SOCGEN Project

User Manual

Document: UM-100
Date: Sep 2012
Rev: 1.00.00

Legal Notices

Tools and documentation are released under the following license

The opencores.org/cde library is released under the following license.

Index

List of Figures

List of Tables

Document versions

Version Number	**Date**	**Author**	**Summary of Changes**
1.00.00	19 AUG 2012	John Eaton	First version.

Tab. 1: Document versions.

Conventions

This document uses the following conventions. An example illustrates each one**.**

Font	**Meaning or Usage**	**Example**
`Courier font`	Messages and system displays	`Building sim`
`Courier Bold`	Literal Commands that you enter	
`Courier italic`	Variables where you supply values	

Tab. 2: Conventions.

1 *Overview*

The continuing growth in the size and complexity of modern System on Chip (SOC) designs can only be sustained by using an effective Design-for-Reuse(DFR) Methodology. This consists of adherence to design standards that fully describe the deliverables passed from the Component Designers to the System Architects and Silicon Makers. Once these standards are in place and enforced then Electronic Design Automation (EDA) tools can be developed to automate the design creation and verification steps.

The SOCGEN project is an open source DFR tool set hosted on the opencores.org web site. It contains a set of design creation tools that when used with other open source or no-cost Electronic Design Automation(EDA) tools will allow a designer to create ,verify and program their SOC designs into a Field Programmable Gate Array (FPGA) or release as a Application Specific Integrated Circuit (ASIC). The project also contains several reference designs as well as Design Standards and a library of useful modules. Socgen provides tool flows for:

- Creation of an SOC RTL hierarchical design.
- Support for Fizzim Finite State Machine Tool
- Simulation of a Test Suite using Icarus Verilog
- RTL Checks (Linting) using Verilator
- Code Coverage using Covered
- Synthesys and FPGA creation using Xilinx web pack

1.1 *Contact Information*

• Maintainer	John Eaton
• Company	Ouabache Designworks
• Email	z3qmtr45@gmail.com

2 Design Environment

2.1 SOCGEN Design Environment

The SOCGEN design environment can be created in any subdirectory by creating that subdirectory and then populating it with repositories containing the needed tools and IP. The easiest way to do this is to checkout the socgen project from the opencores.org site and then delete any unneeded socgen ip and adding any of your ip repositories into the projects subdirectory. Checking out the socgen project will give you the following:

./doc	Contains all socgen documentation
./tools	Contains all socgen tools specific to this DE
./projects	Contains all the vendors component IP.
Makefile	makefile for running all design processes.
test	script to run build and test all socgen IP.

The SOCGEN project on opencores.org contains an example DE along with several example designs and a common library. The ./tools directory contains all necessary tools and may be copied into any DE as well as the common library. SOCGEN tools support the IP-Xact reuse standard so any IP-Xact compatible tool can be used for entering or modifying a design.

The DE is also used to execute the design process scripts. These processes are all command line based and run from a top level Makefile. EDA tools with a

Graphical User Interface (GYI) may be used for nonrepetitive tasks like design entry or debugging but all tool flows must be executable from the command and capable of being run unassisted on a computer farm.

2.2 Tool Flows

2.2.1 INDEX

Scans all xml files in /projects and creates a catalog with the location of all ip-xact and socgen design files. This must be run after any IP is added or removed from the ./projects directory.

Example: `$make index`

2.2.2 WORKSPACE

Builds a directory image of the top level design in a volatile work area that is symbolicily linked back to the original files in the repository. It also symbolicly links all child components used in the design in a children directory located below the parent design.

Cleaning the workspace is done by removing all files and rerunning workspace.

Example: **`$make workspace  opencores.org  or1k`**

or **`./tools/sys/workspace /projects/opencores.org/or1k /work`**

This will create a new subdirectory and symbolicly link all of the or1k files under `/work/opencores.org__or1k`.

It will then create another subdirectory called `/work/opencores.org__or1k/children` where it will link all of the subcomponents used anywhere under the top level module.

All of the files generated by any tool flow will be put in this workspace.

2.2.3 BUILD_OBJ

Runs assembler or C-compiler on all firmware in project sw directories.

Example: **`$make build_obj`**

2.2.4 BUILD_SW

Links all sw libraries and creates bit image files.

Example: **`$make build_sw`**

2.2.5 BUILD_HW

Builds all verilog code for all projects in work area.

Example: **`$make build_hw`**

This flow will traverse all libraries in the /work directory and their children and will execute a command to run all componentGenerators found in any ip-xact component file.

2.2.6 BUILD_FPGAS

Synthesizes and routes all fpgas in work area.

Example: $**make build_fpgas**

This flow will locate any chip found in any toplevel socgen design.xml file and synthesys it using Xilinx webpack.

2.2.7 RUN_SIMS

Runs linting on all testbenches and then runs all verilog simulations.

Example: $**make run_sims**

This flow will locate any sim found in any toplevel socgen design.xml file and simulate it using icarus verilog.

2.2.8 RUN_COVERAGE

Creates code coverage reports on previously run sims.

Example: $**make run_coverage**

This flow will locate any sim found in any toplevel socgen design.xml file and extract the code coverage information from its vcd file in a coverage data file. It then runs coverage to produce a code coverage report.

2.2.9 CHECK_SIMS

Checks all simulation results.

Example: `$make check_sims`

2.2.10 CHECK_FPGAS

Checks fpga compile results.

Example: `$make check_fpgas`

2.2.11 CLEAN

Cleans all old files out of work area.

Example: `$make clean`

2.3 IP Repositories

Design data is NEVER stored in the DE, it is always stored in a revision controlled system(RCS) repository and the "Golden Copy" is kept in a safe and secure location with backups. There are numerous choices for RCS tools but they all provide three basic functions:

• Data Revisioning	Stores all revisions for current and past files.
• Cloning	Provides complete copy of data from master source
• Time Machine	Restores to any time or tag point from the past

The only design data stored in any engineers local DE will be new code that has yet to be checked in and existing code that is in the process of being modified. It is a good design practice to check in this code as often as possible.

All RCS tools will provide these three basic functions but will differentiate themselves by adding many new extra features. DO NOT USE ANY OF THESE FEATURES! Your code is going to outlive whatever RCS tool du jour your team is using and someday you will have to dump out all your code and reenter it into the next latest and greatest RCS tool. If you avoid using any feature but the three basic ones then that will make moving your ip a lot easier.

There are two ways to organize your IP into RCS repositories. The first is “Store by where used” in which every chip design gets one repository where EVERYTHING used to create that design (including the DE) is stored in that repository. This method was popular in the past when designs were smaller and life was simpler and its main attraction was that if needed you could always restore the bits from the archives and you pick up the design exactly as the original designers had left it.

But then it became apparent that not only did you need the IP but you also needed the now quite obsolete and non-supported versions for the EDA tools that were used for the original design. These tools and licenses may not be available and if they are they may not run on your modern hardware.

Bit rot is real. If you want to be able to reissue your design in 10 or 15 years then you must plan on maintaining it and testing it against any and all tool upgrades.

The other big problem was that as the size of designs grew the size of design teams also grew. A chip design today can easily have over 100 engineers from around the world and several companies all working on the rtl code. IP code is valuable and some of it is licensed from companies with rather draconian remedies for what happens if the code that they deliver to you ever shows up on wikileaks.

The prudent engineering manager will limit the team members access to rtl code on a need to know basis. This means that you split the code and store it in multiple repositories and give each team member access to only the repositories needed for their job.

That is the second method and is called “Store by who created it”. Each piece of IP will have a vendor name from their Ip Xact file that determines where the IP is kept in the repositories. You will have IP from several different vendors and you can arrange it so that each vendor can access their own IP but no other vendors.

2.3.1 IP-Xact Repositories

The SOCGEN project imposes some additional requirements for storing IP-Xact enabled IP.

- Repository must contain a spirit:component file with .xml suffix.
- Repository must contain a socgen:ip file named design.xml
- Top level directory must match vlnv Vendor name
- Second level directory must match vlnv Library name
- Some lower level must match vlnv component name.
- Component named directory must contain the entire component

2.4 Projects

The projects directory is used to store all the IP from all the different vendors and make it accessible for use by any other IP component. The top level inside of the projects directory will contain a separate directory for each vendor and the directory name will match the vendor name from the ip-xact file vlnv. Directly underneath the vendor directory will be the library directory and it's name will match the library name from the vlnv.

Component directories must also match the component name from the vlnv but they may be located anywhere under the library directory. Any code that is needed by a component must either reside inside the component's directory or reside in it's own ip-xact repository.

An IP may be added into a DE by cloning it's repository into the DE projects directory. Once IP is added or created it must be added into the socgen registry by typing:

```
$make index
```

2.5 Workspace

Using a piece of IP from some outside source is like buying something from IKEA. It will arrive in a box and some assembly is required. Any IP repository will only contain the minimum files needed for the design process flows to build the finished design. Running all the design processes will grow the repository size by several orders of magnitude.

These generated files can create serious issues. Design work involves creating new files and modifying existing ones. These changes must be tested by running a complete set of design flows before any of the candidate changes are committed back into the repository.

But if you query the repository state after running the design flows then it will report the addition of thousands of new files that are not yet checked in and scattered among the entries will be the 2 or 3 files that you actually need to commit. Good luck in spotting them.

This is why all RCS software includes an IGNORE feature so that you can mask out the thousands of files and only show the ones that are important. The problem is that RCS software is written by software engineers for other software engineers. SOC design flows are more complex than your usual software build scripts. A software engineer knows that all *.obj files are generated and can ignore all of them. A SOC designer does not know if a verilog .v file was hand crafted or generated from a Register or Finite State Machine generator. Configuring the repository IGNORE masking is a huge time sink and should NEVER be used on SOC IP repositories.

The big problem has to do with the complexity that you can sometimes find in real IC designers build scripts. You will see makefiles calling perl code calling shell scripts to run EDA tools and then running SED of their outputs. Thats even before they add recursion so don't be surprised if you receive some IP with a build script that resembles Rube Goldberg on a bad hair day.

Any makefile process can change what it does based on the existence and time stamp of files that exist in the database. A RCS feature that tells you the the repository is in it's virgin state when it actually still has generated files remaining from the previous run is a really really bad idea.

Soc designers should never use the ignore feature from RCS software. Not only is it a huge time sink to maintain but you now have to also test your build scripts to prove that they work under all possible configurations left over from the last build. This number can be huge.

That is why the SOCGEN project provides a workspace for running the design flows. Workspaces are similar to what you get when you use the unix **`lndir`** command. You can check out a repository into a directory called foo and then create another directory named foo_cmp. The **`lndir`** command can then be used to create an exact copy of foo into foo_cmp. The only difference is that foo will contain the actual files while foo_cmp will have symbolic links back to the actual files.

Now you can run all of your design processes in foo_cmp and that is where all the newly created files will be generated. Checking the status of your RCS repository will only show the changes that you have made to the database.

If a design process does delete and recreate a repository file then the new file will only exist in foo_cmp while the original is unchanged in foo.

If a design process writes to an existing repository file then the write will occur in the repository and the status will report that the file has been modified. This can only happen if you check a generated file into the repository. We NEVER check generated files into a repository for just this reason.

One of the benefits of a separate workspace is that cleaning up before rerunning is now a simple **`rm -r *`** command at the top level of the workspace directory. You will no longer have to maintain the clean function in all your makefiles.

Another benefit is that the projects directory could contain a large number of IP components and running all design flows on all components could take forever. A workspace is built only for the top level designs that the designer specifies and no unneeded tools are run.

To create a workspace simply type the workspace command followed by the top level vendors library directory and the name of the workspace directory.

```
$./tools/sys/workspace projects/opencores.org/fpga_or1200/ /work
```

When finished the command will have created a single directory in /work named opencores.org__fpga_or1200 that is symbolically linked back to its source in the /projects directory.

It will also create a directory /work/opencores.org__fpga_or1200/children and that will contain all the children IP used in this design. They will also be symbolically linked back to their sources.

When a design process runs it only operates on designs in the /work directory and will store all generated files in this workspace.

3 Creating a SOCGEN component

3.1 SOCGEN IP Packaging Requirements

Each component must be fully contained within a sub-directory with a name that exactly matches the name field from the components ip-xact component file VLNV. This directory will reside underneath a library directory with a name that exactly matches the library field from the components VLNV.

The component directory will contain a socgen:ip file and a ipxact spirit:component file for each version of the component.

Socgen currently only supports verilog and each component must create a verilog library file for each of it's versions. This can be done by including a componentGenerator in the spirit:component file that runs the ./tools/verilog/gen_verilogLib script.

3.2 *SOCGEN:ip file*

Each component must have a socgen:ip file to provide setup and configuration information for simulation and synthesys design flows.

3.2.1 Xml header,copyright, name space and schema

```
<?xml version="1.0" encoding="UTF-8"?>
<!--
//                                   //
// Copyright boilerplate             //
//                                    //
-->
<socgen:ip                        xmlns:socgen="http://opencores.org"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
```

3.2.2 VLN data and path from library to sim directory

```
<socgen:vendor>opencores.org</socgen:vendor>
<socgen:library>fpgas</socgen:library>
<socgen:component>Nexys2_T6502</socgen:component>

<socgen:library_sim_path>/ip/Nexys2_T6502/sim<
/socgen:library_sim_path>
```

3.2.3 Testbenches

Each component must have at least one testbench for use in simulations and linting. The testbench it self will be a separate ip-xact component but this section identifies it as a testbench and provides setup data for code coverage. Code coverage is only preformed on the blocks identified in a cover block.

```
<socgen:testbenches>
<socgen:testbench>
<socgen:variant>Nexys2_T6502_tb</socgen:variant>
<socgen:version>tb</socgen:version>
<socgen:code_coverage>
  <socgen:cover>
  <socgen:name>T6502_cpu_def</socgen:name>
  <socgen:componentInstance>TB.test.dut.core.T6502.cpu<
  /socgen:componentInstance>
  </socgen:cover>
  <socgen:cover>
  <socgen:name>io_module_def</socgen:name>
  <socgen:componentInstance>TB.test.dut.core.T6502.io_module<
  /socgen:componentInstance>
  </socgen:cover>
</socgen:code_coverage>
</socgen:testbench>

</socgen:testbenches>
```

3.2.4 Configurations

These are all the various setup configurations that will be passed to both the simulation and synthesys flows.

```
<socgen:configurations>
<configuration>
  <socgen:name>T6502_io_irq_2</socgen:name>
   <socgen:parameters>
    <socgen:parameter>
      <socgen:name>RAM_WORDS</socgen:name>
      <socgen:value>2048</socgen:value>
    </socgen:parameter>
    <socgen:parameter>
      <socgen:name>RAM_ADD</socgen:name>
      <socgen:value>11</socgen:value>
     </socgen:parameter>
 </socgen:parameters>
</socgen:configuration>
</socgen:configurations>
```

3.2.5 Sims

This section lists each simulation along with its testbench, configuration and any additional parameters.

```
<socgen:sims>
      <socgen:sim>
        <socgen:name>io_irq_2</socgen:name>
        <socgen:configuration>T6502_io_irq_2</socgen:configuration>
        <socgen:variant>Nexys2_T6502_tb</socgen:variant>
        <socgen:parameters>
          <socgen:parameter>
              <socgen:name>TIMEOUT</socgen:name>
              <socgen:value>800000</socgen:value>
          </socgen:parameter>
        </socgen:parameters>
      </socgen:sim>
</socgen:sims>
```

3.2.6 Chips

This section lists each synthesizable design along with its target technology and configuration.

```
<socgen:chips>

  <socgen:chip>
    <socgen:name>Nexys2_T6502_io_irq_2</socgen:name>
    <socgen:target>
    <socgen:vendor>opencores.org</socgen:vendor>
    <socgen:library>Nexys2</socgen:library>
    <socgen:component>fpga</socgen:component>
    <socgen:part>xc3s1200e-fg320-5</socgen:part>
    <socgen:overlay>
      <socgen:vendor>opencores.org</socgen:vendor>
      <socgen:library>cde</socgen:library>
    </socgen:overlay>
    </socgen:target>
  </socgen:chip>

</socgen:chips>

</socgen:ip>
```

3.3 IP-Xact spirit:component file

Some of the blocks will be parametrized. This section contains a list of those parameters and default values:

3.4 IP-Xact spirit:design file

Some of the blocks will be parametrized. This section contains a list of those parameters and default values:

3.5 IP-Xact spirit:designCfg file

Some of the blocks will be parametrized. This section contains a list of those parameters and default values:

3.6 IP-Xact spirit:busDefinition file

Some of the blocks will be parametrized. This section contains a list of those parameters and default values:

3.7 IP-Xact spirit:abstractionDef file

Some of the blocks will be parametrized. This section contains a list of those parameters and default values:

4 Tools

4.1 Install

SOCGEN requires that numerous other tools are installed on your system in order to function. Most of these can be installed by going into the subdirectory that matches your operating system and typing:

```
$ make install
```

Other tools will need to be downloaded from their sources and manually installed. These include:

- openrisc tool chain or-32-elf.
- Xilinx webpack version 13.3
- Kactus2 IP-Xact editor
- Fizzim State Machine Tool

There is also a sample dot_profile file that may be used to replace your ~/.profile in order to setup your environment upon login.

4.2 YellowPages (yp)

Every ip-xact tool flow has yellowpages support. The scripts will build two registry files that contain the VLNV and file type for every ip-xact and socgen xml file as well as its filename and directory.A lib.pm provides support functions used by all of the socgen tools.

The registry must be rebuilt anytime ip is added or removed from the projects directory. This is done from the top level by typing:

```
$make index
```

4.3 Sys

The total number of signals is 30. This is lower than our first estimates when there was no keyboard controller but still keeps the chip pad limited. A

4.4 Verilog

The total number of signals is 30. This is lower than our first estimates when there was no keyboard controller but still keeps the chip pad limited. A

4.5 Simulation

The

4.5.1 Testbench Creation

4.5.2 Icarus Verilog Simulation

4.5.3 Verilator Linting

4.5.4 Coverage code coverage

4.5.5 Gtkwave waveform viewing

4.6 Synthesys

The

4.7 Regtool

The

4.8 *Fizzim*

The

5 *References*

SOCGEN project
http://opencores.org/project,socgen

IP-Xact standard
http://standards.ieee.org/getieee/1685/download/1685-2009.pdf

Accellera
http://www.accellera.org/apps/org/workgroup/ipxug/download.php/11213/Accellera_Standardized_Bus_Definition_Creation_Workflow.pdf

Accellera
http://www.accellera.org/apps/org/workgroup/ipxug/download.php/11211/Accellera_IP-XACT_Bus_Definition_Creation_Guidelines.pdf5/download/1685-2009.pdf

6 Quick Start Guide

The fastest way to use socgen is to first download the complete project from the opencores.org website. Socgen requires the addition of several opensource tools in order to operate. These can be found under the /socgen/tools/install directories. In some cases a script is provided while others will require that you install a tool from its own website.

Once everything is installed and working then change to the socgen directory and type:

```
make index
```

```
./test
```

after everything has run the results can be checked by typing:

```
make check_sims

make check_fpgas
```