SPECIFICATION AND COMPILATION OF TIMED SYSTEMS

by

Hao Zheng

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

Master of Science

Department of Electrical Engineering

The University of Utah

June 1998

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

SUPERVISORY COMMITTEE APPROVAL

of a thesis submitted by

Hao Zheng

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
	Erik Brunvand
	Christian Schlegel

THE UNIVERSITY OF UTAH GRADUATE SCHOOL

FINAL READING APPROVAL

To the Graduate Council of the University of Utah:

I have read the thesis of Hao Zheng 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

Om P. Gandhi
Chair/Dean

Approved for the Graduate Council

Ann W. Hart
Dean of The Graduate School

ABSTRACT

This thesis presents a framework for the specification and compilation of modules in a system that uses different synchronization paradigms. These *timed* systems are described by using *timed handshaking expansions* (HSE) and a standard hardware description language, namely *VHDL*. Synthesizable subsets of these languages are defined to include constructs for describing timing behaviors, as well as, sequencing, concurrency, choice and looping. A new formal semantic model, *timed event/level structures*, is used to define the behaviors specified by the synthesizable subsets. A compiler is developed to translate the HSE and VHDL specifications to timed event/level structures. This compiler is integrated into `ATACS`, a synthesis tool for timed circuits. Finally we demonstrate our methodology on a practical example, an asynchronous implementation of the Maxlist algorithm.

For my family.

CONTENTS

LIST OF FIGURES

ACKNOWLEDGEMENTS

I would like to express gratitude to my supervisor, Professor Chris Myers, for introducing me into the exciting and challenging asynchronous design field, and for his constant guidance, support, and encouragement throughout my entire research presented in this dissertation. I would like to thank Professor Erik Brunvand and Professor Christian Schlegel for their helpful comments on this dissertation, and for serving on my supervision committee.

During my years at the University of Utah, I received a lot of help from people. I would like to thank Wendy Belluomini, Robert Thacker, Eric Mercer, Brandon Bachman, and Chris Krieger for their technical support. I would also like to thank Doris Marks for her administrative help throughout the years.

Especially, I would like to express my gratitude to my family for their great help and lots of invaluable advice.

This research was supported by a grant from Intel Corporation and NSF CAREER award MIP-9625014.

CHAPTER 1

INTRODUCTION

With clock speeds approaching 1 GHz, designers are beginning to abandon purely synchronous design. Techniques such as "Self-Reseting Logic" and "Opportunistic Time Borrowing," where signals are not locally synchronized with the clock, are used today by designers to achieve performance not feasible with purely synchronous technology. Designers at HAL/Fujitsu use self-timed design methods to speedup their on-chip divider, and Intel has investigated using asynchronous circuits to speedup the instruction length decoder for the x86 architecture. This is because asynchronous circuits have several potential advantages. An asynchronous circuit is one which performs synchronization without a global clock. Therefore, skew in synchronization signals can be ignored, and the extra circuitry for clock drivers and buffers is not necessary. In asynchronous circuits, the speed of the circuit is allowed to change dynamically, so the performance is governed by the average-case delay instead of worst-case delay. The delay of asynchronous circuits can vary significantly over different processing runs, supply voltages, and operating temperatures, but asynchronous circuits can adapt to those variations and can operate correctly under all variations and simply speed up or slow down, as necessary. In an asynchronous system, components can be interfaced without the difficulties associated with synchronizing clocks in a synchronous system. Also, when a faster component becomes available, it can be easily inserted into the system without requiring any other changes to the system resulting in an overall system performance improvement. Asynchronous circuits can also lower system power requirements because asynchronous circuits reduce synchronization power by not requiring additional clock drivers and buffers to limit clock skew. They can also automatically power down unused components. Finally, asynchronous circuits can make efficient use of a dynamic power supply.

Unfortunately, wide application of asynchronous circuits is limited by several problems. Due to the lack of CAD tools that address the complex timing issues involved, the bulk of the design using such techniques is being done manually, which limits their application to

a very small part of the design. Asynchronous circuits must also avoid hazards. A hazard is a spurious signal transition, or glitch. While hazards can be ignored in a synchronous design as they are filtered out by the clock signal, any hazards in an asynchronous design can potentially lead to a malfunction. Therefore, careful design is necessary to avoid hazards in an asynchronous design which often leads to a significant increase in circuit area. Asynchronous circuits have difficulties in interfacing with existing synchronous designs, and may not be suitable for semicustom design because many asynchronous designs require the use of special complex-gates. Asynchronous design may also be unreliable because many asynchronous circuit designers play tricks and make assumptions which must be checked with simulation. However, simulation is not perfect so unreliable designs can be produced. Therefore, formalization and automation of these techniques will allow larger parts of future designs to take advantage of their benefits.

1.1 Specification and Synthesis Methodologies

The first step in any synthesis method is to specify what is to be designed. Many methods have been proposed for the specification of asynchronous designs. Some, however, are restricted to the signal transition level, such as *I-nets* [17], *signal transition graphs* [7] [16], *change diagrams* [26], *asynchronous finite-state machines* [9] [22] [28], and *state graphs* [1]. Some languages do exist which abstract the behavior of the design, but they use non-standard languages such as *communicating sequential processes* (CSP) [14], *Occam* [5], and *Tangram* [4]. Each of these specification methods is also designed for a particular design style and synthesis methodology. Furthermore, none of these methods allows timed systems to be easily specified.

Almost all commercial design tools for the simulation and synthesis of synchronous digital systems employ standard hardware description languages (such as Verilog and VHDL) and abide by common specification practices. This makes designs (and designers) easily portable among tools. We wish to take advantage of this excellent repertoire of already existing tools and knowledge. This approach promises to provide a dual advantage. First, it saves tool development time and effort. Second, using existing HDL and tools that are already familiar to designers enables easier migration into, and assimilation of, the new design technology. It allows designers to think in familiar terms, rather than having to go through difficult training and even a complete brainwash, when new specification methods and tools are radically different.

Therefore, the purpose of this thesis is to develop a methodology that guides the designer in the specification of timed systems using a standard HDL, namely VHDL, in a manner that is independent of design style and synchronization method. The synthesizable subset of the language has been refined and a methodology for the specification of timed systems has been developed. A new semantic model, timed event/level structures (TEL) [2] is used to define the behavior specified in VHDL. A new compiler has been developed to translate the VHDL specifications into the TEL structures. This compiler has been integrated into the CAD tool `ATACS`, which accepts the output of the compiler and synthesizes a timed circuit. Figure 1.1 shows the relationship between the compiler and `ATACS`.

1.2 Outline of Thesis

This thesis is organized as follows. Chapter 1 serves as a general introduction. Chapter 2 describes timed handshaking expansions (HSE), which are derived from Martin's CSP language [14], and how a synthesizable subset of VHDL can be used to model those timed HSE constructs and behaviors. Chapter 3 describes timed event/level structures, the semantic model used to describe behavior in timed HSE and VHDL, and introduces the automatic procedures to interpret our timed HSE and VHDL specifications with this new semantic model. Chapter 4 gives examples to show the promise of this specification method by demonstrating how complex designs can be modeled and synthesized. Chapter 5 gives our conclusions and ideas for future research.

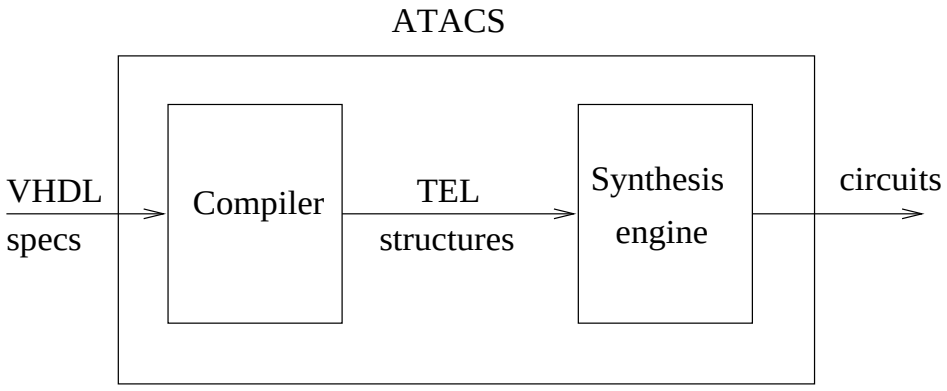

Figure 1.1. The overview of `ATACS`.

CHAPTER 2

SPECIFICATION OF TIMED SYSTEMS

This chapter describes the syntax rules of timed HSE, which is a subset of Martin's CSP language [14], and the synthesizable subset of VHDL. Both languages include constructs for describing timing behaviors, sequencing, concurrency, choice and looping.

2.1 Timed Handshaking Expansions

The behavior of a model is defined by a module in timed handshaking expansions (HSE). Each module consists of one or several processes describing the operations of the model. The processes execute in parallel. Besides the processes, there are also declarations that define variables used by the processes. The syntax of a module is shown as follows:

module_declaration ⇒
    module name;
        declarations
        processes
    endmodule

declarations ⇒
    delay name = delays;
    | mode name = {[initial_value], [delays]};
    {, ...}

mode ⇒ **input** | **output**;

The declaration part includes delay declarations and interface declarations. The delay declarations define delay variables. These delay variables are then used to specify the rising and falling delays of signals. Examples of such delay variables are shown below:

delay envdelay = <10, 40; 15, 50>;

delay gatedelay = <0, 20>;

The first declaration creates a delay variable with two pairs of delays. The first pair specifies the range of time in which a signal is allowed to change from '0' to '1'. The second pair specifies the range of time in which a signal is allowed to change from '1' to '0'. For example, a signal s with a delay specified by delay variable *envdelay* is initially '0', if at time 0 there is a request to change s to '1', this transition occurs at any time x from 10 to 40 time units. After s stays '1' for n time units, there is a request to change it to '0', this transition occurs at any time from x+n+15 to x+n+50. This example is illustrated in Figure 2.1. Note that the second delay variable has only one pair of delays. In this case, both rising and falling delays have the same values.

The interface declarations define the inputs and outputs of a module. When declaring the input and output signals, we can also assign the signals with initial values and delays. The following shows an example of an input signal declaration:

input result = {**true**, envdelay}

Both the initial value and delay are optional. The default value of the initial value is **false**. The default value of the delay is zero to infinity.

The behavior of a module is described by processes which execute in parallel. A process has the following form:

process label; process_body **endprocess**

The process_body may contain actions, selection commands, and repetition commands. The actions are used to assign values to the output signals. In our subset, signals can only take two values: **true** and **false**. When an output signal x is assigned with **true** or **false**, it is denoted by $x+$ or $x-$, respectively.

There are two structures to control the flow of a process. They are *selection*s and *repetition*s. A selection command has the following form:

$$[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]$$

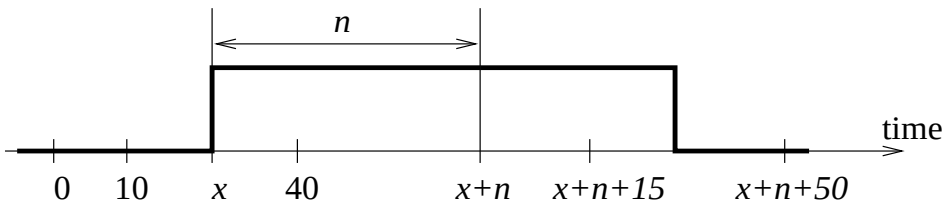

Figure 2.1. A delay definition example.

where G_1 through G_n are boolean expressions, S_1 through S_n are arbitrary program parts (G_i is called a "guard", and $G_i \rightarrow S_i$ is called a "guarded command"). When a process executes a selection command, all guards in that selection command are evaluated first. If one of the guards G_i is **true**, then S_i following that guard is executed. There is a special form of selection commands, $[G]$, which stands for $[G \rightarrow \textbf{skip}]$, and is used to suspend the execution of a process until G evaluates to **true**.

A repetition command has the following form:

$$*[S]$$

where S is an arbitrary program part. $*[S]$ stands for $*[\textbf{true} \rightarrow S]$, and causes S to be executed forever. This is usually used to define a reactive process:

$$*[[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]]$$

When executing this command, the process waits until one of the guards is **true**, then executes the program part following that guard, and repeats. Another type of repetition construct is shown below:

$$[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n; *]$$

The operation of this construct is similar to that of the *selection* construct defined above except that after a guarded command followed by a '$*$' is executed, the control loops back to the beginning of the construct and the construct executes again.

To make a process run as expected, two arbitrary program parts may be composed with three operators: the sequential operator (;); the concurrent operator ($\|$); and the choice operator ($|$). $S_1;S_2$ says that S_2 can start executing only after the execution of S_1 completes. $S_1 \parallel S_2$ says that S_1 and S_2 execute in parallel. $S_1 \mid S_2$ says that either S_1 or S_2 can execute, but not both. Figure 2.2 shows a complete example for a *scsi* controller written in timed HSE.

2.2 Specification of Timed Systems in VHDL

This section introduces a subset of VHDL to specify timed systems. The reason we define a subset of VHDL is that VHDL is a huge and complex language containing many features that cannot be synthesized and all VHDL design tools can only synthesize a subset of it. For example, `SYNOPSYS` does not allow *wait* statements within a process on

```
module scsi;

    delay gatedelay = ⟨0,5⟩;
    delay envdelay = ⟨20,50⟩;

    input ack = {true, envdelay};
    input go = {false, envdelay};
    output req = {true, gatedelay};
    output rdy = {false, gatedelay};

    process scsictrl;
    *[ req-; rdy+; [go]; rdy-; [ack]; req+; [go & ack] ]
    endprocess

    process ackenv;
    *[ [ req]; ack-; [req]; ack+ ]
    endprocess

    process goenv;
    *[ [rdy]; go+; [ rdy]; go- ]
    endprocess

endmodule
```

Figure 2.2. A complete timed HSE example for a *scsi* controller.

any signals other than the clock. Compared with timed HSE, VHDL is more expressive because it allows the specification of circuits hierarchically. In the following several subsections, all features in the synthesizable subset are introduced.

2.2.1 Entities and Architecture Bodies

The description of a VLSI circuit can be divided into two parts: the external view and the internal view. The external view describes the interface between the internal structure and the outside world. It specifies the number and types of the input and output signals. The internal view describes how the circuit implements its function. In VHDL, the *entity declaration* describes the external interface, and one or more *architecture bodies* describe alternative internal implementations.

The syntax rules for entities and architecture bodies are shown in Figure 2.3. The identifier in an entity declaration names the module so that it can be referred to later. The port clause, which is optional, names each of the ports, which together form the interface to the entity. The ports can be thought of as being analogous to the pins of a

entity_declaration ⇒
    entity identifier **is**
        [**port** (interface_list);]
    end [**entity**] [identifier];

interface_list ⇒
    (identifier {, ...} : [mode] type [:= expression]) {; ...}

mode ⇒ **in** | **out** | **inout**

architecture_body ⇒
    architecture identifier **of** *entity*_name **is**
        declarations
    begin
        concurrent_statements
    end [architecture] [identifier];

Figure 2.3. The syntax rules for entities and architecture bodies.

circuit. Each port of an entity has a type, which specifies the kind of information that can be communicated. In this subset, the allowed data types are **bit** and **std_logic**. Each port also has a mode which specifies whether information flows into or out from the entity through the port.

The example shown in Figure 2.4 describes an entity named SPDOR, with two input ports and one output port, and all of them are a single bit. The output port *out1* is declared with a mode of **inout** so that the information on this port can be both sensed or driven by the module SPDOR.

The internal operation of a module is described by an architecture body. In general, an architecture body applies some operations to the values on input ports, generating values to be assigned to output ports. The operations can be described either by processes, which contain sequential statements operating on values, or by a collection of components representing subcircuits, or by both. The identifier in an architecture body names a particular architecture body, and the entity name specifies which module is described by this architecture body. A single entity may have one or several different architecture bodies. The declarations in an architecture body are declarations needed to implement

entity SPDOR **is**
    port (*in1, in2* : **in** std_logic; *out1* : **inout** std_logic);
end SPDOR;

Figure 2.4. The entity and port declarations for an OR gate.

the operations. The items may include many kinds of declarations, but only signal declarations and component declarations are allowed for synthesis. The statements in the architecture body execute concurrently. In our synthesizable subset of VHDL, only process statements and component instantiation statements are allowed for synthesis.

2.2.2 Signal and Component Declarations

When the operation of an architecture body requires generation of intermediate values, internal signals are needed. Before these signals are used, they must be declared through *signal declarations*. The syntax for a signal declaration is shown in Figure 2.5. The declaration simply names each signal, specifies its type and optionally includes an initial value for all signals declared by the declaration. The following shows an example of how a signal is declared:

signal x : std_logic := '0';

After the declaration, those signals can be used by the following concurrent statements in the architecture body. An important point that should be pointed out is that the ports of the entity are also visible inside the architecture body and are used in the same way as signals.

For synthesis of timed systems, it is necessary to know how an environment can behave. The signals are used to connect the environment to the circuit being designed. Since only the signals connected to the outputs of the circuit can be synthesized, the command $--@$ is used to indicate which signals are connected to the outputs of the circuit being designed, and which are not. This command is used only at the highest level which contains concurrent statements for the circuit and its environment, and is ignored by the simulator.

signal_declaration ⇒
    signal identifier {, ...} : type [:= expression];

component_declaration ⇒
    component identifier [**is**]
        [**port**(interface_list);]
    end [**component**] [identifier];

Figure 2.5. The syntax rules for signal and component declarations.

When designing a large and complicated system, a hierarchical approach is a good way to attack the difficulty and complexity. In this subset, component declarations and component instantiation statements are used for hierarchical design. Component instantiation statements are introduced in a later section. The syntax of component declarations is shown in Figure 2.5. Similar to entity declarations, a component declaration simply specifies the external interface to the component. Figure 2.6 shows a simple example of how a component is declared. The declaration defines a component type that represents a flipflop with clock *clk*, clear *clr* and data inputs *d*, and a data output *q*.

component flipflop **is**
    port(*clk, clr, b* : **in** bit, *q* : **out** bit);
end flipflop;

Figure 2.6. An example component declaration.

2.2.3 Concurrent Statements

*Concurrent statement*s in an architecture body describe a module's operations, and are executed in parallel. In our synthesizable subset, the allowed concurrent statements are process statements, which contain sequential statements operating on signal values, and component instantiation statements representing subcircuits. The syntax rules for process and component instantiation statements are shown in Figure 2.7.

The process label identifies the process. An optional sensitivity list may be included in a process after the keyword **process**, but only processes without sensitivity lists are allowed in our synthesizable subset. The declarations in a process statement may contain

process_statement ⇒
    [*process*_label:]
    process [**is**]
        declarations
    begin
        sequential statements
    end process [*process*_label];

component_instantiation_statement ⇒
    [*instantiation*_label:]
    [**component**] *component*_name
        [**port map** (association_list)];

Figure 2.7. The syntax rules for concurrent statements.

many declarative items, but only signal declarations are allowed for synthesis. The sequential statements form the process body and can include *wait, signal assignment, if*, and *loop* statements. These statements are introduced in the next section.

If a component declaration defines a kind of module, then a component instantiation statement specifies a usage of such a module in a design. The syntax rules show that we may simply name a component declared in the architecture body and provide actual signals to connect it to the ports. The label is necessary to identify the component instance. Figure 2.8 shows the block diagram of the *scsi* controller. Its corresponding structural VHDL code is shown in Figure 2.9. It consists of three components: *scsictrl*, *goenv*, and *ackenv*. Four internal signals are declared to connect these three components together. The entity declaration of the *scsi* controller is shown on the top of Figure 2.9. Since the circuit has no input and output signals, there is no port declaration in the entity. Since the internal signals *req, rdy* connect to the outputs of the circuit *scsictrl*, they are specified by the command '– – @ out', and the other two are specified by the command '– – @ in'. Note that all we have done here is to specify the structure of this level of the design hierarchy, without having indicated how these components are implemented.

2.2.4 Sequential Statements

As mentioned in the previous section, a process body may contain sequential statements. It is so called sequential because when the process is activated, it starts executing from the first sequential statement and continues until it reaches the last one. It then starts again from the first one. This would be an infinite loop, and is desirable in electronic circuits because circuits typically operate continuously until the power is shut down.

Many kinds of sequential statements can be contained in a process. In our synthesizable subset, only *wait* statements, *signal assignment* statements, *if* statements, and

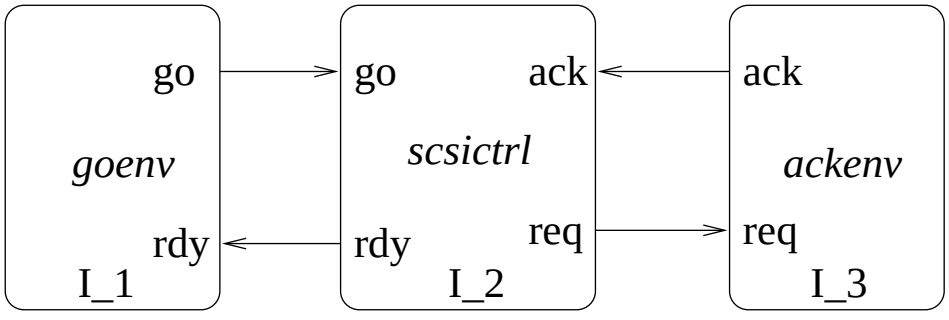

Figure 2.8. The block diagram of the *scsi* controller.

```
entity scsi is
end scsi;

architecture structural of scsi is

    component scsictrl is
        port (go, ack : in std_logic; req, rdy : out std_logic);
    end component scsictrl;

    component goenv is
        port (rdy : in std_logic; go : out std_logic);
    end component goenv;

    component ackenv is
        port (req : in std_logic; ack : out std_logic);
    end component ackenv;

    signal go, ack : std_logic; -- @ in
    signal req, rdy : std_logic; -- @ out

begin

    I_1 : component goenv
        port map (rdy, go);

    I_2 : component scsictrl
        port map (go, ack, req, rdy);

    I_2 : component ackenv
        port map (req, ack);
end architecture structural;
```

Figure 2.9. Structural VHDL code of the *scsi* controller.

loop statements are allowed for synthesis. The syntax rules for sequential statements are shown in Figure 2.10.

The *wait* statements are used to specify when processes respond to changes of signal values. In VHDL semantics, processes would be infinite loops. If *wait* statements are included in a process, the process execution will be suspended until the relevant boolean conditions in the *wait* statements are satisfied. For example, the *wait* statement

wait until req = '1';

causes the executing process to suspend until the value of the signal req changes to '1'. The condition expression is tested while the process is suspended to determine whether

wait_statement ⇒
    [label:] **wait** [**until** *boolean*_expression];

signal_assignment_statement ⇒
    [label:] *signal*_name ⇐ (*value*_expression [**after** *time*_expression]){, ...}

if_statement ⇒
    if *boolean*_expression **then**
        sequential statements
    elsif *boolean*_expression **then**
        sequential statements
    else
        sequential statements
    end if;

loop_statement ⇒
    [**while** *boolean*_expression] **loop**
        sequential statements
    end loop;

Figure 2.10. The syntax rules for sequential statements.

to resume the process. However, even if the condition is true when the *wait* statement is executed, the process still suspends until the appropriate signals change and the condition becomes true again. Thus, the *wait* statement is called "event-sensitive." To just test if a condition is true, ignoring whether there is an event, an *if* statement is put before the *wait* statement. If the condition is true, the *wait* statement is skipped. Otherwise, the process suspends until the condition is true. An example is shown as follows:

```
if req = '0' then
    wait until req = '1';
end if;
```

Signal assignment statements are used to schedule events on signals and change the values of the signals after some delay. There are two events associated with each signal s in a specification. The event $s \Leftarrow$ '0' denotes that signal s is changed from a high to low value, and the event $s \Leftarrow$ '1' denotes that signal s is changed from a low to high value. The delay may be supplied by a function which takes two parameters. These two parameters specify the lower and upper bounds associated with this signal transition. The lower bounds are nonnegative integers, and the upper bounds are an integer greater than or equal to the lower bounds. An example is shown as follows:

$x <= \text{'0'}$ **after** delay(99,109);

The delay may also be specified by a single time value, which defines a bounded delay from the value to ∞. Finally, the delay may be unspecified. In this case, the lower and upper bounds for the signal transition are 0 and ∞, respectively. Since real values can be expressed as rationals within any required accuracy, restricting the bounds to be integers does not limit the expressiveness in practice. Since there are only a finite number of timing parameters, if any are rational, we can multiply all of them by the least common denominator. For simulation, this delay function returns a random delay value between these bounds. For synthesis, the timed circuit that we generate must be synthesized in such a way to guarantee that it operates correctly given that the delay for this event always falls in these bounds. The timing analysis algorithms and synthesis algorithms necessary to generate such timed circuits have been the subject of numerous papers [19, 20, 18, 3].

Finding these *timing constraints*, or the delay bounds to associate with the transitions on these signals is not a trivial task. The timing constraints for input signal transitions can usually be determined from interface specifications or datapath delay estimates. The timing constraints for output signal transitions, however, present a "chicken and egg problem," since the timing constraints cannot be known until the circuit is synthesized, but the circuit cannot be synthesized without giving the timing constraints. The traditional delay-insensitive or speed-independent approaches to asynchronous design assume no timing information. In other words, they assume that delays can be anywhere from 0 to ∞. This conservative assumption can often lead to unnecessarily complex circuit implementations, and limits the designs that can be produced. It is quite reasonable, however, to expect an automatic analysis of the given gate library to produce a safe estimate of the maximum delay for the gates in the library to be used, and by making some assumptions about the complexity of the synthesized logic, this can be used to set the upper bound of the timing constraint for each output signal transition. The lower bound of the timing constraint should usually be set to a very low value since optimizations could potentially reduce the gate to nothing more than a wire. After the circuit is generated, it must be back-annotated with timing information from the gate library and verified to be correct which is the subject of a previous paper [23] and outside the scope of this thesis.

The *if* statements are used to select certain sequential statements to execute depending

on a set of input conditions. The boolean expressions after the keyword **if** are the conditions that are used to control whether or not the statements after the keyword **then** are executed. If the condition evaluates to true, the statements are executed. If none of these conditions evaluates to true, the statements following the **else** are executed. For example:

```
if sel = '1' then
    result ⇐ '1';
else
    result ⇐ '0';
end if;
```

Loop statements are used to describe repetitively executed structures. There are several different forms of *loop* statements in VHDL. Our synthesizable subset supports *infinite* loops and *while* loops. *While* loop statements test conditions before each iteration. If the condition is true, iteration proceeds. If it is false, the loop is terminated. The syntax of *loop* statements is shown in Figure 2.10.

Figure 2.11 shows a complete example including several *wait* and *signal assignment* statements, and an *if-then-elsif* clause. The basic operation of a SPDOR gate is as follows: when either $in1$ or $in2$ goes high, $out1$ goes high which causes x to go low after some delay which will reset $out1$ to low. Therefore, $out1$ is a pulse, and correct operation of this circuit is very dependent on timing. This circuit is not directly synchronized by a clock, though the data it receives and generates may be latched by a clock elsewhere in the design. Such self-synchronizing circuits are typical of the types of timed circuits currently in use today.

An *if-then-elsif* clause may also be used to specify nondeterministic choice made by the environment. In this case, the *if-then-elsif* clause must be proceeded by an assignment to a random variable which is tested in the *if-then-elsif* clause to simulate random behavior by the SPDOR's environment. As an example, consider the environment architecture shown in Figure 2.12. In this example, the variable z is used to get a random value of 1 or 2, and based on this value, it causes the environment to select to send *in1* or *in2*, but not both. For synthesis of timed systems, it is necessary to know how an environment can behave. While this requires more information than is typically needed for traditional cloud of logic synchronous synthesis, this additional information can lead to additional reductions in circuit complexity. For simulation, this environment architecture serves as a

```
entity SPDOR is
    port (in1, in2 : in std_logic, out1 : inout std_logic);
end SPDOR;
architecture SPDOR of OR is
    signal x : std_logic := '1';
begin
    circuit : process
    begin
        wait until in1 = '1' or in2 = '1';
        if (in1 = '1') then
            out1 <= '1' after delay(201,221);
            wait until out1 = '1';
            x <= '0' after delay(99,109);
            wait until x = '0';
            out1 <= '0' after delay(199,219);
            wait until out1 = '0';
            x <= '1' after delay(101,111);
            wait until x = '1';
        elsif (in2 = '1') then
            out1 <= '1' after delay(101,111);
            wait until out1 = '1';
            x <= '0' after delay(49,59);
            wait until x = '0';
            out1 <= '0' after delay(99,119);
            wait until out1 = '0';
            x <= '1' after delay(51,56);
            wait until x = '1';
        end if;
    end process;
end SPDOR;
```

Figure 2.11. The complete example for SPDOR.

testbench which provides nondeterministic input behavior for the circuit being designed. We believe that this coupled with the random delay function will lead to finding more bugs during simulation. Due to their highly-sequential nature, bugs in timed systems may only exhibit themselves for unusual delay and environment behavior which is often missed in deterministic simulations. For example, a bug may only be present when one signal is asserted after its maximum delay and another is asserted after its minimum delay. Another example is that a bug may only show up if the environment does two *in1* events in a row, but you may have only been simulating alternating *in1* and *in2* events.

```
entity SPDOR_ENV is
    port (out1 : in std_logic, in1, in2 : inout std_logic);
end SPDOR_ENV;
architecture BEHAVIOR of SPDOR_ENV is
    variable z : integer;
begin
    environment : process
    begin
        z := random(2);
        if (z = 1) then
            in1 <= '1' after delay(500,550);
            wait until in1 = '1';
            in1 <= '0' after delay(269,299);
            wait until in1 = '0';
        elsif (z = 2) then
            in2 <= '1' after delay(500,550);
            wait until in2 = '1';
            in2 <= '0' after delay(269,299);
            wait until in2 = '0';
        end if;
    end process;
end BEHAVIOR;
```

Figure 2.12. The complete example for the SPDOR's environment.

2.2.5 A Design Package to Simulate Nondeterministic Behavior

To simulate nondeterministic environment and delay behavior, a VHDL design package is developed. The package, shown in Figure 2.13, is used for simulation and ignored by synthesis. It includes two functions. The first function, **random** (number), is a random number generator. It takes an integer *number* as a parameter, and returns a number between 1 and *number*. The second function, **delay** (l,u), takes two integer numbers l and u, and returns a number with type of *time* between l and u. A simulation of SPDOR using this package and `SYNOPSYS` is shown in Figure 2.14.

```
package nondeterminism is
    function random(number:integer) return integer;
    function delay(l,u: integer) return time;
end nondeterminism;

package body nondeterminism is
    function random(number:integer) return integer is
    begin
        return((RAND mod number)+1);
    end random;

    function delay(l,u: integer) return time is
        variable randel : time;
        variable num,iter : integer;
    begin
        randel := 0 ns;
        iter := random(u – l + 1) + l - 1;
        for i in 1 to iter loop
            randel := randel + 1 ns;
        end loop;
        return randel;
    end delay;
end nondeterminism;
```

Figure 2.13. A design package for nondeterministic behavior.

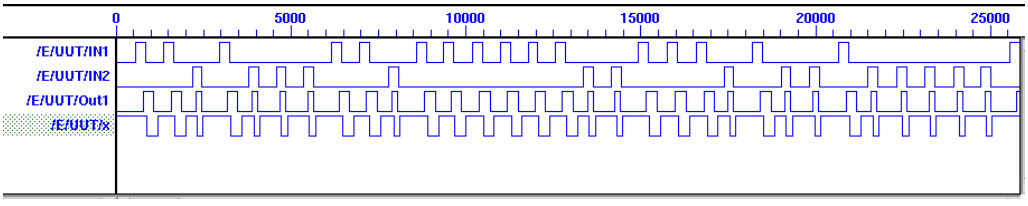

Figure 2.14. A nondeterministic SYNOPSYS simulation of the SPDOR.

CHAPTER 3

COMPILATION

This chapter describes the compilation procedures that are used to translate timed HSE and VHDL specifications into TEL structures. First, we describe TEL structures. Then, we describe composition and renaming rules of TEL structures. Finally, we describe the compilation procedures to interpret timed HSE and VHDL specifications.

3.1 Timed Event/Level Structures

In order to define the behaviors specified by a model in our synthesizable subset of VHDL, we use *timed event/level (TEL) structures*, a variant of Myers' timed event-rule structures [18] with a boolean condition added to each rule in the rule set. Event structures were introduced by Winskel [27], and timing has been added to them in several ways. Subrahmanyam added timing to event structures using temporal assertions [24]. Burns introduced timing in a deterministic version, the event-rule (ER) system, in which causality is represented using a set of rules, and a single delay value is associated with each rule [6]. Myers introduced timed ER structures that extend ER systems with bounded timing constraints and add conflicts from event structures to model nondeterministic behavior (namely, environmental choice). Timed event/level structures, which are first introduced in [2] by Belluomini, extend timed ER structures by associating a boolean expression with each rule.

TEL structures are composed of a set of signals (N), a set of *atomic actions* (A), a set of *events* (E), a set of *rules* (R), and a *symmetric conflict relation* ($\#$). In timed systems, the signal set N contains all input, output, and internal signals. The atomic action set A contains a rising transition and a falling transition, denoted by $x+$ and $x-$ respectively, for each signal x in the signal set N. There is a special kind of action: dummy action, denoted by '$'. The dummy actions do not cause any signal transitions in a system. They are treated as program pointers to indicate how far a process has reached. The occurrence of an action is an event, and it is denoted (a, i) where a is the action and i

is an *occurrence index* for the action. The first instance of this action has $i = 0$, and i increments with each subsequent instance. The event is a dummy event if it is an instance of a dummy action.

The rule set R is used to represent a causal dependence between two events. Each rule of the form $\langle e, f, l, u, b \rangle$ is composed of an *enabling event* e, an *enabled event* f, a *bounded timing constraint* $\langle l, u \rangle$, and a sum-of-product boolean expression over the signals in the signal set N. Informally, a rule states that the enabled event cannot occur until the rule is satisfied and the boolean expression b evaluates to true. The bounded timing constraint places a lower and upper bound on the timing of a rule. A rule is said to be *satisfied* if at least l time units has passed since the enabling event e fired. A rule is said to be *expired* if at least u time units has passed since the enabling event e fired. Again ignoring conflict, an event cannot occur until *all* rules enabling it are satisfied and the boolean expressions associated with those rules are true. This causality requirement is termed *conjunctive*. An event must always occur before *every* rule enabling it has expired. Since an event may be enabled by multiple rules, it is possible that the difference in time between the enabled event and some enabling events exceed the upper bound of their timing constraints, but not for all enabling events. These timing constraints are the same as *max constraints* [15] and *type 2 arcs* [25].

The conflict relation is added to model *disjunctive* behavior and choice. When two events e and e' are in conflict (denoted $e \# e'$), this specifies that either e can occur or e' can occur, but not both. Taking the conflict relation into account, if two rules have the same enabled event and conflicting enabling events, then only one of the two mutually exclusive enabling events needs to occur to cause the enabled event. This models a form of disjunctive causality. Choice is modeled when two rules have the same enabling event and conflicting enabled events. In this case, only one of the enabled events can occur.

The timed event/level structure is defined below in which $\mathcal{N} = \{0, 1, 2, 3, \ldots\}$:

Definition 3.1.1 *A timed event/level structure is* $S = \langle N, A, E, R, \# \rangle$ *where*

1. N *is the set of signals;*
2. A *is the set of atomic actions;*
3. $E \subseteq A \times \mathcal{N}$ *is the set of events;*
4. $R \subseteq E \times E \times \mathcal{N} \times (\mathcal{N} \cup \{\infty\}) \times (\mathrm{b} : \{0,1\}^{\mathcal{N}} \to \{0,1\})$ *is the set of rules;*
5. $\# \subseteq E \times E$ *is the conflict relation.*

Events are labeled using the function $L : E \to A$.

Figure 3.1(a) shows an example that expresses the conjunctive causality. In this TEL structure, $z-$ cannot occur until boolean expression $\langle a \rangle$ and $\langle b \vee c \rangle$ are true, and both events $x+$ and $y+$ occur. Figure 3.1(b) expresses a conflict, that is, either $y+$ or $z-$ can occur, but not both. Figure 3.1(c) shows the TEL structure of an AND gate. The event $c-$ can fire when $c+$ has fired for at least 1 time unit but not over 4 time units, and the boolean expression $\langle \neg a \vee \neg b \rangle$ is true. The event $c+$ can fire when $c-$ has fired for at least 2 time units but not over 6 time units, and the boolean expression $\langle a \wedge b \rangle$ is true.

3.2 Composition and Renaming of TEL Structures

Each process in a timed HSE specification or a VHDL specification is made up of a sequence of sequential constructs that are composed on operators specifying sequencing, concurrency, and choice. Therefore, we need to define a means of composing two TEL structures. To facilitate this composition, two sets are added temporarily to the TEL structure: *first* and *last*. Each element x of the *first* set is of the form: $x = \langle e, l, u, b \rangle$, where e is an event from the event set E, $\langle l, u \rangle$ is a timing constraint that is associated with e, and b is a boolean expression associated with e. The *last* set is simply a subset of the event set. These sets are not part of the final TEL structure and are simply information that must be recorded during the decomposition phase in order to allow the TEL structures to be composed correctly. Intuitively, the *first* set indicates which events are the first to occur in a TEL structure, and the *last* set indicates which events are the last to occur. Each event in the *first* set is also associated with a timing constraint and a boolean expression to indicate under what condition these events can occur. The composition of two TEL structures $S_0 = \langle N_0, A_0, E_0, R_0, \#_0, first_0, last_0 \rangle$ and $S_1 = \langle N_1, A_1, E_1, R_1, \#_1, first_1, last_1 \rangle$ (i.e., S_0 *op* S_1 where $op \in \{;, \|, |\}$) is defined as follows:

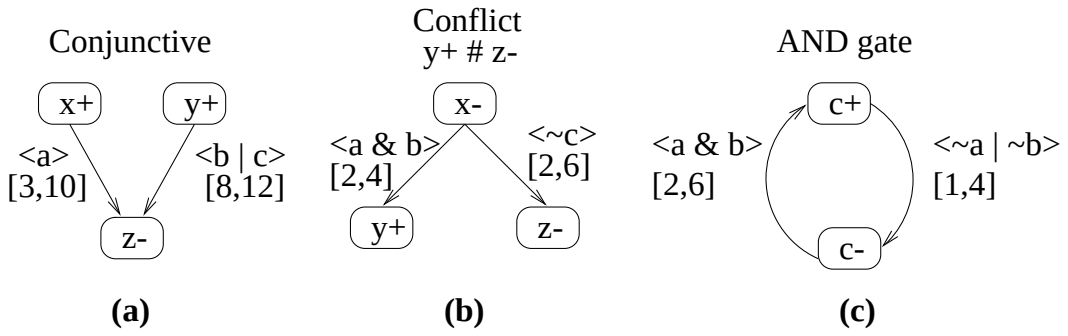

Figure 3.1. Examples of TEL structures.

$$
\begin{aligned}
N &= N_0 \cup N_1 \\
A &= A_0 \cup A_1 \\
E &= E_0 \cup E_1 \\
R &= R_0 \cup R_1 \cup \{\langle x, y.e, y.l, y.u, y.b\rangle \mid x \in \mathit{last}_0 \wedge y \in \mathit{first}_1 \wedge op = \ ;\} \\
\# &= \#_0 \cup \#_1 \cup \{(e, e') \mid (e \in E_0 \wedge e' \in E_1 \wedge op = \ |\ )\} \\
\mathit{first} &= \textbf{if } (\mathit{first}_0 = \emptyset \vee (\mathit{last}_0 = \emptyset \wedge \mathit{last}_1 = \emptyset) \vee op = \| \vee op = |) \textbf{ then } \mathit{first}_0 \cup \mathit{first}_1 \\
&\quad \textbf{else } \mathit{first}_0 \\
\mathit{last} &= \textbf{if } (\mathit{first}_1 = \emptyset \vee \mathit{last}_1 = \emptyset \vee op = \| \vee op = |) \textbf{ then } \mathit{last}_0 \cup \mathit{last}_1 \textbf{ else } \mathit{last}_1
\end{aligned}
$$

The sets of signals, actions, and events are simply merged. The rule set is similarly combined, but in the case in which $op = \ ;$ new rules are added from the last events in S_0 (i.e., the events in the set last_0) to the first events in S_1 (i.e., the events in the set first_1). These rules have the timing constraint and boolean condition associated with the first events. The conflict set is also merged, and if $op = |$ then every event in S_0 is set to conflict with every event in S_1. Finally, new *first* and *last* sets are created. If the structures are being composed in parallel or in conflict, or both last_0 and last_1 are empty, the sets are created by simply taking the union of the sets from each structure. If the structures are being composed in sequence, then in most cases the *first* set equals first_0, and the *last* set equals last_1. The exception is if first_0 is empty then the *first* set equals first_1, and if either first_1 or last_1 is empty then the *last* set is the union of the two last sets from the two structures.

Since a process is repetitive, the TEL structure describing its behavior is infinite. Due to its repetitive nature, however, this infinite behavior can be described with a finite model by adding an additional set of rules R' and an additional set of conflicts $\#'$. A *loop* set is also added temporarily to keep track of the last events before control loops back. When a TEL structure is created, these sets are all initialized to the empty set. To generate these sets, the composition operator is modified as follows:

$$
\begin{aligned}
R' &= R'_0 \cup R'_1 \cup \{\langle x, y.e, y.l, y.u, y.b\rangle \mid x \in \mathit{loop}_0 \wedge y \in \mathit{first}_1 \wedge op = \ ;\} \\
\#' &= \#'_0 \cup \#'_1 \cup \{(e, e') \mid (e \in \mathit{loop}_1 \wedge e' \in \mathit{last}_0 \wedge op = \ ;)\} \\
\mathit{loop} &= \textbf{if } (op = \| \vee op = |) \textbf{ then } \mathit{loop}_0 \cup \mathit{loop}_1 \textbf{ else } \emptyset
\end{aligned}
$$

The R' set is found by first taking the union of the corresponding sets from the structures that are being composed, and then when $op = \ ;$, new rules are added from events in the

$loop_0$ set to the $first_1$ set which creates a loop in the structure. Also, if $op =$; then the events in $last_0$ are set to conflict with the events in $loop_1$. As for the *loop* set, the events in the loop sets from the structures being composed in parallel or in conflict are simply merged and initialized to the empty set when composed in sequence. When control loops back, all events in the last set are moved into the *loop* set.

Before defining the infinite behavior of the TEL structure, we first introduce the renaming rules for TEL structures. When composing structures sequentially or in conflict, multiple occurrences of events with the same name are not allowed. Therefore, before doing the composition, we first resolve any name clashes using the function *rename* which takes two structures and returns the second structure with event names changed such that they do not clash with event names in the first structure. The function $rename(S_0, S_1)$ is defined as follows:

$$\begin{aligned}
N &= N_1 \\
A &= A_1 \\
E &= \{rename(E_0, e) \mid e \in E_1\} \\
R &= \{\langle rename(E_0, e), rename(E_0, f), l, u, b\rangle \mid \langle e, f, l, u, b\rangle \in R_1\} \\
R' &= \{\langle rename(E_0, e), rename(E_0, f), l, u, b\rangle \mid \langle e, f, l, u, b\rangle \in R_1'\} \\
\# &= \{(rename(E_0, e), rename(E_0, e')) \mid e \# e'\} \\
\#' &= \{(rename(E_0, e), rename(E_0, e')) \mid e \#' e'\} \\
first &= \{rename(E_0, e) \mid e \in first_1\} \\
last &= \{rename(E_0, e) \mid e \in last_1\} \\
loop &= \{rename(E_0, e) \mid e \in loop_1\}
\end{aligned}$$

The function *rename* is overloaded above to take a set of events E and a single event (a, i), and it renames (a, i) if there is a name clash with an event in the set E as follows:

$$\begin{aligned}
rename(E, (a, i)) = \; & \textbf{if } (\forall k (a, k) \notin E) \textbf{ then } (a, i) \textbf{ else } (a, i + j) \\
& \textbf{where } (a, j - 1) \in E \wedge (a, j) \notin E.
\end{aligned}$$

For a TEL structure of the form $S_0 = \langle N_0, A_0, E_0, R_0, \#_0, R_0', \#_0'\rangle$, we inductively define its infinite behavior as follows:

$$S_i = loop(S_0, S_0 \| rename(S_0, S_{i-1}))$$

where the function $loop(S_0, S_1)$ is defined as follows:

$$\begin{aligned} R &= R_1 \cup \{\langle e, rename(E_0, f), l, u, b\rangle \mid \langle e, f, l, u, b\rangle \in R_0'\} \\ \# &= \#_1 \cup \{(e, rename(E_0, e')) \mid e \#_0' e'\}. \end{aligned}$$

3.3 Compiling Timed HSE into TEL Structures

In order to interpret the behavior of a model specified in timed HSE, we translate it to a timed event/level (TEL) structure. We first describe how to interpret nonrepetitive constructs, then we describe how to interpret repetitive constructs.

To interpret nonrepetitive constructs, we define the function CTEL which takes a HSE specification and returns a TEL structure of the form: $S = \langle N, A, E, R, \#, first, last\rangle$. For simplicity, the R', $\#'$ and *loop* sets are not listed because they are used only for the repetitive constructs. This function iteratively decomposes the HSE specification into single actions and guards that are composed on the operators (; for sequencing, $\|$ for concurrency, and | for conflict), and it is defined as follows:

$$\begin{aligned} CTEL(p;q) &= CTEL(p); rename(CTEL(p), CTEL(q)), \\ CTEL(p\|q) &= CTEL(p)\|CTEL(q), \\ CTEL(p \mid q) &= CTEL(p) \mid rename(CTEL(p),\ CTEL(q)), \\ CTEL([G]) &= \langle\{s(G)\}, \{\$\}, \{(\$, 0)\}, \emptyset, \emptyset, \{\langle(\$, 0), (0, 0), G\rangle\}, \{(\$, 0)\}\rangle \\ CTEL(a+) &= \langle\{a\}, \{a+\}, \{(a+, 0)\}, \emptyset, \emptyset, \{\langle(a+, 0), D(a+), \mathbf{true}\rangle\}, \{(a+, 0)\}\rangle \\ CTEL(a-) &= \langle\{a\}, \{a-\}, \{(a-, 0)\}, \emptyset, \emptyset, \{\langle(a-, 0), D(a-), \mathbf{true}\rangle\}, \{(a-, 0)\}\rangle \end{aligned}$$

The first rule simply states that the TEL structure for p and q with a ';' between them is obtained by finding the TEL structures for p and q separately, renaming the TEL structure for q, if necessary, and composing these TEL structures using the sequencing operator (;). The second rule states that the TEL structure for p and q with a '$\|$' between them is obtained by finding the TEL structures for p and q separately, and composing these TEL structures using the parallel operator ($\|$). The third rule states that the TEL structure for p and q with a '|' between them is obtained by finding the TEL structures for p and q separately, and composing these TEL structures using the conflict operator (|). The next three rules are for generating TEL structures for a single *guard* and a single *action*. The TEL structure for a *guard* consists of the signals in the support set of the boolean expression (i.e. *s(G)*), the dummy action \$, and a dummy event (\$,0). The

first and *last* sets are initialized to include the dummy event, and the boolean expression associated with the dummy event in the *first* set is set to the boolean expression indicated in the *guard*. Since this event does not cause a signal transition, its timing constraint is $\langle 0, 0 \rangle$. Next, if the input to the function is a single *action*, the function returns a TEL structure with a single signal, a single action, a single event, and both the *first* and *last* sets initialized to include that event, and the timing constraint of the event, for example $a+$, in the *first* set is given by the function $D(a+)$ which returns the bounded delay declared for that signal. The TEL structures of a single guard $[b]$ and a single action $a+$ are shown in Figure 3.2 (a) and (b). Note that in Figure 3.2 (a) there is no enabling event, which means the TEL structure contains no rules, and simply stores the boolean expression in the *first* set.

For a guarded command $[G \rightarrow S]$, since the program part S cannot execute until the guard G evaluates to **true**, it is interpreted as

$$CTEL([G \rightarrow S]) \; = \; CTEL([G]; S)$$

The TEL structure for an example guarded command $[b \rightarrow a+]$ is shown in Figure 3.2. When composing TEL structures, we try to minimize the number of dummy events. In this case, since the last event in the TEL structure for the guard $[b]$ is a dummy event, and the first event in the TEL structure for the action $a+$ is a signal event with a boolean expression **true**, the dummy event can be canceled without changing the meaning of the command by replacing the boolean expression of the first event in the TEL structure for the action $a+$ by the boolean expression in the guard.

For a *selection* construct $[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n]$, the process containing

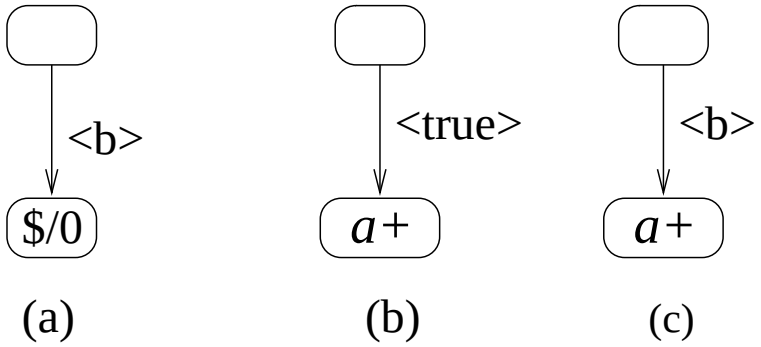

Figure 3.2. TEL structures for a guard $[b]$, an action $a+$ and a guarded command $[b \rightarrow a+]$.

it suspends until one G_i evaluates to **true**, then S_i executes. Each guarded command is interpreted similarly. Then, the TEL structures for all guarded commands are composed in conflict. Therefore, the TEL structure for a *selection* is defined as follows:

$$\begin{aligned} T_1 &= CTEL([G_1]; S_1) \\ T_2 &= T_1 \mid rename(T_1, CTEL([G_2]; S_2)) \\ &\cdots \\ result &= T_{n-1} \mid rename(T_{n-1}, CTEL([G_n]; S_n)) \end{aligned}$$

The resulting TEL structure is shown in Figure 3.3. In this case, all sets in the TEL structures are simply merged, and their events are set to conflict.

To interpret loop constructs, we define two functions. The function *mv* takes a TEL structure, and moves its *last* set to the *loop* set. The function *make_loop* also takes a TEL structure, and makes new rules from the *loop* set to the *first* set. Its definition is shown in Figure 3.4.

There are two kinds of loop constructs in timed HSE: repetitive commands and infinite loops. The infinite loop, $*[S]$, where S is an arbitrary program part, is easy to interpret. The TEL structure of S is created first, then it is made to loop back to form an infinite loop. The creation of a TEL structure for an infinite loop is defined as follows:

$$make\_loop(mv(CTEL(S)))$$

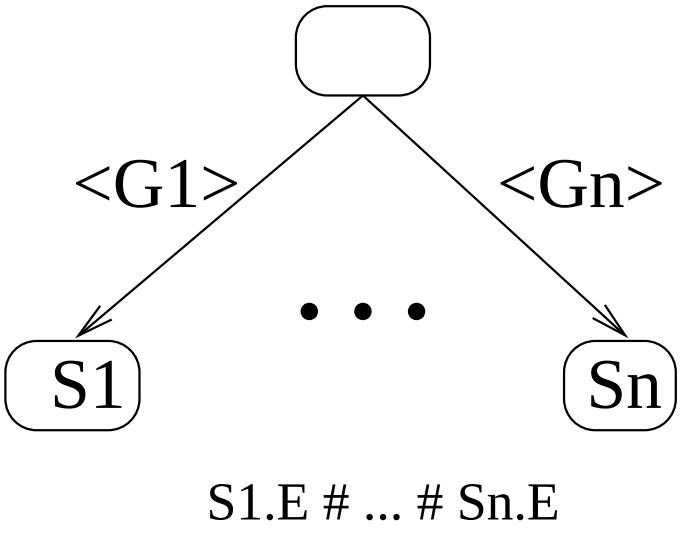

Figure 3.3. The TEL structure for a $[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n]$.

$$
\begin{array}{l}
TEL\ make\_loop(TEL\ S)\{ \\
\quad t\ =\ S.last \\
\quad S.last\ =\ \emptyset \\
\quad S\ =\ S;S \\
\quad S.last\ =\ t \\
\quad return\ S \\
\}
\end{array}
$$

Figure 3.4. The definition of the function *make_loop*.

Another looping construct is the selective repetition command shown as follows:

$$[G_1 \rightarrow S_1 \mid \ldots \mid G_n \rightarrow S_n; *]$$

This construct suspends the process until one of guards G_i eveluates to **true**, then the following program part S_i executes. If the S_i has a '$*$' following it, the control loops back to the beginning of the construct. Otherwise, the control moves to the next command. To interpret this construct, a TEL structure for each guarded command is created. Assume there is a '$*$' following S_n, the function *mv* is called to move the *last* set of the TEL structure for $G_n \rightarrow S_n$ to its *loop* set. Then, the TEL structures for guarded commands are composed in conflict, and the function *make_loop* is called to make loops. The creation of a TEL structure for a selective repetition command is shown as follows:

$$
\begin{aligned}
T_1 &= CTEL([G_1 \rightarrow S_1]) \\
T_2 &= T_1 \mid rename(T_1, CTEL([G_2 \rightarrow S_2])) \\
&\ldots \\
T_n &= T_{n-1} \mid rename(T_{n-1}, mv(CTEL([G_n \rightarrow S_n]))) \\
result &= make\_loop(T_n)
\end{aligned}
$$

An example TEL structure is shown in Figure 3.5.

For a module with multiple processes, its TEL structure is formed by composing all processes in parallel, i.e.,

$$CTEL(P_0 \ldots P_n) \quad = \quad CTEL(P_0) \| \ldots \| CTEL(P_n).$$

3.4 Compiling VHDL Specifications into TEL Structures

In order to interpret the behavior of a model described in our synthesizable subset of VHDL, we translate it to a TEL structure. First, each process within each architecture is

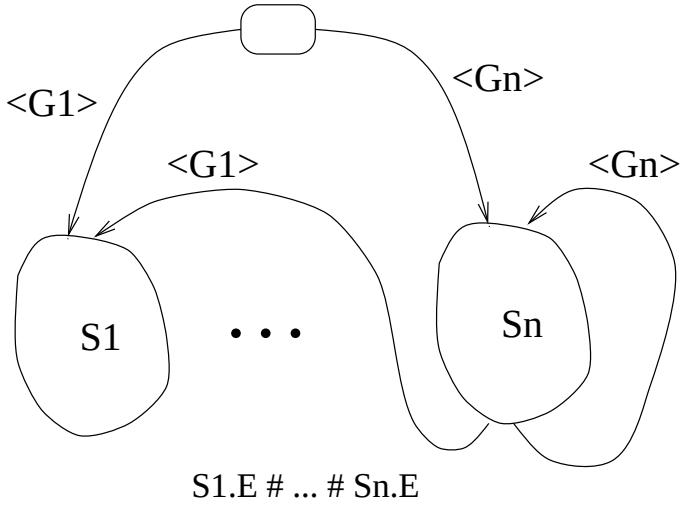

Figure 3.5. The TEL structure for $[G_1 \rightarrow S_1 | \ldots | G_n \rightarrow S_n; *]$.

iteratively decomposed until it is made up of only signal assignment statements and simple wait statements. A simple wait statement is one which is composed of an expression that includes only a single sum-of-products boolean condition. This section first describes a method to interpret a sequence of sequential statements which is then extended to interpret concurrent statements.

3.4.1 Interpretation of Sequential Statements

We first interpret each process within each architecture ignoring repetition. Repetition is addressed in the next subsection. To interpret a nonrepetitive process, we define the function *VTEL* which takes a VHDL specification and returns a TEL structure of the form: $S = \langle N, A, E, R, \#, \mathit{first}, \mathit{last} \rangle$. This function iteratively decomposes the VHDL specification into signal assignment statements and wait statements that are composed on the operators (; for sequencing, $\|$ for concurrency, and | for conflict), and it is defined as follows:

$$
\begin{aligned}
VTEL(p;q) &= VTEL(p); \mathit{rename}(VTEL(p), VTEL(q)) \\
VTEL(\textbf{wait until } b) &= VTEL([b]) \\
VTEL([b]) &= \langle \{s(b)\}, \{\$\}, \{(\$,0)\}, \emptyset, \emptyset, \{\langle(\$,0), 0, 0, b\rangle\}, \{(\$,0)\}\rangle \\
VTEL(a \Leftarrow' 1' \textbf{ after } \mathrm{delay}(l,u)) &= \langle \{a\}, \{a+\}, \{(a+,0)\}, \emptyset, \emptyset, \{\langle(a+,0), l, u, \textbf{true}\rangle\}, \emptyset\rangle \\
VTEL(a \Leftarrow' 0' \textbf{ after } \mathrm{delay}(l,u)) &= \langle \{a\}, \{a-\}, \{(a-,0)\}, \emptyset, \emptyset, \{\langle(a-,0), l, u, \textbf{true}\rangle\}, \emptyset\rangle \\
VTEL(z = \textbf{ int}) &= \langle \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset, \emptyset\rangle
\end{aligned}
$$

The first rule in the definition states that the structure for two sequential statements $p; q$ is obtained by finding the TEL structures for p and q, renaming the events in the structure for q, if necessary, and composing these structures using the sequencing operator (;). The next four rules are for generating TEL structures for a single *wait* and a *signal assignment* statement. First, if the input to the function is a single *wait* statement, the function returns a structure which consists of the signals in the support set of the boolean expression (i.e. *s(b)*), a dummy action \$, and a dummy event (\$,0). The *first* and the *last* set are initialized to include that dummy event $(\$, 0)$, and the boolean expression associated with the dummy event in the *first* set is set to the boolean condition indicated in the *wait* statement. Since this dummy event does not cause a signal transition, its timing constraint is $\langle 0, 0 \rangle$. If the input to the function is a *signal assignment* statement, the function returns a structure with a single signal, a single action, a single event, and the *first* set is initialized to include that event along with the lower and upper bound specified by the delay function and the boolean expression **true**. Note that this event is not put in the *last* set, because completion of signal assignments is not waited for by following statements in VHDL. Finally, if the input to the function is a test of a variable, the function returns an empty structure. These tests are on random variables and appear in the *if-then-elsif* clauses in the environment.

In order to generate the structure for an *if* statement which has the following form:

if b **then** p **end if**;

a TEL structure is obtained for **wait until** b, and it is composed sequentially with a renamed version of the TEL structure for p. And also a TEL structure is obtained for **wait until** $\neg b$ to define the exit condition of the *if* statement. These two TEL structures are composed in conflict. The generation of the TEL structure for an *if-then* clause is defined as follow:

$$VTEL([b]; p) \mid rename(VTEL([b]; p), VTEL([\neg b]))$$

Similarly, the creation of an *if-then-else* clause which has the following form

if b **then** p **else** q **end if**;

is defined as

$$VTEL([b]; p) \mid rename(VTEL([b]; p), VTEL([\neg b]; q))$$

In order to generate the structure for an *if* statement which has the following form:

if b_1 **then** p_1 **elsif** b_2 **then** p_2 **else** p_3 **end if**;

A TEL structure for the *if-then* branch is generated similarly as described above, and composed in conflict with one that checks if the expression is false. If there is an *elsif-then* clause, a TEL structure is created for the *elsif-then* clause, and composed in conflict with one that checks if the expression in the *elsif-then* clause is false. If there is an *else* clause, its TEL structure is composed sequentially with the second TEL structure of the *elsif-then* clause. Finally, the TEL structure for the *elsif-then-else* clause is composed sequentially with the second TEL structure for the *if-then* branch. The creation of the TEL structure for an *if-then-elsif-else* clause is defined as follows:

$$\begin{aligned} T &= VTEL([b_2]; p_2) \mid rename(VTEL([b_2]; p_2), VTEL([\neg b_2]; p_3)). \\ result &= VTEL([b_1]; p_1) \mid rename(VTEL([b_1]; p_1), VTEL([\neg b_1]; T)); \end{aligned}$$

Figure 3.6 shows the graphical representation of the TEL structures for a *signal assignment*, a *wait* and an individual *if* statement. Figure 3.7 shows the graphical representation of the TEL structure for an *if-then-elsif-else* statement.

The generation of TEL structures for infinite loops is described in the first section of this chapter. In this section, we describe how to interpret a *while* loop. Interpreting a *while* statement is a little different from interpreting an infinite loop. Given a *while* loop, we first create an entry condition TEL structure and an exit condition TEL structure. The entry condition TEL structure is the TEL structure for a *wait* statement with a

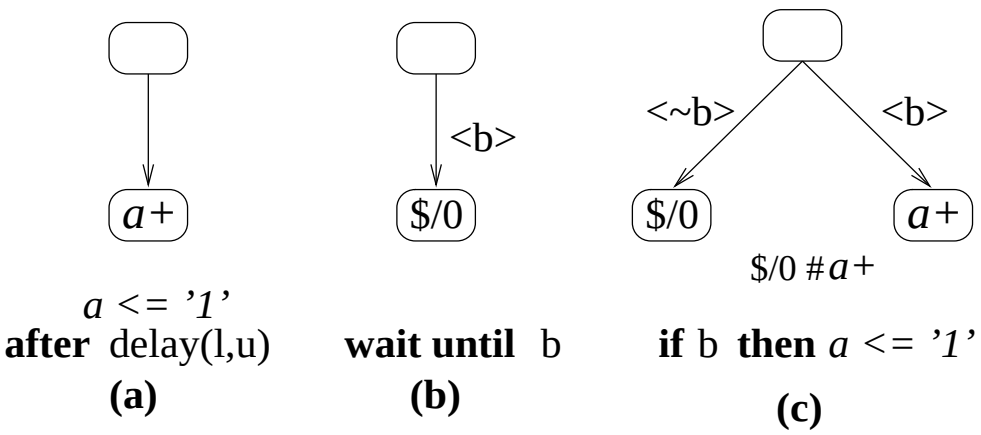

Figure 3.6. TEL structures for a wait, signal assignment and if statement.

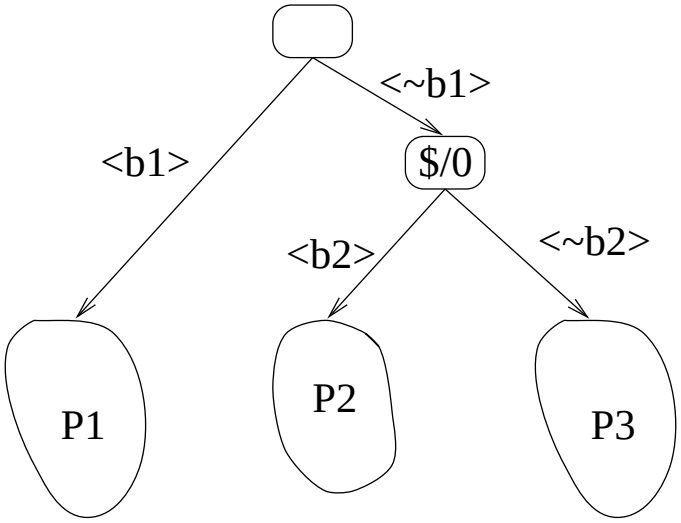

Figure 3.7. The TEL structure for an if-then-elsif statement.

boolean condition which is the same as that of *while* loop condition. This TEL structure is composed with that of the loop body sequentially and the function *mv* is called for the result. This is not complete because of the lack of an exit condition. The exit condition TEL structure is the TEL structure for a *wait* statement with a boolean condition that is opposite to the condition of the *while* loop. The exit TEL structure indicates when the *while* loop terminates. Since a *while* loop is either in the loop or terminated, the exit TEL structure and the above incomplete *while* loop TEL structure are composed in conflict. Then the function *make_loop* is called to make a loop. For a *while* loop statement:

while b **loop** p **end loop**;

the creation of the TEL structure for it is defined as follows:

$$\begin{aligned} T_1 &= mv(VTEL([b]; p)) \\ T_2 &= \mathrm{V}TEL([\neg b]) \mid rename(VTEL([\neg b]), T_1) \\ result &= make\_loop(T_2) \end{aligned}$$

The graphical view of the TEL structure for a *while* loop is shown in Figure 3.8.

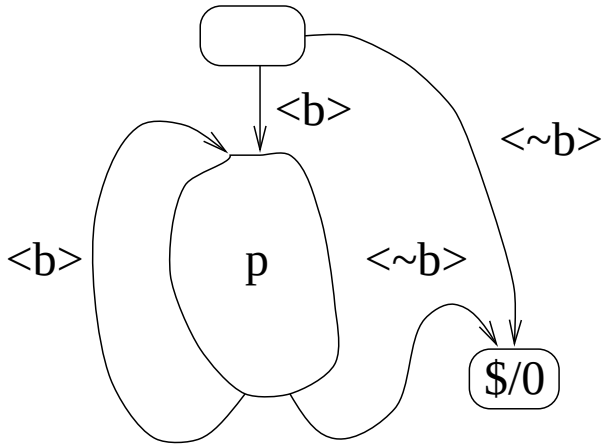

Figure 3.8. The TEL structure of a while loop.

3.4.2 Interpretation of Concurrent Statements

Component instantiation statements and process statements are the concurrent statements allowed in this subset. To interpret a component instantiation statement, the compiler first searches for an entity with the same name as the component name, and creates a TEL structure for it. This TEL structure is then returned to the component instantiation statement, and all signal names in the TEL structure are replaced by actual signal names in the statement. The operation of a process is an infinite loop, thus, after TEL structures for all sequential statements in the process are created and composed, the function *make_loop* is called to make it an infinite loop.

In order to obtain the TEL structure for a complete model, it is now simply a matter of composing all the individual processes P_i and component instantiation statements within each architecture in parallel, i.e.,

$$TEL(P_0 \ldots P_n) \quad = \quad TEL(P_0) \| \ldots \| TEL(P_n).$$

CHAPTER 4

EXAMPLES

A compiler from timed HSE and our synthesizable subset of VHDL to TEL structures using the procedure described in the previous chapter has been incorporated into the timed circuit design tool `ATACS`. The compiler recognizes the entire VHDL-93 language, but it only synthesizes the subset described in this thesis. This chapter describes the specification and design of two examples. The first example is the *sbuf* controller from the HP Post Office [8] benchmark suite. The second example is the controller for our asynchronous implementation of the Maxlist algorithm [21].

4.1 Sbuf Controller

The *Sbuf* controller is used to manage the transfer of packets between a sender and a receiver. First, the receiver sets *req* to high, which requests a line to be sent from the sender. Then, the sender sends the line and raises *sendline*. When the receiver reads the line, it acknowledges the sender by raising *ackline*. Then, the sender lowers *sendline*, and the receiver responds by lowering *ackline*. This protocol continues until the receiver chooses to terminate it. To terminate the packet transfer, the receiver sets *done* high sometime after the falling transition of *sendline* but before it raises *ackline* again. When the sender detects that *done* has risen, it lowers *sendline* and also raises *ack*, indicating it has detected that the packet transfer is over. The receiver then lowers *req, ackline*, and *done* in parallel and the sender, in response to this, lowers *ack*. The corresponding timed HSE code for the *sbuf* body and its environment is shown in Figure 4.1.

The TEL structure for the *Sbuf* body generated by `ATACS` is shown in Figure 4.2. Note that there is a circle across some lines, that means those lines have the same boolean condition associated with them. The rest of this section describes the details of how the TEL structure is generated from the timed HSE code.

The compiler scans and decomposes the HSE code until it finds the basic language constructs. It first recognizes [*req*] and *sendline*+ and generates the TEL structures,

```
module sbuf;
  delay d = ⟨2,5⟩;

  input req = { false, d };
  input ackline = { false, d };
  input done = { true, d };
  output ack = { false, d };
  output sendline = { true, d };

  process sbufbody;
  ∗[[req]; sendline+;
    [¬done ∧ ackline → sendline−; [¬ackline]; sendline+; ∗
    | done ∧ ackline → (ack + ‖sendline−); [¬req ∧ ¬ackline ∧ ¬done]; ack−;]]
  endprocess

  process env;
  ∗[req+; [sendline]; ackline+;
    [¬sendline → (done + ‖ackline−); [sendline]; ackline+;
      [¬sendline ∧ ack]; (req − ‖ackline − ‖done−); [¬ack]
    |¬sendline → ackline−; [sendline]; ackline+; ∗
    ]
  ]
  endprocess
endmodule
```

Figure 4.1. The timed HSE code for the sbuf controller.

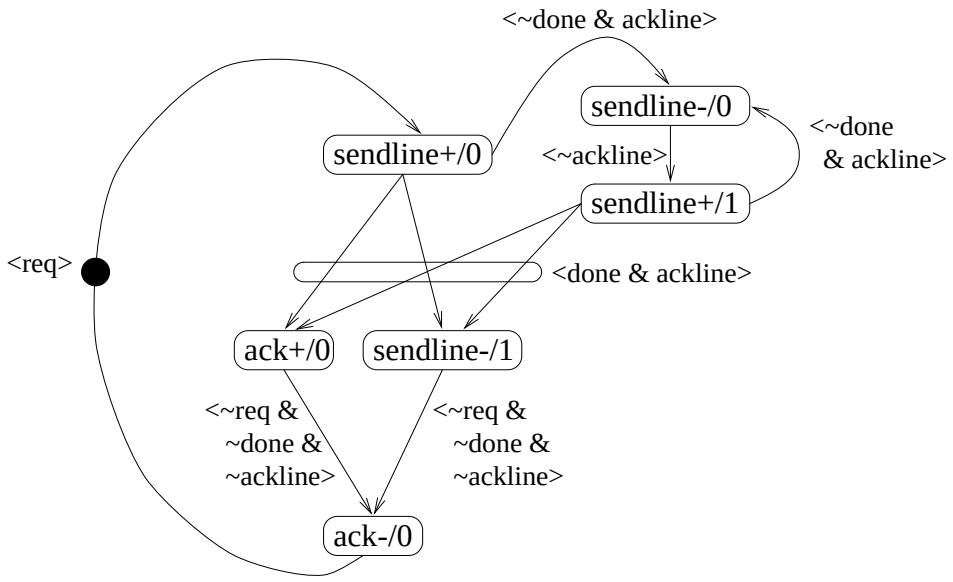

Figure 4.2. The TEL structure for the body of the sbuf controller.

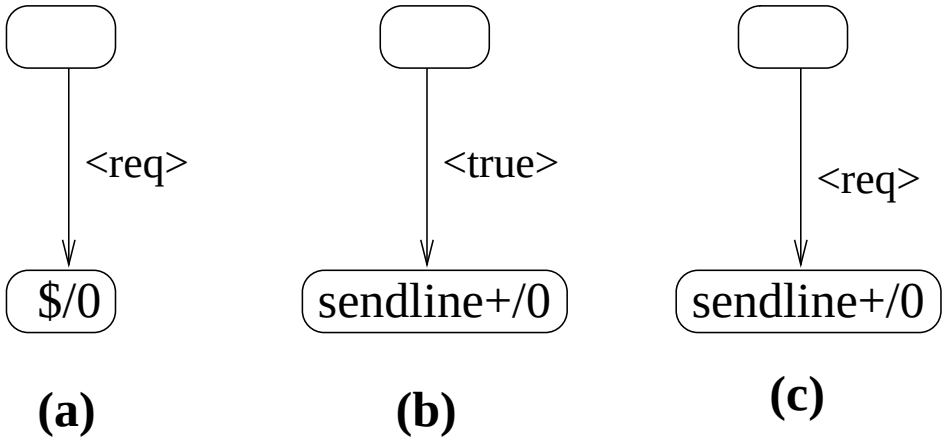

Figure 4.3. The TEL structure for $[req]; sendline+$.

shown in Figure 4.3 (a) and (b). Figure 4.3 (c) shows the composition of the TEL structures shown in (a) and (b) with a sequencing operator ';'.

Now the compiler tries to generate the TEL structure for the selection construct following the first *sendline*+. There are two guarded commands in the selection. As noticed, there is a '$*$' sign following the first guarded command. It says that after this guarded command is executed, the control loops back to the beginning of the *selection* command. For the first guarded command, the TEL structures for the guard and the program part are generated similarly, shown in Figure 4.4 (a) and (b), respectively. Those two TEL structures are composed sequentially as shown Figure 4.4 (c). Because after the execution of that guarded command, the *selection* command will start from the beginning, the function *mv* is called to move the event *sendline*+/2 into the *loop* set as described in the last chapter. This event is used to make a loop back to the beginning of the command. For the second guarded command, the guard is interpreted similarly. In the program part

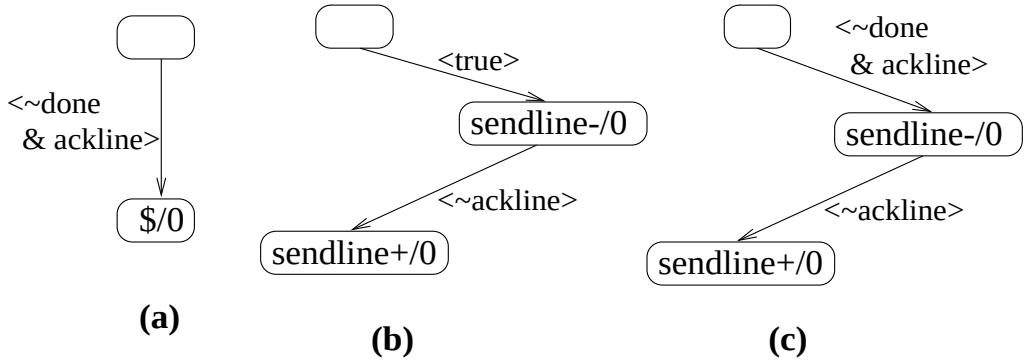

Figure 4.4. $\neg done \wedge ackline \rightarrow sendline-; [\neg ackline]; sendline+; *$.

following the guard, there are two actions executed in parallel, the TEL structures for the two actions are generated separately, and then composed concurrently. The TEL structures for the second guard and the program part following the second guard are shown in Figure 4.5 (a) and (b), respectively. These two TEL structures are composed sequentially to form the TEL structure for the second guarded command, which is shown in Figure 4.5 (c). Now we have the TEL structures for both guarded commands, they are composed in conflict to form the TEL structure for the *selection* construct. Composing TEL structures in conflict is similar to composing them concurrently, all sets in them are simply merged, except that conflicts are generated when composing in conflict. Note that the function *rename* is called to resolve the name clashes. Since there are two occurrences of $sendline-$, the occurrence index of the $sendline-$ in the second guarded command is changed to 1. And also we know if the first command executes, the control loops back to the beginning of the *selection* command. Therefore, the function *make_loop* is called for the *selection* command to make a loop back to the beginning of the command. Ignoring conflicts here, the TEL structure for the *selection* command is shown in Figure 4.6.

The nonrepetitive TEL structures for the process are shown in Figure 4.3 (c) and Figure 4.6, respectively. Composing them sequentially forms the nonrepetitive TEL structure for the process. The TEL structure of the process is shown in Figure 4.7. Finally, the function *mv* is called to the event $ack-/1$ to the *loop* set, and the function *make_loop* is called to make a loop back to the beginning of the process. The result is shown in Figure 4.2.

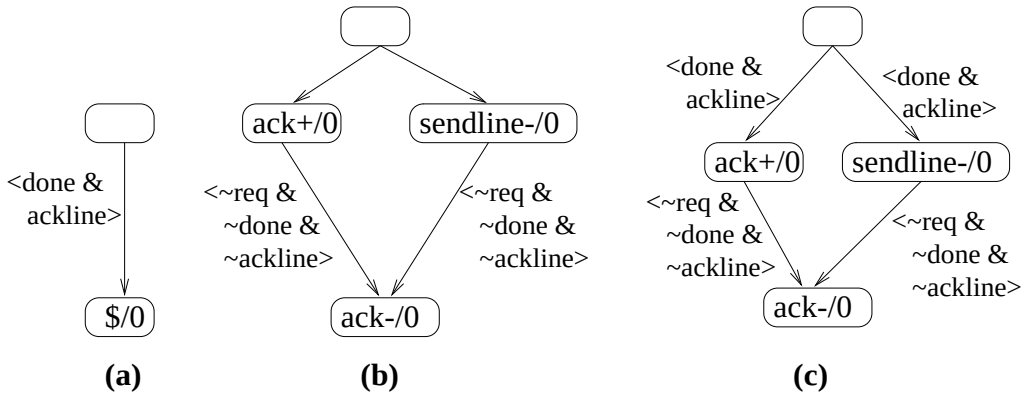

Figure 4.5. $done \wedge ackline \rightarrow (ack+ \| sendline-); [\neg req \wedge \neg ackline \wedge \neg done]; ack-$.

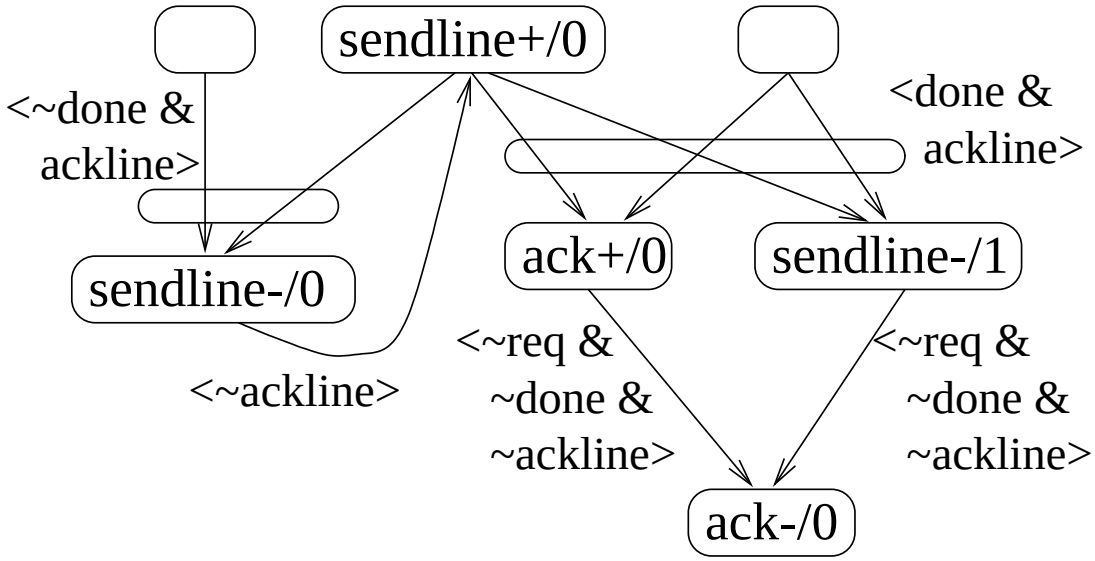

Figure 4.6. The TEL structure for the *selection* construct.

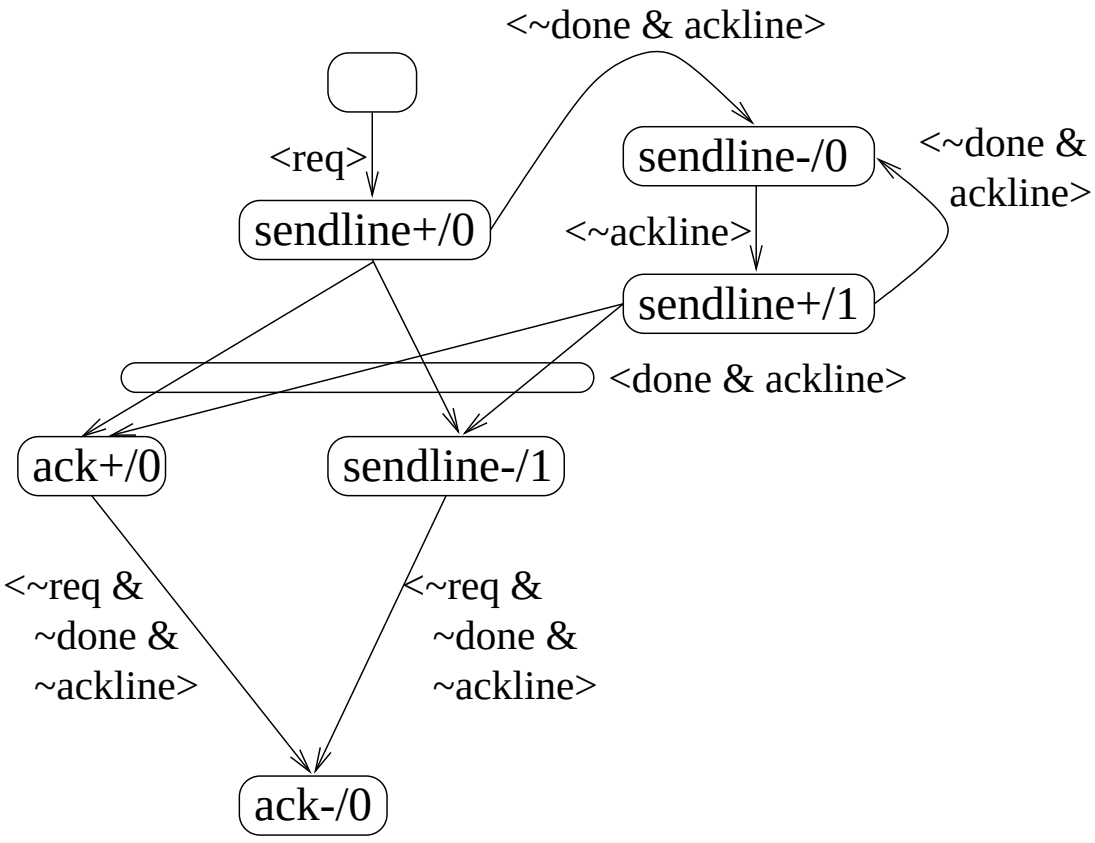

Figure 4.7. The TEL structure for the nonrepetitive process.

4.2 The Maxlist

Many signal and image processing algorithms require the calculation of a running minimum or maximum over a sliding data window. For example, in a normalized least-mean-square (NLMS) adaptation algorithm given in [10], the filter coefficient which is chosen to be modified is the one which is associated with the input sample with the largest absolute value in the window of samples currently in the filter.

In [12], an efficient algorithm is presented for such calculations. This algorithm stores data elements in a pruned list. The data elements which are stored are those which are currently or have the potential of becoming the maximum or minimum within the sliding data window. This pruned list can be substantially smaller than the actual size of the sliding window.

We specified and implemented an asynchronous architecture for the MAXLIST algorithm. We have designed and simulated it in VHDL on a large set of correlated random data samples. Our results show a wide variation in delay due to both data-dependencies and operating conditions. We compare our asynchronous design with an existing synchronous design and the best possible synchronous design with an architecture comparable to ours.

4.2.1 Algorithm

The MAXLIST algorithm generates the pruned list of potential maxima (or minima) as follows. When a new element arrives, it firsts checks to see if an element already on the list has fallen out of the sliding window. If it has, it is removed from the list. Next, it searches the list until it finds the smallest element which is larger than the new element. It adds the new element after this one, and it removes all smaller elements since they will never become the maximum across the window.

An example (courtesy of [12]) is shown in Figure 4.8. In this example, the sliding window is 6 elements long. Initially, element 3 is in the list because it is the maximum, and element 6 is in the list because it is a potential maximum. Elements 1 and 2 are dominated by element 3 since it is larger and appears later in the list. Elements 4 and 5 are dominated by element 6. At time 1, the window shifts, and element 7 is added to the list. At time 2, the window shifts again, element 8 is added, and since it dominates 6 and 7, they are removed from the list. At time 3, element 3 slides out of the window, and element 9 is added, dominating element 8.

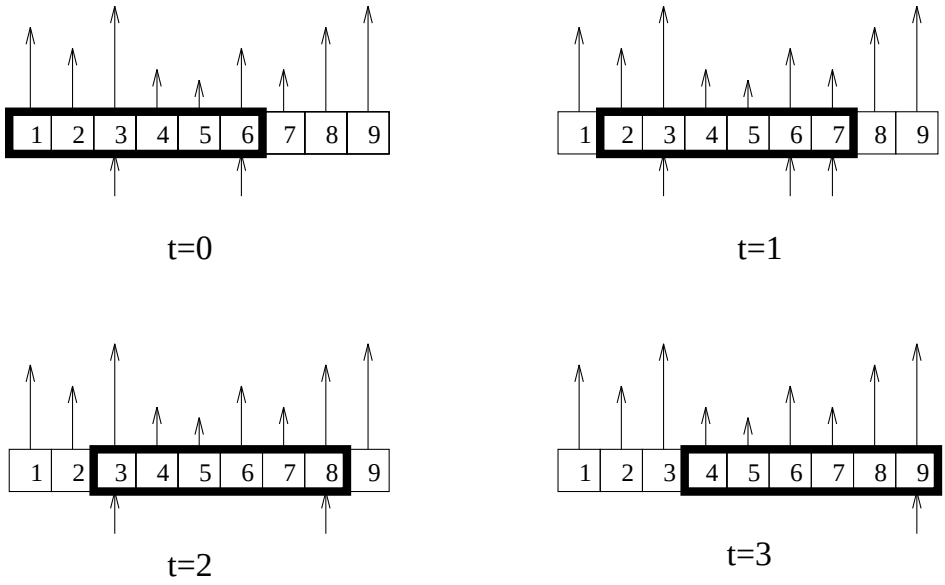

Figure 4.8. Example of the MAXLIST algorithm.

By construction, the elements in the list are ordered by size and age. The head of the list is always the maximum and always the oldest element. The remaining elements have the potential to become a maximum as larger, older elements fall out of the sliding window.

In hardware, the pruned list must be of fixed size. If this size is less than the window size, it is possible that the running maximum or minimum may be in error. In [12], it is shown that the average size of the pruned list for random data goes like $ln(n)$ where n is the size of the window. Since small errors can usually be tolerated in signal and image processing algorithms, the list size is usually chosen to be slightly larger than $ln(n)$.

4.2.2 Architecture

In our asynchronous architecture, we have chosen to compute the maximum over a sliding window of 256 elements with a list size of 8 elements, where each element is represented as an 8-bit value. It is relatively straightforward to adapt our architecture to minimum calculations and to different size windows and lists.

One important architecture decision is how to search the list to find the location where a new element should be inserted. Our initial architecture began the search at the

beginning of the list (i.e., the current maximum element) and worked towards the end. It was brought to our attention that this may result in more comparisons than necessary [11]. As shown in Figures 4.9 and 4.10, by starting the search at the end of the list (i.e., the smallest potential maximum or newest element) and searching backwards, the average number of comparisons is reduced from 5.5 to only 1.4.

Our architecture, depicted in Figure 4.11, is composed of seven main parts: an input latch, a counter, a FIFO, two comparators, an output latch, and a controller. In each data cycle, the following events occur:

1. When the request signal goes high, the data is latched, and the counter is incremented.

2. The current count and the position of the maximum are compared. If they are equal, the maximum has fallen out of the window, and it is shifted out of the FIFO.

3. The new data element is compared with each element in the list beginning with the most recently added element until the insertion position has been found.

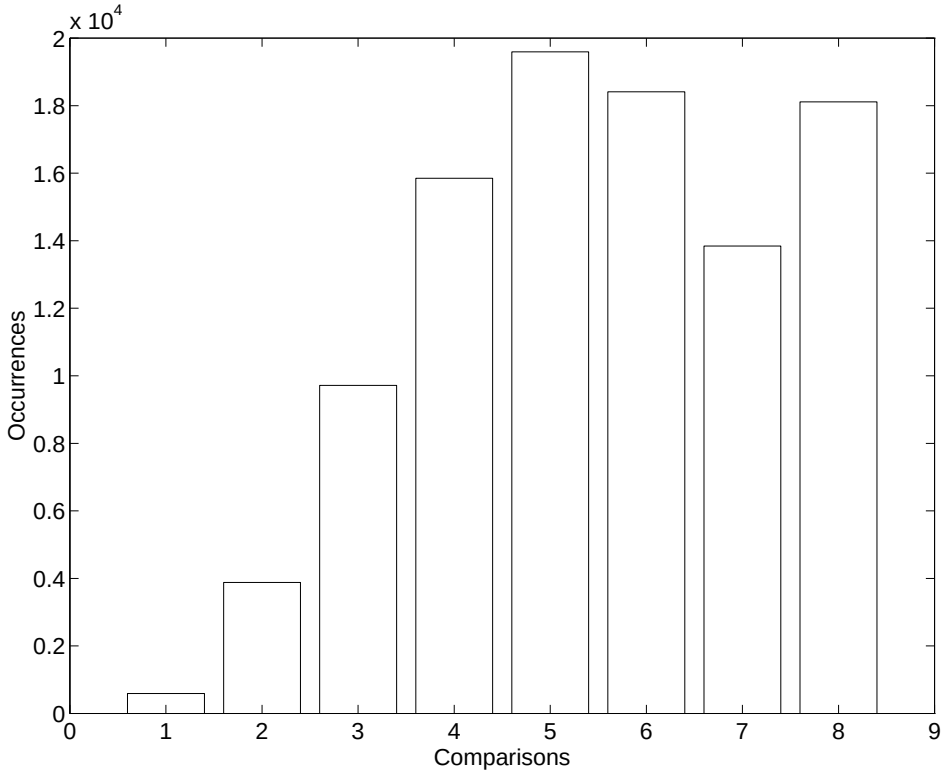

Figure 4.9. Distribution of forward comparisons.

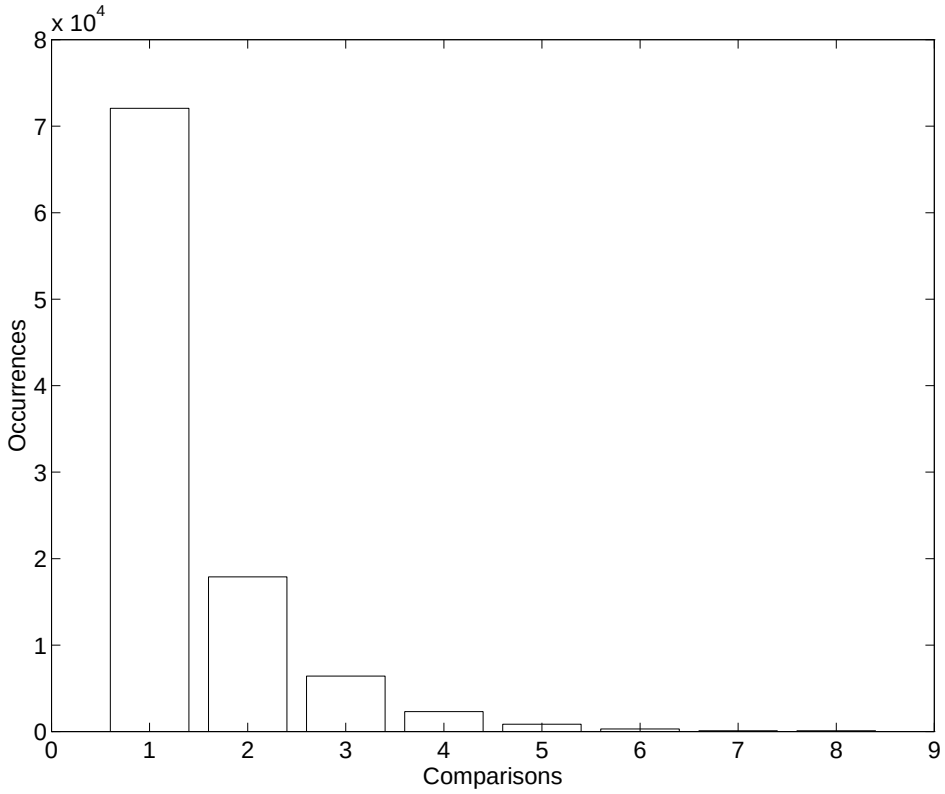

Figure 4.10. Distribution of backward comparisons.

4. The new data element is placed in the location of the oldest element that it is greater than or equal to. If it is smaller than all elements in the list, it is placed in the first empty location. If the list is full, the element is discarded.

5. The maximum data element and its position are output, and the acknowledge signal is asserted.

4.2.3 Implementation

The major blocks that must be implemented in our asynchronous MAXLIST architecture are the FIFO, two comparators, and the controller. The structure of the FIFO is shown in Figure 4.12. The FIFO must be able to shift data when the element at the head of the list has left the data window, put data on the *CMP* bus for the search through the list, and accept inserted data at arbitrary locations while clearing all subsequent locations. The information stored in the FIFO is composed of three parts: a Full/Empty bit, the position (i.e., the count when the data arrived), and the data itself.

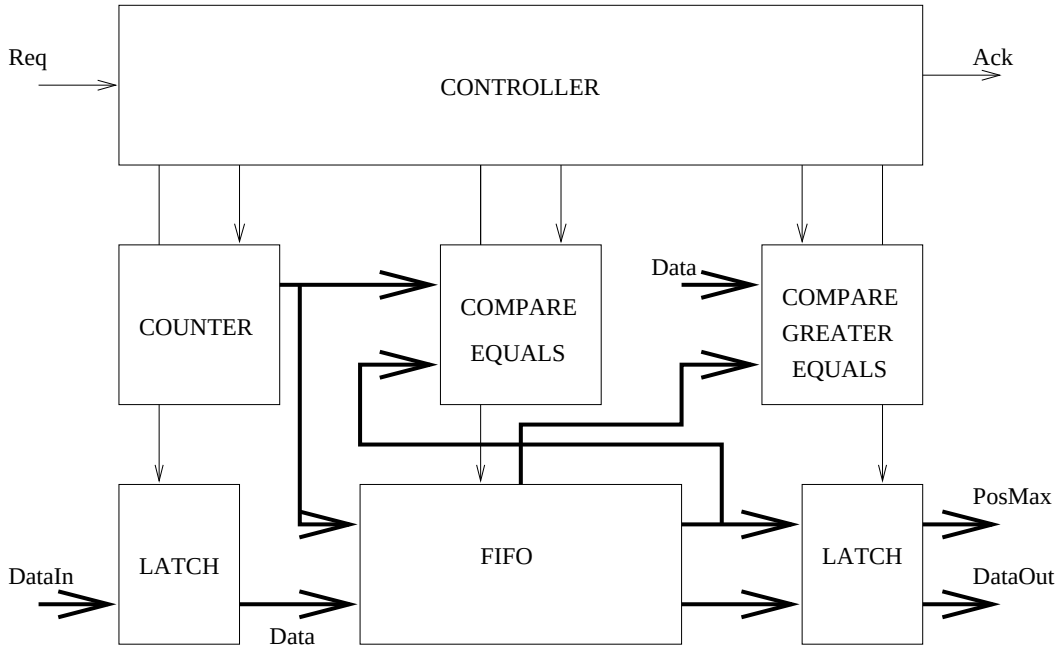

Figure 4.11. Overall block diagram.

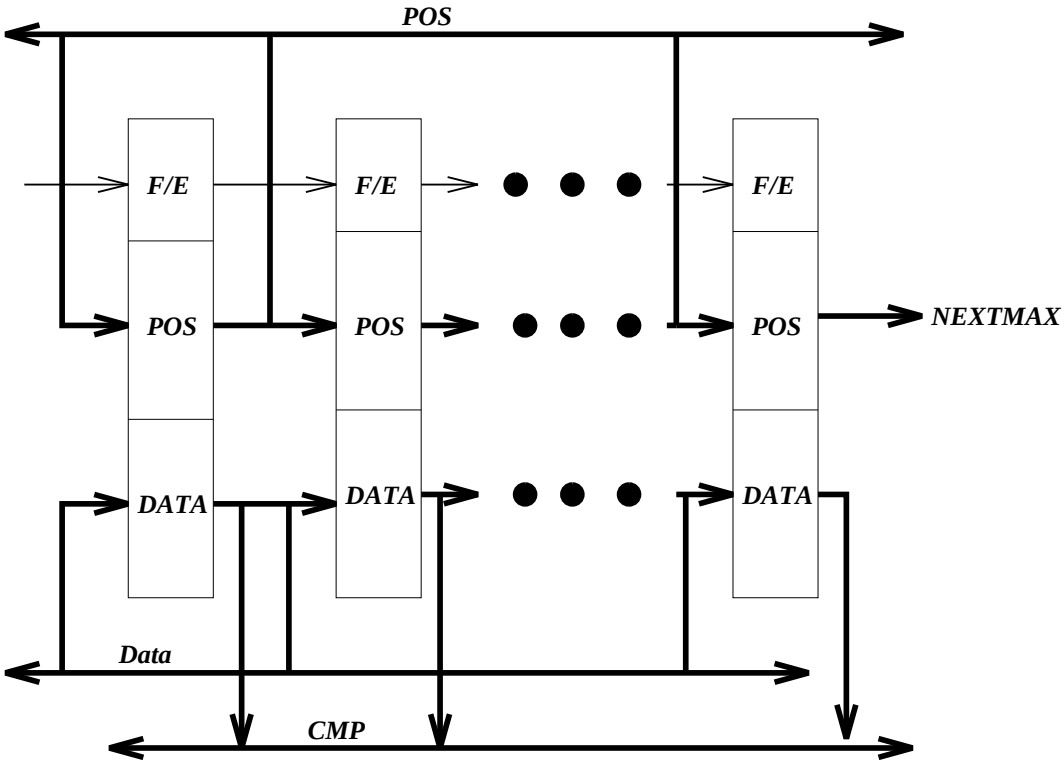

Figure 4.12. Block diagram of the FIFO.

The comparator is composed of eight 1-bit comparators as shown in Figure 4.13. It is started with a request to the highest order bit. Each bit of the comparator returns whether *ai* is greater than (*gt*), less than (*lt*), or equal (*eq*) to *bi*. Only one of the three outputs can be asserted at any time. For the *compare equals* block, *gt* and *lt* are combined by an *or* gate to generate not equal (*neq*), so *neq* is returned when any bit returns *gt* or *lt*, otherwise, *eq* is returned. For the *compare greater equals* block, *gt* and *eq* are combined by an *or* gate to generate greater than or equal (*ge*), *ge* is returned for *gt* and less than (*lt*) is returned for *lt*. If the two bits are equal, the next bit is compared. Finally, if the last bit returns *eq*, then *ge* is also returned for the *compare greater equals* block. This block is highly data-dependent as the comparison may complete at varying times. The asynchronous design methodology takes advantage of this data-dependency to produce a more efficient architecture.

The last important block is the controller. This block is split into ten separate control blocks as shown in Figure 4.14. The *main* block accepts the request when a new datum

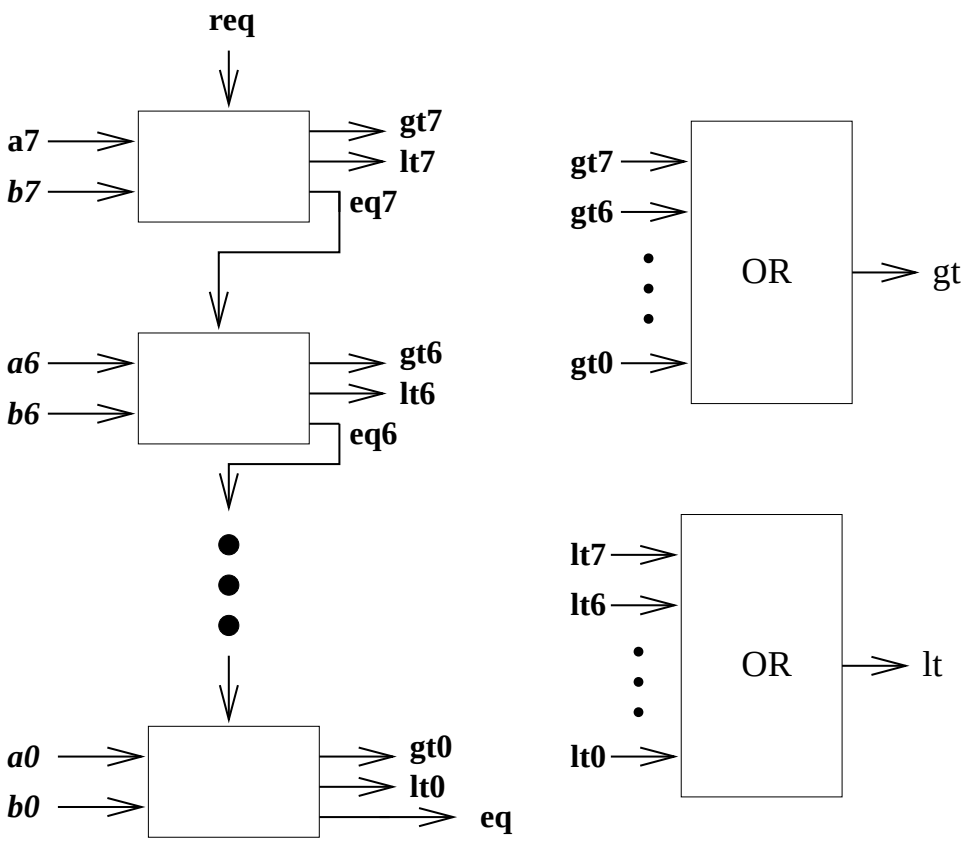

Figure 4.13. Block diagram of a comparator.

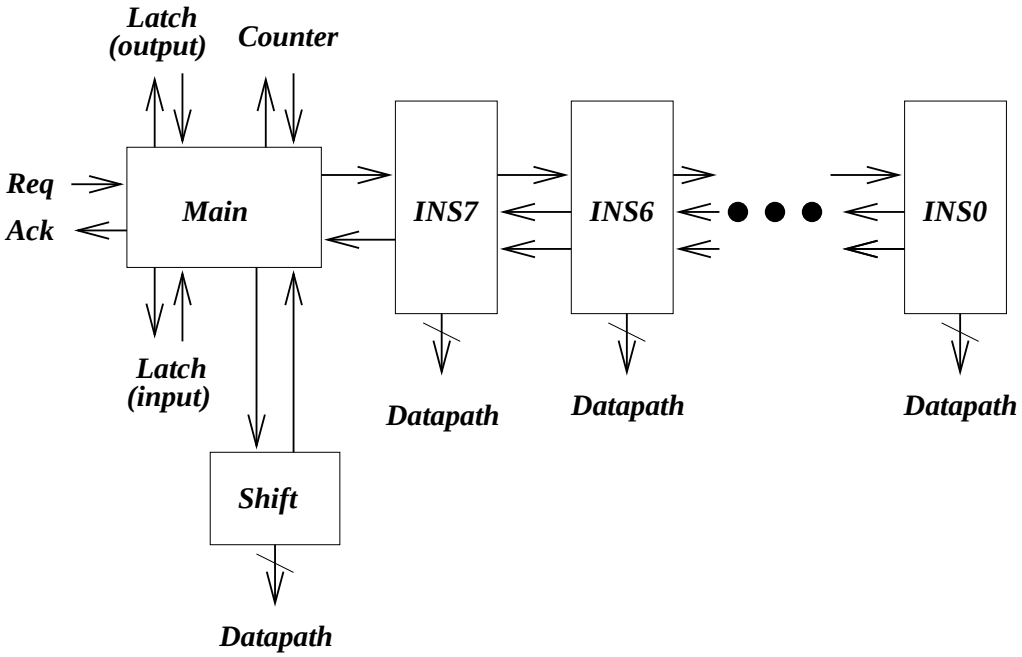

Figure 4.14. Block diagram of the controller.

is ready and sends the acknowledge when the current maximum has been determined, controls the input latches, output latches, and the counter. It also coordinates the *shift* and *insert* control blocks. The *shift* block is called much like a subroutine in software. When called, it handles the control signals related to the counter and maximum position comparison, and it executes the FIFO shift when the comparison determines that they are equal. The *ins7* block is called to check if the new datum can be inserted in the last location. If it can, the *ins7* block asks the *ins6* block to check, etc. until one block cannot accept the data. At that point, a signal is sent back to tell the previous block the data should be inserted in the list position that it controls. That block inserts the data in the list position that it controls, and it forwards an acknowledgement through the *ins* blocks to its left to the *main* block.

4.2.4 Results

We implemented our architecture in VHDL and simulated it for 100,000 correlated random data elements. The data were generated by filtering pseudo-random Gaussian white noise by a single-pole filter, and the output is then scaled and quantized to an 8-bit value. Due to the asynchronous nature of our architecture, it is able to take advantage of data-dependent delay variations. The sources for data-dependent delay variations are in

the counter, each comparator, and the number of elements in the FIFO. These variations result in an extremely variable data delay cycle as shown in Figure 4.15 which depicts a histogram of the delay to accept a new datum and output the current maximum. Over the course of the 100,000 elements, our minimum delay was as small as 29 gate delays and our maximum was as large as 161 gate delays. The average delay is 58.6 gate delays with a standard deviation of 17.3. As mentioned earlier, since the list size is much smaller than the window size, elements may need to be discarded. This event happened 8925 times, but *never* did the dropped element become a maximum in the sliding window.

One advantage of asynchronous design is the ability of an asynchronous design to adapt to operating conditions. The delay of a transistor in a VLSI design can vary significantly depending on the quality of the process run, the operating temperature, and the supply voltage. In a synchronous design, this variation is taken into account by adding a substantial margin to the clock cycle to guarantee that the chip operates correctly even in the most adverse circumstances. In reality, a chip typically comes from an average processing run and runs much cooler and at a higher supply voltage than in

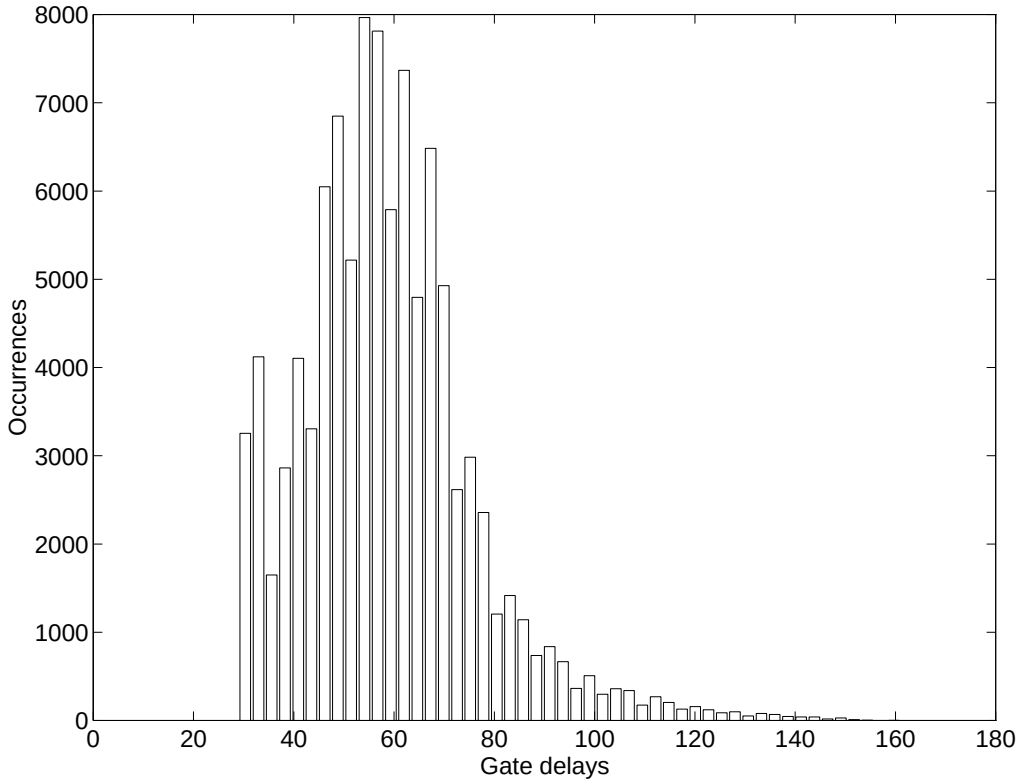

Figure 4.15. Data cycle delay distribution (fixed).

the worst-case. The speed of an asynchronous design adapts to the current operating conditions. We took this fact into account in the simulation by replacing all fixed delay parameters by delay parameters which are randomly generated each cycle within a delay bound from the worst-case down to 50 percent of the worst-case. Our simulation results using these bounded delays are shown in Figure 4.16. The average delay improves to 43.9 gate delays with a standard deviation of 13. The minimum and maximum delays also improve to 18.9 and 122.2 gate delays, respectively.

4.2.5 Comparison

We compared our results with several synchronous implementations of the MAXLIST algorithm that were designed as class projects at the University of Utah. The best implementation designed by Julsgaard and Xu [13] had a clock frequency of 75 MHz for a $1.2\mu m$ CMOS process, and it required $6+2X$ cycles to accept a new datum and output the current maximum where X is the number of comparisons required. On average, they need 1.4 comparisons, or $117ns$. Assuming a $0.5ns$ gate delay for this process, this

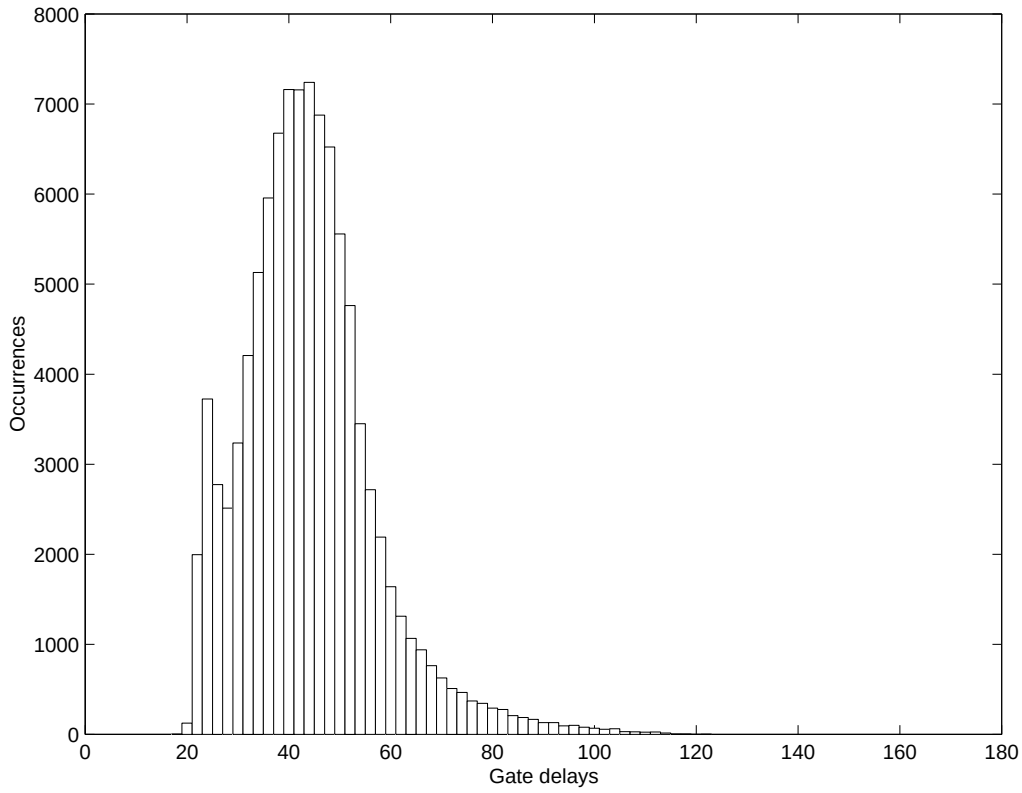

Figure 4.16. Data cycle delay distribution (bounded).

synchronous design requires on average 234 gate delays per data cycle.

In order to draw a fairer comparison, we examine the performance of a hypothetical synchronous design which uses the same architecture as our asynchronous design. For each data element in a synchronous design, one cycle would be required to latch the data and increment the counter. Another cycle is needed to perform the position comparison to see if a shift is necessary. If a shift is necessary, a clock cycle would be needed to perform it. Next, a minimum of two cycles are needed for each comparison that is going to be performed to find the location in which to insert the data into the FIFO. One is needed to determine and obtain the next element to be compared against, and the second is to perform the comparison. After the position is determined, one cycle is needed to insert the element. Finally, one cycle is required to output the current maximum. Putting it all together, we get the following:

$$\text{data cycle delay} \quad = \quad 4 + p(\text{shift}) + 2 \cdot \text{avg}(\text{cmp})$$

In the 100,000 data samples, the list needs to be shifted only 227 times, so $p(\text{shift})$ is negligible. Using 1.4 as the average number of compares, the approximate average data cycle delay in a synchronous design would be about 6.8 cycles. The counter and comparator would require at least one gate delay per bit and at least two more for control and latching data in and out. Thus, the fastest possible clock cycle time would be at least 10 gate delays. Using a 10 gate delay cycle time, the synchronous design would require on average 68 gate delays per data cycle. Therefore, our asynchronous design is at least 14 percent faster considering only data-dependent delay variations and fixed delays, and at least 35 percent faster when operating conditions are also considered using bounded delays.

If we are given a fixed throughput requirement, this speed improvement can be turned into improved power performance by lowering the supply voltage. For example, to get the same performance as the best synchronous design at 5 volts, our asynchronous design can be run at 3.2 volts. This leads to a 59 percent savings in power, since power scales as the square of the voltage.

4.2.6 Compilation

Since our compiler synthesizes a subset of VHDL, only the controller block, which is shown in Figure 4.14, is synthesized. First, the *main* block is considered. Its VHDL code

is shown Figure 4.17. The parameters in a delay function models half a gate delay and a gate delay, respectively. For simplicity, only the architecture body is shown.

The *main* block first waits until a request signal *req* is high, which means the datum is ready. Then, *main* block set *reqin* and *reqcnter* to high to read the datum into the datapath block and to increase the counter by 1. After the above actions complete, it requests the *shift* block to check if it is necessary to shift the *FIFO* block. After the shift completes, the *insert* block is called to check if the new datum can be inserted into the *FIFO* block. Its TEL structure is shown in Figure 4.18. As noted, the graph displays how the signal assignments are handled differently in VHDL and HSE. In HSE, when a signal assignment statement is executed, the statement following it can be executed only after the event defined in the previous signal assignment statement is scheduled and fired. However, in VHDL, when a signal assignment statement is executed, the event is put into a event queue, and will be fired after some delay. It does not affect the execution of the following statements.

The *shift* block is used to control whether to shift the *FIFO* block. When this block is called, it first requests the equality comparator to compare the new datum with the first element in the *FIFO* block. If they are equal, it controls the *FIFO* block to do the

```
process
begin
    wait until req = '1';
    reqin ⇐ '1' after delay(5, 10);
    reqcnter ⇐ '1' after delay(5, 10);
    wait until ackin = '1' and ackcnter = '1';
    shiftreq ⇐ '1' after delay(5, 10);
    reqin ⇐ '0' after delay(5, 10);
    reqcnter ⇐ '0' after delay(5, 10);
    wait until shiftack = '1';
    reqins ⇐ '1' after delay(5, 10);
    shiftreq ⇐ '0' after delay(5, 10);
    wait until ackins = '0';
    reqout ⇐ '1' after delay(5, 10);
    reqins ⇐ '0' after delay(5, 10);
    wait until ackout = '1';
    ack ⇐ '1' after delay(5, 10);
    reqout ⇐ '0' after delay(5, 10);
    wait until req = '0';
    ack ⇐ '0' after delay(5, 10);
end process;
```

Figure 4.17. The VHDL code of *main* block.

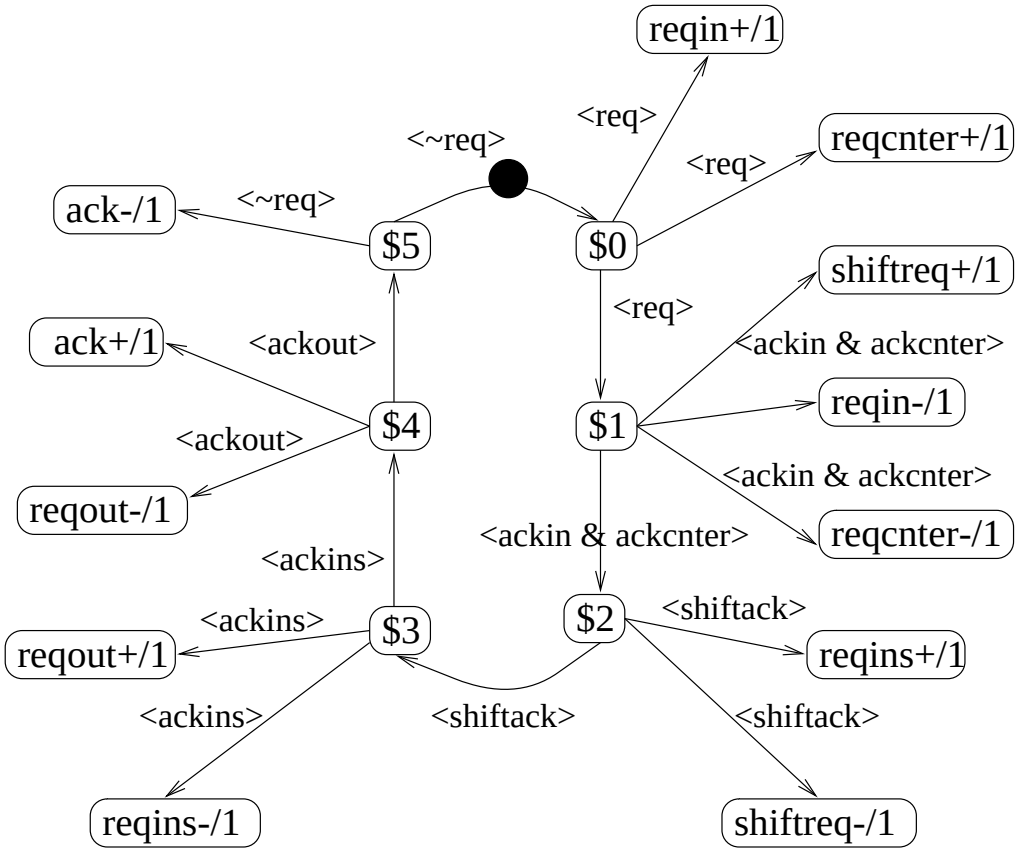

Figure 4.18. The TEL structure for the *main* block.

shift. When the shift is done, it acknowledges the *main* block. Its VHDL code is shown in Figure 4.19. Its TEL structure is shown in Figure 4.20.

The description of *insert* block is given in the previous subsection. Because of its complexity, both its VHDL code and TEL structure are too big to fit in a single page, so they are not shown.

```
process
begin
      wait until req = '1';
      cmp ⇐ '1' after delay(5, 10);
      wait on eq, neq;
      if (eq = '1' and neq = '0') then
            shift ⇐ '1' after delay(5, 10);
            cmp ⇐ '0' after delay(5, 10);
            wait until over = '1';
            ack ⇐ '1' after delay(5, 10);
            shift ⇐ '0' after delay(5, 10);
            wait until req = '0';
            ack ⇐ '0' after delay(5, 10);
      elsif (eq = '0' and neq = '1') then
            cmp ⇐ '0' after delay(5, 10);
            ack ⇐ '1' after delay(5, 10);
            wait until req = '0';
            ack ⇐ '0' after delay(5, 10);
      end if;
end process;
```

Figure 4.19. The VHDL code of *shift* block

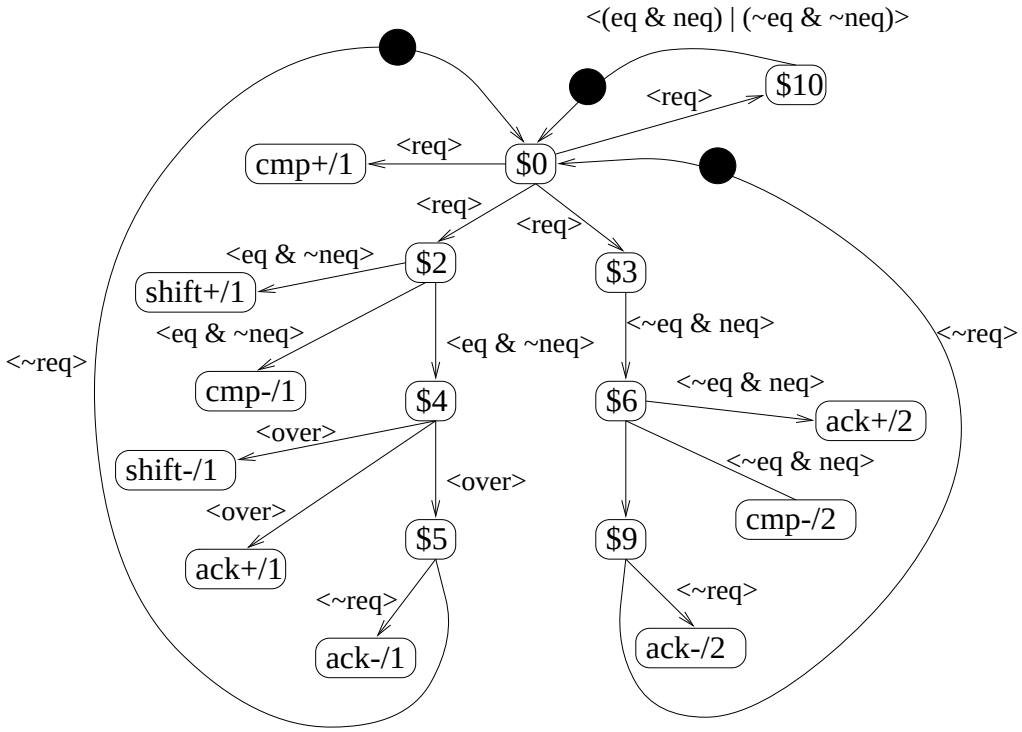

Figure 4.20. The TEL structure for the *shift* block.

CHAPTER 5

CONCLUSION

Asynchronous designs have attracted a lot of attention recently because of their advantages, but their wide application is limited due to their disadvantages. There are many existing specification and synthesis methodologies, some are graph-based, the others are language-based. Each of them is limited to a particular design style and synthesis methodology, and none of these methods allows timed systems to be easily specified.

To take advantage of the benefits of asynchronous designs, we have presented a framework for the specification of timed circuits which is independent of design style and synthesis method, and allows timing to be specified easily. We have refined VHDL to a synthesizable subset which includes constructs to specify circuit hierarchy and behavior. We described the syntax rules for timed HSE and our synthesizable subset of VHDL in Chapter 2. To use VHDL, we have developed a package to allow simulation of nondeterministic environment and delay behavior. This allows us to have a uniform method of specification for both simulation and synthesis. We use a new semantic model, timed event/level (TEL) structures to define the behavior specified by timed HSE and our synthesizable subset of VHDL, and show how TEL structures can be applied to formally define the semantics of timed HSE and our synthesizable subset of VHDL. We developed a compiler to translate the timed HSE and VHDL specifications into TEL structures, which are then fed to the rest of `ATACS` to synthesize timed circuit implementations. We also implemented a DSP algorithm, MAXLIST, using our methods. The simulation shows that our implementation outperforms comparable synchronous counterparts. We also showed the VHDL specification and compilation results of its controller.

In the future, we plan to extend the synthesizable subset of VHDL and improve the compiler to accept more language constructs so that larger and more complex design examples can be specified and synthesized. We also plan to find an approach to implement CSP communication actions in VHDL, thus, to allow designers to specify models at a higher and more abstract level, and also make it easy to translate from CSP specifications

to VHDL specifications, or vice versa. We then plan to develop an automatic tool to do the translation between CSP specifications and VHDL specifications. By doing so, CSP specifications can be simulated with a simulator which simulates VHDL specifications.

REFERENCES

[1] P. A. Beerel, C. J. Myers, and T. H.-Y. Meng, *Automatic synthesis of gate-level speed-independent circuits*, Tech. Rep. CSL-TR-94-648, Stanford University, Novermber 1994.

[2] W. Belluomini and C. J. Myers, *Timed event/level structures.* In collection of papers from TAU'97.

[3] W. Belluomini and C. J. Myers, *Efficient timing analysis algorithms for timed state space exploration*, in Proc. International Symposium on Advanced Research in Asynchronous Circuits and Systems, IEEE Computer Society Press, Apr. 1997.

[4] K. v. Berkel, J. Kessels, M. Roncken, R. Saeijs, and F. Schalij, *The VLSI-programming language Tangram and its translation into handshake circuits*, in Proc. European Conference on Design Automation (EDAC), 1991, pp. 384–389.

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

[6] S. M. Burns, *Performance Analysis and Optimization of Asynchronous Circuits*, PhD thesis, California Institute of Technology, 1991.

[7] T.-A. Chu, *Synthesis of Self-Timed VLSI Circuits from Graph-theoretic Specifications*, PhD thesis, Massachusetts Institute of Technology, 1987.

[8] B. Coates, A. Davis, and K. Stevens, *The Post Office experience: Designing a large asynchronous chip*, Integration, the VLSI journal, 15 (1993), pp. 341–366.

[9] A. Davis, B. Coates, and K. Stevens, *The Post Office experience: Designing a large asynchronous chip*, in Proceedings of the Twenty-Sixth Annual Hawaii International Conference on System Sciences, IEEE Computer Science Press, 1993, pp. 409–418.

[10] S. C. Douglas, *A family of normalized LMS algorithms*, IEEE Signal Processing Letters, 1 (1994), pp. 49–51.

[11] ——, *Private communications*, 1996.

[12] ——, *Running max/min calculation using a pruned ordered list*, IEEE Transactions on Signal Processing, 44 (1996), pp. 2872–2877.

[13] K. Julsgaard and Z. Xu, *A VLSI implementation of the MAXLIST algorithm.* Project report for CS/EE 542, University of Utah, 1995.

[14] A. J. Martin, *Programming in VLSI: from communicating processes to delay-*

insensitive VLSI circuits, in UT Year of Programming Institute on Concurrent Programming, C. Hoare, ed., Addison-Wesley, 1990.

[15] K. McMillan and D. L. Dill, *Algorithms for interface timing verification*, in International Conference on Computer Design, ICCD-1992, IEEE Computer Society Press, 1992.

[16] T. H.-Y. Meng, R. W. Brodersen, and D. G. Messershmitt, *Automatic synthesis of asynchronous circuits from high-level specifications*, IEEE Transactions on Computer-Aided Design, 8 (1989), pp. 1185–1205.

[17] C. E. Molnar, T.-P. Fang, and F. U. Rosenberger, *Synthesis of delay-insensitive modules*, in 1985 Chapel Hill Conference on Very Large Scale Integration, H. Fuchs, ed., Computer Science Press, Inc., 1985, pp. 67–86.

[18] C. J. Myers, *Computer-Aided Synthesis and Verification of Gate-Level Timed Circuits*, PhD thesis, Stanford University, 1995.

[19] C. J. Myers and T. H.-Y. Meng, *Synthesis of timed asynchronous circuits*, IEEE Transactions on VLSI Systems, 1 (1993), pp. 106–119.

[20] C. J. Myers, T. G. Rokicki, and T. H.-Y. Meng, *Automatic synthesis of gate-level timed circuits with choice*, in 16th Conference on Advanced Research in VLSI, IEEE Computer Society Press, 1995, pp. 42–58.

[21] C. J. Myers and H. Zheng, *An asynchronous implementation of the maxlist algorithm*, in International Conferences on Acoustics, Speech, and Signal Processing, vol. 1, April 1997, pp. 647–650.

[22] S. M. Nowick, *Automatic Synthesis of Burst-Mode Asynchronous Controllers*, PhD thesis, Stanford University, Department of Computer Science, 1993.

[23] T. G. Rokicki and C. J. Myers, *Automatic verificaton of timed circuits*, in International Conference on Computer-Aided Verification, Springer-Verlag, 1994, pp. 468–480.

[24] P. Subrahmanyam, *What's in a timing discipline? considerations in the specification and synthesis of systems with interacting asynchronous and synchronous components*, in Hardware Specification, Verification and Synthesis: Mathematical Aspects, Springer-Verlag, 1990.

[25] P. Vanbekbergen, G. Goossens, and H. de Man, *Specification and analysis of timing constraints in signal transition graphs*, in Proceedings of the European Design Automation Conference, 1992.

[26] V. I. Varshavsky, ed., *Self-Timed Control of Concurrent Processes: The Design of Aperiodic Logical Circuits in Computers and Discrete Systems*, Kluwer Academic Publishers, Dordrecht, The Netherlands, 1990.

[27] G. Winskel, *An introduction to event structures*, in Linear Time, Branching Time and Partial Order in Logics and Models for Concurrency. Noordwijkerhout, Norway,

June 1988.

[28] K. Y. Yun, *Synthesis of Asynchronous Controllers for Heterogeneous Systems*, PhD thesis, Stanford University, 1994.