OpenCores HDL modeling guidelines

This document describes the OpenCores HDL modelling guidelines with some examples

Brought to You By OpenCores

Legal Notices and Disclaimers

Copyright Notice

General Disclaimer

The Publisher has strived to be as accurate and complete as possible in the creation of this ebook, notwithstanding the fact that he does not warrant or represent at any time that the contents within are accurate due to the rapidly changing nature of information.

The Publisher will not be responsible for any losses or damages of any kind incurred by the reader whether directly or indirectly arising from the use of the information found in this ebook.

This ebook is not intended for use as a source of legal, business, accounting, financial, or medical advice. All readers are advised to seek services of competent professionals in the legal, business, accounting, finance, and medical fields.

No guarantees of any kind are made. Reader assumes responsibility for use of the information contained herein. The Publisher reserves the right to make changes without notice. The Publisher assumes no responsibility or liability whatsoever on the behalf of the reader of this report.

Distribution Rights

The Publisher grants you the following rights for re-distribution of this ebook.

[YES] Can be given away.
[YES] Can be packaged.
[YES] Can be offered as a bonus.
[NO] Can be edited completely and your name put on it.
[YES] Can be used as web content.
[NO] Can be broken down into smaller articles.
[NO] Can be added to an e-course or auto-responder as content.
[NO] Can be submitted to article directories (even YOURS) IF at least half is rewritten!
[NO] Can be added to paid membership sites.
[NO] Can be added to an ebook/PDF as content.
[NO] Can be offered through auction sites.
[NO] Can sell Resale Rights.
[NO] Can sell Master Resale Rights.
[NO] Can sell Private Label Rights.

Table of Contents

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.

Before you start

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.

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

Subversion (SVN) 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.

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

Once your design is stable SVN 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.

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 SVN repository written both in Verilog as well as in VHDL.

Directory structure

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

blockname	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)
+- rtl	RTL sources
| +- verilog	For verilog sources
| +- vhdl	For VHDL sources
+- bench	Bench sources
| +- verilog	For verilog sources
| +- vhdl	For VHDL sources
+- doc	ut specification, design and other PDF documents here
| +- src	Source version of all documents (Open Office, 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.

In order to provide VATS (Automated Verification System) all the needed data, script file run_sim must be provided in sim/rtl_sim/bin/. VATS will call this script with “-r” parameter to check if design is working against regression test when CPUs are idling, usually overnight. The test will pass only if the last line is “OK”. The rest of the output is ignored by VATS, but it should be as informative as possible, in order to track possible errors.

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.

General design guidelines

General

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.

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

   Tracing a signal will be easier. Enables easy netlist debugging

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.

Reset

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

1. **Recommendation:** Use asynchronous active high reset.

   Using asynchronous reset could result in a smaller core. Using an active high reset makes the core compatible with wishbone spec.

2. ***R*ecommendation:** At reset time, all bi-directional ports should be in input state.

   Scan expects this and it prevents X values.

Clocks

1. **Strong Recommendation:** Signals that cross different clock domains should be double sampled after crossing domains (double sampling is a MUST).

   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.

2. **Recommendation:** Do not use gated clocks unless you have thorough knowledge about the proper way to implement clock gating and 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. ***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.

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 function properly.

Buses

1. **Strong Recommendation:** Compare buses with the same width.

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

2. **Recommendation:** Start buses with bit index 0.

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

3. **Recommendation:** Use MSB to LSB convention. Bit 0 is LSB.

   This is to avoid misinterpretation through the design hierarchy.

4. **Recommendation:** Try to design with a minimum number of interconnecting wires on core interfaces. Do not make buses wider than 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.

5. **Recommendation:** Use WISHBONE SoC Interconnect.

   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 SoC bus, see http://www.opencores.org/?do=wishbone.

Tri-State

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.

Memories

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.

Coding for synthesis

1. **Strong Recommendation:** Use synchronous design practice.

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

2. **Strong Recommendation:** Do not use delay elements.

   It causes synthesis 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. **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.

4. **Recommendation:** Avoid using latches.

   It causes synthesis problems and timing verification problems.

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

   Might cause ASIC synthesis problems and timing verification problems.

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

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

Core I/O ports

1. **Recommendation:** Name core's ports by following conventions from Table . 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 lett
*_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

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

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. Designs that use *_ in Verilog cannot be directly translated into VHDL without changing the port names.

Verilog guidelines

General

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.

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.

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. **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.

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.

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.

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.

Coding for synthesis

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;

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

Coding for simulation and debugging

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

   I.e. monitors etc.

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.
3. **Good Practice:** Try to write '%m' in 'display' command (shows the instance name).

File header

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.

```
//////////////////////////////////////////////////////////////////// ////
////                                                                ////
////  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) 2009 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                       ////
////                                                                //// ///
//////////////////////////////////////////////////////////////////////////
```

VHDL guidelines

General

1. **Strong recommendation:** Use std_logic type for external ports.
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.
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.
4. **Strong recommendation:** Do not use buffer type ports to read output values within the code. Instead use type out 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, causing the buffer type to propagate throughout the entire design.

   ```
   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. **Recommendation:** Define components and constants for each core in a single package.
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).
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.
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.

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 .

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

Coding for synthesis

```
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;
```

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

   This is to prevent inferring of unwanted latches.

2. **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.

3. **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.

4. ***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. **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;
```

6. **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.

Coding for simulation and debugging

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.

File header

1. ***Recommedation:*** *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.

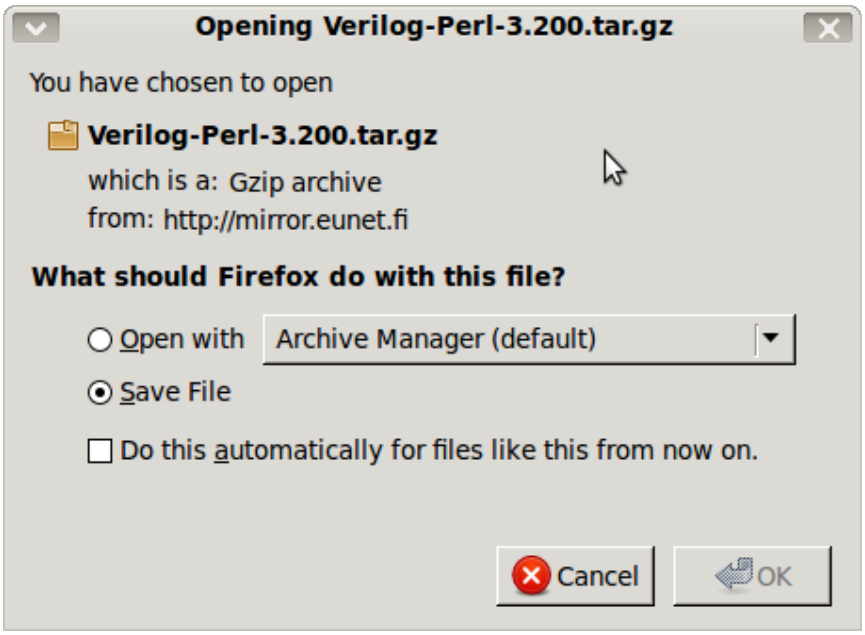
```
------------------------------------------------------------------
----                                                          ----
----  WISHBONE XXX IP Core                                    ----
----                                                          ----
----  This file is part of the 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) 2009 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                 ----
----                                                          ----
------------------------------------------------------------------
```

Preprocessors

Verilog HDL includes a preprocessor similar to the one found in the C programming language. With preprocessor directives it is possible to include or exclude text in a powerful way. The preprocessing is the first step when compiling Verilog HDL. There is also the possibility to run a preprocessor as a stand alone tool prior to Verilog HDL or VHDL compilation. Some preprocessors add functionality as compared to what is normally found in Verilog HDL.

vppreproc - Preprocess Verilog code using verilog-perl

http://search.cpan.org/~wsnyder/Verilog-Perl-3.200/vppreproc



Manual Installation

Download the latest version from <http://www.perl.org/CPAN/>, or from <http://www.veripool.org/verilog-perl>.

"cd" to the directory containing this README notice.

tar xvzf Verilog-Perl-3.200.tar.gz

cd Verilog-Perl-3.200/

Type "perl Makefile.PL" to configure Verilog for your system.

Type "make" to compile Verilog. Some Solaris users have had trouble with "open" being redefined. If this happens, try editing the Makefile to change _FILE_OFFSET_BITS to 32 instead of 64.

Type "make test" to check the package. If you don't have Synopsys' VCS, the test will print a warning, which you can ignore.

Type "make install" to install the programs and any data files and documentation.

Note:

Previous releases of vppreproc were called vppp
Adding a link in /usr/local/bin gives you backward compatibility

Verilog preprocessor

VBPP is a Verilog preprocessor. It has support for most Verilog preprocessing directives and additional directives such as:

- Statement generator ('generate' command in VHDL).
- Expression evaluation.
- Mathematical functions: log2, ceil, floor, round, abs, etc.
- Conditionals: if, switch, etc.

To install in Debian and Ubuntu:

```
sudo apt-get install vbpp
```

```
syntax: vpp [options] filename
options:
  +incdir+...+ Search directory for `include files.
  -D           Define macro.
  -E           Perform C style preprocessing.
  -L           Output `line directive.
  -h           Print help message and exit.
  -q           Suppress status message.
```

Examples

The us of this preprocessor adds some new macros to be used for generic modeling of HDL.

If statement

```
`let x=5
`if (x == 5)
  note that this should get expanded
`else
  you should not see this
`endif
```

For statement

```
`for (i=0; i<4; i++)
a[`i] = `i;
`endfor
```

While statement

```
`let i=10
`while (i>5)
i=`i
`let i=i-1;
`endwhile
```

Switch statement

```
`for (i=0; i<5; i++)
`switch(i)
`case 0
```

```
print out 0
`breaksw
`case 1
print out 1
`breaksw
`case 2
print out 2
`breaksw
`default
print out twice
`breaksw
`endswitch
`endfor
```

Modular design

Always try to identify parts of your design to implement as a standalone IP. This makes reuse of modules easier and eases verification.

Typical examples of reusable modules includes:

1. memories
2. arithmetic functions
3. CRC calculators

When designing IP for reuse adoption to new use cases is important. This can be done in basically two ways, both in Verilog HDL and VHDL

1. generic models with parameters (Verilog HDL) or as generics (VHDL)
2. preprocessor directives
   built-in in Verilog HDL
   requires external preprocessor in VHDL

A generic module can be modified per instance. Modification includes setting length of vectors, reset state of vectors etc. You can not include or exclude top level signals (you can use user defined datatypes in VHDL). With parameters/generics it is possible to adopt a module without changing the implementation. Every instance can have specific functions.

With a preprocessor you can do the following:

1. include or exclude module/entity signals
2. include or exclude text (HDL code)
3. have optional behavior dependent on use case

Example of typical uses for preprocessor macros

> Configuration parameters as CPU accessible registers or as constants. A design targeted for an ASIC might require the possibility to change SDRAM settings after manufacturing for a FPGA implementation the registers might be replaced with constants to lower area requirements.
>
> Optional functionality in a processor might include MMU, cache, etc. This can be done preprocessor text modifications.

When using preprocessor to generate different implementation from a common code base it is important to give unique naming of modules. Suppose we write a generic CRC calculator. With preprocessor commands we could be able to chose between different polynomial to generate checksums. From that code base it is possible to generate a checksum calculator for en Ethernet MAC and USB controller. If these were to be used in the same SoC the module name most be unique.

Module name in non hierarchical designs

Good design practice is to use a file with user defines. Use a define for the module naming

In module_define.v

```
`define MODULE_NAME crc16
```

In design file

```
module `MODULE_NAME
```

Module names in hierarchical designs

For a hierarchical design the situation is more complicated. We want to define a base name that might be used for the top level module and the rest of the modules should have a name where the base name is combined with a sub name.

In module_define.v we define the base name as before

```
`define MODULE_NAME sd_spi
```

The toplevel module should have only the base name. In top_level file:

```
module `MODULE_NAME
```

Design might include a prescaler that have an option to set the divisor with a define. This sub module must have a unique name to be distinguished from other possible prescalers in the same SoC design. We want to append "_prescaler" to the module name.

Alternative 1

We could use a second define in the define file

```
`define PRESCALER _prescaler
```

In prescaler module declaration

```
module `MODULE_NAME`PRESCALER
```

Alternative 2

With Verilog preprocessor vpp a different approach might be used. We have support for string concatenation.

In prescaler module declaration

```
module `MODULE_NAME::_prescaler
```

With the use of a standalone preprocessor both of this methods applies to both Verilog HDL and VHDL implementations.

Build environment

A typical SoC design is made up from a large number of design files. There are a few things we can do to make it easier to run design in different simulation and synthesis tools.

1. Apply preprocessor defines early to avoid dependencies to library and include files
2. concatenate multiple files into one
3. generate variants from common codebase

A suitable application to use for this is make with a project specific Makefile. Application make is available in many platforms including cygwin, Linux, Unix.

The Makefile is a textfile that easily can be extended to include generate statement of variants of implementations from a common code base.

Place this Makefile in directory rtl/verilog or rtl/vhdl.

Makefiles could also be used to remove temporary files, compile sources etc.

Example: variants from common code base

The OpenCores projects Ethernet SMII implements a low pin count interface towards external Ethernet PHY devices.

With a generic design we implement interface for 1,2,3,4 and 8 channels. The make script generates the following:

1. module instance files
   in top level design use
   `include “smii_module_inst_n.v”
2. applies preprocessor commands and concatenates design files for generic design and ACTEL specific with included global clock drivers

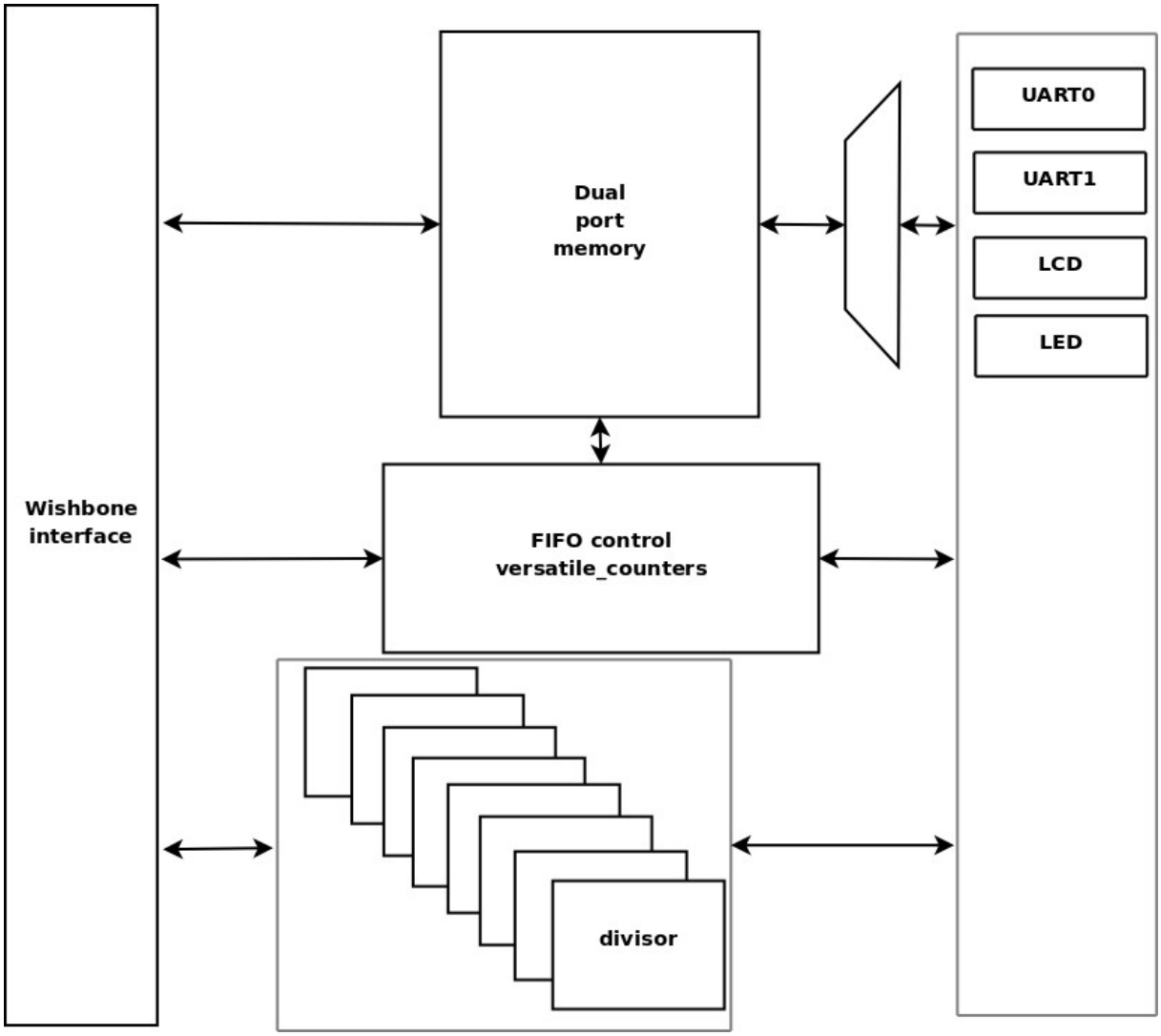
Makefile:

```
comp1:
	vpp -DSMII=1 smii_module_inst.v > tmp.v
	vppp --simple +define+SMII+1 tmp.v > smii_module_inst_1.v

comp2:
	vpp -DSMII=2 smii_module_inst.v > tmp.v
	vppp --simple +define+SMII+2 tmp.v > smii_module_inst_2.v

comp3:
	vpp -DSMII=3 smii_module_inst.v > tmp.v
	vppp --simple +define+SMII+3 tmp.v > smii_module_inst_3.v

comp4:
	vpp -DSMII=4 smii_module_inst.v > tmp.v
	vppp --simple +define+SMII+4 tmp.v > smii_module_inst_4.v

comp8:
	vpp -DSMII=8 smii_module_inst.v > tmp.v
	vppp --simple +define+SMII+8 tmp.v > smii_module_inst_8.v

smii:
	vppp --simple +define+ACTEL generic_buffers.v smii_sync.v smii_txrx.v | cat
copyright.v - > smii_ACTEL.v
	vppp --simple generic_buffers.v smii_sync.v smii_txrx.v | cat copyright.v -
> smii.v

all: comp1 comp2 comp3 comp4 comp8 smii
```

Use case: Versatile IO

Versatile IO is a design to be used for various types of low to medium bitrate IO including

- UART
- IR
- LED control
- LCD character display



The basic idea behind this IP is that multiple, 8 by default, IO functions share a dual port memory used as multiple FIFO channels. On the system side all IO functions share a common wishbone interface.

From the system side this IP appears as 8 individual 16550 compatible UARTs for easy interfacing from software.

This design includes submodules that could be reused and therefor should be implemented as standalone IPs. The following functions are identified:

1. generic counter
2. generic FIFO

Versatile Counter

A highly configurable counter is implemented as a standalone module. In this design the counter is used for two different purposes

1. LFSR counters used as address generators for FIFO pointers
2. Binary up/down counters used for FIFO flags

Versatile counter is available from OpenCores:

http://www.opencores.org/?do=project&who=versatile_counter

The same counter can also be used where the divisor is selected as a static divider.

Versatile FIFO

A true dual port RAM in combination with the versatile counter make up a FIFO with multiple channels.

The FIFO supports both synchronous and asynchronous implementations.

For this particular design a synchronous FIFO is used.

Versatile FIFO is available from OpenCores:

http://www.opencores.org/?do=project&who=versatile_fifo

Reuse of submodules

The use of ready made configurable submodules

- lower the risk of introducing functional errors in the design
- enhance the chance of optimal usage of target technology
  (in this case it is the responsibility of Versatile_FIFO to make efficient use of built-in memory resources)
- shortens design time

Revision history

Rev	Date	Author	Description
2.0	04/17/09	Michael Unneback	Added text about preprocessing, build environment Example designs
1.2	14/07/02	Marko Mlinar	More on directory and file organization
1.1	13/08/02	Damjan Lampret	Added feedback from Richard Herveille.
1.0	24/10/01	Damjan Lampret	Fixed some typing errors. Added Blue Beaver's comment about tri-state. First official version.
0.5	22/10/01	Damjan Lampret	Incorporated feedback from Illan Glasner, David Kessner, Yair Amitay and Lior Shtram. Added I/O ports table.
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.
0.3	07/06/01	Jamil Khatib	Revision history added. Dedicated clock and reset pins added. OpenCores logo added.
0.2	29/05/01	Jamil Khatib	VHDL and Verilog notes are split. Major sections reorganization. Comments from discussions on emails are added.
0.1	15/05/01	Yair Amitay	First Draft

Recommended Resources

ORSoC – http://www.orsoc.se

ORSoC is a fabless ASIC design & manufacturing services company, providing RTL to ASIC design services and silicon fabrication service. **ORSoC** are specialists building complex system based on the OpenRISC processor platform.

Open Source IP – http://www.opencores.org

Your number one source for open source IP and other FPGA/ASIC related information.