# SOCGEN Design Environment

# Users Guide

SOCGEN is  a complete EDA design environment that simplifies the creation of component IP and makes it easy integrate it into a System-on-Chip (SOC). Its tool set uses IP-xact files to build and retarget a complete SOC. It is designed to create of complex designs from components obtained from a variety of different designers.  It is free ,opensourced and available from opencores.org.

## Installation

There are two ways to install and run socgen. The project on opencores also contains a variety of different example IP modules from other opencores projects. These have test suites and may be synthesized into fpgas using the provided scripts.

The second way is to use only the socgen/tools directory and add that to your own design environment to furnish the tools.

### Full SOCGEN Tool set with Example IP

Start by checking out the complete socgen project from opencores.org. You will need a login to do this.

```
  %>  svn co -user <UserName> -passwd <Password>  opencores.org/ocsvn/socgen/socgen/trunk socgen
```

There are several opensource tools needed by socgen that must be installed in your host machine. The directory socgen/tools/install contains install scripts for the various supported operating systems.Once installed then all socgen scripts are called from a Makefile at the top level.  These scripts will create a workspace area for generated files , create all of the verilog files as well as filelists and tool controls, compiles all the software embedded in the design, runs all test suites with linting and code coverage, synthesizes all fpgas and reports the results.

The commands to do all of this are:

```
%>make workspace
%>make build_hw
%>make build_sw
%>make run_sims
%>make build fpgas
%>make check_sims
%>make check_fpgas
```

## SOCGEN Tools only

Start by creating a subdirectory with the name of your design environment and then check out the socgen tools from opencores.org into this directory.

```
%>  svn co -user <UserName> -passwd <Password>  opencores.org/ocsvn/socgen/socgen/trunk/tools tools
```

Copy the Makefile from ./tools/install up to your top level. Create a new subdirectory name projects.All of your ip repositories will be checked out into this directory. This IP must be IP-Xact compliant and it's top level directory name must match the vendor's name and follow the socgen component database guidelines. All of the socgen scripts will work on this IP.

# Using IP-Xact in a modern design for reuse environment

Modern asics have grown to the point where it is no longer possible for a design team to create a chip from scratch in any reasonable amout of time. Todays designs rely on reusing major amounts of code from past designs or 3rd party IP providers. A modern asic will combine modules created by an inhouse design team with many others from outside designers to create a code base that can be synthesized and turned into a IC.

IP-Xact (IEEE-1685) plays a crucial role in this effort. IP-Xact defines sets of file that provides a "Data Sheet" for each component module. This data sheet is not meant for human consumption but rather is designed to ease the importation of that module into the asics tool flows.

There are three different jobs in designing a modern asic and designers need to understand what each one provides.

## Component Designer

A component designer creates a module that performs some usefull task. Along with documentation and IP-Xact files they will also create a test suite that verifies that the component design is correct.

## System Architect

An architect selects various modules, configures them and then interconnects them to create a hierarchial module. They also create documentation, IP-Xact files and a test suite for this module. This module can then be used by other architects until finally you have one module the represents a single IC.

## Silicon Maker

The silicon maker will take that module to synthesize what they can and instantiatates library modules for the rest. It is made testable before it is placed and routed. Timing is closed and the design is made into a silicon die.

IP-Xact is a key player in passing design configuration and other information along with the rtl code. Design for reuse is all about efficiency and that requires a robust and proven EDA tool set capable of using the IP-Xact standard.

## adHoc and bus signals

All signals in a design are either adHoc or a member of a bus. An adHoc signal is any signal connecting two or more instances where the designers are free to pick their own signal name. You can do an entire asic using nothing but adHoc signals but in a large design this leads to chaos and makes the design very hard to understand and maintain. Busses are used to bring order to this mess. You create a bus by first defining some characteristic shared by a subset of the adHoc signals and choosing a name for this group. Then all adhoc signals that belong in this group will change their signal names to match the buses signal name. Furthermore if there are any adhoc signals that are not a member of this group that happen to have signal names that match must change them to something that doesn't collide.

For example a design team will decide to create a bus for all signals that drive the clock port of a flipflop and that they will uses the name "clk". Then all signals driving clock ports must change their names to include "clk" and any non clock port signals using "clk" must find something else that doesn't collide

The IP-Xact files for a design will have different sections for adHoc and buses. This gives the tools more information about bus signals that can be used to the tools advantage.

Both adHoc and bus signals may be passed up the hierarchy but can be changed anytime along the way. For example a lcd controller module creates a 8-bit vga output as adHoc signals. Rather than port these out as adHoc it will create a busInterface for a vga bus consisting of eight RGB levels and a horizontal and vertical sync signal. This bus is then ported up the hierarchy using a hierConnection at each level until the last level before the pad ring. At that level a bus interConnection will be used to connect to the lower level while a busInterface of ten pad signals are created using the same names as the vga bus. The top level padring will use interConnections to connect all the pads to the core.

## Design Repositories

"Every piece of knowledge must have a single, unambiguous, authoritative representation within a system."
From The Pragmatic Programmer

A Revision Control System (RCS) is mandatory for all design work. This provides a location for backup data as well as design history and the ability to restore the design to any previous version. Design for reuse has had a profound change on how component IP is stored. A SOC that was a in-house design could create a single RCS repository that would store the entire design including all of it's subcomponents. When component IP is reused you will have multiple designs all needing the same components. In this case the component designer must create their own repository for each component with a test suite and make that repository available to all users.  A modern SOC is comprised of many different components and each one should maintain a single location to serve as the  source for that component. The design environment for a SOC is a very minimal database that contains only the scripts needed to locate and check out all the repositories needed to build the SOC.

When a piece of component IP is obtained from outside the organisation then then a copy of that IP should be kept in a central location for use in all SOCs that need it.  This avoids have having multiple copies of a component floating around and protects against loss if the original source disappears. It also aids in distributing updates and bug fixes.

SOC design may involve working with an outside asic vendor.  A seperate respository should be used by the vendor to add all of their libraries and components for synthesis. A seperate repository prevents the vendor from accessing source code and documentation that must be controlled.

The choice of RCS software is up to the component designers and any modern SOC will likely use a mixture of systems. When choosing a RCS system remember that your component IP code will likely outlive whatever the RCS-du-jour happens to be and that you will eventually have to port over to a new system.

A design repository for a socgen IP component must be IP-Xact compliant and follow the socgen guidelines for directory structure.

## Workspaces

The build and verification process will expand size of the design environment by several orders of magnitude. It is important that generated files are never stored in a active RCS repository. The possibility that the presence or time stamp of a defunct generated file could alter the build is to much of a risk to take. A workspace is created for the build process that creates a linked file image of all the pertinent repositories in a separate directory. Source files are all available and generated files are stored there. A clean build is guaranteed by deleting the workspace and rebuilding a new one.

It is really hard to keep track of all the new files that you have added that you need to check into the Revision Control System  if they are buried by gigabytes of generated files from the build process. The workspace uses symbolic links to create a area where generated files are kept outside the RCS repository. Never check a generated file in an RCS repository. They should only contain the minimal seed data needed to rebuild the entire design. It should never contain any files that were generated by the build process.

A similar workspace is also created for tools.

# Projects

A project is a database structure that is used to store component IP. All projects are stored under the projects subdirectory in a socgen design environment and delivering a component to another designer simply means creating a package containing the project for the top level and all sub projects that are part of the design. Once received the projects are placed in the projects directory and the component can then be added by any IP-Xact editor by calling form it by it's vlnv.

The top level directory of a project is the vendor name from the IP-Xact vlnv. Under that are as many libraries as needed with the library name used for the directory name.

Libraries are used to manage the database by grouping related components together into one library. Each library can have up to four subdirectories.

## DOC

./doc/  contains any needed documentation.

## BIN

./bin/  contains any needed scripts or tools needed to build the IP or compile the software. These tools are called by componentGenerators in the IP-Xact files

## IP

./ip/  contains all the IP components. The directory name matches the component name from the components vlnv.

## SW

./sw/  contains any sw that must be embedded inside the design.

# Components

A component is used to store the files needed to create the component rtl code. It also can contain a simulation tool flow with test benchs and the test suite that demonstrates the components functionality or a synthesys tool flow to convert the component into gates.

The example designs included in the socgen project use verilog only. Tool flows are included that use Icarus verilog for simulation, Covered for code coverage , Verilator for linting and Xilinx ISE 13.3 for synthesis.

Notice that socgen components do NOT include an extra level to store the different versions of a component. The reason for this is that versioning by its very nature will create designs where very few lines of code are actually changed. If you are changing a significant amount of code then you should not release a new version, you should release a new component.  Placing each version in it's own subdirectory will result in a significant amount of code that is duplicated between the different versions. This would break our rule about having a single place to store each piece of data.

Components can contain /doc and /bin directories are well as others

## IP-XACT

./ip-xact/  contains any needed ip-xact designConfiguration files for this component and all of its versions. These files are only needed for hierarchal designs that contain other socgen components.

It also contains a design.xml file that is a non-ip-xact file. It is used for "yellow pages" functions and to provide setup information to the tool flows. This function will take in a vlnv and return the absolute path to the file containing that ip-xact object.

The design.xml file contains.

(1) List of all designs by their component and version
(2) List of all testbenchs with design_under_test and code coverage information
(3) List of all non-default configurations with name<>value pairs
(4) List of all simulations with testbench and config
(5) List of all synthesizable designs with their component and target

## RTL

./rtl/verilog  contains all the verilog code needed by all versions of the design. These can be any of three possible types:

| | |
|---|---|
| (1) verilogSource | complete verilog module where the module name matches the filename. |
| (2) verilogInclude | consists only of  `define  statements. |
| (3) verilogFragment | is a `include "filename" inside of a verilog module. |

./rtl/xml   contains all the ip-xact components and designs for all the versions of the component. Leaf cells will only have a component file while hierarchal cells will also have a design file.

Most components will have several versions and it is important to understand how ip-xact uses versioning. A component can have different versions released over time. Six months after you release version 1.0 you fix some bugs and rerelease as verison 1.1. In this case the affected verilog files are copied to a different name , modified and the file sets in the ip-xact component files are modified to point to the new files.

You can also version a component over design space. If you have a cpu where the user can configure it with or without I-cache and D-Cache then your component can have four different versions. The best way to configure these is to use parameters. Then each instance can set it's own configuration and you only need one ip-xact component file.

But there are two situations where you cannot use parameters. These are when the parameter will change a port list or when it will change a filelist. If either of these occur then you must create a seperate ip-xact component file with it's own vlnv. This creates a problem because the vlnv has four places to store five fields. Good luck.

Some designers try to avoid this issue by creating a superset design. The design is created with all possible ports and any unneeded output ports are tied off to safe values and the file list includes files that may not be used in the design. This is a design-for-reuse disaster waiting to happen and must NEVER be done.  Have you ever spent several hours assembling something only to end up with a few parts left over? It could be that the manufacturer had several products needing simialar part kits and simply created one superset kit for all of them. Or you could have made a mistake and the thing will break when you try to use it. How much time will you spend to figure out which one is correct? This is why we do not do superset designs. If you make a mistake in your rtl and forget to instantiate a needed module then that module will show up in a simulation as a second top level module that you can track down and fix. If component designers do a superset filelist then your easiest chance of catching that error will be buried under thousands of unused files.

IP-Xact component filenames follow the form

    component_version_config.xml

where

| | |
|---|---|
| component | component name from vlnv |
| version | version name from vlnv |
| config | user assigned configuration name ( _def is for the default configuration) |

IP-Xact design files follow the form

```
    component_verison_config.design.xml
```

./rtl/fsm   A finite state machine generator is a very useful tool. You use a GUI to enter the states and transitions defining the state machine and save that datafile in the ./fsm directory. You then use a componentGenerator in the ip-xact component file to read the fsm datafile and create a new verilogSource or verilogFragment file. Socgen supports the fizzim opensourced FSM tool but any others could be easily added.

A register tool that creates register logic, documentation, #include files etc is also very useful but we do not have a place for the register tool source files. That is because ip-xact supports register descriptions and we use a componentGenerator to read the ip-xact component file and create the verilog.

./rtl/views   Views are the biggest change that most designers will have to adjust to when using ip-xact. Many designers create a single deliverable that is used by all tools. This means that any unsynthesizable code must use a pragma or a `ifdef to hide itself from synthesys tools. Ip-xact lets each tool create its own deliverable that contains exactly what it needs. Nothing more or nothing less.  Each socgen component has a view for simulation(sim) and a seperate on for synthesys(syn).

When the socgen build_hw flow has finished then the rtl/views directories will each contain a ./sim and ./syn subdirectory. Inside these will be one file for each version of the component and the filename with match the component_verison name. This file will be a verilog Library file that contains all the modules needed by that version. This is created by a socgen tool that reads every ip-xact component file to find the filesets for each view and then processes every file in the fileset through a verilog preprocessor putting all the output files into a single verilog library file.

This means that the SOC designer using a component only has to read one file to get all the code of that component. They do not have to set any search paths since these were all resolved by the preprocessor. All of the verilog ttic statements(`) have also been resolved so that there is no danger of name space collisions.

In order for this to work we need to ensure that there are no module name collisions between the different versions of a component. This is accomplished  by the preprocessor by setting

```
`define  VARIANT   component_version
```

and using that variable to set all of the module names and instantiations in the verilog code. The module name for the top most module will be

```
   module  `VARIANT
```

and an instantiation for a module named rx_fsm will be

```
   `VARIANT`RX_FSM   rx_fsm1  (
```

The build_verilogLibrary script is used to create the verilogLibrary file in the views/ directory. This script is run using a componentGenerator in each ip-xact component file that takes all the verilogInclude and verilogSource files in each views fileset and runs them through a verilog preprocessor before storing the results in the views directory.

The top level file for a component can be stored in a verilogSource file but this cannot be modified or reconfigured when the design changes. The preferred method is the run  the build_verilog script from a componentGenerator. This script will read the ip-xact component and design files and recreate the verilog module that they describe. If there is any logic that cannot be described by ip-xact then it is placed in a verilogFragment file listed in the fileset. This file is included by the build_verilog script.

Simulation only code is placed in a verilogFragment file that is listed in the sim fileset but omitted from the syn fileset. If a design uses `ifdef SYNTHESYS  then the syn fileset will have a verilogInclude file with a `define SYNTHESYS. Translate on/off pragmas are not supported.

## SIM

./sim/xml   contains all the ip-xact components and designs for  testbenchs. Each version must have at least one testbench

Testbench filenames follow the form

    component_version_type_tb.xml

where

| | |
|---|---|
| component | component name from vlnv |
| version | version name from vlnv |
| type | only needed if there is more than one testbench for a version |

Testbenches are always hierarchial designs. They will instatiate the design_under_test(DUT) along with all bus_functional_models(BFM)s and other simulation models. The models used in simulations are all independent socgen components that are stored in a different socgen project. Avoid storing simulation models in the component directory. If it is useful for you then the designer using your component may also want to reuse it for their test suite. All shared code must be placed in their own socgen projects.

./sim/verilog  contains any remaining verilog code needed by testbenches that will never be reused by other designers

./sim/icarus/      Each simulation tool has it's own  run area in the sim workspace.  Socgen currently only supports icarus verilog for simulations. Each test in the test suite has its own subdirectory  to store the test and all of it's results files. Separating each test this way makes it easier to farm out a test suite to different cpu's.

Simulation directories follow the form

component_version_config_testSequence

where

| | |
|---|---|
| component | component name from vlnv |
| version | version name from vlnv |
| config | configuration name from ./ip-xact/design.xml file |
| testSequence | user assigned name for code in test_define file |

Inside the directory is the test_define file  which contains the verilog code that directs the entire test. Also included is a vcddump control file (dump_define) to control what signals are dumped as well as a wave.sav file for use be a vcddump viewer.

./sim/lint/ All testbenchs are check for syntax and structrual error by verilator. The log files for each one are stored here.

./sim/cov/ Each testbench can define what areas should be covered by code coverage. The accumulated results for each testbench are stored here.

## SYN

./syn/ise/ Each synthesis tool has it's own work area. Socgen currently only supports Xilinx ISE webpack.  The only thing needed is a bsdl directory with the bsdl file for the design. All other information is either in the ./ip-xact/design.xml file or the component file.

# Tools

## Makefile

clean
all
workspace project=name
build_hw
build_sw
run_sims
build_fpgas
check_sims
check_fpgas

### soc_builder

Usage:  soc_builder   project_name

### soc_generate

Usage:  soc_generate    -prefix  prefix_path -project project_name -lib_comp_sep  lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name

### build_sim_filelist

Usage:  build_sim_filelist       -work_site   work_path  -vendor vendor_name  -project project_name -lib_comp_sep  lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name  root_comp_xml

### build_syn_filelist

Usage:  build_sim_filelist       -work_site   work_path  -vendor vendor_name  -project project_name -lib_comp_sep  lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name  cde_library

### build_coverage

Usage:  build_coverage      -work_site   work_path  -vendor   -project project_name -lib_comp_sep lib_comp_sep -component component_name

**build_verilog**

Usage:  build_verilog     -view  view_name  -prefix  prefix_path -project project_name -lib_com_sep lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name fragment no_port  destination_name  dest_dir

**build_verilogLibrary**

Usage:  build_verilogLibrary     -view  view_name  -prefix  prefix_path -project project_name -lib_com_sep lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name dest_dir

**build_registers**

Usage:  build_registers     -view  view_name  -prefix  prefix_path -project project_name -lib_com_sep lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name -bigendian   dest_dir

**build_fizzim**

Usage:  build_fizzim     -view  view_name  -prefix  prefix_path -project project_name -lib_com_sep lib_comp_sep -component component_name -comp_xml_sep comp_xml_sep -variant variant_name -encoding encoding   source destination

## Rules to live by

(1) plan ahead

You may start a design with the intent that it is only going to be used for one specific purpose only to find out later that other designers want to use it. Create all designs with the intent that they will be reused in ways that you haven't imagined and you won't have to scramble later.

(2) maintain the design

Releasing a chip to production is not the end of the job. You must still continue to maintain the design. You cannot archive a chip data base into offline storage and simply put it on the shelf. Do you really think that you can pull it down 20 years later and recreate the chip? Bit Rot is real. Even if you can read the bits off the magtape that you used to use then you will find that you can no longer get the same version of the tools that you used  to build the chip. The original IC process will be long gone and the current ones have added new requirements that your code doesn't meet.

When you finish a chip you archive an exact copy of all the data and freeze that forever. Your design then continues to live on. When you get a new version of a tool you rebuild and test everything and fix problems. As new processes come online you retarget the design to use them. As component ip is reved you upgrade and run the test suite.

Then when your original product is winding down and someone wants a follow up product then you have a head start.

(3) design for the lowest common denominator

Everybody loves to use some quirky little feature of the design target to squeeze a little extra performance out of the system. But if you do then you are locked into that target and cannot easily reuse the design on a different target. Why do you think they put those features in the first place? Instead you should survey the field and only use the features that all target technologies can match

(4) design in a completely generic technology

Design is a two step process. First the design is created and verified in a completely generic behavioral RTL format and then converted into the target technology. It is tempting to try to save time be designing in the target technology but this will make it harder to reuse.

(5) Automate everything

Handcrafting a design file is a time consuming and error prone operation. Tasks that are preformed on every design should be done by a tool. The designers job is to create the configuration files needed by the tools and let automation do all the work.

(6) Store files based on their source and not their use

Are you creating a chip using IP from Joe's IP Emporium? Why not create a spot inside your chip database for Joes files? Because that is not planning ahead. Later if your lab starts another chip that also uses Joes IP then they will also need access to those files. Create a spot for files where everybody can simply access them by linking the desired files into there database.

(7) Do no mix unlike objects in the same file

"Unlike" is a deliberately nebulous term. It can mean anything and everything. If you have a instance of a hard macro that is unsynthesizable then do not put it in a file along with synthesisable rtl code. If you have code belonging to one designer then do not mix it with code belonging to another. If you do then you have to worry about file locking. Fragment the design so that each object is in it's own file and then use a tool to put them back together.

(8) Layer the design

A full design will consist of several different databases that are layered. Upper ones may override any content from a lower layer.  Requirements created by the Component Designers are only minimums, The Architects and Si-Makers are free to override and tighten any requirement from any lower level.  Parameters should be used to give the downstream  designers the ability to tune the design for the target process.

(9) Reuse other components

The best way to create a reusable component is to build it using other reusable components whenever possible. If something is useful for you then it is likely that others could also need it.  Look through any available libraries before creating a new function and if you have to create one then make it available to others.