A Set of Behavioral Modeling Primitives

by

Chakravarthy S. Kosaraju

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

APPROVED:

James R. Armstrong
Dr. James R. Armstrong, Chairman

Dong S. Ha

Walling R. Cyre

Dr. Dong S. Ha

Dr. Walling R. Cyre

June, 1990
Blacksburg, Virginia
A Set of Behavioral Modeling Primitives

by

Chakravarthy S. Kosaraju

Dr. James R. Armstrong, Chairman

Electrical Engineering

(ABSTRACT)

Modeling is an essential step in the design of digital circuits [7]. The coding of behavioral models for complex devices is a labor intensive task. Even with the use of a tool like the "Modeler's Assistant" [4], the development of behavioral models is time consuming and labor intensive. The use of re-usable code along with a tool like the Modeler's Assistant can speed up model development. This thesis defines a set of higher level primitives which can be used for this purpose. These primitives are built as a macro library into the tool. The Modeler's Assistant together with the modeling primitives provides us with a tool that can simplify the process of model development.
Acknowledgements

I wish to thank my parents, grandparents and my brother for their persistent encouragement and support throughout my education. I would like to thank my advisor Dr. James Armstrong for his guidance and support. Thanks are due to Dr. Cyre and Dr. Ha for agreeing to serve on my committee. Also to be thanked are Chang Cho and Sandeep Shah for their help during the various stages of my thesis. I would also like to thank Shailu and Bala who have helped keep life more interesting, and for standing by me at my times of need. Finally, thanks to all my colleagues in the Computer Research Lab for all the interesting discussions on people, life and politics.
# Table of Contents

Chapter 1. Introduction ........................................................................................................... 1
  1.1 Some Important VHDL Features ..................................................................................... 3
  1.2 Contributions .................................................................................................................. 4

Chapter 2. Process Model Graph ............................................................................................ 6

Chapter 3. The Modeler's Assistant ...................................................................................... 13
  3.1 Language Implementation ........................................................................................... 14

Chapter 4. Need for a Set of Primitives ............................................................................... 19
  4.1 A Set of Primitives ......................................................................................................... 20
  4.2 Description of the Primitives ....................................................................................... 24
  4.3 Some Examples and their Primitives ............................................................................ 31

Chapter 5. Software Design and Data Structures ................................................................. 54
  5.1 An Analysis of the Language ....................................................................................... 56
  5.2 Node Representation ..................................................................................................... 60
# List of Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>Figure 1</td>
<td>Process Model Graph Notation</td>
<td>10</td>
</tr>
<tr>
<td>Figure 2</td>
<td>Process Model Graph: Representation of Signals</td>
<td>11</td>
</tr>
<tr>
<td>Figure 3</td>
<td>Process Model Graph: An Example</td>
<td>12</td>
</tr>
<tr>
<td>Figure 4</td>
<td>An Example</td>
<td>22</td>
</tr>
<tr>
<td>Figure 5</td>
<td>Controlled Counter Block Diagram</td>
<td>32</td>
</tr>
<tr>
<td>Figure 6</td>
<td>Controlled Counter PMG</td>
<td>33</td>
</tr>
<tr>
<td>Figure 7</td>
<td>8214 Interrupt Controller Block Diagram</td>
<td>36</td>
</tr>
<tr>
<td>Figure 8</td>
<td>Interrupt Controller PMG</td>
<td>37</td>
</tr>
<tr>
<td>Figure 9</td>
<td>Register Block Diagram</td>
<td>39</td>
</tr>
<tr>
<td>Figure 10</td>
<td>Register PMG</td>
<td>40</td>
</tr>
<tr>
<td>Figure 11</td>
<td>Organization of the URISC Data Unit</td>
<td>45</td>
</tr>
<tr>
<td>Figure 12</td>
<td>PMG of the URISC Data Unit</td>
<td>46</td>
</tr>
<tr>
<td>Figure 13</td>
<td>Hardwired Controller Design for URISC</td>
<td>47</td>
</tr>
<tr>
<td>Figure 14</td>
<td>PMG of the Hardwired Controller</td>
<td>48</td>
</tr>
<tr>
<td>Figure 15</td>
<td>UART Block Diagram</td>
<td>51</td>
</tr>
<tr>
<td>Figure 16</td>
<td>PMG of an UART</td>
<td>52</td>
</tr>
<tr>
<td>Figure 17</td>
<td>System Architecture</td>
<td>55</td>
</tr>
<tr>
<td>Figure 18</td>
<td>Wait Statement Tree Structure</td>
<td>57</td>
</tr>
<tr>
<td>Figure 19</td>
<td>For Statement Tree Structure</td>
<td>58</td>
</tr>
</tbody>
</table>
Figure 20. Process Node Tree Structure ........................................ 60
Figure 21. Node Structure ......................................................... 62
Figure 22. Initial Screen .......................................................... 95
Figure 23. Entering Unit Name: CCTR .......................................... 96
Figure 24. Unit Menu .............................................................. 98
Figure 25. Choosing the Primitive Menu Item from Add Menu .......... 99
Figure 26. Entering the Name for the Primitive .............................. 101
Figure 27. Indicating the Size of the Input Bus ............................... 102
Figure 28. Indicating if the Bus is a Tri-State Bus ............................ 103
Figure 29. Entering the Name for Enable ..................................... 104
Figure 30. Entering the Name for Input Bus ................................... 105
Figure 31. Entering the Name for Output Bus ................................. 106
Figure 32. Entering the Name for the Generic Delay ....................... 107
Figure 33. Indicating if the Latch is Triggered when Enable is High or Low 108
Figure 34. Saving the Primitive .................................................. 109
Figure 35. Comparator Primitive Chosen ...................................... 111
Figure 36. Saving the Comparator Primitive .................................. 112
Figure 37. Saving the Counter Primitive ...................................... 114
Figure 38. Saving the Decoder Primitive ...................................... 115
Figure 39. Saving the Oscillator Primitive .................................... 116
Figure 40. Adding a Signal ........................................................ 118
Figure 41. Prompt for Signal Name .............................................. 119
Figure 42. Signal OUT has been Added ....................................... 120
Figure 43. Adding the Signal EN ............................................... 121
Figure 44. Adding the Signal UPD .............................................. 122
Figure 45. Adding the Signal CLR .............................................. 123
List of Tables

Table 1. ................................................................................. 14
Table 2. ................................................................................. 15
Table 3. ................................................................................. 23
Table 4. ................................................................................. 61
Table 5. ................................................................................. 68
Table 6. ................................................................................. 79
Table 7. ................................................................................. 80
Chapter 1. Introduction

Advancements in VLSI technology have complicated the process of modeling of digital circuits. The number of transistors on present day chips can exceed a million, and as a result of this, circuit design involves handling a large amount of detailed structural information. Simulation for system testing and verification has become necessary due to the inherent problems involved in testing such huge circuits. Higher level programming languages such as Pascal or C are very frequently used to simulate the behavior of circuits before the actual implementation of the hardware. But these higher level programming languages have been found to be inadequate since they do not possess the necessary language constructs to model hardware and also because they lack a graphical representation for representing these hardware blocks [6,7]. The VHDL Hardware Description Language (VHDL) has been found to be very suitable for modeling since it can model hardware accurately [1,2,10,11,12].

VHDL is being extensively used for various applications like system design validation, synthesis and test generation [19]. The language possesses a rich set of constructs and a structured programming style. Given the various features of the language,
more and more designers are gravitating towards the language. However, the language is very complex to learn and hence it becomes very difficult for the designer to model hardware using all the constructs and features of the language. Also, the wide choice of modeling approaches "enjoyed" with VHDL can lead to models that are poorly written, poorly documented, unstructured and hard to understand. Thus, with the designation of a standard version of VHDL in 1987 by the IEEE [1,3], there is an increasing interest in developing CAD tools which can automate the process of VHDL model generation.

A positive aspect of VHDL is that it is ideally suited for a graphical representation of the behavioral hardware models that have been developed in the language. One such representation was first developed by Dr. Armstrong at Virginia Tech. This representation, called the Process Model Graph (PMG), provides a convenient means for breaking up the whole model into processes, and makes the overall functionality of the system more apparent. Various language constructs can be easily represented. A software tool called the Modeler's Assistant accepts this kind of graphical input and produces the VHDL source code interactively. The Modeler's Assistant, which was the result of earlier research conducted at Virginia Tech, simplifies the modeler's job to a large extent. The advantage of such a tool is that it allows the user to graphically piece together processes and to form syntactically correct VHDL behavioral descriptions. It also makes an interactive learning tool.

In spite of it's usefulness, the development of behavioral models on the Modeler's Assistant is still very time consuming and labor intensive. The use of re-usable code along with a tool like the Modeler's Assistant can speed up the model development. The work described in this thesis involved defining a set of modeling primitives which can
be used for this purpose. After a thorough analysis of models and data books, a set of primitives have been defined. Examples of such primitives are: counters, oscillators, decoders, edge detectors, serial (parallel) to parallel (serial) converters, pulse width and set up and hold time checkers etc. A key issue is the completeness property of the proposed primitive set, i.e. can all behavioral models be built from that set? These primitives are built as a macro library into the tool. If the user needs to build the model of a complicated system, then all he needs to do is pick the primitive processes which make up the system from the library and suitably interconnect them. This improves the productivity of the modeler and simplifies the process of model development. It also results in models that have a well defined structure.

1.1 Some Important VHDL Features

We shall discuss in brief, some of the features of the VHDL language that are necessary for understanding the notation used in this thesis. Interested readers are referred to [3,10,11,12] for further details. A design file is comprised of design units. The entity declaration defines the interface of a design unit and describes it’s inputs and outputs. This includes the port name, it’s data type and it’s mode. The generic list in a design entity defines the generic constants associated with the design entity. The behavior of a design entity is usually specified through an architectural body. An architectural body can be either a behavioral body or a structural body. A behavioral architectural body is made up of processes.

A process statement describes the behavior of some portion of the system. It is usually comprised of an optional process label, followed by the key word process, the
sensitivity list, a process declarative part, the key word begin, the process statement part and then the key word end process. The sensitivity list defines the set of signals which the process is sensitive to. Any change on any signal included in this sensitivity list triggers that particular process and causes the execution of the statements in the process statement part. The process declarative part is comprised of constant declarations and variable declarations among others. The process statement part is a set of sequential statements which keep executing in a sequential fashion when the process is triggered by any changes in the signals included in the sensitivity list. An example of a process is given below.

BUFF: process(INTA)
variable INT: BIT;
begin

-------------- Here the functionality
-------------- of the process BUFF is
-------------- defined.

end process BUFF;

1.2 Contributions

The contributions made by this thesis include the following:

- Developed a set of modeling primitives.
• Developed data structures and a primitive editor to create processes using the primitives.

• Evaluated the completeness property of the primitive set, in developing behavioral models.

• Measured the "speed up" in model development time in using the primitives over the construction of models using graphic tokens corresponding to low level VHDL.

• Verified that the VHDL code produced by the primitives produced executable VHDL.
Chapter 2. Process Model Graph

The structure of a design and the interconnections between various components can be represented using a graphical notation [15,17]. Graphical notation provides an ideal means for visually capturing the information in a design [18]. Some of the most popular forms of graphical notation are the schematics. Schematics represent hardware at the transistor, gate, and the flip-flop level. Hardware can be modeled at each of these three levels, and the textual description can be generated. These models are structural.

Hardware can be modeled at higher levels of abstraction too using a behavioral approach. The complex behavior of these models can be represented using text, but the inter-relationships between various components are better illustrated using pictures. Hence, a pictorial representation showing the structure of the behavioral model is essential. The term “structure of a behavioral model” does not mean that the model is structural, i.e. equivalent to a netlist. It implies that the “parts” of the behavioral model have a clearly defined relationship with one another. A graphical representation which does this, called the Process Model Graph, was developed by Armstrong [1]. The design is partitioned into a set of processes. Each of these processes is given a label, and is re-
presented by an oval in the notation. Ports are represented by small circles on the boundary of the oval representing the process. Ports that are to be included in the sensitivity list of the process are represented by shaded circles. Ports that do not trigger the process are represented by unshaded circles on the boundary of the oval. Constants and variables can be declared in a process and are represented by small squares inside the oval representing the process, since they are internal to each process. An example of this is shown in Fig 1. Data flow between the processes is represented by arcs drawn between the ovals representing the processes. These arcs represent signals and they are directed depending on the mode of the ports they connect. Also, in some cases, each arc is labeled with a designator of the form $S(DEL_S)$ where $S$ is the signal name and $DEL_S$ is the delay of the transmitting process. If a process has a port that is of mode out and another has a port that is of mode in, then the arc between these two processes and linking up these two nodes is directed from the first process to the second. If both the processes have a port of mode inout, then the arc between these two processes is bi-directional. This is illustrated in Fig 2. The ports of the processes that are left unconnected by any of these processes are placed in the port list in the entity description. Different line styles are used to represent the different data types. The thickness of the arc represents the type and dimension of the signal. The ports that are logically linked up by these arcs are listed in the signal declaration list in the architecture declarative part.

The PMG notation described above reduces the amount of information presented to the designer and it helps in the logical partitioning of a design into processes. It makes it very easy for the system designer to follow the data flow in the model and gives a better picture of the overall functionality of the system. It gives the designer a means of visual formalization. An example of a behavioral architectural body represented by
the PMG notation is given in Fig 3. The behavioral architectural body has three processes and these are represented by three ovals in the PMG. The shaded circles representing the ports NINT1, NINT2, NINT3 and NINT4 trigger the processes ENCODER and NAND4. The port INTA triggers the process BUFFER but the port VECT does not. The arc labeled VECT represents the data flow between the processes ENCODER and BUFFER. The ports NINT1, NINT2, NINT3, NINT4, INTA, INT and OUT are left unconnected and are included in the entity description.
The VHDL description for the Process Model Graph in Fig 3. is shown below.

entity INT_CONT
  generic(INTDEL, ENCODEL, BUFDEL: TIME);
  port(NINT1, NINT2, NINT3, NINT4, INTA: in BIT;
       INT: out BIT;
       OUTPUT: out MVL_VECTOR(0 to 7));
end INT_CONT;

architecture BEHAVIOR of INT_CONT is
  signal VECT: MVL_VECTOR(0 to 7) := "00011100";
begin

  NAND4: process(NINT1, NINT2, NINT3, NINT4)
  begin
    INT <= not(NINT1 and NINT2 and NINT3
                   and NINT4) after INTDEL;
  end process NAND4;

  ENCODER: process(NINT1, NINT2, NINT3, NINT4)
  begin
    if (NINT1 = '0') then
      VECT <= "00011100" after ENCODEL;
    elsif (NINT2 = '0') then
      VECT <= "00011101" after ENCODEL;
    elsif (NINT3 = '0') then
      VECT <= "00011110" after ENCODEL;
    elsif (NINT4 = '0') then
      VECT <= "00011111" after ENCODEL;
    else
      VECT <= "00011100" after ENCODEL;
    end if;
  end process ENCODER;

  BUFFER: process(INTA)
  begin
    if (INTA = '1') then
      OUT <= VECT after BUFDEL;
    else
      OUT <= "ZZZZZZZZ" after BUFDEL;
    end if;
  end process BUFFER;
end BEHAVIOR;
Figure 1. Process Model Graph Notation
Figure 2. Process Model Graph: Representation of Signals
Figure 3. Process Model Graph: An Example
Chapter 3. The Modeler's Assistant

The Modeler's Assistant is the result of earlier research conducted at Virginia Tech. This CAD tool [4] uses the Process Model Graph as input and develops the VHDL model interactively from a graphical representation. The tool allows users to pick tokens and piece together processes to develop syntactically correct VHDL description. The processes created earlier can be used again and thus it makes it easier to develop behavioral models. The Modeler's Assistant is also an interactive learning tool using a simple menu driven system.

Because of the complexity of the language, only a particular subset is implemented in the Modeler's Assistant. The vast range of constructs in the language makes it virtually impossible to implement every feature. The following section discusses some of the features of the language that have been implemented.
3.1 Language Implementation

The basic statements required for behavioral modeling have been implemented in the tool. Table 1. summarizes these statements.

<table>
<thead>
<tr>
<th>List of Statements Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity</td>
</tr>
<tr>
<td>Architecture</td>
</tr>
<tr>
<td>Process</td>
</tr>
<tr>
<td>If/Then/Else</td>
</tr>
<tr>
<td>While/Loop</td>
</tr>
<tr>
<td>Case</td>
</tr>
<tr>
<td>Wait</td>
</tr>
<tr>
<td>Sequential Signal Assignment</td>
</tr>
<tr>
<td>Variable Assignment</td>
</tr>
</tbody>
</table>

Data objects can be signals, variables, or constants. The initial version of the Modeler's Assistant supported the declaration of signals and variables. The capability of declaring a constant explicitly was not provided. The initial value associated with a variable was treated as a constant value. The tool has been modified to enable the user to declare constants explicitly. Constants can now be declared in the process declarative part, but not in the architectural declarative part. Constants are represented as small squares inside the oval representing the process in the Process Model Graph. The time delays in signal assignment statements are specified through generics. These generics are treated as constant values. Table 2 summarizes the list of data objects supported by the tool.
Table 2.

<table>
<thead>
<tr>
<th>List of Data Objects Supported</th>
</tr>
</thead>
<tbody>
<tr>
<td>Entity Interface Ports</td>
</tr>
<tr>
<td>Architecture Signals</td>
</tr>
<tr>
<td>Process Variables</td>
</tr>
<tr>
<td>Process Constants</td>
</tr>
<tr>
<td>Generics</td>
</tr>
</tbody>
</table>

The ports declared in an entity description are of type signal. Each of these ports can be of mode in, inout, or out, and have a data type associated with them. The data types currently supported are BIT, BIT VECTOR, INTEGER, REAL, BOOLEAN, MVL, and MVL VECTOR. The data structures created have placed a constraint on the maximum size of BIT VECTOR and MVL VECTOR. Currently, the tool supports a size of 32. The interconnections between the process ports specifies a signal representing the data flow between the processes. The process ports which are linked up by these signals are no longer included in the entity interface list. The tool also supports a set of standard operators and a set of standard functions for type conversions and for manipulating the data types. These functions are included as part of a standard package. Interested readers are referred to [4] for a list of the various operators and the data types supported along with the tool.

The Modeler’s Assistant allows the user to create a process or an architectural body. Architectural bodies are created from the processes and so the first step is the creation of a process. The process is created by specifying the name of the process, upon which an oval is drawn on the screen representing this process. Next, process input and
output ports are specified and the system allows the user to place the ports on the boundary of the oval representing the process. Any ports to be included in the sensitivity list are chosen by picking the sense menu and then choosing the particular port. Next, any variables or constants that are to be included are specified and are placed in the oval.

The next step is the creation of the set of sequential statements that represent the behavior of the process. The tool takes the user through a sequence of menus which ultimately allow the user to pick the particular statement that he would like to add to the process. If the user needs to construct an assignment statement, he picks the assignment token and then picks the target port and the value to be assigned to the port. If the value that is to be assigned is a keyboard value, a particular menu option is chosen which would allow a keyboard value to be entered. If the value of another port has to be assigned to the target port, then he chooses the port that has to be assigned to the target port. The next step is to specify a delay for the assignment. This is done by choosing either the inertial delay or transport delay option. Consider the following example of an assignment statement.

```
OUT <= VECT after BUFDEL ;
--- OUT is an output port.
--- VECT is an input port.
--- BUFDEL is the generic delay
--- associated with the assignment.
```
Here, if the user needs to construct the assignment statement, he picks the assignment token first. He then picks the target port, which in our case is OUT. He then has to choose the value that has to be assigned to the output port. In our case, since we are assigning the value of the input port VECT to the output port, we pick the input port VECT next. The user then specifies the delay BUFDEL by picking the inertial delay token first. He then types in the text BUFDEL when he is prompted to enter the generic name for the delay.

As another example, consider the following statement:

```
if (IN = '1') then
    OUT <= transport '1' after DELAY;
else
    OUT <= transport '0' after DELAY;
end if;
```

In order for the above set of statements to be constructed, the user first picks the `if` token. The expression within the `if` is constructed by selecting the port to be tested and the test value. In our case, since the port that needs to be tested is IN, he picks that and then he picks the is token and then the high token, since the user needs to test if the port IN is high. The next thing he needs to do is to construct the assignment statement within the `then` part of the `if` statement, and for this he picks the target port OUT. He picks the menu item `KEY` next, since he needs to assign the value '1' to the output port, and then types the value 1. He then specifies the delay by picking the transport delay token and types the text "DELAY". This concludes the construction of statements within the `then`
part of the if statement. He then backs up by one menu by selecting the done menu item and after this he starts constructing the statements within the else part of the if statement. Since it is another assignment statement, he creates the statement the same way as the previous one. He then backs up again by selecting the done menu item which signals the end of the construction of the else part of the if statement and also the construction of the if statement. This procedure of creating statements and constructing expressions has been illustrated in the User's Manual Section of [4]. Other types of statements are created similarly, by picking tokens.
Chapter 4. Need for a Set of Primitives

From the discussion of the Modeler's Assistant given in Chapter 3, it can be noticed that the development of behavioral models using the tool is a very complex process, where the user is taken through various menus and is made to pick tokens for constructing the models. The user puts together the whole model by picking tokens corresponding to small language constructs, i.e. single VHDL statements. But, when it comes to modeling very complex devices, this process of token picking can prove very cumbersome and hence makes the process of coding for complex devices, a very labor intensive job. However, if the user is supplied with a set of higher level primitives that can be used along with the tool, then his job of model development would become much easier.

With the advances in IC technology, methods of logic design involve the use of MSI/LSI components. Any complex function can be generated using a basic set of AND, OR, NOT, NAND and NOR gates. However, quite a large number of them would be required for realizing the function. Thus, the development of MSI/LSI inte-
grated circuits has provided us with a set of higher level primitives and makes the job of building a complicated system much easier.

All computers are made up of recognizable circuit blocks. The structure and size of these blocks may vary, but they perform the same general functions. On analyzing any general computer, it can be seen that, most of them have shift registers which convert data from serial to parallel and from parallel to serial. For example, data is shifted into a shift register in a serial form and it is transferred out on eight or more parallel lines to a register. Shift registers can also be used for general data handling operations.

Computer operations of various kinds are controlled by counting circuits, e.g. the program counter. They can control the operation of the shift register by counting the number of bits moved into the shift register. Decoders and encoders are used to convert data from one form to another. Data is usually handled in the computer in the binary form. So, the encoders and decoders are used to convert the information coming in from a ASCII or BCD form to the binary form and vice versa. Information needs to stored in memory and hence memory is another candidate for inclusion in our list of primitives. Also, every machine has an adder, and the process of subtraction is nothing more than a 1's complement operation followed by an increment operation. These are some of the circuits which are generally used for building all computers and thus should be included as primitives.

4.1 A Set of Primitives

If a set of behavioral modeling primitives can accomplish the task of building a complicated system, then the same set of basic primitives can be defined in VHDL and
built as a macro library into the tool. Let us consider a system and see what it is comprised of. As an example, we shall consider an Interrupt Controller [1]. A block diagram of the Interrupt Controller is given in Fig. 4. Its VHDL code was discussed earlier in Chapter 2. As it can be seen from Fig. 4, the Interrupt Controller is comprised of a Priority Encoder, a Tri-State Buffer and a four-input Nand gate. The PMG for the Interrupt Controller is given in Fig 3. If one had to build the model of the Interrupt Controller by constructing the individual statements, then the whole process of developing the VHDL code for this device would be a time consuming and cumbersome one. If one had the Priority Encoder, the Tri-State Buffer and the four-input NAND gate as primitives, then all one would have had to do, would be to pick these primitives from the set of primitives and specify the port types and the dimension ranges of the ports. The next logical step would be the interconnection of the primitives representing the Priority Encoder, the Tri-State Buffer and the Nand gate. This obviously simplifies things and makes the job of the modeler much easier, when one compares this to the earlier approach. The time one would have to spend over each model is vastly reduced and the productivity of the modeler is improved. Other comprehensive systems can be broken down in a similar fashion. It is this kind of approach that we developed [13,14].

Upon studying various models and various comprehensive systems, we have arrived at a set of primitives, which we feel would satisfy the needs of the designer and the modeler [8,9,16,21,22,23,24]. The set of primitives is given in Table 3.
Figure 4. An Example
Table 3.

<table>
<thead>
<tr>
<th>A Set of Primitives</th>
</tr>
</thead>
<tbody>
<tr>
<td>Oscillator</td>
</tr>
<tr>
<td>Counter</td>
</tr>
<tr>
<td>RAM</td>
</tr>
<tr>
<td>Truth Table</td>
</tr>
<tr>
<td>State Table</td>
</tr>
<tr>
<td>SR Flip Flop</td>
</tr>
<tr>
<td>D Flip Flop</td>
</tr>
<tr>
<td>Parity Checking Circuit</td>
</tr>
<tr>
<td>Comparator</td>
</tr>
<tr>
<td>Decoder</td>
</tr>
<tr>
<td>Multiplexer</td>
</tr>
<tr>
<td>Adder</td>
</tr>
<tr>
<td>Population Counter</td>
</tr>
<tr>
<td>Parallel-to-Serial Converter</td>
</tr>
<tr>
<td>Serial-to-Parallel Converter</td>
</tr>
<tr>
<td>Buffer</td>
</tr>
<tr>
<td>Shift Register</td>
</tr>
<tr>
<td>Incremener</td>
</tr>
<tr>
<td>Priority Encoder</td>
</tr>
<tr>
<td>Register</td>
</tr>
<tr>
<td>Latch</td>
</tr>
<tr>
<td>Hold-Time Check</td>
</tr>
<tr>
<td>Setup Check</td>
</tr>
<tr>
<td>Pulse Width Check</td>
</tr>
</tbody>
</table>

In the section 4.3 we present some more examples of comprehensive systems and break them down into a set of primitives. This shall illustrate that comprehensive systems can be broken down into the primitives. Then, if one needs to develop the models for these
systems, all one needs to do is to link up the primitives which make up these systems and generate the VHDL code automatically.

4.2 Description of the Primitives

In the following section, we describe in brief each of the primitives. A list of the primitives available with the tool have been listed in Table 3. We consider each of those and list the inputs and outputs of each one of them. The user is prompted for the names of the inputs and the outputs. The names that have been given below are some general names for the inputs and the outputs. The generic names for the delays in most of these primitives have already been incorporated into the tool. In some other cases, the user is prompted to enter the name for the generic delay.

- Oscillator

   The Oscillator is triggered when the control input is 1 and stops irrespective of the current state when the control input is 0.

   Inputs : RUN, CLK
   Outputs : CLK
   Generics : CLK_PER1, CLK_PER2

   The RUN input triggers the oscillator. CLK is the output as well as the input of the oscillator (feedback). The generic time delays CLK_PER1, and CLK_PER2 give the low and high periods of the clock. The generic names are already incorporated into the tool and need not be provided by the user.

- Counter

   Inputs : UPD, CLR, EN, CLK, Counter type, DATA
Outputs : DATA
Generics : COUNT_DEL
The input UPD indicates the Up or Down nature of the counter. The input CLR
resets the counter. The counter is enabled when the CLK input is rising and the
enable input EN is low. If the UPD input is high, then the counter counts up, else
it counts down. The counter is reset on the rising edge of the CLR input. The size
of the input bus, namely the counter type needs to be provided by the user. The
generic time delay COUNT_DEL has already been incorporated.

- RAM
Inputs : READ, WRITE, CS, Address bus size, ADDR, DATA bus size, DATA,
Number of words, Size of word
Outputs : DATA
Generics : RDEL, WDEL, DISDEL
If the chip select input CS is low, and if the read input READ is high, then the data
is read from the memory. If the write input WRITE is high, then the data is written
into the memory. If CS is high, then the output DATA is tri-stated. The generic
delay RDEL is associated with reading the data from the memory. The delay WDEL
is the delay in writing information into the memory. The delay DISDEL is the delay
in tri-stating the bus. The address bus size and the data bus size need to be provided
by the user.

- Truth Table
Inputs : A maximum of 4
Outputs : A maximum of 4
The values of the truth table that are provided by the user are encoded in the form of an aggregate.

- **State Table**
  
  Inputs : A maximum of 4
  
  States : A maximum of 16
  
  The state table values are encoded in the form of an aggregate.

- **SR Flip Flop**
  
  Inputs : SET, RESET
  
  Outputs : OUT
  
  If the SET input is high and the reset input RESET is low, then the output OUT is high. If both SET and RESET are low, then the output does not change. If RESET is high and SET is low, then the output is low. If both SET and RESET are high (which is not allowed), the OUT is set to an arbitrary state, which in our case is high.

- **D Flip Flop**
  
  Inputs : CLK, SET, RESET, DATA
  
  Outputs : Q, QB
  
  If the clock pulse CLK falls, and both the SET and RESET inputs are low, then the output Q copies the input DATA, and the output QB copies the complement of the input DATA. If the set input SET is high and RESET is low, the output Q is assigned 1 and the output QB is assigned 0. If the set input SET is low and RESET is high, then the output Q is assigned 0 and QB is assigned 1. The set and reset inputs override the clock.

- **Parity Checking Circuit**
Inputs : DATA, Input bus size

Outputs : OUT

The input bus size needs to be provided by the user. The parity of the input DATA is checked and if the parity is odd, then a ‘1’ is output on the output OUT. If parity even, then OUT is assigned ‘0’.

- Comparator

Inputs : EN, Input bus size, A Input, B Input

Outputs : AGB, ALB, AEB

The user needs to provide the size of the input buses. If the enable input EN is low, then all the outputs are low. If EN is high, and if the A input is greater than B, then the output AGB is made high and the rest low. If the A input is less than B, then the output ALB is made high and the rest low. If both the inputs are equal, then the output AEB is made high and the rest low.

- Decoder

Inputs : CS, decoder size, Names for the decoder inputs

Outputs : Data pins

If the chip select input CS is high, then depending on the values of the decoder inputs, the corresponding data output is selected. The user needs to provide the decoder size.

- Multiplexer

Inputs : Number of Select lines, EN, Select lines, Data Inputs

Outputs : OUT
The user provides the number of select lines available on the chip. If the enable input EN is low, then depending on the value of the select lines, the corresponding data input is passed on to the output OUT after a delay.

- Adder
  Inputs : Size of the Input bus, I1, I2, CIN
  Outputs : COUT, SUM
  The user provides the size of input buses. The sum of the input buses I1 and I2 and the carry in CIN is computed and the carry out and the sum is passed onto the outputs COUT and SUM after a delay.

- Population Counter
  Inputs : Size of the Input bus, IN
  Outputs : OUT
  The user needs to provide the size of the input bus. The number of ones in the input bus IN is calculated and the bit vector value of that is passed onto the output bus OUT.

- Parallel-to-Serial Converter
  Inputs : Input bus size, CLOCK, IN
  Outputs : OUT
  If the clock input CLOCK is rising then the input IN is transmitted serially to the output OUT starting from the least significant bit.

- Serial-to-Parallel Converter
  Inputs : Size of the output bus, EN, CLK, SI
  Outputs : OUT
If the clock signal CLK is rising and if the enable input EN is high, then the serial input SI is shifted in and transmitted to the output OUT.

- **Buffer**
  
  **Inputs**: Size of the Input bus, EN, INP, Generic delay name for the buffer  
  **Outputs**: OUT  
  The user needs to specify if the buffer is a tri-state buffer. If the buffer is a tri-state bus and the enable input EN is low, then the output OUT is tristated. If the enable input is high, then the input INP is passed onto the output OUT after a delay.

- **Shift Register**
  
  **Inputs**: Size of the Register, STRB, CLK, IN  
  **Outputs**: OUT  
  If the clock signal CLK is rising and if the strobe input STRB is high, then the bits of the output bus OUT are shifted by one bit. The serial input IN is passed onto the least significant bit of the output.

- **Incrementer**
  
  **Inputs**: Size of Input bus, INP  
  **Outputs**: SUM, COUT  
  The user needs to provide the size of the input bus. The current value of the input INP is incremented and the output is passed to SUM. In case of a carry out, the value is passed to the output port COUT.

- **Priority Encoder**
  
  **Inputs**: Size of the Priority Encoder, INP, EN  
  **Outputs**: OUT, RA
The user needs to specify if the priority encoder has tri-state buses. If the enable input EN is high, then the request activity output RA (this output is used when chips are cascaded) is made high. If EN is low, then the priority value is encoded and passed to the output OUT.

- **Register**
  Inputs : Size of the Input bus, INP, CLK, Generic name for the Register delay
  Outputs : OUT
  The user needs to specify if the input bus INP is a tri-state bus, and if the register is triggered on the rising or falling edge of the enable input CLK.

- **Latch**
  Inputs : Size of the Input bus, INP, EN, Generic name for the Latch delay
  Outputs : OUT
  The user needs to specify if the latch is triggered when the enable input EN is high or low, and also if the input bus INP is a tri-state bus.

- **Hold-Time Check**
  Inputs : Generic name for the hold time
  The user needs to pick the data port and the strobe port that have to be tested.

- **Setup Check**
  Inputs : Generic name for the setup time
  The user needs to pick the data port and the strobe port that need to be tested.

- **Pulse Width Check**
  Inputs : Generic name for the pulse width time
The user needs to pick the strobe port that has to be tested.

4.3 Some Examples and their Primitives

In the section 4.1 we had considered a very simple example of an Interrupt Controller. In this section, we present a set of complicated systems. We analyze these systems and break them up into a set of primitives. We present the Process Model Graph and a block diagram for each of these.

Example 1:

Consider the Controlled Counter illustrated in Fig. 5. A description of the Controlled Counter is given in [1]. The Process Model Graph for the same is illustrated in Fig. 6. The PMG illustrates that the Controlled Counter model is comprised of a set of five processes. The five processes are LATCH, COMP, DECODE, COUNTER, OSCILL. As it can be observed from the block diagram and the PMG, the Controlled Counter is essentially comprised of a set of primitives corresponding to the five processes, i.e. latch, comparator, decoder, counter, and oscillator. These primitives have been built into the tool and Controlled Counter model can be built from them. The process of developing the model of a Controlled Counter using the primitives is thus much easier and faster than it would be, if one had to construct the model from tokens and simple VHDL constructs.
Figure 5. Controlled Counter Block Diagram
Figure 6. Controlled Counter PMG
Example 2:

The next example we shall consider is that of an 8214 Priority Interrupt Controller. The block diagram is given in Fig. 7 and the Process Model Graph of the same is given in Fig. 8. A description of the 8214 Priority Interrupt Controller can be obtained from [27]. As apparent from the PMG, the behavioral model is comprised of a set of primitives corresponding to the processes, i.e. priority encoder, comparator, register, 3-input and gate, 2-input or gate, and D flip-flop. A few specialized processes are necessary for constructing the model of the 8214. The specialized processes are given below.

- PROC_A - This is a special process for which there is no corresponding primitive. It would be constructed using the low level tokens, and would result in the following code.

```vhd
PROC_A : process(ELR,ELTC2,RA,IN)
variable 01 : Bit;
begin
01 := RA and ELTC2 and (not ELR);
if (01 = '0') then
 A0_2 <= "111" ;
else
 A0_2 <= not (OUT) ;
end if;
end process PROC_A;
```
• AND5 - This process represents a five input AND gate and the behavior is given by the following single assignment statement, which again would be constructed by low level tokens.

\[ Q1 \leq RA \text{ and ELTC and IO and Q1 and INTC} \; \]

The Priority Interrupt Controller has been broken down into a set of primitive processes, most of which correspond to our set of primitives.
Figure 7. 8214 Interrupt Controller Block Diagram
Figure 8. Interrupt Controller PMG
Example 3:

The next example shall be that of a buffered Register. The block diagram is given in Fig. 9 and the Process Model Graph of the same is given in Fig. 10. Interested reader may refer to [1] for a description of the Register. The behavioral model can be built from the latch, buffer, and a special select circuit. The select circuit can be realized using a Truth Table primitive. The Register can be modeled using the three primitives listed above.
Figure 9. Register Block Diagram
Figure 10. Register PMG
Example 4:

The next example shall be a Ultimate Reduced Instruction Set Computer (URISC). The block diagram showing the organization of the data unit of a microprogrammed URISC is given in Fig. 11 and the PMG for this is given in Fig. 12. The working of the URISC is elaborated in [20]. The Hardwired controller design for URISC is given in Fig. 13 and the PMG for the same is given in Fig. 14. The behavioral model of the PMG in Fig. 12 is comprised of the following processes:

- **REG_R** - The primitive corresponding to this is a Register. This register is shown as the register R in the block diagram of the data unit shown in Fig. 11.

- **LATCH** - The primitive corresponding to this is a Latch. This latch is essential for latching in the Cin control input.

- **ADDER** - The primitive is an Adder.

- **MUX** - The primitive corresponding to this is a multiplexer. This primitive process models the behavior of the bus on the left.

- **REG** - The primitive is a Register and it corresponds to the register PC in the block diagram of the data unit given in Fig. 11.

- **REG_A** - The primitive is a Register and this corresponds to the register MAR in Fig. 11.

- **PROC_B** - This process is used for modeling the effect of the COMP control input and does not correspond to a primitive. If the COMP input is high, then the com-
plement of the register R is fed to the adder. Else, all zeroes are fed to the adder.
This is a special process whose behavior is given by the following statements:

    if (COMP = '1') then
        DOUTC <= not_DINR ;
    else
        DOUTC <= "0000000000000000" ;
    end if;

- COMP - The primitive is a Comparator. This primitive is essential for comparing
  the output of the adder with a zero. It sets the Z and N latches when the output of
  the adder is equal to zero, and less than zero respectively. The port STRB is mapped
  to '1' and the second comparator input is set to all zeroes when the ports are
  mapped.

- MEMORY - The primitive is a RAM.

- LATCH_1 - The primitive is a Latch. If the output of the adder is zero, then a 1
  is passed to the output of this latch. This is equivalent to setting the output of a
  flip-flop to 1. This corresponds to the unit Z in Fig. 11.

- LATCH_2 - The primitive is a Latch. This corresponds to the unit N in Fig. 11.

- PROC_C - This is a special process which does not correspond to a primitive and
  is used to model the behavior of the unit MDR in Fig. 11. The behavior of this
  process is given by the following set of statements:
If ((MDRIN = '1') and not MDRIN'Stable)) then
    DOUTD <= DIN ;
else
    DOUTD <= DATA ;
end if;

The primitives for modeling the data unit have been given above. Next, we shall consider
the control unit of the URISC shown in Fig. 13 and list the primitives needed for mod-
eling the control unit. The behavioral model of the PMG in Fig. 14 is comprised of the
following processes:

- **OSCILL -** The primitive corresponding to this is an Oscillator.

- **COUNTER -** The primitive corresponding to this is a Counter. The port UPD is
  mapped to a '1' to indicate the Up nature of the counter.

- **DECODER -** The primitive is a Decoder.

- **PROC_D -** A primitive for this process doesn't exist and the behavior of this
  process is given by the following set of statements:

    C0 <= IND(0) ;
    C1 <= IND(1) ;
    C2 <= IND(2) ;
    C3 <= IND(3) ;
• PROC_E - This is an independent process for which a primitive does not exist. Its behavior can be modeled by modeling the set of gates in Fig. 13.

The hardwired controller has also been analyzed and the primitives listed above. Even though, the entire set of primitives for modeling the entire URISC are not available, a sizable number of them are available and the rest can be created.
Figure 11. Organization of the URISC Data Unit
Figure 12. PMG of the URISC Data Unit
Figure 13. Hardwired Controller Design for URISC
Figure 14. PMG of the Hardwired Controller
Example 5:

As a final example, we shall consider the UART [1]. The block diagram is given in Fig. 15 and the Process Model Graph of the same is given in Fig. 16. A description of the UART is given in [1]. As evident from the PMG, the behavioral model is comprised of the primitive processes corresponding to a register, buffer, parallel to serial converter, oscillator, serial to parallel converter, and some specialized processes. The behavior of the special processes is given below:

- **PRO_A** - This is a special process whose behavior is given by the following set of statements.

  If (not IN1'Stable) then
  NINT0 <= '1' after INTDEL,
  '0' after PER ;
  end if;

- **PRO_B** - The behavior of this process is given by the following set of statements.

  ISTRT <= '0' ;  --- ISTRT is of mode Inout
  NINT1 <= '0' after INTDEL,
  '1' after PER2 ;
  CLK <= '1' after DEL ;

- **PRO_C** - This is a special process whose behavior is given by the following set of statements.
If (ISTRT = '1' and not ISTRT'Stale) then

RUN <= transport '1' after C_PER1,
     '0' after C_PER2;

end if;

The UART has been broken down into nine processes. Six of the nine processes can be described using the primitives that are available in our list of primitives and the behavior of the other two can be described by creating the required processes with the set of statements given above.
Figure 16. PMG of an UART
It has been demonstrated that complicated systems can be broken down into a set of logic blocks. Most of these logic blocks correspond to primitives. The set of primitives given in Table 3 covers most of the functions necessary for modeling these complicated systems.
Chapter 5. Software Design and Data Structures

The architecture of the Modeler’s Assistant has an internal database [4], in which the VHDL code is stored in an intermediate format. This intermediate format is a tree structure which is modified and expanded whenever any changes are made to the processes or units. The architecture in [4] has been modified. The modified architecture of the Modeler’s Assistant is shown in Fig. 17. The tree structure of the internal database is traversed by the Process and the Design Unit editors. The Process editor creates the processes and also edits them. Statements, ports, constants, variables, etc. are added by the Process editor. The Design Unit editor links up the various processes by adding signals. The Design Unit editor also has a Primitive editor which adds any primitive that the user needs. The Primitive editor contains the routines which call up the tree structure of the primitives from the internal database. This tree structure is linked up as a branch to the whole tree structure corresponding to a unit. The VHDL generator contains the routines which generate the VHDL code. This generator traverses the tree structure corresponding to a unit, process or a primitive and generates the appropriate VHDL code that the user might need.
The tree structure which is needed as an input to the VHDL generator is called up from the internal database. A pointer is placed at the top of a unit, process, or a primitive, and then the control is passed to the VHDL generator. The structure of the Modeler’s Assistant also shows three libraries. The first one is the Unit library which contains the set of files created with the extension .unt. This library contains information corresponding to the internal data structure of a unit. When a unit is saved after it’s creation, then the information about the unit is saved in the Unit library with the .unt extension.
The same applies when processes and primitives are created. The Process library contains the internal representation of the processes and the Primitive library contains the internal representation of the primitives. Both of these are stored in their respective libraries with the file extensions .mod.

The MetaWINDOW graphics library has been used for the creation of the various graphics routines. In addition, some basic DOS calls and functions have also been used for creating the graphics. The library and the DOS calls can be used for drawing windows, creating figures and different shapes, basic mouse functions, getting the inputs from the screen, etc. The list of displays supported by MetaWINDOW is given in [4]. The graphics library along with the menu system, the help system, mouse driver, and the window functions comprise the secondary system components. The Modeler's Assistant has a menu-driven editor with a virtual coordinate drawing area. Both small and large drawings can be easily handled because of this feature. When the cursor is over a particular menu, then that menu item is highlighted. Further details of the secondary system components can be obtained from [4].

5.1 An Analysis of the Language

An analysis of the statements used for behavioral modeling is essential for understanding the internal representation of the statements. The statements are decomposed into a tree structure. An internal node is defined for each node in the tree structure. Consider an example of a wait statement:

```
wait on INT until (INT = '1') for 100 ns;
```
The tree structure for the wait statement is shown in Fig. 18.

As it can be noticed from the figure, the wait node has four pointers. The first node points to the list of signals in the sensitivity list of the wait statement. The next pointer points to the Boolean expression in the condition clause of the wait statement. The third pointer points to the time expression in the timeout clause. The fourth pointer points to the next statement in the list of statements. A node structure for a for statement has been added to the tool to provide additional capability while creating the primitives. Consider a for statement as an example for illustrating the language analysis.

```plaintext
for I in 7 downto 0 loop
    A := A + 1;
end loop;
```
The tree structure for the for statement is shown in Fig 19.

![Diagram of for statement tree structure]

Figure 19. For Statement Tree Structure

The for node has six pointers pointing to the various values and expressions necessary for creating the for statement. The first pointer field contains the integer value of the left bound of the discrete range. The second pointer field contains the integer value of the right bound of the discrete range. The third and fourth pointers point to the statements within the for loop. The fifth pointer encodes a value which indicates whether the for loop counts up or down. The sixth node points to the next statement in the list of statements of the process.

A few dummy nodes have been added to the list of nodes to store the information about the values in an aggregate. Another node called the check node has been added to store the internal representations for the pulse width, hold time and setup time checks. This shall be further elaborated upon in the later sections. An analysis of the if and the
signal assignment statements can be obtained from [4]. All the statements in the behavioral model are broken up into the kind of tree structure that has been illustrated for the wait statement and the other statements. All the nodes corresponding to the different statements have a pointer pointing to the next statement in the list of statements. So, if one travels down this list of pointers pointing to the next statement in the list, then the various statements can be constructed by piecing together the various pointers, and the behavioral model for the entire process can be created.

The initial implementation of the Modeler's Assistant constrained the maximum of pointers per node to four. But this was found to be inadequate after adding some capabilities to the tool. Initially, each pointer corresponding to a process, had a pointer pointing to the first variable in the list of variables, the first port, the first generic, and the top statement. So, an initial choice of four pointers per node was made. But after adding the capability of adding constants and types, the number of pointers per process node has been increased to six. The choice of adding a type directly in a process has not been implemented, but this node is used when a type declaration needs to declared, when constructing a primitive. Thus each process node has six pointers and this is illustrated in Fig. 20. It should be noted that not all nodes need six pointers and some nodes use these pointer fields to store information about the data types, operator values and such other information. A detailed description of the node structure of the various nodes is given in the following sections. The various data encoding techniques are given in [4].
5.2 Node Representation

The statements in a process are parsed and a tree structure is created. Every node in the tree structure is represented by an internal node and this node is the basic element of representation of the whole data structure. Every node has a set of fields and the fields of the various nodes are used for representing different information depending on the type of the node. A list of the internal node types are given in [4]. The node types that have been added to the list of nodes in [4] are given in Table 4.
Table 4.

<table>
<thead>
<tr>
<th>Name</th>
<th>Label</th>
<th>Value</th>
<th>Corresponding VHDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type</td>
<td>Type</td>
<td>22</td>
<td>Type Definition</td>
</tr>
<tr>
<td>While</td>
<td>While</td>
<td>28</td>
<td>While/Loop</td>
</tr>
<tr>
<td>For</td>
<td>For</td>
<td>29</td>
<td>For/Loop Statement</td>
</tr>
<tr>
<td>Aggregate</td>
<td>Aggregate</td>
<td>30</td>
<td>Aggregate values</td>
</tr>
<tr>
<td>Record</td>
<td>Record</td>
<td>31</td>
<td>Record node</td>
</tr>
<tr>
<td>Constant</td>
<td>Constant</td>
<td>32</td>
<td>A Constant</td>
</tr>
<tr>
<td>Setup</td>
<td>Check</td>
<td>33</td>
<td>Check statements</td>
</tr>
<tr>
<td>Holdtime</td>
<td>Pulselwidth</td>
<td>Checks</td>
<td></td>
</tr>
<tr>
<td>Record</td>
<td>Rec</td>
<td>34</td>
<td>Record Elements</td>
</tr>
</tbody>
</table>

As discussed in the previous section, the structure of the node has been modified. The node structure is shown in Fig. 21.
Every node in our tree structure has six pointers. The various fields of the node are:

- **nodeName** - This field contains the text information about the node. It stores information about the node name or the text representations that need to be printed out.

- **NodeType** - This field indicates the type of the particular node depending on the integer value that is stored in this field. The integer values for the nodes that have been added to the list of nodes are also given in Table 4.

- **NodeRect** - This field is a structure that contains two points that describe the (x,y) coordinates of the object.
• NodePtr(i) - This is the last field of the node structure, and as mentioned before, it has six pointers. When not used as pointers, the values in this field perform different functions depending on the type of the node. They may store information about the node, as might be the case in constants, variables, etc. The functions of each of these pointers will become more apparent when the node structure for the nodes that have been added to the list of nodes is discussed. When used as pointers, the integer values in this field point to the next node in the tree structure.

Macros have been defined for the nodes that have been added and are included in the file Macros.h. The macro names and the fields for the nodes are explained in Table 5. In the discussion that follows, the various nodes that have been modified and the nodes that have been added are discussed. The meanings of the fields for these nodes have been explained.

• Module - This node corresponds to a single process block. The NodeName field of this node gives the name of the process, the NodeType field indicates that it represents a process, and the NodeRect field gives the coordinates of the oval representing the process. When a process is reloaded, the new coordinates are calculated based on the relative offsets. The last field of the module has six pointers. The first pointer points to the linked list of ports. The second pointer points to the linked list of variables, the third pointer points to the generics, the fourth pointer points to the top statement in the list of statements, the fifth pointer points to the list of type declarations, and the sixth pointer points to the list of constants in the process. Each of these pointers points to the first entity in the linked list. The pointer values are absolute. When the process is saved, the current node numbers are written to the
disk and when a process has to be reloaded, a lookup table is used to allocate new numbers depending on the old numbers.

- **Variable** - In the case of a variable, the NodeName field is used to store the name of the variable, the NodeType field indicates that the particular node is a variable, and the last field has five pointers. Only five of the six pointer fields are used and the NodeRect field is redundant. The first pointer points to the next variable in the list. The second pointer field is not used as a pointer, but it stores a value which indicates the type of the variable. The third pointer field indicates whether the particular variable has an aggregate value as it's initial value. The fourth field of the pointer field points to the first value of the aggregate, i.e. if an initial aggregate value for the variable exists. The fifth field of the pointer field gives the node number of the type to which the variable belongs to. The NodeRect field is redundant in the case of a variable.

- **Constant** - Constants are represented like variables. The NodeName field gives the name of the constant, the NodeType field indicates that it is a constant, and the NodeRect field is redundant. The last field has six pointers unlike the case of the variables. The first pointer points to the next constant in the list of constants. The second pointer field is not used as a pointer, but is used to store a value which indicates the type of the constant. The third pointer field stores a value which indicates whether the constant has an aggregate value as the initial value. The fourth pointer field points to the first aggregate value in the aggregate list. The sixth pointer is used for a similar purpose. These aggregate fields, in the case of variables and constants are used for encoding the values that the user inputs in the truth table or
state table, when encoding the state or truth table primitives. The fifth pointer field
gives the node number of the type to which the constant belongs.

- **For** - The NodeName, and NodeRect fields are not used in the case of a For node.
The NodeType field stores a value which indicates that the node is a for node. All
the six pointer fields are used. The first pointer field stores a value which gives the
first loop index. The second pointer field stores a value which gives the second loop
index. The third pointer field points to the first statement in the sequence of state-
ments within the loop. The fifth pointer field points to the next statement within
the loop. The fourth pointer points to the next statement after the loop statement.
The sixth pointer stores a value which gives the direction of the loop, i.e. whether
the loop counts up or down.

- **Type** - The NodeName field gives the name of the type and the NodeType field
stores an integer value which indicates that the particular node is a type node. The
NodeRect field is redundant and the last field has six pointers. The first pointer field
points to the next type declaration in the process declarative part. The second and
third pointer fields have an integer value which give the range of the type. The
fourth pointer field stores a value which gives the node number of the type, of which
this particular node is a type. The type of this node which can be a bit, integer, a
user-defined type, etc. The fifth pointer field stores a value which indicates whether
the type is a record type. The sixth pointer points to the first element of the record,
in case the type is a record type.

- **Aggregate** - The NodeName and the NodeType fields serve the usual purpose. The
first to the fourth pointer fields hold the aggregate values and the fifth pointer field
serves as a pointer to the next aggregate element. The design of the system has
constrained the maximum truth table size to four inputs and four outputs, and the state table size to four inputs and sixteen states. This obviously implies that any row of the truth table or the state table would have at the most four values. Hence, only four pointer fields have been used to store the values. In case the capabilities of the system need to be expanded, the sixth pointer field could also be used to store an aggregate value, or the number of pointers per node could be increased.

- **Record** - The NodeName and the NodeType fields serve the usual purpose. The first pointer field points to the first record element. Similarly, the second, third, and fourth pointer fields point to the second, third, and fourth record elements respectively. The fifth pointer field points to the next record element. This field and the sixth pointer field can be used for enhancing the system capabilities. The fifth pointer field can be used for forming a linked list of the next record element, in case the record has more than the four elements that can be accommodated by the four pointer fields.

- **Rec** - The pointer fields of the Record node point to a node of the Rec type. This node actually holds the information about the particular record element, namely the type, name etc. So, the NodeName field of the Rec node holds the name of the element. The second pointer field holds the type of the record element and the other pointer fields can be used for future enhancements.

- **Check** - The NodeName field holds the textual information about the type of the check being performed. The type of checks currently being implemented in the tool are hold time, setup time and pulse width checks. The first pointer field also holds an integer value which indicates the type of the check. The second pointer points to the data port that is being tested. The third pointer points to the strobe port being
tested. The fourth and fifth pointer fields have been left unused. The sixth pointer field points to the generic node which gives the amount of time, for which the check is supposed to hold.

The names for the various node fields that have been added or modified are given in Table 5. Of these, the nodes Module and Variable have been modified and the rest have been added to the system.
<table>
<thead>
<tr>
<th>Node Name</th>
<th>Field Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Module</td>
<td>ModuleName</td>
<td>String containing the name of the process</td>
</tr>
<tr>
<td>TopPort</td>
<td></td>
<td>Pointer to the first port declaration of this process</td>
</tr>
<tr>
<td>TopVariable</td>
<td></td>
<td>Pointer to the first variable declaration of this process</td>
</tr>
<tr>
<td>TopGeneric</td>
<td></td>
<td>Pointer to the first generic declaration of this process</td>
</tr>
<tr>
<td>TopStmtnt</td>
<td></td>
<td>Pointer to the top of the statement tree for this process</td>
</tr>
<tr>
<td>TopType</td>
<td></td>
<td>Pointer to the first type declaration of this process</td>
</tr>
<tr>
<td>TopConstant</td>
<td></td>
<td>Pointer to the first constant declaration of this process</td>
</tr>
<tr>
<td>Variable</td>
<td>VariableName</td>
<td>String containing the name of the variable</td>
</tr>
<tr>
<td></td>
<td>VariableType</td>
<td>Data type of this variable</td>
</tr>
<tr>
<td>NextVariable</td>
<td></td>
<td>Pointer to the next variable in the list of variables for this process</td>
</tr>
<tr>
<td>VariableOf</td>
<td></td>
<td>Data value which tells if the variable has an aggregate value as it's initial value</td>
</tr>
<tr>
<td>VariableAgg</td>
<td></td>
<td>Pointer to the first value of an aggregate, if it exists</td>
</tr>
<tr>
<td>VariableElems</td>
<td>Pointer to the node number of the type to which this variable belongs</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>Constant</td>
<td>ConstantName String containing the name of the constant</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ConstantType Data type of this constant</td>
<td></td>
</tr>
<tr>
<td>NextConstant</td>
<td>Pointer to the next constant in the list of constants for this process</td>
<td></td>
</tr>
<tr>
<td>ConstantOf</td>
<td>Data value which tells if the constant has an aggregate value as it's initial value</td>
<td></td>
</tr>
<tr>
<td>ConstAggOne</td>
<td>Pointer to the first value of an aggregate, if it exists</td>
<td></td>
</tr>
<tr>
<td>ConstantElems</td>
<td>Pointer to the node number of the type to which this constant belongs</td>
<td></td>
</tr>
<tr>
<td>ConstAggTwo</td>
<td>Another pointer to the aggregate value, if it exists</td>
<td></td>
</tr>
<tr>
<td>Type</td>
<td>TypeName String containing the name of the type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>NextType Pointer to the next type in the list of types for this process</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TypeOps Data value which gives the range of this type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Typeref Data value which gives the range for this type</td>
<td></td>
</tr>
<tr>
<td></td>
<td>TypeElems Stores the data type of this type or a pointer to the type to which this type belongs</td>
<td></td>
</tr>
<tr>
<td>Variable</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>--------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>DummyType</td>
<td>Data value which tells if this type is a record type</td>
<td></td>
</tr>
<tr>
<td>TypeRec</td>
<td>Pointer to the first element of the record if the type is a record type</td>
<td></td>
</tr>
<tr>
<td>For</td>
<td>Data value which gives the first loop index</td>
<td></td>
</tr>
<tr>
<td>ForLeft</td>
<td>Data value which gives the second loop index</td>
<td></td>
</tr>
<tr>
<td>ForRight</td>
<td>Pointer to the first statement in the list of statements within the loop</td>
<td></td>
</tr>
<tr>
<td>NextFor</td>
<td>Pointer to the next statement in the statement tree</td>
<td></td>
</tr>
<tr>
<td>NextStmt</td>
<td>Pointer to the next statement after the first statement in the loop</td>
<td></td>
</tr>
<tr>
<td>NexttoFor</td>
<td>A pointer to the next statement after the first statement in the loop</td>
<td></td>
</tr>
<tr>
<td>ForType</td>
<td>Data value which gives the direction of the loop (up or down)</td>
<td></td>
</tr>
<tr>
<td>Aggregate</td>
<td>String containing the name of the aggregate</td>
<td></td>
</tr>
<tr>
<td>AggName</td>
<td>Pointer to first set of aggregate elements</td>
<td></td>
</tr>
<tr>
<td>Aggl</td>
<td>Pointer to the second set of aggregate elements</td>
<td></td>
</tr>
<tr>
<td>Agg2</td>
<td>Pointer to the second set of aggregate elements</td>
<td></td>
</tr>
<tr>
<td>Agg3</td>
<td>Pointer to the third set of aggregate elements</td>
<td></td>
</tr>
<tr>
<td>Agg4</td>
<td>Pointer to the fourth set of aggregate elements</td>
<td></td>
</tr>
<tr>
<td>NextAgg</td>
<td>Pointer to the next aggregate element</td>
<td></td>
</tr>
<tr>
<td>Record</td>
<td>String containing the name of the record</td>
<td></td>
</tr>
<tr>
<td>RecordName</td>
<td>String containing the name of the record</td>
<td></td>
</tr>
<tr>
<td>Name</td>
<td>Description</td>
<td></td>
</tr>
<tr>
<td>----------------</td>
<td>-----------------------------------------------------------------------------</td>
<td></td>
</tr>
<tr>
<td>FirstRec</td>
<td>Pointer to the first record information stored in the Rec node</td>
<td></td>
</tr>
<tr>
<td>SecondRec</td>
<td>Pointer to the second record information stored in the Rec node</td>
<td></td>
</tr>
<tr>
<td>ThirdRec</td>
<td>Pointer to the third record information stored in the Rec node</td>
<td></td>
</tr>
<tr>
<td>FourthRec</td>
<td>Pointer to the fourth record information stored in the Rec node</td>
<td></td>
</tr>
<tr>
<td>NextRec</td>
<td>Pointer to the next record element in case more record elements need to be added</td>
<td></td>
</tr>
<tr>
<td>Check</td>
<td>String containing the name of the type of check (Setup, Hold Time, Pulse Width)</td>
<td></td>
</tr>
<tr>
<td>TypeName</td>
<td>Data value which indicates the type of check</td>
<td></td>
</tr>
<tr>
<td>CheckType</td>
<td>Pointer to the data port that needs to be tested</td>
<td></td>
</tr>
<tr>
<td>DataPort</td>
<td>Pointer to the strobe port that needs to be tested</td>
<td></td>
</tr>
<tr>
<td>StrbPort</td>
<td>Pointer to the generic value of the time</td>
<td></td>
</tr>
<tr>
<td>CheckTime</td>
<td>String containing the name of the aggregate element</td>
<td></td>
</tr>
<tr>
<td>Rec</td>
<td>Data type of this record element</td>
<td></td>
</tr>
<tr>
<td>SecondRec</td>
<td>Data type of this record element</td>
<td></td>
</tr>
</tbody>
</table>
5.3 Primitive Construction

The primitives are hardcoded and a tree structure has been created for each of the primitives. The names of the ports are input interactively by the user. This allows the primitives to be as generic as possible. For the purpose of hardcoding, each of these primitives is a process and the tree structure starting from the Module node is created. After the necessary values are input by the user, the oval representing the primitive is drawn. We shall consider an example of an oscillator, to demonstrate how a primitive can be hardcoded. The hardcoding for other primitives is done similarly.

osc1l1l : process (CLK, RUN)
begin
  if ((RUN = '1') and not (RUN'stable)) then
    CLK <= transport '1';
  if (RUN = '0') then
    CLK <= transport '0';
  end if;
  if ((CLK = '1') and not (CLK'stable)) then
    CLK <= transport '0' after CLK_PER1,
       '1' after CLK_PER2;
  end if;
end process OSCILL;

The first step in the creation of this primitive is the allocation of a Module node which shall be the root of our tree structure. Since the primitive needs two ports, we
need to allocate two port nodes. A port node is created and the node number is written into the TopPort field of the Module node. Since the port for which the port node has been created triggers the oscillator, it is an input port. During the creation of the port node, the user is asked to input the name of this port. After the user provides the name of the port, the port type, port name and the port mode are written into their respective fields. In the case of our example, the user has specified the name of the port as RUN. The port type which is Bit, the port mode which is in, and the port name RUN are written into the respective PortType, PortMode, and PortName fields. Since this port triggers the oscillator, the Sense bit is also set. This port node is added to the top of the list of port nodes, and the next available node is tagged as a free node. Similarly, the other port in our primitive is created. The user has specified the name of the second port as CLK and this name is written into the PortName field. Since this port is of inout mode and since this port also triggers the oscillator, the port mode is specified as inout and the sense bit is set. Since there are no variable declarations and type declarations, the TopVariable and TopType fields of the Module node are left unused. The next step is the creation of the statements. Since the first step is an if statement, an if node is created. This node number is added to the TopStmt field of the module. The expression within the if statement needs to be constructed and so an Expression node is created and is added to the IfExpr field of the if node. The expression within our if statement is:

\[(\text{RUN} = '1') \text{ and not (RUN}'\text{\textbackslash'Stable})\]

This expression is equivalent to a rising signal and so, the routine which creates a rising signal is invoked. The routine basically breaks up the above expression into two independent expressions namely:

- \((\text{RUN} = '1')\)
not (RUN'Stable)

A new set of expression nodes are created for each of these individual expressions and these expression nodes are added to the ExprLeft and the ExprRight nodes of our first Expression node. The expression type is Boolean, and the expression operator is and. The integer value corresponding to the and operator is added to the ExprOperator field and the string and is added to the ExprText field. The integer value corresponding to the Boolean type is added to the ExprType field. Each of the smaller expressions listed above are broken down and hardcoded.

The set of the statements within the then part of the if statement have to be encoded. Since this statement is an assignment statement, the Asgn node is created next and this node number is added to the ThenStmts field of the if node. The node number corresponding to the port CLK is added to the TargetPort field of the Asgn node. The value '1' needs to be assigned to this target port and hence a Clause node is created and this node number is added to the TopClause field of the Asgn node. A Value node is created and this node number is added to the ClauseExpr field of the Clause node. The text '1' is written in the ValueText field of the Value node. The delay type is transport and the integer value corresponding to the transport delay type is added to the ClauseDelayType field of the Clause node. If a delay value needs to be added to this assignment, then a generic node needs to be created and added to the TopGeneric field of the Module node. This node number would be added to the ClauseDelay field of the Clause node. Any other generic nodes created are added to the list of generics that has been created. Since there are no other statements in the then part and also the else part of the if node, a new if node is created and this is added to the NextStmt field of the first if node. This is
because the next statement after the first if statement is also an if statement. The expression within this if statement which is:

(RUN = '0')

and the statement in the then part of the if statement which is:

CLK <= transport '0';

are created similarly. The next statement after the second if statement is also an if statement and hence another if node is created. The expression within this if which is:

((CLK = '1') and not (CLK'Stable))

is created in an identical fashion to the first if statement. There are two assignment statements within the then part of this if statement. A Clause node is created and the first assignment statement is created as described above. The next assignment statement is an assignment to the same target port and hence in VHDL, it corresponds to the next Clause statement. Another Clause node is created for encoding this statement and this node number is added to the NextClause field of the first Clause node. The encoding of this statement does not differ from the other assignment statements discussed. No other statements need to be added and hence this concludes the construction of the Oscillator primitive. The only detail with which the user needs to be bothered with, is the name of the ports. The VHDL code for the primitive is available in a ready to use fashion and the user need not be bothered with the details of the creation of the primitive and the statements within the primitive. Other primitives have been hardcoded in a similar fashion and the necessary routines which aid in this process of hardcoding have been created.
5.4 Graphics Creation

Additional graphics capabilities have been added to the system. The routines which have been added are included in the file generics.c. These routines perform their graphics capabilities by using the basic DOS and ROM BIOS calls [26]. In this section the routines used for enhancing the graphics capabilities shall be briefly discussed.

The following are the main general purpose routines which can be used for creating and drawing boxes, placing a character on the screen, etc. Additional routines for making windows, highlighting menu items etc., are available in the file menu.c. Only the routines that have been used for the tool are included in the file generics.c.

- **Box** - This routine accepts as parameters, the top row number, the first column number, the width and height of the box, the foreground color, the background color in that order. Based on these parameters, it puts the characters for drawing the box on the screen and thus creates a box.

- **Pch** - This routine accepts as parameters, the character that one needs to place on the screen, the foreground color, the background color, and the number of characters that need to be placed on the screen. It should be noted that the ASCII values of the characters and not the actual characters are accepted as parameters. The ASCII values for the characters can be obtained from [26].

- **Get_vmode** - This returns the current video mode in the system register AL. Two other additional pieces of information can be obtained by the same interrupt call. The screen width in characters is returned in the register AH and the display
page number is returned in the register BH. The routine does not return these values, but can be suitably modified to return these values too, if they are needed.

- **Clear_screen** - This routine clears the screen by invoking the set_vmode routine.

- **Set_vmode** - This routine sets the video mode from the set of 15 standard video modes. A list of the possible video modes is given in [26].

- **Printc** - This routine accepts as parameters, the row number, the column number, the character string that needs to be placed on the screen, the foreground color, and the background color. It places the cursor at the required row and column number and then draws the character string on the screen.

- **Locate** - The locate routine accepts as parameters, the row and column numbers and places the cursor at the required position. Integer values for the row and column numbers are accepted.

- **Curser_remove** - This routine places the cursor at the bottom right corner of the screen. The coordinates of the bottom right corner are assumed to be (25,85); the first number is the row number and the second number is the column number.

Some very elementary routines for drawing boxes, drawing characters have been described above. Other routines which haven’t been used in the tool, but might be useful for future enhancements are available in the file menu.c.
Chapter 6. Performance and Evaluation

6.1 Node Size

The size of the node was initially configured to be 50 bytes. Due to the addition of two more pointers to the node structure, the size of the node has been increased to 54. The number of nodes have also been increased to 800.

6.2 Program Size and Disk Access Time

Benchmarking operations have been performed on the tool and these have been tabulated in Table 6. The times obtained have been obtained using an IBM PC/AT with a 20Mbyte hard disk and without a 8087 co-processor. The Controlled Counter discussed as Example 1. in Chapter 4. has been used for the benchmarking operation. The Counter primitive used in the Controlled Counter model has been used for the benchmarking operations on a primitive. This primitive has five ports. As shown in Table 6., the executable file size is 350KBytes and the program invocation time is 13 sec.
The time needed to save primitive processes and units to the disk and the time needed to generate VHDL to the disk have also been listed in the table.

Table 6.

<table>
<thead>
<tr>
<th>Condition</th>
<th>Value</th>
</tr>
</thead>
<tbody>
<tr>
<td>Program Size</td>
<td>Normal</td>
</tr>
<tr>
<td>Invoke Time</td>
<td></td>
</tr>
<tr>
<td>Save to Disk</td>
<td>Primitive</td>
</tr>
<tr>
<td></td>
<td>Unit</td>
</tr>
<tr>
<td>Load from Disk</td>
<td>Primitive</td>
</tr>
<tr>
<td></td>
<td>Unit</td>
</tr>
<tr>
<td>Generate VHDL to Disk</td>
<td>Primitive</td>
</tr>
<tr>
<td></td>
<td>Unit</td>
</tr>
<tr>
<td>Disk File Size</td>
<td>Primitive</td>
</tr>
<tr>
<td></td>
<td>(Counter.mod)</td>
</tr>
<tr>
<td></td>
<td>(Counter.vhd)</td>
</tr>
<tr>
<td></td>
<td>Unit</td>
</tr>
<tr>
<td></td>
<td>(Cctr.unt)</td>
</tr>
<tr>
<td></td>
<td>(Cctr.vhd)</td>
</tr>
</tbody>
</table>

Compared to the original Modeler’s Assistant, it can be observed from the values for a primitive, listed in Table 6, and the corresponding values listed for a process in Table 5-1 in [4], that there doesn’t exist much difference in the disk file size or the saving time to the disk, and the loading time from the disk.

6.3 Model Creation Time

Further benchmarking operations have been performed on the time that is needed for creating a model on the initial version of the Modeler’s Assistant and the version after adding primitives. Three models have been chosen for these benchmarking oper-
ations. These models are the Register model, Controlled Counter model, and the 8214 Priority Interrupt Control Unit model. The timings for the three cases are given in Table 7.

<table>
<thead>
<tr>
<th>Model</th>
<th>Time (without Primitives)</th>
<th>Time (with Primitives)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Register</td>
<td>5 min 10 secs</td>
<td>1 min 10 secs</td>
</tr>
<tr>
<td>Controlled Counter</td>
<td>26 mins</td>
<td>3 mins 45 secs</td>
</tr>
<tr>
<td>8214 PICU</td>
<td>72 mins</td>
<td>14 mins</td>
</tr>
</tbody>
</table>

It can be observed that in the case of the register there is a speed up of 4.13. In the case of the Controlled counter there is a speed up of 7 and in the case of the 8214 PICU, there is a speed up of approximately 5.15. It should be pointed out that the timings that have been obtained for the construction of the models in the initial version of the Modeler's Assistant have been slightly extrapolated. This is because the initial version of the Modeler's Assistant does not have all the features for constructing such complicated models. As the systems get more and more complicated, the speed up times should improve. But, in the case of the 8214 PICU the speed improvement factor has gone down. There is a logical reason for this, which shall become more apparent after the discussion that follows.
6.4 Primitive Set Completeness

Another factor that needs to be evaluated is the percentage of processes that have been represented by the primitives in the examples that we have looked at in Section 4.3. In the case of the Register all the 3 processes have been represented by primitives giving a coverage figure of 100% (Coverage figure is defined as the ratio of the number of processes represented by primitives to the number of processes in the model). A similar situation is evident in the Controlled Counter primitive, as all the 5 processes are represented by primitives giving a coverage figure of 100%. In the case of the 8214 PICU, 8 out of the 10 processes are represented by primitives. The coverage figure in this case is 80%. In the case of the UART, 6 of the 9 processes are represented by primitives and hence the coverage figure is 66.67%. In the case of the URISC, 13 out of the 17 processes are represented by primitives and hence the coverage figure is 76.5%. On the whole, computing an average of the above figures would give us an effective coverage figure of 79.55%. This implies that our list of primitives is not a complete one. It can be observed that as the complexity of the model increases, the need for specialized processes also goes up. Additional primitives which would enable us to model any system without the need for specialized processes need to be defined and this can be a future area of research.

The speed up in the case of 8214 PICU has decreased compared to that of the Controlled Counter. This is because, the times needed for creating the 2 specialized processes in the case of the PICU are the same in the case with and without the primitives. The time saved is the time that has been saved for creating the other 6 processes. If the coverage figure had been 100%, then the speed up for the PICU would have been much more than the figure that has been projected now.
6.5 VHDL Code Correctness

The models of the Register, Controlled Counter, 8214 PICU, and the UART that were created using the Modeler's Assistant, have been simulated using the MCC VHDL System 1.1 and they have been found to be working perfectly.

6.6 Summary

The use of the process primitives

1. Covers an average of 79% of processes required for a behavioral model

2. Speeds up model development by a factor of five

3. Has no degradatory effect on disk file size and disk saving and loading time

4. Produces models which simulate correctly.
Chapter 7. Suggested Future Enhancements

A few future enhancements to the tool are discussed below. Other enhancements to the tool have been discussed in [4].

- Type definitions

The concept of a type definition has been implemented through its use in the primitives. The necessary node structure for a type definition has been created. The option by which the user can create the type hasn't been implemented in the tool. This could be achieved by creating prompt messages and storing the name of the type, the type it was derived from, the ranges of the elements and such other values in the linked list structure. The new type definition can either be placed in a package, or can be included in the architectural declarative part.

- Port modes
Presently, only the port modes in, out, and inout have been implemented. A possible improvement would be to implement the other port modes; namely buffer, linkage, and open. This would obviously mean that the process model graph notation would have to be modified to represent these port modes.

- User-defined primitives

The user could be given the option to create his own primitive and use it. Two things need to be accomplished to implement this feature. Firstly, the user should be able to create the code that makes up the primitive. Next, he should be able to include the name of this primitive in the menu, listing all the primitives in the tool. An option to delete the user-defined primitive from the tool also needs to be provided. Let us consider the aspect of creating the code that makes up a primitive. This is fairly simple and straightforward and this option already exists in the tool. All primitives are treated like processes. So, if one needs to create a primitive, then all that one needs to do would be to create a process and use it like a primitive. The code within the primitive is developed in an identical fashion to the statements that are added to a process.

The option of including the name of a primitive in the list of primitives poses a slight problem. In the present implementation of the tool, all the menu items that need to appear on the screen are stored in arrays and the name is written onto the screen whenever needed, after reading the contents of the array. So, if the name of a primitive has to appear on the screen, then the name needs to be written to the array structure and the program needs to be recompiled for the name to appear on the screen. This provides a permanent solution to the problem. Another option is
to place the character string which refers to the name of the primitive on the screen in a similar fashion to the way the name of a port is written on the screen. The name needs to be written under the present list of primitives. The user should be able to place the name of the primitive wherever he wants and he should be able to move it around as he pleases. This situation is analogous to adding a port and moving it around on the screen and hence should be easy to accomplish. When a user needs to invoke a primitive, he clicks on the string and the program should be able to call up the particular primitive depending on the coordinates of the point where the user clicks. This obviously implies that a structure needs to be created for storing the name of the new primitive and also it's coordinates on the screen. This provides a simple solution to include a user-created primitive in the menu. Another point which needs to be noted is that the structure corresponding to the menu item needs to be saved since, it can be lost when the user shuffles between menus. However, the above solution is a temporary one.

- Additional comments

The tool lacks in a few aspects. The tool does not enable the user to pick a slice of a signal. Consider the following situation. If a process has a port and another process has a port with the same name, then the tool lists the port twice in the entity port list of the model. A similar problem exists in the case of multiple signals and generics with the same name. This is another problem which needs to be fixed.
Bibliography


Appendix A. User's Manual

System Requirements

The Modeler's Assistant runs on an IBM PC/XT/AT compatible. The requirements for the tool include CGA graphics capability, at least 512Kb of memory, and IBM DOS version 3.1 and up. A Microsoft or Mouse Systems mouse attachment is necessary and a HP LaserJet or IBM Graphics printer for hardcopy support of screen dumps. Other requirements for the tool are listed in the User's Manual section of [4].

Command Descriptions

A brief description of the commands available with the tool has been attached. If the user needs to select a particular menu item, then he needs to place the cursor over that particular menu item and click using the left mouse button.
Main

From this menu, the user can either select *Create* or *Edit*. If the user chooses *Create*, then he can either create a unit, or a process. If he chooses to create a unit using the available primitives, then he clicks on the *Create* option. The user needs to enter the name of the unit next. He is then taken to the *Add* menu.

Add

The user is presently editing an unit and he has the option of adding a *Process*, *Signal*, or a *Primitive*. The menu items *Process* and *Signal* have already been discussed in [5]. The menu item *Primitive* shall be discussed here. This menu item is primarily for choosing a primitive from the list of primitives and adding it to the unit.

Primitive

The following options are available when choosing a Primitive.

- Oscillator - This option is used to add an Oscillator to the Unit. The name of the port which triggers the oscillator and the name of the port which function as the clock needs to be provided by the user. The name is limited to 32 characters. The oscillator keeps generating the clock output continuously until it is stopped by disabling the port which triggers the oscillator. Upon entering the names of the ports, the user has to place the oval representing the Oscillator on the screen.
• Counter - The procedure for adding the Counter primitive is similar to that of an Oscillator. The user provides the names for the ports and also the counter type. The counter can be utmost a 32 bit counter. The functions INCREMENT and DECREMENT are provided in the package VHDLCad that is provided to the user. The package is also listed in the last section of this User’s Manual. The Counter can be a up-Counter or a down-Counter depending on the value of the port which indicates this nature.

• RAM - The names of the ports and also the size of the memory is provided by the user. The functions used for type conversions from Bit_Vector to Integer and vice versa are provided along with the VHDLCad package.

• Truth Table - The Truth Table primitive can have a maximum of 4 inputs and a maximum of 4 outputs. Upon clicking on this menu item, a table is drawn on the screen and the user needs to provide the input and output values. Any duplicated table entry is taken care of, by informing the user that the corresponding row has been duplicated. The user is prompted to re-enter that particular row. The values provided by the user are encoded in the form of an aggregate and this aggregate is used for the signal assignment statements.

• State Table - The State table can have a maximum of 16 states and 4 inputs. Type declarations for the states in the State table need to be included in the standard package, by the user. This declaration is analogous to the enumerated type declaration for the inputs that is given in the primitive. This declaration has not been included with the aggregate because of visibility problems. The values of the state table are also included in an aggregate and the signal assignment statements read the values from this aggregate when needed.
- **SR Latch** - This primitive gives the VHDL code for a SR flip flop. The names of the ports need to be provided by the user.

- **D flip flop** - The procedure for adding this primitive is just like any other. The names for the D input, Clock, Set, Reset, Q, and QBar need to be provided by the user.

- **Parity** - This menu item is for adding a Parity Checking circuit primitive. The names of the ports need to be provided and also the size of the input bus. A maximum size of 32 is allowed. A generic delay value is associated with the value of the parity that is passed on to the output.

- **Comparator** - This menu item is for choosing a comparator primitive. Two input buses of maximum size 32 can be compared. The output ports available are those for indicating that the first input is less than the second, the first is more than the second, and that the first input is equal to the second.

- **Decoder** - This menu item chooses a Decoder primitive. The number of select inputs that the decoder can have is utmost 5, which gives us a maximum of 32 outputs to choose from. The names of the ports need to be provided by the user.

- **MUX** - This menu item chooses a Multiplexor primitive. The multiplexor can have a maximum of 32 inputs.

- **Adder** - This menu item chooses an Adder primitive and can be used for adding two input buses, which can have a maximum size of 32. The names for the inputs, and the Sum and the Carry outputs need to be provided by the user.
• Ppn Counter - This menu item chooses a Population Counter primitive. The Population Counter can have a maximum of 32 inputs and the output gives the number of ones in the input bus.

• Par to Ser - Used to add a Parallel to Serial converter primitive. The user needs to provide the names of the ports. The maximum input bus size can be 32.

• Buffer - Used to add a Buffer primitive. The names of the enable input, the input bus, the output bus and the sizes of the buses also needs to be provided. Information whether the Buffer gets enabled on the rising or falling edge of the enable input is also needed. The maximum buffer size can be 32.

• Ser to Par - This menu item is used to choose a Serial to Parallel converter. The maximum output bus size can be 32. This primitive is analogous to the Par to Ser primitive.

• Shift Reg - This menu item chooses a Shift Register whose maximum size can be 32. Information regarding the names of the ports needs to be provided by the user.

• Incrementor - This item chooses an Incrementor primitive and is analogous to the Adder primitive, except that one of the inputs of the adder is held to 1. The maximum input bus size can be 32.

• Prt Encoder - This menu item chooses a Priority Encoder whose maximum input bus size can be 32 and the maximum output bus size can be 5. The names of the ports need to be provided by the user. Generic delay values for the assignments are assumed and are included in the generic list.
• **Register** - This item chooses a register primitive. The user needs to provide the port names, the generic delay values for the assignments, and the size of the input bus. The information whether the register gets triggered on the rising or falling edge of the enable input and if the input bus is a tri-state bus also needs to be provided. The maximum input and output bus sizes can be 32.

• **Latch** - This menu item chooses a Latch primitive. Basically, the same information that is needed for a register primitive is also needed for a latch primitive. The limitations on the sizes of the buses is the same as that for the register. But here, the information whether the latch gets triggered when the enable input is high or low is needed. Upon providing the information the oval representing the primitive and the smaller circles representing the ports are drawn on the screen. The enable port is represented by a shaded circle since it enables the latch.

After choosing the particular primitive, the user is taken to a new set of menu items. The new set of menu items are `VHDLDump`, `Save`, and `Display`. The user needs to save the primitive next, and hence he needs to click on the `Save` menu item. After the primitive has been saved, it can be called any number of times, since it is treated like a process. The user also has an option of taking a hardcopy of the code for the primitive and for this he needs to click on the `VHDLDump` menu item. If the user needs to zoom in, zoom out, pan or fit everything on the screen, then he needs to click on the `Display` menu item and choose the particular option.
Sample Session

The Modeler's Assistant can be invoked by typing the following at the prompt:

A > vhdlcad

This sample session shall illustrate the process of creating the model for the Controlled Counter. The various menus and screens as one goes through the development process have been illustrated in the pages that follow. On invoking the program the screen shall look like the one in Fig. 22. Since the user needs to create a unit by the name cctr, he clicks on the Create, Unit, menu items successively and then he enters the name of the unit which is cctr in our case. This is illustrated in the Figs. 22 and 23.
Figure 22. Initial Screen
Figure 23. Entering Unit Name: CCTR
After providing the name of the unit, the user is taken to the Unit menu. He next needs to add the primitives one after another. The user first picks the Add menu item out of the Unit menu and this is illustrated in Fig. 24. He next needs to choose the Primitive menu item from the Add menu and this is illustrated in Fig. 25.
Figure 24. Unit Menu
Figure 25. Choosing the Primitive Menu Item from Add Menu
The next step is to choose one of the primitives out of the *Primitive* menu and in our case we first choose the *Latch* menu item, since we wish to add the Latch primitive first. This is illustrated in Fig. 26. Next, the user is taken through a sequence of menus whereby, he enters the name for the Latch primitive that the user wishes to give. He next enters the size of the Input bus, and indicates if the bus is a tri-state bus. These are illustrated in Figs. 27 and 28. He then enters the name for the Enable port, the input bus, the output bus, the generic name for the latch delay, and finally the user tells the tool if the latch is triggered when the enable input is high or low. All these menus are illustrated in the Figs. 29 through 33. The next step involves saving the primitive that has been created and this is illustrated in the Fig. 34. The latch is saved in a file Latch.mod.
Figure 26. Entering the Name for the Primitive
Figure 27. Indicating the Size of the Input Bus
Figure 28. Indicating if the Bus is a Tri-State Bus
Figure 29. Entering Name for Enable
Figure 30. Entering Name for Input Bus
Figure 31. Entering Name for Output Bus
Figure 32. Entering the Name for the Generic Delay
Is the Latch triggered when Enable is High/Low (H/L)?

Figure 33. Indicating if the Latch is Triggered when Enable is High or Low
Figure 34. Saving the Primitive
The user next backs up and then chooses the next primitive that he wishes to add. In our case, this is the Comparator and the various menus whereby the user inputs the names of the ports have been skipped, since the process is analogous to the process of entering the names for the ports in the Latch primitive. Fig. 35 shows that a Comparator primitive is being chosen and Fig. 36 shows that the primitive is being saved.
Figure 36. Saving Comparator Primitive
The Counter primitive is chosen next. The process of saving the Counter primitive is illustrated in Fig. 37. The intermediate menus where the names of the ports and the sizes of the buses are indicated have been skipped, but these are similar to the menus that we have gone through for building the Latch primitive. The Decoder primitive is chosen next and after providing the names of the ports etc., the primitive is saved. This is illustrated in Fig. 38. The process of choosing the Oscillator primitive and saving it is shown in Fig. 39.
Figure 37. Saving Counter Primitive
Figure 39. Saving Oscillator Primitive
Next, we back out of the *Primitive* menu. Since the primitives need to be interconnected, the user needs to click on the *Signal* menu item in the *Add* menu. This is illustrated in Fig. 40.

The source process, the source port, the destination process, the destination port and the name of the port which is *OUT* by default are chosen next. Fig. 41 shows that the user is being prompted for the name of the signal and Fig. 42 shows that a signal *OUT* has been added.

The next stage of the development process is the addition of the other signals. The stages whereby the user is prompted for the source and destination processes and the ports in each of them have not been illustrated. Figs. 43 through 48 show the signals EN, UPD, CLR, CLK, EN2, and IN2 being added. Figs. 49 through 52 illustrate the process of backing up of the various menus and quitting the program.

All the primitives are saved to files with the extension *.mod* and the units to the files with the extension *.unt*. The VHDL menu item in the *Unit* menu provides a hardcopy for the model, by dumping the code to the printer. These files are generated with the extension *.vhd*. The above sequence of menus shows how a behavioral model can be created out of the various primitives.
Figure 40. Adding a Signal
Figure 41. Prompt for Signal Name
Figure 42. Signal OUT has been added.
Figure 43. Adding the Signal EN
Figure 44. Adding the Signal UPD
Figure 46. Adding the Signal CLK
Figure 47. Adding the Signal EN2
Figure 48. Adding the Signal IN2
Figure 49. Backing Out of the ADD Menu
Figure 50. Backing Out of the UNIT Menu
Figure 52. Quitting the Program
VHDLCad Package

The following declarations in the package given below need to be appended to the declarations listed in the VHDLCAD package in [4]. The complete package is referenced by each file generated by the Modeler’s Assistant.
package VHDLCAD is

  type STATE is (A,B,C,D,E,F,G,H,I,J,K,L,M,N,O,P);
  type INPUT is (I0,I1,I2,I3);

  function INCREMENT(INPUT : BIT_VECTOR) return BIT_VECTOR;
  function DECREMENT(INPUT : BIT_VECTOR) return BIT_VECTOR;
  function INTVAL(INPUT : BIT_VECTOR) return INTEGER;
  function INT2BIN(INPUT : INTEGER) return BIT_VECTOR;
  function BIT2MVL_VEC(INPUT : BIT_VECTOR) return MVL_VECTOR;
  function MVL2BIT_VEC(INPUT : MVL_VECTOR) return BIT_VECTOR;

end VHDLCAD;

package body VHDLCAD is

  function INCREMENT(INPUT : BIT_VECTOR) return BIT_VECTOR is
    variable TEMP : Integer := 0;
  begin
    TEMP := INTVAL(INPUT);
    TEMP := ( TEMP + 1 ) rem (2**8);
    return INT2BIN(TEMP);
  end INCREMENT;

  function DECREMENT(INPUT : BIT_VECTOR) return BIT_VECTOR is
    variable TEMP : Integer := 0;
  begin
TEMP := INTVAL(INPUT);
TEMP := ( TEMP -1 ) rem (2**8);
return INT2BIN(TEMP);
end DECREMENT;

function INTVAL(INPUT : BIT_VECTOR) return INTEGER is
variable SUM : Integer := 0;
begin
for N in INPUT'Low to INPUT'High loop
  if INPUT(N) = '1' then
    SUM := SUM + (2**N);  
  end if;
end loop;
return SUM;
end INTVAL;

function INT2BIN(INPUT : INTEGER) return BIT_VECTOR is
variable TEMP_A : INTEGER := 0;
variable TEMP_B : INTEGER := 0;
variable OUT_DATA : BIT_VECTOR;
begin
 TEMP_A := INPUT;
for I in 7 downto 0 loop
  TEMP_B := TEMP_A/(2**I);
  TEMP_A := TEMP_A rem (2**I);
  if (TEMP_B = 1) then
OUT_DATA(I) := '1';
else
  OUT_DATA(I) := '0';
end if;
end loop;
return OUT_DATA;
end INT2BIN;

function BIT2MVL_VEC(INPUT : BIT_VECTOR) return MVL_VECTOR is
  variable TEMP : MVL_VECTOR;
begin
  for I in INPUT'Low to INPUT'High loop
    TEMP(I) := BIT2MVL(INPUT(I));
  end loop;
  return TEMP;
end BIT2MVL_VEC;

function MVL2BIT_VEC(INPUT : MVL_VECTOR) return BIT_VECTOR is
  variable TEMP : BIT_VECTOR;
begin
  for I in INPUT'Low to INPUT'High loop
    TEMP(I) := MVL2BIT(INPUT(I));
  end loop;
  return TEMP;
end MVL2BIT_VEC;
end VHDLCAD;
Vita

Chakravarthy S. Kosaraju was born March 29, 1967 in Hyderabad, India. He graduated with honors from Little Flower Junior College, Hyderabad, and was chosen for the A.P. Merit Scholarship. Chakravarthy entered Birla Institute of Technology & Science (BITS), Pilani in August 1984. He graduated with a Bachelor of Engineering degree in Electrical and Electronics Engineering (with Honors) in June 1988. He then went on to pursue a Master's degree in Electrical Engineering at Virginia Polytechnic Institute and State University (Virginia Tech) in August 1988. Chakravarthy plans to continue research in the areas of VLSI and Design Automation after he graduates in June 1990.

Chakravarthy worked as a programmer trainee at Sonata, Bangalore from January to June 1988. He plans to continue his research activities at Intel, Santa Clara where he shall be working from July 1990. He is a member of the IEEE.

[Signature]

Vita