



OpenCores.Org

OpenCores Coding Guidelines

Authors: Yair Amitay

yair.amitay@flextronicssemi.com

Jamil Khatib

khatib@opencores.org

Damjan Lampret

lampret@opencores.org

Rev. 1.0

October 24, 2001

This page has been intentionally left blank.

Revision History

Rev.	Date	Author	Description
0.1	15/05/01	Yair Amitay	First Draft
0.2	29/05/01	Jamil Khatib	VHDL and Verilog notes are split. Major sections reorganization. Comments from discussions on emails are added.
0.3	07/06/01	Jamil Khatib	Revision history added. Dedicated clock and reset pins added. OpenCores logo added.
0.4	28/7/01	Damjan Lampret	Switched to latest OC document template. Added new introduction chapter. Reorganized and updated old content. Added feedback from Rudi Usselmann, Don Reid, Illan Glasner, David Kessner.
0.5	22/10/01	Damjan Lampret	Incorporated feedback from Illan Glasner, David Kessner, Yair Amitay and Lior Shtram. Added I/O ports table.
1.0	24/10/01	Damjan Lampret	Fixed some typing errors. Added Blue Beaver's comment about tri-state. First official version.

Contents

INTRODUCTION.....	1
BEFORE YOU START	2
2.1. SPECIFICATION DOCUMENT	2
2.2. DESIGN DOCUMENT	2
2.3. CVS AND TEAM WORK.....	3
2.4. VERIFICATION	3
2.5. DIRECTORY STRUCTURE.....	3
GENERAL DESIGN GUIDELINES.....	6
3.1. GENERAL	6
3.2. RESET	6
3.3. CLOCKS.....	7
3.4. BUSES	8
3.5. TRI-STATE	8
3.6. MEMORIES	9
3.7. CODING FOR SYNTHESIS	9
3.8. CORE I/O PORTS.....	10
VERILOG GUIDELINES.....	11
4.1. GENERAL	11
4.2. CODING FOR SYNTHESIS	12
4.3. CODING FOR SIMULATION AND DEBUGGING	12
4.4. FILE HEADER	13
VHDL GUIDELINES	15
5.1. GENERAL	15
5.2. CODING FOR SYNTHESIS	17
5.3. CODING FOR SIMULATION AND DEBUGGING	19
5.4. FILE HEADER	19

1

Introduction

This document contains guidelines and recommendations for HDL coding. Adopting these guidelines will reduce the amount of time required to get high quality IP cores and will reduce possibilities for functional problems. Following these guidelines will improve reusability and readability of the code.

The guidelines are sorted according to main subjects, but most of them are related to other subjects as well. Each guideline is placed in the section where its influence is major, but it can have a marked impact on other sections as well.

The guidelines are of different importance and are classified in the following way:

Good practice - signifies a guideline that is common good practice and should be used in most cases. This means that in some cases there are specific problems that violate this guideline.

Recommendation - signifies a guideline that is recommended. It is uncommon that a problem cannot be solved without violating this guideline. You should read it as a SHOULD rule.

Strong recommendation - signifies a hard guideline, this should be used in all situations unless a very good reason exists to violate it. You should read it as a MUST rule.

This document will change in the future. Anyone is encouraged to make changes or contribute additional content. Latest document is available from OpenCores CVS using module name [common](#).

TO DO list:

- Sections on Code Style, Comment Style, and Module Naming (there are certain guidelines)

2

Before You Start

2.1. Specification Document

Before you jump into HDL coding, try to check existing cores and write a specification document. This will have several advantages:

- clear definition what the core should do and which standards will be supported
- defines profiles of developers for formation of a team

Essentially the core is a black box, and the specification documentation should only be concerned with the interface to this black box. Anyone wishing to use the core should only have to read the specification document while those wishing to modify or add to the core should read design document as well.

For specification document you should use [Specification Template](#) available from OpenCores CVS. At the time of this writing only MS Word Template is available.

2.2. Design Document

While you are coding HDL, try to write design document. If team is working on a core, design document might have to be written before HDL coding begins so that developed blocks will be able to work together without spending too much time on integration. Design document is important because:

- better understanding how the core's internal blocks should work and communicate to each other
- allows work of a team on different parts of the core
- allows future development and contribution by others
- simplifies verification and bug fixing

2.3. CVS and Team Work

Try to share development efforts with others. This way you do not have to do anything yourself and results will come sooner. Also we are doing this for fun and part of fun is also communication with others and team solving problems.

CVS is central OpenCores resource for development and final source storage. Even if you work alone, try to use CVS as much as possible. Do not wait until your design is stable – CVS is meant for development. If you check-in changes on your source file regularly, you can most effectively use advantages of CVS such as comparing two different version of the same file. However for efficient CVS use we recommend that you first spend some time and familiar yourself with it by reading <http://www.opencores.org/cvs.shtml> and CVS manual that is available there.

Once your design is stable CVS will allow others to most effectively download the latest stable version (while you are working on checked-in development version) and send you testing feedback.

Additionally we are integrating Verilog and VHDL Linter tool that will check designs committed to the CVS and in a matter of minutes send a lint report to the developer. Lint rules will be based on guidelines found in this document.

2.4. Verification

As part of an early design stage you will also have to think thoroughly about verification strategy. If you are unfamiliar with verification, try to read [Verification Strategies](#) document.

If your design uses recommended WISHBONE SOC interconnect bus, your next step is to download WISHBONE models. At the time of writing there are several WISHBONE models in OpenCores CVS written both in Verilog as well as in VHDL.

2.5. Directory structure

To simplify integration of various cores into SOC, try to use recommended directory structure.

block_name/	Top level directory of a core
backend/	Top level backend directory
<vendor>/	Vendor specific floorplan, place and route directory structure
sim/	Top level simulations directory
rtl_sim/	RTL simulations
bin/	RTL simulation scripts
run/	For running RTL simulations
src/	Special sources for RTL simulations

	out/	Dump and other useful output from RTL simulation
	log/	Log files
gate_sim/		Gate-level simulations
	bin/	Gate-level simulation scripts
	run/	For running gate-level simulations
	src/	Special sources for gate-level simulations
	out/	Dump and other useful output from gate-level simulation
	log/	Log files
syn/		Synthesis
	<vendor>/	Each synthesis tool has separate directory
	bin/	For synthesis scripts
	run/	For running synthesis scripts
	src/	Special sources for synthesis
	out/	For generated netlists (Synopsys db, verilog)
	log/	Log files (including reports)
lint/		Lint
	bin/	Lint scripts
	run/	For running linter
	out/	Lint report
	log/	Log files
fv/		Formal verification
lib/		Vendor target libraries
rtl/		RTL sources
	verilog/	For verilog sources
	vhdl/	For VHDL sources
bench/		Bench sources
	verilog/	For verilog sources
	vhdl/	For VHDL sources
doc/		Put specification, design and other PDF documents here
	src/	Source version of all documents (Word, StarOffice, Frame Maker)
sw/		Put sources for utilities or software test cases here

If your core requires additional directories, try to add them by following conventions in the suggested directory structure. For example it is very common that *sw* will require several subdirectories.

Subdirectory *lib* should contain vendor target libraries. For example for a standard cell ASIC with a hard block SRAM, this directory should contain two subdirectories. Each subdirectory should contain complete set of library files for front- and backend design process (behavioral models, timing models, LVS netlists, layout abstracts, GDSII layouts). For FPGA at least behavioral models of FPGA primitives should be included here.

Directory structure for *backend* is not precisely defined because it is out of scope of this document. Usually for FPGA backend you will have FPGA vendor specific subdirectory structure with several revisions of mapping, floorplan, place and route. For ASIC subdirectory structure will usually consists of subdirectories *pre_p&r*, *post_p&r*, *post_scan* etc.

3

General Design Guidelines

3.1. General

- 3.1.1. Strong Recommendation: Write descriptive comments. Try to make a habit to comment every assignment or block.**

You will make life much easier for someone who would like to add additional functionality or fix a bug. Not to mention it is good for you as well if you try to change the code after a few weeks.

- 3.1.2. Recommendation: If your core is complex and has several submodules in hierarchy, it is recommended that top level module is for connectivity only without any logic.**

Makes design cleaner and gives an instant insight what are major blocks. Also try to bring all memories and other hard blocks on top level.

If you need some glue logic, create separate module for glue logic.

- 3.1.3. Good Practice: Keep the same signal name through different hierarchies.**

Tracing a signal will be easier. Enables easy netlist debugging

- 3.1.4. Good Practice: Try not to mix active low and active high logic in your core. Stick just to one. Preferred is active high.**

Reduces confusion.

3.2. Reset

Reset makes a design more deterministic and easier to verify. It prevents reaching prohibited states in state-machine at power-up.

- 3.2.1. Recommendation: Make asynchronous active high reset for all flip-flops. Such reset must be synchronized with the clock on SOC top level and it does not require a multiplexer at every flip-flop data input.**

Asynchronous reset will make your core smaller since multiplexer used by synchronous reset, is not required. However it must be handled in a special way not unlike synchronous reset that is already constrained by the clock. Synchronous reset must be synchronized with the clock in SOC top level and reset tree will be built by the backend in case of the ASIC.

- 3.2.2. Recommendation: At reset time, all bi-directional ports should be in input state.**

Scan expects this and it prevents X values.

3.3. Clocks

- 3.3.1. Strong Recommendation: Signals that cross different clock domains should be sampled before and after crossing domains (double sampling is a MUST). See synchronizer_flop in OpenCores CVS in module [common](#).**

Prevents meta-stability state.

To make netlist verification easier, you should use one module (i.e. sync.v, sync.vhd) that will have in, out and clock interface and the first flip-flop should have a unique name as this flip-flop will have timing violation. If it has unique name, it is easier to trace it and "change" it to not pass X's.

Also it should be clear that you pass ONLY the control signal and not the data bus etc.

- 3.3.2. Recommendation: Do not use gated clocks unless you have thorough understanding what is the proper way to implement clock gating and what are the consequences for testing and verification.**

Usually the system integrator and the backend are responsible for clock gating. If target application is required to operate in low power, clock gating can be a powerful feature to achieve that. If low power is not required, explicit clock gating in RTL can cause much longer development because backend must eliminate possibilities for glitches in the clock.

More proper way instead of explicit clock gating in RTL is to use clock enables. If you use clock enables, certain EDA tools such as Synopsys Power Compiler (ASIC) can be used to transform a design with clock enables into a design with gated clocks. This way target application that does not require low power operation and can still use your core without dealing with clock gating problems in explicit RTL clock gating.

- 3.3.3. Recommendation: Do not use clocks or reset as data or as enables. Do not use data as clocks or as resets.**

Synthesis results may differ from RTL. Higher chances for timing verification problems.

In certain cases you might need to use clocks/resets as data or data as clocks/resets. In such a case provide two signals. For example clk and clk_data, where clk drives flops' clock inputs and clk_data drives combinatorial logic.

3.3.4. Good practice: Use minimum number of clock domains per core.

For example, a UART only needs one clock domain - not two or three - to properly function.

3.4. Buses

3.4.1. Strong Recommendation: Compare buses with the same width.

Buses must be of equal width so that comparison works properly.

3.4.2. Recommendation: Start buses with bit index 0.

Some tools don't support buses that don't start with bit index 0.

3.4.3. Recommendation: Use MSB to LSB convention. Bit 0 is LSB.

This is to avoid misinterpretation through the design hierarchy.

3.4.4. Recommendation: Try to design with minimum number interconnecting wires on core interfaces. Do not make buses wider than it is necessary. If possible make data bus narrower and increase address bus width instead.

Lack of routing resources can cause serious problems in the backend and it can affect both timing and area.

3.4.5. Recommendation: Use WISHBONE SOC interconnect bus to connect your core to other OpenCores' cores.

OpenCores selected WISHBONE SOC interconnect as our SOC interconnect. Most our new cores support WISHBONE. To get more information about WISHBONE and to find out why WISHBONE is the only truly free and open source SOC bus, see <http://www.opencores.org/wishbone/>.

3.5. Tri-State

3.5.1. Recommendation: Generally avoid using internal tri-state signals. However for internal monitors tri-state is recommended.

Generally tri-state increases power consumption. It also makes the backend tuning more difficult.

However in certain cases such as in case of internal bus monitors, tri-state implementation might result in much smaller monitor than multiplexer

implementation. But using tri-state monitors with scan can create complications since only one tri-state driver can be enabled and this must be considered when testing the design with scan.

3.6. Memories

- 3.6.1. Recommendation: Use synchronous single-port or dual-port generic memory blocks such as generic_spram and generic_dpram. These blocks already support several ASIC memory vendors as well as several different FPGA vendors. They are in OpenCores CVS under module [common](#).**

This will automatically mean that your design supports several ASIC and FPGA memories and that you do not have to deal with various kinds of memories to support various target technologies. Simply enable the target vendor and link with his target library.

Also using synchronous memories instead of asynchronous memories might allow you to meet timing constraints easier.

3.7. Coding for Synthesis

- 3.7.1. Strong Recommendation: Use synchronous design practice.**

It avoids problems with synthesis, timing verification and in simulation.

- 3.7.2. Strong Recommendation: Do not use delay elements.**

It causes synthesis problems and timing verification problems.

If you use delay elements, you MUST consider worst and best case timing and not be happy with the delay in nominal case. This will make your core reuse unfriendly since it will have to be characterized for every target technology/process.

- 3.7.3. Recommendation: All core's external IOs should be registered.**

It prevents long timing paths and allows you to meet timing constraints easier. It also allows easier verification of the entire SOC

However in certain case you cannot register outputs such as in case of certain PCI output signals.

- 3.7.4. Recommendation: Avoid using latches.**

It causes synthesis problems and timing verification problems.

- 3.7.5. Good Practice: Avoid using flip-flops with negative edge clock.**

Might cause ASIC synthesis problems and timing verification problems.

3.7.6. Good Practice: Core's internal interfaces should be sampled.

This is a design issue however it is recommended in most cases.

3.8. Core I/O ports

3.8.1. Recommendation: Name core's ports by following conventions from Table 1. This simplifies the SOC integration process and backend process and allows automation.

Port	Description
*_i	Core's input port
*_o	Core's output port
*_io	Core's bi-directional port
*_clk_i	Core's clock input port
*_clk_o	Core's clock output port
*_rst_i	Core's reset input port
*_rst_o	Core's reset output port
wb?*_i	Core's WISHBONE input port, ? is optional single letter
wb?*_o	Core's WISHBONE output port, ? is optional single letter
*_pad_i	Core's input port connected to input pad's output
*_pad_o	Core's output port connected to output pad's input
*_padoe_o	Core's output port connected to tri-state pad's output enable
*_clk_pad_i	Core's clock input port connected to clock input pad's output
*_clk_pad_o	Core's clock output port connected to clock output pad's input
*_rst_pad_i	Core's reset input port connected to clock input pad's output
*_rst_pad_o	Core's reset output port connected to clock output pad's input

Table 1. Core I/O ports

*Do not use any other abbreviation except *_clk_* and *_rst_* to mark clock and reset signals. For example do not ***reset*** or ***clock*** etc.*

3.8.2. Recommendation: Use *n to mark active low signals. Do not use *_.

*Using *_ to mark active low signals is possible in Verilog but not in VHDL. Design that uses *_ in Verilog cannot be directly translated into VHDL without not changing the port name.*

4

Verilog Guidelines

4.1. General

- 4.1.1. Recommendation: Try not to use ``include` command. Instead load all files as modules or load them as libraries (`-y -v`).**

`include might have problems with certain tools. If you use them, they should be environment independent.

- 4.1.2. Recommendation: Use non-blocking assignment (`<=#1`) in synchronous process, and blocking assignment (`=`) in asynchronous process.**

Synopsys expects this format. Makes the simulation respond more deterministically.

- 4.1.3. Recommendation: If possible, use parameters instead of definitions (``define`).**

Global definitions cause a lot of trouble when cores from different sources are combined (unless very strict naming conventions are followed).

Also some tools have problems with ``define`, ``ifdef` or ``undef`.

- 4.1.4. Recommendation: Put all definitions (``define`) that cannot be changed into parameters, into one global file.**

Definitions should start with the name of the core to distinguish them from other global definitions pertaining to other cores used in SOC.

- 4.1.5. Good Practice: Try to write one module in one file. The filename should be the same as the module name. Module name should be composed out of the block name and local module name.**

To prevent confusion when debugging an SOC, filename and module name should start with block name and followed by actual local module name.

For example UART design is be composed out of TX unit and RX unit. Module names should be `uart_tx`, `uart_rx` and `uart_top`. Filenames should be `uart_tx.v`, `uart_rx.v` and `uart_top.v`.

- 4.1.6. Good Practice: Try to use instantiation by name (explicit instantiation) and not by place.**

It requires more typing, but makes easier debugging and understanding the code.

- 4.1.7. Recommendation: Use lower case letters for all identifiers. Use upper case letters for definitions ('define').**

Mixing EDA tools that are case sensitive and those that are case insensitive causes problems. Following recommendation not to use upper case letters for identifiers (signal names, port names, module names etc) will avoid EDA tools' problems.

Definitions should use upper case letters only to distinguish them from identifiers.

4.2. Coding for Synthesis

- 4.2.1. Strong Recommendation: Do not use statements such as 'assign #X a = b;' or '#x;' where X is a number of time units of delay.**

These statements are meant primarily for simulation only. For flip-flop models it is recommended that it is modeled with delay unit of 1. Example always q <= #1 d;

- 4.2.2. Good practice: Do not use statements that assign initial values to signals and variables (wire b=1'b0;).**

4.3. Coding for Simulation and Debugging

- 4.3.1. Strong Recommendation: All system tasks for simulation should be contained in a separate file from the core source code.**

I.e. monitors etc.

- 4.3.2. Good Practice: Create a separate timescale.v file, put `timescale command in it and include this file in all RTL source code files. Include command should be wrapped with // synopsys translate_off and // synopsys translate_on directives.**
- 4.3.3. Good Practice: Try to write '%m' in 'display' command (shows the instance name).**

4.4. File Header

- 4.4.1. Recommendation: Use our standard header at the beginning of each file. The header is available from the OpenCores CVS under module name [common](#).**

The header contains basic information about the project, file in question, author(s), license agreement, OpenCores and CVS log.

Default license agreement is GNU LGPL, which allows unrestricted use and at the same time protects author's rights. Complete GNU LGPL license agreement text is available at <http://www.opencores.org/lgpl.shtml>.

CVS log tag is updated automatically whenever file is checked-in to the CVS.

```

////////////////////////////////////
////                                     ////
//// WISHBONE XXX IP Core               ////
////                                     ////
//// This file is part of the XXX project ////
//// http://www.opencores.org/cores/xxx/  ////
////                                     ////
//// Description                         ////
//// Implementation of XXX IP core according to ////
//// XXX IP core specification document.  ////
////                                     ////
//// To Do:                             ////
//// -                                   ////
////                                     ////
//// Author(s):                         ////
//// - First & Last Name, email@opencores.org ////
////                                     ////
////////////////////////////////////
////                                     ////
//// Copyright (C) 2001 Authors and OPENCORES.ORG ////
////                                     ////
//// This source file may be used and distributed without ////

```

```
//// restriction provided that this copyright statement is not  ////
//// removed from the file and that any derivative work contains  ////
//// the original copyright notice and the associated disclaimer.  ////
////                                                                ////
//// This source file is free software; you can redistribute it  ////
//// and/or modify it under the terms of the GNU Lesser General  ////
//// Public License as published by the Free Software Foundation;  ////
//// either version 2.1 of the License, or (at your option) any  ////
//// later version.                                             ////
////                                                                ////
//// This source is distributed in the hope that it will be     ////
//// useful, but WITHOUT ANY WARRANTY; without even the implied  ////
//// warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR    ////
//// PURPOSE. See the GNU Lesser General Public License for more  ////
//// details.                                                  ////
////                                                                ////
//// You should have received a copy of the GNU Lesser General  ////
//// Public License along with this source; if not, download it  ////
//// from http://www.opencores.org/lgpl.shtml                  ////
////                                                                ////
////////////////////////////////////
//
// CVS Revision History
//
// $Log$
//
```

5

VHDL Guidelines

5.1. General

5.1.1. Strong recommendation: Use `std_logic` type for external ports type.

5.1.2. Strong recommendation: Do not assign value of unknown 'x' or check for do not care '-'.

Such values can produce unexpected behavior in both simulation and synthesis.

5.1.3. Strong recommendation: Do not use default values (or initialization) for signals and variables. Use reset to initialize all signals and variables.

Such assignment can cause mismatch between synthesis and simulation.

5.1.4. Strong recommendation: Do not use ports with type `buffer` to read output value within the code. Instead use `out` type and add another variable or signal and assign to it the same output value.

This is because `buffer` type ports can not be connected to other types of ports and so `buffer` type will propagate throughout the entire design ports.

```
PROCESS (CLK, RST_n)
variable out_var : std_logic;
BEGIN -- PROCESS
  IF RST_n = '0' THEN
    Outsignal <= '0';
    out_var <'0';
    outsign2 <= '0';
  ELSIF CLK'event AND CLK = '1' THEN
    Outsign2 <= out_var; -- the same as Outsignal
    out_var := input1 and input2;
    Outsignal <= input1 and input2;
  END IF;
END PROCESS;
```

- 5.1.5. **Recommendation:** Define components and constants for each core in a single package.
- 5.1.6. **Good Practice:** Do not mix between VHDL coding standards for the whole project (i.e. do not mix between VHDL 87 and VHDL 93 constructs).
- 5.1.7. **Good Practice:** Try to write one VHDL design unit in one file. The filename should be the same as the unit name. For example entities and architectures are placed in separate files, the same applies for package and package bodies.
- 5.1.8. **Good Practice:** Try to use instantiation by name (explicit instantiation) and not by place.

For easier debugging and understanding the code.

```
wb_if: wb
  PORT MAP (
    CLK => CLK_I,
    RST_I => RST_I_I,
    ACK_O => ACK_O_I,
    ADR_I => ADR_I_I,
    CYC_I => CYC_I_I,
    DAT_I => DAT_I_I,
    DAT_O => DAT_O_I,
    RTY_O => RTY_O_I,
    STB_I => STB_I_I,
    WE_I => WE_I_I);
```

Inside the core is sometimes permissible to use instantiation by place since it decrease amount of typing by a significant margin.

5.1.9. Good Practice: Try to use configuration to map entities, architectures and components (i.e. to define such mapping explicitly).

So tracing changing between different architectures can be simple in a single file. This can be useful to change simulation from high level to low level architectures

5.1.10. Good Practice: Try to compile each block in a separate library.

5.1.11. Good Practice: Make use of constants and generics for buffer sizes, bus width and all other unit parameters.

This provides more readability and reusability of the code.

5.2. Coding for Synthesis

5.2.1. Strong recommendation: Read variables then write to them (Read before write) unless you know what you are doing.

If variables are written then read, long combination logic and latches (or registers) will be generated. This come from the fact that variables get their value immediately and not like signals.

```
PROCESS (CLK, RST_n)
Variable out_var : std_logic;
BEGIN -- PROCESS
  IF RST_n = '0' THEN
    out_var <'0';
    outsign2 <= '0';
  ELSIF CLK'event AND CLK = '1' THEN
    Outsign2 <= out_var; -- read
    out_var := input1 and input2; -- write
  END IF;
END PROCESS;
```

- 5.2.2. Strong recommendation: Include all signals that are read inside the combination process to its sensitivity list. (i.e. Signals on Right Hand Side RHS of signal assignments or conditions.**

This is to prevent inferring of unwanted latches.

- 5.2.3. Recommendation: Avoid using long if-then-else statements and use case statement instead.**

This is to prevent inferring of large priority decoders and makes the code easier to be read.

- 5.2.4. Strong Recommendation: Do not use statements such as '(b <= a after X ns)' OR 'wait for X ns;' where X is a number of time units of delay.**

These statements are meant for simulation only.

- 5.2.5. Strong Recommendation: Do not use statements that assign initial values to signals and variables (variable B:INTEGER:=0;).**

These statements are meant for simulation only.

- 5.2.6. Recommended: Try to write clock enable as in the below figure within a single clocked process and do not use two different processes one clocked (registers) and one for combinational logic.**

This is because some synthesis tools detects CE operation and map it to CE of FF if it already has. Otherwise CE pin will not be used and external logic will be inferred. This is a common practice for FPGA code.

```
PROCESS (CLK, RST_n)
BEGIN -- PROCESS
  IF RST_n = '0' THEN
    Outsignal <= '0';
  ELSIF CLK'event AND CLK = '1' THEN
    IF (CE = '1') THEN
      Outsignal <= '1';
    END IF;
  END IF;
END PROCESS;
```

- 5.2.7. Good Practice: Try to write fsm in two processes one for sequential assignments (registers) and the other for combinational logic**

This provides more readability and prediction of combinational logic size.

5.3. Coding for Simulation and Debugging

- 5.3.1. Good Practice: Try to write test bench in two parts, one for data generation and checking and one for timing bus interface protocol generation and checking.**

This is to isolate data (results checking) from bus handshake checking and to make it simpler to change the handshake protocol while keeping the same internal logic.

5.4. File Header

- 5.4.1. Recommendation: Use our standard header at the beginning of each file.**

The header contains basic information about the project, file in question, author(s), license agreement, OpenCores and CVS log.

Default license agreement is GNU LGPL which allows unrestricted use and at the same time protects author's rights. Complete GNU LGPL license agreement text is available at <http://www.opencores.org/lgpl.shtml>.

CVS log tag is updated automatically whenever file is checked-in to the CVS.

```
-----
----
---- WISHBONE XXX IP Core
----
```



```
-- $Log$  
--
```