"Any fool can write code that a computer can understand, good programmers write code that humans can understand. So, its very important to learn standardized coding skill"

Generalized coding style:

Though bad coding style does not stop your code from working, it does make it harder for others to understand and makes it more difficult to maintain. Take pride in writing well-ordered and uniformly formatted code.

Good coding practices include:

- **Version control**: Allows developers to work on the same files at the same time, and to restore previous versions of the code
- **Readability**: Code that is easy to read is easier to optimize and follow
- **Code review**: Helps to ensure that developers adhere to standards, and that the code is double checked
- **Error handling**: Helps to catch errors in the code before they cause a failure
- **Line spacing**: Using blank lines sparingly to separate different parts of the code
- **Descriptive names**: Using names that describe what a variable or function does
- **Simplicity**: Keeping the code as simple as possible, and avoiding adding unnecessary features
- **Naming conventions**: Being intentional about how variables, constants, methods, classes, and interfaces are named

Guideline1: Indent your code with spaces

Use a consistent number of spaces to indent your code every time you start a new nested block . 2 or 3 spaces is recommended. Do not use tabs since the tab settings vary in different editors and viewers and your formatting may not look as you intended. Many text editors have an indenting mode that automatically replaces tabs with a defined number of spaces.

Guideline2: Only one statement per line

Only have one declaration or statement per line. This makes the code clearer and easier to understand and debug.

Recommended:

```
// Variable definition:
logic enable;
logic completed;
logic in_progress;
 // Statements:
// (See next Guideline for the use of begin-end pairs
// with conditional statements)
if(enable==0)
  in_progress=1;
else
  in_progress=0;
```

Not Recommended:

```
// Variable definition:
logic enable, completed, in_progress;
 // Statements:
if(enable==0)in_progress=1;else in_progress=0;
```

Guideline3: Use a begin-end pair to bracket conditional statements

This helps make it clear where the conditional code begins and where it ends. Without a begin-end pair, only the first line after the conditional statement is executed conditionally and this is a common source of errors.

Recommended:

```
// Both statements executed conditionally:
if(i>0) begin
  count= current_count;
  target= current_target;
end
```

Not Recommended:

```
if(i>0)
  count= current_count;
  target= current_target; // This statement is executed unconditionally
```

Guideline4: Use parenthesis in Boolean conditions

This makes the code easier to read and avoids mistakes due to operator precedence issues.

Recommended:

```
// Boolean or conditional expression
if((A==B) &&(B>(C*2)) ||(B>((D**2) +1))) begin
  ...
end
```

Not Recommended:

```
// Boolean or conditional expression
If (A==B && B> C*2|| B> D**2+1) begin
  ...
end
```

Guideline5: Keep your code simple

Avoid writing tricky and hard to understand code, keep it simple so that it is clear what it does and how so that others can quickly understand it in case a modification is required.

Guideline6: Keep your lines to a reasonable length

Long lines are difficult to read and understand, especially if you need to scroll the editor to the right to read the end of the line. As a guideline, keep your line length to around 80 characters, break the line and indent at logical places.

Recommended:

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  mbus_seq_item rhs_;
 if(!$cast(rhs_, rhs))begin
     `uvm_error("do_compare","cast failed, check type compatibility")
    return0;
 end
do_compare=super.do_compare(rhs, comparer)&&(MADDR== rhs_.MADDR)&&(MWDATA== rhs_.MWDATA)&&(MREAD== rhs_.MREAD)&&(MOPCODE== rhs_.MOPCODE)&&(MPHASE== rhs_.MPHASE)&&(MRESP== rhs_.MRESP)&& (MRDATA== rhs_.MRDATA);
endfunction: do_compare
```

Not Recommended:

```
function bit do_compare(uvm_object rhs, uvm_comparer comparer);
  mbus_seq_item rhs_;
 if(!$cast(rhs_, rhs))begin
     `uvm_error("do_compare","cast failed, check type compatibility")
    return0;
 end
do_compare=super.do_compare(rhs, comparer)&&
          (MADDR== rhs_.MADDR)&&
          (MWDATA== rhs_.MWDATA)&&
          (MREAD== rhs_.MREAD)&&
          (MOPCODE== rhs_.MOPCODE)&&
          (MPHASE== rhs_.MPHASE)&&
          (MRESP== rhs_.MRESP)&&
          (MRDATA== rhs_.MRDATA);
endfunction: do_compare
```

Guideline7: Use lowercase for names, using underscores to separate fields

This makes it clearer what the name is, as opposed to other naming styles such as PeripheralComp which are harder to read.

Recommended: pcie_axi_scoreboard

Not Recommended: PcieAxiScoreboard

Guideline8: Use prefix_ and _postfix to delineate name types

Use prefixes and postfixes for name types to help differentiate between variables. Pre and post fixes for some common variable types are summarized in the following table:

prefix/postfix	Purpose
_t	Used for a type created via a typedef
_e	Used to indicate a enumerated type
_h	Used for a class handle
_m	Used for a protected class member
_cfg	Used for a configuration object handle
_ap	Used for an analysis port handle
_group	Used for a cover group handle

Guideline9: Use a descriptive typedef when declaring a variable instead of a built-in type

This makes the code clearer and easier to understand as well as easier to maintain. An exception is when the built-in type keyword best describes the purpose of the variable's type.

```
// Descriptive typedef for a 24 bit data sample:

typedef bit[23:0] data_sample_t;
```

Guideline10: Use the end label for classes, functions, tasks, and packages

This forces the compiler to check that the name of the item matches the end label which can trap cut and paste errors. It is also useful to a person reading the code.

```
// Using end labels

package my_pkg;

 //...

 class my_class;

 // ...

   Function void my_function();

   //...

   endfunction: my_function

 task my_task;

   // ...

   endtask: my_task

 endclass: my_class

 endpackage: my_pkg
```

Guideline11: Comment the intent of your code

Add comments to define the intent of your code, don't rely on the users interpretation. For instance, each method in a class should have a comment block that specifies its input arguments, its function and its return arguments. This principle can be extended to automatically generate html documentation for your code using documentation tools such as Natural Docs.

SystemVerilog Do's

- Use a consistent coding style - see guidelines
- Use a descriptive typedef for variables
- Use an end label for methods, classes and packages
- Use `includes to compile classes into packages
- Define classes within packages
- Define one class per file
- Only `include a file in one package
- Import packages to reference their contents
- Check that $cast() calls complete successfully
- Check that randomize() calls complete successfully
- Use if rather than assert to check the status of method calls
- Wrap covergroups in class objects
- Only sample covergroups using the sample() method
- Label covergroup coverpoints and crosses

SystemVerilog Don'ts

- Avoid `including the same class in multiple locations
- Avoid placing code in $unit
- Avoid using associative arrays with a wildcard index
- Avoid using #0 delays
- Don't rely on static initialization order

File use Guideline:

Guideline12: Use lower case for file and directory names. Lower case names are easier to type.

Guideline13: Use .sv extensions for compile files , .svh for `include files. The convention of using the .sv extension for files that are compiled and .svh for files that get included makes it easier to sort through files in a directory and also to write compilation scripts. For instance, a package definition would have a .sv extension, but would reference `included .svh files:

Guideline14: `include .svh class files should only contain one class and be named after that class. This makes it easier to maintain the code, since it is obvious where the code is for each class.

Guideline15: Use descriptive names that reflect functionality. File names should match their content. The names should be descriptive and use postfixes to help describe the intent . Example: _pkg, _env, _agent, etc.

Guideline16: (`include versus import) Only use `include to include a file in one place. The `include construct should only be used to include a file in just one place. `include is typically used to include .svh files when creating a package file. If you need to reference a type or other definition, then use 'import' to bring the definition into scope. Do not use `include. The reason for this is that type definitions are scope specific. A type defined in two scopes using the same `include file are not recognized as being the same. If the type is defined in one place, inside a package, then it can be properly referenced by importing that package. An exception to this would be a macro definition file such as the 'uvm_macros.svh' file.

Directory use Guideline:

Testbenches are constructed of system verilog UVM code organized as packages, collections of verification IP organized as packages and a description of the hardware to be tested. Other files such as C models and documentation may also be required. Packages should be organized in a hierarchy.

Guideline17: Each package should have its own directory. Each package should exist in its own directory. Each of these package directories should have one file that gets compiled - a file with the extension .sv . Each package should have at most one file that may be included in other code. This file may define macros.

pcie_pkg.sv

pcie_macros.svh

For a complex package (such as a UVC) that may contain tests, examples and documentation, create subdirectories:

pcie_pkg/examples

pcie_pkg/docs

pcie_pkg/tests

pcie_pkg/src/pcie_pkg.sv

Guidelines to write class , methods in SV:

Guideline18: Name classes after the functionality they encapsulate use classes to encapsulate related functionality. Name the class after the functionality, for instance a scoreboard for an pcie IP would be named "pcie_ip_scoreboard".

Guideline19: Private class members should have a m_ prefix. Any member that is meant to be private should be named with a 'm_' prefix, and should be made local or protected. Any member that will be randomized should not be local or protected.

Guideline20: Declare class methods using extern. This means that the class body contains the method prototypes and so users only have to look at this section of the class definition to understand its functionality.

```
// Descriptive typedefs:

typedef logic[63:0] raw_sample_t;

typedef logic[15:0] processed_sample_t

 // Class definition illustrating the use of externally defined methods:

class audio_compress;

rand int iteration_limit;

rand bit valid;

rand raw_sample_t raw_audio_sample;

rand processed_sample_t processed_sample;
```

```
 // function: new
// Constructor - initializes valid
extern function new();
 // function: compress_sample
// Applies compression algorithm to raw sample
// inputs: none
// returns: void
extern function void compress_sample();
 // function: set_new_sample
// Set a new raw sample value
// inputs:
//   raw_sample_t new_sample
// returns: void
extern function void set_new_sample(raw_sample_t new_sample);
 endclass: audio_compress
 function audio_compress::new();
  valid=0;
  iteration_limit= $bits(processed_sample_t);
endfunction
 function void audio_compress::compress_sample();
 for(int i=0; i< iteration_limit; i++)begin
    processed_sample[i]= raw_audio_sample[((i*2)-1):(i*2)];
 end
  valid=1;
endfunction: compress_sample
 function void audio_compress:set_new_sample(raw_sample_t new_sample);
  raw_audio_sample= new_sample;
  valid=0;
endfunction: set_new_sample
```