Fossi-Foundation EDA Standards

libreChainEDA

Coding and Packaging Guidelines

3 Sep 2016

Abstract

This document sets forth the coding standards and packaging requirements for all hardware IP used by libreChainEDA tool flows.

Revision Controlled Repository.

The master copy for each component will be kept in a repository under revision control. If that repository is not accessible to the end user via the internet then the master copy should be mirrored from a location that is accessible.

All files needed for all variants and versions of each component IP must be stored inside their repository. No files used by any other components are allowed in any other components repositories. Shared code must be split off and stored as a separate component.

A component repository will contain a one and only one component. If a repository contains more than one componennt then it is a library repository. Each and every component within a library repository must be stored under a unique subdirectory that contains every file needed for that component and no files needed for any other component.

Packaging Standard.

All IP must be packaged in compliance with IEEE 1685-2014 (IP-Xact). Each repository will have one and only one IP-Xact catalog file. Each repository will have a unique identifier as determined by the IP-Xact's VLN.

All IP-Xact files within each repository will share the same values for Vendor, Library and name. The IP-Xact catalog file will use “index” for its version.

IP-Xact does not impose any file or directory naming requirements for any thing within a component repository. There are no naming requirements for the repositories name or how it is stored with other repositories within a library. The only naming requirement is that the vendor name must be a valid URL that is owned by the vendor. A directory with this name may be stored in the same subdirectory

as other vendors directories with no chance of a naming collision.

The smartest thing that the spirit consortium did when they created IP-Xact was to realize that nobody in this industry was ever going to change their legacy designs to meet any ones new standard so they decided to support all legacy packages. The standard does not tell you where to put or how to name your files, it tells you how to document where you decided to place them and under what name.

Module Names.

Each SOC design team must decide how they will manage the module name space for their chip. They will create a algorithm that when given the VLNV of any component will return a unique module name for that component.

Component designers will not have access to this name and so must provide a means so that the SOC design team can override the default module name. This can be done by setting a `define variable to the default name inside of a `nifdef . The SOC design team can then set the module name to any value they choose.

Signal Names.

Component designers who are creating leaf level or flat designs are free to use adhoc names for all of their signals and ports. They are responsible for managing the name space inside of their module.

Component designers who are creating hierarchical designs must use buses for all of their port signals. This is because SOC designers have the final say on signal names in their designs and they do not like to see chips that have 9 different ways to spell “clock”. Signals created by buses are easier to rename using IP-XACT bus extensions.

All signal names are created out of four possible items:

1) Adhoc name for a signal

2) Bus name

3) SubMember name of a bus member

4) Instance name of the nodes driving instance.

There are 24 possible ways to arrange these four items so we have to deal with the problem of 24 endians. The selected solution is instance_adhoc_bus_submember for all signals.

Any of the four fields may be dropped if it turns out to be redundant and not needed to uniquely identify that signal.

`Macro Names.

The `define macro name space  is global and there is no way that it can be managed by component designers before delivering their code. Each component should include a tool flow that will run their code through a preprocessor before it is loaded into the SOC. The `define variable SYNTHESIS should be used to delete non-synthesizable code and the preprocessor will be run twice, once with `SYNTHESIS set to create code for a view named “syn” and a second time with is not set for a view named “sim”.

Simulation and Synthesis Requirements.

The `define variable SYNTHESIS should be used to delete non-synthesizable code from any of core IP modules. Never use # Delay in any core IP module. Never put a `timescale in any core IP module.

Never put a $finish; statement in any core IP code.

Clock System.

Do not mix rising edge clocked flip flops with falling edge clocked ones. Do not mix any clocked flip flops with anything other than flip flops clocked on the same edge. That means no mixing with signals connected to the D or Q ports of a flip flip or with a pad.

If you need to do any clock gating then create a module that contains only that gating logic. Document your design so that the chips DFT engineer can get in there and clean up the mess that you just made.

Reset System.

Every flip flop in every component must have both an asynchronous AND a synchronous reset on it. Do not try to combine them into a single signal. If the component has any logic that must function while the products Power_on_reset signal is active (example Clock dividers or PLL) then that logic must be designed to self start under any or all possible power up conditions and it's synchronous reset must be ported out separate from the products power on reset.

Any combinatorial loops must be broken when the products power up reset is asserted.

Clock Enable System.

Every flip flop should have a clock enable signal. This gives the chip architects more flexibility in terms of clock rates and power. The clock enable takes precedence over the synchronous reset signal.

Non-Synthesizable modules.

There have always been a number of modules that are target technology dependent and may not be synthesizable. LibreChainEDA will maintain a Common Design Environment library(CDE) with models for all of these components that must be used in all designs.

Target technologies that require special handling will create their own versions of these modules and they will be substituted for each CDE module on a per instance basis.

Pads .

Input Pad
Output Pad
Tristate Output Pad
Bi-di Pad
OpenDrain Output Pad
OpenDrain BiDi Pad

Srams.

Asynchronous Ram
Synchronous Ram
Dual Port Synchronous Ram
Dual Clock Synchronous Ram

Multipliers.

Parallel
Serial
Booth

JTAG.

Test Access Port
Wrapper Cell

Clock.

Synthesizer
Driver
Gater