous circuit design.

The tools developed for this thesis manipulate VHDL code using three Perl scripts. The first script transfers channel-level VHDL code to handshaking-level VHDL code, and the second script separates the asynchronous controller from the self-timed datapath components. The third script calculates the bundling constraints and creates the VHDL files to produce these delays. Then, ATACS is used to synthesize the asynchronous controller. ATACS, developed by our research group, can synthesize all the constructs from the channel and handshaking packages. In this case, it takes the handshaking-level VHDL code and synthesizes it into structural-level VHDL code for later synthesis use. Finally Xilinx's Webpack is used to map the ATACS structural-level VHDL code and self-timed datapath components into an FPGA implementation. Xilinx launched the world's first commercial FPGA in 1985 [17]. In this thesis, we used Xilinx's XC4000 series

FPGA as the target technology for this thesis.

Figure 1.1 shows the design flow for this thesis. It includes all the steps to utilize synchronous tools for designing asynchronous circuits . The ideas of each modeling method of the design flow have been given in the previous chapter. This chapter describes how this design flows from one modeling method to another. The connections between each of the modeling methods and how the tools work in the design flow are also described. The design flow starts from a channel-level VHDL code that includes *send* and *receive* statements. There are no synchronous commercial tools that can synthesize this type of VHDL code, so in order to solve this problem, the *Ch2Hs* script transforms the channel-level model into a handshaking-level model. Before putting this channel-level VHDL code into the *Ch2Hs*, we simulate it to make sure it works correctly. The handshaking-level VHDL uses the handshake package that includes *guard* and *assign* statements. At this point, the *DpSyn* script generates the handshaking-level VHDL models for both simulation and synthesis. For the simulation model, it verifies the functionality after the first transformation to avoid unnecessary problems in the later steps. For the synthesis model, it separates the whole circuit into an asynchronous controller and datapath components for synthesis. For the datapath components, there are a top-level file of the circuit in structural-level VHDL code and also two versions of the datapath components. One is for synthesizing the timing report, which is named *dp_reference-number_md*. The others are complete datapath components, including delay elements for later synthesis, and they are named *dp_reference-number*. After generating all of the files, the maximum delay files that are named *dp_reference-number_md* are inserted into WebPack and synthesize them to get the maximum delay timing report. According to this timing report, our tool creates a new timing report, which includes the lower bound and upper bound on each of the functional units and also the unit delay for each buffer. Next, the *CtrlSyn* script parses the timing report to get the maximum delay time, calculates the total bundling constraints, and uses these newly calculated bundling constraints to replace the lower bound and upper bound time of the handshaking statements of the controller. Meanwhile, it also generates the bundling constraint files for WebPack and updates the structural-level delay component files for later synthesis use. At this point, ATACS synthesizes the controller to get the structural-level VHDL code in order to combine with the datapath later. Now, the design flow has reached the point where all of the files are in structural-level VHDL, including the controller, all the datapath components, the delay elements, and the top-level file. After this, we combine all these files together, synthesize them using WebPack, and then implement the design on an FPGA. Finally, we add self-testing circuitry, which includes LFSRs to generate random values and signature analyzers in order to test the whole design. In this design flow, most of the parts have been automated except importing the files into WebPack, which is done manually. Also, picking up the maximum delay time from the delay timing report is done manually because it involves two different operating systems. The scripts running WebPack are under Windows, but ATACS is run under Linux.

This chapter is organized as follows. Section 3.1 describes the channel-level modeling to handshaking-level modeling transformation. Section 3.2 presents the datapath synthesis and delay calculation methodology. Section 3.3 describes the controller synthesis methodology. Section 3.4 describes the validation of the work using the *wineshop* example. Section 3.5 presents the testing procedures and testing results of the whole design, and Section 3.6 presents the conclusion, which summarizes the whole chapter and demonstrates the possibility of using synchronous tools for asynchronous design.

3.1 Channel to Handshaking Transformation

This section introduces our method to translate the channel-level model to the handshaking-level model. This used to be done manually, which is time-consuming, but it is much faster and more accurate this way. The fundamental idea of this script is to translate each channel port and each *send* and *receive* statement into handshaking signals, and also to add the additional internal signals. Compare this thesis work to Erik Peskin's work [23], in which he explored multiple signals translation from channel-level to handshaking-level and tried to find the best handshaking signal. However, this thesis work explores single handshaking signal and tries to automate all the processes after that.

Ch2Hs is separated into three stages, which are the preprocess stage, the extract-information stage, and the translation stage. In the preprocess stage, there are four processes. Every process has the same goal, which is to reformat the VHDL source code to one statement per line format, so the VHDL code can be parsed easily. The first process is *preprocess comment*. The reason to have this process is to remove all the unnecessary lines, to make the whole process easier to automate. The way to do this is to create a lookup table and insert comment-location marks, *$number*, so later on the comments can be restored to their original locations. This way, the method transforms only the parts that need to be transformed and always leaves the other parts the way they were originally. For instance, in the following code, the tool has replaced the comment lines by using "$number", and then uses these indices in the lookup table to store the comments.

```
$0
$1
$2
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
  .....
The lookup table stores something like following:
$0   ``-- Comment 1''
$1   ``-- Comment 2''
  .....
```

The second process is for entity declarations. The purpose is also to reformat the entity declarations that need to have one port declaration per line. The third process reformats component declarations to have one port declaration per line. The fourth process reformats port map declarations to have one connection per line. One of the examples of this stage is given as the following:

```
THE_SHOP: shop port map(
    wine_delivery => WineryShop, wine_selling => ShopPatron);
```

After going through the preprocess port map declaration step, the port map becomes:

```
THE_SHOP: shop
port map(
  wine_delivery => WineryShop,
  wine_selling => ShopPatron
);
```

Overall, the preprocess stage reformats all of the code to make one statement per line, which is very easy for later automation using regular expressions. It is very useful to reformat before going to the further steps.

The second stage of the first script is extract information stage. The purpose of this stage is to collect all the information that the tool needs in order to translate from channel-level VHDL to handshaking-level VHDL. By going through several channel-level VHDL examples, we made observations from the entity port declaration in the entity, which generally looks like *ch* : *inout channel* := *init_channel*, *ch* stands for channel. It must be replaced with *ch_rv*, which represents the handshaking receive signal; *ch_snd*, which represents the handshaking send signal; and *ch_data*, which represents the data signals. This stage can be separated into the following steps.

The first step is extracting the information to expand the channel-port that is declared in the entity-declaration. In general, each channel port needs to be translated into three handshaking signals, *ch_rv* and *ch_snd*, and the data wires *ch_data*. From the entity declaration, our tool can recognize the name of the channel and the data, substitute into the handshaking signals to replace the channel *ch*.

However, additional information such as port direction and data bus width are not available at this point, the tool needs to know the directions of the signals and the width of the data bus. Therefore, the second step is to extract the information for the directions of the handshaking signals. This is accomplished by analyzing the VHDL code and extracting the information from the *send* and *receive* statements in the architecture block.

For the *send* statement *send(ch, data)*, we assume it to be active. The direction of *ch_snd* in this case would be *inout*, the direction of *ch_rv* is *in*, and the direction of *ch_data* is *out*. For the *receive* statement, *receive(ch, data)*, since it is assumed to be passive, the direction of *ch_snd* in this case is *in*, the direction of *ch_rv* is *inout*, and the direction of *ch_data* is *in*. Therefore, in order to know the direction of the handshaking signals *ch_snd* and *ch_rv*, our method has to know which statement they are in, *send* or *receive*.

In addition, the width of *ch_data* is determined by extracting information from the signal-declaration within the architecture block. The signal declarations are located at the beginning of the architecture, which includes the names and the width of the internal signals. In the channel-level VHDL code, channels use internal signals to *send* or *receive* data. So the data width has to be the same as the internal signal data width. By looking at the internal signal declaration, our method can get the information for the data width.

Therefore, for a *send* statement, *send(ch, data)* presents the following information:

1. *ch_rv* is an *input* handshaking signal, which is declared as *in*.
2. *ch_snd* is an *output* handshaking signal, which is declared as *inout*.
3. The direction of *ch_data* is *out*.
4. The width of *ch_data* is the same as *data*, which is extracted from the width of *data*'s signal declaration.

For a *receive* statement, *receive(ch, data)*, presents the following information:

1. *ch_rv* is an *output* handshaking signal, which is declared as *inout*.
2. *ch_snd* is an *input* handshaking signal, which is declared as *in*.
3. The direction of *ch_data* is *in*.
4. The width of *ch_data* is the same as *data*'s, thus, the *data*'s width can be extracted from its signal declaration.

From the above observations, directions for handshaking signals can be obtained from items 1 and 2, the direction and the width of the data from items 3 and 4, then fill in the missing part of the port declaration as described earlier.

However, there is an inconsistency issue from this part. For instance, if there are *send* and *receive* statements on the same channel in the same process, then it would cause confusion for the channel to determine which direction it should be. At this point, an inconsistency error is reported. The other thing is either there are *send* statements continuously or *receive* statements continuously in the input files, but with different widths of the data on them, then it would cause confusion for the channel to choose which data size to be used. At this point, an inconsistency error is reported again.

A local lookup table is created for each *send* and *receive* channel to store all the information, and it is used to replace them with corresponding handshaking statements.

It is better to construct a local lookup table to store all this information for the translation stage and also for the global lookup table usage. The reason to call it local is that it can extract all the information just by looking at this one single file instead of going through all the files. The next paragraph shows how to extract information for port-maps, which needs to be stored in a global lookup table. The following record shows what kind of information a local lookup table should include.

```
%channels{channel-name}
    =>%info{
          send-direction,
          rv-direction,
          data-direction,
          data-type
          }
```

The contents of the lookup table are shown in Table 3.1.

Table 3.1. Local lookup table.

	send(ch,data)	receive(ch,data)
ch_rv	in	inout
ch_snd	inout	in
ch_data	out	in
ch_data width	data width	data width

Fourth, *Ch2Hs* constructs a global lookup table for port-map statements, which is shown below in the top-level VHDL code from the shop example. The channel-level port-map statements are transformed to handshaking-level port-map statements. Since our tool uses an identical naming style as we expand the channel-port declaration, this part is very straightforward. The information needing to be stored in the lookup tables are component names, and all the interface signals. The component names are used for constructing the port map names and the interface signals are used to fill in the details of the port maps. Each port map entry should have a signal name that connects to another signal name. If the interface signals are channels instead of std_logic type, then the method has to use the transformed handshaking signals, which is extracted from the previous steps. Everything on the left side of the connections is interface signals from the entity ports, and everything on the right hand side is internal signals.

The third stage of *Ch2Hs* is to translate all the information. In this stage, it uses the information we have extracted from the earlier stage, and translate it to complete handshaking-level models. In this stage, the modeling method basically handles four transformations, which are channel-port transforms to handshaking-port for both entity declarations and component declarations, the channel type signals declarations in the architecture block, the send and receive statements transformed to 4-phase bundled-data handshaking statements, and transformations from channel to handshaking for port maps.

First of all, this section presents how to translate the entity and component declarations from channel-level VHDL to handshaking-level VHDL. These two declarations fit into the same category because they have the same transformation for the ports. The ports of the entity are the same as the ports in the components. From the comment lines in the *shop* entity, each channel declaration is transformed into a pair of handshaking declarations and a data declaration. For instance,

```
entity shop is
  port(
    -- wine_delivery :  inout channel := init_channel;
    wine_delivery_rv:  inout std_logic;
    wine_delivery_snd :  in std_logic;
    wine_delivery_data :  in std_logic_vector(2 downto 0);
```

```
    -- wine_selling : inout channel := init_channel;
    wine_selling_rv: in std_logic;
    wine_selling_snd : inout std_logic;
    wine_selling_data : out std_logic_vector(2 downto 0)
  );
end shop;
```

```
component shop
  port (
    -- wine_delivery : inout channel;
    wine_delivery_rv: inout std_logic;
    wine_delivery_snd : in std_logic;
    wine_delivery_data : in std_logic_vector(2 downto 0);
    -- wine_selling : inout channel;
    wine_selling_rv: in std_logic;
    wine_selling_snd : inout std_logic;
    wine_selling_data : out std_logic_vector(2 downto 0)
  );
  end component;
```

The component transformation is the same as the entity transformation. Components appear in the top-level VHDL file, and the top-level VHDL file connects all of the different component files in order to simulate and synthesize, so it has to include the entity of the component file, *shop* in this case.

Second, this section presents the transformation for internal channel type signal declarations in the architecture block. Most of these happen in the top-level VHDL file, such as in the following example. The comment line is an internal channel type signal declaration, and *Ch2Hs* transforms it into the three declarations below.

```
--signal WineryShop:channel:=init_channel;
  WineryShop_rv: std_logic;
  WineryShop_snd : std_logic;
  WineryShop_data : std_logic_vector(2 downto 0);
```

Third is the *send* and *receive* statement transformations. This is one of the most important transformations for the script, which transforms the channel-level statements into 4-phase bundled-data handshaking statements. For a *receive* statement, *receive(ch, data)*, the steps are wait until *ch_snd* is set 1, assign *ch_data* to *data*, wait for bundling constraint of the latch, assign *ch_rv* to 1 after some delay, wait until *ch_snd* is set to 0, and assign *ch_rv* to 0 after some delay. For instance, the *receive* statement transformation looks like the following:

```
-- receive(wine_delivery,shelf);
guard(wine_delivery_snd,'1');
shelf <= wine_delivery_data;
wait for 5ns;
assign(wine_delivery_rv,'1',1,3);
guard(wine_delivery_snd,'0');
assign(wine_delivery_rv,'1',1,3);
```

The modeling method comments out the *receive* statement and uses all the information from the lookup table to transform it into 4-phase bundled-data handshaking statements. Since it is a *receive* statement, which is passive, the handshaking statements start with a *guard* statement. The method also adds the delay time after each of the assign(<=) statements.

For the *send* statement, *send(ch, data)*, the steps are: assign the data to *ch_data*, wait for bundling constraint that matches the delay to create the data, assign *ch_snd* to 1 after some delay, then wait until *ch_rv* is set to 1. Next, reset the handshaking by assigning *ch_snd* to 0 after some delay, then wait until *ch_rv* is set to 0. For instance, the *send* statement transformation looks like the following:

```
shelf1 <= shelf + 1;
wait for 5 ns;
-- send(wine_selling,shelf1); -- comment send
wine_selling_data <= shelf1;
wait for 5ns;
assign(wine_selling_snd,'1',1,3);
guard(wine_selling_rv,'1');
assign(wine_selling_snd,'0',1,3);
guard(wine_selling_rv,'0');
```

Similar to the *receive* statement, the modeling method comments out the *send* statement and uses the information from the lookup table to transform it into a 4-phase bundled-data handshaking statement. Since it is a *send* statement, which is active, the handshaking statements start with an *assign* statement. Again, the method adds the delay time after the assign(<=)statement.

The last translation part is the port map statements, which only appear in the top-level VHDL code. Port maps basically map everything together for the circuit, so they include all of the connection statements. The left hand sides of the statements are all the interface signals, and the right hand sides of the statements are all internal signals. For example, the port map for the shop using channel-level VHDL code, is as shown below,

```
THE_SHOP:shop
    port map(wine_delivery => WineryShop,
             wine_selling => ShopPatron);
```

After *Ch2Hs*, it becomes the following handshaking-level VHDL code:

```
THE_SHOP : shop
  port map(
    wine_delivery_rv=>WineryShop_rv;
    wine_delivery_snd =>WineryShop_snd;
    wine_delivery_data =>WineryShop_data;
    wine_selling_rv=>ShopPatron_rv;
    wine_selling_snd =>ShopPatron_snd;
    wine_selling_data =>ShopPatron_data
  );
```

Overall, the above four transformation parts are the major tasks of *Ch2Hs*. One caution for using *Ch2Hs* is that when the transformation involves channel-level port maps, our tool needs to transform all the required VHDL sources in order to know all the information for the channel-level port map. For instance, the *wine_example2.vhd* file as shown below, includes port maps for *winery*, *shop*, and *patron*.

```
-- wine_example2.vhd
library ieee;
use ieee.std_logic_1164.all;
use work.nondeterminism.all;
use work.channel.all;
entity wine_example is
end wine_example;
architecture structure of wine_example is
  component winery
    port(
    wine_shipping:inout channel
    );
  end component;
  component shop
    port(
    wine_delivery:inout channel;
    wine_selling:inout channel
    );
  end component;
  component patron
    port(
    wine_buying:inout channel
```

```
    );
  end component;
  signal WineryShop:channel:=init_channel;
  signal ShopPatron:channel:=init_channel;
begin
  THE_WINERY:winery
    port map(wine_shipping => WineryShop);
  THE_SHOP:shop
    port map(wine_delivery => WineryShop,
             wine_selling => ShopPatron);
  THE_PATRON:patron
    port map(wine_buying => ShopPatron);
end structure;
```

If only this file is given, the tool has no clue how to transform the component declarations and port map declarations, because there is not enough information. Therefore, the method needs to retrieve the required information from other VHDL source code. In this case, it needs to transform the *winery*, *shop*, and *patron* entity declarations first. After getting all the information, it then starts transforming the *wine_example2.vhd* code. The pseudo code for the first script is as shown below:

```
Script       :  Ch2Hs.pl
Dependency   :  Ch2Hs_lib.pm
Description  :  translate channel-level VHDL files to their
corresponding handshake-level VHDL files
Inputs       :  channel-level VHDL files
Outputs      :  all translated handshake-level VHDL files are
put in the ``./hs'' directory
Notes        :  given VHDL files must be supplied in the order
based on their dependencies
Algorithm {
[0]  clear global look-up-table;
for-each given VHDL file {
[1]  clear local look-up-table;
[2]  read the current VHDL file into buffer;
[3]  pre-process the buffer which includes removing comments
temporarilly and reformatting the VHDL file;
[4]  use send/receive statements to extract each channel's
corresponding (direction, data-bus width) information;
[5]  store data extracted in [4] into the local look-up-table;
[6]  use information to translate channel-level entity-declarations
to it corresponding handshake-level entity-declarations and
store the results into the global look-up-table;
[7]  use both the local and global look-up-table to interpret
the port-map interconnections' signal types and append
the results into a local look-up-table;
[8]  use local look-up-table to translate send/receive statements;
[9]  use global look-up-table to translate component-declaration;
```

```
[10] use local look-up-table to translate signal-declaration;
[11] use global look-up-table to translate port-map statements;
[12] add the original comments back to the translated handshake-
level VHDL result
and store it into file;
  }
}
```

3.2 Datapath Synthesis and Delay Calculation

After transforming channel-level VHDL into handshaking-level VHDL, we now present the datapath synthesis method, which transforms the handshaking-level VHDL into synthesable VHDL. In order to make the tool easier to use, the method transforms the handshaking model into two forms of VHDL models, one for controller synthesis and the other one for datapath synthesis.

The datapath architecture used here is a 4-phase bundled-data handshaking model with both passive and active modes. It includes the delay components, latch, combinational logic, etc. This section describes more detail about the datapath architecture used for receives and sends.

Figure 3.1 illustrates a *receive* statement, *receive(ch, data)*, from the channel-level specification. The datapath includes a latch, a delay element, and also the ATACS synthesized controller. A latch is introduced whenever there is a receive

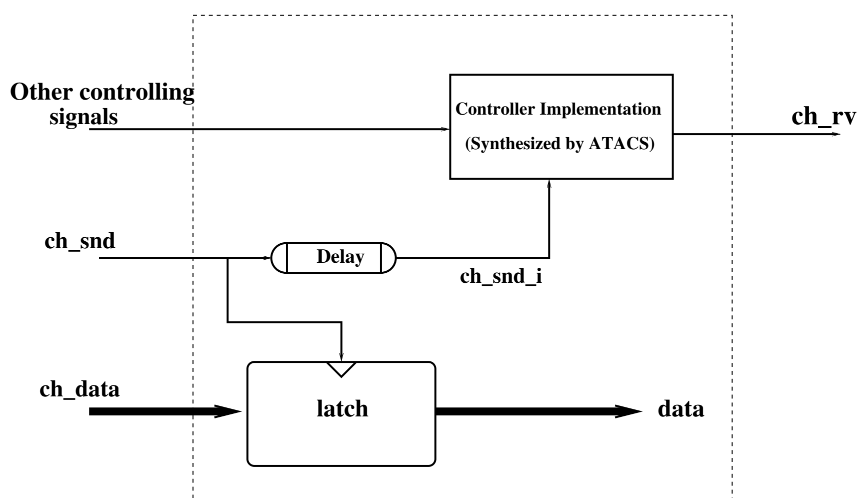

Figure 3.1. Datapath for a receive.

statement; the delay element mimics the matching delay time for the latch. What this circuit does is receive data from its *ch_data* channel and store them into the internal latch *data*. This action starts when *ch_data* is ready and uses the *ch_snd* as a trigger. The environment initiates the channel communication by setting *ch_snd* to be high, which causes the internal latch *data* to latch what is presented in its input, *ch_data*, and also causes the delay element to receive this action and delays it. This delay element matches how much time it takes for the latching action. After the delay time, we assume that the data presented on *ch_data* are safely caught by the latch. Next, when both other controlling signals and the *ch_snd_i* are ready, a completion signal *ch_rv* is generated to notify the next component to use the latch data for computation. Finally, it resets the handshaking signals.

The following is a receive statement from the *shop* example.

```
-- receive(wine_delivery,shelf);
  guard(wine_delivery_snd,'1');
  shelf <= wine_delivery_data;
  wait for 5 ns;
  assign(wine_delivery_rv,'1',1,3);
  guard(wine_delivery_snd,'0');
  assign(wine_delivery_rv,'0',1,3);
```

What this code does is, first, wait until *wine_delivery_snd* goes to high (rising edge), which means the source data are ready. Second, it receives the data from *wine_delivery_data* and stores the result into *shelf*, which means the register *shelf* uses the signal *wine_delivery_snd*'s rising edge to latch data. Third, the circuit waits for 5 ns, which means step 2 required approximately 5 ns delay time to complete the task. Fourth, it notifies the environment by setting *wine_delivery_rv* to high (rising edge), which means after 5 ns of delay time, our method assumes step 2 is completed. Fifth, it waits until *wine_delivery_snd* goes low (falling edge), which means the source data provider says that it notices the data is no longer useful, so it can go ahead to provide new data or do something else. Sixth, the circuit notifies the environment with *wine_delivery_rv* sets to low (falling edge), which means it tells the environment that the datapath component is also ready for the next task.

The datapath for a *send* is shown in Figure 3.2. This figure illustrates a *send* statement, *send(ch, data)*, from the channel-level specification. The datapath

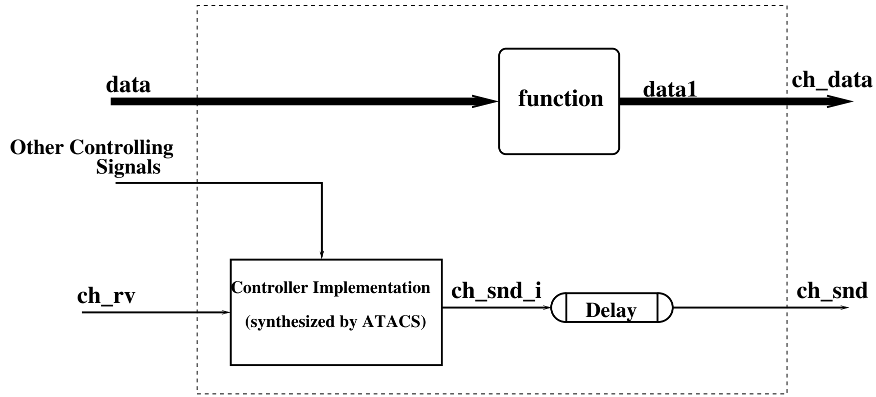

Figure 3.2. Datapath for a send.

includes a functional block, a delay element, and also the ATACS synthesized controller. The functional block does all the calculations. The delay element mimics the matching delay time for the functional block. What this circuit does is to compute the data using the functional block and then send it out through the *ch_data* channel. *Other controlling signals* represents the synthesized precondition setup for initiating a send data action. *Ch_rv* is the trigger to generate the acknowledgment signal *ch_snd*, and *ch_data* is the computation result of this self-timed datapath component. The action starts when all the *data* inputs are ready before the signal *ch_rv* can be triggered. When *ch_rv* is triggered and also the *other controlling signals* are ready, the functional block starts to compute. This is a worst case assumption since the functional block is a transparent logic block. There is no latch or memory element to stop the data which have been passing through the functional block. Therefore, the computation has already begun before the *ch_rv* has been triggered. If we use the delay element to match the worst case evaluation time for the functional block. The total delay time for the signal *ch_snd* to get high is a little bit greater than what it needs to be theoretically. At the end, it resets the handshaking signals and gets ready for the next action.

The following is a *send* statement from the *shop* example.

```
shelf1 <= shelf + 1;
wait for 5 ns;
-- send(wine_selling,shelf);
wine_selling_data <= shelf1;
```

```
wait for 5 ns;
assign(wine_selling_snd,'1',1,3);
guard(wine_selling_rv,'1');
assign(wine_selling_snd,'0',1,3);
guard(wine_selling_rv,'0');
```

What this code does is first calculate *shelf+1* and store the result to *shelf1*, which means to perform the datapath component operation *shelf+1*. Second, it waits for 5 ns, which means step 1 requires approximately 5 ns delay time to complete the task. Third, the circuit sends *shelf1* to the external port *wine_selling_data*, which means to perform the data transfer to send the *shelf+1*'s result to the output port. Fourth, it waits for 5 ns again, which means the assignment requires approximately 5 ns delay time to complete the task. Fifth, the circuit notifies the environment by setting *wine_selling_snd* to high (rising edge), which means after a total of 10 ns delay time, we assume the datapath operation has all completed, and using *wine_selling_snd*'s rising edge to notify the environment that the circuit has data ready to serve. Sixth, it waits until *wine_selling_rv* goes to high (rising edge), which means the data are consumed by the environment. Seventh, it notifies the environment by setting *wine_selling_snd* to low (falling edge), which means telling the environment that the handshaking signals are reset for further communication. Eighth, it waits until *wine_selling_rv* goes to low (falling edge), which means the environment confirms it has also reset all the handshaking signals and is ready for the next transaction.

DpSyn starts with information gathering. The purpose of this script is to transform the handshaking model into the datapath architecture, which includes separating the datapath and controller and generating new components. So the first task uses the same idea from *Ch2Hs*, which constructs lookup tables to store all of the information for later use.

The second major task of this method is controller generation. First, it creates the files *filename_dp0_atacs_t.vhd*, *filename_dp1_atacs_t.vhd* and *filename_ctrl_atacs_t.vhd*, which are templates that need to be updated by the next method *CtrlSyn* and used for ATACS synthesis. It also creates the file, *filename_wp.vhd*, which connects all the other components together. The *shop_wp.vhd* file is as shown below:

```
-- shop_wp.vhd
library ieee;
use ieee.std_logic_1164.all;
entity shop is
--i/p - o/p ports declaration
  port(
    --port decl
       -- wine_delivery :  inout channel := init_channel;
       wine_delivery_data :  in std_logic_vector(2 downto 0);
       wine_delivery_rv :  inout std_logic;
       wine_delivery_snd :  in std_logic;
    --ch1
       -- wine_selling :  inout channel := init_channel;
       wine_selling_data :  out std_logic_vector(2 downto 0);
       wine_selling_rv :  in std_logic;
       wine_selling_snd :  inout std_logic
  );
end shop;
architecture arch_wp of shop is
  --i/p receive side datapath (dp0) component
  component shop_dp0
  port(
  wine_delivery_snd:in std_logic;
  wine_delivery_snd_i:inout std_logic;
  shelf:out std_logic_vector(2 downto 0);
  wine_delivery_data:in std_logic_vector(2 downto 0)
  );
  end component;
  --o/p send side datapath (dp1) component
  component shop_dp1
  port(
  shelf:in std_logic_vector(2 downto 0);
  wine_selling_data:out std_logic_vector(2 downto 0);
  wine_selling_snd:inout std_logic;
  wine_selling_snd_i:in std_logic
  );
  end component;
  --ATACS controller component
  component shop_ctrl
  port(
  wine_selling_rv:in std_logic;
  wine_delivery_rv:inout std_logic;
  wine_delivery_snd_i:in std_logic;
  wine_selling_snd_i:inout std_logic
  );
  end component;
  --signals for dp0-dp1-controller
  signal wine_delivery_snd_i:std_logic;
  signal shelf:std_logic_vector(2 downto 0);
  signal wine_selling_snd_i:std_logic;
  begin
  --i/p receive side datapath (dp0) port map
    shop_dp0_inst:  shop_dp0
```

```
port map (
  wine_delivery_snd => wine_delivery_snd,
  wine_delivery_snd_i => wine_delivery_snd_i,
  shelf => shelf,
  wine_delivery_data => wine_delivery_data
  );
--o/p send side datapath (dp1) port map
  shop_dp1_inst:  shop_dp1
port map (
  shelf => shelf,
  wine_selling_data => wine_selling_data,
  wine_selling_snd => wine_selling_snd,
  wine_selling_snd_i => wine_selling_snd_i
  );
--ATACS controller port map
  shop_ctrl_inst:  shop_ctrl
port map (
  wine_selling_rv => wine_selling_rv,
  wine_delivery_rv => wine_delivery_rv,
  wine_delivery_snd_i => wine_delivery_snd_i,
  wine_selling_snd_i => wine_selling_snd_i
  );
end arch_wp;
```

This output file shows how to connect all the datapath components and the synthesized controller. *Dp_0* is the latch from the *receive* channels, *dp_1* is the functional blocks, and the *shop* is the controller implementation. Finally, it uses port maps to connect all of these components.

There are three component template files, *shop_ctrl_atacs_t*, *shop_dp0_atacs_t*, and *shop_dp1_atacs_t*. This following process is from file *shop_ctrl_atacs_t.vhd*, which shows how the controller process looks after separating from the datapath with only 4-phase handshaking statements in it. It also includes the templates for bundling constraints. The detail of the bundling constraint calculation is shown in the next section. Note all of the newly created internal signals have names ending with an "*_i*".

```
--controller hs process
shop_ctrl_hs:  process
begin
  -- receive(wine_delivery,shelf);
  guard(wine_delivery_snd_i,'1');
  assign(wine_delivery_rv,'1',ctrl_slb,ctrl_sub);
  guard(wine_delivery_snd_i,'0');
  assign(wine_delivery_rv,'0',ctrl_rlb,ctrl_rub);
  -- send(wine_selling,shelf1);
  assign(wine_selling_snd_i,'1',ctrl_slb,ctrl_sub);
```

```
guard(wine_selling_rv,'1');
assign(wine_selling_snd_i,'0',ctrl_rlb,ctrl_rub);
guard(wine_selling_rv,'0');
end process shop_ctrl_hs;
```

The next process shown in the following is from file *shop_dp0_atacs_t*, which shows how datapath_0 looks for a receive statement after separating from the other files and how to handle the bundling constraints. The detail of the bundling constraint calculation is shown in the next section.

```
--dp0 datapath hs process
shop_dp0_hs:  process
begin
  -- receive(wine_delivery,shelf);
  guard(wine_delivery_snd,'1');
  assign(wine_delivery_snd_i,'1',dp0_slb,dp0_sub);
  guard(wine_delivery_snd,'0');
  assign(wine_delivery_snd_i,'0',dp0_rlb,dp0_rub);
  end process shop_dp0_hs;
```

The last process as shown below is from file *shop_dp1_atacs_t.vhd*, which shows how another datapath block looks for a send statement after separating from the other files and how to handle the bundling constraints. The detail of how to calculate all the bundling constraint is shown in the next section.

```
--dp1 datapath hs process
shop_dp1_hs:  process
begin
  -- send(wine_selling,shelf1);
  guard(wine_selling_snd_i,'1');
  assign(wine_selling_snd,'1',dp1_slb,dp1_sub);
  guard(wine_selling_snd_i,'0');
  assign(wine_selling_snd,'0',dp1_rlb,dp1_rub);
  end process shop_dp1_hs;
```

The third major task of this script is the self-timed datapath component generation, which includes all the bundled-data components and finds the maximum delay time from the timing report and also creates our own timing report for the next script, *CtrlSyn*, to use. The datapath generation is based on the information that is stored during the previous parsing stage. *DpSyn* parses each line of the code to separate the control statements and datapath statements. Basically, all of the *guard* and *assign* statements are treated as controller statements, and all of the non-*guard* or *assign* statements are treated as datapath statements. Normally, a complete 4-phase handshake encloses a set of datapath statements, such as assignment(<=), add(+), subtract(−) and so on. This method collects all such statements and refers to them with a unique number for each block. Then, it is simpler to construct datapath components using all these reference-numbers with all of their corresponding statements. The following uses part of the example *shop_hs.vhd*, which is the generated handshaking-level model of *shop.vhd* from the *Ch2Hs* method; it shows parsing results by using the *DpSyn* method, datapath component construction in particular. We use this example throughout the remainder of this chapter.

```
shop:process
begin
  -- receive(wine_delivery,shelf);
  guard(wine_delivery_snd,'1');
  shelf <= wine_delivery_data;
  wait for 5 ns;
  assign(wine_delivery_rv,'1',1,3);
  guard(wine_delivery_snd,'0');
  assign(wine_delivery_rv,'0',1,3);
  shelf1<=shelf+1;
  wait for 5 ns;
  -- send(wine_selling,shelf1);
  wine_selling_data <= shelf1;
  wait for 5 ns;
  assign(wine_selling_snd,'1',1,3);
  guard(wine_selling_rv,'1');
  assign(wine_selling_snd,'0',1,3);
  guard(wine_selling_rv,'0');
end process shop;
```

There are two datapath blocks found from this process. All of the information is stored in the lookup table and looks like the following:

```
Reference #:  0
Statements :  shelf <= wine_delivery_data;
wait for 5 ns;
Reference #:  1
Statements :  shelf1 <= shelf + 1;
wait for 5 ns;
wine_selling_data <= shelf1;
wait for 5 ns;
```

In order to construct the datapath components, besides the above information, there are also some other issues that need to be taken care of. The first issue is how to figure out the entity ports (interfaces) for each datapath component. In order to know that, our tool has to check out the places where the signals appear in an assignment (<=). There are three possibilities. If a signal only appears in the left side of all assignment statements, then the signal should be declared as an output (*out*) signal. If a signal only appears in the right hand side of all assignment statements, then the signal should be declared as an in (*in*) signal. If a signal appears in both sides of the assignment statements, then this signal should be declared as a bidirectional (*inout*) signal. The second issue is how to retrieve the signal type. The corresponding signal type is usually found when *DpSyn* parses the given input VHDL file, which is saved in the lookup table in the previous stage. Therefore, at this point, this method can use the prestored information to get the required data type. The third issue is how to calculate the bundling constraints and create our own timing delay report for the next script, *CtrlSyn*, to use. To do this, *DpSyn* generates two versions of the datapath components. One is to generate the pure datapath for synthesizing the timing report using WebPack, and the other version is to generate the completed self-timed datapath components for synthesizing the whole circuit. For the first version, the generated datapath component files are named *dp_reference-number_md*, according to the same example *shop_hs.vhd*, block 1, then the file should look like the following:

```
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
entity dp_1 is
  port(
      shelf :  inout std_logic_vector(2 downto 0);
      wine_selling_data :  out std_logic_vector(2 downto 0)
  );
end dp_1;
architecture mdelay of dp_1 is
  signal shelf1:std_logic_vector(2 downto 0);
begin
  process
    begin
       shelf1 <= shelf + 1;
       wine_selling_data <= shelf1;
    end process;
end mdelay;
```

In order to get the maximum delay time, we need to import this file into the WebPack to do the synthesis, and the timing report comes out like the following:

```
------------------------------------------------------------------------
```

```
Release 6.1.03i Trace G.26
Copyright (c) 1995-2003 Xilinx, Inc.  All rights reserved.
C:/Xilinx/bin/nt/trce.exe -intstyle is e -e 3 -l 3 -xml dp_1 dp_1.ncd
-odp_1.twr dp_1.pcf
Design file:              dp_1.ncd
Physical constraint file:  dp_1.pcf
Device,speed:              xc2s100,-6 (PRODUCTION 1.27 2003-11-04)
Report level:              error report
Environment Variable      Effect
--------------------      ------
NONE                      No environment variables were set
-----------------------------------------------------------------------------
INFO:Timing:2698 - No timing constraints found, doing default enumeration.
INFO:Timing:2752 - To get complete path coverage, use the unconstrained
   paths option.  All paths that are not constrained will be reported in the
   unconstrained paths section(s) of the report.
Data Sheet report:
-----------------
All values displayed in nanoseconds (ns)
Pad to Pad
---------------+---------------+---------+
Source Pad     |Destination Pad|  Delay  |
---------------+---------------+---------+
req            |ack            |    6.590|
shelf<0>       |shelf1<0>      |    6.519|
shelf<0>       |shelf1<1>      |    7.937|
shelf<0>       |shelf1<2>      |    7.847|
shelf<1>       |shelf1<1>      |    8.029|
shelf<1>       |shelf1<2>      |    7.827|
shelf<2>       |shelf1<2>      |    7.461|
---------------+---------------+---------+
Analysis completed Wed Apr 07 01:30:41 2004
Peak Memory Usage:  43 MB
-----------------------------------------------------------------------------
```

We then analyze this timing report and extract the maximum delay time. Next, we use all of this information to create our own report in order for the next method *CtrlSyn* to parse it easily. This report looks like the following. All of the times are in pico-second (ps) units.

```
>>>delay template for shop:  all delays in ps<<<
      slowgc_delay         8481
      dp0_delay          4660
      dp1_delay          3283
      and_delay          653
      inv_delay          653
```

The other versions of the datapath components are the complete self-timed datapath components, which are as shown below:

```
library IEEE;
```

```
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_unsigned.all;
entity dp_1 is
  port(
      shelf :  in std_logic_vector(2 downto 0);
      wine_selling_data :  out std_logic_vector(2 downto 0);
      wine_selling_snd_i:  in std_logic;
      wine_selling_snd:  out std_logic
  );
end dp_1;
architecture dp4syn of dp_1 is
  signal shelf1:std_logic_vector(2 downto 0);
  component delay_dp_1
    port(
    x:in std_logic;
    xd:out std_logic
    );
  end component;
begin
  process
  begin
       shelf1 <= shelf + 1;
       wine_selling_data <= shelf1;
  end process;
  C_0:  delay_dp_1 port map (
      x => wine_selling_snd_i,
      xd => wine_selling_snd
  );
end dp4syn;
```

Note this file uses the hierarchal way to get all the information. Besides the main datapath, it also has the delay components and port map in the same file. In this way, it does not have to generate the new file each time, it only needs to update the delay file. The implementations of *delay0_dp_0* and *delay0_dp_1* are based on the static timing analysis after WebPack synthesis. The detailed implementation is in next method *CtrlSyn* in the next section.

The algorithm of this script is shown as follows:

```
Script       :  DpSyn.pl
Dependency   :  none
Description :  transforms handshake-level VHDL specification to its
corresponding datapath/controller VHDL and delay
specification files
Inputs       :  handshake-level VHDL file
Outputs      :  all generated files will be put in the  ./<VHDL file name>
=> dp_x.vhd       :  self-timed datapath VHDL files for
WebPack synthesis
=> dp_x_md.vhd   :  matched delay of the datapath dp_x.vhd
using static timing analysis
```

```
=> delay_y.dat    :  default delay element specification
=> y_ctrl_atacs_t.vhd  :  corresponding controller for ATACS synthesis
=> y_dp0_atacs_t.vhd  :  corresponding dp0 for ATACS synthesis
=> y_dp1_atacs_t.vhd  :  corresponding dp1 for ATACS synthesis
=> y_wp.vhd       :  corresponding controller top-level for WebPack
simulation and synthesis
Notes       :  "x" represents 0..n that is depended on how many datapath
segments within the given handshake-level VHDL file, and
"y" is the name of the given VHDL file.
Algorithm {
[0] clear datapath lookup-table for storing every non-guard/assign
segment block;
[1] clear rewiring lookup-table for storing each complete 4-phase
handshaking block;
[2] record all signal declarations which include entity-port and
architecture;
[3] extract non- guard/assign segment blocks and store in the
datapath look-up-table;
[4] extract rewiring information depend on the types (passive
or active) of each complete 4-phase guard/assign block and store the
results into the rewiring lookup-table;
[5] use the datapath lookup-table to generate all self-timed
datapath VHDL files and the default matched delay data file;
[6] use the rewiring lookup-table to generate 3 handshake-level
VHDL files for atacs with each datapath replaced with a handshake
process using guard/assign statements, which is made for ATACS synthesis;
[7] use the rewiring look-up-table and information held in [2] to
generate structural-level VHDL for all the controller and datapth
components with port-map statements, which is made
for simulation, ATACS/WebPack synthesis;
}
```

3.3 Controller Synthesis

DpSyn generates different component files that include the datapath maximum delay files named *dp_reference-number_md*, handshaking-level controllers named *filename_atacs*, self-timed datapath component files named *dp_reference-number*, and the top-level file named *filename_wp*. This section focuses on what the controller synthesis (*CtrlSyn*) script does with these output files. This includes calculating the bundling constraints by using the newly created timing report, updating the handshaking-level controller, and using ATACS to synthesize the controller.

Recall the timing report created by *DpSyn* includes all the timing information for each datapath component. This includes the slowest gC circuit delay that is named *slowgc_delay*; the upper bound *AND* delay that is named *and_delay*; maximum delay time for the datapath that is named *dp_delay*; lower bound delay time for each inverter that is named *inv_delay*. After synthesizing the example using WebPack, our method uses 0 ns for the lower bound setup and reset time for the controller process, and it uses the time result for the most complicated component from the structual level controller as the upper bound setup and reset time for the controller process, which is 8.481ns for n4p2.

After getting all the timing information, we first calculate the number of inverters (even numbers) we need in different examples in order to create a timing delay. The way to calculate this is:

$$\#\text{ of inv} = \lceil \frac{dp\_delay - and\_delay}{\text{inv\_delay} * m} \rceil * (1 + safety\_margin)$$

The m percent of inv_delay is used here to calculate the lower bound of the inverter delay, we assume 90 percent in this case. Safety_margin in this thesis is taken as 50 percent. In addition, the number of inverters here is always rounded up to the next highest even numbers to preserve the polarity.

Next, we can calculate the final setup time and reset time which are used to replace the template for the controller. The equations this method uses are:

$$slb = \frac{\#\text{ of inv} * \text{inv\_delay * m}}{1000}$$

$$sub = \frac{\#\text{ of inv} * \text{inv\_delay}}{1000}$$

$$rlb = \frac{and\_delay * m}{1000}$$

$$rub = \frac{and\_delay}{1000}$$

The m here represents the mininum delay ratio, which is assumed to be 90 percent in this case.

At this point, since we have already generated the handshaking-level controller templates from the second script but with assumed lower bound and upper bound timing ranges in it, *CtrlSyn* uses the extracted maximum delay time to replace these timing ranges in the handshaking statements. In order to explain this clearly, consider the following examples of the controller and datapath templates from *shop_hs.vhd* we generated from *DpSyn*. *CtrlSyn* replaces all the assumed maximum delay times with real delay times calculated from the timing report. These three templates are transformed to the following:

The real time delay from file *shop_dp0_atacs.vhd* is used to update the follow process:

```
--dp0 datapath hs process
shop_dp0_hs:  process
begin
  -- receive(wine_delivery,shelf);
  guard(wine_delivery_snd,'1');
  assign(wine_delivery_snd_i,'1',8,9);
  guard(wine_delivery_snd,'0');
  assign(wine_delivery_snd_i,'0',1,2);
  end process shop_dp0_hs;
```

The real time delay from file *shop_dp1_atacs.vhd* is used to update the following process:

```
--dp1 datapath hs process
shop_dp1_hs:  process
begin
  -- send(wine_selling,shelf1);
  guard(wine_selling_snd_i,'1');
  assign(wine_selling_snd,'1',8,9);
  guard(wine_selling_snd_i,'0');
  assign(wine_selling_snd,'0',1,2);
  end process shop_dp1_hs;
```

The real time delay from file *shop_ctrl_atacs.vhd* is used to update the following process:

```
--controller hs process
shop_ctrl_hs:  process
begin
  -- receive(wine_delivery,shelf);
  guard(wine_delivery_snd_i,'1');
  assign(wine_delivery_rv,'1',0,8);
  guard(wine_delivery_snd_i,'0');
  assign(wine_delivery_rv,'0',0,8);
  -- send(wine_selling,shelf1);
  assign(wine_selling_snd_i,'1',0,8);
  guard(wine_selling_rv,'1');
  assign(wine_selling_snd_i,'0',0,8);
  guard(wine_selling_rv,'0');
  end process shop_ctrl_hs;
```

At this point, ATACS can be used to synthesize a structural-level VHDL code for the controller.

In addition, *CtrlSyn* also generates the structural-level delay elements with all the buffer chains in it. It determines how much delay time is needed for one single buffer and then uses the extracted maximum delay information to calculate how many buffers the circuit needs in total. The way to calculate this is shown earlier. All of these generated files are used for the final synthesis step. The delay elements structure for the first datapath dp_0 is shown in Figure 3.3.

The example below is the delay file for the first datapath dp_0:

```
-- Delay line implementaiton for Xilinx WebPack synthesis
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
-- synopsys translate_off
library UNISIM;
use UNISIM.Vcomponents.ALL;
-- synopsys translate_on
entity delay_lfsr is
   port ( x   :  in    std_logic;
          xd  :  out   std_logic);
end delay_lfsr;
architecture STRUCTURE of delay_lfsr is
   attribute KEEP       : STRING ;
   attribute BOX_TYPE   : STRING ;
   component INV
      port ( I : in    std_logic;
             O : out   std_logic);
   end component;
   attribute BOX_TYPE of INV : COMPONENT is "BLACK_BOX";
   component AND2
      port ( I0 :  in    std_logic;
             I1 :  in    std_logic;
             O  : out   std_logic);
   end component;
   attribute BOX_TYPE of AND2 :  COMPONENT is "BLACK_BOX";
    signal x0:std_logic;
```

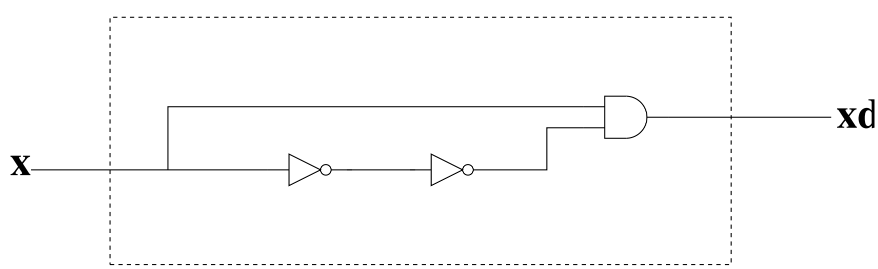

Figure 3.3. Delay elements structure for dp_0.

```
    signal x1:std_logic;
    signal x1_int:std_logic;
begin
    XLXI_0 :  INV
              port map (I=>x, O=>x0);
    XLXI_1 :  INV
              port map (I=>x0, O=>x1);
              x1_int <= x1 after 5 ns;
    XLXA_0 :  AND2
              port map (I0=>x, I1=>x1_int, O=>xd);
end STRUCTURE;
```

According to this WebPack synthesis generated file, there is an instruction called *attribute*, which is associated with symbols or nets in an FPGA or CPLD schematic to indicate their placement, implementation, naming, directionality, or other properties. It is mainly used for signal optimization. In the design fitting process, the implementation of signals can be specified in the UCF (User Constraint File). The fitter interprets signals and reduces logic to create an optimum fit in the device. The attributes that are available for optimizing signal equations in XPLA3 are *keep*, *collapse*, and *noreduce*. In this case, we focus on *keep*, which preserves internal nodes during design implementation [5]. The *keep* constraint is an advanced constraint. When a design is mapped, some nets may be absorbed into logic blocks. When a net is absorbed into a block, it can no longer be seen in the physical design database. This may happen, for example, if the components connected to each side of a net are mapped into the same logic block. The net may then be absorbed into the block containing the components. *Keep* prevents this from happening [6]. The other *attribute* used here is *BOX_TYPE*. The *box_type* constraint currently takes only one possible value: *black_box*. The black_box value instructs XST to not synthesize the behavior of a model. For our method, both *keep* and *box_type* are used for preventing the buffers from optimization in order to create buffer chains to handle the bundling constraints, because Webpack eliminates all even numbers of inverters and reduces all odd numbers of inverters to one.

The algorithm of this script is shown as follows:

Script : `CtrlSyn.pl`
Dependency : `DelayGen.pl`
Description : `based on the given delay information to generate matched`

```
delay component (VHDL specifications) with inverter chain
for all datapath VHDL files from the script "DpSyn.pl" and
also fill in the timing information into the handshake-level
VHDL controller for ATACS synthesis
Inputs      :  script DpSyn.pl's working directory name
Outputs     :  all generated files will be put in the given working
directory
=> delay_dp_x.vhd :  matched delay component of the datapath
   dp_x.vhd using inverter chain
=> y_atacs.vhd    :  timing information updated handshake-level
  controller for ATACS synthesis
Notes       :  "x" represents 0..n that is dependent on how many datapath
segments within the given handshake-level VHDL file, and
"y" is the name of the given VHDL file.
Algorithm {
[0] parse given timing information file;
[2] update timing information for the "y_atacs_t.vhd" and store
the result as "y_atacs.vhd" which can be used for ATACS synthesis;
for each parsed timing description blocks {
[3] use the script "DelayGen.pl" with the current block
timing information to generate structural-level VHDL file
that mimics the matched delay line;
  }
}
```

At this point, we have gotten the real delay timed controller in handshaking-level VHDL code, the bundling constraints files, the complete self-timed datapath components, and the top-level of the circuit. However, the controller is still in handshaking-level VHDL code, so our tool needs to use ATACS to synthesize it. During the synthesis process, ATACS checks the datapath components and ignores the assignments (<=), which leaves only the handshaking statements to synthesize. The synthesis command is *atacs -tsyssv winery_hs.vhd patron_hs.vhd shop_ctrl_atacs.vhd shop_dp0_atacs.vhd shop_dp1_atacs.vhd shop_wp.vhd wine_example2_hs.vhd THE_SHOP.shop_ctrl_inst* in this case. The output structural-level VHDL code looks like the following:

```
library ieee;
use ieee.std_logic_1164.all;
entity THE_SHOP_shop_ctrl_inst is
  port(   wine_delivery_rv :  inout std_logic;
       wine_delivery_snd_i :  in std_logic;
           wine_selling_rv :  in std_logic;
        wine_selling_snd_i :  inout std_logic
       );
end THE_SHOP_shop_ctrl_inst;
architecture structure of THE_SHOP_shop_ctrl_inst is
```

```
  -- internal signals declarations
  signal CSC0:  std_logic;
  -- input complements
  signal wine_delivery_snd_i_bar:  std_logic;
  signal wine_selling_rv_bar:  std_logic;
  -- output complements
  signal CSC0_bar:  std_logic;
  signal wine_delivery_rv_bar:  std_logic;
  signal wine_selling_snd_i_bar:  std_logic;
  component n1p1
  port(a1n :  in std_logic;
       a1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
  component n2p2
  port(a1n :  in std_logic;
       b1n :  in std_logic;
       a1p :  in std_logic;
       b1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
  component n3p2
  port(a1n :  in std_logic;
       b1n :  in std_logic;
       c1n :  in std_logic;
       a1p :  in std_logic;
       b1p :  in std_logic;
       o :  inout std_logic;
       obar :  inout std_logic);
  end component;
begin
  i_0 :  n3p2
    port map(a1n=>wine_delivery_snd_i, b1n=>wine_selling_rv_bar,
             c1n=>wine_selling_snd_i_bar,
             a1p=>wine_delivery_snd_i, b1p=>CSC0_bar,
             o=>wine_delivery_rv, obar=>wine_delivery_rv_bar);
  i_1 :  n2p2
    port map(a1n=>wine_delivery_rv_bar, b1n=>CSC0,
             a1p=>wine_selling_rv_bar, b1p=>CSC0,
             o=>wine_selling_snd_i, obar=>wine_selling_snd_i_bar);
  i_2 :  n1p1
    port map(a1n=>wine_delivery_rv,
             a1p=>wine_selling_snd_i_bar,
             o=>CSC0, obar=>CSC0_bar);
  -- complements of inputs
  wine_delivery_snd_i_bar <= not wine_delivery_snd_i after
0 ns;
  wine_selling_rv_bar <= not wine_selling_rv after 0 ns;
end structure;
```

Besides the structural-level controller, there are also the bundling constraints files, and all of the datapath component files named by the reference-numbers, plus the top level file *filename_wp*. Now all of the files are in structural-level VHDL code, which can be successfully synthesized by WebPack. The synthesis report is shown in Appendix A. Besides the synthesis report, the WebPack also generates a floorplan, which shows the mapping for shop controller onto the FPGA lookup tables. The equivalent schematic of the shop controller floorplan is shown in Figure 3.4.

3.4 Validation

This section describes the validation of the work using the *Wineshop* example. Since the major work of this thesis is to automate the translation procedures from one modeling method to another, the validation is very critical here in order to check the correctness after each translation step for the example. The way we check the validation of the work is to use ModelSim to simulate. First, before any translation, we only have the top level channel VHDL files, and we simulate it to make sure it is a working example. Then we use *Ch2Hs* to translate this channel-level model to handshaking-level model and simulate again. Next, apply *DpSyn* and *CtrlSyn* onto the *shop* file to translate it into a structural level VHDL file, but use the same handshaking-level environments to simulate it. After this, *shop* file is synthesized using WebPack to generate the behavioral level VHDL file, once again, it is simulated using the same handshaking-level environment. Figure 3.5 shows all the different levels of validation. This way, we can simulate all the different levels of the *shop* file but use the same environment files.

The way to simulate examples in ModelSim is to first create a new project and then add all the necessary files into the project. Next, compile all the files in the correct order and start simulating using the top level file. Then, add all the signals in the region into the waveform window. After running several cycles, we can check the correctness from the waveform.

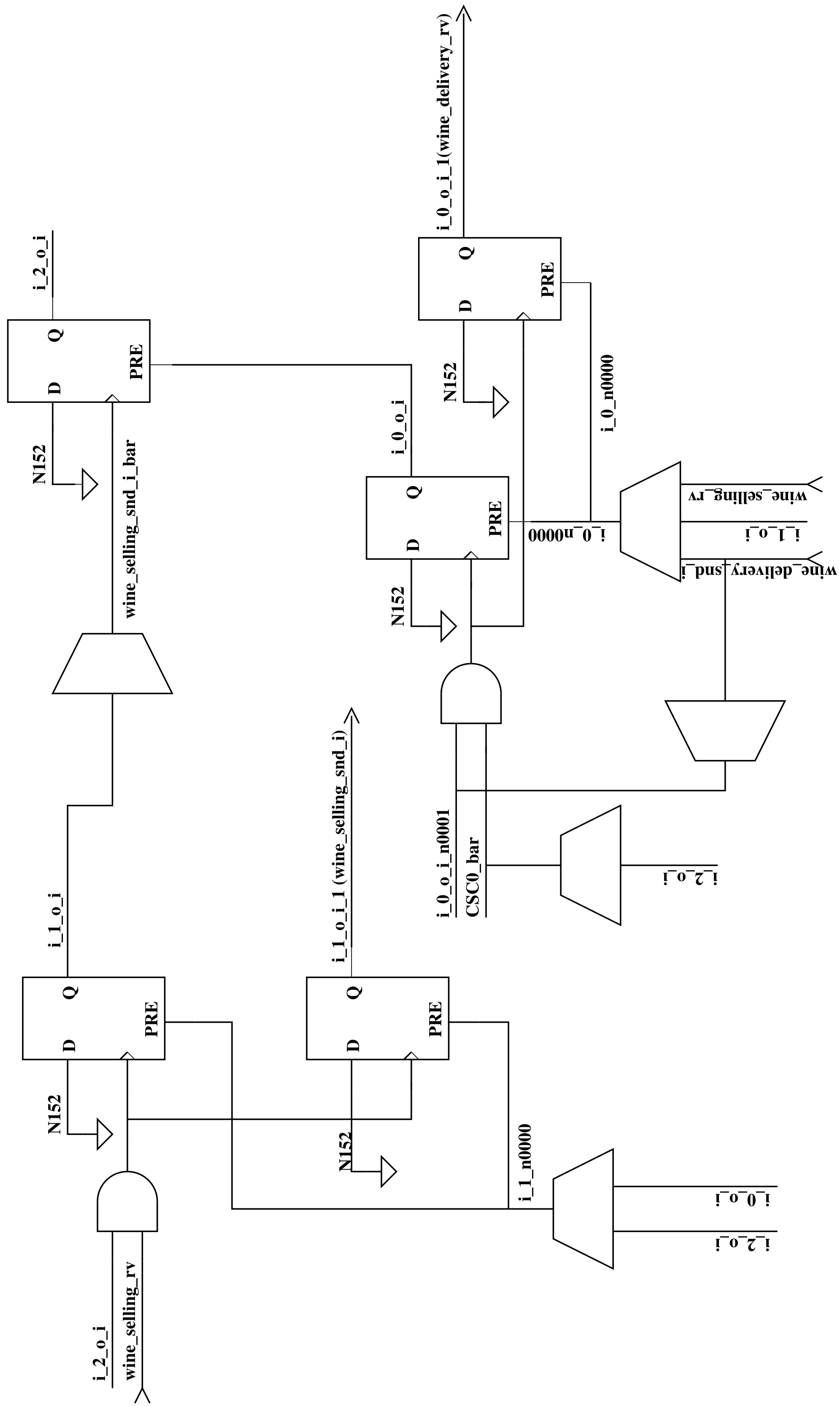

Figure 3.4. Schematic of the floorplan for the shop controller.

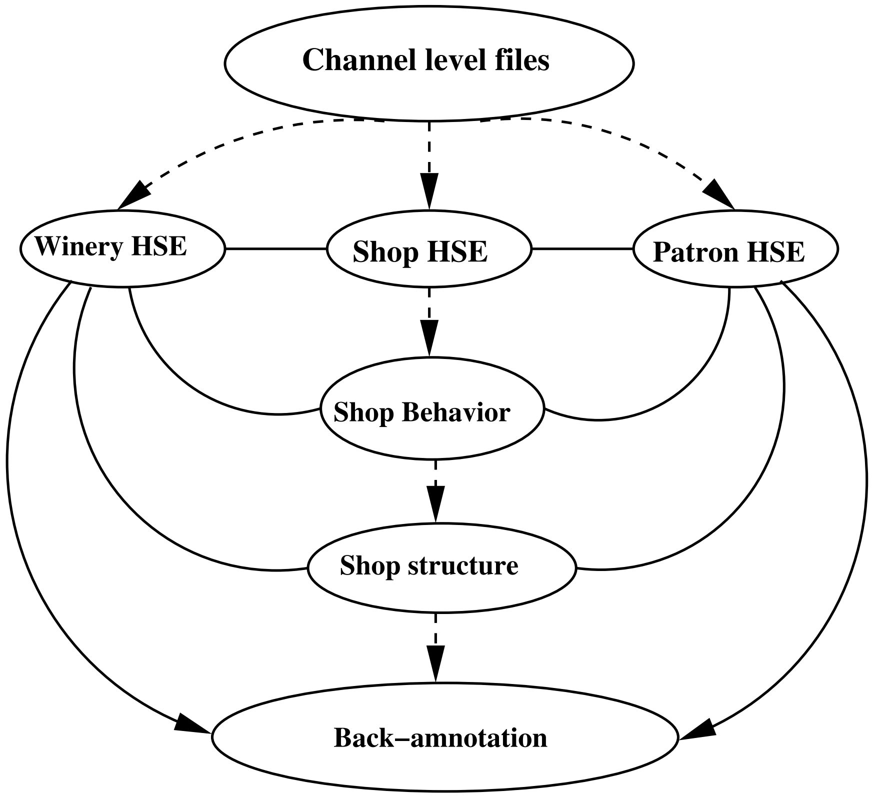

Figure 3.5. Validation for the shop example.

3.5 Testing

Another important issue is the testing of the circuit. There are two major parts involved in testing a circuit: one is generating an appropriate set of test vectors and the other one is comparing the actual output and correct computed output. The input test vectors can be generated in two ways, *off-line test pattern generation* and *concurrent test pattern generation*. However, off-line test pattern generation is not very popular, since it does not reduce the cost of testing and also requires large amount of the memory to store the vectors. Therefore, most of the *built-in self test* (BIST) methods rely on concurrent test pattern generation. *Random testing* and *exhaustive testing* are two more frequently used approaches because they do not depend on the availability of an instruction processor, so they can be used either with or without an instruction processor that is more general than the test program approach. Our method chooses the *random testing* approach since it is applicable to both sequential circuits and combinational circuits. The way to generate the input testing patterns is to use a simple circuit called an *autonomous linear feedback shift register* or *ALFSR*, which is a series connection of delay elements (D flip-flops) with no external inputs and with all feedback provided by means of EXCLUSIVE-OR gates (XORs) [21]. In general, the maximum period for an n-stage ALFSR is $2^n - 1$. There are many compression techniques to do the testing, such as one counting, transition counting, parity checking, syndrome checking and signature analysis, and so on. However, some of them require large amounts of memory, and some of them are costly. A conventional approach would compare the output values bit-by-bit between the actual circuit with the corrected values as previously computed and saved. However, this approach requires a large amount of memory space to store those correct outputs associated with all of the test vectors and also slows down the speed since it compares one-bit at a time. After considering the alternative approaches, the approach we used in this project is simple, requires less memory space, and also brings up the speed of the testing circuit. We compress the output values into a *signature*. *Signature analysis* is a compression technique based on the concept of cyclic redundancy checking (CRC) and realized in hardware using linear feedback shift registers (LFSRs) [22].

The whole testing method can be separated into three stages, which are a stage to generate the inputs to be applied, a stage with the circuit under test, then a stage to analyze the outputs response from the circuit. The random test generator is constructed of a simple circuit called a *linear feedback shift register* or *LFSR*. It randomly generates the test patterns and sends them to our circuit. After going through the circuit, the outputs are sent into the signature analyzer, which analyzes the output value. The testing depends on the final value of the LFSR flip-flops, and the signature depends on the testing bit pattern that is given at the input. At the end, the signature is serially shifted out of the LFSR flip-flops and is compared with the pre-computed results in the VHDL, which we get from the simulation part. If a fault causes the output bit sequence to change, then it usually results in a different signature in the LFSR. However, if there is aliasing, then the fault may not be detected because it is possible for a fault to cause an output bit sequence that produces the same final signature from the LFSR contents as the fault-free circuit. Due to aliasing, any compaction technique can cause some loss of effective fault coverage. In our method, each channel would have one random generator or one signature analyzer. For instance, every *receive* channel has a random generator to get the input data from an *LFSR*, and every *send* channel connects to a signature analyzer for comparing the output data. The whole method is shown in Figure 3.6.

The rest of this section uses the *shop* as an example to explain this method in detail. Applying the testing method we have on the *shop* example, the whole testing circuit should be as shown in Figure 3.7. As we can see going from left to right in this figure, the left part is a random generator, which utilizes an LFSR to generate *pseudo-random* test patterns. *Pseudo-random* is a “fixed” random sequence that is repeated within every cycle, $(2^n - 1)$ numbers. It is a very compact implementation that only requires a few XOR gates and some D flip-flops. The random sequence is also going to be exactly identical after the reset operation. The length of the

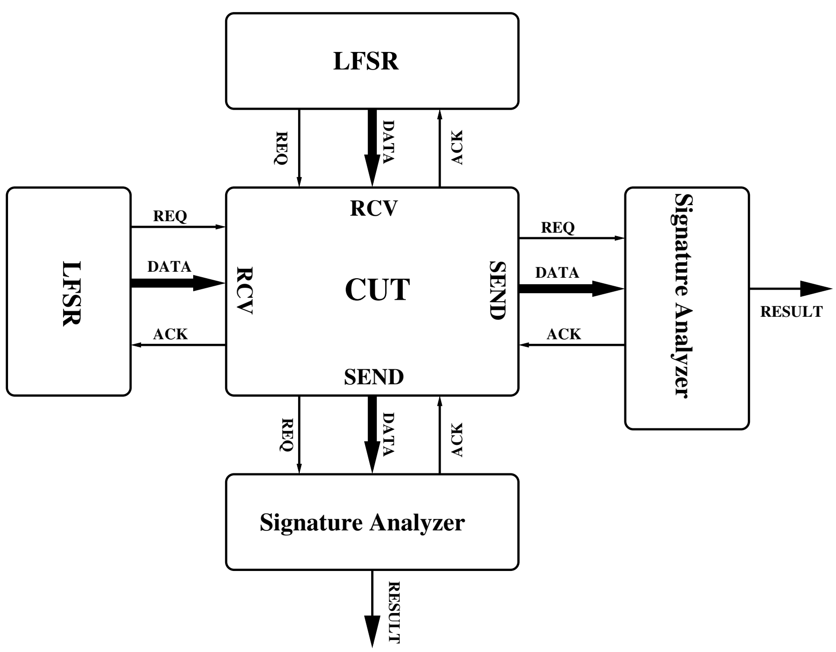

Figure 3.6. Test circuit for channels.

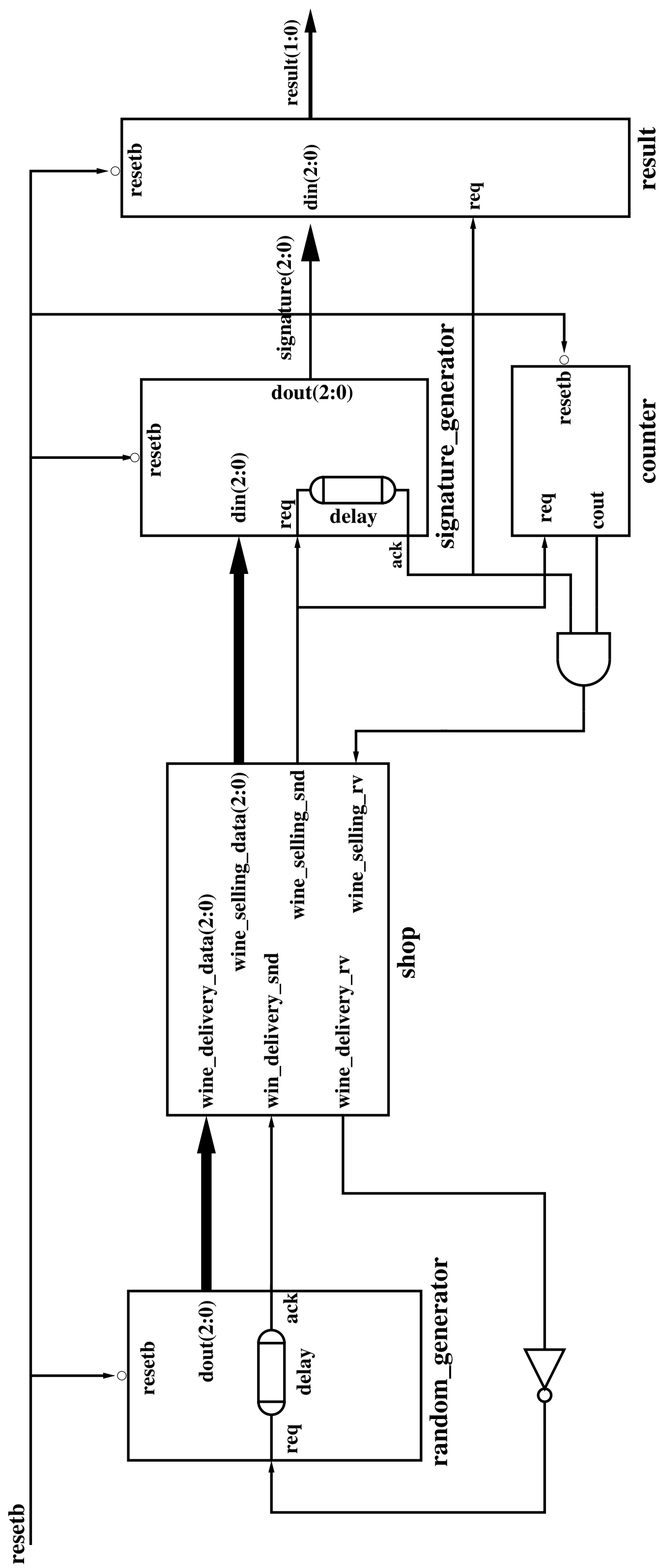

Figure 3.7. Test circuit for shop.

random sequence is $2^n - 1$, and n is the number of D flip-flops used. The second block is the circuit under test (CUT), which is the *shop* example in this case. The third block is a signature generator, which utilizes the same technique (LFSR) as the random generator does. In this signature generator, each stage consists of a 2-input or 3-input XOR gate and a D flip-flop. The choice of using a 2-input or 3-input XOR gate depends on which CRC equation is picked, and also the parallel inputs to each stage are the result generated by the circuit under testing (CUT). How this circuit works is: first the active low reset goes to low and back to high, which resets all of the blocks, then request (*req*) of the random generator is high, the data on dout is ready to send to *wine_delivery_data*, then the acknowledge (*ack*) is set to high, and next the request (*req*) goes low, then the acknowledge goes low, which completes the whole 4-phase handshaking. Now it gets to the CUT, *shop*, when acknowledge (*ack*) of the random generator sets high, which makes the *wine_delivery_snd* go high, so *wine_delivery_data* sends data to the *shelf*, then the *wine_delivery_rv* goes high and then to low in order to complete the 4-phase handshaking, which makes the request (*req*) of the LFSR go to high and start generating the next random input. Meanwhile, the shop block does its own work which increases *shelf* by 1 and sends it to *shelf1*, then *shelf1* sends the data to *wine_selling_data* for output. Next, *wine_selling_snd* goes high, which makes the request (*req*) of the signature generator go high and also increments the counter. After a certain delay time, the *wine_selling_rv* goes to high to tell the shop that the *wine_selling_data* is output successfully. Then the *wine_selling_snd* goes low, and *wine_selling_rv* goes low to complete the 4-phase handshaking communication. Meanwhile, the signature generator would output the data and compare with the correct evaluated data in order to verify the result in the *result* block. In Figure 3.7, we can see that the *wine_selling_rv* is generated by an *AND* of two signals; one is the *wine_selling_snd*, and the other one is the inverting of carry out of the counter (cout). The reason we have this is for halting the signature generator and stopping this self-testing circuit. The way it works is that carry-out (cout) of the counter is initially 0 making *wine_selling_rv* dependent on acknowledge, *ack*, of the *signature analyzer*, which makes the circuit keep going. However, once a certain number of signatures are analyzed, carry-out changes to 1 causing *wine_selling_rv* to halt and stop the circuit. After comparing the signatures, the *result* component outputs the final results. In this case, "00" stands for testing in progress, "01" stands for testing is finished but it fails, "11" stands for finishing the testing and it passes successfully. The implementation of the *random generator*, the *signature generator*, the *counter* and the *result* are shown in Appendix B.

The simulation result for the whole testing circuit is shown in Figure 3.8.

3.6 Conclusion

This chapter presents the complete design flow and its automation for most of the steps. The design flow is for synthesizing the channel-level asynchronous circuits by using synchronous synthesis tools. The goal of the automation is to make the whole design flow easier and more efficient to use.

Three tools, *Ch2Hs*, *DpSyn*, and *CtrlSyn*, are implemented to do the work. The goal of these tools is to automate different steps of the design flow. *Ch2Hs* translates the channel-level model to handshaking-level model, *DpSyn* separates the handshaking-level model into different datapath components and a controller, *CtrlSyn* generates the bundling constraints using the timing report, generates the complete self-timed datapath components and also updates the controller with real time delay for synthesis by ATACS.

The whole design shows that, after getting all of the files in structural-level VHDL, users can use WebPack to simulate and synthesize the circuit. The steps of the whole procedure have been discussed and the possible results have been presented.

Finally, a testing method for the circuit has been introduced, which is built from random generators, the circuit under test (CUT), and signature analyzers. There was an example to show how this self-testing circuit works. However, due to the time constraint, the implementation of the circuit is not completed yet and thus will be discussed more in future work.

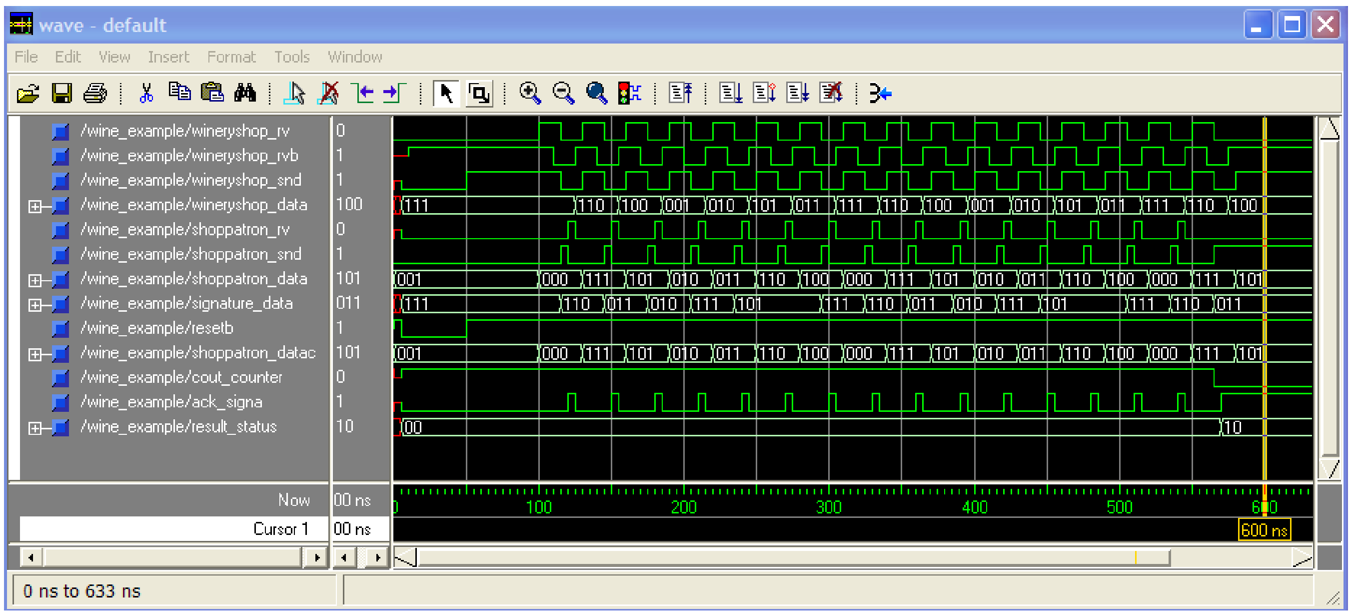

Figure 3.8. Simulation result for testing circuit of shop.

CHAPTER 4

CASE STUDY

This chapter uses several examples as case studies to discuss the method we present in this thesis. For any given example, the most important issue is to successfully produce the circuits by using the method we presented. Therefore, this chapter presents three complete examples to demonstrate the entire design flow for the method. This chapter is organized as follows. Section 4.1 presents an adder example. Section 4.2 describes a 4-bit shifter example. Section 4.3 discusses the example of an asynchronous ditherer for a MPEG decoder.

4.1 Adder

Consider a 4-bit adder as the first example to demonstrate our methods. This 4-bit adder has an interface consisting of five channels, which are the X channel, the Y channel, the Cin channel, the Sum channel, and the $Cout$ channel. When the module receives data on its X, Y, and Cin channels, it extends all of the data to five bits, then adds them together, and stores the result internally. After a delay time for calculation, the module sends the current value of its stored result out of the Sum and $Cout$ channels. Figure 4.1 shows the channel-level block diagram for the adder.

The following channel level VHDL code specifies the behavior of this 4-bit shifter example. The *adder* file specifies what this 4-bit adder does, the *add_env* specifies the environment of the adder, the *adder_top* specifies the top level of the adder. The VHDL code for *adder_env* and *adder_top* are shown in Appendix C.

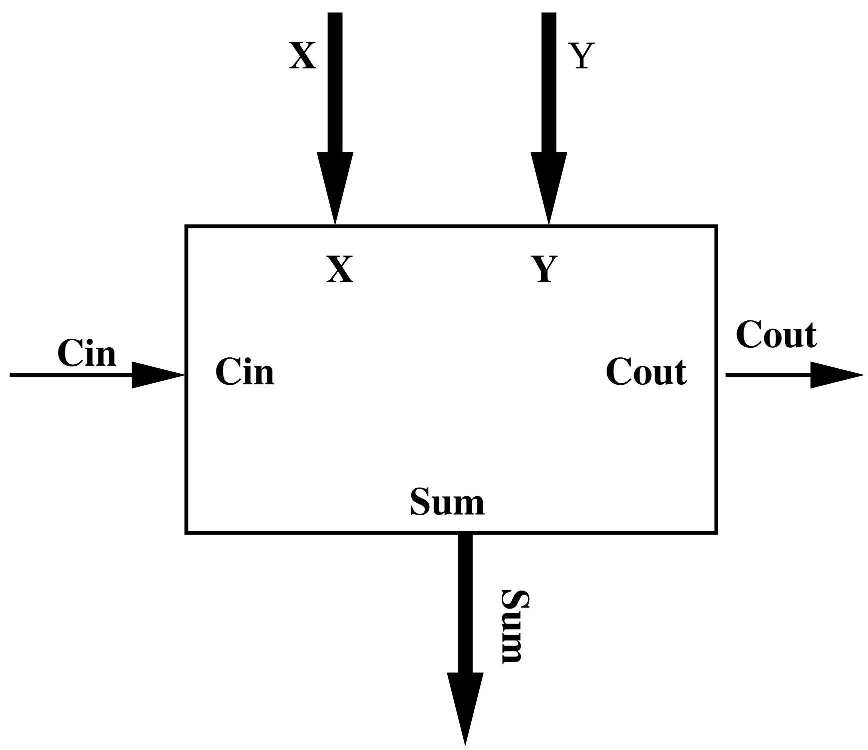

Figure 4.1. Block diagram for a 4-bit adder.

```
------------------------------------------------------
-- Title :  Adder
------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.channel.all;
entity adder is
  port(X:inout channel:=init_channel;
       Y:inout channel:=init_channel;
       Cin:inout channel:=init_channel;
       Sum:inout channel:=init_channel;
       Cout:inout channel:=init_channel);
end entity adder;
architecture behavior  of adder is
  signal a:std_logic_vector(3 downto 0);
  signal b:std_logic_vector(3 downto 0);
  signal c:std_logic_vector(0 downto 0);
  signal s:std_logic_vector(4 downto 0);
begin
process
begin
  receive(X,a,Y,b,Cin,c);
  s <= ("0" & a) + ("0" & b) + ("0000" & c);
  wait for 5 ns;
  send(Sum,s(3 downto 0),Cout,s(4 downto 4));
end process;
end behavior;
```

After going through our tools and using WebPack, the simulation result is shown in Figure 4.2:

4.2 Shifter

Consider the example of a 4-bit shifter, which is constructed by using three identical modules and one special module at the end of the shifter. Each module stores one bit of data. Any identical module has six channels, which are the *Load* channel, the *Shift_in* channel, the *Shift_out* channel, the *Done_in* channel, the *Done_out* channel and the *Output* channel. The special module does not have *Shift_out* and *Done_out* channels. The *Done_in* and *Done_out* channels are only synchronization channels. When a module receives data on its *Load* channel, it stores the bit internally. When a communication is received on the *Shift_in* channel,

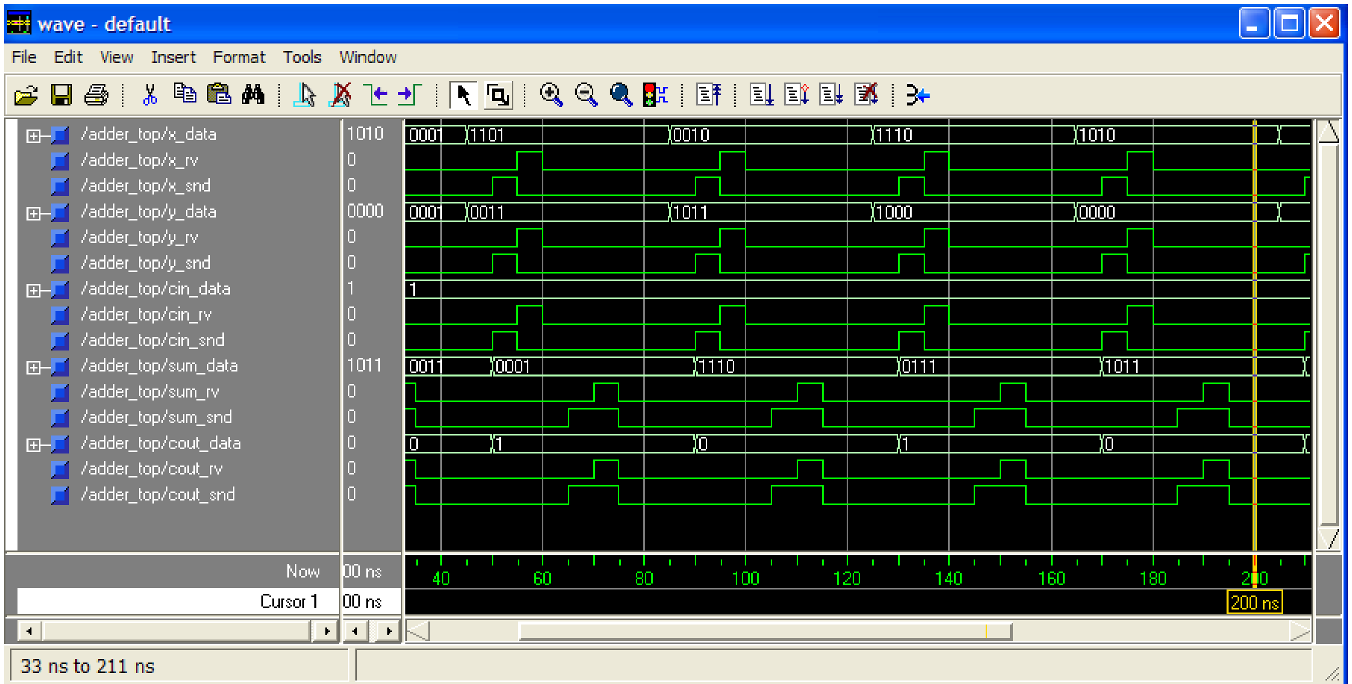

Figure 4.2. Simulation result for adder.

the module sends out the stored bit using the *Shift_out* channel and receives a new data bit on its *Shift_in* channel. When a communication is received on the *Done_in* channel, the module sends its stored data bit out to the *Output* channel and sends a communication on the *Done_out* channel. Figure 4.3 shows the channel-level block diagram for this 4-bit shifter.

The following VHDL code specifies the behavior of this 4-bit shifter example in channel-level VHDL. The *reg_bits* file specifies any identical module of this shifter, the *reg_lsbit* specifies the end bit of this shifter, and the *shifter_env* file specifies the environment. The VHDL code for *shifter_env* is shown in Appendix D.

```
----------------------
-- reg_bits.vhd
----------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity reg_bits is
-- declare i/p - o/p ports
  port(Load:inout channel:=init_channel;
       Shift_in:inout channel:=init_channel;
       Shift_out:inout channel:=init_channel;
       Done_in:inout channel:=init_channel;
       Done_out:inout channel:=init_channel;
       Output:inout channel:=init_channel);
end reg_bits;
architecture behavior of reg_bits is
-- declare local signals
  signal bit:std_logic;
```

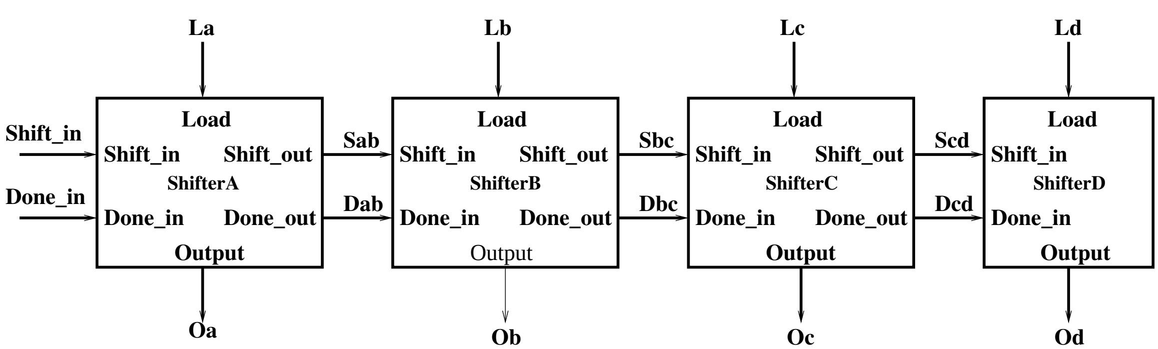

Figure 4.3. Block diagram for a 4-bit shifter.

```
begin
reg_bits:process
begin
           await_any(Load, Shift_in, Done_in);
              if (probe(Load)) then
                 receive(Load,bit);
                          elsif(probe(Shift_in)) then
                send(Shift_out, bit);
                receive(Shift_in, bit);
              else
                send(Done_out);
                receive(Done_in);
                send(Data_out,bit);
              end if;
end process reg_bits;
end behavior;

--------------------------
-- reg_lsbit.vhd
--------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity reg_lsbit is
-- declare i/p - o/p ports
  port(Load:inout channel:=init_channel;
       Shift_in:inout channel:=init_channel;
       Done_in:inout channel:=init_channel;
       Output:inout channel:=init_channel);
end reg_lsbit;
architecture behavior of reg_lsbit is
-- declare local signals
  signal bit:std_logic;
begin
reg_lsbit:process
begin
      await_any(Load, Shift_in, Done_in);
              if (probe(Load)) then
                 receive(Load,bit);
                    elsif (probe(Shift_in)) then
               receive(Shift_in, bit);
              else
                receive(Done_in);
                send(Data_out,bit);
              end if;
      end process reg_lsbit;
end behavior;
```

After going through our tools and using WebPack, the simulation result is shown in Figure 4.4.

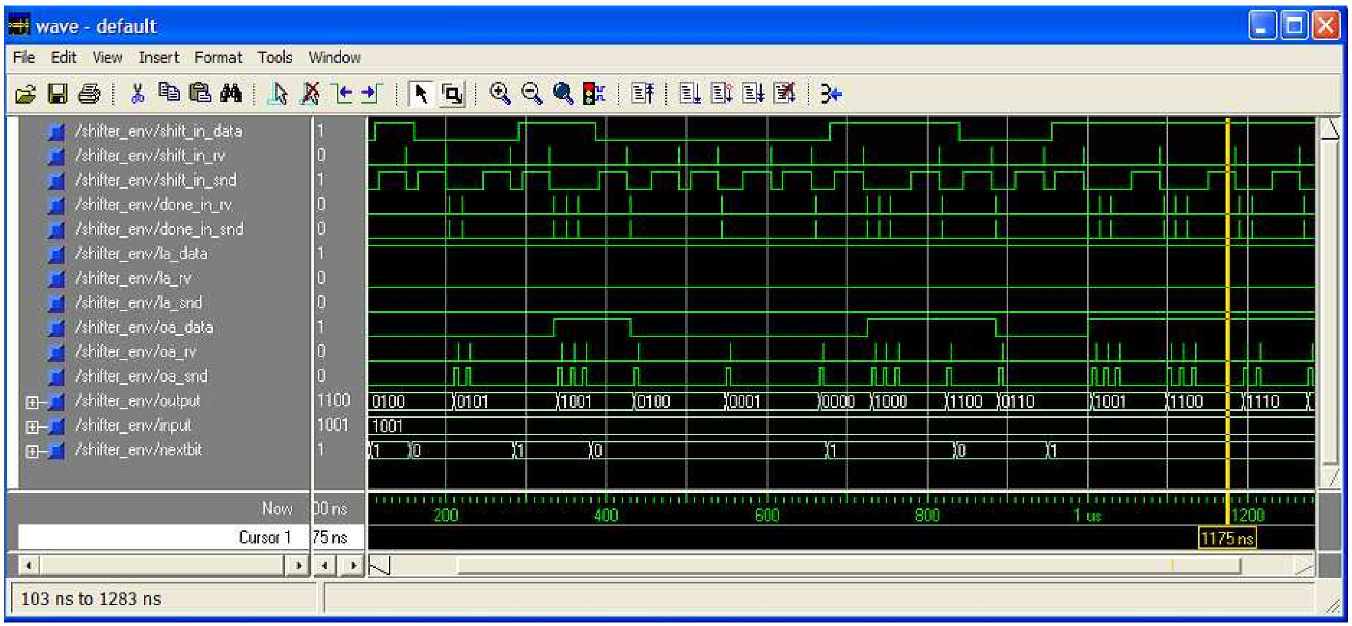

Figure 4.4. Simulation result for shifter.

4.3 An MPEG Ditherer

This section presents the example of a very important operation, dithering, for a *Moving Picture Expert Group* (MPEG) decoder. The input of the ditherer function is a frame that is described using luminance (Lum) and chrominance (C_b, C_r) data. It uses a translation table to map Lum, C_r, C_b into 8−bit RGB color maps. The ditherer function also smooths the image in order to make the picture not too blocky. The design of the ditherer includes both a control circuit and a datapath. For the control circuit, we use channel-level VHDL code, which controls the datapath to receive data, calculate data, send data, etc. For the datapath part, it is broken down into several manageable subcomponents, such as a latch for Lum, a latch for C_b, a latch for C_r, *DACalculation*, *TranslationTable*, Memory, and Counter. Most components are also asynchronous designs using channel-level VHDL code. All of these have been simulated by using ModelSim. In order to simulate this hardware implementation, we need to have a test bench for the whole design, which basically sends the *lum.dat*, *cb.dat*, *cr.dat*, *DA.dat* to the datapath, and after all the calculations in the hardware implementation, the hardware sends the output data to the test bench to compare with the result we have from the software implementation. After the simulation, all of the results matched successfully. Figure 4.5 shows the channel-level block diagram for the ditherer example. The following VHDL code specifies the controller of the ditherer example in channel level, and the rest of the VHDL code are shown in the Appendix E.

```
-----------------------------------------------------------------------------
--
-- Title       :  ctrl
-- Design      :  datapath
-- Author      :  yy
-- date        :  6-8-2002
-----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use std.textio.all;
use work.nondeterminism.all;
use work.channel.all;
use work.handshake.all;
```

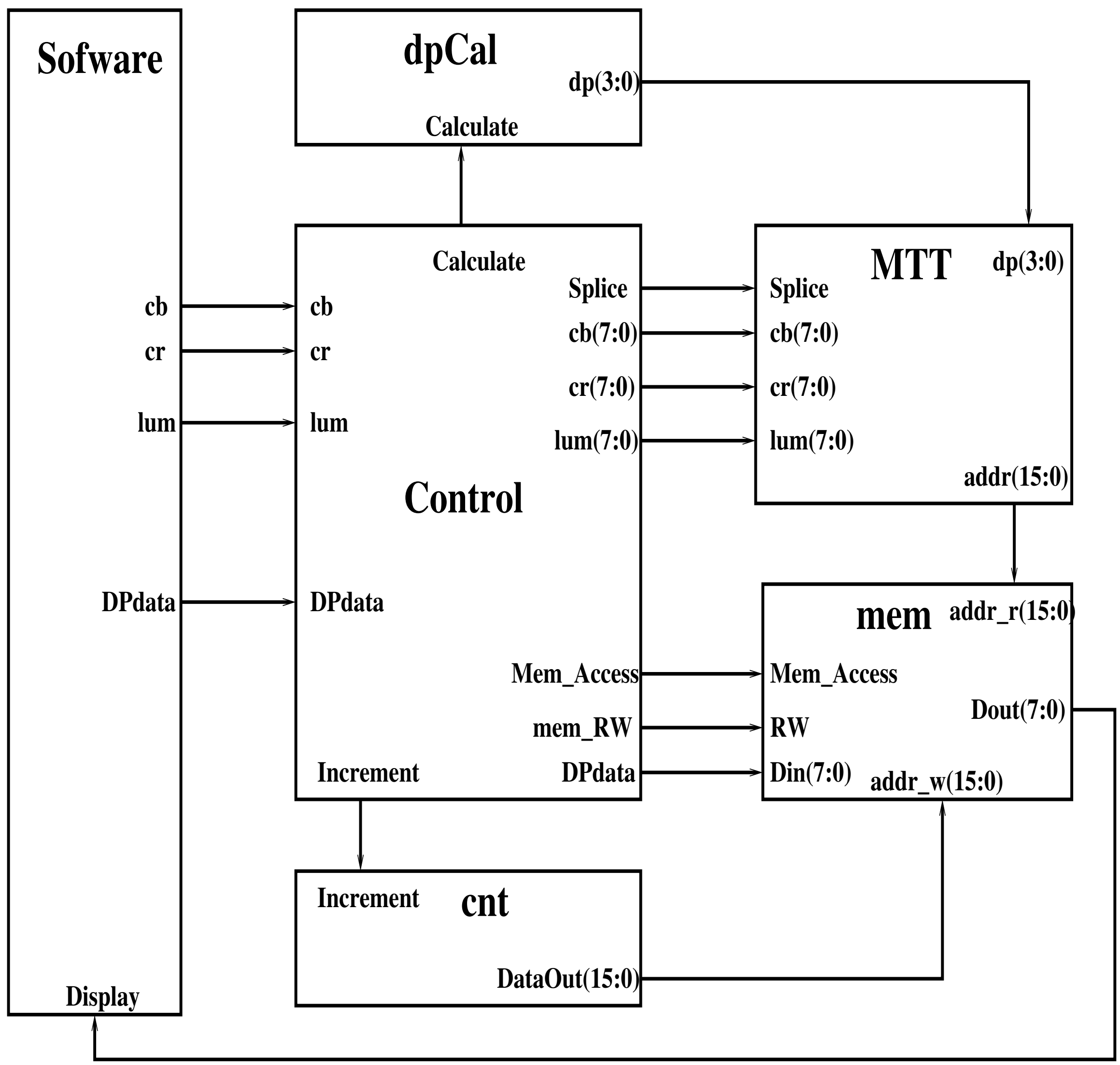

Figure 4.5. Block diagram for ditherer of an MPEG decoder.

```
entity ctrl is
  port(
    DPdata : inout channel := init_channel(receiver => timing(1, 2));
    cr, cb, lum : inout channel := init_channel(receiver => timing(1, 2));
    Increment : inout channel := init_channel(sender => timing(1, 3));
    Mem_Access : inout channel := init_channel(sender => timing(3, 5));
    Calculate : inout channel := init_channel(sender => timing(3, 5));
    Splice : inout channel := init_channel(sender => timing(3, 5));
    cr_d : buffer std_logic_vector(7 downto 0);
    cb_d : buffer std_logic_vector(7 downto 0);
    lum_d : buffer std_logic_vector(7 downto 0);
    DPdata_d : buffer std_logic_vector(7 downto 0);
    reset : out std_logic := '0';
    mem_RW : buffer std_logic := '0'
    );
end ctrl;
architecture ctrl of ctrl is
begin
  Decoder : process
  begin
    await_any (DPdata, cb, cr, lum);
    if (probe (DPdata))then
      vassign(mem_RW, '0', 1, 2);
      receive(DPdata, DPdata_d);
      send(Mem_Access);
      send(Increment);
    elsif (probe (cb)) then
      receive(cb, cb_d);
    elsif (probe (cr)) then
      receive(cr, cr_d);
    elsif (probe(lum))then
      vassign(mem_RW, '1', 1, 3);
      receive(lum, lum_d);
      send(Calculate);
      send(Splice);
      send(Mem_Access);
    end if;
    wait for delay(3, 5);
  end process;
end ctrl;
```

After going through our tools and using WebPack, the simulation result is shown in Figure 4.6.

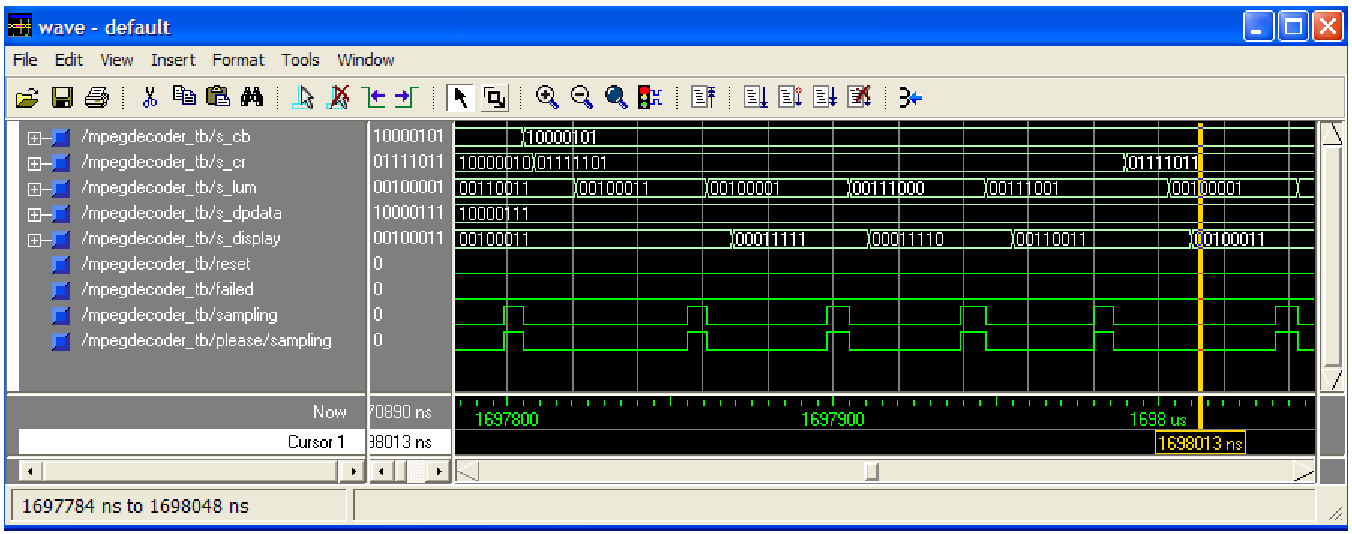

Figure 4.6. Simulation result for MPEG ditherer.

CHAPTER 5

CONCLUSION AND FUTURE WORK

This chapter summarizes the work and describes related future work that is not addressed in this thesis. This thesis presents a methodology by using synchronous synthesis tools for high-level asynchronous circuit design and develops a design flow from an asynchronous model in a high-level description using communication channels and leverages synchronous tools whenever possible. Starting from the channel level model, the *Ch2Hs* script translates the channel-level model into the handshaking-level model. The approach to this step is first to reformat the channel-level VHDL code into the way it suits the script, the second step is to extract the information out of the code and save it into a local and global lookup table for later use, and then translate the channel-level VHDL code into handshaking-level VHDL code by using all the information.

Next, the *DpSyn* script transforms the handshaking-level VHDL we generated from the *Ch2Hs* script into synthesizable VHDL. The input of this script is a 4-phase bundled-data handshaking model with both passive and active modes, and the outputs are *dp_reference-number.vhd* which is a self-timed datapath VHDL file for WebPack synthesis, *dp_reference-number_md.vhd* which is a matched delay of the datapath *dp_reference-number.vhd* using static timing analysis, *delay_filename.dat* which is the timing delay report, *filename_sim.vhd* which is the corresponding controller for VHDL simulation, *filename_atacs_t.vhd* which is the corresponding controller template for ATACS synthesis, and *filename_wp.vhd* which is the corresponding controller for WebPack synthesis.

Finally, the *CtrlSyn* script presented in this thesis focuses on controller synthesis by using the output files from the *DpSyn* method, which is based on the timing delay report to generate matched delay elements with inverter chains for all the datapath files and also to replace the timing information in the *filename_atacs_t.vhd* file for ATACS synthesis. The output files from this method are bundling constraint files *delay_reference_number*, and the handshaking-level controller *filename_atacs.vhd* for ATACS to synthesize.

In addition, after synthesizing the controller by ATACS and all the datapath by WebPack, this thesis combines the result and simulates and synthesizes it using WebPack to validate the design. The synthesized circuit is tested using LFSRs and signature analyzers after implementing it onto the FPGA.

Furthermore, after running through a particular example successfully, this thesis also applies more case studies using our methods. Working with more examples, we also improved our tool in order to work not only for serial send and receive statements, but also parallel *send* and *receive* statements, *probe*, *if*, and *else* statements, etc. The reason we have chosen these three examples as case studies is that they all have their specified statements we can apply our scripts on in order to make them work better with all the cases. For instance, adder has parallel *send*, parallel *receive* statements and also bit-slicing statement; shifter has *await_any* statement, *if*, and *else* statements, and send or receive statement has no data signal in it; ditherer of MPEG decoder is a bigger example has all of the specification above, and also has *vassign* statement. This enables the tool to work successfully on more complicated examples and more realistic examples.

During this thesis work, we have faced several difficulties, such as how to analyze the delay timing report in order to calculate the maximum matched delay for each datapath, how to create the synthesis inverter chains for bundling constraints, how to modify the tool in order to fit more general examples, etc. The methods presented in this thesis are all adjustable scripts, so we can always modify them accordingly for different examples.

Besides all of these usages, the tool has its limitation too, such as that if *guard_and* and *guard_or* appear in the VHDL file at the same time, the script will not work; the second limitation is if the controller is too big, then ATACS would have a hard time synthesizing it: there are too many states explorsion; the third limitation is the design hardard-free, and how this can be determined. If it is not harzard-free, what can be done to the design flow to make it harzard-free.

5.1 Future Work

This section presents related future work for this thesis. First, we would like to completely automate the whole design flow, because there are two steps in the design flow that are not automated. One is to analyze the synthesized time delay report manually, the other one is to import datapath files into WebPack and controller files into ATACS manually. Thus, more research for automation is needed on parsing and analyzing the reports and synthesizing files under different operating systems in order to simplify and improve the design flow.

More development is needed to improve our methods, so they can be applied in more complicated examples, more realistic examples, and more general examples. The tool described in this thesis can handle all of the instructions of our channel package and handshaking package specifications, but there are still some limitations that need to be translated properly, which require more research, especially for complicated examples in which it is difficult to track the errors.

In addition, more work needs to be done on implementation for the hardware. In this thesis, it verifies the synthesized result, which comes out of WebPack, and then puts it into the testing circuit which is constructed using LFSRs and signature analyzers and simulated again. After simulating it successfully, we could implement it onto a FPGA to see if it actually works. Due to time constraints, we did not implement the circuit onto an FPGA. However, it is very interesting and critical to see how this thesis contributes in the real world.

Finally, we can also work more on the testing step. This thesis chooses a signature analyzer to test the result, which sometimes causes some loss of effective fault coverage due to aliasing. Therefore, more research on testing can be done in order to make the circuit more accurate.

APPENDIX A

WEBPACK SYNTHESIS REPORT OF SHOP EXAMPLE

```
Release 6.1.03i - xst G.26
Copyright (c) 1995-2003 Xilinx, Inc.  All rights reserved.
--> Parameter TMPDIR set to __projnav
CPU : 0.00 / 0.39 s | Elapsed :  0.00 / 1.00 s
--> Parameter xsthdpdir set to ./xst
CPU : 0.01 / 0.40 s | Elapsed :  0.00 / 1.00 s
--> Reading design:  shop.prj
TABLE OF CONTENTS
  1) Synthesis Options Summary
  2) HDL Compilation
  3) HDL Analysis
  4) HDL Synthesis
     4.1) HDL Synthesis Report
  5) Advanced HDL Synthesis
  6) Low Level Synthesis
  7) Final Report
     7.1) Device utilization summary
     7.2) TIMING REPORT
=========================================================================
*                      Synthesis Options Summary                        *
=========================================================================
---- Source Parameters
Input File Name                  :  shop.prj
Input Format                     :  mixed
Ignore Synthesis Constraint File :  NO
Verilog Include Directory        :
---- Target Parameters
Output File Name                 :  shop
Output Format                    :  NGC
Target Device                    :  xc2s100-6-tq144
---- Source Options
Top Module Name                  :  shop
Automatic FSM Extraction         :  YES
FSM Encoding Algorithm           :  Auto
FSM Style                        :  lut
RAM Extraction                   :  Yes
RAM Style                        :  Auto
ROM Extraction                   :  Yes
ROM Style                        :  Auto
```

```
Mux Extraction                     :  YES
Mux Style                          :  Auto
Decoder Extraction                 :  YES
Priority Encoder Extraction        :  YES
Shift Register Extraction          :  YES
Logical Shifter Extraction         :  YES
XOR Collapsing                     :  YES
Resource Sharing                   :  YES
Multiplier Style                   :  lut
Automatic Register Balancing       :  No
---- Target Options
Add IO Buffers                     :  YES
Global Maximum Fanout              :  100
Add Generic Clock Buffer(BUFG)     : 4
Register Duplication               :  YES
Equivalent register Removal        :  YES
Slice Packing                      :  YES
Pack IO Registers into IOBs        :  auto
---- General Options
Optimization Goal                  :  Speed
Optimization Effort                :  1
Keep Hierarchy                     :  NO
Global Optimization                :  AllClockNets
RTL Output                         :  Yes
Write Timing Constraints           :  NO
Hierarchy Separator                :  _
Bus Delimiter                      :  <>
Case Specifier                     :  maintain
Slice Utilization Ratio            :  100
Slice Utilization Ratio Delta      :  5
---- Other Options
lso                                :  shop.lso
Read Cores                         :  YES
cross_clock_analysis              :  NO
verilog2001                        :  YES
Optimize Instantiated Primitives   :  NO
=========================================================================
=========================================================================
*                          HDL Compilation                              *
=========================================================================
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/delay_dp_1.vhd inLibrary work.
Architecture structure of Entity delay_dp_1 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/delay_dp_0.vhd in Library work.
Architecture structure of Entity delay_dp_0 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/a1.vhd in Library work.
Architecture behavior of Entity a1 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/n2p2.vhd in Library work.
Architecture behavior of Entity n2p2 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/n4p2.vhd in Library work.
Architecture behavior of Entity n4p2 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/n1p1h.vhd in Library work.
Architecture behavior of Entity n1p1h is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/THE_SHOPS.vhd in Library work.
```

```
Architecture structure of Entity shop_ctrl is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/dp_0.vhd in Library work.
Architecture dp4syn of Entity dp_0 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/dp_1.vhd in Library work.
Architecture dp4syn of Entity dp_1 is up to date.
Compiling vhdl file C:/xilinxwork/shop_goodsyn0/shop_wp.vhd in Library work.
Architecture arch_wp of Entity shop is up to date.
=========================================================================
*                           HDL Analysis                                *
=========================================================================
Analyzing Entity <shop> (Architecture <arch_wp>).
Entity <shop> analyzed.  Unit <shop> generated.
Analyzing Entity <shop_ctrl> (Architecture <structure>).
Entity <shop_ctrl> analyzed.  Unit <shop_ctrl> generated.
Analyzing Entity <a1> (Architecture <behavior>).
Entity <a1> analyzed.  Unit <a1> generated.
Analyzing Entity <n2p2> (Architecture <behavior>).
Entity <n2p2> analyzed.  Unit <n2p2> generated.
Analyzing Entity <n4p2> (Architecture <behavior>).
Entity <n4p2> analyzed.  Unit <n4p2> generated.
Analyzing Entity <n1p1h> (Architecture <behavior>).
Entity <n1p1h> analyzed.  Unit <n1p1h> generated.
Analyzing Entity <dp_0> (Architecture <dp4syn>).
Entity <dp_0> analyzed.  Unit <dp_0> generated.
Analyzing Entity <delay_dp_0> (Architecture <structure>).
Entity <delay_dp_0> analyzed.  Unit <delay_dp_0> generated.
Analyzing Entity <dp_1> (Architecture <dp4syn>).
Entity <dp_1> analyzed.  Unit <dp_1> generated.
Analyzing Entity <delay_dp_1> (Architecture <structure>).
Entity <delay_dp_1> analyzed.  Unit <delay_dp_1> generated.
=========================================================================
*                           HDL Synthesis                               *
=========================================================================
Synthesizing Unit <delay_dp_1>.
  Related source file is C:/xilinxwork/shop_goodsyn0/delay_dp_1.vhd.
WARNING:Xst:1780 - Signal <x0> is never used or assigned.
  Unit <delay_dp_1> synthesized.
Synthesizing Unit <delay_dp_0>.
  Related source file is C:/xilinxwork/shop_goodsyn0/delay_dp_0.vhd.
WARNING:Xst:1780 - Signal <x0> is never used or assigned.
  Unit <delay_dp_0> synthesized.
Synthesizing Unit <n1p1h>.
  Related source file is C:/xilinxwork/shop_goodsyn0/n1p1h.vhd.
WARNING:Xst:1778 - Inout <o> is assigned but never used.
WARNING:Xst:1778 - Inout <obar> is assigned but never used.
WARNING:Xst:737 - Found 1-bit latch for signal <o_i>.
Unit <n1p1h> synthesized.
Synthesizing Unit <n4p2>.
  Related source file is C:/xilinxwork/shop_goodsyn0/n4p2.vhd.
WARNING:Xst:1778 - Inout <o> is assigned but never used.
WARNING:Xst:1778 - Inout <obar> is assigned but never used.
WARNING:Xst:737 - Found 1-bit latch for signal <o_i>.
Unit <n4p2> synthesized.
```

```
Synthesizing Unit <n2p2>.
  Related source file is C:/xilinxwork/shop_goodsyn0/n2p2.vhd.
WARNING:Xst:1778 - Inout <o> is assigned but never used.
WARNING:Xst:1778 - Inout <obar> is assigned but never used.
WARNING:Xst:737 - Found 1-bit latch for signal <o_i>.
Unit <n2p2> synthesized.
Synthesizing Unit <a1>.
  Related source file is C:/xilinxwork/shop_goodsyn0/a1.vhd.
WARNING:Xst:1778 - Inout <o> is assigned but never used.
Unit <a1> synthesized.
Synthesizing Unit <dp_1>.
  Related source file is C:/xilinxwork/shop_goodsyn0/dp_1.vhd.
WARNING:Xst:1778 - Inout <shelf1> is assigned but never used.
    Found 3-bit adder for signal <shelf1>.
    Summary:
  inferred   1 Adder/Subtracter(s).
Unit <dp_1> synthesized.
Synthesizing Unit <dp_0>.
  Related source file is C:/xilinxwork/shop_goodsyn0/dp_0.vhd.
WARNING:Xst:737 - Found 3-bit latch for signal <shelf>.
Unit <dp_0> synthesized.
Synthesizing Unit <shop_ctrl>.
  Related source file is C:/xilinxwork/shop_goodsyn0/THE_SHOPS.vhd.
WARNING:Xst:1780 - Signal <wine_delivery_snd_i_bar> is never used or
assigned.
Unit <shop_ctrl> synthesized.
Synthesizing Unit <shop>.
  Related source file is C:/xilinxwork/shop_goodsyn0/shop_wp.vhd.
WARNING:Xst:1778 - Inout <wine_selling_snd> is assigned but never used.
Unit <shop> synthesized.
=========================================================================
HDL Synthesis Report
Macro Statistics
  Latches                          :  4
  1-bit latch                      :  3
  3-bit latch                      :  1
  Adders/Subtractors               :  1
  3-bit adder                      :  1
=========================================================================
=========================================================================
*                       Advanced HDL Synthesis                          *
=========================================================================
=========================================================================
*                         Low Level Synthesis                           *
=========================================================================
Optimizing unit <shop> ...
Loading device for application Xst from file 'v100.nph' in environment
C:/Xilinx.
Mapping all equations...
Building and optimizing final netlist ...
Found area constraint ratio of 100 (+ 5) on block shop, actual ratio is 0.
Latch shop_ctrl_inst_i_2_o_i has been replicated 1 time(s) to handle
iob=true attribute.
```

```
Latch shop_ctrl_inst_i_3_o_i has been replicated 1 time(s) to handle
iob=true attribute.
============================================================
*                          Final Report            *
=============================================================
Final Results
RTL Top Level Output File Name     :  shop.ngr
Top Level Output File Name         :  shop
Output Format                      :  NGC
Optimization Goal                  :  Speed
Keep Hierarchy                     :  NO
Design Statistics
  IOs                        :  10
Cell Usage :
  BELS                       :  9
  GND                        :  1
  LUT1                       :  4
  LUT2                       :  2
  LUT3                       :  1
  LUT4                       :  1
  FlipFlops/Latches          :  8
  LD                         :  3
  LDP_1                      :  1
  LDPE_1                     :  4
  Clock Buffers              :  1
  BUFGP                      :  1
  IO Buffers                 :  9
  IBUF                       :  4
  OBUF                       :  5
=========================================================================
Device utilization summary:
---------------------------
Selected Device :  2s100tq144-6
  Number of Slices:                  8  out of   1200     0%
  Number of Slice Flip Flops:        8  out of   2400     0%
  Number of 4 input LUTs:            8  out of   2400     0%
  Number of bonded IOBs:           9  out of     96       9%
  Number of GCLKs:                 1  out of      4       25%
=========================================================================
TIMING REPORT
NOTE: THESE TIMING NUMBERS ARE ONLY A SYNTHESIS ESTIMATE.
      FOR ACCURATE TIMING INFORMATION PLEASE REFER TO THE TRACE REPORT
      GENERATED AFTER PLACE-and-ROUTE.
Clock Information:
------------------
-----------------------------------+------------------------+-------+-
Clock Signal                       | Clock buffer(FF name)  | Load  |
-----------------------------------+------------------------+-------+-
shop_ctrl_inst_i_3_o_i:Q          | NONE                    | 1     |
wine_delivery_snd                 | BUFGP                   | 3     |
shop_ctrl_inst_i_4_o_i:Q          | NONE                    | 4     |
-----------------------------------+------------------------+-------+
Timing Summary:
```

```
---------------
Speed Grade: -6
   Minimum period: No path found
   Minimum input arrival time before clock: 4.864ns
   Maximum output required time after clock: 8.778ns
   Maximum combinational path delay: No path found
Timing Detail:
--------------
All values displayed in nanoseconds (ns)
=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'wine_delivery_snd'
  Offset:              2.520ns (Levels of Logic = 1)
  Source:            wine_delivery_data<1> (PAD)
  Destination:        dp_0_inst_shelf_1 (LATCH)
  Destination Clock:  wine_delivery_snd falling
  Data Path:  wine_delivery_data<1> to dp_0_inst_shelf_1
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     IBUF:I->O             1   0.776   1.035  wine_delivery_data_1_IBUF
                                              (wine_delivery_data_1_IBUF)
     LD:D                      0.709          dp_0_inst_shelf_1
    ----------------------------------------
    Total                      2.520ns (1.485ns logic, 1.035ns route)
                                       (58.9% logic, 41.1% route)
=========================================================================
Timing constraint: Default OFFSET IN BEFORE for Clock 'shop_ctrl_inst_i
_4_o_i:Q'
  Offset:              4.864ns (Levels of Logic = 2)
  Source:            wine_delivery_snd (PAD)
  Destination:        shop_ctrl_inst_i_3_o_i (LATCH)
  Destination Clock:  shop_ctrl_inst_i_4_o_i:Q rising
  Data Path:  wine_delivery_snd to shop_ctrl_inst_i_3_o_i
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     BUFGP:I->O            5   0.657   1.566  wine_delivery_snd_BUFGP
                                             (wine_delivery_snd_BUFGP)
     LUT1:I0->O            2   0.549   1.206  shop_ctrl_inst_i_3_o_i__n00011
                                            (shop_ctrl_inst_i_3_o_i__n0001)
     LDPE_1:GE                0.886           shop_ctrl_inst_i_3_o_i
    ----------------------------------------
    Total                      4.864ns (2.092ns logic, 2.772ns route)
                                       (43.0% logic, 57.0% route)
=========================================================================
Timing constraint: Default OFFSET OUT AFTER for Clock 'shop_ctrl_inst_i
_4_o_i:Q'
  Offset:              6.897ns (Levels of Logic = 1)
  Source:            shop_ctrl_inst_i_2_o_i_1 (LATCH)
  Destination:        wine_selling_snd (PAD)
  Source Clock:      shop_ctrl_inst_i_4_o_i:Q falling
  Data Path:  shop_ctrl_inst_i_2_o_i_1 to wine_selling_snd
                                Gate     Net
```

```
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LDPE_1:G->Q           1   1.194   1.035  shop_ctrl_inst_i
                      _2_o_i_1 (shop_ctrl_inst_i_2_o_i_1)
     OBUF:I->O                 4.668          wine_selling_snd_OBUF
                                              (wine_selling_snd)
    ----------------------------------------
    Total                      6.897ns (5.862ns logic, 1.035ns route)
                                       (85.0% logic, 15.0% route)
=========================================================================
Timing constraint:  Default OFFSET OUT AFTER for Clock 'wine_delivery_snd'
  Offset:              8.778ns (Levels of Logic = 2)
  Source:            dp_0_inst_shelf_0 (LATCH)
  Destination:        wine_selling_data<0> (PAD)
  Source Clock:      wine_delivery_snd falling
  Data Path:  dp_0_inst_shelf_0 to wine_selling_data<0>
                                Gate     Net
    Cell:in->out      fanout   Delay   Delay  Logical Name (Net Name)
    ----------------------------------------  ------------
     LD:G->Q               3   1.194   1.332  dp_0_inst_shelf_0
                                              (dp_0_inst_shelf_0)
     LUT3:I1->O            1   0.549   1.035  dp_1_inst_Madd_shelf1_Mxor
                          _Result<2>_Result1 (wine_selling_data_2_OBUF)
     OBUF:I->O                 4.668          wine_selling_data_2_OBUF
                                              (wine_selling_data<2>)
    ----------------------------------------
    Total                      8.778ns (6.411ns logic, 2.367ns route)
                                       (73.0% logic, 27.0% route)
=========================================================================
CPU : 2.56 / 3.22 s | Elapsed :  2.00 / 3.00 s
-->
Total memory usage is 57020 kilobytes
```

APPENDIX B

VHDL CODE FOR TESTING CIRCUIT OF SHOP

```
----------------------------------------------
-- LFSR random number generator
----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
entity lfsr is
  generic
  (
    DATA_WIDTH: natural := 3
  );
  port
  (
    resetb:  in std_logic;
    req:  in std_logic;
    ack:  inout std_logic:='0';
    dout:  out std_logic_vector(DATA_WIDTH-1 downto 0):="111"
  );
end lfsr;
architecture arch_lfsr of lfsr is
  signal r :  std_logic_vector (dout'length-1 downto 0);
  signal new_bit :  std_logic;
  signal ack_int :  std_logic;
   component delay_lfsr
      port ( x  :  in  std_logic;
             xd :  out   std_logic);
   end component;
begin
  new_bit <= r(2)  xor r(1);
  update_shift_register:
  process(req, resetb)
  begin
    if resetb = '0' then
      r <= (others=>'1');
    elsif req'EVENT and req='1' then
      r <= r(r'length-2 downto 0) & new_bit;
    end if;
  end process;
  dout <= r;
```

```
delay_inst : delay_lfsr
  port map (x=>req, xd=>ack_int);
      ack <= (resetb and ack_int);
end arch_lfsr;

-----------------------------------------------
-- LFSR based signature analyzer
-----------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity signa is
  generic
  (
    DATA_WIDTH: natural := 3
  );
  port
  (
    resetb:  in std_logic;
    din:  in std_logic_vector(DATA_WIDTH-1 downto 0);
    req:  in std_logic;
    ack:  inout std_logic:= '0';
    dout:  out std_logic_vector(DATA_WIDTH-1 downto 0):= "111"
  );
end signa;
architecture arch_signa of signa is
  signal r :  std_logic_vector (dout'length-1 downto 0);
  signal new_bit :  std_logic_vector (dout'length-1 downto 0);
  signal ack_int :  std_logic;
   component delay_lfsr
      port ( x  :  in  std_logic;
             xd :  out   std_logic);
   end component;
begin
  new_bit(0) <=  r(2)  xor r(1) xor din(0);
  new_bit(1) <=  r(0)  xor din(1);
  new_bit(2) <=  r(1)  xor din(2);
  update_shift_register:
  process(req, resetb)
  begin
    if resetb = '0' then
      r <= (others=>'1');
    elsif req'EVENT and req='1' then
      r <= new_bit;
    end if;
  end process;
  dout <= r;
delay_inst : delay_lfsr
  port map (x=>req, xd=>ack_int);
  ack <= resetb and ack_int;
end arch_signa;
```

```
-------------------------------------------------
-- counter
------------------------------------------------
-- resettale counter
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity counter is
  port
  (
    resetb:  in std_logic;
    req:    in std_logic;
    cout:    out std_logic:= '0'
  );
end counter;
architecture arch_counter of counter is
  signal count :  std_logic_vector (3 downto 0);
  signal r:std_logic;
begin
    update_count_register:
  process(req, resetb)
  begin
    if resetb = '0' then
      count <= (others=>'0');
      r <= '0';
    elsif req'EVENT and req='1' then
      if count < 15    then
        count <= count + 1 ;
        r <= '0';
      else
        count <= count;
        r <= '1';
      end if;
    end if;
  end process;
  cout <= r;
end arch_counter;

------------------------------------------------
--signature data error checker for wine shop
------------------------------------------------
--signature data error checker for wine shop
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
entity result is
  port
  (
    done_in:  in std_logic;
```

```
    din:    in std_logic_vector(2 downto 0);
    status:  out std_logic_vector(1 downto 0)
);
end result;
architecture arch_result of result is
  signal result_compare:  std_logic_vector (2 downto 0);
  signal result_ok:  std_logic;
  constant sign_right:  std_logic_vector (2 downto 0) :="011";
begin
  result_compare <= din xnor sign_right);
  result_ok <= result_compare(2) and result_compare(1) and result_compare(0);
  status(0) <= done_in;
  status(1) <= result_ok and done_in;
end arch_result;
```

```
---------------------------------------------------------
-- wine_example2.vhd
---------------------------------------------------------
-- wine_example2.vhd
library ieee;
use ieee.std_logic_1164.all;
--use work.nondeterminism.all;
--use work.handshake.all;
entity wine_example is
end wine_example;
architecture structure of wine_example is
  component lfsr
  generic
  (
    DATA_WIDTH: natural := 3
  );
  port
  (
    resetb:  in std_logic;
    req:    in std_logic;
    ack:    inout std_logic:='0';
    dout:    out std_logic_vector(DATA_WIDTH-1 downto 0):="111"
  );
end component;
  component shop
  port (
    wine_delivery_rv :  inout std_logic;
    wine_delivery_snd :  in std_logic;
    wine_delivery_data :  in std_logic_vector(2 downto
0);
    wine_selling_rv :  in std_logic;
    wine_selling_snd :  inout std_logic;
    wine_selling_data :  inout std_logic_vector(2 downto 0)
  );
```

```
  end component;
component signa
  generic
  (
    DATA_WIDTH: natural := 3
  );
  port
  (
    resetb:  in std_logic;
    din:    in std_logic_vector(DATA_WIDTH-1 downto 0);
    req:    in std_logic;
    ack:    inout std_logic:='0';
    dout:    out std_logic_vector(DATA_WIDTH-1 downto 0):="111"
  );
end component;
component counter
  port
  (
    resetb:  in std_logic;
    req:    in std_logic;
    cout:    out std_logic:='0'
  );
end component;
component result
  port
  (
    done_in:  in std_logic;
    din:    in std_logic_vector(2 downto 0);
    status:  out std_logic_vector(1 downto 0)
);
end component;
  --signal WineryShop:channel:=init_channel;
  signal WineryShop_rv :  std_logic;
  signal WineryShop_rvb :  std_logic;
  signal WineryShop_snd :  std_logic;
  signal WineryShop_data :  std_logic_vector(2 downto 0);
  --signal ShopPatron:channel:=init_channel;
  signal ShopPatron_rv :  std_logic;
  signal ShopPatron_snd :  std_logic;
  signal ShopPatron_data :  std_logic_vector(2 downto 0);
  signal Signature_data  :  std_logic_vector(2 downto 0);
  signal resetb:  std_logic;
  signal ShopPatron_datac :  std_logic_vector(2 downto 0);
  signal cout_counter:  std_logic:='0';
  signal coutb_counter:  std_logic:='1';
  signal ack_signa:  std_logic:='0';
  signal Result_status  :  std_logic_vector(1 downto 0);
```

```
begin
    resetb <= '1', '0' after 5 ns, '1' after 50 ns;
  THE_WINERY : lfsr
  port map(
    resetb => resetb,
    req => WineryShop_rvb,
    ack => WineryShop_snd,
    dout => WineryShop_data
  );
WineryShop_rvb <= not WineryShop_rv after 10 ns;
  THE_SHOP : shop
  port map(
    wine_delivery_rv =>WineryShop_rv,
    wine_delivery_snd =>WineryShop_snd,
    wine_delivery_data =>WineryShop_data,
    wine_selling_rv =>ShopPatron_rv,
    wine_selling_snd =>ShopPatron_snd,
    wine_selling_data =>ShopPatron_data
  );
ShopPatron_datac <= ShopPatron_data;
  THE_PATRON : signa
  port map(
    resetb => resetb,
    din => ShopPatron_datac,
    req => ShopPatron_snd,
    ack => ack_signa,
    dout => Signature_data
  );
  THE_COUNTER : counter
  port map(
    resetb => resetb,
    req => ShopPatron_snd,
    cout => cout_counter
  );
coutb_counter <= not cout_counter;
ShopPatron_rv <= ack_signa and coutb_counter;
  THE_RESULT : result
  port map(
    done_in => cout_counter,
    din => Signature_data,
    status => Result_status
  );
end structure;
```

APPENDIX C

VHDL CODE FOR ADDER

```
-----------------------------------------------
--adder_env.vhd
-----------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity adder_env is
  port(X:inout channel:=init_channel;
       Y:inout channel:=init_channel;
       Cin:inout channel:=init_channel;
       Sum:inout channel:=init_channel;
       Cout:inout channel:=init_channel);
end entity adder_env;
architecture behavior  of adder_env is
  signal a:std_logic_vector(3 downto 0);
  signal b:std_logic_vector(3 downto 0);
  signal c:std_logic_vector(0 downto 0);
  signal s:std_logic_vector(3 downto 0);
  signal d:std_logic_vector(0 downto 0);
  signal t:std_logic_vector(4 downto 0);
begin
process
begin
  a <= selection(16,4);
  b <= selection(16,4);
  c <= selection(1,1);
  wait for 5 ns;
  send(X,a,Y,b,Cin,c);
  receive(Sum,s,Cout,d);
  t <= ("0" & a) + ("0" & b) + ("0000" & c);
  wait for 5 ns;
  assert (s=t(3 downto 0))
    report "wrong answer"
```

```
    severity warning;
  assert (d=t(4 downto 4))
    report "wrong answer"
    severity warning;
end process;
end behavior;
-------------------------------------------------
--adder_top.vhd
-------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.channel.all;
entity adder_top is
end entity adder_top;
architecture structure of adder_top is
  component adder
    port(X:inout channel:=init_channel;
         Y:inout channel:=init_channel;
         Cin:inout channel:=init_channel;
         Sum:inout channel:=init_channel;
         Cout:inout channel:=init_channel);
  end component;
  component adder_env
    port(X:inout channel:=init_channel;
         Y:inout channel:=init_channel;
         Cin:inout channel:=init_channel;
         Sum:inout channel:=init_channel;
         Cout:inout channel:=init_channel);
  end component;
  signal X:channel:=init_channel;
  signal Y:channel:=init_channel;
  signal Cin:channel:=init_channel;
  signal Sum:channel:=init_channel;
  signal Cout:channel:=init_channel;
begin
  THE_ADDER:adder
    port map(X => X,Y => Y,Cin=> Cin, Sum => Sum, Cout => Cout);
  THE_ADDER_ENV:adder_env
    port map(X => X,Y => Y,Cin=> Cin, Sum => Sum, Cout => Cout);
end structure;
```

APPENDIX D

VHDL CODE FOR SHIFTER

```
----------------------
-- shifter_env.vhd
----------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.handshake.all;
entity shifter_env is
end shifter_env;
architecture structure of shifter_env is
  component reg_bits
  port (
    load_data : in std_logic;
    load_rv : inout std_logic;
    load_snd : in std_logic;
    shift_in_data : in std_logic;
    shift_in_rv : inout std_logic;
    shift_in_snd : in std_logic;
    shift_out_data : out std_logic;
    shift_out_rv : in std_logic;
    shift_out_snd : inout std_logic;
    done_in_rv : inout std_logic;
    done_in_snd : in std_logic;
    done_out_rv : in std_logic;
    done_out_snd : inout std_logic;
    output_data : out std_logic;
    output_rv : in std_logic;
    output_snd : inout std_logic
  );
  end component;
  component reg_lsbit
  port (
    load_data : in std_logic;
    load_rv : inout std_logic;
    load_snd : in std_logic;
    shift_in_data : in std_logic;
    shift_in_rv : inout std_logic;
    shift_in_snd : in std_logic;
    done_in_rv : inout std_logic;
```

```
  done_in_snd :  in std_logic;
  output_data :  out std_logic;
  output_rv :  in std_logic;
  output_snd :  inout std_logic
);
end component;
--signal Shift_in:channel:=init_channel;
signal Shift_in_dataX : std_logic;
signal Shift_in_rvX : std_logic;
signal Shift_in_sndX : std_logic;
--signal Done_in:channel:=init_channel;
signal Done_in_rvX : std_logic;
signal Done_in_sndX : std_logic;
--signal La:channel:=init_channel;
signal La_data :  std_logic;
signal La_rv :  std_logic;
signal La_snd :  std_logic;
--signal Lb:channel:=init_channel;
signal Lb_data :  std_logic;
signal Lb_rv :  std_logic;
signal Lb_snd :  std_logic;
--signal Lc:channel:=init_channel;
signal Lc_data :  std_logic;
signal Lc_rv :  std_logic;
signal Lc_snd :  std_logic;
--signal Ld:channel:=init_channel;
signal Ld_data :  std_logic;
signal Ld_rv :  std_logic;
signal Ld_snd :  std_logic;
--signal Oa:channel:=init_channel;
signal Oa_data :  std_logic;
signal Oa_rv :  std_logic;
signal Oa_snd :  std_logic ;
--signal Ob:channel:=init_channel;
signal Ob_data :  std_logic;
signal Ob_rv :  std_logic;
signal Ob_snd :  std_logic;
--signal Oc:channel:=init_channel;
signal Oc_data :  std_logic;
signal Oc_rv :  std_logic;
signal Oc_snd :  std_logic;
--signal Od:channel:=init_channel;
signal Od_data :  std_logic;
signal Od_rv :  std_logic;
signal Od_snd :  std_logic;
--signal Sab:channel:=init_channel;
signal Sab_data :  std_logic;
signal Sab_rv :  std_logic;
signal Sab_snd :  std_logic;
--signal Dab:channel:=init_channel;
signal Dab_rv :  std_logic;
signal Dab_snd :  std_logic;
--signal Sbc:channel:=init_channel;
```

```
  signal Sbc_data :  std_logic;
  signal Sbc_rv :  std_logic;
  signal Sbc_snd :  std_logic;
  --signal Dbc:channel:=init_channel;
  signal Dbc_rv :  std_logic;
  signal Dbc_snd :  std_logic;
  --signal Scd:channel:=init_channel;
  signal Scd_data :  std_logic;
  signal Scd_rv :  std_logic;
  signal Scd_snd :  std_logic;
  --signal Dcd:channel:=init_channel;
  signal Dcd_rv :  std_logic;
  signal Dcd_snd :  std_logic;
--@synthesis_off
  signal Youtput:std_logic_vector(3 downto 0);
  signal yinput:std_logic_vector(3 downto 0):="1001";
  signal nextbit:std_logic_vector(0 downto 0);
--@synthesis_on
begin
    shifterA : reg_bits
  port map(
    Load_data =>La_data,
    Load_rv =>La_rv,
    Load_snd =>La_snd,
    Shift_in_data =>Shift_in_dataX,
    Shift_in_rv =>Shift_in_rvX,
    Shift_in_snd =>Shift_in_sndX,
    Done_in_rv =>Done_in_rvX,
    Done_in_snd =>Done_in_sndX,
    Shift_out_data =>Sab_data,
    Shift_out_rv =>Sab_rv,
    Shift_out_snd =>Sab_snd,
    Done_out_rv =>Dab_rv,
    Done_out_snd =>Dab_snd,
    Output_data =>Oa_data,
    Output_rv =>Oa_rv,
    Output_snd =>Oa_snd
  );
  shifterB : reg_bits
  port map(
    Load_data =>Lb_data,
    Load_rv =>Lb_rv,
    Load_snd =>Lb_snd,
    Shift_in_data =>Sab_data,
    Shift_in_rv =>Sab_rv,
    Shift_in_snd =>Sab_snd,
    Done_in_rv =>Dab_rv,
    Done_in_snd =>Dab_snd,
    Shift_out_data =>Sbc_data,
    Shift_out_rv =>Sbc_rv,
    Shift_out_snd =>Sbc_snd,
    Done_out_rv =>Dbc_rv,
    Done_out_snd =>Dbc_snd,
```

```
    Output_data =>Ob_data,
    Output_rv =>Ob_rv,
    Output_snd =>Ob_snd
  );
  shifterC : reg_bits
  port map(
    Load_data =>Lc_data,
    Load_rv =>Lc_rv,
    Load_snd =>Lc_snd,
    Shift_in_data =>Sbc_data,
    Shift_in_rv =>Sbc_rv,
    Shift_in_snd =>Sbc_snd,
    Done_in_rv =>Dbc_rv,
    Done_in_snd =>Dbc_snd,
    Shift_out_data =>Scd_data,
    Shift_out_rv =>Scd_rv,
    Shift_out_snd =>Scd_snd,
    Done_out_rv =>Dcd_rv,
    Done_out_snd =>Dcd_snd,
    Output_data =>Oc_data,
    Output_rv =>Oc_rv,
    Output_snd =>Oc_snd
  );
  shifterD : reg_lsbit
  port map(
    Load_data =>Ld_data,
    Load_rv =>Ld_rv,
    Load_snd =>Ld_snd,
    Shift_in_data =>Scd_data,
    Shift_in_rv =>Scd_rv,
    Shift_in_snd =>Scd_snd,
    Done_in_rv =>Dcd_rv,
    Done_in_snd =>Dcd_snd,
    Output_data =>Od_data,
    Output_rv =>Od_rv,
    Output_snd =>Od_snd
  );
process
  variable z:integer;
begin
  -- send(la,input(3),lb,input(2),lc,input(1),ld,input(0));
  --@synthesis_off
  la_data <= Yinput(3);
  lb_data <= Yinput(2);
  lc_data <= Yinput(1);
  ld_data <= Yinput(0);
  wait for 5 ns;
  --@synthesis_on
  --nolatch
  assign(la_snd,'1',1,3,lb_snd,'1',1,3,lc_snd,'1',1,3,ld_snd,'1',1,3);
  guard_and(la_rv,'1',lb_rv,'1',lc_rv,'1',ld_rv,'1');
  assign(la_snd,'0',1,3,lb_snd,'0',1,3,lc_snd,'0',1,3,ld_snd,'0',1,3);
  guard_and(la_rv,'0',lb_rv,'0',lc_rv,'0',ld_rv,'0');
```

```
loop
    --Decide if we should shift or complete.
    z:=selection(2);
    if (z=1) then
      --@synthesis_off
      nextbit <= selection(2, 1);
      wait for 5 ns;
      -- send(shift_in,nextbit(0));
      shift_in_data <= nextbit(0);
      wait for 5 ns;
      --@synthesis_on
      --nolatch
      assign(shift_in_sndX,'1',1,3);
      guard(shift_in_rvX,'1');
      assign(shift_in_sndX,'0',1,3);
      guard(shift_in_rvX,'0');
    else
      -- send(done_in);
      assign(done_in_sndX,'1',1,3);
      guard(done_in_rvX,'1');
      assign(done_in_sndX,'0',1,3);
      guard(done_in_rvX,'0');
      -- receive(oa,output(3),ob,output(2),oc,output(1),od,output(0));
      guard_and(oa_snd,'1',ob_snd,'1',oc_snd,'1',od_snd,'1');
      --@synthesis_off
      Youtput(3) <= oa_data;
      Youtput(2) <= ob_data;
      Youtput(1) <= oc_data;
      Youtput(0) <= od_data;
      wait for 5 ns;
      --@synthesis_on
      assign(oa_rv,'1',1,3,ob_rv,'1',1,3,oc_rv,'1',1,3,od_rv,'1',1,3);
      guard_and(oa_snd,'0',ob_snd,'0',oc_snd,'0',od_snd,'0');
      assign(oa_rv,'0',1,3,ob_rv,'0',1,3,oc_rv,'0',1,3,od_rv,'0',1,3);
    end if;
    wait for 5 ns;
  end loop;
end process;
end structure;
```

APPENDIX E

VHDL CODE FOR MPEG DITHERER

```
----------------------------------------------------------------------------
--
-- Title      :  dpCal
-- Design     :  datapath
-- Author     :  yy
-- date       :  6-8-2002
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity dpCal is
  generic(
    fwidth :  integer := 15
    );
  port(
    reset :  in std_logic;
    Calculate :  inout channel := init_channel(receiver => timing(1, 3));
    dp :  out std_logic_vector(3 downto 0)
    );
end dpCal;
architecture dpCal of dpCal is
  type NumType is array (0 to 15) of integer range 0 to 15;
  constant dpi :  NumType := (0,8,12,4,2,10,14,6,3,11,15,7,1,9,13,5);
begin
  dpCalculate :  process
    variable cnt :  integer := 0;
    variable index :  integer;
    variable w :  integer := fwidth*16;
  begin
    await(Calculate);
    --@synthesis_off
    index := ((cnt/w/2) mod 2)*8 +cnt mod 8;
    dp <= conv_std_logic_vector(dpi(index), 4);
    cnt := cnt+1;
    --@synthesis_on
    wait for delay (3, 5);
    receive(Calculate);
  end process;
```

```
end dpCal;

--------------------------------------------------------------------------------
--
-- Title      :  mtt
-- Design     :  datapath
-- Author     :  yy
-- date       :  6/8/2002
--------------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
USE work.nondeterminism.ALL;
USE work.channel.ALL;
USE std.textio.ALL;
entity mtt is
  GENERIC(
    fheight :  integer := 11;
    fwidth :  integer := 15;
    pagesize :  integer := 64
    );
  port(
    Splice :  inout channel := init_channel(receiver => timing(1,3));
    cr :  in std_logic_vector(7 downto 0);
    cb :  in std_logic_vector(7 downto 0);
    lum :  in std_logic_vector(7 downto 0);
    addr :  out std_logic_vector(15 downto 0);
    dp :  in std_logic_vector(3 downto 0)
    );
end MTT;
architecture mtt of mtt is
begin
  ttb_cal:  process
    VARIABLE pos :  integer := 0;
    VARIABLE w :  integer := fwidth*16;
    VARIABLE idp :  integer;
    --for MAKETTable
    TYPE  ntype IS ARRAY (0 TO 2) OF integer;
    CONSTANT ttb :  ntype := (33,97,161);
    VARIABLE ttx :  integer := 0;
    VARIABLE tty :  integer := 0;
    --VARIABLE ttz :  integer := 0;
    VARIABLE ix :  integer := 0;
    VARIABLE iy :  integer := 0;
  begin
    await(Splice);
    --@synthesis_off
    ix := conv_integer(cb);
    iy := conv_integer(cr);
    idp:=conv_integer(dp);
    ttx := 3;
    WHILE ix<ttb(ttx-1)+idp*4 LOOP
```

```
      ttx := ttx-1;
    END LOOP;
    tty := 3;
    WHILE iy<ttb(tty-1)+idp*4 LOOP
      tty := tty-1;
    END LOOP;
    addr <= dp & conv_std_logic_vector(ttx,2)&
            conv_std_logic_vector(tty,2) & lum;
    wait for delay(3,5);
    --@synthesis_on
    receive(Splice);
  end process;
end mtt;

-----------------------------------------------------------------------------
--
-- Title        :  mem
-- Design       :  datapath
-- Author       :  YY
-- date         :  6-8-2002
----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity mem is
  port(
    Mem_Access :  inout channel := init_channel(receiver => timing(1, 3));
    addr_r :  in std_logic_vector(15 downto 0);
    addr_w :  in std_logic_vector(15 downto 0);
    Din:  in std_logic_vector(7 downto 0);
    RW : in std_logic;
    Dout :  out std_logic_vector(7 downto 0)
    );
end mem;
architecture mem of mem is
  constant ttMemorySize :  integer := 16*4*4*256;
  type ttype is array (0 to ttMemorySize - 1)of std_logic_vector(7 downto 0);
  signal tt :  ttype;
begin
  memory :  process
  begin
    await(Mem_Access);
    --read delay is less that write
    --@synthesis_off
    if (RW = '0') then
      tt(conv_integer(addr_w)) <= Din;
      wait for delay(3, 5);
    else
      Dout <= tt(conv_integer(addr_r));
      wait for delay(1, 3);
```

```
    end if;
    --@synthesis_on
    receive(Mem_Access);
  end process;
end mem;

-----------------------------------------------------------------------------
--
-- Title      :  cnt
-- Design     :  datapath
-- Author     :  yy
-- date       :  6-8-2002,
-----------------------------------------------------------------------------
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.nondeterminism.all;
use work.channel.all;
entity cnt is
  port(
    Increment :  inout channel := init_channel(receiver => timing(1, 3));
    DataOut :  buffer std_logic_vector(15 downto 0) := (others => '0')
    );
end cnt;
architecture cnt of cnt is
begin
  cnt :  process
  begin
    await(Increment);
    --@synthesis_off
    DataOut <= DataOut + 1;
    --@synthesis_on
    wait for delay (3, 5);
    receive(Increment);
  end process;
end cnt;

------------------------------------------------------------------------------
--
-- Title      :  datapath
-- Design     :  datapath
-- Author     :  yy
-----------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.channel.all;
entity datapath is
  port(
    Reset :  in std_logic;
    Increment :  inout channel := init_channel;
    Calculate :  inout channel := init_channel;
    mem_RW : in std_logic;
```

```
    Mem_Access : inout channel := init_channel;
    Splice : inout channel := init_channel;
    cb : in std_logic_vector(7 downto 0);
    cr : in std_logic_vector(7 downto 0);
    lum : in std_logic_vector(7 downto 0);
    mem_Din : in std_logic_vector(7 downto 0);
    mem_Dout : out std_logic_vector(7 downto 0)
    );
end datapath;
architecture datapath of datapath is
  component cnt
    port(
      Increment : inout channel := init_channel(receiver => timing(1, 3));
      DataOut : buffer std_logic_vector(15 downto 0) := (others => '0')
      );
  end component;
  component dpcal
    port(
      reset : in std_logic;
      Calculate : inout channel := init_channel(receiver => timing(1, 3));
      dp : out std_logic_vector(3 downto 0)
      );
  end component;
  component mem
    port(
      Mem_Access : inout channel := init_channel(receiver => timing(1, 3));
      addr_r : in std_logic_vector(15 downto 0);
      addr_w : in std_logic_vector(15 downto 0);
      Din : in std_logic_vector(7 downto 0);
      RW : in std_logic;
      Dout : out std_logic_vector(7 downto 0)
      );
  end component;
  component mtt
    port(
      Splice : inout channel := init_channel(receiver => timing(1, 3));
      cr : in std_logic_vector(7 downto 0);
      cb : in std_logic_vector(7 downto 0);
      lum : in std_logic_vector(7 downto 0);
      addr : out std_logic_vector(15 downto 0);
      dp : in std_logic_vector(3 downto 0)
      );
  end component;
  signal cbCal : std_logic_vector (7 downto 0);
  signal cnt_addr : std_logic_vector (15 downto 0);
  signal crCal : std_logic_vector (7 downto 0);
  signal dpIn : std_logic_vector (3 downto 0);
  signal lumCal : std_logic_vector (7 downto 0);
  signal mtt_addr : std_logic_vector (15 downto 0);
begin
  addr : mtt
    port map(
      Splice => Splice,
```

```
      addr => mtt_addr,
      cb => cb,
      cr => cr,
      dp => dpIn,
      lum => lum
      );
  dp :  dpcal
    port map(
      Calculate => Calculate,
      dp => dpIn,
      reset => reset
      );
  memo :  mem
    port map(
      Din => mem_Din,
      Dout => mem_Dout,
      RW => mem_RW,
      Mem_Access => Mem_Access,
      addr_r => mtt_addr,
      addr_w => cnt_addr
      );
  pos :  cnt
    port map(
      Increment => Increment,
      DataOut => cnt_addr
      );
end datapath;

-----------------------------------------------------------------------------
--
-- Title       :  MpegDecoder
-- Design      :  datapath
-- Author      :  yy :p
-----------------------------------------------------------------------------
library IEEE;
use IEEE.std_logic_1164.all;
use work.channel.all;
entity MpegDecoder is
  port(
    sampling :  out std_logic;
    Display :  inout std_logic_vector(7 downto 0);
    DPdata :  inout channel := init_channel;
    cb :  inout channel := init_channel;
    cr :  inout channel := init_channel;
    lum :  inout channel := init_channel
    );
end MpegDecoder;
architecture MpegDecoder of MpegDecoder is
  component ctrl
    port(
      DPdata :  inout channel := init_channel(receiver => timing(1, 2));
      cr, cb, lum :  inout channel := init_channel(receiver => timing(1, 2));
      Increment :  inout channel := init_channel(sender => timing(1, 3));
```

```
      Mem_Access :  inout channel := init_channel(sender => timing(3, 5));
      Calculate :  inout channel := init_channel(sender => timing(3, 5));
      Splice :  inout channel := init_channel(sender => timing(3, 5));
      cr_d :  buffer std_logic_vector(7 downto 0);
      cb_d :  buffer std_logic_vector(7 downto 0);
      lum_d :  buffer std_logic_vector(7 downto 0);
      DPdata_d :  buffer std_logic_vector(7 downto 0);
      reset :  out std_logic := '0';
      mem_RW : buffer std_logic := '0'
      );
  end component;
  component datapath
    port(
      Reset :  in std_logic;
      Increment :  inout channel := init_channel;
      Calculate :  inout channel := init_channel;
      mem_RW : in std_logic;
      Mem_Access :  inout channel := init_channel;
      Splice :  inout channel := init_channel;
      cb :  in std_logic_vector(7 downto 0);
      cr :  in std_logic_vector(7 downto 0);
      lum :  in std_logic_vector(7 downto 0);
      mem_Din :  in std_logic_vector(7 downto 0);
      mem_Dout :  out std_logic_vector(7 downto 0)
      );
  end component;
  signal Increment :  channel := init_channel;  -- tells counter to count
  signal Calculate :  channel := init_channel;  -- tells dpCalc to do
its thing
  signal Mem_Access :  channel := init_channel;  -- tells memory to handle access
  signal mem_RW : std_logic;
  signal Splice :  channel := init_channel;  -- tells MTT to splice address
  signal Reset :  std_logic;
  signal n_cb :  std_logic_vector (7 downto 0);
  signal n_cr :  std_logic_vector (7 downto 0);
  signal n_DPdata :  std_logic_vector (7 downto 0);
  signal n_lum :  std_logic_vector(7 downto 0);
begin
  control :  ctrl
    port map(
      DPdata => DPdata,
      DPdata_d => n_DPdata,
      cb => cb,
      cb_d => n_cb,
      Increment => Increment,
      cr => cr,
      cr_d => n_cr,
      Calculate => Calculate,
      lum => lum,
      lum_d => n_lum,
      mem_RW => mem_RW,
      Mem_Access => Mem_Access,
      Splice => Splice,
```

```
      reset => Reset
      );
  datap : datapath
    port map(
      Reset => Reset,
      cb => n_cb,
      Increment => Increment,
      cr => n_cr,
      Calculate => Calculate,
      lum => n_lum,
      mem_Din => n_DPdata,
      mem_Dout => Display,
      mem_RW => mem_RW,
      Mem_Access => Mem_Access,
      Splice => Splice
      );
--@synthesis_off
  sampling <= mem_RW when probe(Mem_Access) else '0';
--@synthesis_on
end MpegDecoder;

------------------------------------------------------------------------------
--
-- Title      :  enviornment for mpegdecoder
-- Design     :  datapath
-- Author     :  yy
-- date       :  6-8-2002
------------------------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use std.textio.all;
use work.nondeterminism.all;
use work.channel.all;
use work.handshake.all;
entity mpegdecoder_tb is
  generic(
    fheight :  integer := 11;
    fwidth :  integer := 15
    );
end mpegdecoder_tb;
architecture TB_ARCHITECTURE of mpegdecoder_tb is
  file fh_cbx :  text;
  file fh_crx :  text;
  file fh_lum :  text;
  file fh_dp :  text;
  file fh_out :  text;
  signal s_cb :  std_logic_vector(7 downto 0);
  signal s_cr :  std_logic_vector(7 downto 0);
  signal s_lum :  std_logic_vector(7 downto 0);
  signal s_DPdata :  std_logic_vector(7 downto 0);
  signal s_output :  std_logic_vector(7 downto 0);
  signal s_display :  std_logic_vector(7 downto 0);
```

```
  component mpegdecoder
    port(
      sampling : out std_logic;
      Display : inout std_logic_vector(7 downto 0);
      DPdata : inout channel;
      cb : inout channel;
      cr : inout channel;
      lum : inout channel );
  end component;
  signal reset : std_logic := '1';
  signal failed : std_logic := '0';
  signal sampling : std_logic;
  signal DPdata : channel := init_channel(sender => timing(1, 2));
  signal cr, cb, lum : channel := init_channel(sender => timing(1, 2));
  signal Display : std_logic_vector(7 downto 0);
begin
  read_file : process
    variable v_line : line;
    variable good : boolean;
    variable v_int : integer;
  begin
    wait for delay (3, 5);
    --@synthesis_off
    reset <= '0';
    --@synthesis_on
    wait for delay (3, 5);
    -- open/read/send translation table to decoder
    --
    file_open(fh_dp, "dp.dat", read_mode);
    --@synthesis_off
    while not endfile(fh_dp) loop
      --@synthesis_on
      --read data from dp.dat
      readline(fh_dp, v_line);
      read(v_line, v_int, good);
      assert good
        report " dp.dat Text I/O read error!"
        severity error;
      s_DPdata <= conv_std_logic_vector(v_int, 8);
      wait for 1 ns;
      send(DPdata, s_DPdata);
      --@synthesis_off
    end loop;
    --@synthesis_on
    file_close(fh_dp);
    report "dp_file is closed successfully!";
    file_open(fh_out, "out.dat", read_mode);
    file_open(fh_cbx, "cbx.dat", read_mode);
    file_open(fh_crx, "crx.dat", read_mode);
    file_open(fh_lum, "lum.dat", read_mode);
    --@synthesis_off
    while not endfile (fh_cbx) loop
      --@synthesis_on
```

```
  --read data from cbx.dat
  readline(fh_cbx, v_line);
  read(v_line, v_int, good);
  assert good
    report " cbx.dat Text I/O read error!"
    severity error;
  s_cb <= conv_std_logic_vector(v_int, 8);
  wait for delay (3, 5);
  --read data from crx.dat
  readline(fh_crx, v_line);
  read(v_line, v_int, good);
  assert good
    report " crx.dat Text I/O read error!"
    severity error;
  s_cr <= conv_std_logic_vector(v_int, 8);
  wait for delay (3, 5);
  send(cb, s_cb);
  send(cr, s_cr);
  --read 4 data from lum.dat
  --@synthesis_off
  for dummy in 1 to 4 loop
    --@synthesis_on
    readline(fh_lum, v_line);
    read(v_line, v_int, good);
    assert good
      report " lum.dat Text I/O read error!"
      severity error;
    s_lum <= conv_std_logic_vector(v_int, 8);
    wait for delay (3, 5);
    send(lum, s_lum);
    --  Testing
    --read file from out.dat
    --@synthesis_off
    readline(fh_out, v_line);
    read(v_line, v_int, good);
    assert good
      report " out.dat Text I/O read error!"
      severity error;
    s_display <= conv_std_logic_vector(v_int, 8);
    wait for delay(1, 3);
    guard(sampling, '1');
    assert Display = s_display
      report "Miss-match!  "
      severity warning;
    if Display/=s_display then
      failed <= '1';
    end if;
    wait for delay (1, 1);
    guard(sampling, '0');
    --@synthesis_on
    --@synthesis_off
  end loop;
end loop;
```

```
    --@synthesis_on
    file_close(fh_out);
    report "out_file is closed successfully!";
    file_close(fh_cbx);
    report "cbx_file is closed successfully!";
    file_close(fh_crx);
    report "crx_file is closed successfully!";
    file_close(fh_lum);
    report "lum_file is closed successfully!";
    wait;
  end process;
  please :  mpegdecoder
    port map (
      sampling => sampling,
      Display => Display,
      DPdata => DPdata,
      cb => cb,
      cr => cr,
      lum => lum
      );
end TB_ARCHITECTURE;
```

REFERENCES

[1] http://www.cs.utah.edu/classes/cs5830/papers/FDL00.pdf.

[2] http://www.model.com/products/se.asp.

[3] http://www.xilinx.com/xlnx/xil_prodcat_landingpage.jsp?title=ISE+WebPack.

[4] http://www.async.ece.utah.edu/tools/atacsman.html#TECHNOLOGY.

[5] http://www.xilinx.com/publications/products/ cool/apps_pdf/xapp352.pdf.

[6] http://www.toobox.xilinx.com/docsan/xilinx4/data/docs/xst/constraints4.html.

[7] V. Akella. *An Integrated Framework for the Automatic Synthesis of Efficient Self-Timed Circuits from Behavioral Specifications*. PhD thesis, University of Utah, 1992.

[8] V. Akella and G. Gopalakrishnan. SHILPA: A high-level synthesis system for self-timed circuits. In *Proc. International Conference on Computer-Aided Design (ICCAD)*, pages 587–591. IEEE Computer Society Press, Los Alamitos, CA, Nov. 1992.

[9] V. Akella and G. Gopalakrishnan. Specification and validation of control-intensive IC's in hopCP. *IEEE Transactions on Software Engineering*, 20(6):405–423, 1994.

[10] K. v. Berkel. *Handshake Circuits: An Intermediary Between Communicating Processes and VLSI*. PhD thesis, Eindhoven University of Technology, 1992.

[11] K. v. Berkel and M. Rem. VLSI programming of asynchronous circuits for low power. In G. Birtwistle and A. Davis, editors, *Asynchronous Digital Circuit Design*, Workshops in Computing, pages 152–210. Springer-Verlag, New York, 1995.

[12] E. Brunvand. *Translating Concurrent Communicating Programs into Asynchronous Circuits*. PhD thesis, Carnegie Mellon University, 1991.

[13] E. Brunvand. Designing self-timed systems using concurrent programs. *Journal of VLSI Signal Processing*, 7(1/2):47–59, Feb. 1994.

[14] S. M. Burns. Automated compilation of concurrent programs into self-timed circuits. Master's thesis, California Institute of Technology, 1988.

[15] S. M. Burns and A. J. Martin. Syntax-directed translation of concurrent programs into self-timed circuits. In J. Allen and F. T. Leighton, editors, *Advanced Research in VLSI*, pages 35–50. MIT Press, Cambridge, MA, 1988.

[16] S. M. Burns and A. J. Martin. Synthesis of self-timed circuits by program transformation. In G. J. Milne, editor, *The Fusion of Hardware Design and Verification*, pages 99–116. Elsevier Science Publishers, New York, 1988.

[17] M. D. Ciletti. *Advanced Digital Design with the Verilog HDL*. Pearson Education, Inc., 2003.

[18] C. J.Myers, W. Belluomini, K. Killpack, E. Mercer, E. Peskin, and H. Zheng. Timed circuits: A new paradigm for high-speed design. In *Proc. Asia and South Pacific Design Automation Conference*, pages 335–340, Feb. 2001.

[19] T. C. Larry Wall and J. Orwant. *Programming Perl*. OŔeilly Media, Incorporated, Sebastopol, CA, third edition, 2000.

[20] M. Ligthart, K. Fant, R. Smith, A. Taubin, and A. Kondratyev. Asynchronous design using commercial HDL synthesis tools. In *Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems*, pages 114–125. IEEE Computer Society Press, Los Alamitos, CA, Apr. 2000.

[21] E. J. McCluskey. *Logic Design Principles*. Prentice-Hall, Upper Saddle River, NJ, 1986.

[22] A. D. F. Miron Abranmovici, Melvin A. Breuer. *Digital Systems Testing and Testable Design*. AT&T Bell Laboratories and W.H.Freeman and Company, Gordonsville, VA, 1990.

[23] E. R. Peskin. *Protocol selection, implementation, and analysis for asynchronous circuits*. PhD thesis, University of Utah, 2002.

[24] R. Smith and M. Ligthart. High-level design for asynchronous logic. In *Proc. Asia and South Pacific Design Automation Conference*, pages 431–436, Feb. 2001.