Verilog Programming Standards for Franzoids

9/19/94

1. Motivation

In order to promote understandability, quality, and maintainability of Verilog models developed during research or class activities, the following coding standards have been developed for students working under Dr. Paul Franzon. The standards are designed primarily for anyone who must understand your code; such as someone writing a program that must interface with yours, or someone helping to find bugs in your code, or the instructor who will decide your grade. They help everyone work more efficiently and reliably together since programs written by different people will have the same style, and thus will be easier to understand. But they are also for you, the programmer, too. They can help you to make decisions, reduce the opportunity for error, and give you the satisfaction and recognition of having written a clear and understandable program. Many of the ideas for these standards came from the book, *C Style: Standards and Guidelines; Defining Programming Standards for Professional C Programmers*, by David Straker (Prentice Hall, 1992).

Part of your grade for will come from how well you meet these standards.

2. General principles

1. *Think of the Reader*. The fundamental principle for writing good code is to always think of making the code readable for someone else. The ultimate test of the understandability of a piece of code is the ease with which someone else, chosen at random, can read and understand from that reading what the code is for, what it is doing, and why it is doing it the way it is. Sometime, the reader will be you, working with an old program or someone else’s work. Immediacy of understanding is important, as any delay can affect the contents of short-term memory, causing re-reading and possible error. So whenever you write a program always think first: will the reader easily understand what I mean and what the code is doing?

2. *Keep it Simple*. It is easy to write code which is concise and understood by the writer. But, with a little more thought, that code could be made much simpler. You must strike a balance between performance and simplicity. Style should only be sacrificed for speed in areas of known bottlenecks. Simple code is also easier to test.

3. *Be Explicit*. There are a number of features which let you say things implicitly. For example, the statement, `if ( yard )` implicitly compares yard to zero. A reader, however, might not find this immediately clear. What is really meant is, `if ( yard != 0 )`. You should always say precisely what you mean and avoid implicit usage.

4. *Be Consistent*. Changing the way things are done is a certain way to confuse the reader and could lead to a serious error. You should apply all elements of style consistently, including naming, commenting, layout, and language usage.
5. Minimize Scope. Always try to minimize unnecessary scope. Thus, if there is no reason to refer to an item in some place, then it should not be possible to reference it there. This applies to visual scope, where an item should be placed close to its usage, as well as logical scope.

3. Comments

Comments and identifier names together make up all the readable English in the code. Individually and collectively they describe the program and help the reader understand the meaning of the code.

3.1 General Commenting Principles

1. Be Clear and Concise. Comments should not be short and cryptic, nor should they be long and wordy. The art of commenting is to clearly and succinctly explain what is happening, and possibly why.

2. Make Every Comment Count. Comments should add value; they should not just restate the code. As a general rule shorter comments should paraphrase what is happening, with descriptions of how being put into longer comments.

3. Comments Must Be Correct. The code and comment should agree. A wrong comment is worse than no comment.

4. Make Comments Distinguishable From the Code. Use clear separation between the code and the comment.

    code statement;
    /*---- Comment -----------------------------------------*/

5. Use Different Characters to Emphasize Meaning.

    /* USE ALL CAPITALS FOR WARNINGS AND IMPORTANT COMMENTS */
    /* !! Exclamation marks warn !! */
    /* ?? Question marks indicate doubt ?? */
    /* ---- Solid lines delimit chunks of code ------------ */

3.2 Comment Types

1. Heading Comment. This is used to start a major item, such as a file or module. It helps the reader understand, navigate, and use the code beneath it. Use heading comments at the start of each file and at the start of all modules. Use the templates provided.
2. **Block Comment.** This is a small-scale heading comment used to describe a small section of code. A block comment typically occupies between one and five lines and describes either the code above it, summarizing the current status; or the code below it, detailing what is to happen; or a combination of both. Balance the comment size with the code size. In general, use one to three lines of comment for every five to ten line block of code. Indent block comments to the block which contains them. The format for a block comment is as follows:

```c
/*==== Equals for Major Subsections ==========================*/
/*---- Minus for Minor Subsections --------------------------*/
/*-------------------------------------------------------------*/
* Block comment format.
*-------------------------------------------------------------*/
```

3. **Trailing Comment.** This describes the action of one line of code. Trailing comments typically appear on the same line as the code they describe and are very brief. Use them in special situations rather than on every line. As a default start trailing comments in column 40. Trailing comments are especially useful in data declarations where they can be valuable in explaining the intent of a variable, and possibly the value range of the variable.

4. **Trailing vs. Block Comments.** Trailing comments are more difficult to use than block comments, as they must share line space with the code. They can also intrude into code space, making the code less immediately distinguishable. However, they save on vertical space and allow a more precise description of a code line. Block comments occupy more vertical space but are easily separated from the code and are thus often preferable.

Trailing comments should generally be used for situations where lines of code require specific clarification, such as around complex code, and to mark the end of blocks, rather than as a standard item for most lines.

Depictions of comment types and contents are shown in the code layout examples, section 7.3.

4. **Identifier Names**

Like comments, names must be clear, concise, and meaningful. A name, when read, should sound natural, and should help the reader understand its purpose without having to read the rest of the code. Long names can be very descriptive but unwieldy to use. Short names are concise but may be too cryptic to understand. Somewhere in between is a balance, where the meaning of a name is clear and easy to use. Use the following paradigm.

1. **Convention.** Start each word with an uppercase letter, and do not separate words: for example, WindSpeed. Since built-in primitives are lower-case, this convention makes it immediately clear whether a user defined module, primitive, function or task is being called, or a library primitive is being called.
2. Length. Identifier length should be between 5 and 20 characters. Use longer names for items with greater scope. Shorter names are acceptable for items of limited scope.

3. Short Names. Very short names can be used only with very limited scope (just a few lines), and where their usage is so obvious that a descriptive name is not needed. These names do not need to be capitalized. Reference the list below.

<table>
<thead>
<tr>
<th>Identifier</th>
<th>Use</th>
</tr>
</thead>
<tbody>
<tr>
<td>i, j, k</td>
<td>counter indices</td>
</tr>
<tr>
<td>x, y, z</td>
<td>coordinates</td>
</tr>
</tbody>
</table>

4. Abbreviations. Abbreviations are useful for reducing the length of a name, but can easily be over-used and reduce clarity. An abbreviated word should stand on its own; thus a good test is: can the abbreviated word be pronounced? A good abbreviation can be clearly understood both when looked at and when spoken. Here are some strategies for abbreviating.

a. Use the beginning of the word, e.g. Char, Para, Buf.
b. Remove vowels from the word, e.g. Chrctr, Prgrph, Bffr.
c. Retain major sounds of the word, e.g. Crctr, Pgph, Bufr.
d. Select critical letters of the word, e.g. Chr, Pgh, Bfr.
e. Use well-established names, e.g. c, Para, Buf.

When you use an abbreviation, use the same spelling of the abbreviation consistently throughout your program. Thus avoid, WindowHgt, with BoxHght.

5. Module and User Defined Primitive Names. Modules and Primitives represent a physical piece of the design. As such, a noun best labels these parts. Thus you might have an Adder or a Counter.

6. Function and Task Names. Functions and Tasks perform actions on things. Therefore, use a verb—noun scheme for naming them: for example, CreateNode( ). In complex situations the noun may not be sufficient to describe the thing. In this case, add an adjective to qualify the noun: for example, CreateFirstNode( ). In some cases adding a preposition to the name (i.e. by, in, for, from, to, with, between, etc.) may help to describe the function: for example, CreateNodeFor( NewData ).

7. Register or Net Names. Registers and Nets hold or pass information, so a noun is needed to describe the information held or passed. This gives rise to names like, OperandBus[31:0]. Sometimes an abstract noun, or noun—abstract- noun combination might be needed (an abstract noun describes something intangible). For example, an abstract noun identifier could be Count, and a noun—abstract-noun identifier could be CycleCount. Again, you can add adjectives for complex situations: for example, InstructionCycleCount.
8. **Replacement Items and Symbolic Constants.** Replacement items are those which cause replacement by the preprocessor, i.e. ‘defines, and symbolic constants are parameters. Use all capital letters for these items. Here are some examples.

```c
#define DIVIDE 'b001
```

```c
parameter EXECUTION_TIME = 5; // Clock cycles
```

9. **Names to Avoid.** When creating names in your program keep in mind a simple test. It should be possible to read your code over a telephone and for a listener, who knows the standard used, to write down the code correctly. You should, therefore, avoid the following.

a. Do not use names that sound alike, i.e. *Which & Witch*. There should be no alternate spelling possibilities for the names you choose.

b. Do not use names that look alike, such as those that vary in only one letter, such as *Last-Word and LostWord*.

c. Do not start variables with the underscore _.

d. Be careful with names that have numerals in them. Numerals can be confused with letters.

10. **File Names.** File names should end with .v, for example filename.v.

5. **Code Layout**

The way you physically set out the code can have a large impact on its readability. Layout is the most visible aspect of coding and hence it is the area which is often debated. Apply the following layout standards.

1. **Code Grouping and Size.** You should put pieces of a module’s code together in chunks separated by white space. There is a rule of nature which applies to grouping situations called the ‘rule of seven’. The rule says control is optimized at about seven pieces, and many groups of seven appear in nature. You should look for points to insert comments or blank lines at about every seven lines. The rule of seven is not an exact measure, however. Seven plus or minus two works well. Based on this, a module would contain no more than nine blocks of nine lines each, or 81 total lines of code. Use tasks to condense pieces of code.

2. **Line Size.** Put one action per line. This generally equates to one statement per line. If an action is longer than the line length, look for separate sub-actions to determine a logical wrap point.

3. **Line Wrapping.** If a statement is too long to fit on a single line indent the continuation on the next line, i.e.

```c
Register = PrimeFactor / ( CorrectionFactor + FindOffset( ) );
```

Always look for the most readable presentation when deciding the wrap point.
For a function or task call with many parameters, put one parameter per line. This clarifies the call and allows comments for each parameter. For example,

```vhdl
SelectMaterial( *MaterialName,
    Hardness, /* Rockwell number */
    Conductivity, /* Ohms per meter */
    MeltPoint ); /* Degrees F */
```

4. Spacing.

a. **Tab Length.** Use a 4 space indentation level.

b. **Object References.** Do not space between object references, such as an wire name and its array size, for example: Address[15:0].

c. **Unary Operators.** Do not space between a unary operator (!, ~, -, &, ~&, |, ~|, ^, ~^) and its operand. For example, ~StatusRegister. Do put a space on the side away from the operand to make the association of the operation clear.

d. **Binary Operators.** Put equal spaces on either side of a binary operator, such as *, /, %, +, -, ==, !=, &&, ||, <, etc. For example;

```vhdl
BlockArea = BlockLength * BlockHeight;
```

e. **Commas and Semicolons.** Do not put a space before a comma or semicolon, but do put one after it, just like in writing. For example,

```vhdl
for ( i = 0; i < 'MAX_NUMBER;  i = i + 1
```

f. **Parentheses.** In general, put a space after the beginning parentheses and before the closing parentheses (to make the contents stand out). In nested parentheses the inner sets need not be spaced to differentiate them from the outer set, but always space matching pairs equally. Do not nest more than three levels of parentheses.

g. Put a space between a keyword and its following parentheses, e.g.

```vhdl
if ( Source2 == 0 ) or @ ( posedge Clock )
```

h. Do not space between an instantiation of a module, primitive, function, or task name and its first parentheses since the parentheses is closely associated with the function. For example,

```vhdl
$display( "%d", Number ); Do put a space on the inside of the parentheses.
```

5. **Statement Groups.** Indent the begin - end or fork - join to match the block of statements.

```vhdl
initial
begin
    Status = 'FREE;
    if ( Source2 == 0 )
      fork
        CalculateResult;
        RequestOutputBus;
      join
    end
```
6. Heading Comments and File Templates

6.1 Header Files

It is useful in Verilog to create a file which contains common context (replacement or ‘define) items. This is particularly good for a group of engineers working on the same project. Use the following file heading comment at the start of each header file in you generate.

/* Header File **********************************************
 * FILE NAME:filename.v
 * DESCRIPTION:
 * Write a simple description of the purpose of this file.
 * USAGE:
 * Describe when this file should be ‘included.
 * NOTES:
 * Include any other help for the reader, including references.
 * REVISION HISTORY:
 * Date     Programmer    Description
 * 
 *H*/

/*------------------------------------------------------------
 * Conditional Inclusion. The following ‘ifdef allows this header to be referenced in multiple
 * modules, but only included once in a program that contains more than one module.
 *-----------------------------------------------------------

‘ifdef filename.v
// do nothing
‘else
‘define filename.v

/**== Included Files ========================================
 (Minimize the nesting of header files)
‘include ...

/**== Declarations ========================================
‘define ...

This may be subdivided into logical sections.

‘endif  //**** End of File filename.v ***************************/
6.2 Modules

Use the following heading comment at the start of each module. Include those subsections that apply.

/*Module ..............................................................................
 *  * NAME:
 *  *
 * DESCRIPTION: Simple description of the main purpose of the module.
 *  *
 * NOTES:
 *  *
 * REVISION HISTORY:
 *    Date    Programmer     Description
 *    M*/

//=== Context ===============================================*/
`include ...
`define ...

This may be subdivided into logical sections.

//=== Declarations ...........................................................*/

module ModuleName
  (Inputs,
   And,
   Outputs
  );

//=== Inputs ...............................................................*/
input Inputs;       // description

//=== Outputs .............................................................*/
output [31:0] Outputs;       // description

//=== Nets .................................................................*/
wire ...
tri ...
/*---- Registers ---------------------------------------------*/
reg ...

/*---- Integers ----------------------------------------------*/
integer ...

Use similar sections for time, real, parameter, and event declarations.

/*==== Initialization ========================================*/
initial
begin
    Set the variables to an appropriate starting value.
    end

/*==== Operation =============================================*/

/*==== Single line comment, Major point in the code ===============*/
(Indent to match the current block.)

Statements;

/*==== Single line comment, Minor point in the code ==============*/
(Indent to match the current block.)

Statements;

/* Block Comment. Comment on what has just happened and/or what is about to happen.
* Indent to match the current block.
*---------------------------------------------*/

Statements;

/*==== Debug =================================================*/

`ifdef DEBUG_ModuleName
$gr_waves( );
$gr_regs( );
`endif

endmodule //---- ModuleName ------------------------------------
6.3 Tasks, Functions, and Specify Blocks

Tasks, Functions, and Specify Blocks occur within a module and can be considered a major subsection of the module. Start each with the following block comments.

/**** Task ==============================================================
 *
 * NAME:
 *
 * DESCRIPTION:
 *
 *-------------------------------------------------------------*/

 task

 endtask //---- Name --------------------------------------------

/**** Function ==============================================================
 *
 * NAME:
 *
 * DESCRIPTION:
 *
 *-------------------------------------------------------------*/

 function

 endfunction //---- Name ----------------------------------------

/**** Specify Block ==============================================================
 *
 * NAME:
 *
 * DESCRIPTION:
 *
 *-------------------------------------------------------------*/

 specify

endspecify //---- Name -----------------------------------------
6.4 User Defined Primitives

User defined primitives are independent of modules and at the same level in the hierarchy as module definitions. Use the following heading comment at the beginning of each user defined primitive.

/* UDP ************************************************************
 * NAME:
 * DESCRIPTION: What function is being defined.
 * NOTES:
 * REVISION HISTORY:
 * Date     Programmer       Description
 *U*/

/***** Declarations -----------------------------------------------*/

primitive PrimitiveName

(Output,
 Input1,
 ...,
 Input10
);

/***** Output ------------------------------------------------------*/

output Output;
reg Output;

/***** Inputs ------------------------------------------------------*/

input Input1,
       Input10;       // description

/***** Initial State -----------------------------------------------*/

initial Output = InitialValue;
/*==== Truth Table ===============================*/
table
   // Label the Columns with a comment
endtable
endprimitive  //---- PrimitiveName -----------------------------
7. Constructs

Procedural Blocks

initial
begin
  sequential statements;
end

always @ ( expression )
begin : NamedBlock
  sequential statements;
end //---- NamedBlock ---------------------------

always wait ( expression )
fork
  parallel statements;
join

if (while and repeat use the same construction)
if ( expression )  statement;

if ( expression )
begin
  statement;
  statement;
end

if ( long expression which wraps onto the next line )
begin
  statement;
  statement;
end

if... else
if ( expression )
begin
  statement;
  statement;
end
else ( expression )
begin
statement;
statement;
end

if...else if...else
if ( expression )
begin
statement;
end
else if ( expression )
begin
statement;
statement;
end
else ( expression ) Always use a default else.
begin
statement;
end

Conditional
Variable = ( expression ) ? statement: statement;

Variable = ( expression ) ?
    statement:
    statement;

for
for ( expression; expression; expression )
begin
statement;
statement;
end

for ( expression;
    expression;
    expression )
begin
statement;
statement;
end
for ( long expression which wraps onto the next line; long expression which wraps onto the next line; long expression which wraps onto the next line )
begin
statement;
statement;
end

case, casex, casez

    case ( expression )

        value1 :
        begin
            statement;
            statement;
        end

        value2, value3 :
        begin
            statement;
            statement;
        end

        default : Always include a default.
        begin
            statement;
            statement;
        end

    endcase //---- expression ---------------------------

Instantiations.

    item InstanceName( parameter1, parameter2, parameter3 ); For calls with several parameters. Comment as needed.
Declarations

input VariableName1;  // One declaration per line.

reg [31:0] VariableName2;  // Preserve the vertical alignment
                           // .. on variable names.

/**----- Allowable values for VariableName2 ----------
 * Value1  - description
 * Value2  - description
 *-------------------------------------------------------------------*/

Put variable limitations close to their declaration.