

Protecting Confidentiality of VHDL Models

Satish Acharya
Integrated Microcomputer Division
Intel Corporation
1900 Prairie City Road, Folsom, CA 95630

1.0 Abstract

VHDL provides the ability to use external industry standard CAD tools for chip design. Sometimes it is necessary to provide VHDL models with the original source code to CAD vendors to resolve tool bugs. In order to protect the intellectual property and prevent any disclosure of proprietary information, it is desirable to hide critical design information but still provide the source code to vendors for tool debug.

VEP (VHDL Encryption Program) is a VHDL code encryption utility that has been developed to protect the confidentiality of design information. This tool uses a set of predefined rules to encrypt specific signals, variables and design unit names. The resulting model is still syntactically correct VHDL but has minimal design specific information. VEP has been used on over 7000 VHDL files for exchanging design data with tool vendors.

2.0 Introduction

With the growing popularity of VHDL as a hardware description language, new CAD tools are constantly introduced into the market to assist in the design and implementation of VHDL based designs. During the usage of these tools, design engineers and system level simulation engineers often have to deal with unexpected behaviors or bugs in the tools themselves. Bugs that can be reproduced are communicated to the CAD vendor to provide fixes in future releases of the tool. However, in order to fix these bugs, tool vendors frequently request the original VHDL code that caused the erratic behavior or a test-case so that the problem can be replicated at the other end.

The highly sensitive and proprietary nature of some VHDL designs prevents release of the original VHDL source code to the vendor. This problem was also experienced by the U.S. Air Force during the development of weapon system electronics for their F-22 program [1]. One approach to solve this problem is to invent a small test-case that can also manifest the tool bug. Unfortunately, it is not always feasible to come up with such a test-case especially when designs are very large. An alternative to developing a testcase is to encrypt the original VHDL model before transferring it to the vendor. Encryption provides additional security when used with Non-Disclosure Agreement (NDA) which is frequently signed between the participating agencies before release of any proprietary information. The purpose of encryption is to hide critical design information in the VHDL code. The encrypted model must retain its original functionality to reproduce the bug in the tool. Since the original functionality is preserved, the encrypted model is still synthesizable and simulatable if the original one was.

3.0 The Encryption Problem for VHDL Models

In VHDL based designs, comments along with design unit and signal/variable names contain a lot of design information. Guided by naming conventions, **signal**, **variable**, **entity**, **architecture** and other design unit names tend to be very descriptive and explicitly state their functionality. Comments disclose various design details, how each design unit interacts with other units and the functions of various signals/variables. The task of encryption, hence, involves modifying these descriptive names and comments and still maintain original functionality. It is recommended to scramble the comments rather than eliminating them in order to maintain correspondence in line numbers between original and encrypted VHDL models. This will ease up future communication between the vendor and design engineers. The resulting model must be syntactically correct VHDL and retain its original functionality with minimal design specific information. This is very desirable for model maintenance and reuse.

Unfortunately, the size and complexity of today's designs eliminates all attempts of hand manipulations to perform the desired encryption and hence arises the need for an automated tool. Comments can be easily modified using a simple LEX-based utility but encoding the descriptive names was found to be a non-trivial task.

The encrypted model produced by a VHDL encryption tool must have following properties:

- It must be syntactically correct VHDL code.
- It must retain the functionality of the original model.
- If the original model is synthesizable or simulatable, the encrypted model should also be.
- Comments must be encrypted and correspondence in line numbers be preserved between the original and the encrypted model.
- It must have signal, variable and design unit names encrypted.

VEP (VHDL Encryption Program) is an encryption utility that is developed to hide the critical design information in VHDL models. It encodes comments and modifies specific signals, variables and design unit names.

4.0 VEP

Before presenting the details of VEP design, we discuss various approaches that were looked at to encrypt VHDL files.

4.1 Global Substitution of all identifiers

The most straight-forward and obvious way to encrypt a VHDL file satisfying the above requirements is to construct a lexical analyzer that recognizes all the VHDL identifiers in the model, assigns a unique but arbitrary name to each one of them and then performs global substitution on them with the encoded names. Every VHDL identifier in a design unit is encrypted. The encode names can be anything that is automatically generated - for example *var_1*, *var_2* and so on.

However, the problem with this approach is that it also encrypts names such as **std_logic_vector** or **Integer** that are declared in standard packages. Another problem is

that it modifies all instances of a name even if it appears in different contexts. For example - if **read** is a locally defined signal, it is translated in all the design files into a unique name even if it is used in the context of a function (**read** is also a built-in function defined in one of the standard IEEE libraries.) This will result in compilation errors. The translation is also performed on identifiers such as **ieee** and user-defined library names which is not desirable.

4.2 Global Substitution of selected identifiers

The above problems suggest filtering identifiers that are declared as **entity**, **architecture**, **configuration**, **component**, **signal/port**, **variable** and **constant** names. Since it is easy to recognize declarative statements, one could possibly perform global substitution on only names with the encoded ones in the design units. Since descriptive names with the above declarations convey the bulk of design information, it is adequate for most purposes to encrypt only these names.

Unfortunately, even this approach fails for cases when, for example, an identifier is declared both as an **entity** as well as a **library** name. Global substitution would also encrypt the instance where the identifier is used as a **library name**. Another example is if an identifier is declared locally as a **variable** and elsewhere as a **record**, the instance where the identifier is used as a **record** is also encrypted which might result in syntax errors during compilation.

In order to produce syntactically correct VHDL code, the context of every occurrence of an identifier in the design unit needs to be analyzed first before a substitution is performed. An identifier can be translated only if it appears in the context of one of the above stated declarations. This calls for a VHDL parser that can recognize and infer the context of every occurrence of an identifier.

4.3 VEP Basics

VEP has a built-in Yacc-based parser and uses the context-free grammar specified in BNF format in the VHDL Language Reference Manual (LRM) [2]. Implementing a VHDL-parser is a difficult problem because the grammar itself is ambiguous. Removing the ambiguities is also a non-trivial task. This is due to the fact that in VHDL, an identifier can occur either in a declaration or in the use of a declaration. Recognizing the context of an identifier from a declaration is easy but it is hard to infer the context from its use. We need an unambiguous grammar in order to examine the context of every occurrence of an identifier. Instead of investing a lot of time and effort in removing ambiguities, a grammar that had most of its ambiguities resolved was obtained from the public domain. This grammar, refined by Sidharta Datta at University of Cincinnati, was enhanced to accept certain VHDL constructs which it did not recognize. The modified grammar still has 3 shift/reduce conflicts and 3 reduce/reduce conflicts but is adequate for VEP use.

The grammar is capable of parsing each input VHDL file and recognize identifiers from their context as to which of **entity**, **architecture**, **configuration**, **component**, **signal/port**, **variable** and **constant** declarations does that particular instance belongs to. Any identifier that falls into one of these declarations can be potentially encrypted. However, due to conflicts in the grammar, the parser in some cases fails to infer the right declaration from the context of an identifier. Because of the complexity and effort involved in improving the VHDL parser, VEP took a different but simplified approach. Instead of trying to solve the problem of encrypting identifiers specific to a subset of declarations, VEP

attempts to solve the complement of the problem. It focuses on recognizing identifiers in design units that must not be encrypted namely **type**, **subtype**, **procedure**, **function**, **package**, **package body** and **library** declarations. The built-in parser of VEP parses each VHDL file and obtains the context of every instance of an identifier in declarative statements and places the identifier in one of the two sets, **A** or **B**. An identifier is placed in **set A** if it occurs as **type**, **subtype**, **procedure**, **function**, **package**, **package body** or **library** declarations else it is placed in **set B**.

Thus, it would seem like **B - A** is the desired set of identifiers that can be globally substituted with their associated cryptic names. Unfortunately, the set **B - A** frequently contains identifiers that were declared elsewhere for example in the standard libraries. Encrypting such identifiers would result in compilation errors. To solve this problem, VEP uses a list of identifiers known as **extended keywords**. This list is generated using a modified version of the parser and consists of all identifiers in the standard libraries with following declarations: **type**, **subtype**, **enumerated type**, **procedure**, **function**, **package**, **package body**, **library**, **access**, **attribute**, **file**, **record** or **constant**. Any identifier in a design unit that belongs to **extended keywords** is not encrypted irrespective of the context of its use. Call the **extended keywords** as **set C**. Finally, the set of identifiers that can be safely encoded is the **set D** given by:

$$D = B - (A \cup C) \quad \dots\dots\dots [1]$$

The **set D** consists of those VHDL identifiers present in the design units that are neither **extended keywords** nor they belong to any of **type**, **subtype**, **procedure**, **function**, **package**, **package body** or **library** declarations (see Figure 1).

4.4 VEP Implementation Details

VEP's flow broadly consists of following three steps (see Figure 2):

- Generate a list of identifiers for encryption (namely, **set D** in equation [1]),
- Build an **Encryption Table** that contains the mapping between the original identifiers and their corresponding encoded names, and
- Modify comments and identifiers in the VHDL files using the **Encryption Table**.

Each input VHDL file is first run through a LEX-based utility **tolower** to convert all characters into lower case. Character and string literals are excluded because they are case sensitive. The result is then run through another VHDL utility **vhdlcaps** to capitalize all VHDL keywords. **vhdlcaps** uses a list of VHDL keywords specified in **vhdl.pats**. These two utilities make parsing of a VHDL file easier by ensuring consistency in the case of the reserved words. The result is then fed to another utility **gentable** which accumulates identifiers in sets **A** and **B** from each VHDL file. **E T Generator** uses sets **A**, **B** and **extended keywords** to generate set **D** using the UNIX utility **comm**. It then assigns unique names of the form *var_i*, where *i* = 1, 2, 3,..... to each item in set **D** and builds an **Encryption Table**. During the final pass, VEP encrypts the VHDL files using the **Encryption Table**. **encrypt** is a LEX-based utility that reads in each VHDL file, recognizes every identifier and if found in the **Encryption Table**, substitutes the identifier with its corresponding encoded name in the VHDL file. It also modifies comments by replacing each character in the comments by an **X**. The **Encryption Table** that VEP gen-

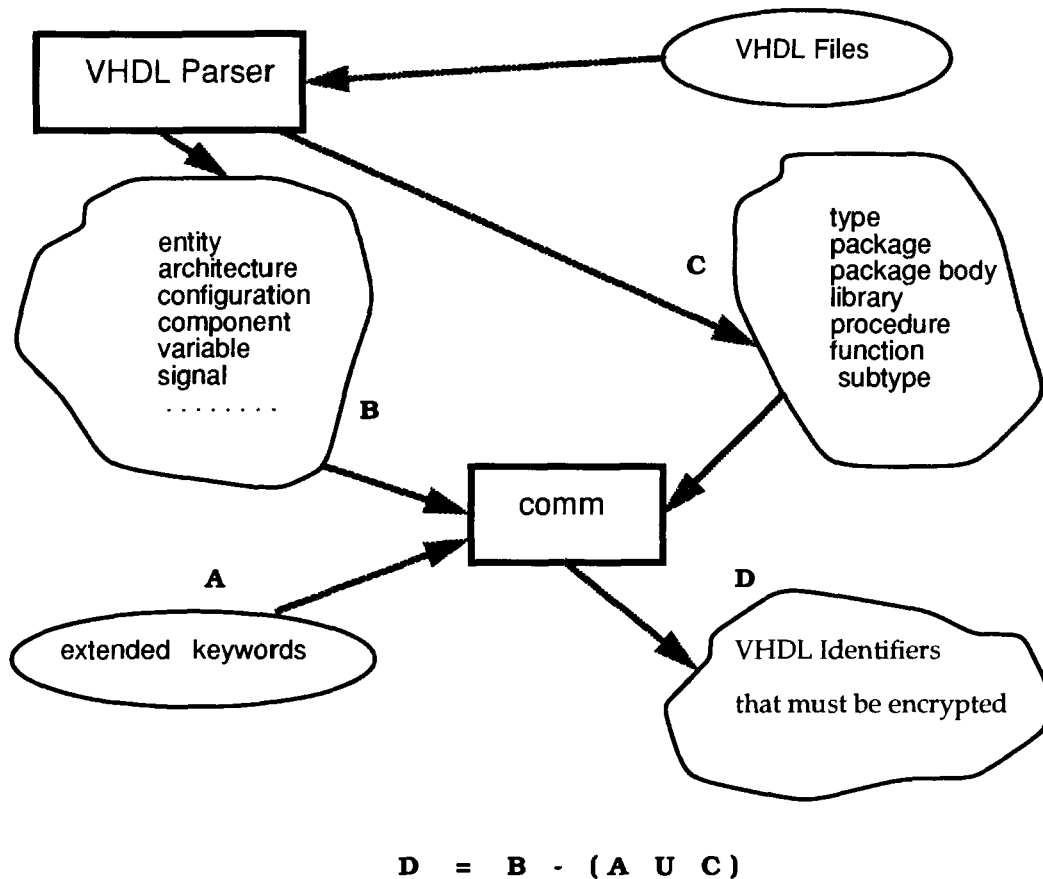


Figure 1. Encryption of identifiers in VEP

erates can be referred by end-users to obtain the mapping between the original and encoded names.

VEP is very flexible and modular in its design for ease of implementation and any future enhancements. In the process, however, it makes four passes through each input VHDL file. Performance was not considered a critical factor during the design of VEP.

4.5 Limitations of VEP

Like every other tool, VEP has its own limitations. VEP accepts only syntactically correct VHDL code. If an identifier occurs both in sets **B** and **(A U C)** in equation [1], it is not encrypted. This occurs when for example an identifier occurs both as a **library** name as well as an **entity** name. Another example is when an identifier occurs as a **signal** name in a design unit but as a **function** name in one of the standard libraries. Since VEP modifies comments, any directives for Synthesis Tools embedded in the comments are also modified for example: -- synopsys translate_off . VEP works only on stand-alone VHDL models i.e. the list of VHDL files presented to VEP must be complete with the sole exception of

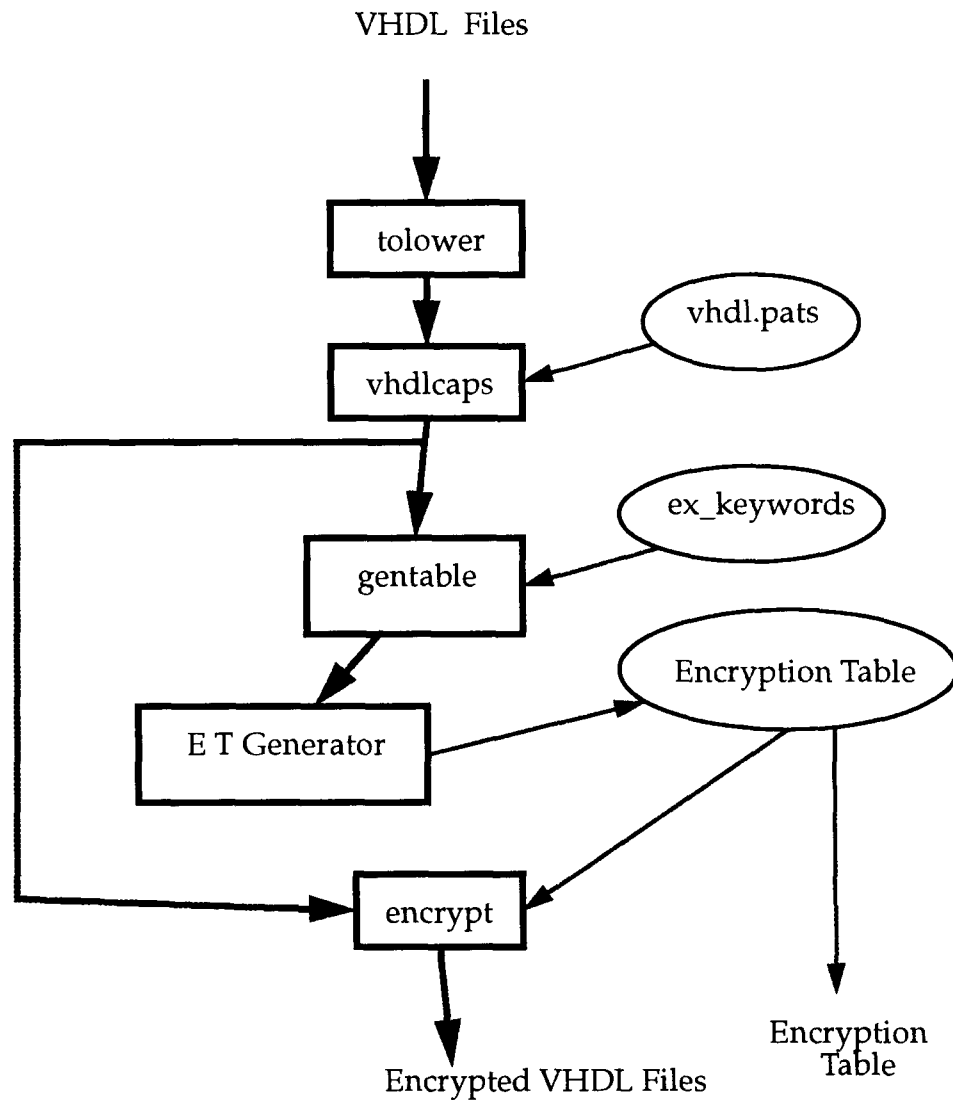


Figure 2. VEP Flow

the standard libraries such as **ieee** and **std_developer's kit**. Without the complete list, VEP sometimes encrypts instances of identifiers that are declared in the file not included in the model. This might lead to compilation errors. Indeed, users do have a choice of encrypting only a subset of files submitted to VEP.

5.0 Examples

We now present two examples of VHDL code to illustrate the capability of VEP. We have chosen a simple combinational circuit and a finite state machine since they represent

different coding styles that are most commonly used in chip design. For each example, the original code is listed first followed by the encrypted version.

5.1 Example of a combinational circuit

Consider the following VHDL model as an example of a combinational circuit:

```
-- FILENAME : adder.vhdl
-- AUTHOR   : Satish Acharya
-- PROJECT  : PROJ
--
-- This is an example of a simple 1-bit full adder

ENTITY full_adder_ent IS
  PORT (
    -- this is an interface declaration for
    -- 1-bit full adder.
    a, b, cin : IN BIT;
    sum, cout : OUT BIT );
END full_adder_ent ;

ARCHITECTURE full_adder_b_arc OF full_adder_ent IS
  BEGIN -- Behavioral description
  PROCESS (a, b, cin)
  BEGIN
    sum <= a xor b xor cin ;
    cout <= (a and b) or (a and cin) or (b and cin);
  END PROCESS;
END full_adder_b_arc;

CONFIGURATION full_adder_cfg OF full_adder_ent IS
  FOR full_adder_b_arc
  END FOR;
END full_adder_cfg;
```

The encrypted model as the result of a run through VEP is:

```
--XXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XXXXXXXXXXXXXXXXXXXXXXXXXXXX
--XX
--XXXXXXXXXXXXXXXXXXXXXXXXXXXX

ENTITY var_5      IS
  PORT (
    --XXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

```

--XXXXXXXXXXXXXXXXXXXXX
  a, b, var_1 : IN bit;
  var_6, var_2 : OUT bit );
END var_5      ;

ARCHITECTURE var_3      OF var_5      IS
  BEGIN --XXXXXXXXXXXXXXXXXXXXX
  PROCESS (a, b, var_1)
  BEGIN
    var_6 <= a XOR b XOR var_1 ;
    var_2 <= (a AND b) OR (a AND var_1) OR (b AND var_1);
  END PROCESS;
END var_3      ;

CONFIGURATION var_4      OF var_5      IS
  FOR var_3
  END FOR;
END var_4      ;

```

VEP generates an **Encryption Table** as:

```

cin var_1
cout var_2
full_adder_b_arc var_3
full_adder_cfg var_4
full_adder_ent var_5
sum var_6

```

This encryption table can be used to map the encrypted names to the original names.

5.2 Example of a finite state machine

Consider the following VHDL model representing a finite state machine:

```

LIBRARY IEEE;
  USE IEEE.std_logic_1164.ALL;
LIBRARY STD_DEVELOPERSKIT;

ENTITY mealy_ent IS
PORT (
  a : IN      std_ulogic;
  clock : IN   std_ulogic;
  reset : IN   std_ulogic;
  z : INOUT   std_ulogic );

```

```

END mealy_ent;

ARCHITECTURE mealy_arc_d OF mealy_ent IS
    TYPE state_type IS (S0, S1);
    -- enumerated data type for FSM states
    SIGNAL pres_state   : state_type;
    SIGNAL next_state   : state_type;
BEGIN
    -- Next state assignment; synchronizing process
    sync: PROCESS (reset, clock)
    BEGIN
        IF ( reset = '1') THEN
            pres_state <= S0;           -- async reset
        ELSE
            IF (clock'EVENT and clock = '1') THEN
                pres_state <= next_state; -- assign next_state
            END IF;
        END IF;
    END PROCESS sync;
    -- process to hold combinational logic that determines next-state
    comb : PROCESS (pres_state, z)
    BEGIN
        CASE pres_state IS
            WHEN S0 =>                -- state S0
                IF (a = '0') THEN
                    z <= '0'; next_state <= S0;
                ELSE
                    z <= '1'; next_state <= S1;
                END IF;
            WHEN S1 =>                -- state S1
                IF (a = '0') THEN
                    z <= '0'; next_state <= S0;
                ELSE
                    z <= '0'; next_state <= S1;
                END IF;
            WHEN OTHERS =>            -- default state
                z <= '-'; next_state <= S0;
                -- go to S0 the idle state
        END CASE;
    END PROCESS comb;
END mealy_arc_d;

```

The encrypted code generated by VEP is as follows:

```

LIBRARY ieee;

```

```

USE ieee.std_logic_1164.ALL;
LIBRARY std_devloperskit;

```

```

ENTITY var_3 IS
PORT (
  a : IN std_ulogic;
  clock : IN std_ulogic;
  var_6 : IN std_ulogic;
  z : INOUT std_ulogic);
END var_3 ;

```

```

ARCHITECTURE var_2 OF var_3 IS

```

```

  TYPE state_type IS (var_7, var_8);
  --XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
  SIGNAL var_5 : state_type;
  SIGNAL var_4 : state_type;

```

```

BEGIN
  --XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

```

  var_9: PROCESS (var_6, clock)
  BEGIN
    IF ( var_6 = '1') THEN
      var_5 <= var_7; --XXXXXXXXXXXXXXXX
    ELSE
      IF (clock'event AND clock = '1') THEN
        var_5 <= var_4 ; --XXXXXXXXXXXXXXXXXXXX
      END IF;
    END IF;
  END PROCESS var_9;

```

```

  --
  XXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX

```

```

  var_1 : PROCESS (var_5 , z)
  BEGIN
    CASE var_5 IS
      WHEN var_7 => --XXXXXXXXXXXX
        IF (a = '0') THEN
          z <= '0'; var_4 <= var_7;
        ELSE
          z <= '1'; var_4 <= var_8;
        END IF;
      WHEN var_8 => --XXXXXXXXXXXX
        IF (a = '0') THEN
          z <= '0'; var_4 <= var_7;
        ELSE
          z <= '0'; var_4 <= var_8;
        END IF;
      WHEN OTHERS => --XXXXXXXXXXXXXXXXXXXX

```

```

        z <= '-'; var_4    <= var_7;
        --XXXXXXXXXXXXXXXXXXXXXXXXXXXX
    END CASE;
END PROCESS var_1;
END var_2 ;

```

The **Encryption Table** is as follows:

```

comb var_1
mealy_arc_d var_2
mealy_ent var_3
next_state var_4
pres_state var_5
reset var_6
s0 var_7
s1 var_8
sync var_9

```

6.0 Conclusion

VEP has been developed to enable exchanging of VHDL models while protecting the confidential information in them. The names and comments in a model are encrypted because they reveal significant amount of proprietary design information. The encrypted model generated by VEP is syntactically correct VHDL and preserves the original functionality but has minimal design information. VEP encrypts both dataflow and behavioral models and has been tested on more than 7000 VHDL files for exchanging design data with tool vendors.

7.0 Acknowledgments

The author wishes to thank Sanjay Tayal and Kapila Udawatta of Intel-IMD and Pranav Shah of Design Technology at Intel in Santa Clara for their help and valuable feedback during the course of VEP's development. The author also acknowledges the guidance and support of Michele Ruscito and Zia Khan and lastly, special thanks to Sidharta Datta for his wonderful VHDL grammar without which VEP would have never achieved its current capability.

8.0 References

- [1] T. Harris, A. E. Rosenberg, *VHDL and It's application for Department of Defense Contracts*, VIUF, pp.67-73, Spring 1993, Ottawa, Canada.
- [2] IEEE Standard VHDL Language Reference Manual, IEEE Std 1076-1987, March 31, 1988.