APPLICATION NOTES: A VHDL BASED DoD PROGRAM

Raytheon

Michael Hajj, Wayne Jessop, Chris McCoy
1001 Boston Post Road
Marlborough, MA 01752
(508) 490-3619, -3535, -3593
(508) 490-3434 fax

ABSTRACT:

Equipment Division of Raytheon Company has recently completed the design of a fault tolerant, tightly coupled multiprocessing computer system for an aerospace application. The system includes 11 modules consisting of 8 ASICs, 17 FPGAs, commercial off the shelf complex components, and off the shelf MSI and SSI devices. VHDL was used as the primary vehicle for modeling, simulation, and validation that has been integrated into Raytheon's existing design environment. This paper emphasizes the lessons learned in the process of developing and integrating this system. Topics to be covered include: migration to VHDL, design implementation, VHDL coding techniques, plus functional and timing verification tips through the use of hardware accelerators.

1.0 INTRODUCTION

A major DoD program was designed and developed using VHDL as the backbone of the design methodology. The program was a blend of several different types of digital designs including: ASICs, FPGAs, commercial off the shelf complex components, and off the shelf MSI and SSI devices. This heterogeneous mixture of designs required a flexible CAE environment embedded in a toptown design methodology with bottom up verification using VHDL as the vehicle for modeling and simulation. Additionally, this environment provided the capability to support several minutes of real time subsystem simulation. This accomplishment was largely due to the initial efforts that were invested to pathfind and evaluate CAE tool options and development of a concrete design methodology for all facets of system design [1].

2.0 RAYTHEON's TOOL SUITE

In order to meet the aggressive design and performance goals of the program, Raytheon adopted three major VHDL tool components (Figure 1). The front end component to this environment is a VHDL simulator residing on a SUN workstation platform. The middle component is a synthesis tool for gate level generation that also resides on a SUN workstation platform. The back end component is a hardware accelerator for both RTL and gate level simulation and verification. A fourth component, ASIC emulation, was integrated into the tool suite as a final level of verification. This last component, will not be discussed in this paper. After careful evaluation of VHDL based CAE tools, the IKOS Voyager Series was chosen for VHDL software simulation and gate level accelerated simulation. For synthesis, Racal-Redac SilcSyn was chosen. For further details pertaining to the selected
tool suite (including the emulator platform) or the adopted design methodology, the reader is referred to [2].

3.0 MIGRATION TO VHDL

Since very few designers knew VHDL and its application toward an effective system to ASIC design implementation, it became apparent that experienced individuals were needed to guide the team in its initial use of VHDL. Therefore, a VHDL consulting group was hired to fulfill this role. After being exposed to the chosen VHDL based methodology and Raytheon's internal CAE design environment, the group worked closely with the lead engineers and VHDL experienced engineers to perform the following:

1) Supply VHDL training. This includes tutorial sessions and hands-on work.
2) Support the design team through the duration of the project, in the following areas:
   A- Behavioral VHDL modeling for test bench generation.
   B- RTL VHDL modeling for ASIC Design development.
3) Generate modeling guidelines.

3.1 Training

Proper training plays a crucial role in adopting VHDL for complex digital computer system design and development. Likewise, establishing a self-motivated team that is willing to adjust and absorb a new design paradigm plays a significant role in the success of migrating to a top-down VHDL methodology. A team of more than forty five designers was trained in VHDL. The training covered the VHDL essentials with a concentration on developing VHDL testbenches and the effective use of std_developersKit packages by the VHDL Technology Group. Figure 2 illustrates the background of the team, its previous experience in both hardware and software disciplines, and the amount of time it took to become comfortable with VHDL. Once VHDL training was established, designers underwent another set of training sessions to learn how to operate the tools. The tools training was arranged with the vendors ahead of time. A defined plan was drawn with them to focus on the how's to operate the tool and not on the why's their tool is better. Overall each designer underwent an average of seven days of training both in VHDL and tools. Special attention was
<table>
<thead>
<tr>
<th>Designers</th>
<th>VHDL Experience</th>
</tr>
</thead>
<tbody>
<tr>
<td>18</td>
<td>No VHDL Experience</td>
</tr>
<tr>
<td>2</td>
<td>VHDL Experience through work</td>
</tr>
<tr>
<td>1</td>
<td>VHDL Experience through academics</td>
</tr>
<tr>
<td>2</td>
<td>VHDL Experience through work/academics</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Designers</th>
<th>Background</th>
</tr>
</thead>
<tbody>
<tr>
<td>11</td>
<td>Module Design</td>
</tr>
<tr>
<td>3</td>
<td>Software Development</td>
</tr>
<tr>
<td>1</td>
<td>ASIC Design</td>
</tr>
<tr>
<td>3</td>
<td>Module/Software Development</td>
</tr>
<tr>
<td>3</td>
<td>Module/ASIC Design</td>
</tr>
<tr>
<td>1</td>
<td>Module Design/Software Development/ASIC Design</td>
</tr>
<tr>
<td>1</td>
<td>Module Design/Software Development/FPGA Design</td>
</tr>
<tr>
<td>23</td>
<td>Total (23 out of 51 designers responded to our survey)</td>
</tr>
</tbody>
</table>

Figure 2 - Teams Design Background and VHDL Experience

given to the VHDL subsets supported by the SilicSyn synthesis tool.

3.2 Support

The support was organized to provide both on-site and phone assistance. On-site support was planned for the first two months followed by phone support for the duration of the project. However, our experience showed that designers made use of the on-site support and minimal usage of the phone support.

3.3 Guidelines

The consulting group was also responsible in deriving a set of modeling guidelines. The aim of these guidelines was to: acclimate the design team to digital logic modeling using VHDL, to foster consistent and uniform style, to preserve model compatibility, and to properly manage VHDL databases. These guidelines helped facilitate the writing and reading of VHDL models (especially during code review) and improved simulation performance. Experience showed that these guidelines had to be a living document, evolving as designers advanced both in VHDL modeling and tool usage. Since our methodology covered a wide set of VHDL based tools, more detailed sets of guidelines were later developed documenting the following:

A) Examples of RTL VHDL code for both combinatorial and sequential digital circuits. Good design practice requires familiarity with the synthesis tool. Having good examples of VHDL code and their equivalent gate level implementation is extremely helpful to understand the synthesis behavior and its interpretation of different VHDL subsets.

B) Techniques in generating efficient VHDL RTL level code to produce optimum gate level implementation.

C) Tips and rules to perform accelerated VHDL RTL simulation.

D) Instructions to perform mixed level simulations.

A subset of the styles and modeling techniques are discussed in the following paragraphs.

3.3.1 General VHDL Modeling Guidelines

Because VHDL is a powerful description language, two important goals must be satisfied when writing the source code for a given project: it must be easy to understand and it must be modifiable. Easy-to-understand code benefits anyone who must read the code, especially if the original designer is not available to clarify any ambiguity. Modifiable code is achieved by maintaining a uniform coding style. This implies good coding practice, such as consistent indentation and informative
comments. In fact, liberal use of comments throughout the code listings is highly recommended.
The following are applicable rules to implement uniform, readable source code.

3.3.1.1 Code Structure

The overall appearance of the code determines the overall readability. For that reason, reserved words and punctuation (e.g. semi-colons) should be lined up on adjacent lines of code. Indentation should be used to show nesting and subordination.

3.3.1.2 Concurrent Statement Labels

To enhance traceability throughout the design, particularly when using the IKOS Voyager system for gate level simulation, the optional label (identifier) should be used for processes, concurrent signal assignment statements, concurrent procedure calls, etc. A descriptive name should be chosen to easily identify the source.

3.3.1.3 Code Hierarchy

The internal model should be structured similar to the real hardware functional block diagram. This is a true aid when cross referencing the code with large diagrams.

3.3.1.4 Informative Data From Specifications

All relevant information from the written specifications should be documented in the comments wherever feasible. Examples are block diagrams, truth tables, signal waveforms, and existing models. Pertinent section and subsection numbers found in the specs should also be included in the comments for effective traceability from the VHDL code to the specification.

3.3.1.5 Requirement Traceability

Working on a Department of Defense (DoD) program requires particular attention to the decomposition of requirements from the customer generated specifications to the actual implementation of the lowest level design. A requirement is defined in a specification via a SHALL statement, as shown in Figure 3. These shalls can in turn result in lower level SHALLs in subsequent lower level specifications. In the example shown, the ASIC specification calls for an 8-bit ALU which must be capable of performing two's complement arithmetic. The VHDL model must reflect these requirements in both the code and comments. In order to differentiate that a hard requirement is being implemented, the actual verbiage from the spec, surrounded by a line of hash marks (#), is placed just prior to the code which implements it. The SHALL itself contains the paragraph number and its own number from the written document. Both of these items allow the user to quickly locate a particular requirement.
a) ASIC specification

3.1.3.5.1.5 Error Handling

In the event of an error, the ALU must ensure that the conditional flags are not asserted and the data table is not corrupted. There are multiple error sources for the ALU. The first error source is associated with memory data. If a parity error is detected, the ALU SHALL[1] clear the conditional flags, set the processing error flag bit, and clear the 8-bit data word. Furthermore, if the parity error is generated by memory bank 2 then a chip shutdown SHALL[2] occur.

b) VHDL code

```
-- Is this an ALU operation? --

if (ALU_SEL='1') then
  -- If a parity error is detected, the ALU SHALL[1] clear the conditional flags, set the processing error flag bit, and clear the 8-bit data word.
  if (PARITY_ERROR='1') then
    EQZ <= '0';  -- Clear the Equal to Zero Flag
    LTZ <= '0';  -- Clear the Less Than Zero Flag
    GTZ <= '0';  -- Clear the Greater Than Zero Flag
    PROCERR <= '1';  -- Set the Process Error Flag
    DATA <= (others=>'0');  -- Clear the Data Word
  end if;
else
  .......
```

c) Output Response File

<table>
<thead>
<tr>
<th>SHALL</th>
<th>Description</th>
<th>Test Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.2.6.1[1]</td>
<td>Memory Parity Error</td>
<td>Passed</td>
</tr>
<tr>
<td>3.2.6.1[2]</td>
<td>Memory Bank 2 Error</td>
<td>Passed</td>
</tr>
</tbody>
</table>

Figure 3 - Requirement Traceability

The testbench must similarly be written to include the verbiage and SHALL number. Additionally, the testbench must signify when a SHALL is being tested and the outcome of the test. Figure 3 illustrates this in an output response file. Therefore, full traceability is complete. One can locate how it is implemented in the VHDL code, how it is tested in the appropriate testbench, and verify how it functioned in the output response file through a simple search or find command.

3.3.2 VHDL Synthesis Guidelines

All the ASIC VHDL models were partitioned into two main blocks as illustrated in Figure 4. One block is referred to as the core of the ASIC, which is the RTL VHDL model of the ASIC, and the second block is the top level where the core and the I/O buffers are instantiated. This block is referred to as the shell.
does). This can detect problems in the design of the ASIC, allowing the designer to correct them at an early stage. Figure 4 also illustrates the hierarchy of a given ASIC. VHDL models were implemented for the I/O buffers to validate the simulation. Figure 5 illustrates the modeling of the buffers, which was done such that the interconnects from the I/O buffers to the core are defined as std_ulogic, whereas the interconnects from the I/O buffers to the ports of the shell are defined as std_logic. For the bi-directional buffers, a signal must also be provided from the core model to control the direction (transmitting or receiving) of the pad.

3.3.2.1 Signal Polarity

All inner signals within the ASIC VHDL core were defined as active low. Outer ports/signals were defined as active high to avoid glitches and other board layout parasitic effects.

3.3.2.2 Logic Types

The use of the std_ulogic type in the core of the ASIC model was encouraged. Std_ulogic does not carry a resolution function that defines how values of multiple sources of a given signal are to be resolved into a single value (std_logic does). This can detect problems in the design of the ASIC, allowing the designer to correct them at an early stage. Figure 4 also illustrates the hierarchy of a given ASIC. VHDL models were implemented for the I/O buffers to validate the simulation. Figure 5 illustrates the modeling of the buffers, which was done such that the interconnects from the I/O buffers to the core are defined as std_ulogic, whereas the interconnects from the I/O buffers to the ports of the shell are defined as std_logic. For the bi-directional buffers, a signal must also be provided from the core model to control the direction (transmitting or receiving) of the pad.

3.3.2.3 Handling Asynchronous Inputs

All asynchronous inputs coming into the sequential logic of the ASIC design that get synchronized to the internal clock were double buffered to avoid metastability. These inputs are considered asynchronous functional inputs. Other asynchronous
inputs are considered purely asynchronous since they come into the combinatorial logic of the design and have no connection with any clocked signals.

3.3.2.4 Signal initialization

VHDL permits the initialization of signals when declared in an ARCHITECTURE. These signals are then forced to start with a specified logic value during the software simulation. At the beginning of the simulation, before the first simulation cycle, there is an initialization step which consists of initializing each signal or generic parameter with a value according to a set of rules prescribed in the IEEE 1076 VHDL Language Reference Manual (LRM). Then all processes are executed once. VHDL signals are driven by potentially many different sources (either processes or component ports) at the same time. The value of a signal at any point in time is calculated by arbitrating between all of the driving values for that signal and choosing an appropriate value. A source in VHDL contributes to a signal's value. The key to correctly initializing a VHDL signal is to correctly initialize all of its sources [3]. VHDL provides an explicit mechanism for specifying the default value for a signal which is the 'left of the signal's type. For example, the IEEE standard 1164 has the 'U' as the 'left value. Therefore for signals that have no source, the value defaults to 'U'. This value may be overwritten during the declaration part of signals where the user can define an initialization default value. For example, the following declaration will cause all process driver sources of the signal 'flag' to be set to '0':

SIGNAL flag : std_ulogic := '0';

Since the SilcSyn synthesizer ignores signal initializations, conflicts can occur between the software simulation and the gate level simulation. Gate level simulators emulate real hardware implementations including the electrical characteristics of the circuits. Initialization of actual circuits is done through the reset function of the design. Therefore, signal initializations in the VHDL architecture of the ASIC design were discouraged. Designers were instructed to initialize the desired signals in the reset implementation function of the design. If a designer decided not to implement the initialization of the desired signals in the reset function, then the VHDL testbench/test vectors were used to propagate the desired logic value to the selected signals.

3.3.2.5 Edge Detector

Since the program's ASIC designs were based on an edge-sensitive, single phase clocking scheme, only one frequency clock signal input is supplied to a given ASIC. This design scheme, termed Static Synchronous design [4], defines that all storage elements are sensitive to rising edge of a common clock signal. The clock period is longer than the settling time of the circuit. If the clock was to stop, the system would stay in its current state as long as power is maintained, hence the classification as static.

The SilcSyn guidelines require the assignment of the CLOCK_SOURCE attribute during the declaration of the clock signal and the use of the wait statement in a process to infer storage elements. The CLOCK_SOURCE attribute is used to identify the clock for optimization, timing analysis, and fanout compensation purposes. Therefore, the SilcSyn synthesis tool tries to build an efficient clock tree based on the usage of the clock signal within the design. However, suppose an ASIC design has a second clock source that has a low frequency or is used in a limited fashion within the design. Listing the clock signal in the sensitivity list or the WAIT statement of a clocked PROCESS will force the synthesizer to try building another clock tree for that clock. Figure 6a illustrates a small circuit (rising edge detector) that can help circumvent this problem. The edge detector circuit allows a second clock source in the ASIC design without being listed in the WAIT statement of the PROCESS. The circuit also encourages synchronous design. The pulse generated from the circuitry is exactly one clock pulse cycle long. The decoded output is used as the new clock signal. A VHDL
same state, the variable is implemented as combinational logic. If the variable is assigned in a different state from which it is used, it will be implemented as a flip-flop. However, if the variable is assigned and read within the same state and also read in a different state, SilcSyn builds two branches for this variable. One is a combinational branch and the other is a clocked branch, muxed together. Experience shows that two problems can arise due to this implementation. First, the number of gates noticeably increases in the implementation of state machines, and, secondly, these dual paths confuse the static timing analyzer into reporting false timing paths. Therefore, the use of variables was limited to the second case.

3.3.3 VHDL Acceleration Guidelines

Since all ASICs and FPGAs were to be synthesized using Racal-Redac's SilcSyn, the models had to strictly adhere to Racal-Redac SilcSyn's modeling guidelines. However, these guidelines are broad enough that two problems can surface: 1) Different VHDL representations may result in functionally equivalent yet structurally different designs, and 2) The code may behave differently in gates than in the software world.

What is RTL accelerated simulation?
The IKOS Voyager environment has introduced a unique VHDL simulation feature known as RTL accelerated simulation. IKOS has integrated SilcSyn's architectural synthesis, referred to as the Architectural compiler or Arccom, into its environment. By using Arccom, the RTL VHDL code is mapped into RTL primitives and netlisted together to form an RTL structural model residing on the IKOS accelerator platform. An example of an RTL structure is illustrated in Figure 7.
Even though the VHDL model is a structural netlist of IKOS primitives, the VHDL representation is preserved on the software platform, allowing users to transparently perform source level debugging.

Benefits of RTL acceleration
This integration offered several unique capabilities in the development of ASICs and in accelerating system level simulation.

1- Performing RTL accelerated simulation (hereafter referred to as RTL acceleration) shortened the iterative cycle of verifying the gate level implementation with its behavioral counterpart. Not only is the Arccom cycle considerably shorter then the gate level synthesis process, but RTL acceleration qualifies VHDL code to hardware structure. Discrepancies found during RTL acceleration will most likely be exhibited at the gate level implementation as well. To eliminate these problems, designers can perform source level debug or examine the RTL level diagrams generated during architectural synthesis. Thus, considerable time and effort was saved. As a result, RTL acceleration was mandated for all ASIC and FPGA designs prior to actual gate level synthesis.

2- RTL accelerated simulation is two to four times faster than gate level simulation. This speed improvement enabled successfull application of the adopted top-down, bottom-up methodology which calls for ASIC/FPGA validation at each design level:

A- ASIC/FPGA stand alone validation at the gate level.

B- ASIC/FPGA validation in the module level VHDL model (the ASIC/FPGA at the RTL level).

C- ASIC/FPGA validation in the subsystem level VHDL model (the ASIC/FPGA at the RTL level residing on the module VHDL model).

This verification is done by using the same testbench throughout the validations levels. The testbenches are modeled at the behavioral level making extensive use of the Std_developersKit® packages. These testbenches are used to test the functional requirements of the ASICs as identified in the specifications both before and after the synthesis process. IKOS provided the
capability of preserving the VHDL testbenches throughout the ASIC validation phases. As such, these testbenches exercise the ASIC as a VHDL model as well as its equivalent gate netlist. Engineers generating the testbenches for subsystem and ASIC levels are well acquainted with the circuit's behavior and use that knowledge to generate high-quality test vectors.

1- Users were cautioned that in accelerated simulation, the designer is no longer simulating software models.

2- Before attempting to accelerate the complete design, individual blocks of the design hierarchy should be RTL accelerated first to ease the debugging process.

3- For combinatorial designs, all inputs to the combinatorial block were listed in the sensitivity list of the process statement. This insures that no discrepancies are manifested when comparing software to structural simulation.

4- To avoid performance degradation during RTL acceleration, users were instructed to minimize the use of software links connecting hardware accelerated instances. In other words, limit the interaction between models on the accelerator with models running on the software platform.

3.4 Software Utilities

It was imperative that a uniform modeling approach be common to the ASICs and the modules. As a result, several utility tools were created and made accessible to all designers. This toolbox became a kind of checklist before the release of the chips to the foundries. For instance, the types of I/O pads and the manner in which they are netlisted became an issue. Instead of having all of the designers waste valuable time trying to resolve this issue, a tool was created. The two main inputs of this tool are the top level entity of the ASIC core and the foundry it was targeted to. The generated VHDL structural netlist file was comprised of a top level entity containing an instantiated ASIC core connected to its associated I/O pads. An additional feature built into the tool was the wiring of the I/O pads to create the parametric NAND tree (see Figure 5 above). Another tool generated the appropriate Parametric NAND Tree testbench which was used not only to verify functionality but also to produce a test vector set for testing at the foundry.

Besides the previously mentioned toolbox, other tools were needed to make the overall system design methodology work within the current Raytheon system. Because module designs had to be released to Raytheon's internal Printed Wiring Board (PWB) shop, schematics had to be captured on Mentor Graphics. Yet, it was also necessary for each module to be simulated in a VHDL environment, particularly since all the ASICs and FPGAs were designed using VHDL. These simulations were to be executed using the IKOS Voyager series and its associated hardware accelerator. Therefore, a schematic-to-VHDL translator (STV) was written to fulfill the module/subsystem simulation requirement. The STV took the basic Mentor symbols and created a structural VHDL program containing the netlisted components, their appropriate declarations, and all signals. The Mentor schematic symbols could also be enhanced through attributes during schematic capture to include the VHDL library name, entity/architecture of VHDL model (e.g. FPGAs, ASICs, unique parts which did not have an IKOS model, etc.) or a remove property (e.g. for termination resistors) which the STV used in the creation of the structural code.

Internally developed tools played an essential role in contributing to the overall success of the program.

4.0 VHDL DATABASE MANAGEMENT

In order for a project of this magnitude to succeed, forethought into the organization of the VHDL model/testbench database is
imperative. Testbench designers, ASIC modelers, module designers, and subsystem leads may potentially need to concurrently access the same set of models. Needless to say, this is a situation which occurs more predominately toward the latter stages of development, when mixed mode simulations are occurring while updates are being performed. Libraries for software simulation must physically reside elsewhere from the acceleration libraries, likewise for the synthesized libraries. Furthermore, released versions of code need a tamperproof home. These situations proved to be troublesome for the first designs.

As a result, the VHDL code organization evolved out of the first designs. Figure 8 illustrates the database structure for an ASIC. Essentially, an account is split into four directories: software simulation, VHDL acceleration, synthesis, and configuration management. Each of the branches of this tree has a UNIX script associated with it -- a script which binds the logical library names found in the actual VHDL code to their physical locations. For example, when a user wishes to use the acceleration libraries the accel_setup script must be executed. Additionally, each branch of the tree contains a UNIX symbolic link to the latest version of the code found in the configuration branch. This alleviates the problem of people using outdated code or inadvertently corrupting the wrong libraries. Furthermore, this facilitates the maintainability of the design.

The organization of the libraries is, in its own right, an issue. Should the design (i.e. subsystem, module, ASIC/FPGA, subfunction, ...) be made up of one library containing all entities and sub-entities, or should the libraries reflect the hierarchy of the design? The answer, though somewhat dependent on the application, lies predominately in the visibility of the particular VHDL unit and its relative weight in the overall design hierarchy. In other words, if a subfunction is used repeatedly across design boundaries, then it should be placed into a separate library. A
perfect example of this would be a specific type of ALU block (see Section 6.0) which is replicated throughout an ASIC. If the VHDL block is a major unit in the system, whether it is an ASIC, module, or subfunction, then it should be placed in a separate library. One should be cautioned not to create an excessive number of libraries as this can adversely affect the maintainability of the code. Multiple libraries within an ASIC proved to be a difficult task for the RTL accelerator at first. A tool bug which was later fixed.

5.0 TESTBENCH METHODOLOGY

The basic test methodology employed on the program holds true regardless of whether a subsystem, module, ASIC, or FPGA is being tested. For ASIC verification, enhancements are made to the testbench to verify timing as discussed below. A brief discussion into the design methodology needs to be addressed.

For the basic design, requirements flow down from the customer specification into lower level design specifications. These lower level design specifications effectively partition the design into the various components necessary to implement the system (e.g. subsystems, modules, and ASICs). Each component is then functionally blocked out and reviewed before coding can begin. Concurrent to the conceptual design, a test plan is written describing the basic flow of tests necessary to verify the design. Following a concept review, an RTL level model is created which is verified in the Voyager software simulation environment via a behavioral testbench. Once basic datapaths and simple control are verified, the RTL model is accelerated on the IKOS platform, where the same testbench is used to reverify that the design still meets its functional goals. The testbench is updated, if need be, to verify all requirements as found in the development specifications. As mentioned previously, the testbench is annotated with the actual specification's verbiage, the SHALL number, and the capability to report the results of the requirement test in the output response file.

Only ASIC testbenches require further enhancements. In all other cases, only functional requirements were verified. This is somewhat due to the fact that modules/subsystems are mixed mode simulations, containing some behavioral models, some RTL models, and some IKOS gate models. ASICs, on the other hand, must obviously work at speed and therefore, must be checked against their respective specified timing constraints before being released to the foundries. Timing specifications can be placed into the testbench at any stage, but have little meaning until the VHDL has been synthesized into gates. Dynamic timing analysis is performed on the IKOS platform using the foundry supplied technology libraries. The same testbench which was used to verify both the software and RTL accelerated code is applied once again. Only now, timing is also checked. Once the chip performs as expected, the netlist is released to the foundry. After foundry layout, back annotation files are supplied from the foundry and are linked into the IKOS environment. Another round of dynamic timing analysis is performed driven by the same testbench.

Timing specifications were generally implemented in the code as constants. [However, in hindsight, it would have been preferable to create a package containing all of the timing information. This, of course, is better VHDL practice since it allows the package to be modified without disturbing the actual testbench.] The timing specs were then used within the testbench to control when inputs were to be applied or when outputs were to be strobed. For inputs, good data was driven for a duration of tsetup to thold. The inverted data was applied outside this band (see Figure 9a). Data was strobed twice on the output: once at the specified output delay, Tco, and once at the next rising edge as shown in Figure 9b.
6.0 STREAMLINING VHDL DESIGN

Typically a design begins with a specification. This spec is then decomposed into functionally related blocks. At this time a functional block diagram is created showing the major components of the design and its interconnect with the outside environment (i.e. the entity). Lower level block diagrams are created, each time revealing more details of the actual implementation which directly corresponds to lower level entities instantiated as components in VHDL. This methodology of breaking down the design into hierarchical blocks is an effective design technique and is highly recommended. Looking back, a graphical tool was needed to capture block diagrams and to automatically produce structural VHDL netlists. A considerable amount of time was spent simply hand translating block diagrams to their associated VHDL counterparts.

As an example, the high level block diagram of a general purpose ALU is shown in Figure 10. The design of this ALU had several goals: 1) It was to be coded in such a way as to be generic enough for a multitude of designs to use it. Data word size, logical functions, mathematical functions, etc. all had to be modifiable. 2) The ALU model was to be written so that another user would not need intimate knowledge of the lower level structure. As a result, the ALU was comprised of three major blocks: glue logic, the controller, and the ALU core logic. The glue logic merely contained input and output staging registers for the ALU, thus allowing a user to add or delete them depending on their individual timing needs. The ALU core logic consisted of the arithmetic machine (i.e. the logical operators, addition, subtraction), a two's complement multiplier, and supporting logic (e.g. scratch pad registers, muxes, input tap registers, constants, etc.). The core was thus a complicated subfunction requiring a detailed lower level block diagram. The last block of the ALU was not very block diagram intensive, but it does illustrate another interesting design concept.

The controller was implemented using the dual process explicit state machine strategy. Enumerated data types were used for the state register, thus allowing the synthesis tool control of the encoding scheme. However, the most unique feature
Figure 10a - ALU Block Diagram
Figure 10b - ALU Logic Block Diagram
of the ALU design was in the combinatorial process. Here, a pseudo-macro language was
developed to control the movement and operation of data throughout the
ALU. In other words, a user could merely write code within a state to
perform an add, multiplication, logical operations, etc, all without
having to know which registers to enable or what control lines to set.
Programs, therefore, were created by linking states together. Figure 11
illustrates a small code excerpt. This language was simply implemented as
procedure calls, where the procedure would set the ALU control word.

By using these strategies to design the controller, a user could rapidly tailor
the ALU block to functionally meet his/her functional goals. Furthermore, this decomposition
of functionality into hierarchical blocks provides a technique to rapidly transition to VHDL coding.

7.0 SUMMARY

Proper training and hands on consulting support were crucial to
ensure a smooth migration to a VHDL based design approach. Utilizing
VHDL resources such as Std_developersKit
packages and behavioral models for complex devices greatly enhanced our
topdown design environment. Furthermore, in-house development of software utilities
were needed to bridge the gap between our present central CAE design environment to
the VHDL based design environment.

To implement our concept design, a considerable amount of time was initially
invested into developing a concrete and proven methodology for all facets of system
design. Likewise, modeling guidelines were adopted for such things as requirements
traceability, RTL coding techniques, VHDL code organization, and database structures.
For ASIC and FPGA designs, VHDL RTL
models and behavioral testbenches were

```
OUT_DECODER: process (CURRENT_INST)
 begin
 procedure subbi (a_op : in a_src;
 b_op : in b_src;
 ) is
 begin
 ALU_SEL <= isubb;
 case a_op is
 when zeros =>
 MATHSEL <= CONSTANT_REG;
 CONSEL <= ZEROS;
 when Reg1 =>
 MATHSEL <= MEMREG1;
 when others =>
 NULL;
 end case;
 case b_op is
 MATHSEL <= CONSTANT_REG;
 CONSEL <= ZEROS;
 when DReg1 =>
 MATHSEL <= DESTREG1;
 when others =>
 NULL;
 end case;
 begin
 case CURRENT_INST is
 end case;
 begin
 end;

 -- Calculate y(x) --
 -- ---------
 when ST4 =>
 subbi (Reg1, Previous_Data); -- y(x) = y(x-1) - z
 -- Store y(x) --
 -- ---------
 when ST5 =>
 store( alu_result, areg, dreg); -- Store
 result
 --
 end case;
 end process;
```

Figure 11 - Controller Code Excerpt

simultaneously generated from their respective specifications. RTL acceleration
in the IKOS Voyager environment was introduced. This step has a unique
capability to qualify the VHDL code for
hardware structure before the actual
synthesis process, thus optimizing the
debug iteration process. This was especially
important in striving to maintain a
topdown design, bottom up verification
methodology.

Testbenches were constructed in such a way
as to be used throughout the various levels
of validation: component level, module
level, and subsystem level. In addition,
IKOS Voyager, which was the simulation
tool for both functional and timing
verification, provided the capability of
preserving the VHDL testbenches through three levels of simulation: software simulation, RTL accelerated simulation, and gate level accelerated simulation. At the gate level simulation, testbenches were instrumental in supplying efficient and concise functional test vectors for foundry release. Since the gates are now targeted toward a particular technology, detailed timing issues could be checked.

For such an enormous design job, various solutions were improvised for many real life problems which naturally surfaced. Many lessons were learned and best of all a solid VHDL infrastructure was established. This includes tools, libraries, methodology, as well as a trained staff. VHDL proved to be a very powerful environment for toptodown design.

8.0 ACKNOWLEDGEMENT

The authors wish to thank the design team and EDA vendor team members for their insightfulness and dedication to this program.

9.0 REFERENCES


