Incorporating VHDL Signal/Wait Semantics into Synthesis

Sanjiv Narayan
Frank Vahid
Daniel D. Gajski

Technical Report #92-32
April 2, 1992

Dept. of Information and Computer Science
University of California, Irvine
Irvine, CA 92717
(714) 856-8059

vahid@ics.uci.edu
narayan@ics.uci.edu

Abstract

VHDL signals and wait statements provide great expressive power for behavioral descriptions. However, due to their simulation semantics, most high-level synthesis tools do not handle these constructs and severely restrict their use, eliminating much of their power. In this report, we introduce a set of transformations to convert signals and wait statements to equivalent constructs that are easily handled by high-level synthesis tools. They greatly enlarge the synthesizable VHDL subset, thus increasing the usefulness and practicality of the language as an input to high-level synthesis. These transformations can also serve as a basis for converting a VHDL process to a form suitable for generation of software.

Contents

1 Introduction 1

2 VHDL Simulation Semantics 4

2.1 Signal Assignment Statements 4

2.2 Wait Statements 7

3 Problem Formulation 8

4 W/S Transformations 9

4.1 Signal Assignment Statements 10

4.2 Wait Statements 11

4.3 Common Simplifications 13

5 Synthesizing efficient hardware 16

5.1 Avoiding excess registers 16

5.2 Branch-path elimination 17

5.3 External timer elimination 18

5.4 Mapping processes to combinational logic 19

6 Conclusions 20

7 Acknowledgements 21

8 References 21

List of Figures

1 An example exploiting signal semantics 2

2 W/S transformations in an overall synthesis methodology 3

3 Latching signal values driven by each process 4

4 Hardware templates for three kinds of signals 6

5 Wait statement flowgraph 8

6 Obtaining hardware from VHDL specifications with signal assignments and general wait statements 9

7 Basic signal hardware template 10

8 The Wait/Signal Transformations 11

9 Variables created by the Wait/Signal Transformations for a signal S written to in a process . . . 13

10 Template for wait statement where signals in sensitivity clause and condition clause are identical (wait_template_2) 14

11 Wait/Signal Transformation Algorithm 15

12 Additional variable does not change dataflow graph 17

13 A wait statement simplification 18

14 Eliminating false branches 18

15 Eliminating the external timer for timeouts in wait statements. 19

1 Introduction

The adoption of VHDL [1, 2] as an IEEE standard has resulted in growing acceptability of the language as an input to high-level synthesis [3, 4]. First, as designer VHDL literacy continues to grow, behavioral descriptions become easier to write and become more useful for documentation purposes. Second, high quality simulation and debugging tools are widely available. Finally, a large number of synthesis tools accept VHDL as an input.

However, process-level VHDL is complex with regard to the semantics of signal and wait statement constructs. The reason is that such constructs depart from traditional sequential programming language constructs. As a result, it is more difficult to synthesize hardware from VHDL than from other languages such as HardwareC [5], Verilog [6], or ISPS [7]. Therefore, synthesis tools place heavy restrictions on the allowable use of signal and wait constructs; such restrictions are often refered to as language subsetting.

Such restrictions are unfortunate, since the signal and wait constructs have a high degree of expressiveness. The VHDL wait statement can replace many lines of sequential code, as will be demonstrated later in this report. VHDL signals differ from variables in that they not only have a value, but they have that value at a particular *time*. Thus signals, unlike variables, can be shared by several concurrent processes. Also, VHDL ports, through which all communication by an entity to the external environment takes place, are actually signals themselves. In addition, the experienced VHDL modeler soon discovers that the time aspect of signals provide a very elegant means for specifying parallelism in a sequential behavior.

For example, Figure 1 shows a pipelined processor, where each concurrent stage is modeled as a procedure with sequential statements. The procedures are called from a main process. The description contains much concurrency even though the statements comprising the stages occur sequentially. For example, IR1 in *stage*1 and IR2 in *stage*2 are updated simultaneously, and the IR1 used in *stage*2 is the old value of IR1 written by *stage*1. In the figure, the assignment to PC in the BRANCH instruction of *stage*2 supercedes the increment during instruction fetch in *stage*1, i.e. it does not use the incremented value of PC. A description where each stage is modeled as a process is not as simple. It will involve signal resolution along with a prioritizing of the concurrent assignments to the PC

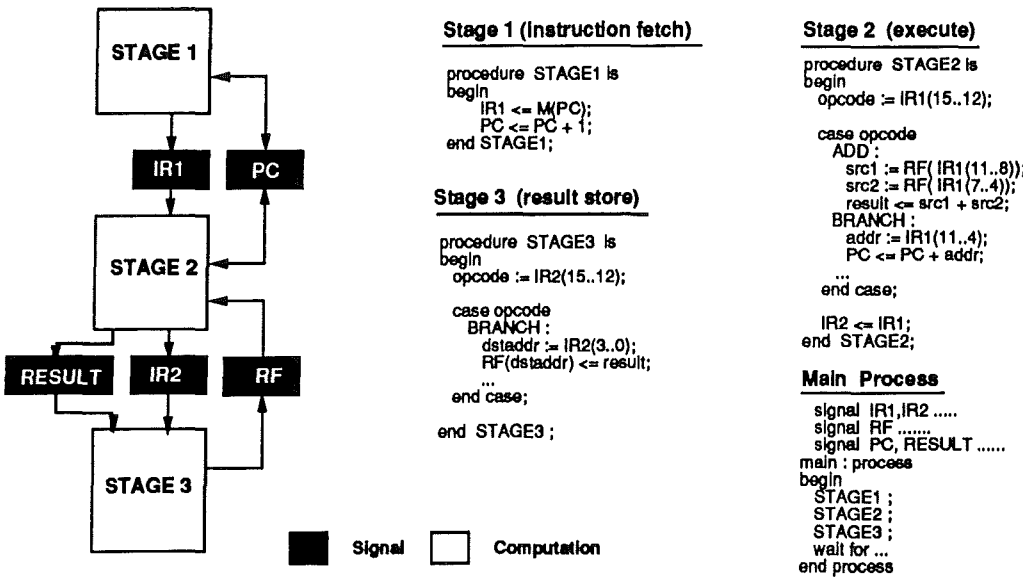


Figure 1: An example exploiting signal semantics

signal.

Current synthesis tools impose restrictions on the VHDL constructs that can be used in an input specification, severly limiting the expressive power of signals and waits. For example, some tools require that a signal may not be both read and written by the same process while other tools ignore the *on* clauses of wait statements. To overcome these restrictions, we developed transformations to convert signals and waits into constructs easily handled by existing high-level synthesis (HLS) methodologies. In current HLS methodology, each process is converted to an graph-type representation containing control and dataflow operations; we shall refer to all such representations as CDFG's. CDFG optimizations and transformations are performed with the goal of enabling more efficient hardware to be synthesized. HLS tasks such as scheduling, allocation, and binding are applied to the CDFG, and a structural module is output that implements the process. The modules for the multiple processes that comprise the entire behavioral description are then connected.

Techniques are well-known for obtaining a CDFG for traditional sequential program constructs found in each process, such as variable assignments, branches, and procedure calls. There are many HLS approaches which incorporate these techniques [4, 5, 8, 9, 10]. However, to the best of our knowledge, no published techniques exist for synthesizing VHDL signal and wait statement semantics. Handling these constructs is not a trivial task, either.

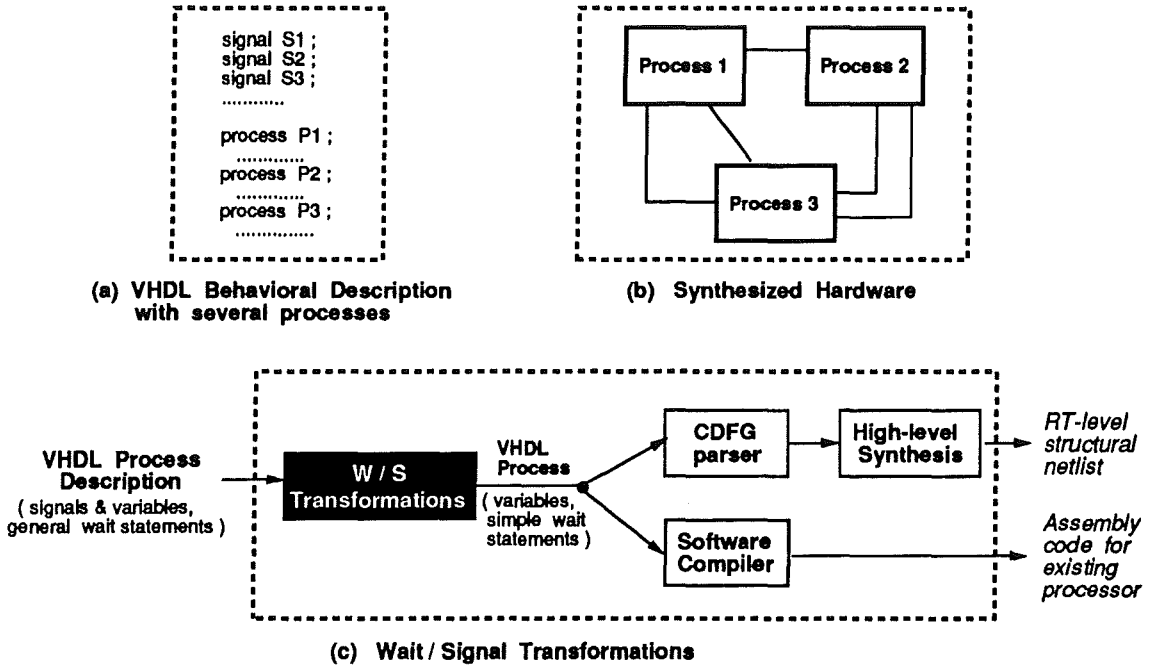


Figure 2: W/S transformations in an overall synthesis methodology

In this paper, we propose the Wait/Signal (W/S) transformations to convert VHDL processes into equivalent processes containing only variable assignments and trivial wait statements which are sensitive only to a clock. Such variable assignments and trivial wait statements are easily handled by existing HLS tools. Thus, W/S transformations provide a major step forward in enlarging the synthesizable VHDL subset.

In addition, the W/S transformations serve as a basis for providing a path from a VHDL process behavior to a software implementation on an existing processor. The VHDL process with signal and wait statements is transformed into a description with only sequential constructs, which can be easily mapped to the instruction set of a processor. It is therefore possible to perform hardware/software tradeoffs from the same input VHDL specification.

In this report, we present the transformations as a front-end to existing synthesis tools, as shown in Figure 2, merely for the purpose of ease of presentation; an implementation will more likely incorporate them with an existing tool such as a CDFG creation tool.

This report is organized as follows. In Section 2 we summarize the relevant aspects of signal and wait semantics. We would like to mention that we discuss signal semantics in the context of processes

only. VHDL concurrent signal assignments can be modeled using an equivalent process statement, so are not covered separately in this paper. In Section 3, we formulate the specific problem we wish to solve. In Section 4, we introduce the W/S transformations for converting signals and wait statements to a form handled by existing synthesis tools. These transformations have been developed with a view to synthesize synchronous hardware. On the surface, it would appear that the transformed VHDL will be implemented using complex hardware. Section 5 describes why this is not so; that in fact standard CDFG optimizations already found in most synthesis tools will yield efficient and practical hardware.

2 VHDL Simulation Semantics

2.1 Signal Assignment Statements

The syntax for a VHDL signal declaration is:

signal identifier : [resolution-function-name] type [signal-kind] [:= expression] ;

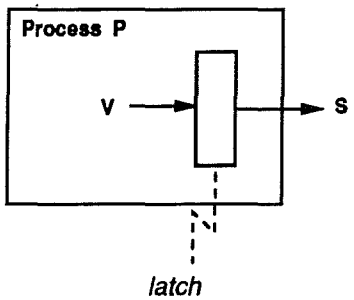
signal-kind ::= **register** | **bus**

Each process which writes to a signal is called a *driver* for that signal. If a signal is written to by more than one process, a resolution function is required to combine the multiple driver values into one *resolved value* for the signal. VHDL allows resolution functions to be arbitrarily complex. However, to be able to synthesize feasible hardware, we limit resolution functions to represent the

```
signal S : integer ;

P: process
     variable V : integer ;
begin
     V := 1 ;
     S <= V ;
     wait for 50 ns ;
     V := 2 ;
end process P ;
```

(a) Code showing change in the source of signal S



(b) Latching signal S

Figure 3: Latching signal values driven by each process

technology-specific wire characteristics. For example, in some technologies, multiple drivers on the same wire result in a wired-or value, which can be modeled by a resolution function using a look-up table. This enables the synthesis tool to synthesize each process independently as a module, and then connect the wires representing the signal driven by each module.

The value which a process writes to a signal may need to be stored. To see this, consider the VHDL code segment in Figure 3. According to VHDL signal semantics, process P should continue to drive S with the value "1" even after 50 ns, at which time V is set to "2". Since the source of the value written to signal S may change, a storage for the signal's value is implied. However, if only constants are assigned to the signal in the process, or if the signal is updated in the same control step whenever any source in the previous assignment is updated, a storage for the signal is not required.

Signals can be of three kinds: simple (or no-kind), bus, or register. Their semantics are examined separately and are explained with the help of the hardware *templates* of Figure 4. Signal semantics and synthesis are discussed in detail in [11].

Simple signals can have multiple drivers as shown in Figure 4(a). However, the drivers of a simple signal cannot be turned off (i.e., a null assignment "S <= *null*;" is not permitted in the process). Consequently, a simple signal has all of its drivers active at all times. Signals of bus kind are different from simple signals in that the drivers can be turned off by a null assignment. This results in a tristate driver being placed after the latch in Figure 4(b). In addition to resolving the values written by the different drivers, the resolution function must specify a value for the case when all the drivers are turned off. Register kind signals are identical to bus signals except that in the event that all the drivers are turned off, the signal retains its last resolved value. This is achieved by using an additional latch which stores the last resolved value of the signal, as in Figure 4(c).

Having introduced signal semantics external to processes, we now discuss the semantics of signals associated with updating a process' driver using sequential process statements.

The VHDL signal assignment statement syntax is:

signal_target <= expression [**after** time_expression] ;

| **null** [**after** time_expression] ;

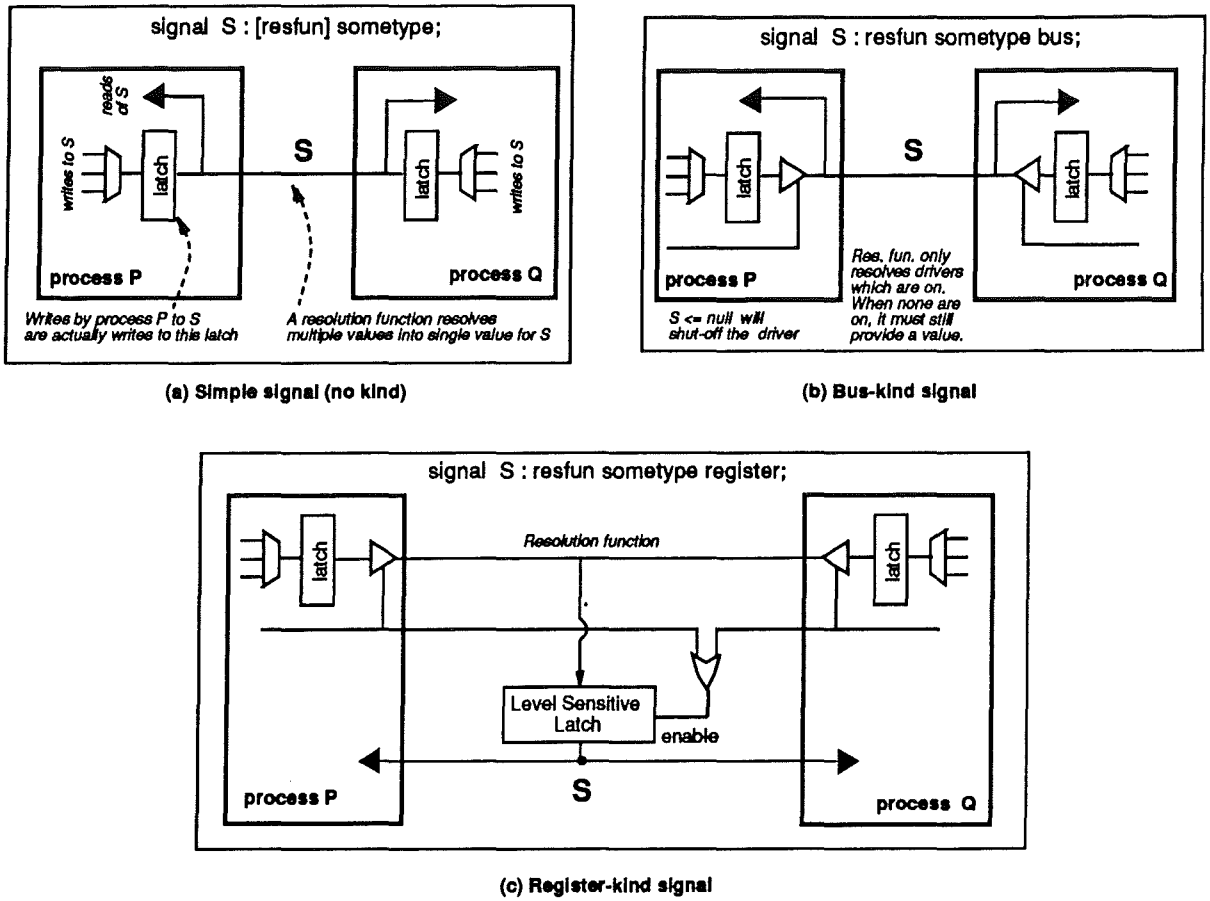


Figure 4: Hardware templates for three kinds of signals

Evaluating the expression determines the next value of the signal driven by the process. We currently do not permit *after* clauses in signal assignments, which specify the time when the signal will be updated. Therefore, the value driven by the process is updated with the next value when the next *wait* statement is encountered. This is distinct from variable assignments (whose syntax is "variable_target := expression"), where an assignment causes an immediate change in the value of the target variable. We illustrate this difference with the help of the following VHDL code segment involving signal assignments:

```
A <= B;
B <= A;
wait for 10 ns;
```

The values of A and B are not updated until the wait statement is encountered. Hence the above statements have the same effect as a swap of the values of A and B. However, if the statements are

incorrectly interpreted as variable assignments, both A and B will get the same value (i.e. value B). For a more detailed description of signal semantics, refer to [1].

2.2 Wait Statements

The syntax of the VHDL wait statement is:

wait_statement	::=	**wait** [sensitivity_clause] [condition_clause] [timeout_clause] ;
sensitivity_clause	::=	**on** *signal*_name {,*signal*_name}
condition_clause	::=	**until** condition
timeout_clause	::=	**for** *time*_expression

The sensitivity list specifies the signals to which the wait statement is sensitive. When an event occurs on a signal in the sensitivity list, the condition clause specifies a condition that must be met for the process to resume execution. The timeout clause specifies the maximum time that the process will be suspended at the current wait statement.

The semantics of the wait statement is explained with the help of the flowgraph of Figure 5(a). The function *current_time* provides the current simulation time, while *advance_time* advances simulation time to the point when the next event occurs. These two functions are used to determine when the timeout interval has expired.

A process suspended at a wait statement can resume in two ways: either an event occurs on a signal in the sensitivity list *and* the condition in the condition clause evaluates to true, or the timeout interval in the timeout clause expires.

VHDL defines default values for all the clauses in the wait statement, shown in Figure 5(b). Thus, if some of the clauses are omitted in the wait statement, the resulting flowgraph can be simplified greatly. For example, in the absence of a condition clause, the default value is true and we can thus eliminate the branch-node D from the flowgraph of Figure 5. Similarly, in the absence of a timeout clause, the branch node labeled B can be eliminated along with the statement block A.

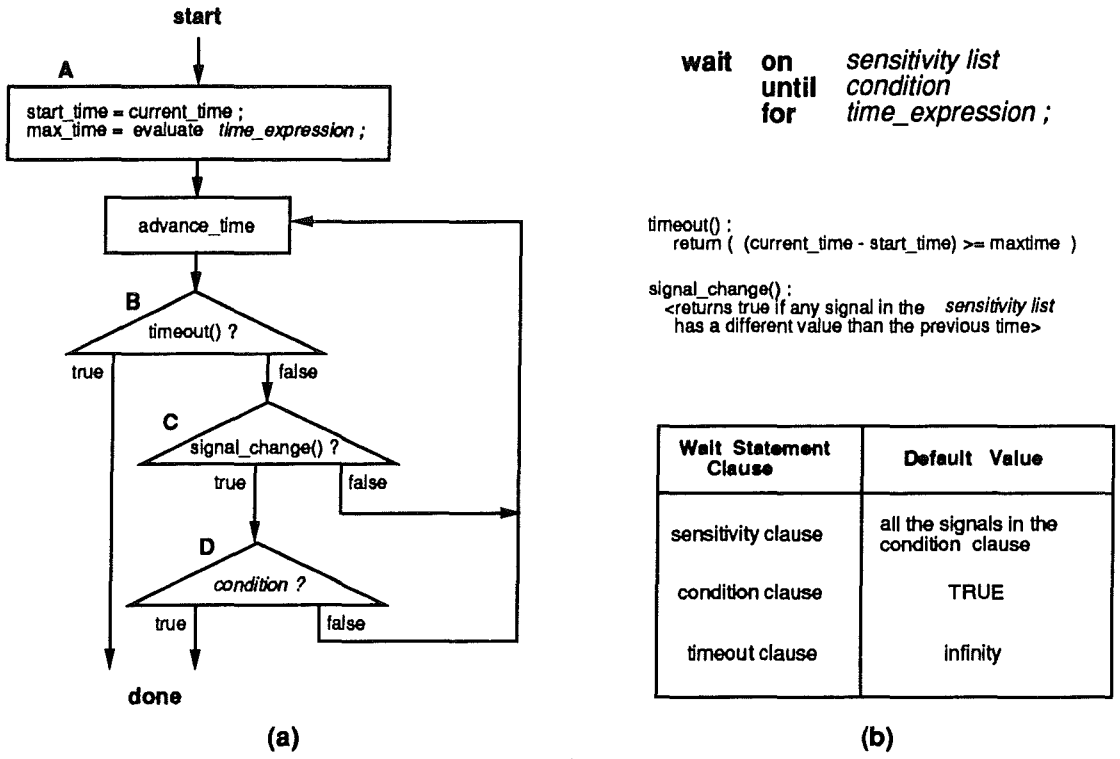


Figure 5: Wait statement flowgraph

3 Problem Formulation

Having introduced the simulation semantics of signals assignments and wait statements, we now examine synthesis approaches to obtain hardware from VHDL descriptions containing these constructs. Two approaches are shown in Figure 6. We may synthesize hardware directly from the VHDL description (box A) specified by the designer. However, due to the complex signal and wait statement semantics, such a direct synthesis from input VHDL descriptions is difficult to implement.

However, if we could somehow transform these constructs into an equivalent description (box B) containing only variables and simple wait statements, we could invoke traditional high-level synthesis tools to obtain hardware from the transformed specification. These W/S (wait/signal) transformations, which would enable us to eliminate all signals and complex wait statements, provide the main motivation for the research presented in this paper.

It must be mentioned at this point that the semantic preserving W/S transformations may add extra variables for each signal. However, the resulting implementation (box D) will not have any

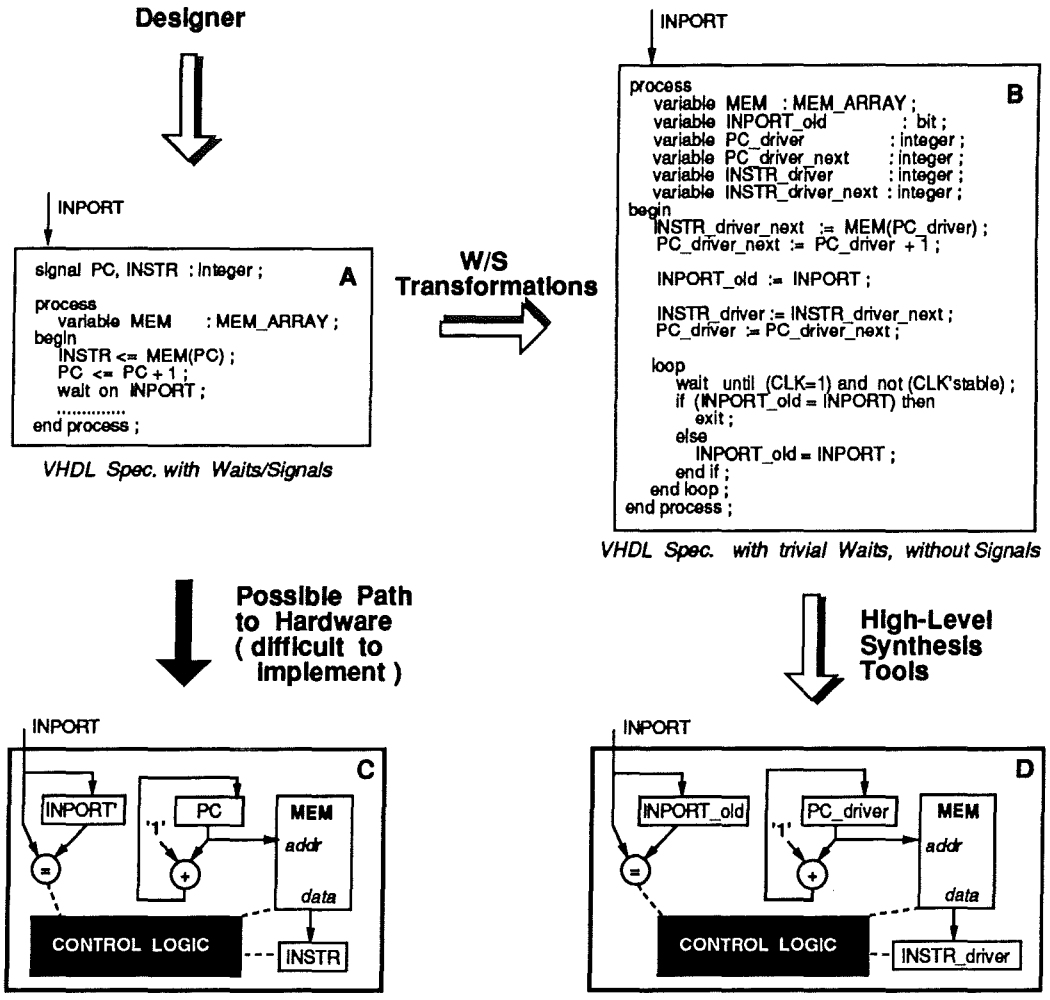


Figure 6: Obtaining hardware from VHDL specifications with signal assignments and general wait statements

excess hardware than would otherwise be required if we had attempted to synthesize hardware (box C) directly from the VHDL description with signal and wait statements. The transformations are presented in the next section. Techniques for generating efficient hardware from the transformed VHDL description are presented in Section 5.

4 W/S Transformations

We assume that the subsequent high-level synthesis tool uses the signal driver template shown in Figure 7. *S_driver* represents the value of the signal S driven by the process. Since several processes could be driving the signal, *S_resolved* represents its resolved value. This is also the value that is used

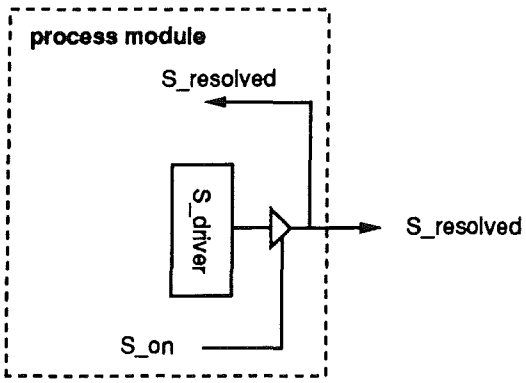


Figure 7: Basic signal hardware template

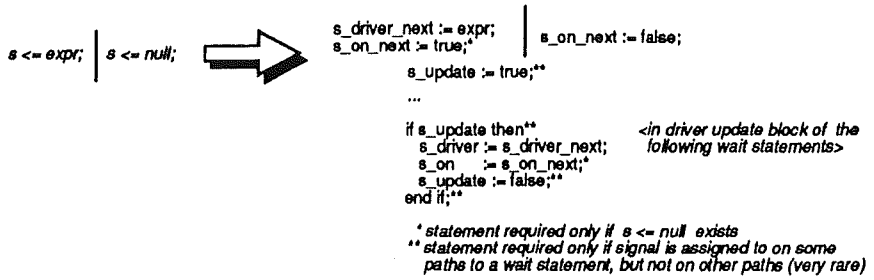
in the right-hand side of all assignments to S. As explained in the previous section, a tristate buffer may be required in case the driver is turned off in the process by a null assignment. *S_on* represents the control signal for this tristate buffer. *S_driver* and *S_resolved* are of the same type as the signal S while *S_on* is of type boolean. In addition, we assume that the subsequent synthesis tool recognizes these three variables in that they represent the various sub-components of the template shown in the figure.

Figure 8 shows the W/S transformations applied to the VHDL signal assignment and wait statements. We now discuss these transformations in detail.

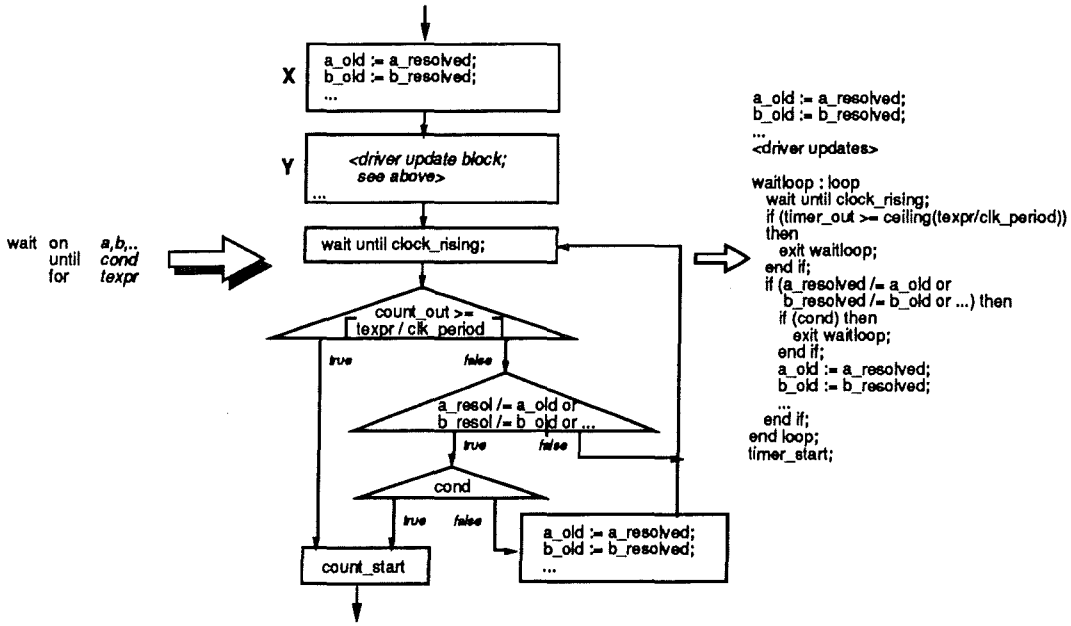
4.1 Signal Assignment Statements

Figure 8(a) shows how signal assignment statements are represented only using variables. In a signal assignment "`S <= expression ;`", the driver for S is only updated with the value of the *expression* at the next wait statement. We need to store this next value of S in a new variable called *S_driver_next*. Thus, all assignments to S are replaced by "`S_driver_next := expression ;`".

In case the driver for signal S is turned off in the process, then each non-null assignment to S must turn on the driver. This can be achieved by adding "`S_on_next := true`" after the assignment statement, where the variable *S_on_next* represents the next value of *S_on*; *S_on* will be updated when the next wait statement is reached. A null assignment to S in the process should turn off the driver. This is achieved by replacing "`S <= null ;`" by "`S_on_next := false ;`".



(a) Signal Assignment Statement



(b) Wait Statement (wait_template_1 flowgraph)

Figure 8: The Wait/Signal Transformations

4.2 Wait Statements

We will first define some of the terminology used in this section. A *statement block* is defined as the set of statements between any two successive wait statements. The term *preceding_paths* refers to all paths leading from any preceding wait statement to the current one, while *following_paths* represents all paths from the current wait statement to the next wait statement.

Before we present the transformations associated with wait statements, we briefly discuss how they are interpreted with a view to synthesize hardware. A wait statement indicates that all targets of signal assignment statements in the statement block preceding it have been updated with their new values.

Thus, a wait statement implies an explicit clock boundary for synthesis. However, the synthesis tool is free to add more clock boundaries to implement the computations in the statement block, as we shall see later in Section 5.

In VHDL, all signals assigned a value in a statement block are updated at the next wait, and these updated values are available to statements following the wait statement. Thus, all computations that are performed in the statement block must have completed before the process can resume execution due to the expiry of the timeout interval. Consequently, *a timeout clause represents a timing constraint on the synthesis of the statement block preceding the wait.* It also implies that the process, when synthesized, is suspended at the wait statement for an amount of time equal to the difference between the timeout interval and the time required to perform the computations. For example, if the statement block requires 200 ns to compute the new values for all the signals, the next wait statement "`wait for 300 ns`" will effectively wait for 100 ns after the computations have been completed. On the other hand, a sensitivity list or a condition clause will be evaluated only after all the computations in the preceding statement block have been completed.

The flowchart representing the wait statement implemented using only "`wait until clock_rising`" is shown in Figure 8(b). The equivalent VHDL code generated for this template is also shown in the figure. This code is used by the W/S transformations as a template for replacing wait statements. We would like to mention that the clock boundary can be specified in any manner acceptable to the synthesis tool. For example, a rising clock can be represented as:

```
wait until (CLK = '1') and not(CLK'stable) ;
```

To be able to monitor a change on the signals in the sensitivity list we need to store the current value (*S_resolved*) of each sensitivity list signal S in the variable *S_old*. This is shown for signals in box X in the flowchart of Figure 8(b). The value *S_old* can then be compared with *S_resolved* after each rising edge of the clock to detect a change on S.

If the signal S was assigned a value in the preceding statement block, the driver value *S_driver* needs to be updated, as shown in box Y of in the flowchart of Figure 8(b). Since there could be several preceding paths leading up to the wait statement under consideration, it might be the case that the signal S is updated on only some of those paths. Thus, an additional boolean variable is set to true

whenever the signal is assigned to in any of the paths. At the wait statement, if *S_update* is true, we can update *S_driver* with the variable *S_driver_next* (computed at the previous signal assignment).

To implement timeout clauses, we can use a counter which is incremented on every clock. In pure VHDL behavior, the statements between two wait statements take zero time to execute. In synthesized hardware, these statements may require one or more clock cycles to execute. To maintain the same timing with respect to any external interface, the counter is started when we leave a wait statement so that it can be used by the next wait statement to determine the time elapsed since the previous wait. A timeout is detected whenever the counter value, *count_out*, is greater than or equal to the timeout expression expressed in terms of clock cycles (i.e. $\lceil timeout\_expression/clock\_period \rceil$). As explained earlier, this also ensures that the time spent at the wait statement includes the time required to perform the computations in the preceding statement block. The function *count_start* initializes the counter. We assume that the functions *count_start* and *count_out* are recognized by the subsequent synthesis tool.

VARIABLE	TYPE	DESCRIPTION	WHEN CREATED
S_driver	same as S	Value of S driven by process.	Always created if process writes to S.
S_resolved	same as S	Signal value resolved from multiple process drivers. This is the value used in all expressions involving S.	If S is a resolved signal.
S_driver_next	same as S	Value with which the driver, S_driver, will be updated at the next wait statement. Appears as the target for all assignments to S.	If S is assigned a value AND occurs in an expression, in the same statement block between two wait statements.
S_old	same as S	Old value of the signal, which is used to detect a change in the signal.	If S occurs in the sensitivity list of a wait statement.
S_update	boolean	Indicates whether S is to be updated at the next wait statement. Set to TRUE everytime S is assigned a value. Set to FALSE at wait.	If S is assigned a value on some paths (but not all) between two wait statements.
S_on	boolean	Control Input of the driver's tristate buffer.	If signal is assigned a NULL value
S_on_next	boolean	Value with which the S_on will be updated at the next wait statement.	If signal is assigned a NULL value

Figure 9: Variables created by the Wait/Signal Transformations for a signal S written to in a process

4.3 Common Simplifications

Figure 9 summarizes all the seven variables that may be required to implement a signal S in the most general case. However, we will rarely need all of these variables. In this section we present some

simplifications that are part of the W/S transformations that reduce the number of variables used for any given signal.

First, for a signal S, we can eliminate the variable *S_update* if it is updated on all paths between every two successive wait statements, which is usually the case. Second, in case S is an *unresolved* signal written by the process under consideration, *S_resolved* always equals *S_driver*. Thus all occurrences of *S_resolved* in the transformed VHDL can be replaced by *S_driver*. Third, if the driver for S is never turned off using a null assignment, the boolean variables *S_on* and *S_on_next* will not be needed.

Another simplification can be invoked with wait statements whenever all the signals in the condition clause and sensitivity list are identical. This is very common in VHDL descriptions, especially since it is the the default when no sensitivity list is explicitly specified. In such cases, checking for a change on the signals and then evaluating the condition is redundant, because a change in condition value implies a signal in the sensitivity list has changed. If the condition is false before the wait statement, we only need to wait until it becomes true, which also implies that some sensitivity list signal must have changed. If the condition was true before the wait statement, we must first wait until it becomes becomes false, then wait until it becomes true. The template shown in Figure 10 can be applied to avoid using the *S_old* variables.

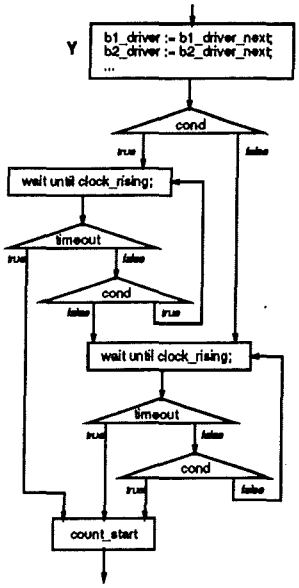


Figure 10: Template for wait statement where signals in sensitivity clause and condition clause are identical (wait_template_2)

```
for each signal S do
      Replace all reads of S by reads of S_resolved

      if there is a null assignment to S in process then
            Modify every non-null assignment to S, replacing
                  S <= expression ;  by  S_driver_next := expression ;
                                         S_on_next := true ;
            Modify every null assignment to S, replacing
                  S <= null ;    by  S_on_next := false ;
      else
            Modify every assignment to S, replacing
                  S <= expression ;  by  S_driver_next := expression ;
      endif
endfor

for each wait statement do
      if sensitivity-list signals are the same as the condition-clause signals then
            Replace with wait-template-2, leave section Y empty   (Figure 10)
      else
            Replace with wait-template-1, leave section Y empty   (Figure 8)
      endif

      for each signal S in preceding paths do
            if all preceding paths assign to S then
                  Add to section Y of template :  S_driver := S_driver_next ;
                  if there is a null assignment to S in process then
                        Add to Section Y of template :  S_on := S_on_next ;
                  endif
            else
                  After each write to S in a preceding path, add :  S_update := true ;
                  Add to section Y of template :  if  S_update  then
                                                      S_driver := S_driver_next ;
                                                      S_update := false ;
                                                end if ;
                  if there is a null assignment to S in process then
                        Add to Section Y of template :  S_on := S_on_next ;
                  endif
            endif
      endfor
endfor

for each signal S do
      if (process assigns to S) AND (S is unresolved) then
            Replace all reads of S_resolved by reads of S_driver
      endif
endfor
```

Figure 11: Wait/Signal Transformation Algorithm

Even after the above-mentioned simplifications are performed, the transformations performed by the high-level synthesis tool would further optimize these variables and very few of them will actually be implemented as storage. The next section discusses these optimizations.

The Wait/Signal transformations are summarized in Figure 11.

5 Synthesizing efficient hardware

The many variables and complex templates introduced by W/S transformations might appear to lead to a complex hardware implementation. In general, this is not the case since the dataflow representation in a CDFG eliminates many intermediate variables, and CDFG transformations eliminate many branches and statements. It is not our purpose here to discuss CDFG representations and transformations in detail. Details of these can be found in [5, 8, 12]. Instead, we shall illustrate that efficient hardware is obtainable by applying some of the common transformations.

5.1 Avoiding excess registers

A common misconception of synthesis from VHDL is that variables correspond to registers. In fact, a variable may be implemented as a register or as a *wire*. A register is required only when the value of variable is updated in one control-step and read in a subsequent control-step (or when explicitly annotated as a register in the behavioral description).

For example, consider the following functionally equivalent VHDL processes:

```
P: process(B,C)
   variable  temp : integer ;
   variable A : integer ;
begin
   temp := B + C ;
   A    := temp + D ;
end process;
```

```
Q: process(B, C)
   variable A : integer ;
begin
   A := B + C + D ;
end process ;
```

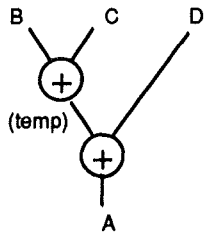


Figure 12: Additional variable does not change dataflow graph

Although one description uses an intermediate variable, both descriptions result in the dataflow graph shown in Figure 12. If two adders are available and the delay of two successive additions does not exceed the clock period, then no intermediate registers are needed; otherwise, a register is needed between the two addition operations. The *temp* variable has no role in this decision; it merely serves to enhance readability of the behavioral description.

Now recall that wait statements in the description denote explicit clock boundaries. Much of the behavior between such boundaries can be represented using a dataflow graph. Therefore, the *sig_driver_next*, *sig_on_next*, and *sig_update* variables will usually be mapped to dataflow arcs, as was *temp* in the above example, and will thus rarely require a register. Recall the swap example of Section 2. The *A_driver_next* and *B_driver_next* variables will be mapped to wires.

5.2 Branch-path elimination

If a particular path of a branch can never be reached due to the condition for that path always being false, then the condition leading to that path, along with the path's operations, can be deleted. While such code is rarely written by the modeler, it occurs quite often after the W/S transformations. For example, consider wait-template-2. If it can be determined that the until-condition is initially false, as it often can be, the flowchart reduces to that shown in Figure 13.

As another example, consider the simple code portion shown in Figure 14(a). The code after transformations is shown in flowchart form in Figure 14(b). After CDFG creation, simple dataflow analysis of the branch condition results in determination that the condition is always true, as shown in Figure 14(c). Hence we can eliminate the branch condition and the false path. After doing so, *pc_old* is written but not read, so it too can be eliminated. Although these optimizations are performed on

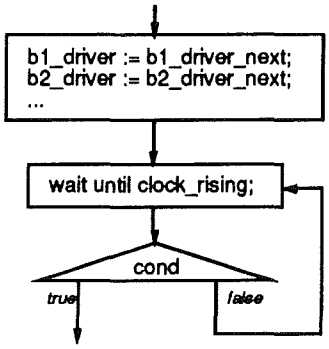


Figure 13: A wait statement simplification

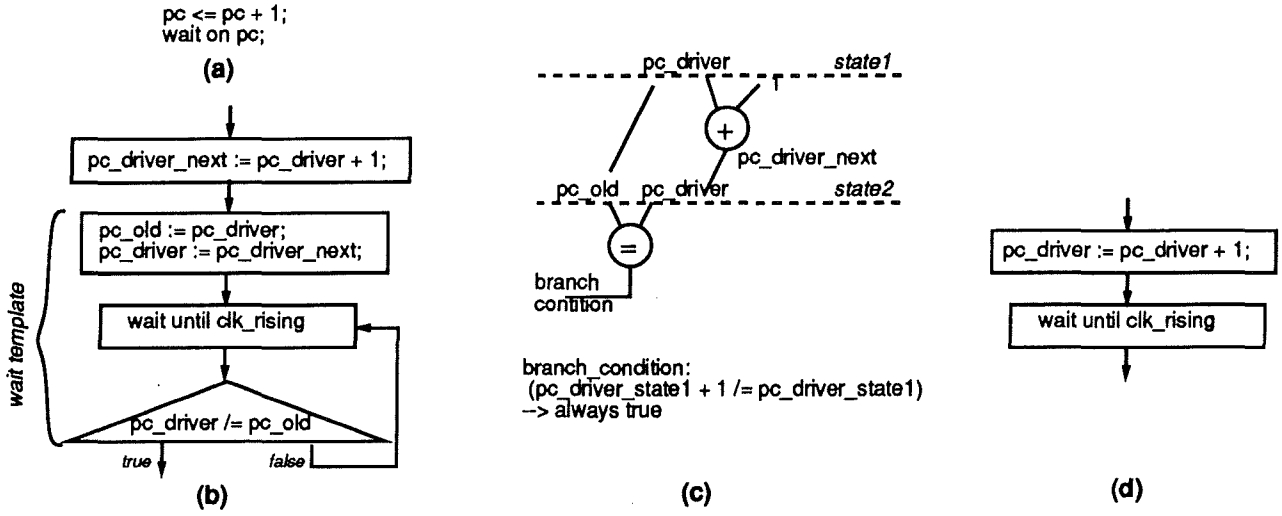


Figure 14: Eliminating false branches

the CDFG, for illustrative purposes we show the equivalent code in Figure 14(d). Note its simplicity. Also note that as discussed above, no register will be needed for *pc_driver_next*.

5.3 External timer elimination

After scheduling the CDFG, all the control-steps are known. Therefore, it is always possible for the synthesis tool to eliminate the need for the external timer-counter that was assumed before we knew all the control-steps in the CDFG. A counter variable can be created that is incremented on each clock, and incorporated into the scheduled CDFG. Often this variable itself will then be eliminated, especially after loop unrolling transformations. Such an external-timer elimination should be incorporated into any HLS tool interfacing with the W/S transformed VHDL.

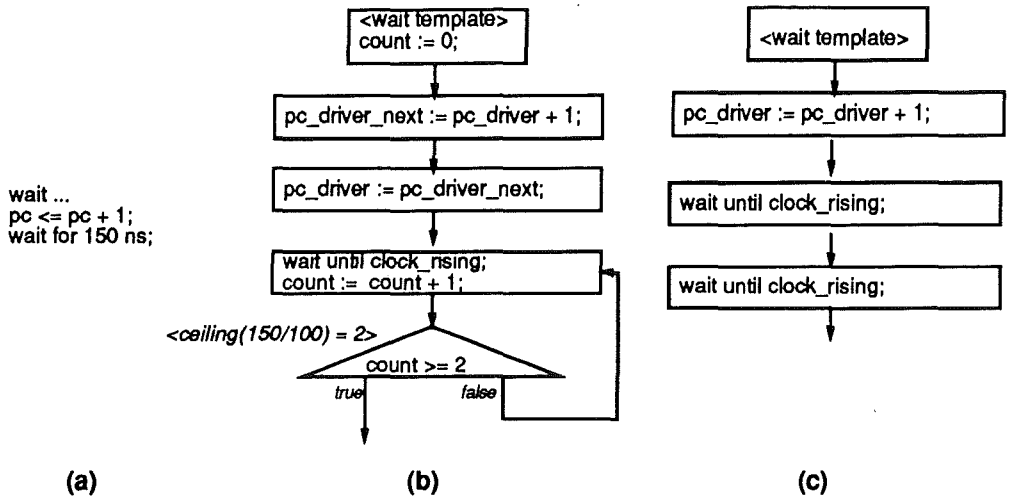


Figure 15: Eliminating the external timer for timeouts in wait statements.

For example, consider the code in Figure 15(a). Assume a clock period of 100 ns. W/S transformations would use the timer portions of the wait-template in Figure 8. Figure 15(b) illustrates a simple CDFG transformation in which a variable called *count* is declared; all occurrences of *count_start* are replaced by $count := 0$, all clocks are followed by $count := count + 1$, and all occurrences of *count_out* are replaced by *count*. Complete independence from an external timer is thus achieved. The design can be further improved by loop unrolling, as shown in Figure 15(c). The variable *count* becomes useless and is therefore eliminated.

5.4 Mapping processes to combinational logic

Many current tools assume that a process with a sensitivity list represents combinational logic. Therefore, several restrictions must be imposed on the allowable sequential statements in such a process, such as allowing only fixed-bound loops, and requiring that all read signals appear in the sensitivity list. We now demonstrate that such an assumption and its subsequent restrictions are unnecessary.

A process with a sensitivity list is equivalent to a process with a single wait statement with the same sensitivity list. Hence, W/S transformations can be applied and the traditional HLS methodology employed. If all signals that are read appear in a sole wait statement's sensitivity list, and if the schedule results in a one state controller, then the state-register serves no purpose so can be eliminated, resulting in combinational logic.

Note that the former restrictions are eliminated. If the behavior can be implemented as combinational logic, it will be. Otherwise, registers and a controller will be used.

The synthesis-tool optimizations presented in this section, ensure that no additional hardware is generated from the W/S transformed description, than would anyway be necessary to preserve the semantics of the signal and wait statement semantics in the original VHDL description.

6 Conclusions

We have implemented the W/S transformations. The implementation includes a VHDL parser and internal representation manipulation routines (approximately 7,500 lines of C code). The input to the W/S transformations is any general VHDL process, and the output is another VHDL process without any signal assignments and containing only wait statements sensitive to the clock. The transformations themselves require under one second of CPU time for a 1000 line VHDL process.

Among the design descriptions that were used to test our W/S transformations were the three-stage processor of Figure 1, the Rockwell I/O backplane custom integrated circuit and an industrial RISC signal processor chip. We are currently integrating the W/S transformations with the VHDL synthesis tool that is under development.

In this paper we have presented a technique to incorporate VHDL signal and wait semantics into high-level synthesis. The W/S transformations increase the expressive power of VHDL specifications that are synthesizable by enlarging the synthesizable subset of VHDL. This reduces the restrictions which are placed on designers writing VHDL behavioral descriptions intended as input to high-level synthesis tools. The W/S transformations are easy to incorporate into existing synthesis methodologies. In addition they can provide a path from VHDL to software which can be mapped to a processor, thus enabling the designer to perform hardware/software tradeoffs from the same input description.

7 Acknowledgements

This work was supported by the Semiconductor Research Corporation (grant #91-DJ-146). We are grateful for their support.

8 References

[1] *IEEE Standard VHDL Language Reference Manual*, 1988.

[2] R. Lipsett, C. Schaefer, and C. Ussery, *VHDL : Hardware Description and Design.* Kluwer Academic Publishers, 1989.

[3] D.D. Gajski, Nikil Dutt, C.H. Wu and Y.L. Lin, *High-Level Synthesis: Introduction to Chip and System Design.* Kluwer Academic Publishers, 1991.

[4] R. Walker and R. Camposano, *A Survey of High-Level Synthesis Systems.* Massachusetts: Kluwer Academic Publishers, 1991.

[5] G. Micheli and D. Ku, "HERCULES - A System for High-Level Synthesis," in *Proc. 25th DAC*, 1988.

[6] D.E. Thomas and P. Moorby, *The Verilog Hardware Description Language.* Kluwer Academic Publishers, 1991.

[7] M. Barbacci, "Instruction Set Processor Specifications (ISPS) : The Notation and Its Applications," in *IEEE Transaction on Computers*, January, 1981.

[8] R. Camposano, L. Saunders, and R. Tabet, "VHDL as Input for High Level Synthesis," *IEEE Design and Test of Computers*, March 1991.

[9] D. Thomas, et. al., *Algorithmic and Register-Transfer Level Synthesis: The System Architect's Workbench.* Kluwer Academic Publishers, 1990.

[10] J. Lis and D. Gajski, "Synthesis from VHDL," in *Proc. of the ICCD*, 1988.

[11] L. Ramachandran, F. Vahid, S. Narayan and D. Gajski, " Semantics and Synthesis of Signals in Behavioral VHDL." UC Irvine, TR ICS 92-28, 1992.

[12] R. Walker, *Design Representation and Behavioral Transformation for Algorithmic Level Integrated Circuit Design.* PhD thesis, Carnegie Mellon Unversity., April 1988.