April 22, 2004

Mr Stephen Bailey
Chairman, IEEE 1076 Working Group
6664 Cherokee Court
Niwot, CO 80503

Dear Mr. Bailey,

Cadence Design Systems hereby grants permission to the Institute of Electrical and Electronics Engineers (IEEE) to reprint and/or modify the attached document entitled “A Mechanism for VHDL Source Protection”.

It is understood that if the material is modified by IEEE, IEEE will be the sole owner of the copyright to the resulting document, namely IEEE Std 1076 – IEEE Standard VHDL Reference Manual, being developed by the IEEE 1076 Working Group, subject to Cadence’s underlying right to the unmodified material.

If requested acknowledgement of the original work will be placed in the front matter of the new work.

Very truly yours,

Mitch Weaver
Vice President, SFV
Cadence design Systems, Inc.

A Mechanism for VHDL Source Protection

1 Overview

This section describes the purpose and organization of this document.

1.1 Intent and scope of this document

The intent of this specification is to define the VHDL source protection mechanism. It defines the rules to encrypt the VHDL source. It also defines the format of the encrypted VHDL file. Its primary audiences are the implementor of tools that produce encrypted VHDL, or the tools that consume and process the encrypted VHDL.

1.2 Syntactic description

This specification has been described using the context-free syntax described in the IEEE Standard VHDL Language Reference Manual (IEEE Std. 1076-1993) section 0.2.1.

2 Lexical Conventions

2.1 Pragmas

For encryption of the VHDL source, the pragmas are defined in the following format. This sub clause specifies the syntactic mechanism that shall be used for specifying pragmas, without standardizing on any particular pragmas.

pragma ::= **-- pragma protect** { pragma_expression } **\n**

pragma_expression ::= pragma_keyword

| pragma_keyword = pragma_value

| pragma_value

pragma_value ::= constant_expression

| string

pragma_keyword := **begin**

| **end**

| **data_keyowner**

| **data_keyname**

| **data_method**

| **key_keyowner**

| **key_method**

| **key_keyname**

| **begin_protected**

| **end_protected**

| **key_block**

| **end_key_block**

| **data_block**

| **end_data_block**

| **digest_block**

| **end_digest_block**

3 Compiler Directives

3.1 Protected Envelopes

Protected Envelopes specify a region of text which shall be encrypted prior to analysis by the source language processor. These regions of text are structured to provide the source language processor with the specification of the cryptographic algorithm, key, envelope attributes, and textual design data.

All information which identifies a Protected Envelope is introduced by the **protect** pragma. This pragma is reserved by this standard for the description of Protected Envelopes, and is the prefix for specifying the regions and processing specifications for each protected envelope. Additional information is associated with the pragma by appending pragma expressions.

Envelopes may be defined for either of two modes of processing. Encryption envelopes specify the pragma expressions for encrypting source text regions. Decryption envelopes specify the pragma expressions for decrypting encrypted text regions. Decryption envelopes may contain other envelopes within their enclosed data block. The number of nested decryption envelopes that can be processed is implementation-specified.

```
hdl_envelope ::=          encrypt_envelope
                 | decrypt_envelope

encrypt_envelope ::=      protect_pragma encrypt_content_params begin_pragma source_text end_pragma

encrypt_content_params ::=        key_block_params [data_params_set]

key_block_params ::=      {key_params_set}

key_params_set ::=        key_keyowner_pragma key_keyname_pragma key_method_pragma

data_params_set ::=       data_keyowner_pragma data_keyname_pragma data_method_pragma
```

decrypt_envelope ::= begin_protected_pragma decrypt_content_params decrypt_data_block end_protected_pragma

decrypt_content_params := {decrypt_key_block}

decrypt_key_block ::= key_params_set key_block digest_block

key_block ::= key_block_pragma encoded_text end_key_block_pragma

digest_block ::= digest_block_pragma encoded_text end_digest_block_pragma

decrypt_data_block ::= data_block digest_block

data_block ::= data_block_pragma encoded_text end_data_block_pragma

protect_pragma ::= --**pragma protect \n**

begin_pragma ::= **--pragma protect begin \n**

end_pragma ::= **--pragma protect end \n**

key_keyowner_pragma ::= **--pragma protect key_keyowner=**string **\n**

key_keyname_pragma ::= **--pragma protect key_keyname=**string **\n**

key_method_pragma ::= **--pragma protect key_method=**string **\n**

data_keyowner_pragma ::= **--pragma protect data_keyowner=**string **\n**

data_keyname_pragma ::= **--pragma protect data_keyname=**string **\n**

data_method_pragma ::= **--pragma protect data_method=**string **\n**

begin_protected_pragma ::= **--pragma protect begin_protected \n**

end_protected_pragma ::= **--pragma protect end_protected \n**

key_block_pragma ::= **--pragma protect key_block \n**

end_key_block_pragma ::= **--pragma protect end_key_block \n**

digest_block_pragma ::= **--pragma protect digest_block \n**

end_digest_block_pragma ::= **--pragma protect end_digest_block \n**

data_block_pragma ::= **--pragma protect data_block \n**

end_data_block_pragma ::= **--pragma protect end_data_block \n**

Note:
source_text: The source text encompasses all the text, comments, included pragma directives, user code etc.

encoded_text: is the binary data encoded in printable characters, spanning over multiple lines. This can contains both the encrypted and the message digest data.

The pragma expressions between the protect_pragma and the begin_pragma in a encryption envelope or between the begin_protect_pragma and end_protect_pragma are processed to encrypt or decypt the data in the envelopes.

Examples:

```
library IEEE;
use IEEE.std_logic_1164.all;
package pack_inst is
--pragma protect
--pragma protect data_keyowner =owner1
--pragma protect data_method = RC5
--pragma protect data_keyname = data_test1.1
--pragma protect key_keyowner = keyowner1
--pragma protect key_method = RC4
--pragma protect key_keyname = key_test1.1
--pragma protect key_keyowner = keyowner2
--pragma protect key_method = DES
--pragma protect key_keyname = key_test1.2
--pragma protect begin
signal sigp_protected : std_logic  ;
--pragma protect end
end pack_inst;
```

After processing the above input VHDL the encrypting tool should generate data similar to the following:

```
library IEEE;
use IEEE.std_logic_1164.all;
package pack_inst is
--pragma protect begin_protected
--pragma protect key_keyowner=keyowner1
--pragma protect key_keyname=key_test1.1
--pragma protect key_method=RC4
--pragma protect key_block
T/R0BKmye8wSe1N/JJdpeF3ga6182MsHa15sGnOPLiVkVehOYX4unuoXG6W65Nuy
FY6FWTX+TskQu+qyW+5mVFeMFOtPa6UD8Lfy2S8MuzTDVCGpg8d9k7nXb92SLdeC
fuE/rUhMCQEOtF0sRvAcLGX5Mh3dUql3bncGe8CC2s1yDzmHdDwjuotUN3xDaZVM
sqRv98aQ6gZT5Dg=
--pragma protect end_key_block
--pragma protect digest_block
X9PyX59giDALGPEeblCyRkc3f7E=
--pragma protect end_digest_block
--pragma protect key_keyowner=keyowner2
--pragma protect key_keyname=key_test1.2
--pragma protect key_method=DES
--pragma protect key_block
JgRb6WfTB471/JeMPU/Z6wYS/JE5gz6kExQo2XBDmXto/Zy6KCD5vQWEDqd/PWW0
OIoXudfttDIr5WmN/UvHA2FUHb6BrVVsqNDlTZQZBjMGHxCJpDNZvuLezV9fnia6
pPJTG2pGg6FQMol1ouTK2X39Z/Tn/Q2unXPSLM91Ftd7y58oc/VjQZ4rEV05DzLw
8BuRP9/CdG3A5ICYbXn+Xg==
--pragma protect end_key_block
```

```
--pragma protect digest_block
dkSkDU5xwADj+B7HhomnCn9A8tc=
--pragma protect end_digest_block
--pragma protect data_block
liK5iC2NkCoqsbvVio9k8ak2/mZslgagqY5U570gsbcEHHAcTpMF5NYNqlusKUE8
17mw8C24N3H6CoRiJB1VHA==
--pragma protect end_data_block
--pragma protect digest_block
TgouRSRZcvyiJFdcJiev1uX6dZM=
--pragma protect end_digest_block
--pragma protect end_protected
 end pack_inst;
```

3.1.1 Processing protected envelopes

Two modes of processing are defined for protected envelopes. Envelope encryption is the process of recognizing encryption envelopes in the source text and transforming them into decryption envelopes. Envelope decryption is the process of recognizing decryption envelopes in the input text and transforming them into the corresponding clear text for the parsing step that follows.

3.1.1.1 Encryption

VHDL tools that provide encryption services shall transform source text containing encryption envelopes. The tool replaces each encryption envelop with a decryption envelope by encrypting the source text according to the specified pragmas. Source text which is not contained in an encryption envelope shall not be modified by the encrypting language processor.

In the encryption block if the data pragmas (data_keyname, data_keyowner, data_method) are defined, the specified key and algorithm are used along with the session key to encrypt the data. If these pragmas are absent, a random session key is generated and used to encrypt the data. The encrypted data is enclosed in the **data_block** pragmas. This session key data (or the information about the session key) and the information about the encryption algorithm is encrypted by another key and is output between the **key_block** pragmas. This second key and the algorithm are specified by the key pragmas. If the key pragmas (key_method, key_owner, key_name) are absent in the encryption block, the tool's internal key is used.

3.1.1.2 Decryption

VHDL tools that support compilation of encrypted data internally decrypt the decryption envelopes according to the specified pragma expressions.

4 Protected Envelopes

5 Envelope Directives

Protected envelopes are specified as lexical regions delimited by protect pragma declarations. The semantics of a particular protect pragma declaration is specified by its pragma expressions. This standard reserves the keyword names listed in the following table for use as keywords to the protect pragma. These keywords are defined in section 6.1, with a specification of how each participates in the encryption and decryption processing modes. Some keywords are used exclusively in the encryption envelope, some are used exclusively in the decryption envelope, where as some are used in both kind of envelopes.

The following pragma keywords are relevant to encryption envelopes only:

<empty>	Opens a new encryption envelope
begin	Opens a input data block for encryption
end	Closes an encryption envelope

The following are used only in the decryption envelope:

begin_protected	Opens a new decryption envelope
end_protected	Closes a decryption envelope
key_block	Begins an encoded block of key data
end_key_block	Closes an encoded block of key data
data_block	Begins a block of encrypted data
end_data_block	Closes a block of encrypted data
digest_block	Begins an encoded block of authentication code data for data integrity
end_digest_block	Closes the authentication code

The following are used both by the encryption and decryption envelopes.

data_keyowner	Identifies the owner of the data encryption key
data_method	Identifies the data encryption algorithm
data_keyname	Specifies the name of the data encryption key
key_keyowner	Identifies the owner of the key encryption key
key_method	Specifies the key encryption algorithm
key_keyname	Specifies the name of the key encryption key

The scope of **protect** pragma declarations is completely lexical and not associated with any declarative region or declaration in the HDL text itself.

In the protection envelopes where a specific pragma keyword is absent, the VHDL tool shall use the default value. VHDL tools that perform encryption should explicitly output all relevant pragmas keywords (including the ones for which default values were used) for each envelope in order to avoid unintended interpretations during decryption.

5.1 Envelope encoding keywords

5.1.1 begin

5.1.1.1 Syntax

```
begin
```

5.1.1.2 Description

ENCRYPTION INPUT: The **begin** pragma expression is used in the input text to indicate to an encrypting tool the point at which encryption begins. All text, including comments and other protect pragmas, between the **begin** pragma expression and the corresponding **end** pragma expression is encrypted and is stored in the output format using the **data_block** pragma expression.

Nesting of pragma **begin**/**end** blocks is not supported, although there may be **begin_protected**/ **end_protected** blocks containing previously encrypted content inside such a block. They are simply treated as a byte stream and encrypted as if they were text.

ENCRYPTION OUTPUT: none

DECRYPTION INPUT: none

5.1.2 end

5.1.2.1 Syntax

```
end
```

5.1.2.2 Description

ENCRYPTION INPUT: The **end** pragma expression is used in the input clear text to indicate the end of the region that shall be encrypted

ENCRYPTION OUTPUT: none

DECRYPTION INPUT: none

5.1.3 begin_protected

5.1.3.1 Syntax

```
begin_protected
```

5.1.3.2 Description

ENCRYPTION INPUT: If found in an input file during encryption **begin_protected**/**end_protected** block and its contents are treated as input clear text. This could result from a situation where a previously encrypted model is being re-encrypted as a portion of a larger model. An additional requirement is that any other protect pragmas inside the **begin_protected**/**end_protected** block shall not be interpreted or override pragmas in effect. In this way, nested encryption will not corrupt pragma values in the current encryption in process.

ENCRYPTION OUTPUT: After encrypting a **begin**/**end** block during encryption, the encrypting tool produces a corresponding **begin_protected**/**end_protected** block in the output file. This block begins with the **begin_protected** pragma expression. Following **begin_protected** all pragma expressions required as encryption output shall be generated prior to outputting the **end_protected** pragma expression. In this way protected blocks are completely self-contained avoiding any undesired interaction when using multiple encrypted models during the decryption process.

Note that this does not begin a block of encrypted data or keys, the **data_block** and **key_block** pragma expressions are used for this purpose and they are found within a **begin_protected**/**end_protected** block.

DECRYPTION INPUT: The **begin_protected** pragma expression begins a previously encrypted region. A decrypting tool accumulates all the pragma expressions in the block for use in decryption of the block.

5.1.4 end_protected

5.1.4.1 Syntax

```
end_protected
```

5.1.4.2 Description

ENCRYPTION INPUT: This pragma expression indicates the end of a previous **begin_protected** block. This indicates that the block is complete and new pragma expression values shall be accumulated for the next envelope.

ENCRYPTION OUTPUT: The **end_protected** pragma expression shall be output to indicate the end of a protected block.

DECRYPTION INPUT: The **end_protected** pragma expression indicates the end of a set of pragmas that should be sufficient to decrypt the current block. Upon encountering **end_protected** a tool shall verify that all required information is present.

5.1.5 data_keyowner

5.1.6 Syntax

```
data_keyowner=<string>
```

5.1.6.1 Description

ENCRYPTION INPUT: The **data_keyowner** specifies the company or tool that is providing the keys used for encryption and decryption of the data. The keys might be provided by an IP Author, the encrypting tool, the IP consumer, or possibly even a third party distributor of the IP. It has to be a value which is available in the tool's key database. If this pragma is absent the encrypting tool shall use its own embedded key. If specified, the tool reads the key from the database and uses this to encrypt the data block.

ENCRYPTION OUTPUT: The **data_keyowner** is encrypted with the **key_method** and found in the **key_block**.

DECRYPTION INPUT: During decryption, the **data_keyowner** is combined with the **data_keyname** to determine the appropriate secret/private key to use during decryption of the **data_block**.

5.1.7 data_method

5.1.7.1 Syntax

```
data_method=<method_name>
```

5.1.7.2 Description

ENCRYPTION INPUT: The **data_method** pragma expression indicates the encryption algorithm that shall be used to encrypt subsequent **begin**/**end** block. The encryption method is an identifier that is commonly associated with a specific encryption algorithm.

This standard specifies the following values for the **data_method** pragma expression. Additional identifier values are implementation-defined:

DES	Data Encryption Standard
RSA	RSA Public Key
RC2	RSA RC2
RC4	RSA RC4
RC5	RSA RC5
RC6	RSA RC6

Editor's Note: The above list should be replaced with a normative reference to an existing registry of encryption algorithm identifiers. IETF and W3C are potential registries, and others may exist.

ENCRYPTION OUTPUT: The **data_method** is encrypted with the **key_method** and found in the **key_block**.

DECRYPTION INPUT: The **data_method** indicates the algorithm that should be used to decrypt the data_block.

5.1.8 data_keyname

5.1.8.1 Syntax

```
data_keyname=<string>
```

5.1.8.2 Description

ENCRYPTION INPUT: The **data_keyname** pragma expression provides the name of the key or key pair that is used to decrypt the **data_block**. A given **data_keyowner** will typically have multiple keys that they have shared in different ways with different vendors or customers. This pragma expression indicates which of these many keys has been used.

ENCRYPTION OUTPUT: When a **data_keyname** is provided in the input, it indicates the key that is to be used for encrypting the data. The encrypting tool must be able to combine this pragma expression with the **data_keyowner** and determine the key to use. The **data_keyname** is encrypted using **key_method** and and encoded in the key_block.

DECRYPTION INPUT: In use models where the **data_keyowner** has provided a secret/private key to a Tool Vendor, or a Tool Vendors secret key has been used, then a unique key name must be identified for each key during this exchange. This key name is then used to identify at decryption time which of many possible secret keys for a given key owner should be used for decryption.

5.1.9 data_block

5.1.9.1 Syntax

```
data_block
```

5.1.9.2 Description

ENCRYPTION INPUT: A **data_block** should never be found in an input file unless it is contained within a previously generated **begin_protected**/**end_protected** block in which case it is ignored.

ENCRYPTION OUTPUT: The **data_block** pragma expression indicates that a data block begins on the next line in the file. An encrypting tool takes each **begin**/**end** block, encrypts the contents as specified by the **data_method** pragma expression, and then encodes the block. The resultant text is generated as the output.

DECRYPTION INPUT: The **data_block** is first read in the encoded form. The encoding is reversed, and then the block should be decrypted in-memory for consumption.

5.1.10 digest_block

5.1.10.1 Syntax

```
digest_block
```

5.1.10.2 Description

ENCRYPTION INPUT: none

ENCRYPTION OUTPUT: A Message Authentication Code (MAC) is used to ensure that the IP has not been modified. In Message Authentication Code, the encrypting tool generates the message digest (fixed length, computationally unique identifier corresponding to a set of data). The message digest is generated for both data_block and the key_block.

DECRYPTION INPUT: In order to authenticate the message, the consuming tool shall first decrypt the message, then generate the message digest on the original message, and compare the two message digests. If the two don't match this means that either the MAC or **data_block** or the key_block has been altered, and the tool can error out.

5.1.11 key_keyowner

5.1.11.1 Syntax

```
key_keyowner=<string>
```

5.1.11.2 Description

ENCRYPTION INPUT: The **key_keyowner** specifies the company or tool that is providing the keys used for encryption and decryption of the key information. The value of the **key_keyowner** also has the similar constraint as mentioned in the **data_keyowner** values.

ENCRYPTION OUTPUT: The **key_keyowner** should be unchanged in the output file.

DECRYPTION INPUT: During decryption, the **key_keyowner** can be combined with the **key_keyname** to determine the appropriate secret/private key to use during decryption of the **key_block**.

5.1.12 key_method

5.1.12.1 Syntax

```
key_method=<method_name>
```

5.1.12.2 Description

ENCRYPTION INPUT: The **key_method** pragma expression indicates the encryption algorithm that shall be used to encrypt the keys used to encrypt the **data_block**. The same names and formats are used for **data_method** and **key_method**. The values have the same constraint as mentioned for the **data_method** values.

ENCRYPTION OUTPUT: The **key_method** remains unchanged in the output file.

DECRYPTION INPUT: The **key_method** indicates the algorithm that shall be used to decrypt the **key_block**.

5.1.13 key_keyname

5.1.13.1 Syntax

```
key_keyname=<string>
```

5.1.13.2 Description

ENCRYPTION INPUT: The **key_keyname** pragma expression provides the name of the key or key pair that should be used to decrypt the **key_block**. A given **key_keyowner** will typically have multiple keys that they have shared in different ways with different vendors or customers. This pragma expression indicates which of these many keys has been used.

ENCRYPTION OUTPUT: When a **key_keyname** is provided in the input, it indicates the key that shall be used for encryption of the data encryption keys. The encrypting tool must be able to combine this pragma expression with the **key_keyowner** and determine the key to use. The **key_keyname** itself should be output as clear text in the output file.

DECRYPTION INPUT: In use models where the **key_keyowner** has provided a secret/ private key to a Tool Vendor, or a Tool Vendors secret key has been used, a unique key name must be identified for each key during encryption. This key name is then used to identify at decryption time which of the many possible secret keys for a given key owner shall be used for decryption.

5.1.14 key_block

5.1.14.1 Syntax

```
key_block
```

5.1.14.2 Description

ENCRYPTION INPUT: A **key_block** shall never be found in an input file unless it is contained within a previously generated **begin_protected**/**end_protected** block in which case it is ignored.

ENCRYPTION OUTPUT: The **key_block** pragma expression indicates that a key block begins on the next line in the file. An encrypting tool takes **data_method**, **data_keyname** and **data_keyowner** to form a text buffer. This buffer is then encrypted with the appropriate **key_method,,key_keyname** and **key_keyowner**. Then the encrypted region is be encoded. The output of this encoding shall be generated as the contents of the **key_block**.

Where more than one **key_block** pragma expression occurs within a single **begin**/**end** block, the generated key blocks shall all encode the same data decryption key data. Multiple key blocks are specified for the purpose of providing alternative decryption keys for a single decryption envelope.

DECRYPTION INPUT: The **key_block** is first read. The encoding is reversed and then the block internally decrypted. The resulting text can now be parsed to determine the keys required to decrypt the **data_block**. If for a **key_block** the specified key is not available, the tool should try the subsequent **key_block**s for availability.

6 Appendix A

6.1 Encryption/Decryption Flow

This section describes the various scenarios which can be used for IP Protection, and it also shows how to achieve the desired effect of securely protecting, distributing, and decrypting the model.

The data that needs to be protected from access or from unauthorized modification, should be placed in within the protect **begin**/**end** block. As the tool encrypts all the information in the **begin**/**end** block, the information is also protected.

6.2 Tool Vendor Secret key encryption system

In the secret key encryption system the key is tool vendor proprietary and will be embedded within the tool itself. The same key is used for both encryption and decryption. (In the EDA domain this is the simplest scenario and is roughly equivalent to the existing Verilog-XL `protect technique). It has the drawback of being completely tool vendor specific. Using this technique, the IP author can encrypt the IP and any IP consumer with appropriate licenses and the same tool vendor can utilize the IP.

If the key pragma are absent in the encryption block, the tool uses its internal key to encrypt the data block. As usual the session is specified by the data pragmas, i.e. data pragmas are specified the mentioned key is used, otherwise a random key is generated to encrypt the data.

6.3 Digital Envelopes

Editor's Note: This is the preferred exchange form in that it permits use of session keys to limit the amount of cipher text exposure for the exchanged encryption keys. The following text is incorrect in the assumption that asymmetric algorithms are the only useful exchange key mechanisms.

In this mechanism, each user will have a public and private key. The public key is made public while the private key remains secret. The sender encrypts the message using a symmetric key encryption algorithm, then encrypts the symmetric key using the recipient's public key. The recipient then decrypts the symmetric key using the appropriate private key and then decrypts the message with the symmetric key. In this way a fast encryption methods processes large amount of data, yet secret information is never transmitted without encryption. In digital envelopes, using the above encryption technology (secret key encryption system, where the key will be given by the IP author/end user), encryption tool will protect the IP. This symmetric key and algorithm information is them encrypted with a public key, the corresponding private key of which is available to the tool. So only the tool can decrypt the symmetric key internally and decrypt the protected IP.

Instead of using the public key of public/private key pair, a tool specific embedded key can also be used to encrypt the **key_block**. In this case also as only the tool knows its embedded key, only it can internally decrypt the design, hence the same effect can be achieved. The only disadvantage is that the tool's embedded key will have to be provided to the IP Author in some form.

The data_method and **data_keyowner**/**data_keyname** are used to encrypt the **data_block**. The encrypting tool then encrypts the **data_keyowner** and **data_keyname** pragmas with the **key_keymethod**/**key_keyname** and puts them in the **key_block** along with **data_method**. Alternatively if a dynamic session key is generated, the session key itself is encrypted along with the data method and put in the key block.

In the first approach the **data_keyowner**/**data_keyname** should also be present with the decrypting tool. No such dependency exists with the second approach as the key is present in the file itself.

For better security in the first approach the encrypting tool can actually read the **data_keyowner**/ **data_keyname** key and put it in the **key_block** as **data_decrypt_key**. Which not only will remove the dependency mentioned above, but will also protect against the hit & trial breaking of the **data_block** with the existing keys at the IP users end.