Document Number	ENG-85857
Revision	B
Author	Jane Smith

Verilog Coding Guidelines

This document describes coding styles and guidelines for writing Verilog code for ASIC blocks and test benches.

Reviewers

Reviewer	**Name and Title**
ASIC Manager	Bob Parker, Mgr, Hardware Engineering

Modification History

Rev	Date	Originator	Comments
A	10/30/00	Jane Smith	Added test bench information on file name, reporting results
B	11/2/00	Jane Smith	Added document number and sign-off page

Table of Contents

1 Overview

In the course of projects, we are all constantly reviewing, maintaining, updating, or inheriting each other's code. To make these painful tasks a bit easier, the following coding guidelines have been identified.

These conventions also allow us to create tools to automate mundane parts of the coding process such as exploding module declarations or creating hierarchical modules.

One of the most important issues in reading code is style. Given this, the following style guidelines should be adhered to religiously.

There is a template for modules: $AIRO/asic/verilog/templates/module.v. The header by itself is in: $AIRO/asic/verilog/templates/header.v.

All of the parts are labeled. Code should be entered into the appropriate section. Comment headers are to be left in the code, even if there is no code entered in that section. It makes it easier for someone reading it to know there are no assign statements (as an example) if the header exists but has no code.

2 Alignment/Space

2.1 Tabs

All alignment related white space should be 4 position tab characters. All popular UNIX text editors can be made to use four position tabs instead of eight position tabs. The following UNIX command can be used to create an alias that will print Verilog using enscript.

```
alias printcode ‘/usr/5bin/pr -t -e4 \!$ | enscript -b\!$’
```

2.2 Text file width

Files must be restricted to 160 columns wide. These limits aid in creating readable printouts. The template is set up for 120 character width.

2.3 White space around operators

White space between operators can make code more readable. The exact spacing to leave is left as individual preference. One example to improve readability is shown below.

Incorrect Example:

```
    if((my_signal1==1’b0)&&(my_bus[3:0]==4’d5)) begin
```

Correct Example:

```
    if ((my_signal == 1’b0) && (my_bus[3:0] == 4’d5))
        begin
```

2.4 Nested indentation levels

Indentation levels are to be used to show code nesting. Case structures should use multiple indentation levels to line up action statements. Blank lines may be used as desired to improve code readability.. The Verilog keyword `begin` should appear on a line by itself so it can line up with its end statement. The begin/end block should always be used, even if there is only one statement. This makes adding lines of code much easier with fewer errors.

Incorrect Example:

```
    if ( this ) begin
        for ( i == 0; i < 10; i = i + 1 ) begin
        statement1;
    statement2;
        end
    statement3;

    statement4;
    end
    else
        statement5;
```

Correct Example:

```
if ( this )
    begin
        for ( i == 0; i < 10; i = i + 1 )
            begin
                statement1;
                statement2;
            end

        statement3;
        statement4;
    end

else
    begin
        statement5;
    end
```

Case statements are a little more complex. The `begin/end` structure should always be used in a case definition, indentation levels should be used to offset the statements that are encapsulated, but the use of blank lines can be used or omitted to best show the statement groupings.

Incorrect Example:

```
case ( myBus[3:0] )

    4'b0000 :    my_signal1 = TRUE;

    4'b0001 :    my_signal1 = FALSE;

    4'b0010 :
        begin
            my_signal1 = TRUE;
            my_signal2 = FALSE;
        end

    4'b0100 : my_signal2 = FALSE;

    default : my_signal1 = TRUE;

endcase
```

Correct Example:

```
case ( myBus[3:0] )

    4'b0000 :
        begin
            my_signal1 = TRUE;
        end

    4'b0001 :
        begin
            my_signal1 = FALSE;
        end

    4'b0010 :
        begin
            my_signal1 = TRUE;
            my_signal2 = FALSE;
        end

    4'b0100 :
        begin
            my_signal2 = FALSE;
        end

    default :
        begin
            my_signal1 = TRUE;
        end
endcase
```

2.5 Alignment

Aligning code statements can seem like a menial task, but it significantly adds to the understanding of code. Alignment should be used in declarations, assignments, multi-line statements, and end of line comments.

Incorrect Example:

```
reg[3:0] my_signal1;
reg[31:0] myDecodedSignal1;
reg[4:0] my_signal2, my_signal3, my_signal4;
wire[2:0] mySelect;
```

Correct Example:

```
//----------------------------------------------------------
// Signal Declarations: reg
//----------------------------------------------------------
reg     [3:0]       my_signal1;             //description
reg     [31:0]      my_decoded_signal1;     //description
reg     [4:0]       my_signal2,                     //description
reg                 my_signal3,             //description
reg                 my_signal4;                     //description

//----------------------------------------------------------
// Signal Declarations: wire
//----------------------------------------------------------
wire    [2:0]       mySelect;               //description
```

<u>Incorrect Example:</u>

```
    if ( myBoolean ) begin

         my_signal1 = TRUE;
         my_delayed_signal1 = !your_signal;
    end
```

<u>Correct Example:</u>

```
    if ( myBoolean )
         begin
              my_signal1                 = TRUE;
              my_delayed_signal1         = !your_signal;
         end
```

Some complex Boolean expressions make much more sense if they are expressed as multi-line aligned statements. The unaligned example is...

<u>Incorrect Example:</u>

```
    if ( ( my_signal1 && your_signal1 ) || ( my_signal2 && your_signal2 )
    ||( my_signal3 && your_signal3 ) ) begin
```

<u>Correct Example:</u>

```
    if ( ( my_signal1 && your_signal1 ) ||
          ( my_signal2 && your_signal2 ) ||
          ( my_signal3 && your_signal3 ) )
         begin
```

3 Naming conventions

3.1 Modules/Variables

All modules and signal names will be lower case, delimited by underscores "_".

Correct Example:

```
        module my_module (
            ena_fft,
            ena_mdi,
            fft_in,

            mdi_out,
            my_signal1,
            my_signal2
        );
```

All constant names should be upper case with underscore delimiters (**`MY_TRUE`**), with underscore delimiters every 4 characters (**`7'b00X_10XZ`**).

3.2 Special Case Variables

3.2.1 Clocks

Clocks have special naming requirements. These restrictions are placed to help with synthesis and back-end tools.

A clock signal has a prefix "clk", followed by a description, then a suffix.. The suffixes are "_drv", and "_src", If a signal starts with "clk", it must end with either "_drv" or "_src" where "_drv" indicates the signal is the output of a clock fanout tree and drives flip-flop clock inputs ONLY; "_src" indicates the signal is the input to a clock fanout ONLY."

Correct Example:

```
        module my_module (
            clk_a_drv,
            clk_mdi_src,
            clk_mdi_drv,
        );
```

Incorrect Example:

```
        module FftBlock
            clk_Fft,
            clkFft,
            fftClk_drv,
            my_Signal_src
```

3.2.2 Flip-Flops & Latches

Flip-flops also have special naming requirements to help synthesis and back-end CAE tools.

There are no prefixes, only the four suffixes: "_ff", "_nxt", (normal ffs), "_meta", and "_sync". (for special synchronizing ffs) The output of a flip-flop is named: name_ff, the value the flip-flop will be loaded with on the next clock is name_nxt. The signal name_ff becomes the Q output of the flip-flop while name_nxt is the D input. It helps readability of the combinatorial block to know that a signal is a flip-flop output or is to be loaded into a flip-flop. The flip-flop instance is named "name_ff_reg" by the synthesis tool.

(Add figure to show flip-flops)

The special suffixes, "_meta", and "_sync" are used when two flip-flops are put next to each other to synchronize a signal to a new clock. The first is named reg_meta, and its output can be metastable, the second flip-flop has reg_meta as an input and reg_sync as an output.

Latches use "_lat" for the output, and "_nxt" for the input.

Correct Example:

```
always @ ( posedge clkFft_drv or `RESET_EDGE reset_x )

    if ( reset_x = `RESET_ON )
        begin
            my_signal0_ff       <= 1'b0;
            my_signal1_ff       <= 1'b0;
            my_signal1_dup_ff   <= 1'b0;
            my_signal2_meta     <= 1'b0;
            my_signal2_sync     <= 1'b0;
        end

    else
        begin
            my_signal0_ff       <= my_signal0_nxt;
            my_signal1_ff       <= my_signal1_nxt;
            my_signal1_dup_ff   <= my_signal1_nxt;
            my_signal2_meta     <= async_input;
            my_signal2_sync     <= my_signal2_meta;
        end
end
```

3.2.3 Using Signal Source in Name

Understanding and synthesis can be improved by coding the name of the block that sources a signal into its name. An example, described below is the configuration bits in the PHY chips. If the source is known, it is a good idea to use it as a prefix to the name

The PHY ASICs have many registers that contain programmable setup information written by the MAC. These bits are static during the operation of the PHY. They are named with a prefix "cfg_ ". When a bit is named with this prefix, it can be synthesized with no timing constraints to the rest of the PHY circuit.

3.2.4 Active Low signals

All active low signals have the suffix "_n". If a clock is active low, the "_n" is before the "_drv" or "_src", such as clk_a_n_drv.

3.2.5 Active level globally defined

In some cases, the active level of a signal is defined globally. This is useful for resets or pad enables whose active level may be different for different libraries. Or there may be a different active level for FPGAs and the ASIC. The suffix “_x” is used, and a compare is always made to a constant that is globally defined for the design.

3.2.6 Resets

Resets are named special to allow for libraries with different senses (active high or active low). It is desirable to buffer the reset centrally rather than have it reset in every block. Otherwise, since the reset runs all over the chip, it could require a lot of buffering and therefore there could be a lot of skew on release of the reset from some flip-flops to others. This is controlled by buffering it in much the same way as a clock, in a centralized block. The problem is that different vendors have all of their flips-flops with either active high or active low reset (not both). We have had to use the same netlist for both FPGAs and ASICs and had to use different active levels for each. We handle this by naming the asynchronous reset “reset_x” and always comparing to a globally defined constant, “RESET_ON”. The active sense can then be changed for the whole design. Another define, called “RESET_EDGE” must be declared globally that is used in the sequential always block to indicate posedge or negedge.

If an active low reset is used, then a global file for the ASIC will have the following:

```
`define RESET_EDGE   negedge
`define RESET_ON     1’b0
```

For an active high reset, the global file will contain:

```
`define RESET_EDGE   posedge
`define RESET_ON     1’b1
```

and the flip-flop will be defined as (for either case)

```
always @ (posedge clk_drv or `RESET_EDGE reset_x)
    begin
        if (reset_x = `RESET_ON)
            begin
                reg1_ff  <= 1’b0;
                reg2_ff  <= 1’b0;
            end

        else
            begin
                reg1_ff  <= reg1_nxt;
                reg2_ff  <= reg2_nxt;
            end
    end
```

3.2.7 Summary of prefixes & suffixes for naming signals

Prefix	clk	Clock signal, must either drive a clock fan-out tree or be the driven signal from a fan-out tree that runs only to flip-flop clock inputs. Used with either “_drv” or “_src” suffix

	cfg	Configuration bit, written by the MAC and static during operation of the PHY
Suffix	_drv	Used only with “clk” prefix. Runs directly to flip-flop clock inputs only.
	_src	Used only with “clk” prefix. It runs only to a clock fan-out tree whose output has suffix “_drv”.”
	_ff	Output , Q, of a flip-flop
	_nxt	Input, D, of a flip-flop
	_n	Active low signal.
	_x	Used when sense is library dependent and is defined globally for the design (as in resets or I/O pad enables)

3.3 Signal widths

Never assign different width signals to each other.

Incorrect Example:

```
reg     [3:0]    my_ignal,
reg     [2:0]    your_signal;

my_signal = your_signal;.
```

Correct Example:

```
reg     [3:0]    my_signal
reg     [2:0]    your_signal;

my_signal[2:0] = your_signal;
```

Along these lines, always completely specify the width of constants. For instance

Incorrect Example:

```
reg     [3:0]    my_signal

my_signal= my_signal+ 1’d1;
```

This will work, but a more specific declaration would be

Correct Example:

```
reg     [3:0]    my_signal

my_signal = my_signal + 4’d1;
```

4 Module Coding

Each module has sections for parameters, ports, signal declarations, assign statements, assign statements, instantiations, sequential logic, and combinatorial logic. They are organized as in the template file $AIRO/asic/verilog/templates/module.v.

Each module will have separate sections for sequential and combinatorial logic. Sequential and combinatorial logic will not be combined in the same always block.

4.1 Module declaration

The module declaration consists of the module name, the IO list, and the input/output declarations. The module name should reside on a line by itself and the opening parenthesis. The IO list should be as inputs in alphabetical order, outputs in alphabetical order, and bidirects in alphabetical order, with a blank line between each group. Only one signal/vector should be listed on each line. Any changes to the module IO list should be integrated in the same fashion.

Correct Example:

```
module my_module (
    //-----------------------
    my_input1,
    my_input2        [2:0],
    my_input3,

    my_output1       [31:0],
    my_output2,

    my_inOut
);
```

The module input/output declaration is a simple cut and paste of the IO list in the module declaration. These declarations should also be one per line in alphabetical order. Notice the alignment in the following example. Make sure the parameter list is also alphabetical, especially since they can be re-declared in the instantiation by position only.

Correct Example:

```
//-----------------------
// Parameters
//-----------------------
parameter CONST1 = 1'b0;
parameter CONST2 = 3'b010;

//-----------------------
// Input Ports
//-----------------------
input                 my_input1;
input [2:0]           my_input2;
input                 my_input3;

//-----------------------
// Output Ports
```

```
//----------------------
output [31:0]        my_output1;
output               my_output2;
//----------------------
// Bidirectional Ports
//----------------------
inpout               my_inout;
```

4.2 Module instantiations

Modules will be instantiated with their ports connected by name, rather than position. Signals should be in the same order as in the module declaration (alphabetical by inputs, outputs, in/outs). There should be one signal per line. If you want to instantiate **my_module** whose ports are named *my_signal1* and *my_signal*2...

Correct Example:

```
my_module U1 (
    .my_signal1( current_signal1 ),
    .my_signal2( current_signal2 )
);.
```

4.3 Sequential Logic

Sequential logic blocks generate the flip-flops in a design. There should be no other logic generated inside the sequential block. No delays should ever be coded into synthesizable logic, including the sequential logic block. The sequential block is always coded with non-blocking statements. There should never be any blocking statements.

The sequential block is an always block with the clock (and possibly an asynchronous reset) in the sensitivity list. The flip-flops are defined as shown using variable names: name_ff, name_nxt, possibly name_meta and name_sync, or possibly an input port. Flip-flops can be set or cleared with the asynchronous reset. They should not be initialized if not required.

Correct Example:

```
always @ ( posedge clk_drv or `RESET_EDGE reset_x )
    begin
        if reset_x = `RESET_ON)
            begin
                reg1_ff   <= 3'b000;
                reg2_ff   <= 1'b0;
                reg3_ff   <= INIT_CONST;
            end

        else
            begin
                reg1_ff   <= reg1_nxt;
                reg2_ff   <= reg2_nxt;
                reg3_ff   <= reg3_nxt;
                reg4_ff   <= reg4_nxt;
                reg5_meta <= input_a;
                reg5_sync <= reg5_meta;
            end
    end
end.
```

4.4 Combinatorial Logic

The combinatorial logic block is used to code all of the non-sequential logic. It is all of the logic except the flip-flops. It generates the D inputs to the flip-flops which will end up as gates and uses the Q outputs of the flip-flops for other logical information.

The combinatorial block uses blocking statements only. No delays are allowed in synthesizable logic,

Correct Example:

```
always @ ( input_a or input_b or state_ff or reg3_ff)
    begin
        case ( state_ff )
            IDLE :
                begin
                    state_nxt   = WRITE;
                    output1     = input_a;
                end
            WRITE:
            .
            .
            .
        endcase
    end
```

4.5 Defines

Defines are not used in normal module coding. They may be used occasionally to set up some global variables in a top-level file. They should not be in a normal module.

4.6 Assignment statements

Assign statements are included in the template, but are not commonly used. They may be used to assign a flip-flop output to an output port, but all combinatorial logic is done in the combinatorial logic block.

Correct Example:

```
assign outputA    = reg2_ff;
```

Incorrect Example:

```
    Assign signal1 = signal2 ^ signal3;
```

5 Comments

Liberal use of comments is strongly encouraged. Adding obvious comments is discouraged. Basically, extensive comments that proceed blocks of code, coupled with sparse back references, guides the reader through the code. Clever methods of implementation should **ALWAYS** be highlighted with comments, especially if the reader might question the abilities of synthesis tools.

Incorrect Example:

```
// assert myReq1
my_req1 = TRUE;

// wait for the acknowledge
if ( my_ack ) begin

    { my_select[27:1], temp } = 1 << my_channel[4:0];

end
```

With a little different commenting strategy, more important information can be imparted to the reader.

Correct Example:

```
//
// Request that my_module grant access to the RAM. When the
// acknowledge comes, decode the channel into access
// selects.
//
// The decoder is coded as a left shift operator with a
// dummy initial position. Believe it or not, this syntax
// actually synthesizes correctly.
//

my_req1 = TRUE;

if ( my_ack )
    begin
        // shift based decoder
        {my_select[27:1], temp} = 1 << my_channel[4:0];
    end
```

5.1 End of line comments

End of line comments are not usually appropriate, but, if they are used, all of the comments within a section should be aligned.

Incorrect Example:

```
my_signal1 = your_signal1; // advance the signals,

if ( your_signal2 && !your_signal3 ) begin // check 2 and 3,

    my_signal2 = TRUE; // then assert request
```

If the comments are aligned, they make much more sense (like reading a story).

Correct Example:

```
      my_signal1 = your_signal1;                    // Advance the signals,

      if ( your_signal2 && !your_signal3 )          // check 2 and 3,
          begin
              my_signal2 = TRUE;                    // then assert request
          end
```

6 Test Benches

6.1 File Name

The file name for a test bench will be *module_tb.v*, where module matches the hdl top-level name to be tested.

6.2 Self-Checking Test Bench Results

Test benches that are self-checking will generate an output file in the ./output directory named module_tb.rpt. It will contain the following case-sensitive words:

ERROR	will be contained in all error messages
PASS	will be contained in all messages indicating test ran correctly
WARNING	can be used for non-fatal messages

This will make it easier to post-process files and put together summaries of testing. It will be helpful to add the module/test case to error messages.