

# Topdown ASIC Design Using VHDL

M. Zia Ullah Khan, Ph.D.  
Integrated Microcomputer Division  
Intel Corporation  
1900 Prairie City Road, Folsom, CA 95630

## 1.0 Abstract

VHDL is a convenient language for developing of ASICs using a top down design approach. A methodology addressing good VHDL coding practices and synthesizability requirements is described. Problems encountered and their solutions in using this methodology on an ASIC project are also discussed.

## 2.0 Introduction

The Peripheral Component Interconnect (PCI) bus is a new standard for local bus in personal computers. It is aimed at providing high performance capabilities in personal computers that are currently only available in high end engineering workstations. Intel Corporation has developed a new chip, the 82378IB, to provide a bridge between the PCI bus and the ISA expansion bus. The 82378IB also integrates many common I/O functions in ISA-based personal computers.

The 82378IB was implemented as an ASIC using VHDL for modeling, simulation and synthesis. The power and flexibility of VHDL lends itself easily to a topdown design paradigm. A good coding methodology is essential to best use VHDL in a large scale design project. Besides promoting good coding styles and practices this methodology should also address how best to use synthesis and simulation tools in a topdown design paradigm.

In this paper we describe our experiences in using a VHDL-based topdown design approach. We describe the coding methodology used for 82378IB's development, the problems encountered and our solutions for these. We conclude with our suggestions for improving the VHDL design environment.

## 3.0 Design Goals

A design team was formed to start development of 82378IB. A number of goals were defined for this project. Some of the major goals were:

- time to market
- boost designer's productivity
- multiple sourcing of ASICs
- design reusability
- fully synthesizable models

The 82378IB was implemented as an ASIC in gate array technology to meet these design goals. Besides providing fast time-to-market, ASIC methodology makes it possible to

manufacture a chip at several external fabs. VHDL was chosen for modeling, simulation and synthesis because it enables the designers to use a topdown methodology and develop silicon-independent models.

#### **4.0 Design Flow**

Using ASICs for implementing the 82378IB design was a bold new step for our division as we had no previous experience in using ASICs. We also had not used VHDL for implementing a chip before. To help train the design team a detailed design methodology was developed. Before discussing the details of the design methodology we present an overview of the design process. A block diagram of the design flow is shown in Fig. 1.

A high level behavioral model of 82378IB implementing basic functions of the chip was developed first. It had enough functionality to run basic cycles on the PCI and ISA buses but did not have all the features of the chip. This behavioral model was instantiated in a system simulation environment to verify it's interactions with other components in the system. Despite limited functionality, this behavioral model was useful in validating the architecture of the chip and debugging the system simulation environment.

With the architectural concept verified, a detailed design was started. This effort was aimed at developing a synthesizable model of the chip. Later, this RTL model was substituted in place of the behavioral model to exercise it in a system environment. Besides verifying 82378IB in a system environment it was also simulated in a stand alone environment. A detailed test plan was generated to ensure a complete and thorough validation of the chip. This plan called for simulating many different permutations of cycles to exercise the chip in a system environment and in a stand alone environment.

While functional validation effort was in progress, work on synthesizing the model was started. The synthesis process revealed many coding inefficiencies and/or tool limitations. Solutions for these problems were found and necessary changes were incorporated in either the VHDL model of the chip or the synthesis scripts. Any bugs found during this time were also quickly fixed. Incremental synthesis was done on the modified RTL model to update the netlist.

The netlist was checked for timing violations by static analysis and simulations using pre- and post-layout parasitics. This was done in a vendor supplied tool as VHDL does not have a complete solution for back annotating the parasitics. This work required some changes in the design that was either done in VHDL models that were incrementally synthesized or the netlist was edited manually. After verifying the timing performance the netlist was sent for fabrication.

#### **5.0 Coding Methodology**

VHDL can be used to model hardware in a very abstract style. These abstract models must be refined to include implementation details. These refinements must be done while keeping in mind the limitation of synthesis tools. A well defined methodology is necessary to guide designers in writing VHDL models that are targeted for synthesis. Besides synthesizability concerns, a design methodology should also promote good coding practices that allow reuse of models and ease development of useful tools and utilities. A methodology should also suggest known good methods for coding the commonly used design styles.

A detailed coding methodology was developed to satisfy the requirements mentioned above. This methodology covers good coding practices, synthesizable constructs of VHDL, templates for coding basic building blocks (e.g. multiplexer, latches, flip flops,

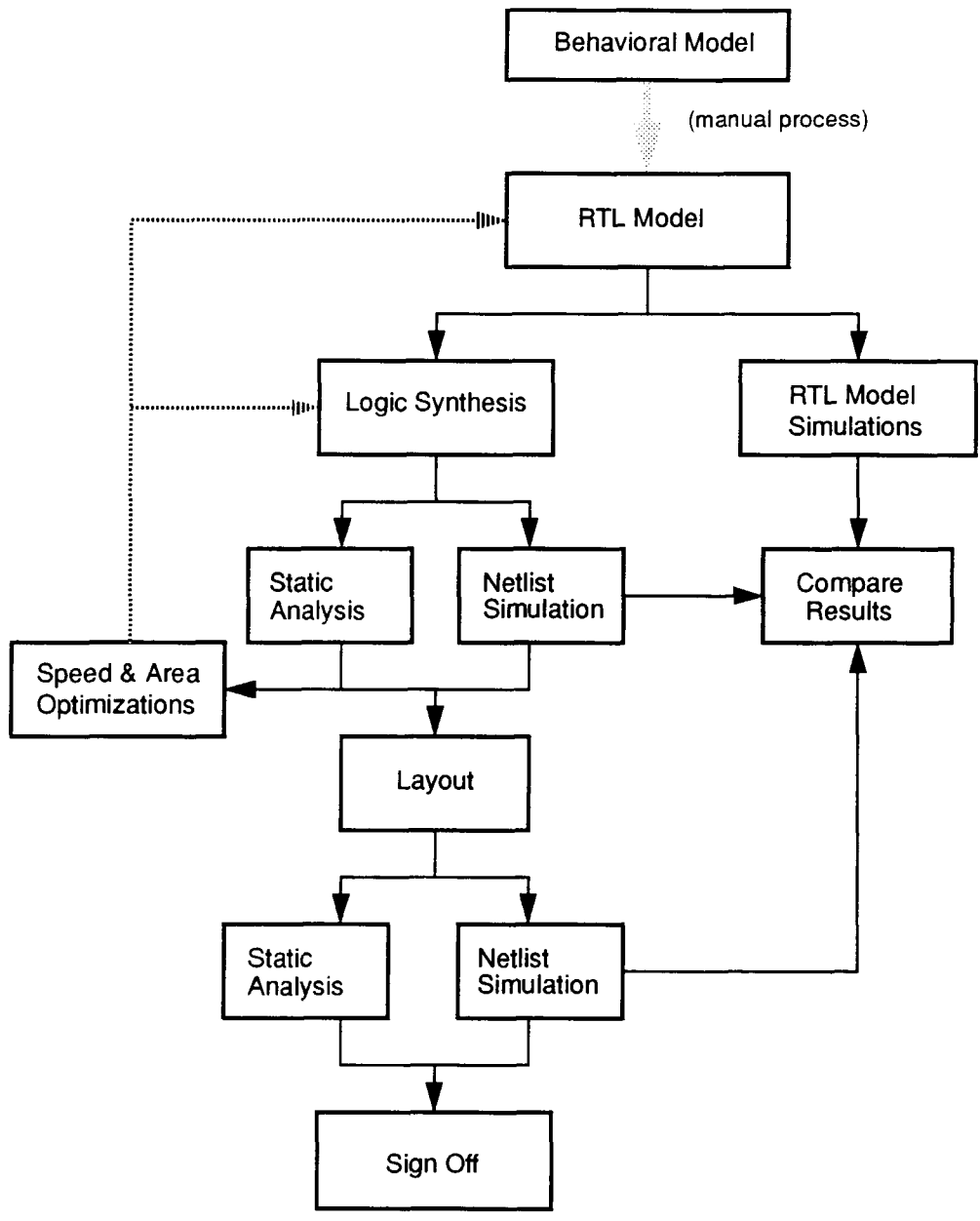


Fig. 1. ASIC Design Flow

finite state machines) and known limitations of current tools and their work arounds. Salient features of this methodology are discussed in the following sections.

### 5.1 Logic Value System

The first step in using VHDL is the choice of a logic value system. The default binary value system using BIT and BIT\_VECTOR is clearly inadequate for practical designs. After an evaluation we chose the 9-value logic system defined by IEEE standard 1164. This standard defines base types called std\_ulogic (unresolved) and std\_logic (resolved) that can have the following values: 'U', 'X', '0', '1', 'Z', 'W', 'L', 'H', '-'. A package called std\_logic\_1164 is available from IEEE that defined these types and boolean operations on these types. The use of 9-value logic system was considered useful as it allows the designers to see initialization or bus conflict problems during VHDL simulations and correct them at an early state of the design process.

The std\_logic\_1164 package does not define any arithmetic or logic operations on basic data types. To avoid spending our resources in developing these functions we decided to look at commercially available solutions. We found the Standard Developer's Kit from VHDL Technology Group to be an acceptable solution for our simulation needs [1]. This is a set of packages that provide a number of useful functions that operate on std\_ulogic and std\_logic types defined in std\_logic\_1164 package. Unfortunately, these functions were not synthesizable and thus could not be used in code targeted for synthesis. This led us to consider the solution from Synopsys (the arithmetic and attributes packages). These packages define functions on UNSIGNED and SIGNED types that are subtypes of BIT. We modified these packages appropriately to make UNSIGNED an subtype of std\_ulogic. With these changes in place we could use the IEEE's 9-value logic system with all the arithmetic operations provided by the Synopsys synthesis tool. The individual designer refer to these packages in their files and were not allowed to develop packages of their own.

### 5.2 Object Declarations

VHDL allows multiple port or signal declarations in the same statement. For instance, consider the port declarations for a 1-bit adder shown below.

```
ENTITY adder_ent IS
  PORT ( A, B : IN std_ulogic;
         Z : OUT std_ulogic );
END adder_ent;
```

The ports A and B are declared on the same line. In our methodology all ports, signals and variables are declared such that there is only one declaration per line. Thus, the above mentioned 1-bit adder will be declared as follow:

```
ENTITY adder_ent IS
  PORT (
    A : IN    std_ulogic;
    B : IN    std_ulogic;
    Z : OUT   std_ulogic );
END adder_ent;
```

This coding style allows easy addition and deletion of signals that comes in handy during initial model development where frequent changes in interface signals is common. Additionally, it is easy to parse entity declarations with simple awk/shell programs thus easing the development of useful utilities for doing routine tasks.

### 5.3 Component Instantiations

In VHDL a hierarchical design is described by instantiating lower level entities as component in the architecture of the higher level entity. Component instantiation can be done either using names association or positional association. For an example of two styles consider a 1-bit adder whose entity is defined as follow:

```
ENTITY adder_ent IS
  PORT (
    a      : IN    std_ulogic;
    b      : IN    std_ulogic;
    c_in   : IN    std_ulogic;
    s      : OUT   std_ulogic;
    c_out  : OUT   std_ulogic
  );
END adder_ent;
```

This component in an architecture can be instantiated in the following ways:

*Named instantiation:*

```
my_adder: adder_ent PORT MAP (a=>x, b=>y, c_in=>carry_in, s=>sum, c_out=>carry_out);
```

*Positional instantiation:*

```
my_adder: adder_ent PORT MAP ( x, y, carry_in, sum, carry_out);
```

At first glance the positional association seems more compact and convenient. But this method of instantiation depends upon the exact sequence of port declaration. This could be a problem as during design process the order and number of port signals changes frequently. It is very difficult and time consuming to track down problems due to incorrect instantiations. On the other hand, instantiation by named association does not depend upon the order of the instantiation and therefore does not require modification when port list order is changed. Changes in interface signal list are easy to incorporate in models. This type of instantiation is also easy to read as it clearly states the exact signal bindings and one need not look up the declaration order. Our methodology requires that all component instantiation be done via names associations.

### 5.4 Propagation of Xs

One of the reason for selecting the 9-value logic system was the ability to propagate Xs in the model. This is useful for debugging initialization and bus contention problems. However, we found that it's scope is limited by coding style. For instance consider the following code fragment:

```
mux_0: PROCESS (sel, data_1, data_2)
BEGIN
  IF (sel = '1') THEN
    z <= data_1;
  ELSE
    z <= data_2;
  END IF;
END PROCESS mux_0;
```

This model implies a multiplexer that is controlled by the signal sel and it's output is connected to data\_1 when sel='1' and data\_2 otherwise. An interesting situation occurs when sel='X'. Should the output of the multiplexer be an 'X' or data\_2? According to the VHDL code, z <= data\_2 is the correct assignment because of the ELSE statement. But

that is not how the netlist simulator sees it. An 'X' on the sel line is propagated through so that this problem can be seen at an output signal. This model could cause initialization problems in netlist simulation but they can not be seen at VHDL model. To avoid such problems our methodology requires that such structures be coded as follows:

```
mux_1: PROCESS (sel, data_1, data_2)
BEGIN
  IF (sel = '1') THEN
    z <= data_1;
  ELSIF (sel='1') THEN
    z <= data_2;
  ELSE
    z <= 'X';
  END IF;
END PROCESS mux_1;
```

Or:

```
mux_2: PROCESS (sel, data_1, data_2)
BEGIN
  CASE sel IS
    WHEN '1' =>      Z <= data_1;
    WHEN '0' =>      Z <= data_2;
    WHEN OTHERS =>  Z <= 'X';
  END CASE;
END PROCESS mux_2;
```

From a logic design point of view, both these models (mux\_1 and mux\_2) are functionally equivalent to the previous model (mux\_0) but these models propagate Xs while the previous model did not. This model will show the X problem in VHDL simulations. It is important to note that the synthesis tool makes use of the don't care condition and does not generate extra logic.

## 5.5 Finite State Machine Design

There are two common methods for coding finite state machines (FSM) in VHDL; using signals for state variables or using variables for state variables. In FSM models using signals for state variables two processes are used, one for inferring state variables and other for defining the state transitions. Models that use variables for storing state variables use one process. This process is used to infer the storage elements for the state vector and also to define state transitions.

Using signals for representing state variables allows continuous tracking of the value of the state vector during simulation because the simulator keeps a record of signal values. This makes it very easy to debug the design. However, using signals may have some detrimental effect on simulation speed. Using variables for representing state variables is faster in terms of simulation speed but the values of state vectors can not be monitored at all times. This is because the variables in VHDL have only instantaneous values and unlike signals there is no history associated with variables. We adopted the two process method because it is easier for coding and debugging.

## 5.6 IO Pad Ring

The ASIC vendors provide a variety of IO buffers based on their drive and transient response characteristics. Even though they are functionally equivalent from a logic design point of view, the differing circuit characteristics require the choice of appropriate buffer for particular signals. To use the desired buffers we instantiated them in our

design. This was very convenient as the design was hierarchical and the buffers were instantiated at the highest level of hierarchy.

### **5.7 Megafunctions**

The 82378IB design also uses some pre-designed megafunctions provided by the silicon vendor. The vendor did not have a VHDL model of the megafunctions and the netlist of the design was encrypted to protect their proprietary design. This presented a problem as without a VHDL model for the megafunctions we could not simulate the entire design in VHDL. This was addressed by writing behavioral models for the megafunctions and verifying them against test vectors provided by the silicon vendor. This required significant effort and resources. It would be highly desirable that the silicon vendors provide VHDL models of their megafunctions.

### **5.8 Synchronous vs. Asynchronous Reset**

Our initial plan called for using synchronously resettable flip flops exclusively. This worked fine in our VHDL simulation models but we found that the gate level model of the chip would not initialize. An analysis revealed that during logic optimization the synthesis tool created logic structure that were not initializable. This was implementation specific and could not be controlled in VHDL models. After much experimentation we decided to go with asynchronous reset scheme that could be easily implemented by the tool.

### **5.9 Design Data Management**

In order to manage our design files we defined a file naming convention that included an identifier in the file name to indicate its contents. For instance, all entity files had `_ent.vhdl` suffix in their names. Our methodology called for keeping all design units (i.e. entity, architecture, configuration, packages, etc.) in separate files. The rationale behind this requirement was many fold:

- Keeping entity and architecture in separate files allows keeping several architectures around for what-if analysis before committing to one particular implementations.
- Having the architecture in a separate file one could make changes to it and then just analyze only the architecture for simulation.

### **6.0 Synthesizable VHDL**

A common problem in using a synthesis tool on VHDL models is that there are many constructs in the language that are not synthesizable. As the language was primarily designed for simulation and documentation, it has many constructs that have no equivalent in hardware e.g. records, process sensitivity lists, etc. Also, the current synthesis technology is not sufficiently advanced enough to implement some of the language constructs even though they may have precise hardware equivalents.

The problem of unsynthesizable constructs in VHDL has been addressed by synthesis tool vendors by defining a subset of VHDL that is acceptable to their tools. As there is no standard governing these subsets, different vendors have chosen different subsets based upon the capabilities of their tools. However, the situation is not so bad as it sounds; there is a significant overlap among the subsets defined by various synthesis tools.

In our design environment we use synthesis tools from Synopsys. Therefore, we used the VHDL subset as defined by Synopsys for coding RTL models. We found this subset to be broad enough to do our designs without overly restricting designers. However, there were

some exceptions where even more limitations were placed on the designers in the interest of uniform coding style. For instance, the Synopsys tools support the WAIT ON construct in VHDL. But we decided to use only processes with sensitivity list and 'EVENT attributes to imply edge sensitive flip flops.

### **6.1 Process Sensitivity List**

The process sensitivity list is an interesting concept in VHDL. It is essentially a simulation trick to improve the performance of the simulator. There is no hardware equivalent of this construct. Therefore, the synthesis tools ignore the sensitivity list when implementing the logic in gates. In case of an incorrect sensitivity list, the results of the RTL model and the synthesized netlist may be different. We found this to be a major problem. Therefore, all designers were required to include all the signals being read inside a process in it's sensitivity list. We were aware that this could cost more CPU time but we decided to take the penalty in simulation throughput and go for accuracy of results.

### **6.2 Flip Flop Inference**

One of the powerful capability of current synthesis tools is their ability to infer a storage element (latch or flip flop) from the VHDL description. This is a very useful capability as it allow the designer to write abstract models that are technology independent. However, this requires careful coding practices.

Initially we decided to code all flip flops and latches so that they could be inferred by the synthesis. For example a simple positive edge sensitive flip flop was coded as follows:

```
my_flop: PROCESS (clk, d)
BEGIN
    IF (clock = '1' and clock'EVENT) THEN
        q <= d;
    END IF;
END PROCESS my_flop;
```

Our design uses both positive and negative edge triggered flip flops. The synthesis tool could only infer positive edge triggered flip flops. When faced with a negative edge flip flop code, the tool would infer a positive edge flip flop and add an inverter to the clock line. This was an unacceptable implementation. To solve this problem, we added key words to the process labels that were supposed to infer negative edge flip flops. These key words were used later in synthesis process to select the inferred (positive edge) flip flops and swap them with negative edge flip flops.

### **7.0 Netlist Simulations**

Our initial impulse was to use VHDL for netlist simulations as well. This would allow the designers to continue to remain in one simulation environment. But we had to go to another simulations environment because of the following reasons:

- VHDL models of the cells in the target library are needed for simulating the netlist.
- VHDL does not have the necessary features to do back annotation of paracitics.
- Simulating in VHDL at such a low level is too slow.
- The silicon vendor had a different sign-off simulator.

The silicon vendor did not have VHDL models for cells in the design library. Therefore we could not do netlist simulation in VHDL. Although we had to use a different simulator

for netlist simulations, we feel that this unnecessarily forces the designers to learn another language and simulator. It would be desirable that the silicon vendors provide VHDL model if their libraries and VHDL have the capability to back annotate paracitics. Improvement in simulation speed is also necessary to be able to simulate netlists in reasonable time.

## **8.0 Conclusions**

VHDL provides a very powerful environment for topdown design. A methodology for making efficient and effective use of VHDL in a design environment is presented. Our design methodology was a great success. We were able to produce functional silicon in record time. Despite a significant learning curve, most designers became very productive in a short time. The power of flexibility of VHDL was found to be very useful in modeling our design at a high level of abstraction.

A strong methodology was instrumental in creating synthesizable models in an efficient manner. To make the best use of VHDL's capabilities it is important to adhere to a design methodology that accommodates the requirements of synthesis tool. The lack of back annotation capability in VHDL requires the users to use a different simulator for netlist validation.

## **9.0 Acknowledgment**

The author wishes to thank the 82378IB project team members who used this methodology and provided valuable feedback.

## **10.0 References**

2. Std\_DevelopersKit, VHDL Technology Group, Allentown, Pennsylvania, USA.