AMD

StimGen User Guide

Peter Jakobsen

Release 58

This document is a work in progress

Table of Contents

Illustration Index

Index of Tables

1 Definitions

Term	Definition
CDS	Component Design System
DUT	Device Under Test
SEQ	Sequences
SG	StimGen
SG4	StimGen
VBCR	Virtual BroadCast Register
VER	Virtual Enable Register
VIR	Virtual Index Register
VMCR	Virtual MultiCast Register
VR	Virtual Register
VSR	Virtual Select Register
VXCR	Virtual Xcast register (VBCR and VMCR)

2 References

Google Test: http://code.google.com/p/googletest/

Visitor Pattern: http://en.wikipedia.org/wiki/Visitor_pattern

3 Introduction

StimGen is a tool that enables transfer of design information and reuse of design verification patterns into other platforms such as manufacturing ATE (Automated Test Equipment), SLT (system level test platforms) or validation platforms.

StimGen draws information from the design header files, from tagged information in SGAPI compliant documentation, from spreadsheets with pin information, and from the test sequences coded into StimGen by verification. These tests are then modified and reformatted to be used by other users in different environments.

Part of the power of StimGen is that it can automatically calculate the sequence of commands required to access a register. That is, the database contains the access mechanisms required to reach a particular target register. The user can request that a particular register have a field (symbolically specified) set to a new value. Depending on the environment, the software may be able to directly access the register (in verification), or may issue multiple JTAG commands to configure the scan access before issuing another JTAG command to set the register bits.

StimGen4 is the 4th generation of the code and is written in C++. The code may be compiled into both a program and a library. The library may be loaded by other programs so that the knowledge of registers, fields within registers and how to access them is made available to other users such as verification or validation.

SG4 has the ability to output the requested operations in multiple formats. Currently there are several interfaces into the verification environment, STIL files for ATE, SVF files for our customers and more being added regularly.

StimGen4 also uses other Open Source Software (Google Test) to help setup a test environment that allows the end users to specify generation of a subset of the available tests. This improves run time and allows the user to modify a single test without regenerating everything else.

NOTE:

This document is intended to provide information on what StimGen is and what it's capabilities are. Due the fact that it is integrated with many different environments, on multiple platforms, with different operating systems and command shells, we cannot provide explicit examples or exact syntax for all possible situations. We believe and expect that most users will be familiar enough with different toolsets that they will be able to adapt the examples to their environment.

3.1 StimGen4 Version

In March of 2015, we moved from a single number version to a three number version. Hence you will see something like version 55.1.2. The reason was to convey significance of the changes to the program. If the first (major) number stays the same, the program is guaranteed to be backwards compatible. If it increments, then you must recompile all code and may need to edit some routines to address differences in the API. If the second number (minor) changes, then new features have been added but all interfaces are backwards compatible. If the third number (patch) changes, then bug fixes

occurred in the program. Hopefully this will allow users to better evaluate the risks of moving to a new version.

4 Quick Start

StimGen has purposely been designed to operate stand-alone as well as in other environments. This section of the document, the Quick Start guide, will not discuss integration in detail but only the process of building the model. <mark>The first thing to do is build a StimGen model stand-alone and validate it.</mark> The model build is a compilation of data from many individuals and teams so before we start working on anything new, we should validate the program is working as is. This will help to avoid spending time debugging problems that are not ours.

For purposes of this quick start we will use the Carrizo project as the example data. The SIGAPI toolset uses a common format for specifying a device as an argument to the different programs. It essentially mimics the directory structure used by the Perforce (revision control software) repositories, with the directory delimiters replaced by underscores. The syntax is often *<family>_<project>_<revision>* although a minor revision may be appended onto the string. So the Carrizo project is specified for an SIGAPI project with '-proj 15h_CZ_A0'

4.1 Step 1: Setting Up Your Environment

> First thing is to get StimGen4, then check out a project.

It is outside the scope of this document to provide a instructions in the use of other commercial tools. We also do not provide instructions on setting environmental variables across all operating systems and command shells. However below is a checklist of operations that need to be performed prior to beginning, your syntax may vary.

- Identify the version of StimGen4 you are going to use. There are three options
    1. Point to a pre-installed release on design linux machines

        /proj/verif_release_ro/StimGen/<RELEASE>

    2. Point to a pre-installed release on peo linux machines

        /groups/sigdata/StimGen/<RELEASE>

    3. Create a perforce client and checkout a local version of StimGen4

        P4PORT=atlvp4p01:1671

        View:

          //depot/stimgen/main/... //<CLIENT>/...

          //depot/sigtools/main/... //<CLIENT/sigtools/...

- Compile the program with the *make* command. It should not have any errors.

- Set environmental variable SG4COREBASE to the appropriate StimGen4 code base

      setenv SG4COREBASE <PATH_TO_STIMGEN>/<RELEASE>/StimGen4

- Create a perforce client and checkout a version of the project that you want to operate on

  - Example: p4 -patlvp4s04:1671 client -d *some_name*

- We recommend creating a file called P4CONFIG with the following parameters in the directory that you checked out the design files.

    - P4PORT=atlvp4s04:1671

    - P4CLIENT=*some_name*

- Set environmental variable P4CONFIG to P4CONFIG.

- p4 client -t 15h_CZ_A0   *-t causes a description of what will be checked out*

- p4 sync   *this will cause all files to be checked out*

4.2 Step 2: SGXML Generation and Validation

This first task is to generate the SGXML (or find an existing copy). In general, checking out the latest documentation and generating the SGXML is preferable. SGXML is the abbreviation for StimGen Xtended Markup Language. XML is an industry standard and we leverage open source software to read it. The SG signifies it uses a StimGen schema to for definitions of components in the XML.

We will also need to locate custom modules definitions and sequences. Source information is any valid SIGAPI compliant documentation be it DFT ODT specs, SGXML, etc. To generate the sgxml file necessary for SG4, run the following command:

```
$SG4COREBASE/bin/genSG4.pl -proj 15h_CZ_A0 [ -regaccess ] [ -link ]
```

The -regaccess flag is optional. If included it will enable register access tests as part of the regression tests, otherwise they are disabled by default.

The -link flag is optional. By default you will get a copy of any custom IP/SOC .h/.cpp files for the SG4MODEL. With -link specified these files will be symbolically linked on linux and hard linked on Windows to the source.

Next we will review the results of the genSG4 command. If there are problems, we will need to get them corrected by notifying the appropriate document owners and/or file UBTS (Universal Bug Tracking System) or JIRA ticket as required.

The SGXML now needs to be validated to ensure that it is complete and complies to the schema ensuring that tools can parse it. Run the following command   validate SGXML file

```
/tool/pandora64/.package/libxml2-2.6.31/bin/xmllint \
        --noout --xinclude –-schema $SG4COREBASE/../sigtools/schema/SGXML.xsd \
        sgxml/carrizo_A0.xml
```

You should see the following if there are no problems.

```
        sgxml/carrizo_A0.xml validates
```

If there are problems correct them or contact SIGAPI or StimGen developers for assistance.

The SGXML can be viewed in a web browser allowing you to view the data model, instance, register, bitfield data to see exactly what the valid names are that should be used for StimGen patterns.

4.2.1 Generating SGXML for Functional Registers

By default, SG4 will only generate SGXML for DFT and fuse register definitions. For many users these are the only registers required. To access functional registers SGXML must be generated and customer register access sequences loaded into the model. To generate SGXML for the functional registers add to the genSG4.pl command line the appropriate arguments to load the functional registers from the desired source: using following options to generate functional registers SGXML

- -bkdg Load registers defined in FrameMaker BKDG source

- -scs   Load SCS registers defined in Excel spreadsheet

- -svd   Load 'System Visible Registers' from SVD XML source

- -arm   Load ARM registers originating from ARM XML register definitions

==Any combination of these functional register definitions can be loaded together.== Be sure that the source data exists. GenSG4 will terminate if it is unable to locate the source data in the SIGDATA repository.

When functional registers are loaded they can be accessed in the same way as DFT registers assuming that register access methods are in place. These methods are beyond the scope of the quick start and will be discussed later.

4.3 Step 3: Generate S4MODEL

This step is optional as an independent step. Step 4 will build this automatically.

StimGen loads design information including hierarchy and register definitions from SGXML at runtime. When there are custom IP or SOC modules and or sequences the SGXML will be accompanied with SG4/include and SG4/src and optional SG4/tests. In order to access this additional functionality these source files must be compiled in to a dynamic library that is loaded at runtime. StimGen can operate on only the SGXML but users will not have the benefit of previous work.

When genSG4.pl is run it will copy or link these custom files into the new include, src and test directories if they aren't already there. In addition a standard test pattern will be created to test the SGXML import and model destruction and ensure that all DFT registers are accessible.

If there are no problems build the StimGen shared library

```
    make  SG4MODEL
```

4.4 Step 4: Run Regression Tests

By default, genSG4.pl generates a standalone regression test and will also copy any existing IP and SOC regression tests from the sigdata repository into the local build environment. While not necessary it is strongly advised that these be run to ensure there are no known problems with the data model and the accompanying custom sequences. To run the regression tests simply run command

```
    make test
```

All regression tests are based on Google's gtest (see references). For Carrizo you should see output like this and a PASS at the very end

```
    carrizo_A0_test
    Running main() from gtest_main.cc
    [==========] Running 2 tests from 2 test cases.
    [----------] Global test environment set-up.
    [----------] 1 test from DSM_B0_Test
    [ RUN      ] DSM_B0_Test.CallingAllFunctions

    << STDOUT REMOVED >>
```

```
[       OK ] DSM_B0_Test.CallingAllFunctions (7 ms)
[----------] 1 test from DSM_B0_Test (7 ms total)

[----------] 1 test from carrizoA0Test
[ RUN      ] carrizoA0Test.ModelBuild
Total FCH IR Operations:    0
Total FCH IR Shift Cycles:  0
Total FCH DR Operations:    0
Total FCH DR Shift Cycles:  0
Total tdr IR Operations:    0
Total tdr IR Shift Cycles:  0
Total tdr DR Operations:    0
Total tdr DR Shift Cycles:  0
[       OK ] carrizoA0Test.ModelBuild (260 ms)
[----------] 1 test from carrizoA0Test (282 ms total)

[----------] Global test environment tear-down
[==========] 2 tests from 2 test cases ran. (289 ms total)
[  PASSED  ] 2 tests.
```

If these tests all pass, you are ready to integrate StimGen into your environment.

4.5 Step 5: Integrating StimGen into your Environment

To utilize StimGen we must construct a model of the device under test or dut. StimGen includes a factory class to build the model given a path to the SGXML and the presence of libSG4MODEL.so in the run directory.

```
        #include "StimGen.h"
        void main ( char** argv, int argc ) {
          StimGen::Module dut;
          try {
            StimGen::ModuleFactory factory( "sgxml" );
            factory.import( "carrizo.xml", "", &dut );
            StimGen::Event_Visitor_Debug debug_visitor;
            dut.add_event_visitor ( debug_visitor );
            dut["ATECONFIG"]["SOC_APU_FCH_OVERRIDE"](1)
                              ["SOC_PADRCVEN"](1).write().apply();
1.          } catch ( exception &x ) {
2.            std::cout << "Exception: " << x.what() << std::endl;
3.          } catch ( const char* const &x ) {
4.            std::cout << "Exception: " << x << std::endl;
5.          } catch ( const std::string &x ) {
6.            std::cout << "Exception: " << x << std::endl;
7.          } catch ( ... ) {
8.            std::cout << "default exception caught" << std::endl;
9.          }
        }
```

Line 1: Include the StimGen header files

Lines 5-6: Load the sgxml and any custom libraries and return the top level dut

Lines 7-8: Add a generic event visitor pattern to echo all events created. You will provide a custom visitor pattern specific to your environment.

Line 8: Write to the ATECONFIG register and instruct stimGen to generate and process the events via

the apply command

Lines 4,9-18: Catch any exceptions that may be thrown during execution which stimGen may be providing more detail about

4.6 Sample Flow Charts:
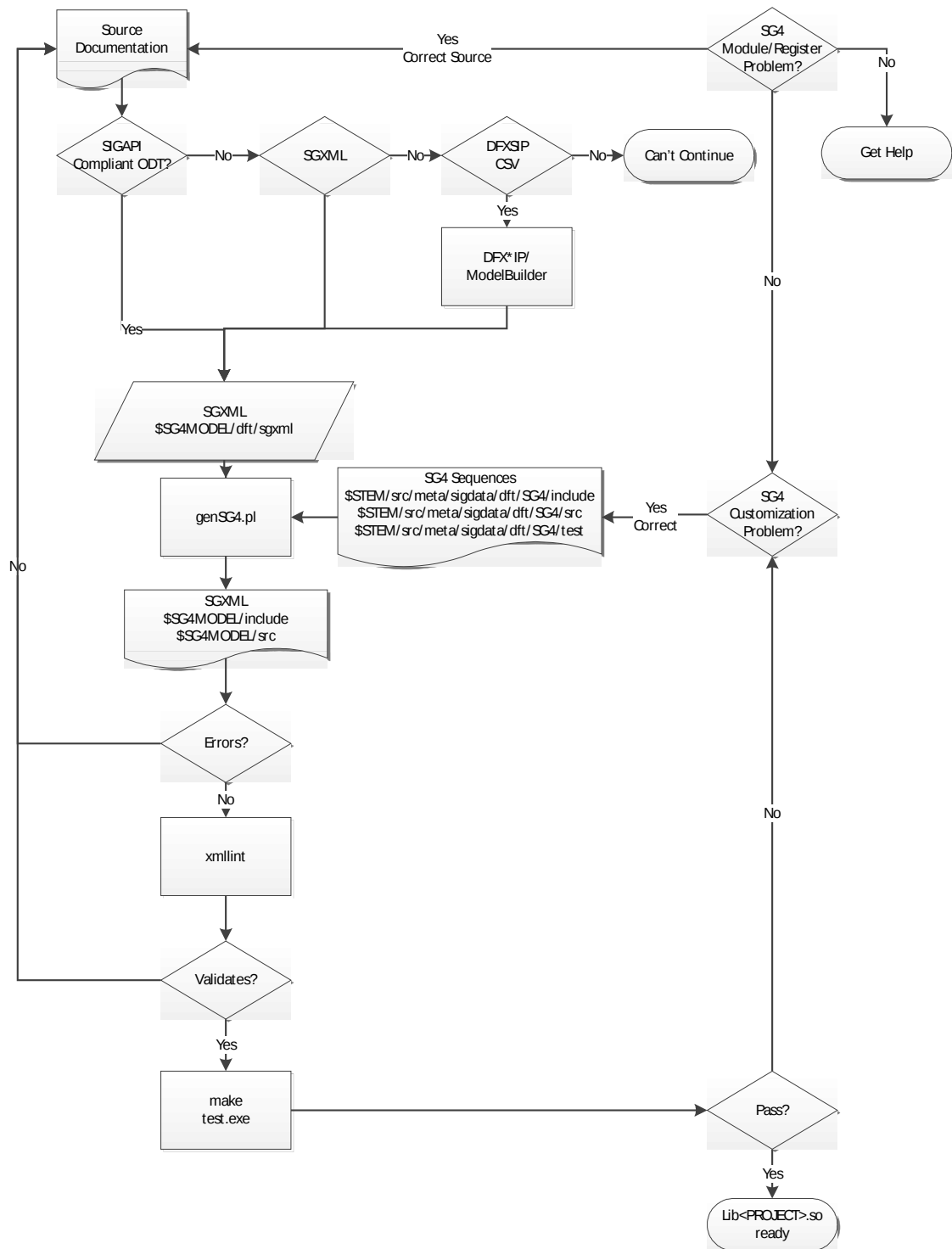
Illustration 1: Flow chart showing generic flow for producing SGXML from documentation and compiling to produce shared library
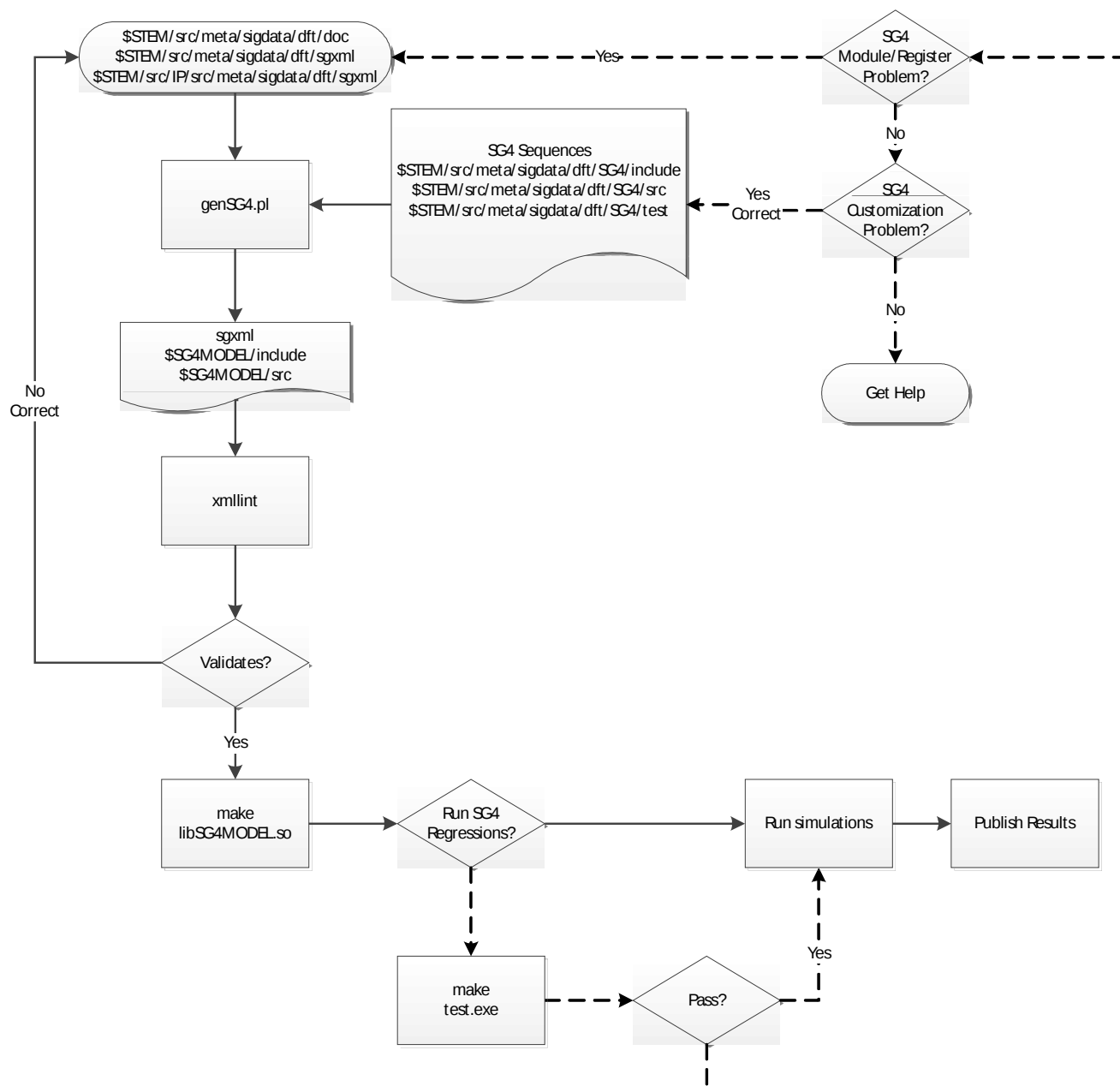
Illustration 2: Flow chart for taking SGXML to Simulation

5   Pattern Writing

One of StimGen's primary benefits is to abstract away the details about register access and allow you, the test pattern writer, to focus on what the test needs to do. Most of the operations will involve reading and writing registers that are available somewhere in the design behind some sort of access mechanism such as JTAG, SRBM, PCI operations etc. StimGen isolates you from the access mechanism and will generate the lower level sequences it takes to read/write data from/to the target register. If you need to know how the data gets to and from the target register, then you are an advanced user and more information will come later. If you don't care how the data gets to or from the register to the interface, then this chapter is for you.

StimGen builds a model that is composed of modules, pins, registers and bitfields. Modules contain pins, registers and other modules. Registers contain one or more bitfields which are used to access single or multiple bits by name.   All of these building blocks have names associated with them and are closely modeled after verilog conventions. Note that these names may but most likely will not align with design RTL implementation. StimGen is built off of documentation which abstracts away much of the design hierarchy and discusses or describes it in more human friendly generalities. To view these names load the design SGXML in your favorite web browser to see what module, register, pin and bitfield names are available in the design.

Everything in StimGen is an object which you call methods against or assign values. The key to using StimGen is knowing how to locate the object of interest. Also note that as with any library, there are probably multiple methods for achieving your desired purpose. For example, in the next section we show how to set a single pin. On a device with 500 pins, the user could repeat that method 500 times specifically naming each pin. However, we also show three lines of code that sets default values on the majority of pins, with specific pins called out for special treatment. Knowing your library can save you a lot of time.

5.1 Understanding the Queue

To write a test pattern, we want to execute a sequence of events. These events may be setting up pins or requesting a set of serialized accesses of a set of registers via JTAG. We also want to be able to accumulate events and have some control over which vector events go into. The way that StimGen implements this is through a queue. All of your operations go into a queue with a timestamp that is automatically appended. Eventually, when everything is written in the queue, we issue an apply(). This can cause multiple output handlers to visit the queue and generate their output.

For example, let's assume we wanted to generate test vectors in two different formats for two different test platforms. We start by specifying an initial set of pins values. When all the initialize values are specified, we can advance the timer. Now pin values are accumulated in the queue for the next test vector until we advance the timer again. We repeat this sequence until we have vectors that reset the part and assert PWROK. Next we put a register access command into the queue. This is going to automatically expand into one or more JTAG events that will could possibly generates 10s or 100s of cycles. Now we tell StimGen to process the queue with the apply() method.

The apply() method will call the two different Event_Visitor handlers (a C++ term) to inspect the queue

and generate their output. The first test platform is a tester and outputs vectors using 01LH to represent drive0, drive1, measure0 and measure1 respectively. The second test platform uses SVF files and will use LHDU for the same values respectively. Hence we get two files in two very different formats from a common source.

5.2 apply()

As described above, the apply() statement initiates pattern generation for all of the events stored in the queue. Below in 'Writing a Register' we describe how StimGen takes care of configuring the network access mechanisms to required to write (or read) the requested data. So during the apply() a single event may be expanded into multiple events required to setup the network. By default, StimGen will also reset (or teardown) the network at the end of the reply so that we are in a known good state.

Therefore, optimally, a user would put all events that are relevant to the network setup within a block of code that is operated on by a single apply(). Perhaps an example makes that clearer. If there are 10 registers to configure to run a memory BIST test, then it is best to have all of those in the event queue and one apply() statement. Once StimGen sets up the network for the 1st register write, it will know and use that setup for the other register writes and we generate a minimum number of cycles. After the apply() is done the network is shutdown.

There are some instances where we want an apply() (perhaps in a subroutine that initializes the part) and we don't want to tear down the network. In that case we use an optional argument to the apply() method of type EventApplyOptions. The command would look like the following:

```
// Mark any register required to access the target for reset when done
// but don't reset network access registers until next apply()
dut.apply( EventApplyOptions::delay_auto_reset() );
```

The statement above tells the apply() not to reset/tear down the network at the end. The network will be reset at end of the next apply() unless that one also wants to delay. There is another option that tells StimGen that it should not even mark the registers for reset. That is, the user prefers to own the responsibility of managing the networks. In

```
// don't mark any network access registers for reset
dut.apply( EventApplyOptions::ignore_auto_reset() );
```

Note that these options are only relevant for the one apply() where they are specified. There are persistent controls that can be set on a per part basis through a configuration file. (See Chapter 20 – ConfigManager). There are also more options that will be described in later sections of this document where they are relevant.

5.3 Set a Pin

To access a pin, we need to know the module name. For the following example, we have a pointer named dut that points to a module. We use it to call a method to directly write a pin to an enumerated pin value.

```
dut.write_pin("BP_PWROK", PV_0);
```

Enumerated pin values basically is a subset of defines that we use to standardize on meanings. These

defines were picked from the most common usage symblols. { PV_0 = drive 0; PV_1 = drive 1; PV_H = expect 1; PV_L = expect 0; PV_X = don't care; PV_Z = high impedance; PV_p = pulse pin high, PV_P = pulse pin low } Remember our earlier discussion of the queue, where H = expect 1 in STIL format, whereas H = drive 1 in SVF format ? PV_H makes it clear we mean we are expecting a 1.

There are several methods for iterating through the entire list of pins and setting values. There are also some methods that do that for us such as the following. In the example below, we setup a few pins for special values, then pass that list into the initialize_all_pins() method along with values we want to default to all the other pins. The 2nd to 4th arguments are values for inputs, outputs and bidirectional pins.

```
pin_overrides("BP_POWER_OK", PV_1)("BP_TDI", PV_0)("BP_TDO", PV_Z);
dut.initialize_all_pins(pin_overrides, PV_0, PV_X, PV_Z );
dut.apply();
```

The method apply() is called when we want to write the accumulated set of pin or register writes in the queue to the one or more of the SG4 formats.

5.4 Measure a Pin

5.5 Writing a Register

What is really clever here is that the user doesn't need to know whether a register is directly accessed through JTAG, is a functional register that is set through the HDT interface or requires multiple operations to configure IEEE1500 or STAC architectures to setup access to it. You merely specify what value you want and StimGen will figure out how to get it there. Further serial protocols (JTAG) will be expanded into multiple cycles generating just the right amount of SHIFT-DR states to read the selected register.

In the following code, we create a register pointer that is initialized by looking up a register by a unique name. Then we set the bit fields in the register by their symbolic names and finally write it into the queue.

```
Register & bistconfig_load(*get_register("BISTCONFIG_LOAD"));
bistconfig_load["REQUEST"](1)
              ["READONLY"](0)
              ["ADDR"](start_addr + offset)
              ["DATA"](v)
              ["VALID"](0)
        ;
bistconfig_load.write();
```

5.6 Read a Register

The following code is very similar to writing data. We get a pointer to register, then we define what values we expect for different bit fields and then issue the read into the queue.

```
Register &bistcfg = *get_register("BISTCFG");
bistcfg["START_FINISHED"].set_measure_value("1'b1");
bistcfg["RUNNING"].set_measure_value("1'b0");
bistcfg.read();
```

5.7 Write and Read Register

Often registers are accessed via JTAG (or another serial interface) and therefore operations are inherently write and read. We can specify this operation with the following compacted method. You will often see this in the automatically generated tests that make sure we can access all of the registers specified in the sgxml file. The test will verify that all specified registers also had their access mechanism specified.

Here we are specifying the entire register hierarchy relative to the top module dut.

```
dut["Top.L3DATA.L3A0TAG0_FATALERR"]("11'h0", "11'h0").write();
dut["IDCODE"](0x1234567,0x1234567).read();
```

5.8 Poll a Register

To poll a register is to read a register repeatedly until a desired state is measured or some number of attempts have been made. Implementation of polling is different from one environment to the next. In the case of STIL generation for PEO there is no concept of polling; instead the pattern should wait some fixed amount of time before issuing the first read. In environments such as verification or when communicating with a device, we can repeatedly read a register we can issue reads until the desired state is reached. Its up to each visitor pattern to determine how to handle based on the environment.

To initiate a poll operation in stimGen, set the desired register or bitfield measure value and then call the poll method with number of reference clock cycles to wait and the number of retries to allow.

```
dut["CPU0.L2.BISTCONFIG"]["DONE"].set_measure_value(1);
// Wait 1 million cycles before issuing the first of 5 possible  reads
dut["CPU0.L2.BISTCONFIG"].poll(1000000,5);
```

This pattern will wait one million cycles and then read the CPU0.L2.BISTCONFIG["DONE"] bit at most five times until the bit has been asserted. What this generates in each visitor pattern is up to each visitor pattern. Be sure to review the output and ensure its doing what you are expecting and that the stimulus is correct for the target environment.

5.9 RegisterAccessOptions

The previous paragraphs dealt with a standard read or write. There are times when we want to do a little something extra.

5.9.1 Chain Check

The purpose of the **chain check** option is to shift a pattern through a chain and verify that the documented length of the chain is the actual length. This is done by passing a *RegisterAccessOption* variable to the read or write method for the register. Example code is shown below.

```
RegisterAccessOptions ccopts = RegisterAccessOptions::chain_check();
dut["IR"](0xFF, 0x01).write( ccopts );
dut["REG08"]("0xA5", "0xE7" ).write( ccopts );
dut["REG01"]("0x1", "0x0" ).write( ccopts );
dut.apply();
```

In line 1, we create an option variable that we will pass to the write methods below. There are multiple ways to get and set a *RegisterAccessOptions* but chain check is used so much that it has it's own constructor and initializer. This will generate codes that pushes a unique header 0xDEAD through the ring. If you see this signature come through at the expected time your chain is the expected length.

5.9.2 Chain Check with Pause

If we are using the IEEE1149.1 (JTAG) protocol to access the registers, then our design must comprehend that at sometimes it is in the PAUSE-IR or PAUSE-DR mode and should hold the bit data in it's current position until we resume shifting. SG4 has a special algorithm for addressing this kind of test. A calculation is performed to generate a *signature* and *header* bit pattern and specify an interval to execute PAUSE states. For clarity, we will use a non-optimized example. Assume we have a 64 bit chain and select a 16 bit *header* (0x555) and 8 bit signature. We shift in the 16 bit *header* and enter PAUSE, which tests that bits [63:48] are holding their data, then we shift forward another 16 bits (also pushing the *signature* into the chain) and PAUSE to test that bits [47:32] are holding data, then we repeat to test [31:16] and finally [15:0], then finally shift our 0x5555 header out and compare to see if it is as expected. If any register continued to clock, the *header* would be corrupted. Another 8 clocks and we read our *signature*. Our total clock count was the chain length + header length + signature length + 4 PAUSE states (3 clocks each) for a total of 64 + 16 + 8 + 12 or 96 clocks.

```
RegisterAccessOptions ccopts = RegisterAccessOptions::chain_check();
ccopts.set_option( RegisterAccessOptions::ENABLE_PAUSES );
dut["IR"](0xFF, 0x01).write( ccopts );
dut["REG08"]("0xA5", "0xE7" ).write( ccopts );
dut["REG01"]("0x1", "0x0" ).write( ccopts );
dut.apply();
```

5.9.3 Chain Check with Pause Debug

Our previous test uses few clocks but only tells us whether we passed or failed, which makes it great as a production GO / NO GO test. But if we primarily care about validating our design and want to find out which bit failed, there is another option we can added in, **DEBUG_PAUSES**. This causes a different algorithm to execute. In this case, to test our 64 bit chain we will shift in 64 bits of 0x555... data, enter the PAUSE-DR state and then shift our pattern out. Now if any bit position is non-static during PAUSE-DR it will be corrupted. Our clock count is calculated with the same equation, which yields 64 + 64 + 8 + 3 or 139 clocks (~50% larger). The overhead for this type of pattern goes up substantially as the chain length increases. A 10K ring can be tested in 10,350 clocks (optimized) or 20,000 clocks (to get debug capability).

```
RegisterAccessOptions ccopts = RegisterAccessOptions::chain_check();
```

```
ccopts.set_option( RegisterAccessOptions::ENABLE_PAUSES );
ccopts.set_option( RegisterAccessOptions::DEBUG_PAUSES );
dut["IR"](0xFF, 0x01).write( ccopts );
dut["REG08"]("0xA5", "0xE7" ).write( ccopts );
dut["REG01"]("0x1", "0x0" ).write( ccopts );
dut.apply();
```

5.9.4 Read-Modify Write

Another *RegisterAccessOption* is for read-modify-write operations. This options is primarily valuable to a system level validation but works like this. Assume a chain with a known length of 1000 bits and we would like to change bit 400. We want to flag our Visitor pattern to shift and recirculate bits up to the target (400th) bit, give us time to modify it and insert it back into the chain and then shift the remaining number of clocks so that all bits return to their original position. Essentially, we dump the ring but remain in PAUSE-DR then reload the ring with their original content so that our UPDATE-DR state will only change the bits that we have set.

NOTE: This implies that our design comprehends read-write modify. Supposed we have a single bit that kicks of a long self-test operation when the bit is set and the UPDATE-DR state executes. Then reloading the ring will start that self test again. A proper design must automatically reset that bit when a CAPTURE-DR state (for that chain) is executed. Otherwise, it becomes the responsibility of those writing patterns to read-modify-write those bits as well.

5.10   Clocks

We can define any pin to be a free running clock. First we should recognize that we have an internal clock cycle that SG4 runs on which corresponds in the ATE world to one vector. A free running clock might be defined merely by giving it one of the enumerated pin values of PV_p or PV_P. This would make the pin pulse within the 'vector'.\

However, we might wish to create a clock that has an integral multiple of 'vectors' as it's cycle time. We can create such a clock with a command that is similar to the following:

```
Clock *fclk = dut.create_clock("CLK_PIN", 4, 0, 2 );
```

The parameters to the create_clock method are the pin name, number of internal cycles this clock spans, the cycle to rise on and the cycle to fall on. Since the rise time precedes the fall, we will also make the assertion that the clocks quiescent state is low. This command causes a free running clock to be over-layed on top of the test sequence. It will span four cycles, rising on the 0th cycle and falling on the 2nd cycle. The clock is enabled by default.

Sometimes the user may not have access to the original pointer that was returned by create_clock(). It is also possible to look-up a clock by it's name or by a known pin pointer.

```
Clock *fclk = dut.get_clock( "CLK_PIN" );
Clock *fclk = dut.get_clock( Pin * pptr );
```

There are two commands that the user has to control the clock.

```
fclk->enable( true | false );
fclk->shutdown();
```

The first and foremost thing to realize about these commands is that they are not part of the

vector_queue that we have talked about before. These commands are processed immediately, whereas all other commands are placed in a vector_queue that is only processed at apply(). Therefore, they can be thought of as controlling the state of the clock from the point of the last apply() forward. For coding clarity, it is strongly recommended that they always appear immediately **AFTER** an apply() so that it is apparent that they are applied to the clock during the next section of code.

The enable method can be used to 'freeze' the clock in it's current state. The clock resumes running when enable is set true again. So in our example, if we froze the clock high in cycle 0, and it was disabled, it would remain in the high state until we re-enabled it. At that time, it would start counting internal cycles again, so it would remain high for one more cycle (it's cycle 1) before going low (it's cycle 2).

The shutdown method has the effect of disabling the clock after it completes a full cycle. In this case, if we shutdown() in cycle 0, it would continue to run until it completed (cycle 3) and then would not run again unless we issued an enable(true) command.

There is yet another way to influence the clock although the user will generally want to avoid this. If the user *wanted* to glitch the clock, the user can use write_pin() to directly set the current state of the clock. That state will then persist until the next clock rising or clock falling cycle. NOTE: The clock state is first updated by write_pin() statement, then by the clock control parameters. Therefore if a write_pin("Clk", PV_0); occurs in the same cycle as a clock's rising edge, the rising edge will win. In that case, the user would have needed to issue a disable to avoid the rising edge.**Clock Alignment**

There are times where the user will want to align multiple clocks. A primary example of this might be when production would like to be able to divide patterns on boundaries that enable reassembling them at run time but keeping clocks in a continuously operating state. This can be achieved via an option to the apply command.

```
dut.apply( EventApplyOptions::align_clocks( 2, "ClkA ClkB ClkC", 200 ));
```

The first argument of the align_clocks option constrains the final vector count to be a multiple of this number. This is useful for ATE that requires the vectors to be a multiple of 2, 4 or 8 (sometimes referred to as 'xmode') for better compression or to enable higher frequencies. If there is no 'xmode' then the user should specify '1'.

The second argument is a string that contains the names of clocks that the user wants aligned.

The optional third argument can constrain the number of vectors to try before giving up on alignment. The software will calculate it's own limit if this is not specified, although this number may be pessimistic.

Example 1: if xmode is 2, ClkA period is 7, ClkB is 13 ...
     The method correctly calculates limit of 2 * 7 * 13 = 112 vectors.

Example 2: if xmode is 8, ClkA is 8, ClkB is 16 ...
     The method would calculate a overly pessimistic limit of 8 * 8 * 16 = 1024
      but the user can specify a more correct limit of 16.

If a clock is specified but not currently enabled, a warning is generated and alignment proceeds without that clock. If alignment fails, an exception is thrown. Since it may not be immediately obvious how clock alignment could fail, we provide the following example. If 2 clocks with periods of 2

vectors/cycle were started on adjacent cycles, they will always be out of phase and can never align.

Note that phase is defined as the 0th cycle of the clock (not it's logic state).

Also note that we don't actually want the clock to output the 0th state. That is, we would like all clocks to end on their last state so that the *next* vector is the 0th state. For clarity, assume we have a clock that is a logical high for 4 vectors then logically low for 4 vectors. When we align, we want to have output the 4th of the 4 low states and returned. We are now ready to split the pattern. The next sequence of vectors will start at the 0th state with the 1st logic high state of the clock.

5.11 Multiple Time Domains

We have mentioned the concept of a time stamp in the queue discussion above. We will now expand on the capabilities of controlling the times and aligning multiple events. There are multiple methods shown below for doing some of these operations, the following are recommended as the best practice for the majority of users.

StimGen has the ability to control the timing of multiple clock domains. We will define the term 'clock_tick' to mean the next integral unit of time for events to be executed or scheduled. A particular interface may require multiple clock_ticks for it's protocol clocks.

NOTE: The following assumes a minimum of StimGen version 48. Earlier versions will likely execute sequentially instead of allowing parallel operation to be specified.

We start by looking at simple pin sequence. At any time we may add changes to pin state. Once we have defined all of the changes the following commands can be used to move to the next clock_tick. All of the following will happen in the same time domain.

```
dut.write_pin("A", PV_1); // set a pin at the current clock_tick [0]
dut.write_pin("B", PV_1); // set another pin at the same clock_tick
dut.next_cycle( );        // move to the next clock_tick [1]
dut.write_pin("B", PV_0); // change this pin
dut.increment_time( 20 ); // move 20 clock_ticks forward [end time is 21]
dut.loop( 20 );           // an alternative to the last command
```

Now lets assume that we would like to issue a command to some protocol (such as a JTAG interface). This is usually done through a command to write a register (as shown in previous sections). Assume the following is appended to the sequence we used above

```
dut["OPTIONS"](0xa1a1a1a1, 0x0000 ).write(); // this starts at time [21]
```

At this point, we know we started some JTAG operation that starts at time [21]. We don't know if we needed to send the IR command, or were already setup to write to the OPTIONS register. And, depending on the clock setup we've done, this could take 16, 32, or 64 cycles. So StimGen also keeps a counter for this JTAG protocol/interface. If we add the following JTAG command,

```
dut.loop( 12, "jtag1" ); // jtag1 is our 1st jtag interface
```

it will start at the later of time [22] OR when the previous JTAG command completes. So if we issue multiple JTAG commands in a row, they will each execute as soon as possible. Meanwhile, the current clock_tick for the pins is still sitting at [21]. So if were to add a second JTAG command that goes through a second interface,

```
dut["BIST"](0x9350000000001, 0x0000 ).write(); // this starts at time [21]
dut.loop( 1024, "jtag2" ); // jtag2 is our 2nd jtag interface
```

these commands would operate in parallel to the first jtag interface. Again, the loop statement would execute as soon as the BIST register was written. Meanwhile, the current clock_tick for the pins is still sitting at [21]. We could access a third interface, or continue with other pin operations. Assume we did another clock tick operation.

```
dut.loop(14978); // gets us to [15000]
```

Any new JTAG operations would start at the later of their current protocol time, or the new pin time of [15000]. So if our loop was defined with the correct time to wait, then everyone would be resynchronized to start at [15000].

If we didn't want the pattern to wait another ~15000 clocks, we could simply have terminated with an apply() who will wait for all protocol/interfaces to finish with their queue events before closing that sequence.

We also might want to wait for all operations to be complete (synchronize) before continuing, and want to resume issuing new commands as soon as possible, but not want to reset our current access states that were setup by the previous operations. In that case, it's good to know that the apply() command also accepts EventApplyOptions as an argument.

```
apply( EventApplyOptions::delay_auto_reset()); // synchronize clock domains
```

The net result of the previous command is that all currently specified operations are completed and all time domains will update to the current clock_tick, but no dut state is changed as a result of the apply().

5.12   Special Queue Directives

```
   A comment will be output by every event handler and comments are always converted to the
correct syntax for their output.
```

```
dut.write_comment("Adding a description of something important here");
```

An environmental statement would only be used by the event handlers that it is directed towards. In the example below this would be STIL handler. Most often, the accompanying string is passed to output as shown. This allows the user to insert additional commands or data into their output file. There are also environmental commands that change the behavior of the program for the specified output format.

```
dut.write_env( "STIL", "timestamp 123;" );
```

The current set of extra environmental commands is shown below.

The Synopsys ATPG visitor there are two environmental commands that changes whether or not vectors that don't have any changes are printed.

```
dut.write_env( "ATPG_Synopsys", "SUPPRESS_EMPTY_VECTORS" );
dut.write_env( "ATPG_Synopsys", "WRITE_EMPTY_VECTORS" );
```

The Debug visitor has environmental commands to specific verbosity

```
dut.write_env( "DEBUG", "REPORT_FIELDS" );
dut.write_env( "DEBUG", "REPORT_NO_FIELDS" );
```

The Qinit visitor also has environmental commands to specify reporting granularity:

```
dut.write_env( "Qinit", "report_protocol" );
dut.write_env( "Qinit", "report_register" );
dut.write_env( "Qinit", "report_bitfield" );
```

The STIL visitor has the following environmental commands:

```
dut.write_env( "STIL", "<filename>" );
dut.write_env( "STIL", "COMPRESS_VECTORS" ); // create loops where possible
dut.write_env( "STIL", "EXPAND_VECTORS" );   // don't create loops
dut.write_env( "STIL", "EXPAND_LOOPS" );     // expand explicit loops
dut.write_env( "STIL", "CREDENCE_PAD_VECTORS" ); // add vectors at end
dut.write_env( "STIL", "NO_PAD_VECTORS" );
dut.write_env( "STIL", "93K_XMODE ##" ); // where ## = { 2,4,...,32}
```

There is yet another special command for looking at the event queue that can be helpful for debug. In the StimGen Config Manager there is an option bit for getting additional comments to the screen when expanding sequences. The following code enables those comments.

```
SGCM.report_queue_events = true;
```

6 Writing Portable Patterns

One of the primary goals of StimGen is to enable the re-use of single source test patterns in a variety of environments. Even if you don't foresee running your pattern in other environments, you should still create a portable pattern, simply to run it in the StimGen stand-alone environment to verify that there are no problems with the pattern.

> Running ten minutes or ten days of simulation only to have an exception thrown because of an incorrect register or bitfield name is expensive. In the StimGen stand-alone environment, your pattern will go through a quick verification phase.

The advantage of re-using test patterns is clear: it saves you and your colleagues the effort of re-writing, re-verifying, and re-debugging patterns that have already been completed. This ecosystem is enabled by the pattern creator following simple rules for portability and consistency to ensure compatibility between environments.

- Build an identical model for vector generation
    - Define the top level variable name as "dut" to access the model
- Model and state initialization
    - Initial register state
    - Initial pin state
- Test pattern entry
- Test pattern exit
- Environment capabilities and or limitations

The StimGen model will be assumed to be equivalent at this time. By that, we mean that we assume no changes in the model hierarchy or modification of register names. Users should either generate the SG4MODEL from the same sigdata perforce changelist or pass along their SG4MODEL generated by genSG4. See Quick Start Guide for instructions on building the SG4MODEL. (In time this will be replaced by the SG4DB which will be fully self-contained. As of 2014-11-17 this is still in development.)

Initial state is the hardest item to synchronize on. By default, StimGen will put all registers into their documented reset state. The patterns will then need to utilize StimGen method calls to change the values if needed. This default state should be sufficient most, if not all, of the time. The real challenge is in standardizing on a common way to initialize pins to the correct state. The Runtime Configuration Manager will be used to establish initial pin state definitions and ensure consistency across environments. The project's dft/SG4/<PROJECT>_rtcfg.sgxml file will be used for this.

Test pattern entry and exit are two items that are defined below and must simply be adhered to. C pre-processor (CPP) macros will be used and must be defined appropriately for each environment. These macros will be responsible for test pattern entry and exit and building and initializing the model to establish correct start points.

Every environment has its own set of unique capabilities and or limitations. Test patterns must be agnostic of these. We can pass information to the environment but should not query the environment to setup or run the dut. StimGen provides a generic write_env method whose purpose is to enable access to environment features when run in a supporting environment and ignored otherwise. The write_env method depends entirely on the callback installed into the visitor pattern in use to implement functionality.

6.1 Portable Pattern Rules

To write a portable pattern, the rules listed below must be followed. These will be discussed in more detail later. The rules are formatted as 'Rule SG4_PP##' for SG4 Portable Pattern ## (some 2 digit decimal value)

SG4_PP01:   Patterns must use only standard C++ and methods from the StimGen library.
SG4_PP02:   Environment specific directives must be supplied through the write_env method.
SG4_PP03:   Patterns must use standard header <PROJECT>_fixture.h to implement the test entry, begin and end macros, load and initialize the model.
SG4_PP04:   Patterns must use the standard entry macro SG4_TEST_F.
SG4_PP05:   Patterns must start with SG4_BEGIN to initialize the dut (registers and pins). It may also load the SG4 Model if the entry macro does not do so.
SG4_PP06:   The first argument of the SG4_BEGIN macro must be the name of the initial pin state to use when assigning initial pin values.
SG4_PP07:   Patterns must end with macro SG4_END to cleanup after the SG4_BEGIN macro.
SG4_PP08:   Patterns must use the Runtime Configuration Manager to initialize pins. The <PROJECT>_rtcfg.sgxml must define initial pin states.
SG4_PP09:   Patterns must use Module::write_comment or SG_LOG for reporting messages. Patterns may not use std::cout or std::cerr or an environment messaging system.

The following sections in this document will explain those requirements and how to meet them.

6.2 Pattern Format

The most challenging part of reusing a test pattern is not the content of the pattern but standardization of the entry and exit points, file header, and definition of the initial state. This will be defined below and must always be strictly adhered to. The pattern format definition will be based on the StimGen stand-alone environment and utilize SIGAPI/SIGDATA naming conventions. This will enable compliance checking for portability by using the StimGen stand-alone environment.

6.2.1 Pattern Skeleton

All projects must use a common pattern skeleton. Portable Pattern Skeleton below provides a simple pattern skeleton.

```
#include "<PROJECT>_fixture.h"

SG4_TESTCASE_BEGIN( TESTNAME )

SG4_TEST_F ( <PROJECT>DUTTESTF, TESTNAME ) {
  // Build the model. Set registers to documented reset state
  // Set the initial state of the dut
  SG4_BEGIN ( /* Required initial pin state name */,
              /* optional jtag timing name, default to "default" */
              /* optional args needed for project, needs to be kept in sync */
            );

//*************************************************************************
// insert test content here
//*************************************************************************

  SG4_END();
}

SG4_TESTCASE_END( TESTNAME )
```

Text 1: Portable Pattern Skeleton

All environments must provide a single #include file called

        \<PROJECT>_fixture.h

    where PROJECT is of the form used by SIGAPI:

        \<NAME>_\<REVISION>

    This will match the top level SGXML file name for most projects. Here are a few examples:

2. amur_A0_fixture.h
3. nolan_A0_fixture.h
4. zen_core_A0_fixture.h

Additional include files may be specified, but they must be standard C++ or StimGen header files. Any

other files included may (and typically will) render the pattern not usable on other environments. Note that common methods or sequences should be added to the SG4MODEL so that they can be reused. Such methods should not be located in local header or implementation files.

There are five required macros and one optional macro that will/may be defined via the <PROJECT>_fixture.h file. The macros provide the entry point into the test pattern and perform any necessary setup and cleanup required. These macros are defined in the table below and expanded in the sections that follow.

Macro	Description
SG4_TESTCASE_BEGIN(...)	Required. Insert any headers, defines, macros, other before test method entry. Normally empty.
SG4_TEST_F(<PROJECT>TESTF, TESTNAME) -or- SG4_TEST_F(<PROJECT>DUTTESTF, TESTNAME)	Required. Main test entry. In the stand-alone environment google test provides enhanced functionality via test fixtures. In other environments this maps to the required test entry method. If you are using 'dut' as a pointer use <PROJECT>TESTF and if you are using 'dut' as a reference use <PROJECT>DUTTESTF.
SG4_BEGIN(PinStateName, MTapDefinitionName, ...)	Required. Do any required initialization and set the initial pin states and optionally specify clock and protocol definitions
SG4_END(...)	Required. Free up any resources allocated by SG4_BEGIN(). If the SG4_TEST_F mapping requires are return value it should be encoded into SG4_END arguments
SG4_TESTCASE_END(...)	Required. Do any clean up of SG4_TESTCASE_BEGIN()
SG4_VERIF	Optional. Is #ifdef or #ifndef to limit operations applied in verification where long delay loops can be abreviated.

6.2.2 SG4_TESTCASE_BEGIN(…)

This macro takes any number of arguments and inserts any methods, variables, defines, or enumerations before and outside the SG4_TEST_F test method entry point.
SG4_TESTCASE_END(...) should be used to terminate the SG4_TESTCASE_BEGIN(...) in the case that '{}'s are used.

6.2.3 SG4_TEST_F(FIXTURE, TESTNAME)

The macro SG4_TEST_F(FIXTURE,TESTNAME) will be the standard entry point. When run in stand-alone mode the default FIXTURE <PROJECT>TESTF or <PROJECT>DUTTESTF provided by genSG4 will be used and SG4_TEST_F will be converted to TEST_F. The type of fixture selected will provide either a top level pointer or a reference for the device under test. In other environments SG4_TEST_F will be mapped to the entry point required by that environment. The macro arguments may or may not be used in these environments. See examples below.

6.2.4 SG4_BEGIN(InitialialPinStateName, MTAPJtagTimingName, args…)

The SG4_BEGIN macro includes any setup required by the environment and must set initial pin state, and optionally clock and protocol timing. The SG4_END macro must be used to free any resources allocated by SG4_BEGIN operations. If for example, SG4_BEGIN allocates memory, SG4_END must delete them.

In the stand-alone environment, the default google test fixture (TEST_F) takes care of dut creation. In other environments SG4_BEGIN must define the 'dut' variable and load the SG4MODEL and add any visitor patterns required. See the example project fixtures that follow for examples and further explanation.

6.2.4.1 Top Level Device Under Test (dut)

There are two options for the top level model which are controlled by choice of the fixture. The dut may be a pointer or a reference. Historically only a pointer was supported, but going forward the dut reference should be used. Use testfixture <PREFIX>TESTF for a pointer and <PREFIX>DUTTESTF for a reference.

6.2.4.1.1 Top Level Pointer: dut

When the top-level SG4MODEL is loaded and stored in a pointer called 'dut' use testfixture <PREFIX>TESTF. All patterns will use the variable 'dut' to access the top-level SG4 model. The variable must be of a type derived from StimGen::Module. If a custom top-level module exists for the dut then it must be of that type. That type could be any of the following forms depending on the project:

```
StimGen::Module *dut;               // No custom top level (typical for IPs)
StimGen::<NAME> *dut;               // Example StimGen::amur, where a generic
                                    //    module covers all revisions
StimGen::<NAME>_<REVISION> *dut;    // Example StimGen::DSM_B0 where there is
                                    //    a custom revision
```

The dut must be set up by either the SG4_TEST_F or the SG4_BEGIN macro. The setup may operate slightly differently depending on the environment. The following snippet of code should be used to create the dut, where <<MODULE_TYPE>> is one of the types described above.

```
<<MODULE_TYPE>> *dut = new <<MODULE_TYPE>>( /* MODULE_CONSTRUCTOR_ARGS */ );
```

```
StimGen::ModuleFactory factory ( PATH_TO_SGXML, "." );
factory.import( "<<PROJECT>>.sgxml", "", dut);
```

The MODULE_CONSTRUCTOR_ARGS and PATH_TO_SGXML must be fill in correctly as required by the specific project.

When a dut pointer is in use case must be taken to ensure that it is destroyed and all resources are freed up. A singleton or shared pointer should be used to control and manage the dut lifetime.

6.2.4.1.2 Toplevel Reference: dut

The top-level SG4MODEL must always be loaded and stored in a pointer called 'dut' (for "device under test"). All patterns will use the variable 'dut' to access the top-level SG4 model. The variable must be of a type derived from StimGen::Module. If a custom top-level module exists for the dut then it must be of that type. That type could be any of the following forms depending on the project:

```
StimGen::Module &dut;                 // No custom top level (typical for IPs)
StimGen::<NAME> &dut;                 // Example StimGen::amur, where a generic
                                         module covers all revisions
StimGen::<NAME>_<REVISION> &dut;   // Example StimGen::DSM_B0 where there is
                                         a custom revision
```

The dut must be set up by either the SG4_TEST_F or the SG4_BEGIN macro. The setup may operate slightly differently depending on the environment. The following snippet of code should be used to create the dut, where <<MODULE_TYPE>> is one of the types described above.

```
<<MODULE_TYPE>> dut( /* MODULE_CONSTRUCTOR_ARGS */ );
StimGen::ModuleFactory factory ( PATH_TO_SGXML, "." );
factory.import( "<<PROJECT>>.sgxml", "", &dut);
```

The MODULE_CONSTRUCTOR_ARGS and PATH_TO_SGXML must be fill in correctly as required by the specific project.

When a dut reference is in use care must be taken to construct it correctly. It is recommended that a singleton be used.

6.2.4.2 SG4 Model Initialization

The SG4_BEGIN macro must setup the initial state for the SG4MODEL. This includes:

- Initializing register state.

- Initializing pin state.

- Setting clocks.

- Initializing protocols and setting up their timing requirements.

When the SGXML is imported, SG4 will automatically set the register state to the documented reset state. To enable portable test patterns, StimGen must define a set of common pin state names. Each project will need to define the initial pin states appropriately. The SG4Model must be initialized by passing one of these pin state names to SG4_BEGIN as the first argument:

- "cold_reset"
- "warm_reset"
- "running"

Other initial pin state names can be defined but those listed above must always be present. To load the initial pin state the following method must be used

```
SGCM.load_pin_initialization( dut, initial_pin_state_name );
```

SGCM is a macro to retrieve the instance of the StimGen::ConfigManager singleton.

6.2.5 SG4_END(...)

The SG4_END macro must do any cleanup that is required when the test pattern completes, to undo what SG4_TEST_F or SG4_BEGIN has done. If SG4_BEGIN is loading the SG4MODEL into dut, SG4_END must delete the variable to free up all of the memory it has allocated. See the example project fixtures that follow for examples and further explanation.

In some environments the entry method must return some value. In these cases, SG4_END should setup to provide the return value.

6.2.6 SG4_TESTCASE_END(…)

This macro takes any number of arguments used to insert any methods, variables, defines, or enumeration after and outside the SG4_TEST_F test method. SG4_TESTCASE_END should be used to balance SG4_TESTCASE_BEGIN in the case that '{}'s are used.

6.2.7 SG4_VERIF

The SG4_VERIF macro is optional and should only be used to abbreviate looping in the verification environment where modeling permits. The default behavior must always be inclusion of full loop time for PEO, ATPG and other environments. Use of this macro is discouraged but necessary at times. Here is a sample usage where silicon requires a ~1ms delay but simulation does not need to wait this amount of time.

```
#ifndef SG4_VERIF
        dut.loop(1000000); // Wait ~1ms
#endif
```

6.3 Environment-Specific Directives

Using code or methods outside of C++ and StimGen methods introduces dependencies and therefore assumes capabilities or features that may or may not be available in other environments. Use of library calls/methods outside of those provide by C++ or the StimGen library are therefore prohibited. When communication with a specific environment is required patterns must use the StimGen method

```
StimGen::Module::write_env( const std::string &target,
                            const std::string &directive
                          )
```

The target environment's visitor pattern must be set up to respond to this event by installing appropriate

callback handlers. All other visitor patterns will ignore write_env targets that they do not recognize. This is key. Here are two sample write_env method calls. One will be processed by a visitor pattern that supports a target "TWI" and the second that targets "STIL".

```
//dut.write_env( target, directive             );
dut  .write_env( "TWI",  "jtagRegMgr->reset()" );
dut  .write_env( "STIL", "set vddio=3.5V;"     );
```

Here is an example visitor pattern hook to process directives that targets the TWI visitor pattern. Note that the first write_env call above will be processed and the second will be ignored.

```
    class STIMGEN_API TWI_Environment_Callback
        : public StimGen::Event_Visitor_Environment_Callback {
     public:
      TWI_Environment_Callback()
             : StimGen::Event_Visitor_Environment_Callback( "TWI" ) {}

      Environment_Callback_Status invoke( const std::string &directive,
                                                std::string &/*result*/
        ) {
        std::size_t i;
  if ( directive == "jtagRegMgr->reset()" ) {
    cJtagReqMgr* jtagReqMgr =
      dynamic_cast<cJtagReqMgr*>(
        cTestControl::GetComponentPointer("JTAG_REQ_MGR")
      );

    if(jtagReqMgr == NULL) {
      FATAL_ERROR("Unable to cast JTAG_REQ_MGR to desired type");
    }

    jtagReqMgr->reset();
    return MatchedAndStop;

  } else if ( directive.find("debug_lib.nPRESETDBF_deasserted")
          != std::string::npos
  ) {
    cDebugTestLib debug_lib;
    std::string tmp;

    eat_white_space( directive );
    getline( directive, tmp, '(' );

    eat_white_space( directive );
    getline( directive, tmp, ')' );

    uint32_t cycles = atoi( tmp.c_str() );
    debug_lib.nPRESETDBG_deasserted(cycles);
    return MatchedAndStop;

  } else if ( directive == "ThreadWaitOnPwrOkAsserted" ) {
    ThreadWaitOnPwrOkAsserted (MAX_WAIT_ON_PWROK_ASSERTED);
    return MatchedAndStop;

  } else if ( directive == "ThreadWaitOnColdResetDone" ) {
    ThreadWaitOnColdResetDone (MAX_WAIT_ON_COLD_RESET_DONE);
    return MatchedAndStop;

  }
  // No handler found here allow callback processing to continue
  return MatchedAndContinue;
  }
  };

  // Elsewhere in code or test pattern add callback to TWI visitor pattern
  StimGen::shared_ptr<TWI_Environment_Callback>
```

```
                cb(new TWI_Environment_Callback());
        twi_visitor.add_environment_callback( cb );
```

Line 1:   Define the TWI environment callback class derived from
                     StimGen::Event_Visitor_Environment_Callback
Lines 3-4:   Implement the constructor and set the target type supported to 'TWI'
Lines 6-31:   Override the invoke method and process the incoming directives as needed for the
                     given environment. It is up to the user to define and make the appropriate
                     connections to the environment.
Lines 21, 37, 41, 45: After processing the given target don't allow any other callback handlers to
                     process this directive.
Line 49:   If no handler for the directive is found allow any other installed callbacks to process
                     the directive.

When a new environment specific directive is needed the environment or the local test pattern can add
the needed functionality. If it is a common directive then it should go into an environment/global
location so that others can utilize the functionality as needed.

6.4 Project Fixture (<PROJECT>_fixture.h)

```
#ifndef <PROJECT>_FIXTURE_H
#define <PROJECT>_FIXTURE_H


#include "<PROJECT>.h"     // Top level SG4MODEL header which includes StimGen.h
// Insert additional includes here


// Fill in the appropriate information for these MACROS if needed
// If not needed they must be defined as is to remove them from compilation
#define SG4_TESTCASE_BEGIN(...)           // Insert anything needed before and
                                          // outside of method entry
#define SG4_TEST_F(FIXTURENAME,TESTNAME) \
          TEST_F(FIXTURENAME,TESTNAME) // Test pattern entry point
                                          // Call google test TEST_F
#define SG4_BEGIN(PinStateName,args...) \// Test pattern setup
          initialize(PinStateName)     // Pass along only the initial pin
                                          // state name.  Ignore all other
                                          // arguments
#define SG4_END(...)                      // Test pattern cleanup - do nothing
#define SG4_TESTCASE_END(...)             // Insert anything needed after and
                                          // outside the test method


class <PROJECT>BaseTESTF: public testing::Test {
  public:
    static StimGen::<ModuleType> *device_under_test;
    StimGen::Event_Visitor_Debug debug_visitor;
    StimGen::Event_Visitor_Checker checker;
#ifdef SG4_DV_EVENT
    StimGen::Event_Visitor_SVSequencer_Event verif_visitor;
#elif SG4_DV_PIN
    StimGen::Event_Visitor_SVSequencer_Pin   verif_visitor;
#endif
    <PROJECT>BaseTESTF ();
    static void SetUpTestCase();
    static void TearDownTestCase();
    void initialize( const std::string& initial_pin_state_name );
    void SetUp ();
    void TearDown ();

};

class <PROJECT>TESTF: public <PROJECT>BaseTESTF {
  public:
    StimGen::<ModuleType> *dut;
    <PROJECT>TESTF ();
};

class <PROJECT>DUTTESTF: public <PROJECT>BaseTESTF {
  public:
    StimGen::<ModuleType> &dut;
    <PROJECT>DUTTESTF ();
};
#endif <PROJECT>_FIXTURE_H
```

Text 2: Skeleton <PROJECT>_fixture.h for a google test framework

It is the requirement of the project environment fixture #include file to build the model, pull in all of the necessary includes for any visitor patterns required, and define the macros that will be used in the test patterns. The example in Text 2 provides the default skeleton that utilizes google test framework.The default <PROJECT>_fixture.h utilizes google test and will only map SG4_TEST_F to google test's TEST_F macro. The SG4_TESTCASE_BEGIN, SG4_END and SG4_TESTCASE_END macros are cleared out.

The default SG4_BEGIN method will take the initial pin state name and pass it along to its initialize method. All other arguments are ignored. For testing basic pattern portability this is all that is needed. To fully enable pattern reuse the various environments need to sync up all arguments and ensure that they are consistent in meaning.

6.4.1 Sample Project Fixtures

6.4.1.1 Stand-alone Project Fixture

```cpp
#ifndef STYX_B0_FIXTURE_H
#define STYX_B0_FIXTURE_H

#include "styx.h"
#include "gtest/gtest.h"

#define SG4_TESTCASE_BEGIN(...)
#define SG4_TEST_F(FIXTURE, TESTNAME) TEST_F(FIXTURE,TESTNAME)
#define SG4_BEGIN(PINSTATE,args...) initialize(PINSTATE)
#define SG4_END(...)
#define SG4_TESTCASE_END(...)

using namespace StimGen;
class STYXB0TESTF: public testing::Test {
  public:
    static styx *dut;

    Event_Visitor_Debug debug_visitor;
    Event_Visitor_Checker checker;

    void initialize ( const std::string& initial_pin_state ) {}

    STYXB0TESTF ();
    virtual ~STYXB0TESTF ();

    static void SetUpTestCase();
    static void TearDownTestCase();

    virtual void SetUp ();
    virtual void TearDown ();

};

#endif // STYX_B0_FIXTURE_H
```

Text 3: Stand-alone styx_B0_fixture.h

The default fixture builds on google test and relies on much of its functionality. The SG4_TEST_F macro is converted to the google test TEST_F macro. The SG4_BEGIN method picks off the initial pin state argument and passes it along to the google test fixture's initialize method which will setup the SG4MODEL for processing. The SG4_END macro is eliminated from compilation.

6.4.1.2 PEO STIL Project Fixture

```cpp
#ifndef STYX_B0_FIXTURE_H
#define STYX_B0_FIXTURE_H

#include "PeoGTFixture.h"

using namespace StimGen;

#define SG4_TESTCASE_BEGIN(...)
// Map FIXTURE to PeoGTFixture which takes care of business
#define SG4_TEST_F(FIXTURE,TESTNAME) TEST_F(PeoGtFixture,TESTNAME)

#define SG4_BEGIN(args...) initialize(args)   \
  dut.write_env("STIL", "W Clkin_100MHz;");

#define SG4_END(...)
#define SG4_TESTCASE_END(...)

#endif // STYX_B0_FIXTURE_H
```

Text 4: PEO Styx Fixture

PEO has developed a significant amount of infrastructure which utilizes their own google test fixture. The PEO <PROJECT>_fixture.h is simply going to connect the two together. The SG4_TEST_F is mapped to the google test TEST_F macro. All SG4_BEGIN initializes the pin state and inserts a STIL write env directive to insert the required wave form table information into the first line of the body of the generated STIL. The SG4_TESTCASE_BEGIN, SG4_END and SG4_TESTCASE_END macros are removed from compilation.

Care must be take in coordinating arguments for the SG4_BEGIN method. These are taken one for one and each arguments meaning must be aligned. (NOTE: This will take some work and we may need to revisit and attempt to formalize.)

Here is the base PEO test fixture defined in PeoGTFixture.h

```cpp
#ifndef PEOGTFIXTURE_H_
#define PEOGTFIXTURE_H_

#include "StimGen.h"
#include "gtest/gtest.h"
#include "PeoDutWrapper.h"

using namespace StimGen;

class PeoGTFixture : public testing::Test {
 public:
  static PeoDutWrapper *dut;

  PeoGTFixture();
  virtual ~PeoGTFixture();

  void initialize(const std::string& pin_configuration_name,
                  const std::string& jtag_configuration_name   = "default",
                  const std::string& mdio_a_configuration_name = "default",
                  const std::string& mdio_b_configuration_name = "default",
                  const std::string& clock_configuration_name  = "",
                  const std::string& stil_configuration_name   = "default",
                  const std::string& stil_file_name_override   = ""
                 );

  std::string get_current_test_name();
  static void SetUpTestCase();
  static void TearDownTestCase();

 protected:
  virtual void SetUp();
  virtual void TearDown();
  Event_Visitor_Debug debug_visitor;
  Event_Visitor_STIL_Credence *stil;
```

```cpp
#ifndef STYX_B0_FIXTURE_H_
#define STYX_B0_FIXTURE_H_

#include "StimGen.h"
#include "StyxSg4Wrapper.h"

#define SG4_VERIF
#define SG4_TESTCASE_BEGIN(...)
#define SG4_TEST_F( FIXTURE, TESTNAME ) void TESTNAME ()
#define SG4_BEGIN(args...) \
  SGCM.enable_reset_network_control_on_apply=true; \
  SGCM.use_verilog_four_state_compare = false;     \
  StimGen::cStyxSg4Wrapper* dut = StimGen::cStyxSg4Wrapper::Instance();  \
  dut.initPwr11();

#define SG4_END(...) \
  delete dut;      \
  dut = NULL;

#define SG4_TESTCASE_END(...)

#endif // STYX_B0_FIXTURE_H_
```

Text 6: Styx Verification Fixture

6.4.1.3 Styx Verification Project Fixture

The Styx verification environment is unique in that it creates methods to call in the verification environment through SystemVerilog DPI calls. You'll notice in the example above that the SG4_TEST_F TESTNAME argument is converted to the method name which SystemVerilog will call. An opportunity which presents itself for this type of structure, is a simple script to extract the SG4_TEST_F TESTNAME and automatically generate the appropriate DPI import method definitions for inclusion during VCS compilation. This will only work when the method takes no arguments.

The SG4_BEGIN macro in this example is setting up StimGen ConfigManager settings and is then setting up the dut variable. The method StimGen::cStyxSg4Wrapper::Instance() is returning the result of loading the SG4MODEL and SGXML. In this case the test pattern is given ownership of dut and is responsible for its destruction. The SG4_END macro which will be called at the end of the test pattern will delete the dut, releasing all memory associated with it. It is the responsibility of the fixture owner to understand ownership of the dut and handle it appropriately.

The SG4_BEGIN method here purposely has a problem; It does not accept any arguments and always sets the initial pin states to those defined in the initPwr11() method. Any other initial pin state is not supported by this fixture.

6.4.1.4 Amur Verification Project Fixture

```cpp
#ifndef AMUR_A0_FIXTURE_H_
#define AMUR_A0_FIXTURE_H_
#include "amur.h"
#include "sg_twi_visitor.h"
#include "dfx_testcase.h"
#include "svdpi.h"

class SG4_Wrapper {
 public:
  StimGen::amur dut;
  SG4_Wrapper () : dut("A0") {
    char *repo_path = getenv("REPO_PATH");
    if ( repo_path == NULL ) {
      throw StimGen::StimGenGenericException() << "Environment variable
        'REPO_PATH' is not defined.";
    }
    std::string sg4model_path( repo_path );
    sg4model_path += "/soc/fullchip/src/chip/verif/dfx/stimgen/sg4model";

    StimGen::ModuleFactory factory( sg4model_path + "/sgxml", sg4model_path );
    factory.import("amur_A0.sgxml", sg4model_path + "/libSG4MODEL.so", &dut ) ;
    dut.add_event_visitor( twi_visitor);
  }
  ~SG4_Wrapper() { Env->Finish(); }
 private:
  StimGen::TWI_Visitor twi_visitor;
  SG4_Wrapper( const SG4_Wrapper& );
  SG4_Wrapper& operator=( const SG4_Wrapper& );
};
// Provide overrides to enable running pattern in standalone mode
#define SG4_TESTCASE_BEGIN(TESTNAME,args...) TESTCASE_BEGIN(TESTNAME)
#define SG4_TEST_F(FIXTURE,TESTNAME)  int ThreadMethod(void)
#define SG4_BEGIN(...)                                          \
    TWI_BEGIN();                                                \
    SG4_Wrapper sg4_wrapper;                                    \
    StimGen::amur *dut = &sg4_wrapper.dut;                      \
    dut.write_comment("TWI TEST:: STARTING TWI TEST");  \
    dut.apply();                                               \
    dut.write_env( "TWI", "ThreadWaitOnPwrOkAsserted"); \
    dut.write_comment("TWI TEST: PWROK Asserted");      \
    dut.loop(1000);                                            \
    dut.write_env( "TWI", "ThreadWaitOnColdResetDone"); \
    dut.write_comment("TWI TEST: Cold Reset Done");      \
    dut.loop(500);
#define SG4_END(...) TWI_FINISH()
#define SG4_TESTCASE_END(TESTNAME,args...) TESTCASE_END ( TESTNAME )
```

Text 7: Amur Verification Fixture

The Amur verification fixture is similar to the Styx one but includes the wrapper class in the same fixture. This allows us to see how the SG4MODEL/StimGen::amur dut is loaded and how visitor

patterns are added in.

Amur verification utilizes the TWI agent in its verification environment. The test pattern entry point is different from all others we have seen. This SG4_TEST_F example

```
        SG4_TEST_F ( AMURA0TESTF, PLLTESTDEBUG ) {
```
Will be converted to
```
        int ThreadMethod(void) {
```

which is the standard entry point for TWI based test patterns. It also utilizes the SG4_TESTCASE_BEGIN and SG4_TESTCASE_END macros.

This fixture suffers from the same problems as the Styx fixture. The SG4_BEGIN method is hard coded and will always start us in a state where we have exited cold reset.

6.5 Verify Pattern Portability

To verify pattern portability you will need to use the default environment produced by genSG4.pl. See the Quick Start chapter for instructions. To test your pattern perform the following steps:

```
    cd <SG4MODEL> # the directory where you ran genSG4.pl
    make clean    # recommended by not required
    make test PP=<PATH_TO_YOUR_TEST_PATTERN>
```

If the make command succeeds your pattern will be portable to other environments. If it fails to compile you will need to correct the compilation and or runtime issues.

The PP=<PATH_TO_YOUR_TEST_PATTERN> forces the Makefile to insert the default test fixture previously described and points to the test source for compilation. It is recommended that you run 'make clean' before running the test to eliminate any previously compiled objects.

Your test pattern may use a valid C++ file extension of either '.cpp' or '.cc'. No other file extensions are supported. If you are using a different file extension try creating a symbolic link to your source file using one of the accepted file extensions. If this does not work contact the SG4 developers for assistance.

7 Integrating StimGen Into Your Environment

StimGen4 utilizes a C++ concept known as a <u>visitor pattern</u> to support easily integrating StimGen into your environment(s). To integrate StimGen you must define one or more of the StimGen::Event_Visitor derived classes that expand the various low level event types generated as a result of operations like read or write or write_comment into the format your environment can consume. There are two basic types of environments that StimGen easily supports which are either high level event based or pin level.

High level events based environments are environments that don't need to drive pins directly because they have some other tool or third party tool handling this. An example of this is AMD's Wombat or the ARM DBOX(BOZO what's the name) where these tools accept high level JTAG IR/DR shift values and they fill in the pin information through software or an FPGA or utilizing some other technology. Another example is in verification where different teams have so called "jtag agents" which provide the same functionality but for a simulation.

Pin level environments are environments where pins must be driven directly. These environments rely on StimGen to generate pin stimulus and measure information. There are a variety of environments that require this functionality. One example is getting a test pattern loaded onto an ATE tester which requires generation of STIL vectors. A second example are the ATPG tools that need to drive pins to initialize the DUT before attempting to generate test patterns.

All StimGen event visitors will derive from the StimGen::Event_Visitor class which provides support of high level events only. This class provides the foundation for generation of pin level stimulus which is handled by the derived class StimGen::Event_Visitor_Pin.

7.1 Base Class StimGen::Event_Visitor

The base class StimGen::Event_Visitor will provide default implementations for all events which simply echo that an unprocessed event was found. Its up to you to override the functionality. In general, the SG4 development team will create the support required for the different environments. The following information is provided for those users that wish to create their own Event_Visitor class.

The table below defines all of the StimGen::Events that should be handled in your visitor class

Event Type	Description
Event_ARM_AMBA3_AHB_Lite	ARM broadside AMBA event.
Event_ARM_AMBA_AXI4	ARM AXI4 event.
Event_Comment	Comment produced from StimGen::Module::write_comment()
Event_Environment	Environment specific directive produced by StimGen::Module::write_env(). Each environment event has a specific target environment to be applied from. If your Event_Visitor needs to handle the target you should do so. If it is not targeted for your environment you can safely ignore it

Event Type	Description
Event_I2C	Register read or write operation that has been routed to a specific I2C interface.
Event_IEEE1149_1_DR*	JTAG TDR read or write operation that should be processed on the JTAG DR side of the FSM.
Event_IEEE1149_1_IR*	JTAG TIR read or write operation that should be processed on the JTAG IR side of the FSM.
Event_IEEE1500_DR*	IEEE1500 WDR read or write operation that should be processed with the corresponding SELWIR signal selecting the WDR
Event_IEEE1500_IR*	IEEE1500 WIR read or write operation that should be processed with the corresponding SELWIR signal selecting the WIR
Event__IEEE802_3	IEEE802.3 register read or write operation.
Event_Loop	Loop instruction with currently limited functionality. If a protocol is associated with the loop you should issue the specified number of protocol "clock cycles" otherwise wait the specified number of environment units. If the target environment is STIL for an ATE wait the specified number of tester cycles. This may or may not work as desired in various environments
Event_Pin	Read or write a pin
Event_Register	Base functionality for all of the register based transactions. In some cases allowing Event_IEEE*_?R events to default to StimGen::Event_Register is desirable

* IEEE1149 and IEEE1500 operations are split into separate IR and DR operations because AMD DFT register networks will perform a number or register operations in the DR state because of nested/embedded IEEEP1687 and IEEE1500 operations where the nested/embedded IR operation is part of the parent IP or SOC DR operation.

7.2 Sample Event_Visitor Header

```
#include "sg_Event.h"
#include "sg_Event_Visitor.h"
namespace StimGen {
/**************************************************************************
   * Class: Event_Visitor_Debug
   *
   * Visitor class to dump event information to STDOUT for all concrete events
   *
   **************************************************************************/
  class STIMGEN_API Event_Visitor_Debug : public Event_Visitor {
    public:
```

```
/*************************************************************************
 * Constructor: Event_Visitor_Debug
 *
 * Event_Visitor_Debug Constructor
 *
 * Debug visitor should be added to Module by passing reference to
 * Module::add_event_visitor
 * Module will then own the Visitor and handle clean up
 *
 * Parameters:
 *    None
 *
 * Example:
 * (code)
 *    dut.add_event_visitor( new Event_Visitor_Debug() );
 * (end)
 *
 *************************************************************************/
Event_Visitor_Debug();

virtual ~Event_Visitor_Debug ();

/*************************************************************************
 * Function: visit ( Event_Comment* )
 *
 * Visitor method to process a comment event.
 * Do not call this method directly, it will be called from the Event_XX*
 * argument being passed in via it's overridden visit method.
 *
 * Parameters:
 *    Event_Comment* - Comment
 *
 * Returns:
 *    None
 *
 *************************************************************************/
virtual void visit ( Event_Comment* );
virtual void visit ( Event_Environment* );
virtual void visit ( Event_I2C* );
virtual void visit ( Event_IEEE1149_1_DR* );
virtual void visit ( Event_IEEE1149_1_IR* );
virtual void visit ( Event_IEEE1500_DR* );
virtual void visit ( Event_IEEE1500_IR* );
virtual void visit ( Event_IEEE802_3* );
virtual void visit ( Event_Loop* );
virtual void visit ( Event_Pin* );
virtual void visit ( Event_Register* );
                    /************************************************
*******************
 * Method: set_report_fields
 *
 *  Turn on the option to report individual bitfields
 *
 * Parameters:
```

```
     *    bool - True enables bitfield reporting and false disables it
     *
     ***********************************************************************/
    void set_report_fields( bool );

  private:
    bool report_fields;

  };
} /* namespace StimGen */
```

7.3 Sample Event_Visitor Implementation

```cpp
#include "sg_Event.h"

StimGen::Event_Visitor_Debug::Event_Visitor_Debug ()
    : Event_Visitor() {
}

StimGen::Event_Visitor_Debug::~Event_Visitor_Debug () {
}

void StimGen::Event_Visitor_Debug::set_report_fields( bool on_off ) {
  report_fields = on_off;
}

void StimGen::Event_Visitor_Debug::visit ( Event_Comment* e ) {
  std::cout << "// " << e->get_comment() << std::endl;
}

void StimGen::Event_Visitor_Debug::visit ( Event_I2C* e ) {
  std::cout << e->type() << "   "
      << e->get_register()->get_hier_name()
      << "  Slave Address = " << e->get_protocol()->get_slave_address()
      << "  load="
      << e->get_register()->get_load_value().value_as_hex_verilog() << "  meas="
      << e->get_register()->get_measure_value().as_hex_verilog()
      << std::endl;
}

void StimGen::Event_Visitor_Debug::visit ( Event_IEEE1149_1_DR* e ) {
  visit( dynamic_cast<Event_Register*>(e) );
}

void StimGen::Event_Visitor_Debug::visit ( Event_IEEE1149_1_IR* e ) {
  visit( dynamic_cast<Event_Register*>(e) );
}

void StimGen::Event_Visitor_Debug::visit ( Event_IEEE1500_DR* e ) {
  visit( dynamic_cast<Event_Register*>(e) );
}

void StimGen::Event_Visitor_Debug::visit ( Event_IEEE1500_IR* e ) {
  visit( dynamic_cast<Event_Register*>(e) );
}
```

```cpp
void StimGen::Event_Visitor_Debug::visit ( Event_IEEE802_3* e ) {
  visit( dynamic_cast<Event_Register*>(e) );
}

void StimGen::Event_Visitor_Debug::visit ( Event_Loop* e ) {
  if ( e->has_protocol() ) {
    Pin* clk = e->get_protocol()->get_clock();
    std::cout << "Loop " << clk->get_name() << " " << e->get_count() << std::endl;
  } else {
    std::cout << "Loop " << "System Clock " << e->get_count() << std::endl;
  }
}

void StimGen::Event_Visitor_Debug::visit ( Event_Pin* e ) {
  std::cout << ((e->is_read()) ? "Read:  " : "Write: ")
       << e->get_pin()->get_name()
       << " = "
       << e->get_load_value().as_hex_verilog()
       << std::endl;
}

void StimGen::Event_Visitor_Debug::visit ( Event_Register* e ) {
  Register* r = e->get_register();
  Number load_value;
  Number measure_value;
  r->append_load_and_measure_values(load_value, measure_value);
  std::cout << e->type() << "   "
       << r->get_hier_name() << "  load="
       << load_value.value_as_hex_verilog() << "  meas="
       << measure_value.as_bin_verilog()
       << std::endl;
  load_value.fill_mask();
  std::list<BitField*> mismatches;
  e->set_read_value_and_compare(measure_value, mismatches);

  for (std::list<BitField*>::iterator i = mismatches.begin();
     i != mismatches.end();
     i++
     ) {
    std::cout
    << (*i)->get_hier_name()
       << " bits "
       << (*i)->get_width() + (*i)->get_offset() - 1
       << ":"
       << (*i)->get_offset()
       << " mismatch got("
       << (*i)->get_read_value().as_hex_verilog()
       << ") expected("
       << (*i)->get_measure_value().as_hex_verilog()
       << ")"
       << std::endl;
  }
```

```cpp
if ( report_fields ) {
  std::list<const Register*> active_registers;
  e->get_register()->get_visible_registers(active_registers);
  int width = e->get_register()->get_width();

  std::cout << "  Register Bitfields:\n";
  std::cout << "  TDI --> \n";
  for (std::list<const Register*>::iterator i = active_registers.begin();
      i != active_registers.end();
      i++
      ) {
    if ( !(*i)->is_virtual() ) {
      std::vector<BitField*> bf = (*i)->get_bitfields();
      for (std::vector<BitField*>::iterator j = bf.begin();
          j != bf.end();
          j++
          ) {
        std::cout
        << "     "
           << width - 1
           << ":"
           << width - (*j)->get_width()
           << " "
           << (*j)->get_hier_name()
           << " = "
           << (*j)->get_load_value().value_as_hex_verilog()
           << "   "
           << (*j)->get_read_value().value_as_hex_verilog()
           << std::endl;
        width -= (*j)->get_width();

      }
    }
  }
  std::cout << "  --> TDO\n";
  }
}
```

7.4 Sample Event_Visitor Integration

Now that we've defined and implemented the visitor pattern Event_Event_Visitor_Debug we need to integrate it into the stimGen model. To do this simply add it to the top level module using method

> void StimGen::Module::add_event_visitor (Event_Visitor&);

```cpp
    StimGen::ModuleFactory factory ( "sgxml", "." );
    factory.import( "SOC.xml", "", &dut);
    StimGen::Event_Visitor_Debug evd;
    evd.set_report_fields(true);
    dut.add_event_visitor(evd);
```

Lines 1-2:   Create and load the top level module from sgxml

Line 3: Create a local instance of the desired visitor pattern. Make sure that it remain in scope until the test pattern has been completely processed.

Line 4: Turn on the debug visitor patterns bitfield reporting

Line 5: Add the instance of the visitor pattern to the dut. The dut does not take ownership of the visitor pattern. The user is responsible for any clean up and destruction of the visitor pattern. In this case default destruction is all that is necessry.

Once you've added one or more event visitors to the top level module each visitor pattern will be called for each event that is created from the Module::apply() method.

8 Pin Based Event Visitor Patterns

If your environment requires access to individual pin data you will need to utilize the Event_Visitor_Pin pattern. This pattern builds on the Event_Visitor pattern. StimGen controls expanding all protocol operations into their pin stimulus equivalents. This is built into StimGen so that there is one consistent conversion for all environments. This is important to ensure that all environments treating these operations consistently and it eliminates other tools performing the same functionality.

When the DUT detects that an event visitor pattern derived from Event_Visitor_Pin is added to the model it will automatically insert the Event_Visitor_Pin_Expander visitor pattern to the list of visitor patterns. The Event_Visitor_Pin_Expander is responsible for conversion of all high level events to their equivalent pin operations. Conversions involve parallelizing events and adding "time" into the picture.

StimGen's primary use model is vector generation of ATE test patterns so much of StimGen are build on ATE requirements. Time on the ATE is based on the system reference clock and one tester vector typically corresponds to one reference clock. StimGen uses a vector to represent time so all pin events will be represented in terms of vector number and value pairs. (Don't confuse std::vector with this vector). These will be referred to as the vector queue going forward.

The Event_Visitor_Pin_Expander will provide all of the necessary functionality to convert all events into the pin data in the vector queue. There is only one vector queue created to service all pin based visitor patterns and it therefore provides read-only access to all data to guarantee data integrity. The Event_Visitor_Pin_Expander provides the implementation for the previously discussed visit methods which fill the vector queue and does not allow derived patterns to override these implementations. Event_Visitor_Pin derived patterns can not use the high level events because there is no time associated with these events. The Event_Visitor_Pin_Expander adds time or vector numbers to the operations and parallelizes them. The data in the vector queues is not usable until all events in the event queue have been processed. When all data is available the Event_Visitor method 'apply' will be executed allowing each visitor pattern to process the contents of the vector queue.

8.1.1 Vector Queue

The vector queue is a class that manages three std::maps where the key of each is the vector number that synchronizes pin assignments, comments and environment directives. Each of these maps is defined as

```
typedef std::map<unsigned int, std::string> Comment_Vector_Queue;
typedef std::map<unsigned int, std::map<std::string, std::string> >
                                        Environment_Vector_Queue;
typedef std::map<unsigned int, std::map<Pin*, char> > Pin_Vector_Queue;
typedef std::map<Pin*, char> Pin_Char_Map;
```

The vector queue will contain all pin operations for the specified vector and will contain only information where the pin state changes. All comment and environment directives should always be processed before processing pin information.

8.1.2 Event Visitor apply Method

The Event Visitor's apply method all all visitor patterns to queue up data and dump it all at once. In the case of pin event visitor patterns it indicates when all pin data is available for processing. The following sample code should be used for custom pin based event visitor pattern apply implementation.

NOTE: THIS APPLY() EXAMPLE IS OUT OF DATE. SPECIFICALLY, IT DOES NOT COMPREHEND FREE RUNNING CLOCKS BUT ALSO IS NOT CYCLE ACCURATE WITH OTHER PIN BASED VISITORS WHICH IS A MAJOR CONCERN FOR PORTING PATTERNS BETWEEN DIFFERENT ENVIRONMENTS AND SEEING THE SAME BEHAVIOR.

```cpp
void StimGen::Event_Visitor_DumpInfo::apply () {

  // Reset the class comment and environment iterators
  comment_iter = vector_queue->get_comment_queue().begin();
  environment_iter = vector_queue->get_environment_queue().begin();

  // Check for comments and environment directives before
  // we start processing pin events.  If there are no pin
  // events these will get processed here.

  process_comments(0);
  process_environment_directives(0);

  unsigned int vector = 0;

  // Process all pin information
  for (Pin_Vector_Queue::const_iterator i
        = vector_queue->get_pin_queue().begin();
     i != vector_queue->get_pin_queue().end();
     i++
     ) {
    vector = i->first;
    ofh << "vector=" << (tester_vector + vector) << "\n";
    process_comments(vector);
    process_environment_directives(vector);

    unsigned int id;
    for ( Pin_Char_Map::const_iterator j = i->second.begin();
        j != i->second.end();
        j++
    ) {
      ofh << j->first->get_name() << "=" << j->second << "\n";
    }
  }
  // Keep track of tester vectors across all apply statements
  tester_vector += vector + 1;
      // Update tester_cycle + 1 for the completion of last vector

}

void StimGen::Event_Visitor_DumpInfo::process_comments (
  unsigned int vector
) {
  if ( comment_iter != vector_queue->get_comment_queue().end() ) {
    if ( vector >= comment_iter->first ) {
      ofh << "$comment\n";
      ofh << comment_iter->second << "\n";
      ofh << "$end\n";
      comment_iter++;
    }
  }
}

void StimGen::Event_Visitor_DumpInfo::process_environment_directives (
```

```
      unsigned int vector
  ) {
    if (    environment_iter
         != vector_queue->get_environment_queue().end()
    ) {
      if ( vector >= environment_iter->first ) {

        for (Environment_Vector_Queue::const_iterator
            e = environment_iter->second.begin();
            e != environment_iter->second.end();
            e++
        ) {
          if ( e->first == "DumpInfo" ) {
            ofh << "  " << e->second << "\n";
          }
        }
        environment_iter++;
      }
    }
  }
```

Line 1:   Implement the apply method which will be called after all queue events have been processed and expanded by the Event_Visitor_Pin_Expander.

Line 4-5:   Get fresh iterators to the first comment and environment queue

Line 11-12:   Process the first set of comment and environment operations if there are any at time zero. There may be events that contain only comments or environment operations so we can't rely on the loop starting on line 18 to include them

Line 14:   The apply method will be called each time that the user issues apply in their pattern. The events in the queue are atomic and there is no knowledge of previous operations to the current apply so each apply effectively starts at vector 0. If you need to accumulate time you visitor pattern must do so.

Line 17:   Iterate through all pin events that have been inserted into pin queues. Remember Pin_Vector_Queue's key is an unsigned int for the time when the next set of pin changes occur.

Line 22:   Get the vector number associated with this set of pin changes.

Line 23:   This visitor pattern is accumulating vectors so report the current vector number

Lines 24-25:   Process any comments and environment directives scheduled at the current vector

Line 28-32:   Iterate over all of the pins that have been assigned a new value and dump them to the file

Line 36:   Now that we have exited the loop update the accumulated vector count.

Lines 41-52:   Insert into the output all of the comments available at the current vector count. There may be more than one if the pattern has issued multiple write_comment commands

Lines 54-74:   Insert into the output all environment directives that target the DumpInfo visitor pattern. Make sure that you skip directives that are not for your visitor pattern and continue searching while the current vector in the environment queue is less than the vector passed in.

9   Available Visitor Patterns

9.1 Abstract Event Visitor Patterns

9.1.1.1 Event_Visitor_Debug

The Event_Visitor_Debug is an event based visitor pattern design to generate STDOUT describing events generated.

9.1.1.2 Event_Visitor_Checker

Visitor pattern that is filled with a list of expected events and compares that list against what StimGen produces.   The checker visitor pattern is used extensively for regression tests to ensure that StimGen is producing vectors as expected.

9.1.1.3 Event_Visitor_HSTXML

Generates a XML file containing JTAG and Pin operations to be executed on AMD ATE testers.

9.1.1.4 Event_Visitor_Qinit

Event Event_Visitor_Qinit is an event based visitor pattern generating ATPG initialization sequences for ATPG QuickInit tool.

9.1.1.5 Event_Visitor_SVF

Generates industry standard serial vector format (SVF) for delivery of vectors to tools or external customers.

9.2 Pin Based Event Visitor Patterns

9.2.1.1 Event_Visitor_ATPG_Mentor

Generates Mentor stimulus for ATPG pattern generation

9.2.1.2 Event_Visitor_ATPG_Synopsys

Generates Synopsys stimulus for ATPG pattern generation

9.2.1.3 Event_Visitor_EVCD

The Event_Visitor_EVCD is a pin based event visitor pattern that will create an Enhanced Vector Change Dump or EVCD file for viewing of waveforms in tools like gtkWave or Debussy.

9.2.1.4 Event_Visitor_STIL_Advantest_93K

The Event_Visitor_STIL_Advantest_93K is a pin based event visitor pattern designed to generate STIL for Advantest 93K Digital ATE testers.

9.2.1.5 Event_Visitor_STIL_Credence

The Event_Visitor_STIL_Credence is a pin based event visitor pattern designed to generate STIL for Credence Sapphire ATE testers.

9.2.1.6 Event_Visitor_TBV

Generate simple cycle by cycle pin stimulus to be replayed in simulation environment utilizing StimGen TBV testbench.

10   Batch-Mode Event Visitors

The original event visitors implemented in StimGen fully processed each event in turn; i.e. the load_value was loaded into the target, and read_value was extracted from it during processing of the event. When event n+1 was processed, processing of event n (and all earlier events) was complete.

Some event visitor implementations can benefit greatly from batch-mode processing. For example, The Wombat debug adapter allows multiple shifts to be sent to it in one TCP/IP packet, thereby incurring the round-trip network latency only once for a group of shifts. To support this, a new event visitor base class was added: *Event_Visitor_Batched*. Visitors supporting batch-mode operation are derived from Event_Visitor_Batched instead of Event_Visitor, and they must define the various event-specific *process* methods. See the in-line documentation in Event_Visitor_Batched for further details.

10.1   Read Callbacks

A batch-mode event visitor must process the return values of the batch in-order after the batch has executed. This will update the read_value on the affected objects within the StimGen model. But if the batch includes multiple accesses to a single object (for example, repeated reads to the NBSPRACCESS register, as would be the case with batched reads via SPR back-door access), only the final read_value will be visible in the StimGen model after execution of the batch; all of the other read_value's would be lost. If the intermediate read_values are needed, the user must use read callbacks to get them. A read callback is an object instance which implements the ICallback interface, which defines a single *invoke* method. This is best illustrated by example:

```
class STIMGEN_API myReadCallback : public ICallback
{
  void invoke(void)
  {
    // Do something useful here
  }
  // Other members will be needed to facilitate appropriate actions
};

…

Register *r = dut->get_register("NBSPRACCESS");

myReadCallback cb1;
r->read(RegisterAccessOptions::Read_Callback(&cb1));

myReadCallback cb2;
r->read(RegisterAccessOptions::Read_Callback(&cb2));
```

In this example, cb1's invoke method will be called by StimGen at a time when the first read has been completed (and NBSPRACCESS's read_value updated), and cb2's invoke method will be invoked when the second read has been completed. Through this mechanism, all of the intermediate read values can be extracted.

11 Verification Integration

In order to utilize StimGen in verification we must be able to build the model on the fly and pull data from various locations to build a directory structure similar to what SIGAPI expects (a simplified version of /proj/sigdata that contains all required information). The /proj/sigdata repository is not acceptable for verification purposes because it is not guaranteed to be aligned with the current RTL that verification is working with. As a result IP and SOC teams must store StimGen source information with their RTL. This gets further complicated by the fact that there are multiple build environments.

Regardless of the environment teams must supply register definitions either in the form of SIGAPI compliant Open Office DFT documents or SGXML schema compliant XML. If the IP or SOC has custom sequences (header and or source files) these must be stored relative to these source documents.

11.1   SIGBASE Directory Structure

All IP/SOC environments must have the same basic directory structure to use StimGen. We will use variable $SIGDATA to define the base location of the information for use within this documentation. It is strongly recommended that $SIGDATA point to a directory 'sigdata' which contains the required directory structure and be consistent with /proj/sigdata. This also provides a hint as to what information can be found in this directory.

Table 1: Basic Directory Structure Definition

Directory	Description
$SIGDATA/dft/doc	SIGAPI compliant Open Office DFT documents
$SIGDATA/dft/sgxml	SGXML schema compliant XML
$SIGDATA/dft/SG4/include	C++ header class defining custom sequences
$SIGDATA/dft/SG4/src	C++ implementation for custom methods
$SIGDATA/dft/SG4/t	gtest based unit/regression tests

11.2   ALE/Module Environment

ALE and modulefile based environments may put their $SIGDATA anywhere they desire in their repository so long as the build environment is able to publish to a defined location.

11.3   CDS Compliant Environment

All StimGen data will be stored under $STEM/src/meta/sigdata/dft ($SIGDATA=$STEM/src/meta/sigdata). The directory structure for IP and SOCs will be as follows

Table 2: CDS Directory Structure Definition

Directory	Description
$STEM/src/meta/sigdata/dft/doc	SIGAPI compliant Open Office DFT documents

Directory	Description
$STEM/src/meta/sigdata/dft/sgxml	SGXML schema compliant XML
$STEM/src/meta/sigdata/dft/SG4/include	C++ header class defining custom sequences
$STEM/src/meta/sigdata/dft/SG4/src	C++ implementation for custom methods
$STEM/src/meta/sigdata/dft/SG4/t	gtest based unit/regression tests

The sigdata directory structure will be referred to as the "CDS SIGDATA Repository" going forward.

11.4   StimGen Design Data

The source of StimGen data will always be the SGXML. If SIGAPI compliant Open Office DFT documents are provided they will need to be run through SIGAPI script genSG4.pl to generate the SGXML.

```
$SG4COREBASE/bin/genSG4.pl -proj <PROJECT> \

        -dft <PATH_TO_ODM> \

        -outputdir <OUTPUT_DIR> \

        -gen_sgxml
```

If the current IP or SOC has external IP dependencies these IPs must be checked out into the build environment or stimGen will pull default information from /proj/sigdata which is most likely not desirable.

If the IP has SGXML it should be available in

        $STEM/import/<IPNAME>/src/meta/sigdata/dft

If the IP must be built and the SGXML generated it should be available and published by the IP to

        $STEM/out/<SOME_PATH>/IP/pub/src/meta/sigdata/dft

When SGXML for the top level IP or SOC are produced they must all be published into

        $STEM/out/amd64rh3.0.VCS/<PROJECT>/common/pub/src/meta/sigdata/dft/sgxml

To support IP resue SGXML utilizes XML xi:xinclude directives to pull in existing SGXML. The XML rules stipulate that the files which are referenced by xi:xinclude are specified either as an absolute or relative path. Environment variables or command line options such as -I for g++ cannot be used. In order to prevent parent IPs or SOCs from hard coding paths to a specific version of an IP SGXML the design process will publish all final SGXML to a single directory, allowing all xi:xinclude directives to reference the file as though its in the same directory. All xi:xinclude elements in the generated XML should assume that the file to be included is in the directory and only specify the file name. This will require that all files be correctly and uniquely named to avoid collisions but simplifies many other aspects of the build the process and publishing into the primary SIGDATA repository for other teams to utilize.

If the StimGen model of an IP or SOC include custom sequences and regression tests all header files and source files must be published to

$STEM/out/<mark>amd64rh3.0.VCS</mark>/<variant>/common/pub/src/meta/sigdata/dft/SG4/include

$STEM/out/<mark>amd64rh3.0.VCS</mark>/<variant>/common/pub/src/meta/sigdata/dft/SG4/src

$STEM/out/<mark>amd64rh3.0.VCS</mark>/<variant>/common/pub/src/meta/sigdata/dft/SG4/test

11.4.1   Requirements

It is the responsibility of the build environment to gather all stimGen data files (sgxml, SG4/include, SG4/src) and publish the files to the final destination.

11.4.2   Validating XML

When all sgxml has been published into the target directory it should be checked with xmllint to ensure it is complete and valid. To run xmllint run the command

```
/tool/pandora64/.package/libxml2-2.6.31/bin/xmllint \
  --schema $SG4COREBASE/../sigtools/schema/SGXML.xsd \
  --noout \
  --xinclude
$STEM/out/amd64rh3.0.VCS/<PROJECT>/common/pub/src/meta/sigdata/dft/sgxml/<PROJECT>.xml
```

If there are errors they must be corrected before continuing.

11.4.3   Building and Testing the StimGen Model

StimGen loads SGXML directly to build a model that contains only register definitions. There are however several IPs and SOCs which contain custom sequences, override default functionality or provide register access sequences for registers not directly accessible through a test interface such as JTAG. The customizations require that IP and or SOC models provide custom C++ class definitions and method implementations. These are all stored in the SG4/include and SG4/src directories. To build a working and usable model, we must gather these files up and put them into the build environment. If such files exist they will be compiled into their own shared library so that they can be compiled and linked in at runtime which is essential for several environments.

To build the StimGen model use command

```
$SG4COREBASE/bin/genSG4.pl \
        -proj <PROJECT><IPREV> \
        -sgxml $STEM/out/amd64rh3.0.VCS/<PROJECT>/common/pub/src/meta/sigdata/dft/sgxml \
        -outputdir $STEM/out/amd64rh3.0.VCS/<PROJECT>/common/pub/src/meta/sigdata/dft/build \
        -gen_sgxml
```

This local build can be built and tested standalone. It is recommended that the larger build process perform a test build of the SG4 model to ensure there are no problems prior to integration into other environments. To build the test model run the command

```
cd $STEM/out/amd64rh3.0.VCS/<PROJECT>/common/pub/src/meta/sigdata/dft/build \
```

&& make test

If IPs or SOCs StimGen models have accompanying regression tests, they will be run as part of this test. If any test fails contact the test owner. You should not continue if these regression tests fails and are not understood.

If a test fails or you wish to run only a specific test run with options

   `<PROJECT>_test --gtest_filter="FILTER"`

For more details about gtest visit http://code.google.com/p/googletest/

11.4.3.1   <PROJECT>_test

genSG4.pl will generate a simple build model based gtest t/<PROJECT>_test.cpp. To run this test standalone run

   `<PROJECT>_test –gtest_filter="<PROJECT>_TEST.ModelBuild"`

Here is a sample of what the test will contains where <PROJECT>=SOC:

```
1.    #include <iostream>
2.    #include "SOC.h"
3.    #include "gtest/gtest.h"
4.
5.    using namespace StimGen;
6.
7.    TEST ( SOC_TEST, ModelBuild ) {
8.      StimGen::SOC *dut;
9.      EXPECT_NO_THROW( { dut = new StimGen::SOC(); } );
10.     EXPECT_NO_THROW( { delete dut; } );
11.     dut = NULL;
12.   }
```

Lines 1-3: Pull in the necessary header files

Line 5:   Add StimGen to the current namespace

Line 7:   Create a gtest SOC_TEST.ModelBuild

Line 9: Create an instance of the <PROJECT> and make sure that no exceptions get thrown. If an exception is thrown there is a problem with the source documentation.

Line 10: Call the SOC destructor and ensure no exceptions are thrown. If an exception is thrown there is a problem with the StimGen core code. Contact StimGen development team.

11.4.3.2   <PROJECT>_main

A standalone application <PROJECT>_main will also be automatically created in t/main/<PROJECT>_main.cpp and can be used to quickly code up a test pattern. The application will allow you to code up test patterns and ensure that you have no issues with register or bitfield names. Debugging these names in the environment is much faster and easier in this environment than from within the environment StimGen is being integrated.

Here is a sample <PROJECT>_main where <PROJECT>=SOC

```cpp
1.      #include <iostream>
2.      #include "SOC.h"
3.
4.      using namespace StimGen;
5.      int main ( int argc, char** argv ) {
6.        StimGen::SOC *dut;
7.
8.        try {
9.          dut = new StimGen::SOC();
10.         std::cout << *dut << std::endl;
11.
12.       } catch ( exception &x ) {
13.         std::cout << "Exception: " << x.what() << std::endl;
14.       } catch ( const char* const &x ) {
15.         std::cout << "Exception: " << x << std::endl;
16.       } catch ( const std::string &x ) {
17.         std::cout << "Exception: " << x << std::endl;
18.       } catch ( ... ) {
19.         std::cout << "default exception caught" << std::endl;
20.       }
21.
22.       delete dut;
23.       dut = NULL;
24.
25.       return 0;
26.    }
27.
```

Lines 1-2:   Pull in the necessary header files

Line 4:   Add StimGen to the current namespace

Line 5:   Implement 'main'

Line 6:   Create a pointer to the <PROJECT> (StimGen::SOC)

Line 8-20:   Wrap code that utilizes StimGen in a try/catch block so that we can report on any problems encountered

Line 9:   Create an instance of the <PROJECT>

Line 10:   Print the StimGen model which will print all instances and all registers

Line 12-20:   Catch any exceptions that might be thrown

Line 22-23:   Delete the <PROJECT> after we've handled any exceptions so that dut contains valid information for error reporting.

To build and run <PROJECT>_main run command
    make <PROJECT>_main && ./<PROJECT>_main

11.4.4   Generating Documentation

12   SG4 System Verilog Sequencer

12.1   Introduction

There are three use models for SG4 in verification.

13. Verification of the design
14. Generation of reusable IP/SOC sequences for reuse at higher levels of integration or by other teams
15. Generation of complete IP/SOC test patterns to be delivered to other teams

Item one has been the primary focus of verification. Items two and three are moving center stage.

SG4 can be used in many ways, but the desired model mimics a producer consumer model where IP design and verification are the initial producer of SG4 data and sequences.   These are consumed by other teams working with that IP(e.g. ATPG, PPE) and by teams that are integrating the IP. This will continue up to the SOC that will produce SOC level content to be consumed by silicon validation, ATE, SLT and other back-end teams.

Each team working on a design has a slightly different view of the DUT. Wrappers and layers of abstraction are typically added to customize and simplify interaction with the DUT. SG4 is designed to generate stimulus and measure data for a variety of DUT views. The common interface to all views are the pins. Pins can be grouped together to implement protocols or higher levels of abstraction. SG4 takes high level read and write operations and boils that down to a set of pin assignment and measure operations. The DUT pin interface provides the foundation for all SG4 usage and is the common intersect for all SG4 customers.

SG4 is able to generate a stand-a-lone verification environment. By design, this environment does not use any existing verification infrastructure but is designed to integrate it or be integrated into it. It may or may not be appropriate to use this auto-generated environment. Small IPs can certainly use them. SOC teams delivering patterns to verification may use this to strip the simulation environment to the bare bones mimicking a tester. In other environments you may need to slice open the environment to implant SG4 into it. This chapter will provide explanation of the SG4 DUT interface, environment, provide details on how to use the test benches SG4 provides and how to integrate SG4 into custom environments.

12.2   Assumptions

SG4 makes no assumptions about the environment that it is being integrated into. It'll generate a stand-a-lone environment and the components to integrate into other environments. SG4 builds on the vcs command line options and not AMD build systems. The expectation is that the integrating team will take these options and plug them into the appropriate build system.

When testing silicon on a tester we can only access the DUT through its pins. SG4 views and treats the DUT exactly the same way. It is the DUT pin interface definition that is used to build the stand-a-lone environments and facilitate communication between SG4 and System Verilog. SG4 requires that the pin interface of the DUT is fully defined and assumed that SG4 will be able to drive and measure any

of the interface pins.

Test patterns must all be authored using [SG4 Portable Pattern Rules](#). This requires that the test pattern have no knowledge about its runtime environment and only use SG4 directives and C++98(soon to be C++11) standard include files. Any directives that need to be passed to the runtime environment must utilize the SG4 Module::write_env command.

12.3   Integration of SG4 into System Verilog

SG4 supports generating two types of stimulus:

28. Pin: SG4 generates vectors entirely at the pin level duplicating vectors generated for and applied at silicon test.

29. Event: SG4 generates events that provide higher levels of data abstraction based on well defined and industry standard interfaces, in addition to pin operations. Event based stimulus allows existing infrastructure to provide implementations. This may or may not match what is applied at wafer and package test.

Depending on your desired use model, there are several ways to integrate SG4 into System Verilog. All environments must enable SG4 to drive and measure any DUT pin. If this functionality is not provided SG4 cannot be fully utilized and it will be a matter of time for obstacles to appear. It is recommended that the SG4 DUT interface described below is inserted to provide access to any and all interface pins.

When SG4 is used at the pin level, integration is fairly straight forward and SG4 drives the simulation. The pin environment is designed to mimic silicon test on a tester and provide a slim, fast and lite weight environment. SG4 should be the only object driving pins in the test bench. You are encouraged to add monitors to spy and interpret operations being applied to the DUT.

There is significantly more flexibility, opportunity and complexity when integrating SG4 into an event based environment. SG4 provides a complete solution that can be used or allows you to choose components to integrate into your environment. The one exception is the communication channel between SG4 and System Verilog which is the SG4 SVSequencer_Event visitor pattern and its companion System Verilog class sg4_base_virtual_sequence. You must create a class that derives from this base class and implements the various event processing tasks and functions. These tasks or function are required to do the translation of event to pin wiggle(s). There are many event types that SG4 supports and generates and all must be supported or runtime errors will be generated if unimplemented event support is invoked.

12.4   System Verilog SG4 DUT Pin Interface

All communication between SG4 and the DUT is through its interface. SG4 will generate a System Verilog interface for the DUT based on the documented pin list. The interface provides the ability to drive and measure pins providing the baseline functionality to build on. All of the SG4 System Verilog drivers and monitors operate through this interface. If the interface is changed by adding, removing or renaming a pin the interface and the System Verilog test bench related files must be rebuilt to ensure they are in sync.

The interface defines an array of logic objects for driving pins and a wire bus for measuring pins. Each

pin is assigned the same index in the driver and measure bus. All pins are sorted first by type (output, input, inout, etc) and then alphabetically by bump name within its type. Once sorted each pin is assigned an index into the logic array and bus. This sorting algorithm and pin assignment is use in both the SG4 visitor patterns and the System Verilog DUT interface to ensure alignment. (BOZO do we need to add sanity tests?)

Protocols defined by the documentation will also be added to the DUT interface. The protocol pins will reference the DUT interface pins to drive and measure them appropriately. Event packets will interact with the protocol interfaces when SG4 drivers are used. The protocol interfaces provide higher levels of abstraction to simplify communication with the DUT. When using the pin based environment these interfaces are not driving pins but may be monitoring and reporting activity on their interface pins.

12.5   SG4 System Verilog Test Benches

12.5.1   Event Test Bench

SG4 provides an event visitor pattern and companion System Verilog class to transmit event packets bidirectionally. SG4 will create the packet, pass it to System Verilog to apply to the DUT and then pass results back to SG4.

12.5.1.1   Top Level Loop

Illustration 5 SG4 System Verilog Pin Based Test Bench shows the high level processing and communication between SG4 and the System Verilog sg4_base_event_virtual_sequence::body task. The hexagon shapes indicate points where the process will block until the other passes control back. The red dotted lines show where the other process will pass control from and to.

A Verilog initial block must instantiate the System Verilog sg4_base_event_virtual_sequence derived class and start it. The body task will fork to create two processes:

1. SG4 google test framework that will launch the specified SG4 portable test pattern SG4_TEST_F. See Portable Pattern Rules for details.

2. System Verilog Event Sequencer that provides queues for SG4(process #1) to fill and then forks processes to handle processing of the event packets.

Control will be passed back and forth between the two processes. The System Verilog code will immediately block until SG4_TEST_F has loaded the queues and an apply has been issued. The SG4_TEST_F utilizes exported System Verilog DPI functions to pass event packets across so that it consumes no simulation time.

When an apply is issued control will be passed to the System Verilog sg4_event_base_virtual_sequence by setting a flag. The System Verilog sg4_event_base_virtual_sequence::body will then process all events and upload results to SG4 for post processing and checking. It will then pass control back to SG4_TEST_F to repeat the process until the entire pattern has been processed, at which point, the two processes will complete and join to end the simulation.

Intial
sequencer.begin

Fork SV Handler
And
SG4_TEST_F

SG4_TEST_F

Protocol[0]
Data
Queue

Protocol[N]
Data
Queue

Write/ Read
Apply

Apply will fill System
Verilog Protocol
Data Queues via
exported DPI
functions

Wait for
gtest_done
or
SG4 Pending
Events?

Assert SG4 Pending
Events

Pending Events

Fork Protocol Handlers

Protocol[0]
Process

Protoxol[N]
Process

gtest_done

Join
Upload results.
Clear SG4 Pending
Events

More operations to perform

Wait for
processing
to complete

SG4_TEST_F
Done

Done

SV Done (clean-up)

Set gtest_done=1

Join
Report Pass/Fail

Illustration 3: SG4 and System Verilog Process Event Handler Flow Chart

12.5.1.2   Event Processing Loop

A System Verilog event packet is defined for each SG4 event type. The SG4 event visitor pattern creates the packet via DPI function calls (Note that a Verilog function consumes zero simulation time.) These functions build a packet and push it into the appropriate queue for processing. For protocol based events the packet is pushed into the queue for the target protocol. All other event packets will be inserted into a global synchronization queue.

Each SG4 event is assigned an unique event ID. No two events within an apply will be assigned the same event ID. If SG4 merges events together you will see noncontinuous or gaps in event IDs. There is no event memory in SG4 that spans an apply statement so the event ID assigned to new events will restart at zero.

After all event packets have been queued, the System Verilog sg4_base_virtual_sequence class method process_events will fork to create a separate process for each protocol. Each process will pass one event packet to its handler for processing. Upon completion it will return results and wait for the next event packet to process. The sg4_base_virtual_sequence::process_events child processes will feed events to handler until it must synchronize events. Synchronization is required when the current event ID is larger than the next synchronization event ID. The parent process manages the synchronization queue and will block until all child processes pause or finish processing. The parent process will then process all consecutive synchronization event(s) and then trigger an event allowing all child processes to resume.

12.5.1.3   Event Handlers

The event handlers will receive as its only argument the corresponding event packet. The task or function must pass the appropriate data to the desired driver and insert results. SG4 will provide a default driver for each packet type which may or may not be used. That is up to the integrating team to decide. The following table describes all of the event processing tasks and functions that must be implemented.

Table 3: SG4 System Verilog Event Handlers

Name	Description
Tasks	
process_event_sg4_arm_amba3_ahb_lite	Tasks that must be overridden and connected to the appropriate handler/driver. Each protocol handler will receive a packet for processing from the sequencer. It must complete the processing and return. All protocol handlers will be operating in parallel. The sequencer will schedule perform any synchronization required with other SG4 event types.
process_event_sg4_amr_amba_axi4	
process_event_sg4_ieee1149_1	
process_event_sg4_ieee1500	
process_event_sg4_ieee802_3	
process_event_sg4_i2c	
process_event_sg4_loop	Insert specified number of system clocks

Name	Description
process_event_sg4_pin	Drive or measure a pin
Functions	
process_event_sg4_comment	Insert comments for the the source pattern. These are important to debugging patterns and understanding progress of pattern execution.
process_event_sg4_environment	Pass environment specific directives from the source pattern to the environment.

12.5.1.4   *Event Test Bench Block Diagram*

Illustration 4 below shows the pure System Verilog test bench produced by genSG4.pl. The base System Verilog class sg4_event_base_virtual_sequence handles generating the main process to run the Google test fixture with the selected SG4_TEST_F and a second process to receive the SG4 event packets and distribute them to the process_event methods implemented by the derived sg4_event_virtual_sequence class. SG4 is instantiated in Google test fixture. SG4 provides a default implementation which passes information to the correct SG4 interface to handle conversion of the packet to pin drive and measure operations. When plugging in custom event handlers the process method must pass the appropriate data to and from the appropriate handler.
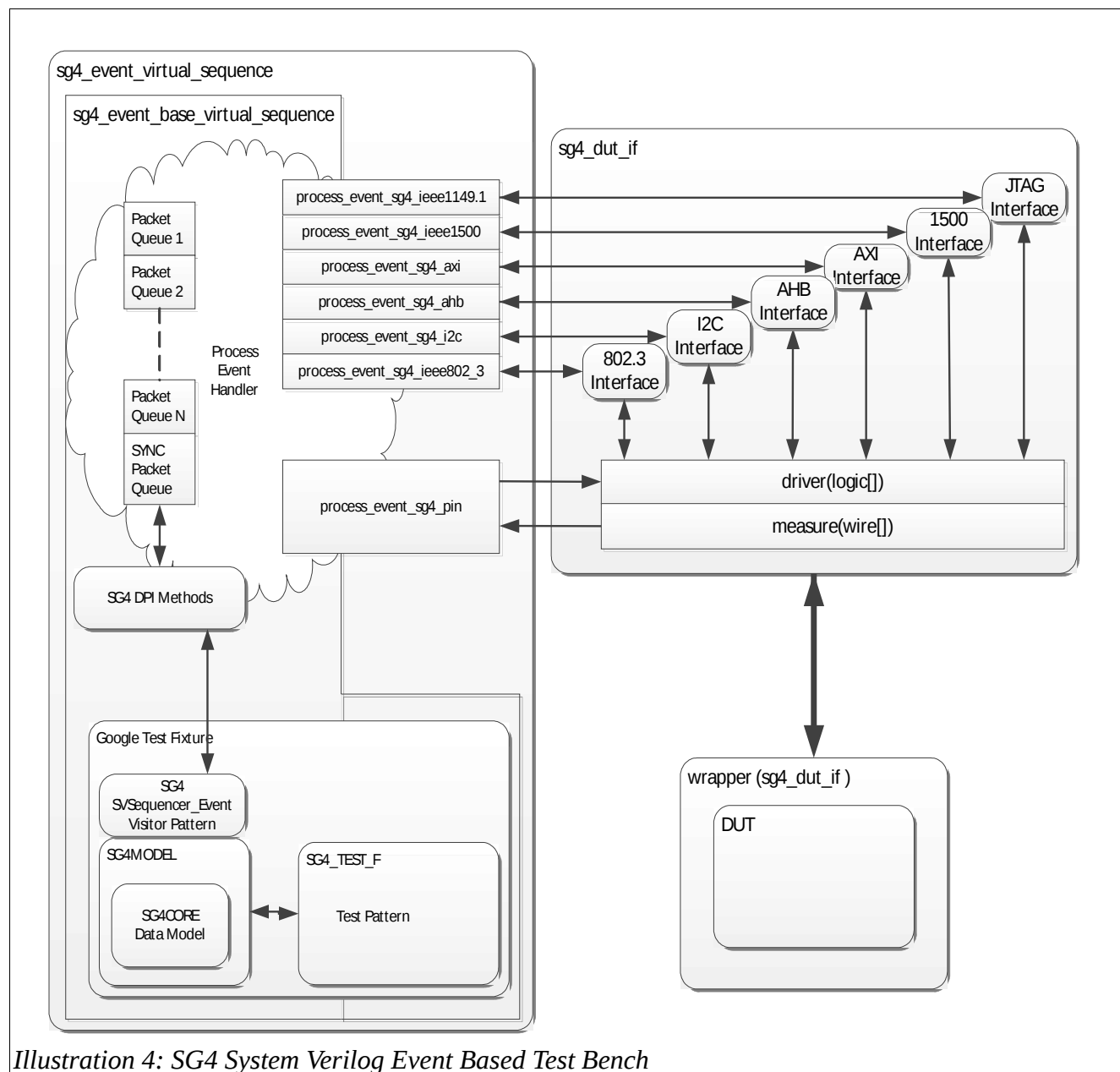
Illustration 4: SG4 System Verilog Event Based Test Bench

12.5.2   Pin Test Bench

SG4 provides a pin visitor pattern and companion System Verilog class to transmit pin drive and measure operations. When SG4 is converting events to pin drive and measure operations, it utilizes the SG4 DUT pin interface to do so.

The DUT interface generates a simulation tick which SG4 will use to drive and measure pins at the appropriate time. A tick corresponds to a single tester vector and is broken up into two edges: a rising and falling edge. This allows SG4 to convert pulses of either PV_P or PV_p to the appropriate levels. SG4 will drive and measure pins via System Verilog exported functions (zero simulation time) and then call a System Verilog task(time consuming) to advance to the next edge. This will be repeated until the

AMD ◢

entire SG4_TEST_F has been processed.

Illustration 3 shows the genSG4 System Verilog pin based test bench. It has the same basic structure as the event based test bench, however, it uses the DUT interface drive logic array and measure bus. In this environment, SG4 converts events to pin operations. Instead of using the event based visitor pattern and companion System Verilog event virtual sequence it utilizes pin based versions.
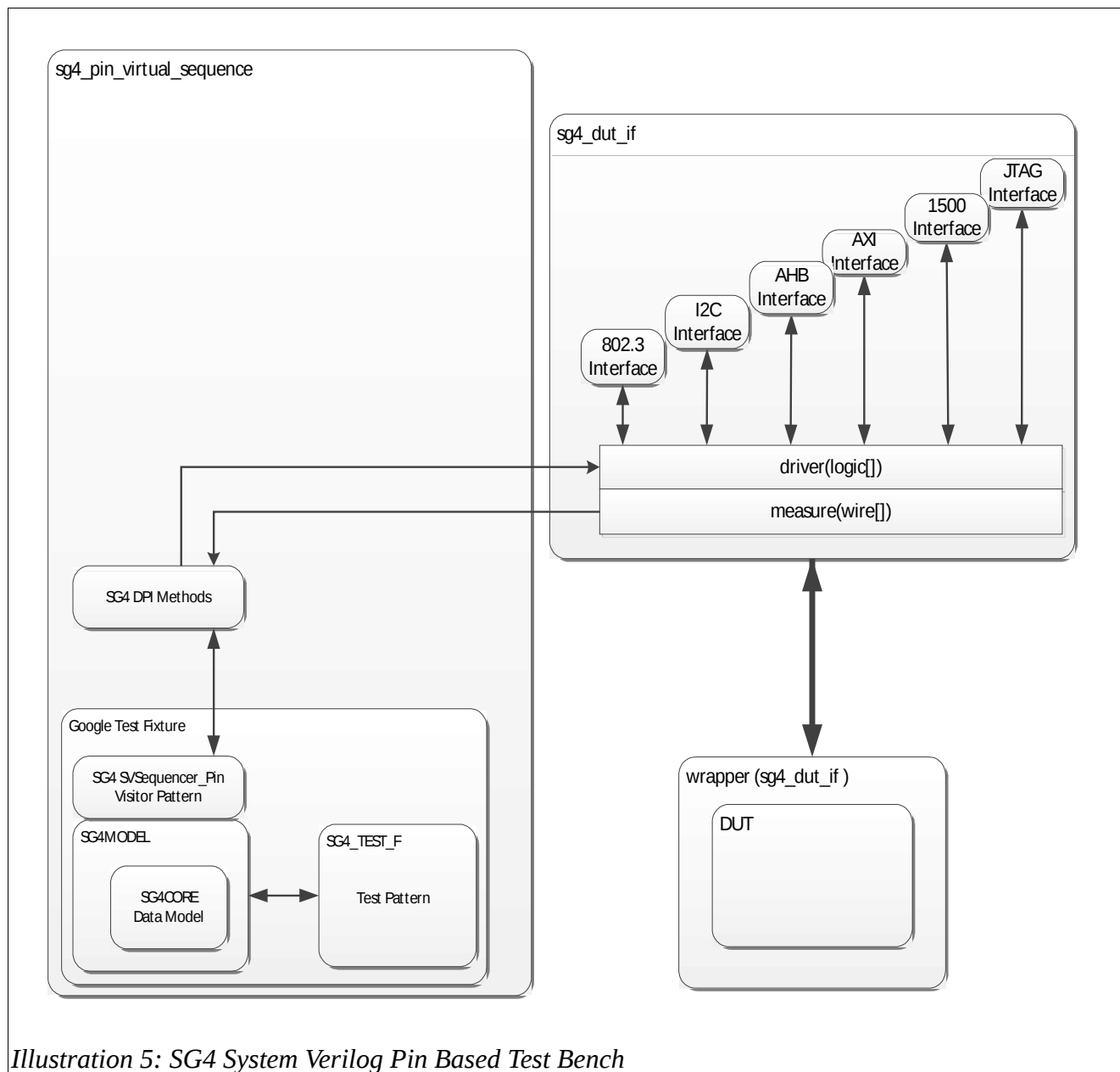
Illustration 5: SG4 System Verilog Pin Based Test Bench

12.6   *Working with the SG4 System Verilog Test Benches*

12.6.1   Building the SG4 System Verilog Test Benches

genSG4.pl will generate completely functional test benches for the pin and event based System Verilog Sequencers when command line option '-sv' is added. Event based test benches will be generated for both pure System Verilog and UVM based event test benches. All three test benches assume that the documented DUT interface is complete. Table 3 below describes the files generated by genSG4.pl for the DUT. These files build the verification environment based on SG4 System Verilog drivers and environment.

Table 4: Description of the System Verilog files generated by genSG4.pl for pin and event based benches

File	Description
sverilog/sg4/dut/interface.svh	Top level DUT interface used in both event and pin test benches
sverilog/sg4/dut/wrapper.sv	Top level DUT wrapper used to connect the sg4_dut_if to the top level Verilog module.
sverilog/tb/sg4_event_seq.sv	System Verilog(non UVM) sequencer to pass command packets to the interface methods
sverilog/tb/sg4_tb.sv	Top level System Verilog test bench that supports both event and pin sequencer and utilizes a `define to pick one or the other.
sverilog/uvm/sg4_top_env.sv	Top level System Verilog UVM environment that sets up the protocol agents.
sverilog/uvm/sg4_top_pkg.sv	Top level System Verilog package to pull in all of the UVM include files.
sverilog/uvm/sg4_top_seq.sv	Top level System Verilog UVM sequencer that implements the SG4 process hooks that connects the SG4CORE visitor pattern to System Verilog.
sverilog/uvm/sg4_top_tb.sv	Top level System Verilog UVM test bench for event sequencer.
sverilog/uvm/sg4_top_test.sv	Top level test that will invoke google test's gtest_main method and gather up command line arguments
sverilog/uvm/sg4_top_th.sv	Top level System Verilog Test harness
sverilog/sg4_dv_event_uvm.f	VCS -file command file argument for event UVM sequencer testbench.
sverilog/sg4_dv_event.f	VCS -file command file argument for event sequencer test bench.
sverilog/sg4_dv_pin.f	VCS -file command file argument for the pin sequencer test bench.

genSG4.pl generated test benches utilize the DUT interface and a Verilog wrapper module as seen in Illustrations SG4 System Verilog Event Based Test Bench and SG4 and System Verilog Process Event Handler Flow Chart above. The Verilog wrapper module takes as its argument the DUT interface that it connects to the instantiation of the DUT. The user should only need to do modify the DUT instantiation to align with the actual module interface if they are different (indicating incorrect documentation.)

The SG4 generated test benches are driven by a single Makefile. This Makefile is the same Makefile that services all stand-a-lone SG4 based environments including SG4 regression testing. There are a few variables that must be set for the Makefile. The macro SG4_DV is required to enable the System Verilog build. SG4_DV must be assigned one of the three possible values defined in the table below.

Table 5: Valid SG4_DV Assignments

SG4_DV Value	Description
EVENT_UVM	Build the SG4 UVM event base test bench. Adds option -file sverilog/sg4_event_uvm.f to vcs command line. Adds -DSG4_DV_EVENT to the g++ fixture compile command. Requires that environment variable UVM_HOME is defined and pointing to the UVM installation.
EVENT	Build the SG4 pure System Verilog event test bench. Adds option -file sverilog/sg4_event.f to vcs command line. Adds -DSG4_DV_EVENT to the g++ fixture compile command.
PIN	Build the SG4 pure System Verilog pin test bench. Adds option -file sverilog/sg4_pin.f to vcs command line. Adds -DSG4_DV_PIN to the g++ fixture compile command.

The macro SG4_DUT_F must also be set if you want to instantiate the DUT. If not specified the wrapper will not instantiate the DUT allowing you to compile and test the test bench if you wish. Use this to generate stimulus and see what happens. SG4_DUT_F must point to a vcs -F command options file that contains the vcs command line options to load the DUT. The -F option allows internal paths to be located relative to the location of the command options file. At a minimum it must identify the location of the verilog modules required.

SIGDATA now provides a test model called SG4_Car. To build this model issue the following commands

```
genSG4.pl -ip SG4_Car_A0 \
          -sv \
          -link
make test
make simv SG4_DV=PIN \
          SG4_DUT_F=/proj/sigdata/ip/SG4_Car/A0/rtl/SG4_Car.f
```

then run all tests in the t directory against it to ensure that SG4 is generating expected vectors for the SG4MODEL. The 'make simv' command will generate the vcs executable which we will use to simulate our patterns against.

You can skip the simv build and build the sim target that will be discussed next.

12.6.2   Running a Simulation

12.6.2.1   Using SG4 Makefile

The same Makefile described above can be used to run the simulation. It requires an additional macro to specify what google test or SG4_TEST_F portable pattern to run. When genSG4.pl is run all existing regression tests are installed locally and are potential candidates for simulation. A test called SG4CARA0TESTF.SANITY is provided. To run this test via the Makefile we must set macro GTEST_ARGS. This macro is the standard macro that specifies what test(s) should be run. To run the SANITY test use the following make command

>   make sim GTEST_ARGS="--gtest_filter=SG4CARA0TESTF.SANITY" \
>   SG4_DV=PIN \
>   SG4_DUT_F=/proj/sigdata/ip/SG4_Car/A0/rtl/SG4_Car.f

This will build the pin based test bench if it doesn't already exist. To build the event or UVM event environment simply set SG4_DV to EVENT or EVENT_UVM respectively. If you are switching between test bench types you will need to run 'make clean' and rebuild the executable. (BOZO can we build and support all at the same time?)

The Makefile will convert all gtest arguments that are prefixed with --gtest to Verilog plusargs. For the example above --gtest_filter=SG4CARA0TESTF.SANITY will become

>   +gtest_filter=SG4CARA0TESTF.SANITY

BOZO we need to add support on base sequencer. The SG4 ConfigManager will also accept command line options that are prefixed with --sg_. These will be converted to Verilog plusargs with pref +sg_. See Config Manager for details on usage.

It is always advised that you run your test pattern through the SG4 stand-a-lone regression test environment first to ensure that all string names are correct and SG4 does not throw any exceptions. If you skip this step you may spend hours simulating your pattern only to have SG4 throw an exception because a register, bitfield, pin or instance name contains a typo. To do this run command

>   make test GTEST_ARGS="--gtest_filter=SG4CARA0TESTF.SANITY"

When using the Makefile notice that the target has changed from 'sim' to 'test' and the GTEST_ARGS remain the same. You can keep the SG4_DV and SG4_DUT_F macros on the make test command line.

12.6.2.2   Using Command Line simv

You can bypass the Makefile all together and call the simv executable directly. Use the following command template to run the simulation

        simv +gtest_filter=<<FIXTURE>>.<<TESTNAME>>

The command to run our SG4_Car SANITY test is

        simv +gtest_filter=SG4CARA0TESTF.SANITY

You may encounter problems running simv directly depending on your environment. Shared object files (.so/.dll) are dynamically loaded and must be found in the environment based on environment variables and compile/link time options.

For more details on the google test command line arguments see https://code.google.com/p/googletest/wiki/Documentation. If you need to see the exact commands and arguments the make command is issuing add option '-n' to the make command and it will echo the commands with out running them.

12.6.2.3   Supported System Verilog Write Environment

Target	Directive Description
SV_SET_TICK_EDGE_TO_EDGE_DELAY	Integer value. Set the delay from one edge of the DUT's tick to the next edge. This should be set at the beginning of the pattern and remain constant from the point forward.
IEEE1149.1_SET_CYCLE_DEFINITION	Format: "name,type,vectors_per_cycle,tck_up_vector, tck_down_vector,measure_vector,drive_vector". Name is the name of the interface. Type is one of 'DEFAULT_CYCLE', 'TMS_TRANSITION' or 'SHIFT_CYCLE'. All other values are integers.

12.7   Integrating SG4 into Custom Environments

In cases where you will be providing your own test bench you are responsible for SG4 insertion based on your environment needs and capabilities.

You must create and instantiate a SG4 sequencer class that derives from sg4_base_virtual_sequence and implement the process_event_sg4* methods defined in table Event Processing Tasks and Functions. Once instantiated you must call the sg4_base_virtual_sequencer::body method from an initial block to connect SG4 and System Verilog. Use any of the three stand-a-lone SG4 generated test benches and the Makefile as reference.

12.7.1   Building the System Verilog Model

12.7.1.1   Common Build Components

SG4 makes no assumptions about the verification model and how it came to be, it assumes that the

documented DUT interface is correct and complete. SG4 makes no assumptions about how it will be integrated into your environment. SG4 builds a Synopsys vcs command line option file to pass to vcs using the '-file OPTIONS_FILE' command line option. The -file argument allows the options file to use environment variables and shell commands to build arguments. In this case it utilizes SG4 specific environment variables. This file contains all of the includes and preprocessor directives that are needed to use the event or pin based test benches. One option file will be generated for each type of test bench.

The option file only allows for the inclusion of verilog arguments. Any arguments to be passed to the C/C++ compiler must be passed via the raw command line. To link in SG4CORE and export the DPI methods the following g++ command line arguments must be added

> -file <<OPTIONS_FILE>> \
> -LDFLAGS "-rdynamic $(SG4LDFLAGS)" \
> -lSG4TEST

To deliver the SVSequencer visitor pattern in the SG4COREBASE the DPI methods must be exported by the linker into the main executable. The '-rdynamic' exports these DPI methods and allows SG4 to link them at runtime. (The -rdynamic option may pollute and bloat the exported symbols. You can create a custom linker script to minimize this list of exported symbols to only those that are needed. ==BOZO we need to do this in time and provide instruction.== Unless someone else does this first and provides instruction)

The SG4LDFLAGS should be set to

> -L$(SG4TESTLIB)/$(SG4LIBDIR)
> -L$(SG4COREBASE)/$(SG4LIBDIR)
> -L$(SG4MODELPATH/$(SG4LIBDIR)
> -Wl,-rpath,$(SG4TESTLIB)/$(SG4LIBDIR),-rpath,$(SG4MODELPATH)/$(SG4LIBDIR),-rpath,$(SG4COREBASE)/$(SG4LIBDIR)

Where $(SG4TESTLIB) is the base directory where portable test patterns are compiled.

Where $(SG4MODELPATH) is the base directory where the SG4MODEL has been compiled.

Where $(SG4LIBDIR) is

> lib/x86_64/Linux/gcc/<<GCC_VERSION>>/<<SG4VERSION>>

Here is an example

> lib/x86_64/Linux/gcc/4.8.1/55

12.7.1.2   *Pin Model Build Requirements*

Building the pin based environment requires that vcs option

> -DSG4_DV_PIN

be added to the command line.

12.7.1.3   *Event Model Build Requirements*

Building the event based environment requires that vcs option

-DSG4_DV_EVENT

be added to the command line.

12.7.1.4   UVM Event Model Build Requirements

Building the event based environment requires that vcs option

-DSG4_DV_EVENT -DSG4_UVM

be added to the command line.

12.7.2   Running a Simulation

Assume the top level vcs executable is called simv.

12.7.2.1   Pin Module Simulation Command line

simv +gtest_filter=<<FIXTURE>>.<<TESTNAME>>

12.7.2.2   Event UVM Module Simulation Command line

simv +UVM_TESTNAME=gtest_test +gtest_filter=<<FIXTURE>>.<<TESTNAME>>

All UVM test patterns require plusarg UVM_TESTNAME be present. SG4 provides the test gtest_test that passes the gtest plusargs to gtest_main.

13   Using SG4 for Static Pattern Development

13.1   SG4 Ownership

13.2   SG4 Environment

The SG4 pattern development environment has been built around the google test(gtest) framework and makefiles. The gtest framework enables us to:

16. Create one or more 'fixture's or SG4 setups that are leveraged across multiple tests

    1. Load SG4 data

    2. Open output files(STIL, ATPG)

    3. Set initialize state consistently

17. Easily add new tests by simply adding a new test file into the appropriate directory location

18. Build all, single or a subset of files (STIL) based on command line options

19. Integrate SG4 self checks if desired

20. Consistently and easily report problems

13.2.1   SIGDATA SG4 Test Patterns

13.2.1.1   PEO SG4 Test Patterns

```
peo/SG4/include
peo/SG4/src
peo/SG4/t
peo/SG4/mbist/*.cpp
peo/SG4/mbist/include
peo/SG4/mbist/src
peo/SG4/mbist/t
peo/SG4/ddr
peo/SG4/ro
peo/SG4/scan
peo/SG4/crest
peo/SG4/displayport
peo/SG4/fuses
peo/SG4/...
```

13.2.2 Anatomy of the SG4 gtest Environment

13.2.2.1 Test Pattern

The best place to start is with a sample test pattern. For the majority of users this is all you will need to know and understand.

```
10.    #include "kaveri_test_fixture.h"
11.    using namespace StimGen;
12.
13.    TEST_F ( KAVERIA0DUTTESTF, Mbist_Pwr10_BistCfgCpuUnb ) {
14.      dut.initPwr10();
15.      dut.test_logic_reset( 10 );
16.      dut.apply();
17.
18.      dut["IPCONFIG"]
19.        ["IPCPU0"](1)
20.        ["IPCPU1"](1).write().apply();
21.    }
```

Line 1:   Load the top level test fixture that provides basic functionality.

Line 2:   Tell the compiler to look for methods in StimGen namespace if objects or methods aren't found in the current name space. If not provided objects or methods would need to be prefixed with 'StimGen::'.

Line 4:   TEST_F is a gtest macro which registers this test into the framework. TEST_F takes to arguments. The first (KAVERIA0DUTTESTF) is the name of the test fixture to use. The test fixture setups the environment and will be discussed in more detail later. The second argument (Mbist_Pwr10_BistCfgCpuUnb) is the name of the test. This specific test fixture will automatically load the SG4 model, open the STIL file and initialize it appropriately. This won't always be the case so be sure you understand what the test fixture is doing for you. If it doesn't meet your needs you may need to look for another or create a new one.

Lines 5-12:   Test pattern content. This test fixture defines a pointer 'dut' which points to the top level SG4 model to generate stimulus against. See SG4 user guide for information on generating stimulus.

13.2.2.2 Test Fixture

The test fixture is the glue that connects your test pattern to the SG4 model and integrates it into the gtest framework. Its important to note that the SG4 model provides access to custom test patterns and or test sequences provided by others at the IP or SOC level. Its not necessary for everyone to understand the test fixture but it needs to be understood in order to setup a new fixture or project.

The following code excerpt defines the Kaveri test fixture. Implementation of the methods are details that are handled in another file.

```
1.     #ifndef KAVERI_TEST_FIXTURE_H_
2.     #define KAVERI_TEST_FIXTURE_H_
3.
4.     #include "gtest/gtest.h"
5.     #include "kaveri_peo.h"
```

```
 6.
 7.    using namespace StimGen;
 8.
 9.    class KAVERIA0DUTTESTF: public testing::Test {
10.      public:
11.        static kaveri_peo &dut;
12.
13.        kaveriA0Test ();
14.        virtual ~kaveriA0Test ();
15.
16.        std::string get_current_test_name ();
17.
18.        static void SetUpTestCase ();
19.        static void TearDownTestCase ();
20.
21.        virtual void SetUp ();
22.        virtual void TearDown ();
23.
24.      private:
25.        void open_credence_stil ();
26.
27.        Event_Visitor_STIL_Credence *stil;
28.    };
29.
30.    #endif /* KAVERI_TEST_FIXTURE_H_ */
```

Lines 1-2,31:   Setup C/C++ preprocessor directives to prevent loading definition multiple times

Line 4-5:   Load necessary include files to provide required base functionality. 'gtest/gtest.h' provides the gtest framework while kaveri_peo.h wraps the SG4 model with custom methods used only in the PEO SG4 environment. Methods defined here will more than likely only be available to PEO. If methods need to be available in other environments they should be written generically or integrated into the SG4MODEL.

Line 7:   Tell the compiler to look for methods in StimGen namespace if objects or methods aren't found in the current name space. If not provided objects or methods would need to be prefixed with StimGen::.

Line 9:   Define the gtest test fixture called KAVERIA0TESTF. The first argument of the TEST_F macro instructs gtest to use this test fixture for the test pattern. To enable use of this test fixture it is derived from gtest class testing::Test. This will then allow gtest to call various predefined methods allowing the test fixture to be setup or torn down at the appropriate times. See gtest documentation for more details.

Line 11,18-19: Defines a static reference to the top level device under test or dut. The dut is defined as a static variable so that we can load the SGXML and SG4MODEL once and reuse the model across multiple test patterns. The model will be loaded once when the static method SetUpTestCase is called (defined on line 18). The dut will remain valid until all tests have been run at which time TearDownTestCase will be called to clean up(defined on line 19). Dut is a static variable and has memory so care has to be taken between test patterns to ensure we restart at the same initial state.

Line 13:   Test fixture constructor which will perform any necessary initialization work

Line 14:   Test fixture destructor which will clean up as required.

Line 16:   Helper method 'get_current_test_name' which will get from the gtest framework the name of the current test being executed. This is used to setup generated file names.

Lines 21,22:   The SetUp and TearDown methods will be called at the start and end of each test pattern, unlike the SetUpTestCase and TearDownTestCase methods which are static and called only once at the start and finish respectively. The Setup method is responsible for resetting the duts state and opening all visitor pattern (STIL generation). The TearDown method will perform any tasks necessary after each test pattern is executed.

Lines 25,27:   Private method open_credence_stil creates and initializes the stil SG4 visitor pattern which generates the STIL file for the given test pattern. It'll set the generated file name to the gtest pattern name and append to it '.stil'.

13.2.2.3   SG4 Model Wrapper

The SG4 model wrapper creates an extra layer of functionality around the default and generic SG4 model. The wrapper in some cases is not needed. In other cases its absolutely required to handle interfacing with the integrating environment, while there are other cases where its simply a 'nice to have' to extend functionality that isn't applicable in all environments. In the case of the PEO environment, its used to extend functionality and allow us to support multiple users pointing to a single/common SG4 model.

The SG4 model wrapper should be owned and maintained by an identified person and under go review whenever changes are made. Additional changes must be made in a branch and fully vetted before being integrated into the main line which may be disruptive to other users.

BOZO: At this time the SG4 Model Wrapper is an advanced topic and is going to be temporarily skipped.

13.2.2.4   Accessing Command Line Options

Its often desirable to use command line options to adjust behavior of the pattern. The SG4 ConfigManager provides this functionality. For details see SG4 ConfigManager Command Line Options.

13.3   PEO SG4 Environment Setup

13.3.1   Requirements

PEO will need an environment that enables multiple users to develop patterns in parallel but utilize a common set of SG4MODEL (sgxml and SG4MODEL library)

13.3.2   Assumptions

- All users working on a project will be working in a single project directory structure.
- All users will be using the same version of SG4CORE

13.3.3   Environment Variables

To standardize the environment and make it easy to access key variables associated with SG4 several environment variables have been defined. These variables will be setup by an environment initialization script. The following table defines and describes the variables.

Variable	Description	Example
SG4PROJPATH	Base directory where all SG4 project data will reside	/tool/kv_debug
SG4VERSION	Version of SG4 to use	25
SG4COREBASE	Location of the SG4 core	/proj/verif_release_ro/ $SG4VERSION/StimGen4
SG4PROJECT	SIGAPI project	15h_KV_A0
SG4FAMILY	SIGAPI family component of SG4PROJECT	15h
SG4PRODUCT	SIGAPI product component of SG4PROJECT	KV
SG4REV	SIGAPI revision component of SG4PROJECT	A0
SG4MINORREV	SIGAPI minor revision component of SG4PROJECT	If SG4PROJECT=15h_KV_A0b then SG4MINORREV is 'b'
SG4PRODUCTNAME	Long name for the product	kaveri
SG4PROJECTTOP	${SG4PRODUCT_NAME}_${SG4REV}${SG4MINORREV}	kaveri_A0
SG4MODEL	Directory name with SG4PROJECT where the base SG4 model data will be located. Directory will contain SGXML and IP/SOC sequences that are compiled into libSG4MODEL.so that is loaded at runtime. It is recommended that the SG4MODEL be built and updated by a designated member of PEO and be stored in Read-Only directory named according to the template SG4PROJECT.SG4MODEL.SG4VERSION.YYMMDD Where: • SG4PROJECT is the variable previously defined • SG4VERSION is the SG4 version used to build the model • YY is the build year • MM is the build month using '0' for months 1-9 • DD is the build day using '0' for days 1-9 Such a naming convention communicates necessary information about the model and	15h_KV_A0.SG4MODEL.25.131220

AMD Confidential

Variable	Description	Example
	when it was created. Additional simple 'ls -lrt' directory list commands will nicely sort directories in a timely fashion which is easy to understand	
SG4WORKDIR	Each users unique work directory where the PEO SG4 perforce test program will be checked out for use, modification, etc.	$SG4PROJPATH/user/pjakobse/
P4CONFIG	Name of the file to location P4 information	P4CONFIG

13.3.4   SG4MODEL Setup

The SG4MODEL should be owned and released by an identified user. This user will be responsible for determining what version of SG4 should be used and building local SGXML and the libSG4MODEL.so for use in the various SG4WORKDIRs.

To build the SG4MODEL run the command

```
/proj/sigdata/bin/setup_SG4MODEL.init.csh \
    <SG4PROJECT> \
    <SG4PRODUCTNAME> \
    <SG4VERSION> \
    <OPTIONAL_QUALIFIER>
```

Notice that the SG4* arguments are directly related to the environment variables previously defined. These command line arguments will be used to setup all of the other environment variables.

13.3.4.1   Create the SG4MODEL

1. Create the directory

2. Create user initialization script for future calls

3. Clone the Hudson P4CLIENT and create a new client whose name matches $SG4MODEL

4. USER must save file

5. Check out data from Perforce

6. Run genSG4.pl

7. Run XMLLINT

8. Build and Run regression tests

9. Remove group and other write access

13.3.5   SG4WORKDIR Setup

Each pattern developer will have there own working directory and the environment variable

SG4WORKDIR should be used identify where it is. The SG4WORKDIR will be a perforce workspace where a p4 client will be created and test patterns checked out and modified as needed.

To create the local SG4WORKDIR run the following command providing the path to the desired SG4MODEL to build against

```
/proj/sigdata/bin/setup_SG4WORKDIR.csh $SG4PROJPATH/$SG4MODEL
```

This will create a local workspace in the location

```
$SG4PROJPATH/user/$USER
```

Or to build with a variant or qualifier name

```
/proj/sigdata/bin/setup_SG4WORKDIR.csh $SG4PROJPATH/$SG4MODEL  QUALIFIER
```

Which will create a local workspace in the location

```
$SG4PROJPATH/user/$USER/QUALIFIER
```

13.3.5.1   Create the SG4WORKDIR

1. Source the $SG4PROJPATH/$SG4MODEL/SG4MODEL.init.csh to get access to environment variables

2. Create directory $SG4PROJPATH/user/$USER/$QUALIFIER

3. Setup P4CONFIG file

4. Create new P4CLIENT with name $USER.$SG4PROJECT.$SG4MODEL

5. User must setup the client as needed

6. Checkout user contents from Perforce

13.3.6   SG4WORKDIR Initialization

The SG4WORKDIR only needs to be setup once but to start using it or to return to it some time later it needs to simply be initialized. To initialize your environment run the following commands

```
# Setup environment
source <PATH_TO_DESIRED_SG4WORKDIR>/SG4WORKDIR.init.csh

# If you aren't in your SG4WORKDIR simply run
cd $SG4WORKDIR
```

At this point your environment is in a state where you can pick up and run where you left off.

13.4   SG4WORKDIR Directory Structure

The SG4WORKDIR contains three items that must always be present.

- Makefile – Used to build the test programs

- include directory – Contains gtest test fixture class definition and any supporting class definitions such as SG4 wrappers

- src directory – Contains implementation of methods defined in the include class definitions

These three items are shared globally across all users and if changes are made they have to be done with much care especially for the contents of the include and src directories.

13.5   Eclipse

Eclipse is an Integrated Development Environment or IDE which assists in software development and in our case test pattern development. Here are a few of the advantages or features of Eclipse:

- Command completion

- Inline help information

- Inline lists of methods and attributes available for objects

- Hold 'Ctrl' and click on a method or attribute and Eclipse will take you to its definition

- Automatic correction of common typos

- Static analysis of files which identifies issues

- Code refactoring

- Code reformatting

- Integrated support for version control systems (Perforce)

While it takes a bit of time to setup Eclipse the benefits will save you significantly more time when providing information and catching what would have been compile time problems.

13.5.1   Eclipse Start Up

Our SG4 pattern development environment utilizes several environment variables. To use these variables within Eclipse we must either

- Start Eclipse in an initialized environment

- Add environment variables to Eclipse environment

We're going to focus only on starting Eclipse in an initialized SG4 environment.

To start Eclipse run command

```
/proj/dft_coe/bin/eclipse
```

This will start the application and a GUI will pop up. Accept the default workspace location when prompted.

13.5.2   Eclipse Setup

There are several things we must do to take advantage of Eclipses capabilities. These are one time setup steps per SG4WORKDIR.

13.5.2.1   Import SG4WORKDIR and Connect to Perforce
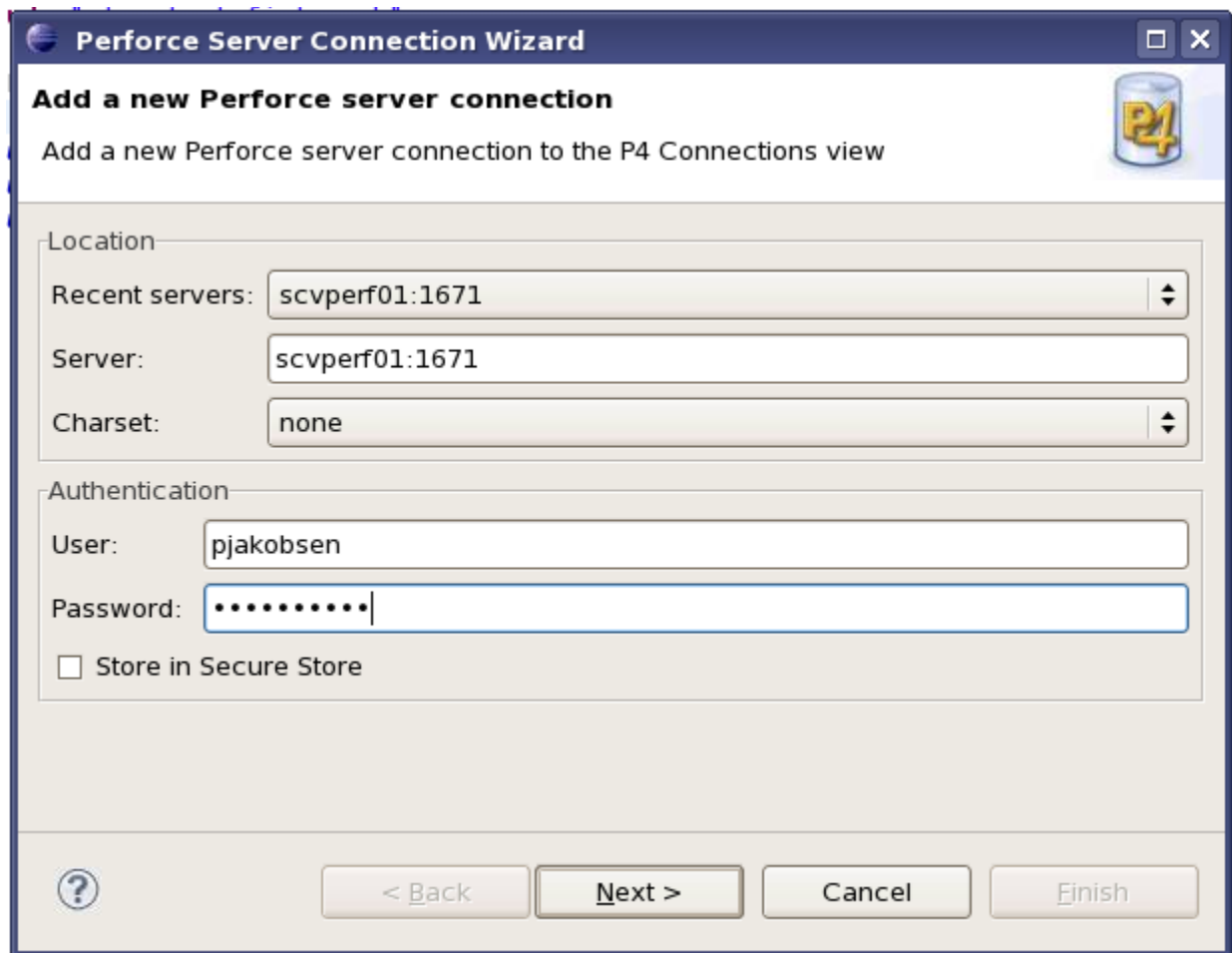
Our SG4WORKDIR is built entirely on a Perforce client or checkout. To get started with Eclipse we need to point Eclipse to our Perforce client by adding a P4 Connection. From the menu bar select

    Window → Open Perspective → Other → Perforce → Click 'OK'

In 'P4 Connections' tab in the upper left, click the '+' icon to add a new connection.
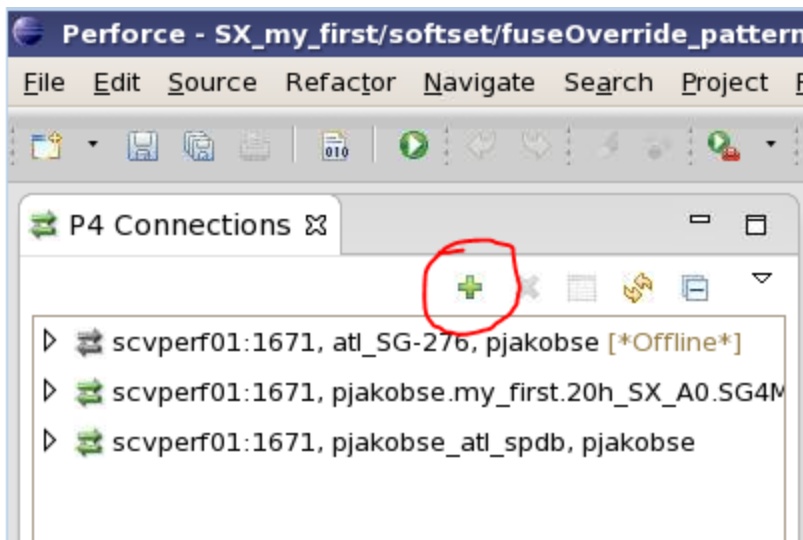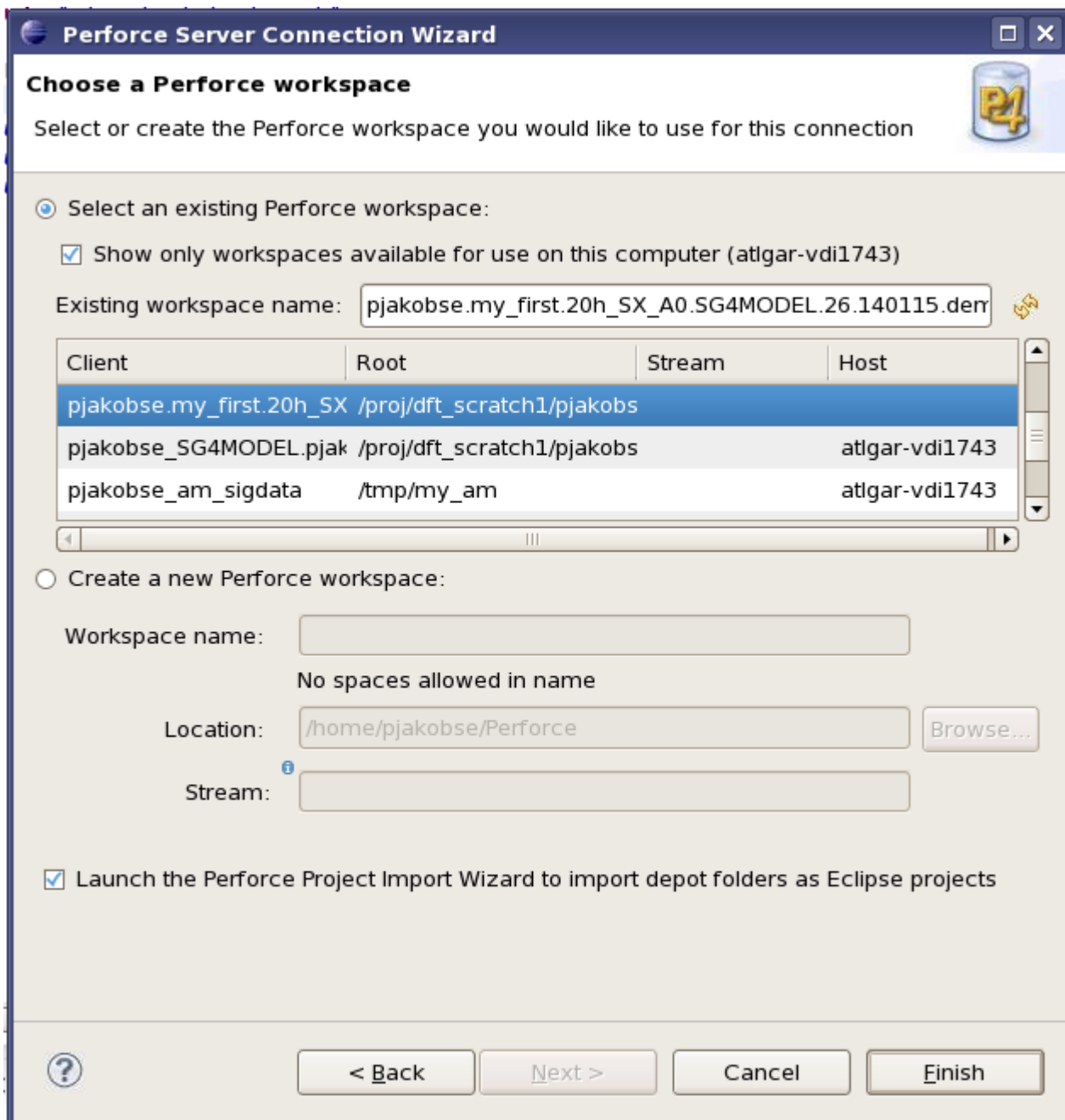
In the pop-up, fill in the appropriate Server and User/Password information. (Use your windows userid)

Click 'Next' and Eclipse will attempt to establish a connection with the server and get a list of all your existing clients. Either type the name of your client or scroll through the list until you locate the desired client to work with.

Click 'Finish'. Another pop-up window will appear to allow you to 'Choose remote folders to import as projects'. Click 'Cancel'

AMD Confidential
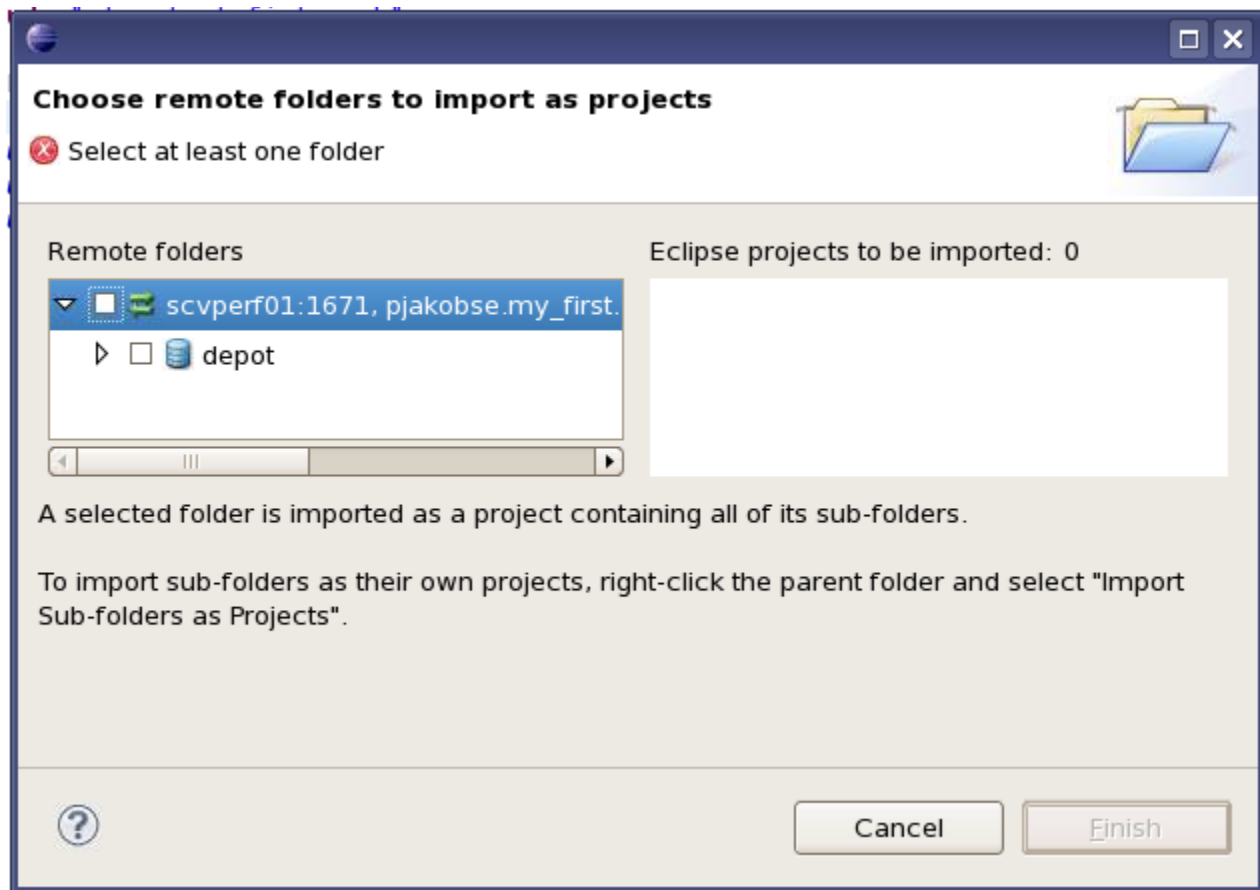
In your 'P4 Connections' tab, you will now see and entry for the client you just added. In my case 'pjakobse.my_first.20h_SX_A0.SG4MODEL.26.140115.demo'
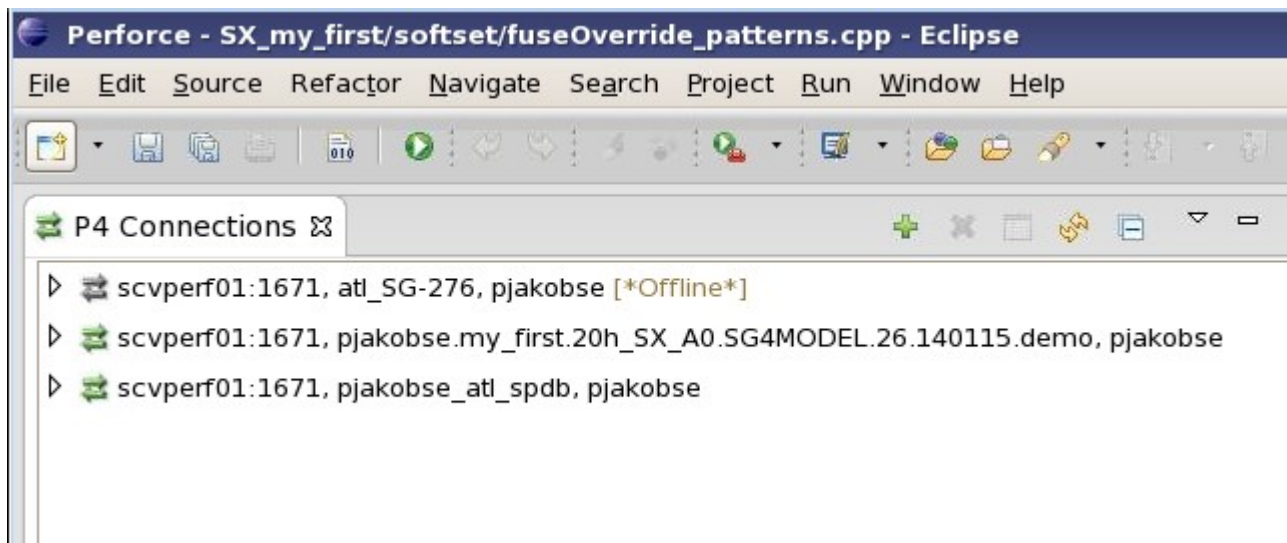
Expand it and navigate to the directory that you checked out. In my case //depot/sigdata/20h/SX/peo/SG4/main and right click on the folder and select 'Import as Project with name'
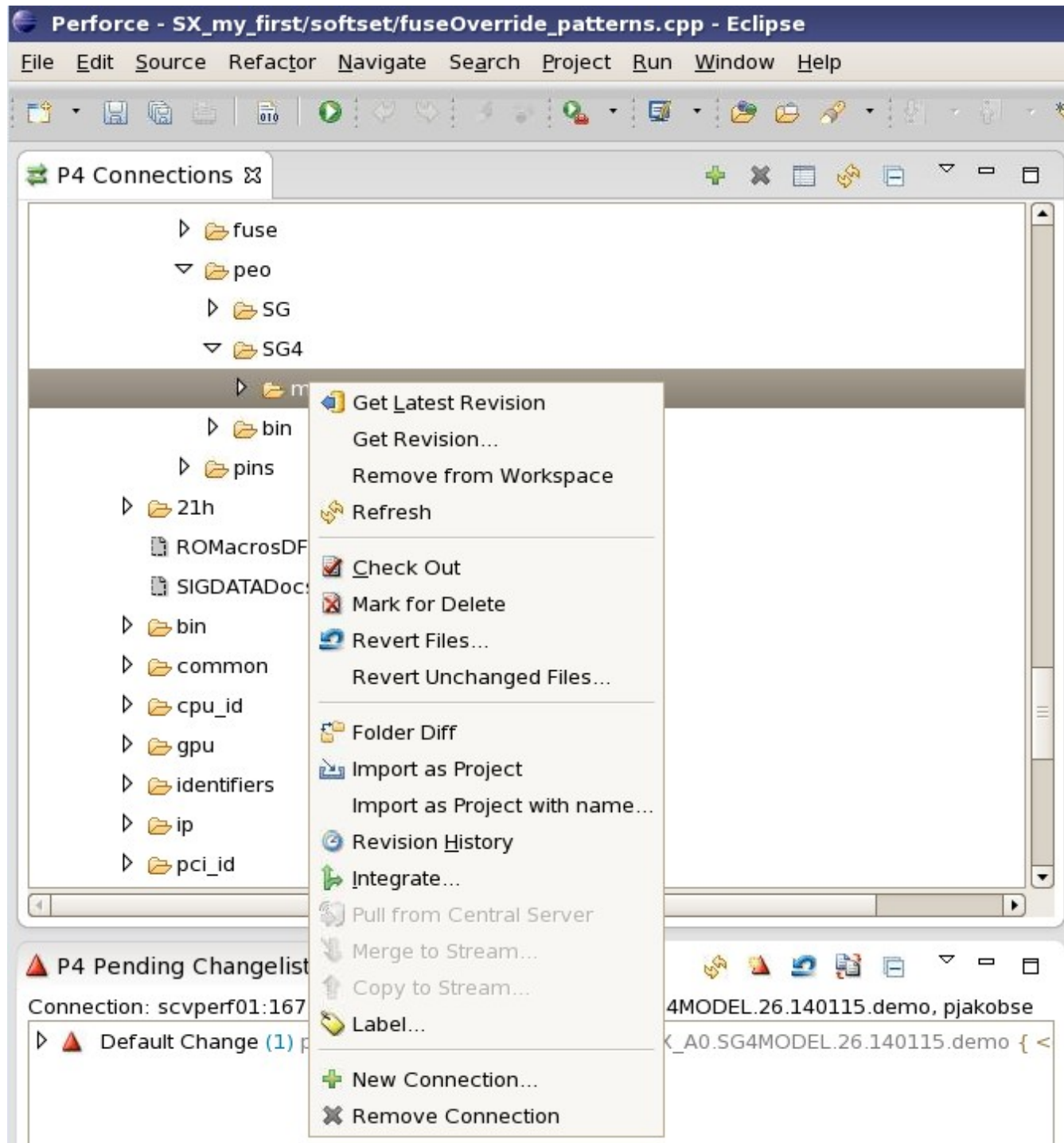
Select a project name(SX_peo for me) and click 'OK'.
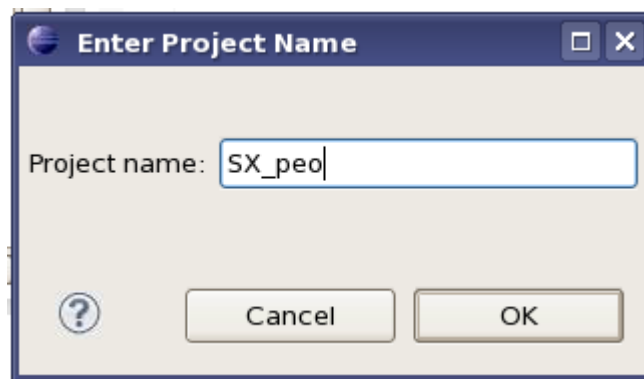
This will import into your Eclipse workspace a new project called 'SX_peo' which will be connected to perforce. If you edit an existing file Eclipse will automatically check the file out for you and when you add files you can simply click on the file and select

'Team → Mark for Add'

or the appropriate operation. There are many commands available to you.

13.5.2.2   Convert Generic Eclipse Project to a C/C++ Project

From the menu bar select

'Window → Open Perspective → Other → C/C++'

and click 'OK'
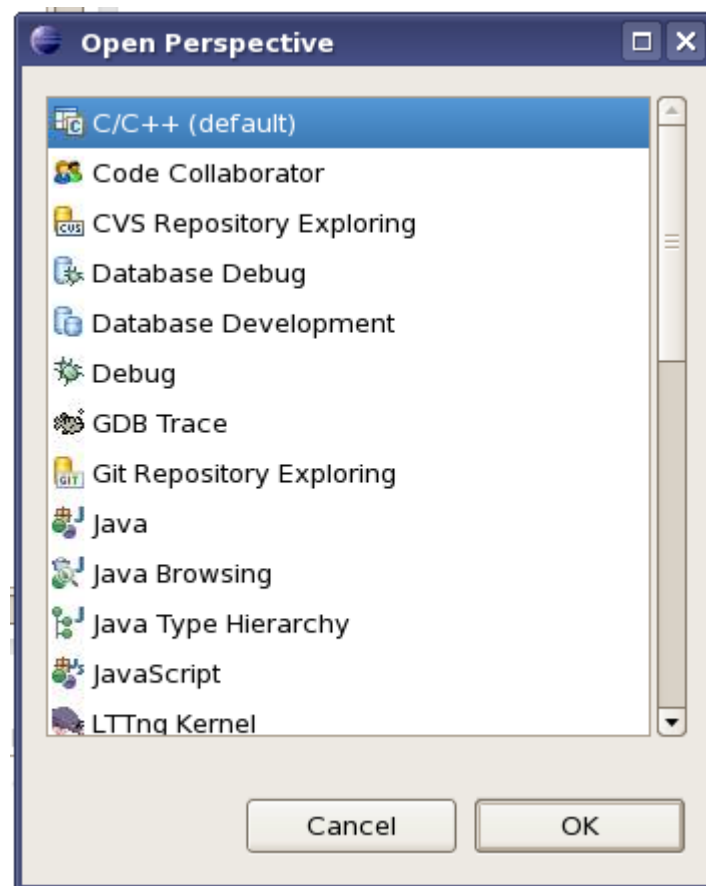
In the 'Project Explorer' tab you will now see a new folder with your project name.
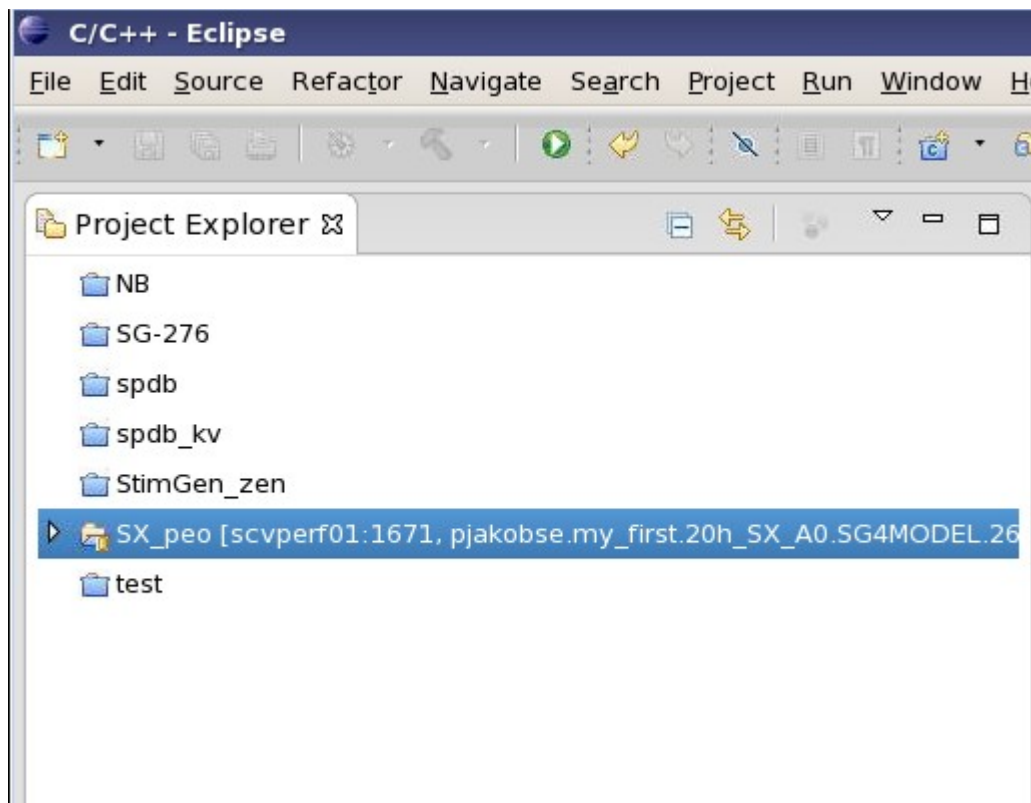
We now need to tell Eclipse that this is a C++ project so that it will do lots of extra 'stuff' for us. Select from the menu bar

'File → New → Convert to a C/C++ Project (Adds C/C++ Nature)'
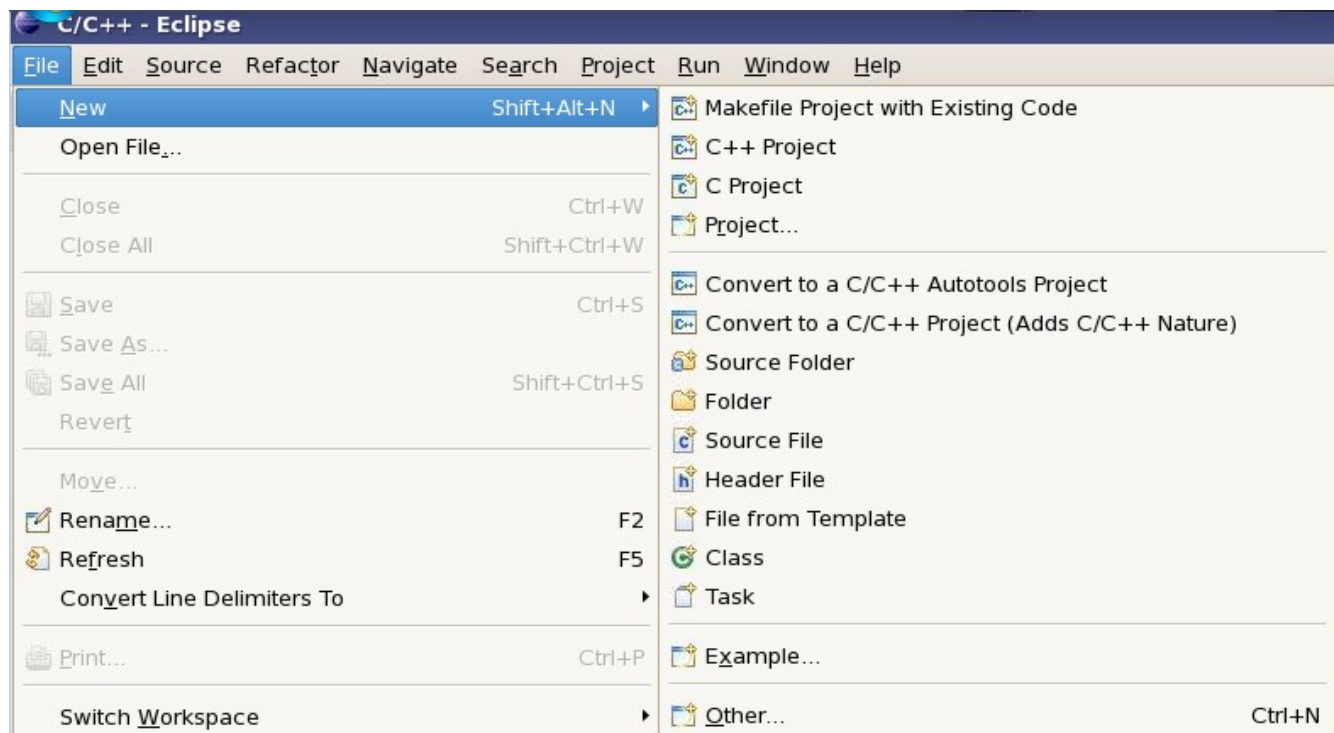
This will pop up the window shown below. Select 'Project Type' of 'Makefile project' and 'Toolchains' of 'Linux GCC' and click 'Finish'
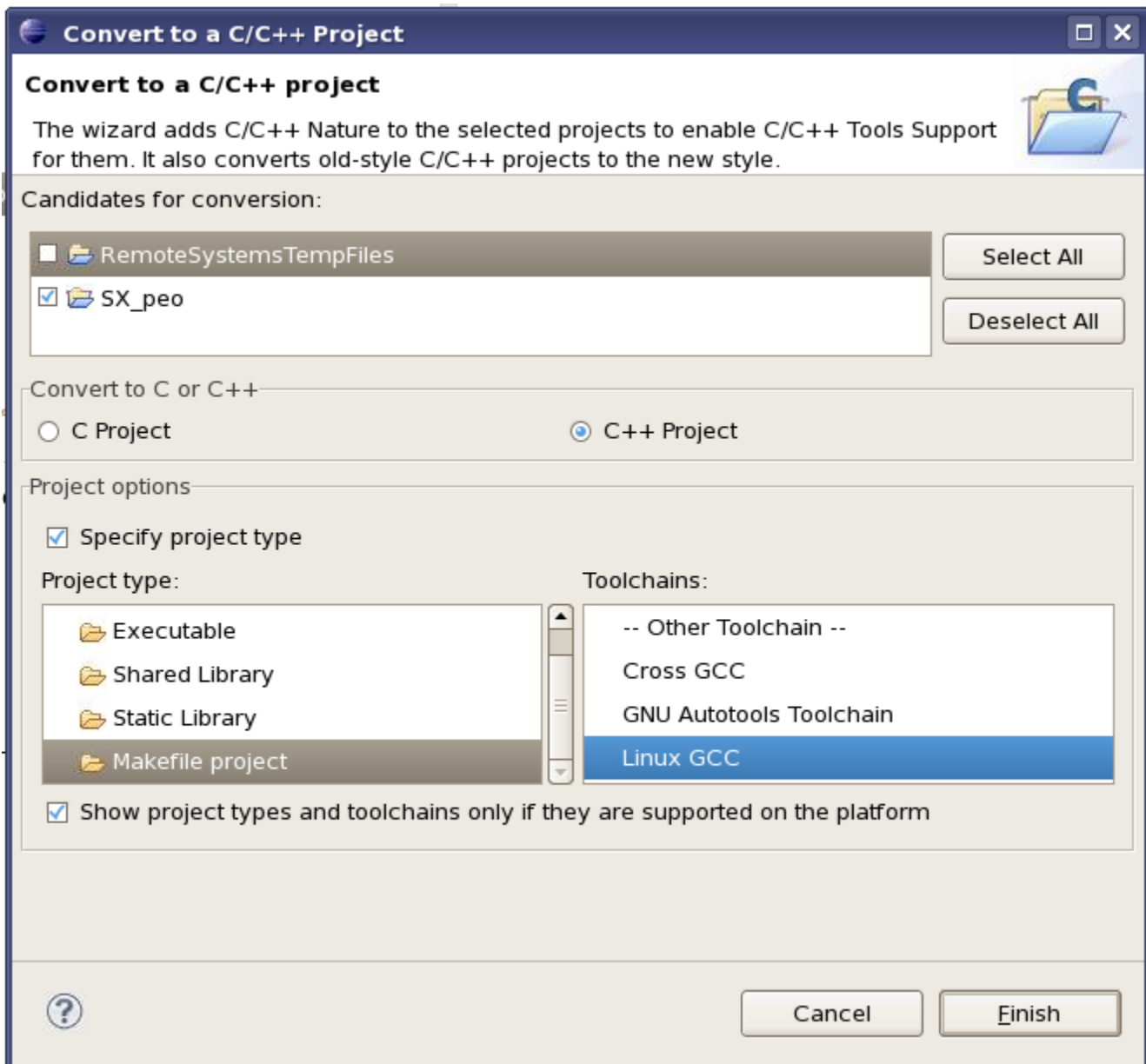
13.5.2.3　　　　Setup Eclipse Include Paths

Now we need to provide information to Eclipse to tell it where to find code. Right click on the new project and select 'Properties' (or select the project and hit 'Alt - Enter'). Navigate to

> 'C/C++ General → Paths and Symbols → Includes tab

Select Languages 'GNU C++' and then click the 'Add' button. Add all of the includes listed below

> ${env_var:SG4COREBASE}/include

${env_var:SG4COREBASE}/../StimGen4_test/thirdparty/gtest-1.6.0/fused-src

${env_var:SG4PROJPATH}/${env_var:SG4MODEL}/include

When you are finished click OK. (NOTE: For these environment variables to work, Eclipse must have been started in an SG4WORKDIR environment that defines these variables.)
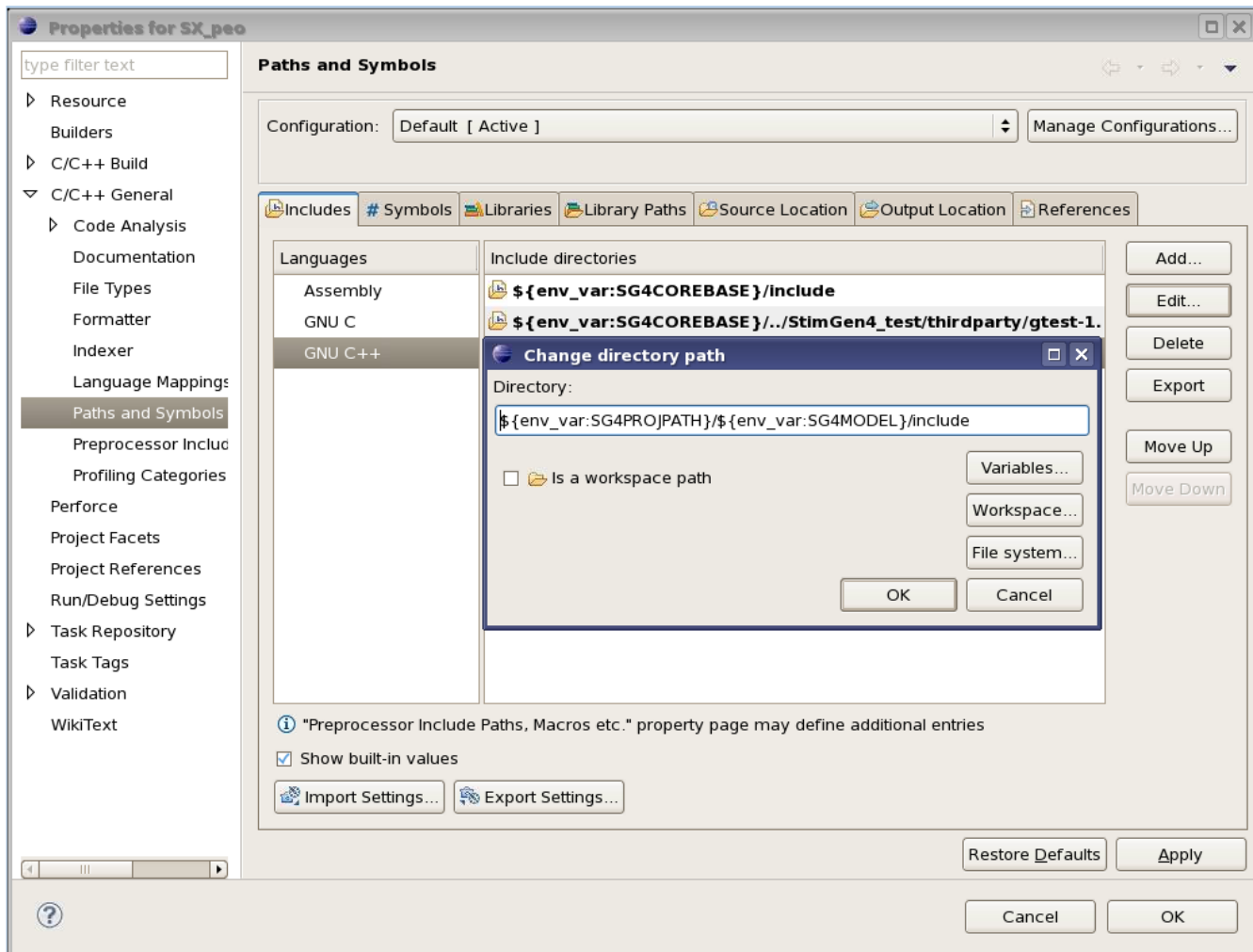
When you click 'OK', Eclipse will index all of the newly discovered code. Open up a test pattern and you should see code without any yellow or red lines under the text indicating that Eclipse static analysis of the code did not find any problems.

13.5.3   Compile in Eclipse

To compile your programs we need to run the make command. Right click on the 'Makefile' in your project and select 'Make Targets' and then 'Create'. We need to create the first target
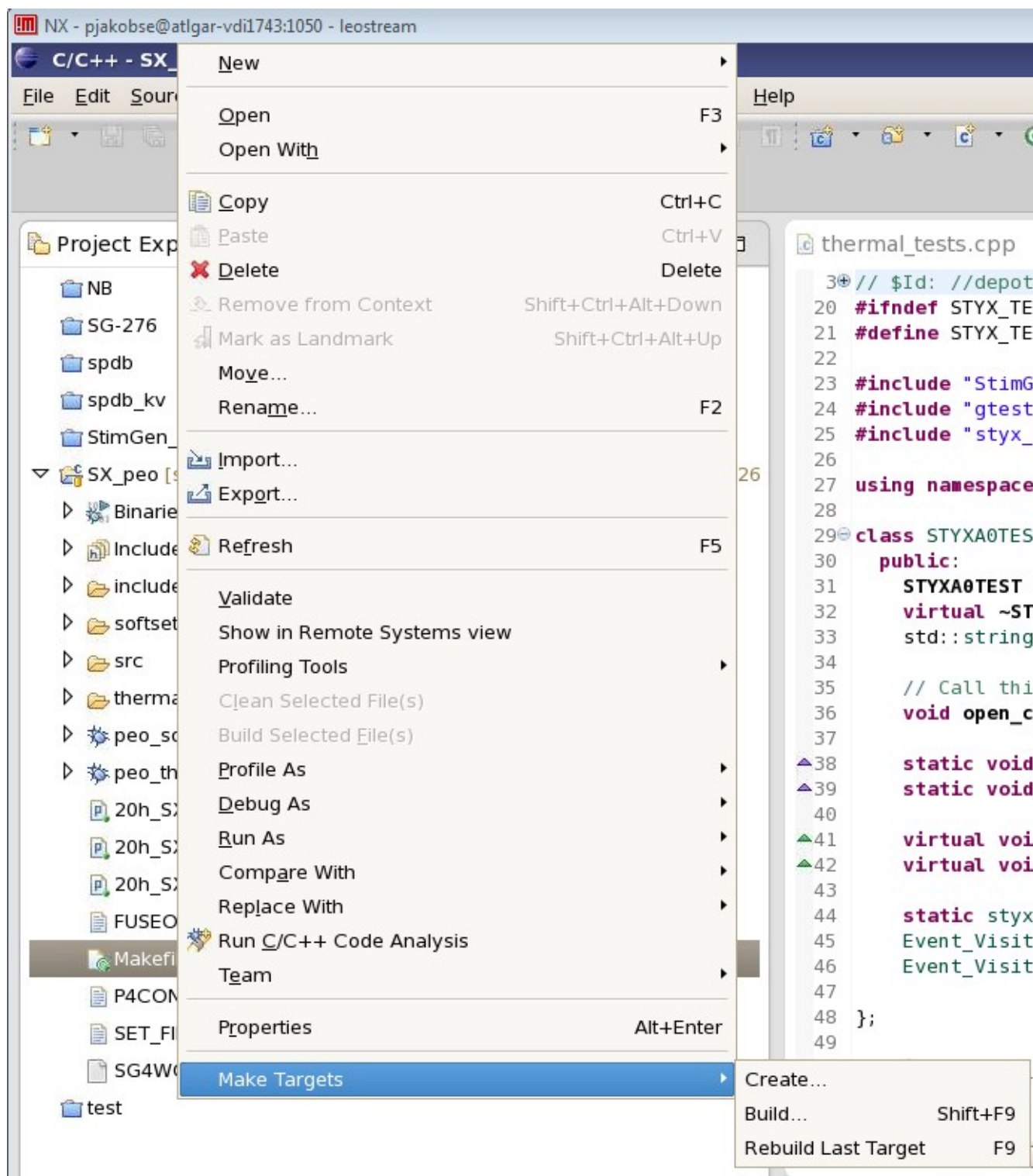
In the pop up the appears set the 'Target name:' to a valid target defined in your Makefile and hit 'OK'.
In my case I have a target called 'peo_thermal'
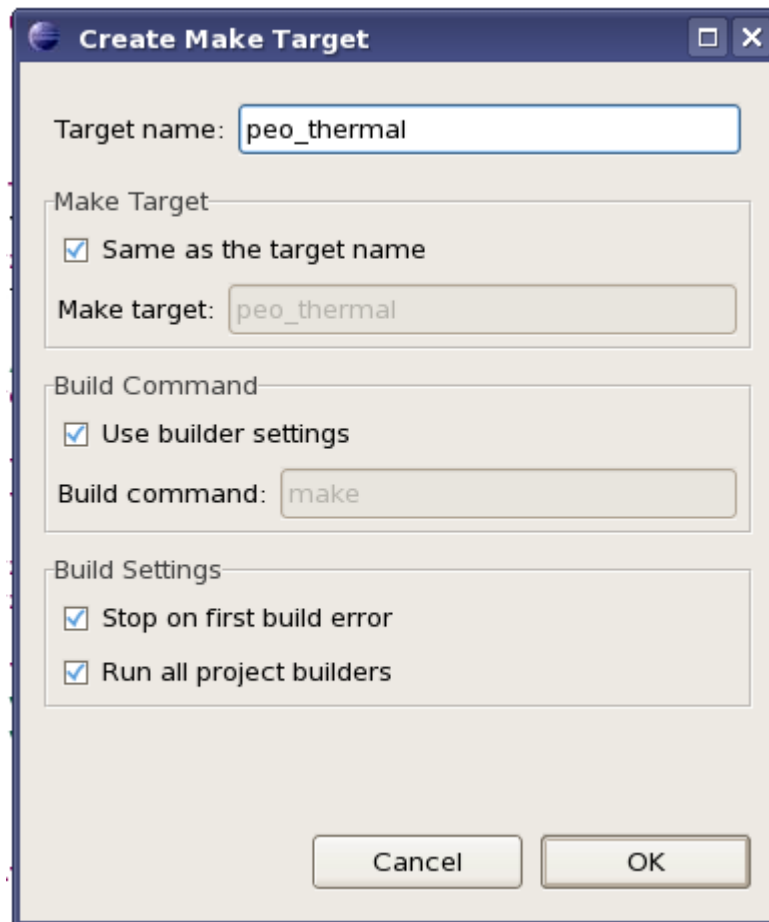
Now that we've informed Eclipse about the target we can build it. Right click on the Makefile again, select 'Make Targets' and select 'Build'. You'll now see your target listed here. You can now add more targets or select the desired target to build.
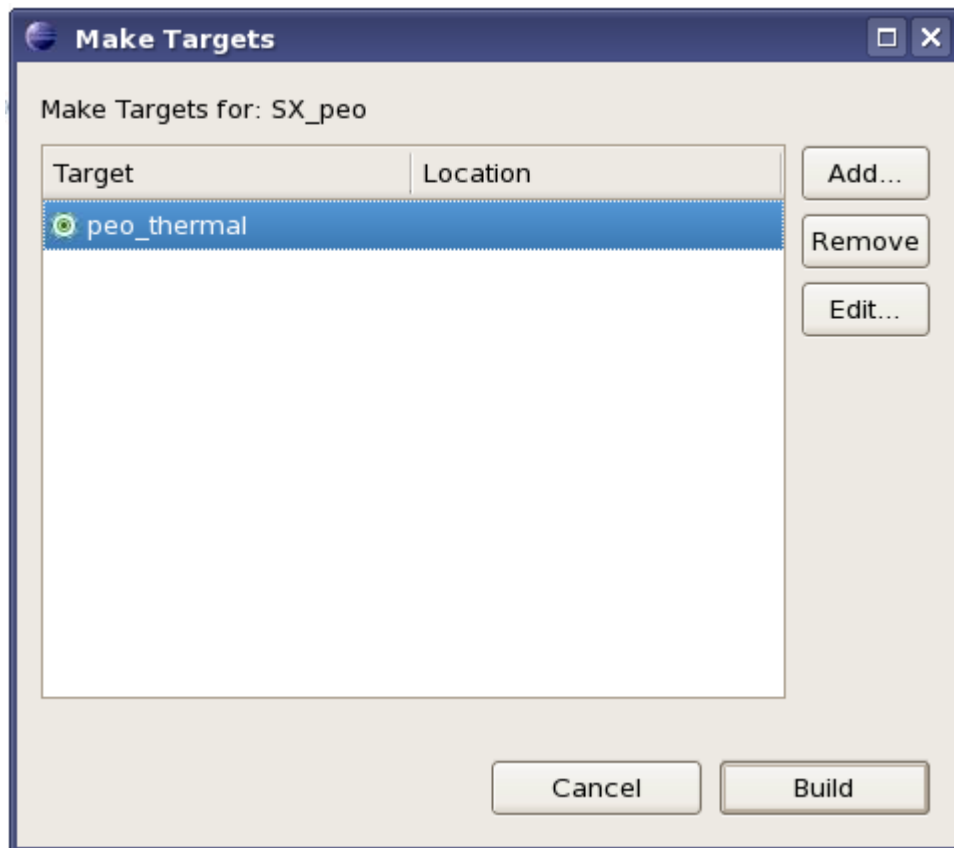
When you click 'Build', Eclipse will compile the code and send all output into the console tab.
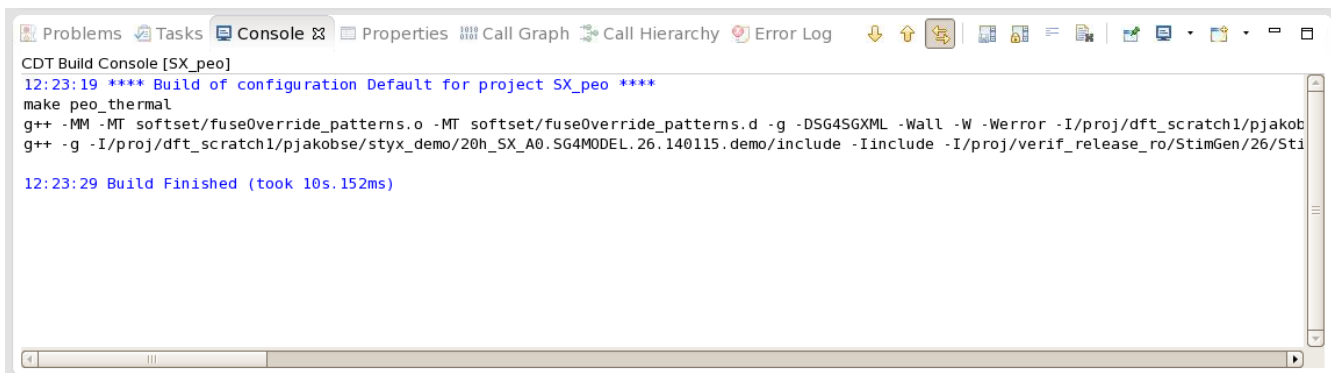
It there no problems it'll look similar to what's shown in the figure above. If there are problems, Eclipse will recognize them, make them red, update the code in its editor to identify the problems with red underlines and allow you to click on the individual error messages in the console and take you directly to that line of code (just as you can with links in a web browser). The figure below contains a syntax error to show what was just described. (The yellow lines in the console are actually red but didn't import nicely)

```
 40
▲41     virtual void SetUp ();
 42     virtual void TearDown ();
 43
⊗44     static styx_peo *dut;
 45     Event_Visitor_STIL_Credence *stil;
 46     Event_Visitor_Debug debug_visitor;
 47
 48 };
 49
 50 #endif /* STYX_TEST_FIXTURE_H_ */
 51
```

```
🔲 Problems 🔲 Tasks 🔲 Console ⊠ 🔲 Properties 🔲 Call Graph 🔲 Call Hierarchy 🔲 Error Log    ⬇ ⬆ 🔲  | 🔲 🔲 🔲 🔲 | 🔲 🔲 ▾ 🔲 ▾ 🔲 🔲
CDT Build Console [SX_peo]
12:27:04 **** Build of configuration Default for project SX_peo ****
make peo_thermal
g++ -MM -MT src/styx_test_fixture.o -MT src/styx_test_fixture.d -g -DSG4SGXML -Wall -W -Werror -I/proj/dft_scratch1/pjakobse/styx_demo/20h
g++ -MM -MT softset/fuseOverride_patterns.o -MT softset/fuseOverride_patterns.d -g -DSG4SGXML -Wall -W -Werror -I/proj/dft_scratch1/pjakob
g++ -g -DSG4SGXML -Wall -W -Werror -I/proj/dft_scratch1/pjakobse/styx_demo/20h_SX_A0.SG4MODEL.26.140115.demo/include -Iinclude -I/proj/ver
include/styx_test_fixture.h:44: error: expected ';' before 'static'
src/styx_test_fixture.cpp: In static member function 'static void STYXA0TEST::SetUpTestCase()':
src/styx_test_fixture.cpp:59: error: 'dut' was not declared in this scope
src/styx_test_fixture.cpp: In static member function 'static void STYXA0TEST::TearDownTestCase()':
src/styx_test_fixture.cpp:63: error: 'dut' was not declared in this scope
src/styx_test_fixture.cpp: In member function 'virtual void STYXA0TEST::SetUp()':
src/styx_test_fixture.cpp:67: error: 'dut' was not declared in this scope
src/styx_test_fixture.cpp: At global scope:
src/styx_test_fixture.cpp:87: error: no 'void STYXA0TEST::TearDown()' member function declared in class 'STYXA0TEST'
```

13.5.4   Run Executable in Eclipse

To run the executable the first time, locate it in your project space and right click on it and select

'Run As → Local C/C++ Application'

Eclipse should then launch your application and send STDOUT to the console.

14   PEO Integration

PEO's primary use of StimGen is for generation of STIL for use on ATE. Currently StimGen does not have perl binding adequate to make it compatible with StimGen version 1, 2 and 3. These are planned but automation and additional infrastructure needs to be put into place.

14.1   Event_Visitor_STIL_Credence

To use the STIL visitor pattern add a configured instance of Event_Visitor_STIL_Credence to the DUT.

```
StimGen::ModuleFactory factory ( "sgxml", "." );
factory.import( "SOC.xml", "", &dut);
StimGen::Event_Visitor_STIL_Credence vp_stil(
        &dut,
        "STIL_FILE_NAME.stil",
        "Pattern_Name"
);
vp_stil.set_signal_group_include( "stil_SIGGRP.inc" );
vp_stil.set_signal_include( "stil_SIG.inc" );
vp_stil.set_timing_mode ( "normal" );
vp_stil.set_wft_include ( "stil_wft.inc" );
vp_stil.set_keyword ( "ABC" );
vp_stil.add_include ( "Extra_include_file1.inc" );
vp_stil.set_credence_padding(true);
vp_stil.initialize();
dut.add_event_visitor(vp_stil);
```

There are other options available to configure the pattern output which are listed in the documentation section Pattern Writing. These are accessed through the write_env("STIL", …) commands and may be tester specific. For example, the option "CREDENCE_PAD_VECTORS" will add vectors to a this visitor, but not to the 93K visitor described below. However, in the example above, we used an alternate method set_credence_padding() to do the same thing through a method which only exists in this Visitor.

14.2   Event_Visitor_STIL_Advantest_93K

Very similar to the code above except the visitor type is Event_Visitor_STIL_Advantest_93K. As with the Credence visitor above, there are additional environmental options that can set test specific output. For example, "93K_XMODE ##", where ## is the modulo number to use for vector generation.

14.3

14.4   *Splitting STIL Files*

The following applies to both of the visitors listed above. The method for splitting a pattern file is accomplished by adding write_env("SPLIT", "filename") after an apply() at the place where the split should occur.

```
TEST_F( simpleDevice, split_93K_test ) {
    Event_Visitor_STIL_Advantest_93K stil ( &dut, "file1.stil", "patternName" );
    dut.add_event_visitor( stil );
    stil.initialize();
    dut.write_env ( "STIL", "93K_XMODE 8" );
    dut.write_pin ( "A", PV_0 );
    ...
    dut.apply();

    dut.write_env ("SPLIT", "file2" );
    dut.write_pin ( "A", PV_1 );
    ...
    dut.apply();
    stil.close_file();
}
```

A couple of things to note here. When we executed the "SPLIT", we automatically closed the first file and opened the second file. Notice that the second filename, "file2" does not require the ".stil" extension added to it. When we exited the test, we called close_file(), which closed the 2nd file. If we didn't have the close_file() it would still be taken care of by the destructor for the Event_Visitor, but it's a little cleaner to be explicit about it.

15   ATPG

StimGen is currently depending on Qinit to generate Mentor and Synopsys stimulus until that functionality has been integrated into StimGen. (Note the perl version of stimGen does support generation of Mentor and Synopsys initialization patterns)

15.1   Background

Historically, ATPG initialization sequences have used Qinit as the language of choice. It is a custom Domain Specific Language or DSL with a very well defined and focused use model and target customers. StimGen test patterns (ATPG initialization patterns) are purposely designed to be integrated into a variety of environments, used as is and cover a large number of users. This is very ambitious and difficult to do and requires that test patterns use the StimGen API only (the least common denominator). The StimGen API today is C++ and as a result test patterns are C++. There are plans to provide bindings to dynamic languages and to support IEEEP1687 PDL but that is in time. Qinit can be used to generate other formats to feed into other environments but this requires compilation of that pattern to other target languages. StimGen's goal is to eliminate much of the need for conversion to other formats for different environments and use patterns as is, natively.

15.2   Strategy

StimGen today can be run integrated into various environments or standalone. Regression tests are one usage for standalone functionality. Regression tests are built around gtest and there is a simple and nice interface for both writing patterns and running patterns. See the chapter on creating regression tests (BOZO does this exist yet).

The ATPG tools require additional and extra customizations to generate vectors and control the model. As an example cut points are inserted which are treated as primary input pins. To enable this customization we'll take advantage of gtest's test fixture. The test fixture sets up the environment for all tests. This should work nicely for ATPG where we need to have a consistent and controlled environment for pattern generation. Using the gtest infrastructure will allow us to customize the stimGen model and create test patterns and do so in an environment built on makefiles so dependencies can be established to ensure patterns get regenerated as source information changes.

The benefit of this setup is ease of use for the end user. We could alternatively setup an environment like verification's where users define a test method that gets dynamically linked to the model and executed. There's much more work necessary to build and maintain this environment.

15.3   Getting Started

To get started with stimGen for ATPG we are going to use the same flow described for verification and for running stimGen stand-a-lone. Here are the steps

1.   Run genSG4 with the -gen_atpg command line option

       genSG4.pl -proj <PROJECT>   -gen_atpg

    The -gen_atpg option will instruct genSG4 to pull copies of files from SG4/atpg into your build

area in addition to creating the SGXML and pulling SG4/include, SG4/src and SG4/t. Its important to note that these are copies of the files and any updates made will need to get inserted into your local perforce checkout. The SG4/include and SG4/src directories contain any custom sequences defined for the IP and SOC used in the current project. The SG4/t directory contains auto-generated regression tests and any regression tests provided by IP and or SOC teams for their respective piece of the pie.

If there are existing files in SG4/atpg you can continue, otherwise you need to create the gtest test fixture that will be used for pattern generation.

2. Optional: Validate the sgxml is valid by running command (BOZO add a target in the makefile)

   xmllint --schema $SG4COREBASE/../sigtools/schema/SGXML.xsd \

       --noout \

       --xinclude \

       sgxml/<TOP_LEVEL>sgxml

3. Optional: Run regression tests to ensure we can build the model and ensure that all sequences that should work do work. Run the command

   make test

   If the regression tests fail you should fix if its a local problem, contact the owner of the failing test or contact dl.sg4.developers for help.

4. If you have content in the local ./atpg directory run command

   make test TEST_DIR=atpg

This will compile the atpg patterns in the atpg directory and run them. This will build the test.out executable and run all patterns upon successful compilation. Each pattern should produce its own atpg (QINIT, Mentor or Synopsys) source pattern. If you want to generate only a specific pattern use gtest option –gtest_filter="____" to identify which pattern to build.

15.4 ATPG gtest Test Fixture

The Test Fixture is responsible for loading the design, adding extra pins, cut points, etc. Ideally it should be consistent for all project patterns. If other setups are needed simply create unique class with a different name. For general information about the gtest test fixture see gtest documentation.

15.4.1 Sample test fixture

```
31.      #ifndef _15h_CZATPG_BASE_H_
32.      #define _15h_CZATPG_BASE_H_
33.
34.      #include <iostream>
35.      #include "StimGen.h"
36.      #include "gtest/gtest.h"
37.
```

```
38.      using namespace StimGen;
39.
40.      class CZATPG: public testing::Test {
41.        public:
42.           Module *dut;
43.           Pin* refclk;
44.           Event_Visitor_Debug   debug_visitor;
45.           Event_Visitor_Checker checker;
46.           bool SCM;
47.
48.           CZATPG () : checker(NULL) {
49.             SCM = true;
50.             StimGen::ModuleFactory factory ( "sgxml", "." );
51.             dut = factory.import( "carrizo_A0.sgxml", "");
52.
53.             refclk = dut->add_pin( "REFCLK", StimGen::Pin::Input );
54.
55.             checker.set_dut( dut );
56.           }
57.
58.           virtual ~CZATPG () {
59.              delete dut;
60.              dut = NULL;
61.           }
62.
63.           void refclks ( int count = 1 ) {
64.      #define USE_QINIT_LOOP 1
65.      #ifdef USE_QINIT_LOOP
66.             std::stringstream ss;
67.             ss << "loop " << count << " {\n"
68.                << "  cycle { REFCLK = pulse 1; }\n"
69.                << "}\n";
70.             dut->write_env ( "Qinit", ss.str() );
71.      #else
72.             for ( int i = 0; i < count; i++ ) {
73.               *refclk = 1;
74.               dut->next_cycle();
75.               *refclk = 0;
76.               dut->next_cycle();
77.             }
78.             dut->apply();
79.      #endif
80.           }
81.
82.        protected:
83.           virtual void SetUp () {
84.              // Set global configuration options
85.              ConfigManager &cfgmgr = ConfigManager::Instance();
86.              cfgmgr.use_verilog_four_state_compare = true;
87.              cfgmgr.check_broadcast_load_value_consistency = true;
88.              cfgmgr.enable_read_auto_broadcast_to_serial = false;
89.              cfgmgr.enable_reset_network_control_on_apply = true;
90.              cfgmgr.report_queue_events = false;
91.              cfgmgr.report_reset_registers = false;
```

```
92.
93.          debug_visitor.set_report_fields( true );
94.
95.        // Add the checker visitor pattern so that we can embedded into
96.        // our patterns checks that ensure the stimgen generated ir/dr
97.        // operations are indeed correct.  Think of this as self checking
98.        dut->add_event_visitor( checker );
99.
100.       // I always find it useful to generate debug informatino to stdout
101.       // I've gotten good and deciphering 0's and 1's in this format
102.       dut->add_event_visitor( debug_visitor );
103.     }
104.
105.   virtual void TearDown () {
106.     unsigned int mismatches = checker.get_mismatches();
107.     EXPECT_EQ( mismatches, 0);
108.     checker.reset();
109.   }
110.
111.  };
112.
113.  #endif
```

Lines 1,2,84:   Standard CPP header protection so we don't include the file multiple times

Lines 4-6:   Include the necessary header files

Line 8:   Instruction compiler to import the StimGen namespace so that we don't need to prefix StimGen objects with 'StimGen::'

Line 10:   Define the test fixture 'CZATPG'. This must be a unique name otherwise you'll get compilation errors. The class must derive from testing::Test.

Line 12-16:   Declare public variables to be used in the test patterns.

Line 12:   The dut is a pointer to the top level model built by stimGen. Here it is defined as the generic StimGen::Module (Note that 'StimGen::' is not required because of line 8. If there are custom sequences for the IP or SOC this could be of that type StimGen::<TOP_LEVEL> (for Carrizo it could be 'carrizo' if there are no rev specific versions or 'carrizo_A0' if there is an A0 specific revision available)

Lines 14,15:   Declare two visitor patterns for use in tests patterns. Note that we did not include the ATPG specific visitor patterns as they will be added per output file to create files unique output files for each.

Line 18:   Test fixture constructor.

Line 20-21:   Use the ModuleFactory to load the SGXML model for the project. The first argument to the factory is the directory where the sgxml is located. Recall that genSG4 creates a local sgxml directory with all files in it. The import call's first argument is the file that contains the top level definition. In this case 'carrizo_A0.sgxml'

Line 22:   The pins.xls file for this project does not contain a pin called 'REFCLK' which is used in the ATPG model. We'll add this extra pin to the model so that we can use it within our test pattern. Adding pins like this makes it impossible to share this test pattern

with other environments. If possible test patterns should only use pins available on the dut.

Line 25:   Add the 'dut' to the checker visitor pattern. This checker needs access to the data model to extract various pieces of information from the model.

Lines 28-31:   The test fixture must contain a virtual destructor and must delete the model created returned from the ModuleFactory.

Lines 33-50:   Create a method for pulsing the REFCLK pin previously created. This method uses preprocessor directives to control how to generate stimulus by either explicitly insering Qinit visitor pattern loop directives or to explicitly drive the pin.

Lines 53-72:   Test fixture setup method used to setup the test fixture once its been constructed.

Lines 54-61:   Set various StimGen flags to control functionality and debug information. See corresponding documentation.

Line 68,72:   Add the checker and debug visitor patterns to so that useful information can be generated. You can add as many visitor patterns to to the dut as you want. These are optional but may be very useful. Note that we did not add the ATPG visitor patterns

Lines 75-79:   When we are done with the test fixture the TearDown method is called. This currently used to report the number of miscompares that the checker found.

15.4.2   Sample Test Pattern

```
1. #include "15h_CZ_atpg_base.h"
2. #include "sg_Event_Visitor_Qinit.h"
3.
4. using namespace StimGen;
5.
6. TEST_F ( CZATPG, KernCZ_programming ) {
7.   Event_Visitor_Qinit qinit ( "KernCZ_programming.sg.qinit" );
8.   dut->add_event_visitor( qinit );
9.
10.     Module* fch = dut->get_instance("FCH");
11.
12.     if ( SCM ) {
13.       dut->write_env( "Qinit", "MentorTimeplate = tp_capture_scm" );
14.     } else {
15.       dut->write_env( "Qinit", "MentorTimeplate = tp_capture" );
16.     }
17.
18.     fch->write_comment( "LOAD FCH JTAG REGISTERS\nTO OVERRIDE LDT_PWROK /
   LDT_RST_L iMODE SIGNAL");
19.     (*dut)["IPCONFIG"]["IPFCH"](1).write().apply();
20.
21.     (*fch)["ATECONFIG"](0)   // Set all bits in the register to 0
22.                              // Now set bits of interest
23.                      ["ATECFGJtSpare"](1)
24.                      ["JtAteCgPllCfgSel"](1)
25.                      ["JtAteCgPllGainEn"](0x6)
```

```
26.                          ["JtAteCgXtalCfgSel"](1)
27.                          ["JtPrepIoS5SafeState"](1)
28.                          ["JtPrepIoSafeState"](1)
29.                          ["JtAteDftTxEn"](1)
30.                          ["JtShortInit"](1)
31.                          ["JtAteDftRxEn"](1).write();
32.
33.      fch->write_comment ( "Set IO Safe State" );
34.      (*fch)["ATECONFIG"]["JtIoS5SafeState"](1)
35.                          ["JtAteIoSafeState"](1)
36.                          ["JtAteUsb3IoSafeState"](1).write();
37.
38.      fch->write_comment ( "Sysdebug" );
39.
40.      //##########################################################################
41.      //# KernCZ :: SYSDEBUG REGISTER ( Address : 10'h2A0 )
42.      //# - 39 JtSysDbgOverrideCpuPwrOk
43.      //# - 38 JtSysDbgCpuPwrOk
44.      //##########################################################################
45.        (*fch)["SYSDEBUG"]("64'h00000600_00000000");
46.
47.        (*fch)["SYSSCAN"](0)
48.                          ["JtSysScanEn"](1).write();
49.
50.        fch->apply();
51.      }
```

Line 1:   Include the test fixture definition.

Line 2:   Include the ATPG QINIT visitor pattern

Line 4:   Instruction compiler to import the StimGen namespace so that we don't need to prefix
                 StimGen objects with 'StimGen::'

Line 6:   Create a new test name 'KernCZ_programming' using test fixture 'CZATPG'. The test
                 fixtures are all created using MACROS.

Line 7:   Create the qinit visitor pattern. Its first argument is the name of the file to create.

Line 8:   Add the qinit visitor pattern to the dut. When ever the apply method is called events
                 are passed to the visitor pattern and process accordingly.

Line 9:   Create a local variable to hold an instance of the FCH so that we can apply operations
                 to the FCH directly without having to create operations initiated from the top level
                 dut.

Lines 10-16:   Use a test fixture attribute to pass special directives to the qinit visitor pattern only.
                 Recall the we also add debug and checker visitor patterns to the dut and they will
                 ignore all write_env directives that are not targeted to them. The write_env directive
                 allows us to pass information to environments which know how to handle them.

                 Its important to note that all of these directives and coming read/write operations are
                 queued up internally within stimGen and aren't passed to the visitor patterns until and
                 apply is issued.

Line 28:   Insert a comment into the generated output

Line 19:   Write to register called IPCONFIG's bitfield IPFCH and schedule a write operation to

	set it to a 1. This write is taking advantage of the ability to chain operations and assignments together. Note that there is an apply issued here and all queued up operations will be processed.
Lines 21-31:	There is a lot going on in this one line. We are going to write to register ATECONFIG. Note that the first thing we do is assign the entire register to a 0 using the overloaded operator(). This is available for registers and bitfields which we saw above. This is clearing out all previsously written data and reset information.

Its very important to understand that StimGen is keeping track of the state of registers and if you don't explicitly assign a value to the register or field it will reload the previously load value or documented reset value.

Another important item to note is that we don't have to know what the register width or bitfield width is; the stimGen data model knows what the width is from the documentation and takes care of that for use. This is true for all of the bitfields as well. If you feel obligated to specify register or bitfield widths in the assignments you can use verilog notation as long as it is double quoted ("8'hFF").

Now that the register value has been set to 0 we're going to go and assign individual bitfields to desired values. All fields not called out will be written to 0 in this case because of the register assignment previously. Had we not written the register to 0, any bitfields not explicitly called would be rewritten to their previous written or reset state.

| Lines 33-48: | More register write and comment instructions |
| Line 50: | The final apply for the pattern. Make sure that all patterns end with an apply otherwise stimGen will throw and exception indicating that there are pending operations queued up that did not get processed. |

15.5   Adding ATPG Patterns

To add new ATPG patterns simply create a new '.cpp' file in the atpg directory or add a new 'TEST_F' method into an existing file. All '.cpp' files in the atpg directory will be included when you rerun the 'make atpg' command.

15.6   Embedding Self Check Code

The checker visitor pattern enables us to check that StimGen is generating the correct protocol stimulus. This enables self checking. The checker visitor pattern queues up a list of protocol based events that should be created within an apply. For more information on the Checker visitor pattern see (BOZO what's the link)

16   Module

The StimGen::Module is the top level container and one of the primary objects to work with. Every module contains:
5. Pins
6. Registers
7. Instances of other modules
8. Protocols

16.1   Pins

For the most part pins are ignored at all levels of the design except for the top level where they are associated with protocols.

16.2   Registers

The primary purpose of StimGen is to read and write registers. Each level of hierarchy in a design must have uniquely named registers. Different levels of hierarchy may have registers with the same name that have either the same of different definitions.

There are several types of registers supported in stimGen. The basic register has a static definition and all other registers in some fashion pick and choose zero or more of these basic registers to stitch together based on some condition to form a meaningful definition.

A register is used to represent state and therefore has various values associated with it. The register can have one to an infinite number of bits wide.

You can not create a copy of a register in stimgen and must get either a reference or pointer to the register. Never delete a register, stimGen will do that for you and handle its own garbage collection.

16.2.1   Write a Register

To write a register from a module you need to get the register by name, assign a value to it and call the write method against it. There are a number of ways to do this. Assume 'instance' is a module that has a 16 bit register called foo.

```
// Write 1st way ( Not recommended because we need to get foo multiple
// times which can get expensive
instance["foo"]="16'hcafe";
instance["foo"].write();

// Write a 2nd way
Register &foo = instance["foo"];
foo = "16'hcafe";
foo.write();

// Write a 3rd way
instance.get_register("foo")->set_load_value("16'hcafe")->write();

// Write a 4th way
instance.["foo"]("16'hcafe").write();
```

16.2.2   Read a Register

To read a register from a module you need to get the register by name, assign the measure value and call the read method against it. There are a number of ways to do this. Assume 'instance' is a module that has a 16 bit register called foo.

```
// Read 1st way ( Not recommended because we need to get foo multiple
// times which can get expensive
instance["foo"].set_measure_value("16'hcafe");
instance["foo"].read();

// Read a 2nd way
Register &foo = instance["foo"];
foo.set_measure_value("16'hcafe");
foo.read();

// Read a 3rd way
instance.get_register("foo")->set_measure_value("16'hcafe")->read();

// Read a 4th way
// Set write data to X to rewrite previous data in the case we write
// and read at the same time ( JTAG access )
instance.["foo"]("X", "16'hcafe").read();
```

17   XCastModule

There is a special type of module that contains broadcast register networks. In order to broadcast, the networks must have registers that have exactly the same definitions and access mechanisms. The XCastModule redefines the operator[] and get_register methods so that they will return a Register derived class that contains a list of registers that you can then write or read as defined above.

18   Number and MeasureNumber

18.1   Number

18.1.1   Overview

Class "Number" represents a unsigned fixed-width four-state bit vector. It has semantics similar to a Verilog packed logic array. A "Number" has a width (in bits) and a value, which is less than 2^{width} . The width is stored in an "uint32_t" so it can, in theory, support any width up to 2^{32} (provided there sufficient memory to allocate the storage for the value).

Class "Number" is not intended to be used as a base class. It has no virtual methods (in particular, its destructor is non-virtual). **This ensures that there is no vtable pointer associated with a "Number" object. "Number" objects are the size of a single pointer, so they can be efficiently passed by value.**

The definitions of the "Number" methods and operators corresponds closely to the definitions for logic vectors in IEEE Std 1800-2012. **The main places where definitions differ are operators "==" and "<". In Verilog, these return X if either argument is X. The operators defined here return a "bool". "Number" operator "==" is equivalent to Verilog operator "===" (case equality). Method "is_logically_equal" performs a test equivalent to Verilog operator "==". "Number" operator "<" is described below.**

18.1.2   Bit State Representation

Enumeration "Bit4State" is used to represent the four-state value of a single bit in function arguments and return values. There is an additional mask for 'R' meaning "record", which is used only in class "MeasureNumber". **Note that 'R' is not actually a state of bit and should not be applied on arithmetic operations.** The detailed definition of "R" is described in "MeasureNumber" section. The enumeration has the following definition:

```
enum
Bit4State
{ Bit4State_0     = 0
, Bit4State_1     = 1
, Bit4State_X     = 2
, Bit4State_Z     = 4
} ;
```

18.1.3   Constructors

18.1.3.1   Constructor with unsigned decimal integer

```
1) Number( );
```

Default constructor, the width of new "Number" object is 0.

2) Number(`const` `Number` & num);

Copy constructor. No deep copy is made, only the reference count is incremented.

3) Number(`const` `Number` & num , `uint32_t` width);

Constructs a Number using the value of existing Number "num". The width of the new Number will be "width". "num" will be truncated or 0-extended on the left side to match "width".

4) Number(`uint32_t` value , `uint32_t` width = 32);

   Number(`uint64_t` value , `uint32_t` width = 64);

   Number(`int32_t` value , `uint32_t` width = 32);

   Number(`int64_t` value , `uint32_t` width = 64);

Construct a "Number" object with value "value" and width "width".

5) Number(StimGen::`Number`::`RangeRef` range_ref) ;

Construct a Number object with a RangeRef object 'range_ref'. [msb,lsb] of 'range_ref' is assigned to [msb-lsb,0] of the new Number object. The width of the new Number object is msb – lsb + 1.

18.1.3.2   *Constructor with string representing number*

6) Number(std::`string` `const` & value , `uint32_t` width = 0);

Construct a "Number" with value "value" and width "width". Value is a string representing number. The width is defined differently in different cases which is described in the tables below. The string representing number consists of two parts: 1) prefix; 2) number value after the prefix; except for a special case that are used to initialize all the bits with the same value. The prefix format and the definition is described as below. "<width>" is a string of decimal digits beginning with a non-zero digit(i.e., a positive decimal number with no leading zeros). **Note that if 1) no prefix is present, 32'd will assumed(see table II: Verilog format, last row); 2) no number value after prefix, it will throw exceptions.**

Table I: C format

C format: C-style prefix + value		
Prefix	Description	Examples

0b, 0B	Binary number. If the constructor parameter "width" is not specified, the width of new "Number" object will be 64.	`// n0 is "64'h00000001"`
		`StimGen::Number n0("0b1") ;`
		`// n1 is "64'h0000000X"`
		`StimGen::Number n1("0bX") ;`

0x, 0X	Hexadecimal number. If the constructor parameter "width" is not specified, the width of new "Number" object will be 64.	`// n0 is "64'h0000000A"`
		`StimGen::Number n0("0xA") ;`
		`// n1 is "64'h0000000X"`
		`StimGen::Number n1("0xX") ;`

Table II: Verilog format

Verilog format: Verilog prefix + value		
Prefix	Description	Examples

'b
'B
	Unsized binary number. If the constructor parameter "width" is not specified, the width of new "Number" object will be determined by the length of the string.	```cpp
		// n0 is "1'b1"
		StimGen::Number n0("'b1") ;
		// n1 is "4'b1111"
		StimGen::Number n1("'b1_111") ;
		// n2 is "5'b01111"
		StimGen::Number n2("'b1_111" , 5) ;
		``

<width>'b
<width>'B
	Sized binary number. If the constructor parameter "width" is not specified, the width of new "Number" object will be determined by the <width> prefix of string	```cpp
		// n0 is "1'b1"
		StimGen::Number n0("1'b1") ;
		// n1 is "5'b01111"
		StimGen::Number n1("5'b1_111") ;
		// n2 is "6'b001111"
		StimGen::Number n2("5'b1_111" , 6) ;
		``

'h
'H
	Unsized hexadecimal number. If the constructor parameter "width" is not specified, the width of new "Number" object will be determined by the length of the string.	```cpp
		// n0 is "4'h1"
		StimGen::Number n0("'h1") ;
		// n1 is "8'hA0"
		StimGen::Number n1("'hA_0") ;
		// n2 is "4'hA"
		StimGen::Number n2("'hAAAAA" , 4) ;
		``

<width>'h
<width>'H
	Sized binary number. If the constructor parameter "width" is not specified, the width of new "Number" object will be determined by the <width> prefix of string.	```cpp
		// n0 is "4'h1"
		StimGen::Number n0("1'h1") ;
		// n1 is "4'h0"
		StimGen::Number n1("4'hA_0") ;
		// n2 is "4'hA"
		StimGen::Number n2("16'hAAAA" , 4) ;
		``

<width>'d
<width>'D
	Sized decimal number. If the constructor parameter "width" is not specified, the width of new "Number" object will be determined by the <width> prefix of string. Once there is an 'X' in the decimal string, all digits will be 'X'. Note that "unsized decimal number" is not a legal case.	```cpp
		// n0 is "4'h1"
		StimGen::Number n0("1'd1") ;
		// n1 is "32'hXXXXXXXX"
		StimGen::Number n1("32'd111X") ;
		// n1 is "32'hXXXXXXXX"
		StimGen::Number n1("32'd111X") ;
		``

No-prefix	32'd is assumed.	```cpp
		// n0 is "32'h1"
		StimGen::Number n0("1") ;
		``

Note that for all the cases above, if the constructor parameter "width" is larger that the width determined by the string representing number, zero-extension on the left side will occur.

Table III: Special format

Special format: ' + one digit		
Format	Description	Examples

'0
'1
'X
'x
'Z
	Initialize all the bits with the same digit. If the constructor parameter "width" is not specified, the width of new "Number" object will be 64.	```cpp
		// n0 is
		// "64'hXXXXXXXXXXXXXXXX"
		StimGen::Number n("'X") ;
		// n1 is
		// "64'h1111111111111111"
		StimGen::Number n1("'1") ;
		// n2 is
		``

'z		`// "16'hZZZZ"` `StimGen::Number n("'Z", 16) ;`

The valid characters for the value after prefix are described as below. Note that illegal characters will throw exceptions.

Table IV: Value after the prefix

Value after the prefix	
Prefix	Legal characters after the prefix
0b, 0B 'b, 'B <width>'b, <width>'B	Binary strings may contain only the following characters after the prefix: - '0' - '1' - 'x', 'X' - 'z', 'Z' - '_'(a separator, will be ignored in string parse)
0x, 0X 'h, 'H <width>'h, <width>'H	Hexadecimal strings may contain only the following characters after the prefix. - '0' -- '9' - 'A' -- 'F', 'a' -- 'f' - 'X', 'x' - 'Z', 'z' - '_'(a separator, will be ignored in string parse)
<width>'d, <width>'D	Decimal strings may contain only the following characters after the prefix. - '0' -- '9' - 'X', 'x' - 'Z', 'z' - '_'(a separator, will be ignored in string parse) **Note that if the string contains 'X' or 'Z' then all the bits will be set to 'X'.**

18.1.4   Bit and range access

18.1.4.1   *Bit Access*

`Bit4State operator [] ( uint32_t offset ) const;`

`BitRef operator [] ( uint32_t offset );`

Return bit value at index "offset" from the current "Number" object. The non-constant version returns a "BitRef" proxy object, which is an lvalue that allows modifying operations on the bit. The constant version returns the value as a "Bit4State" rvalue. Both give an error if "index" is outside the index

range of the current object.

18.1.4.2   *Range Access*

```
Number operator ( ) ( uint32_t msb , uint32_t lsb ) const;
```

```
RangeRef operator ( ) ( uint32_t msb , uint32_t lsb );
```

Return bit range [msb:lsb] from the current "Number". The non-constant version returns a "NumberRef" proxy object, which is an lvalue that allows modifying operations on the bit range. The constant version returns the value as a "Number" rvalue. Both give an error if [msb:lsb] is empty or not entirely within the index range of the current object.

```
Number get_interval( uint32_t offset , uint32_t width ) const;
```

Extracts a subset of continuous bits and returns a new Number object. The subset is [offset , offset + width).

18.1.5   Assignment operators

```
1) Number & operator =( const Number other );
```

Copy value of "other" to current "Number" object. It does not change the width of current object. Assigned value is truncated or zero-extended on the left side to match the width of current "Number" object. Examples:

```
StimGen::Number n0( "4'b1111" );
StimGen::Number n1( "3'bXXX" );
StimGen::Number n2( "5'bXXXXX" );
StimGen::Number n3( "4'b01XZ" );

n0 = n1; // n0 is 4'b0XXX
n0 = n2; // n0 is 4'bXXXX
n0 = n3; // n0 is 4'b01XZ
```

There are two other assignment operators for range assignments. "RangeRef" is a reference to a certain range of the "Number" object.

```
2) RangeReference & operator = ( RangeRef & other );
```

```
3) RangeReference & operator = ( Number & other );
```

Some examples of the usage of range assignment operators:

```
StimGen::Number n0( "4'b1111" );
StimGen::Number n1( "3'bXXX" );
StimGen::Number n2( "5'bXXXXX" );
StimGen::Number n3( "4'b01XZ" );

n0 = n1( 1 , 0 ); // n0 is 4'b00XX
n0 = n2( 3 , 0 ); // n0 is 4'bXXXX
n0 = n3( 3 , 2 ); // n0 is 4'b0001
n0( 3 , 2 ) = n1( 1 , 0 ); // n0 is 4'bXX01
```

18.1.6   Binary logical operators and arithmetic operators

```
STIMGEN_API Number operator &( Number lhs , Number rhs );
STIMGEN_API Number operator |( Number lhs , Number rhs );
STIMGEN_API Number operator ^( Number lhs , Number rhs );
STIMGEN_API Number operator +( Number lhs , Number rhs );
STIMGEN_API Number operator -( Number lhs , Number rhs );
STIMGEN_API Number operator *( Number lhs , Number rhs );
STIMGEN_API Number operator /( Number lhs , Number rhs );
STIMGEN_API Number operator %( Number lhs , Number rhs );
STIMGEN_API Number operator <<( Number lhs , Number rhs );
STIMGEN_API Number operator >>( Number lhs , Number rhs );
```

Binary operators are implemented as namespace-scope functions. This allows conversion to be applied equally to both arguments. The functions are implemented using the assignment versions of the operators, which are defined as member functions (see below).

The bitwise logical and arithmetic binary operators all take two "Number" arguments. **They return a "Number" whose width is the maximum of the widths of the arguments, and zero-extend the narrower argument to the width of the wider argument before applying the operation.**

Binary operators are defined on X and Z as in IEEE Std 1800-2012.

In arithmetic operators, if any bit in "lhs" or "rhs" is X or Z, then the result is all Xs, as in IEEE Std 1800-2012.

18.1.7   Assignment binary operators and arithmetic operators

```
Number & operator &=( Number lhs , Number rhs );
Number & operator |=( Number lhs , Number rhs );
Number & operator ^=( Number lhs , Number rhs );
Number & operator +=( Number lhs , Number rhs );
Number & operator -=( Number lhs , Number rhs );
Number & operator *=( Number lhs , Number rhs );
Number & operator /=( Number lhs , Number rhs );
Number & operator %=( Number lhs , Number rhs );
Number & operator <<=( Number lhs , Number rhs );
Number & operator >>=( Number lhs , Number rhs );
```

Assignment versions of the binary operators are implemented as member functions. These functions are used by namespace-scope operators to implement the underlying operators. **The assignment causes the value of the result of the binary operation to be truncated or zero-extend on the left side to the width of the current "Number". Note that the rule about unknown bit values applies even to bits that are of range of the result.** For example, the following code assign 4'bXXXX to "n"

```
StimGen::Number n( 0 , 4 ) , p( "6'bX01010" );
n += p;
```

18.1.8   Comparison operators and functions

```
STIMGEN_API bool operator ==( Number lhs , Number rhs );
```

```
STIMGEN_API bool operator !=( Number lhs , Number rhs );
```

This is equivalent to Verilog "==="(case equality) and "!==" respectively. Return true if "lhs" and "rhs" are bit-by-bit equal, no equal respectively. If the widths of "lhs" and "rhs" differ, the shorter one is zero-extended on the left side before the comparison.

```
STIMGEN_API bool operator <( Number lhs , Number rhs );
```

```
STIMGEN_API bool operator <=( Number lhs , Number rhs );
```

```
STIMGEN_API bool operator >( Number lhs , Number rhs );
```

```
STIMGEN_API bool operator >=( Number lhs , Number rhs );
```

Return true if "lhs" is less than, less than or equal to, greater than, greater than or equal to, respectively, "rhs". If the widths "lhs" and "rhs" differ, the shorter is zero-extended before the comparison. **Note that these operators return "false" in cases where the arguments contain X or Z. In Verilog, comparisons return X in these cases.**

```
Bit4State is_logic_equal( Number const rhs ) const;
```

```
bool is_case_equal( Number const rhs ) const;
```

"is_logic_equal" is equivalent to Verilog "==", which returns an X when there is any of the bit of this or "rhs" is X. "is_case_equal" is equivalent to Verilog "===".

```
bool is_equal_wildcard_X_and_Z( Number rhs ) const ;
bool is_equal_wildcard_X( Number rhs ) const ;
```

Test if the current Number is equal to "rhs". In the first function, 'X' and 'Z' are wild-card state which means the following statements are all true:

X == 0, 0 == X, X == 1, 1 == X, X == Z, Z == X, X == X

Z == 0, 0 == Z, Z == 1, 1 == Z, Z == X, X == Z, Z == Z.

In the second function, only 'X' is wild-card state, which means the following statements are all true:

X == 0, 0 == X, X == 1, 1 == X, X == Z, Z == X, X == X

Z != 0, 0 != Z, Z != 1, 1 != Z, Z == X, X == Z, Z == Z.

18.1.9   Testing and Query functions

```
uint32_t num_<v>s( uint32_t msb , uint32_t lsb ) const;
uint32_t num_<v>s( ) const;
```

Return the number of <v>s, for <v> is one of 0, 1, X, Z, 0_or_1, X_or_Z. The first version counts <v> within range[msb,lsb]. The second version counts <v> in the whole number.

```
bool has_<v>( uint32_t msb , uint32_t lsb ) const;
bool has_<v>( ) const;
```

Return true if any of the bits has <v>, for is one of 0, 1, X, Z, 0_or_1, X_or_Z. The first version tests <v> within range[msb,lsb]. The second version tests <v> in the whole number.

```
bool is_<v>( uint32_t offset ) const;
```

Return true if the bit at index "offset" is <v>, for <v> is one of 0, 1, X, Z, 0_or_1, X_or_Z.

18.1.10   Type conversion functions

```
std::string to_<base>( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). No prefix is added.

```
std::string to_<base>_c( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). C-style(Please refer *Table I*) prefix is added.

```
std::string to_<base>_v( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). Verilog-style(Please refer *Table II*) prefix is added.

18.1.11   Capacity change functions

```
Number & resize( uint32_t size );
```

Resize the current Number object to width "size". If "size" is less than the current width, it will truncate the left side to fit "size"; if "size" is equal to the current width, nothing will happen; if "size" is larger than the current width, it will 0-extend on the left side to fit "size".

```
void append( const Number other );
```

```
void prepend( const Number other );
```

"append" appends "other" to the right side of "this"; "prepend" appends "other" to the left side of "this". The new width of "this" is the sum of "this" and "other".

18.2   MeasureNumber

18.2.1   Overview

Class "MeasureNumber" is an extension based on "Number" Class to support R mask in addition to traditional 4 states. It's used to represent "measure_value" in "StimGen::Register". Any places require "measure_value" should use "MeasureNumber". Note that "MeasureNumber" is not an inheritance from "Number" class but takes "Number" as a public data member. All the member functions of "Number" class can be accessed by calling "measure_number_obj.num.functions(parameters)". The data structure of "MeasureNumber" is described below. There are no binary logic operations or arithmetic operations are defined in "MeasureNumber", but users can always define their own operations by directly manipulate public data member "num" and "rmask".

18.2.2   Data structure

```
STIMGEN_API
class STIMGEN_API MeasureNumber
{
public:
  functions( parameters );

private:
  functions( parameters );

public:
  StimGen::Number num;
  std::vector<bool> rmask;
};
```

18.2.3   Definition of R

"R" means "record", which tells the users to record the bit value for post-processing despite what the value is. The value of a single bit is defined in the following table:

num	rmask	Bit value
0	0	0
1	0	1
X	0	X
Z	0	Z
0	1	R
1	1	R
X	1	R
Z	1	R

18.2.4   Constructors

18.2.4.1   Constructors with "Number"

```
1) MeasureNumber( );
```

Default constructor, the width of new "MeasureNumber" object is 0. Note that the size of "rmaks" is also 0.

```
2) MeasureNumber( const MeasureNumber & mnum );
```

Copy constructor.

```
3) MeasureNumber( const MeasureNumber & mnum , uint32_t width );
```

Constructs a "MeasureNumber" using the value of existing "MeasureNumber" "mnum". The width of the new "MeasureNumber" will be "width". "mnum" will be truncated or 0-extended on the left side to match "width".

```
6) MeasureNumber( uint32_t value , uint32_t width = 32 );
   MeasureNumber( uint64_t value , uint32_t width = 64 );
```

Construct a "MeasureNumber" object with value "value" and width "width". "rmaks" is all set to 0.

```
7) MeasureNumber( const Number value );
```

Construct a "MeasureNumber" object with a "Number" object "value". "num" is initialized with "value", and "rmask" is all set to 0s.

18.2.4.2   Constructor with string representing number

Most cases are the same as in "Number", except that the value after prefix will take 'R' and 'r' as legal characters.

Table IV: Special format

Special format: ' + one digit		
Format	Description	Examples
'0 '1 'X 'x 'Z 'z 'R 'r	Initialize all the bits with the same digit. If the constructor parameter "width" is not specified, the width of new "Number" object will be 64.	```// n0 is` `// "64'hXXXXXXXXXXXXXXXX"` `StimGen::Number n("'X") ;` `// n1 is` `// "64'h1111111111111111"` `StimGen::Number n1("'1") ;` `// n2 is` `// "16'hZZZZ"` `StimGen::Number n("'Z", 16) ;` `// n3 is` `// "16'hRRRR"` `StimGen::Number n("'R", 16) ;```

Value after the prefix	
Prefix	Legal characters after the prefix
0b, 0B 'b, 'B <width>'b, <width>'B	Binary strings may contain only the following characters after the prefix: - '0' - '1' - 'X', 'x' - 'Z', 'z' - 'R', 'r' - '_'(a sperator, will be ignored in string parse)

0x, 0X 'h, 'H <width>'h, <width>'H	Hexadecimal strings may contain only the following characters after the prefix. - '0' -- '9' - 'A' -- 'F', 'a' -- 'f' - 'X', 'x' - 'Z', 'z' - 'R', 'r' - '_'(a sperator, will be ignored in string parse)
<width>'d, <width>'D	Decimal strings may contain only the following characters after the prefix. - '0' -- '9' - 'X', 'x' - 'Z', 'z' - 'R', 'r' - '_'(a sperator, will be ignored in string parse) Note that if the string contains 'X' or 'Z', all the bits will be set to 'X'. If the string constains 'R', all the bits will be set to 'R'

18.2.5   Assignment operators

`MeasureNumber & operator =( const MeasureNumber other );`

Copy value of "other" to current "MeasureNumber" object. It does not change the width of current object. Assigned value is truncated or zero-extended on the left side to match the width of current "Number" object. Note that "rmask" is also truncated or zero-extended on the left side.

18.2.6   Binary logical operators and arithmetic operators

Note that only shifting operators are defined in the class.

`STIMGEN_API MeasureNumber operator >>( MeasureNumber n , uint32_t distance );`

`STIMGEN_API MeasureNumber operator <<( MeasureNumber n , uint32_t distance );`

Shifting operators will shift both "num" and "rmask" accordingly. Note that users can always shift only "num" or "rmask" by directly manipulating "num" and "rmask".

18.2.7   Comparison operators

`STIMGEN_API bool operator ==( MeasureNumber lhs , MeasureNumber rhs );`

Returns "(lhs.num == rhs.num && lhs.rmask == rhs.rmask)". If the length differ, the shorter value is resized(please refer function "resize") to the length of longer before comparing.

`STIMGEN_API bool operator !=( MeasureNumber lhs , MeasureNumber rhs );`

Returns "!(lhs == rhs)".

`bool MeasureNumber::matches( const Number other ) const;`

Returns true if every bit of "n" where the current "rmask" is "false" is equal to the corresponding bit of

the current "num", and false otherwise. If "n" and the current "MeasureNumber" have different lengths, the shorter one is resized(please refer function "resize") to the length of the longer one before comparing.

18.2.8   Testing and Query functions

```
uint32_t num_Rs( uint32_t msb , uint32_t lsb ) const;
uint32_t num_Rs( ) const;
```

Return the number of Rs. The second version counts <v> in the whole number.

```
bool has_R( uint32_t msb , uint32_t lsb ) const;
bool has_R( ) const;
```

Return true if any of the bits has R. The second version tests <v> in the whole number.

```
bool is_R( uint32_t offset ) const;
```

Return true if the bit at index "offset" is R.

18.2.9   Type conversion functions

```
std::string to_<base>( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). No prefix is added. The string may contain 'R'. Please refer section "Definition of R" for details.

```
std::string to_<base>_c( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). C-style(Please refer *Table I*) prefix is added. Please refer section "Definition of R" for details.

```
std::string to_<base>_v( ) const;
```

Converts the Number to a string representing number of base <base>, for <base> is one of "bin"(binary), "dec"(decimal) and "hex"(hexadecimal). Verilog-style(Please refer *Table II*) prefix is added. Please refer section "Definition of R" for details.

18.2.10   Capacity change functions

```
void append( const MeasureNumber other );
```

```
void prepend( const MeasureNumber other );
```

"append" appends "other" to the right side of "this"; "prepend" appends "other" to the left side of "this". The new width of "this" is the sum of "this" and "other".

```
MeasureNumber resize( uint32_t length ) const;
```

Returns a copy of the current "MeasureNumber", resized to length "len". "num" and "rmask" are truncated if "len" is less than the length of the current `MeasureNumber', and zero-extended if it is greater.

18.3   Interface Transition Guide

Old functions	New functions
Number (const std::string &value, unsigned int b = 0)	Number (const std::string &value, unsigned int width = 0)
Number (const char* const &value, unsigned int b = 0)	Number (const std::string &value, unsigned int width = 0)

void load_width_and_value_from_string (const std::string&)	Use format '<base><value> in the above constructors it will load width and value.
	
	Examples:
	Number n0("'b0110") ; // n0 is 4'b0110
	Number n1("'hAAXXX") ; // n1 is 20'hAAXXX

Number (unsigned char value , uint32_t width = 8) Number (char value , uint32_t width = 8) Number (unsigned int value , uint32_t width = 32) Number (int value , uint32_t width = 32) Number (unsigned long value , uint32_t width = 32) Number (long value , uint32_t width = 32) Number (unsigned long long value , uint32_t width = 64) Number (long long value , uint32_t width = 64)	Number(uint32_t value , uint32_t width = 32) Number(uint64_t value , uint32_t width = 64) Number(int32_t value , uint32_t width = 32) Number(int64_t value , uint32_t width = 64) Note: User needs to explicitly specify the width when constructing a number narrower than 32 bits.

string as_<base>_no_prefix()	string to_<base>()
	
	Examples:
	n.as_bin_no_prefix() => n.to_bin( )
	n.as_hex_no_prefix() => n.to_hex( )
	
	note: in the old interface, as_<base>_no_prefix can convert X to 0 or 1, which depends on the previous value. In the new interface to_<base>_c() will convert X to X, which will not depend on the previouse value.
	
	Examples:
	Number n("4'bXXXX");
	std::cout << n.as_bin_no_prefix() ; // output: 0000
	std::cout << n.to_bin() ; // output: XXXX

string as_<base>()	string to_<base>_c() note: in the old interface, as_<base> can convert X to 0 or 1, which depends on the previous value. In the new interface to_<base>_c() will convert X to X, which will not depend on the previouse value. See the examples above.
string as_<base>_verilog()	string to_<base>_verilog()
unsigned int as_unsigned_int () const	uint32_t to_uint32( ) const
unsigned long as_unsigned_long () const	uint32_t to_uint32( ) const or uint64_t to_uint32( ) const This depends on the compiler used.
unsigned long long as_unsigned_long_long () const	uint64_t to_uint64( ) const
value_as_<base>()	TBA
void Bit_On (unsigned int index)	n.Bit_On(i) => n[i] = StimGen::Bit4State_1
void Bit_Off (unsigned int index)	n.Bit_Off(i) => n[i] = StimGen::Bit4State_0
void Mask_On (unsigned int index)	n.Mask_On(i) => n[i] = StimGen::Bit4State_X
void Mask_Off (unsigned int index)	The return of the old function could be 0, 1, or Z. User should set to appropriate value accordingly
void clear_mask ()	fill(StimGen::Bit4State_X)
void fill_mask ()	The return of the old function could be all 0, all 1, or all Z. User should set to appropriate value accordingly.
void Fill( )	fill(StimGen::Bit4State_1)
bool is_mask_full () const	!n.has_X( )

`bool is_mask_empty () const`	`n.is_all_X( )`
`bool is_bit_masked ( unsigned int index ) const`	`bool is_X( uint32_t index ) const`
`bool is_bit_z ( unsigned int index ) const`	`bool is_Z( uint32_t index ) const`
`bool is_full () const`	`bool is_all_1( ) const`
`bool is_zero () const`	`bool is_all_0( ) const`
`bool has_zero() const`	`bool has_0( ) const`
`bool bit_test () const`	`bool is_1( ) const`
`void Concat ( const Number& n )`	`void append ( const Number n  )`
`void Prepend ( const Number & n )`	`void prepend ( const Number n )`
`void Interval_Copy( const Number& n, unsigned int offset, unsigned int start, unsigned int width )`	`void interval_copy( const Number& n, unsigned int offset, unsigned int start, unsigned int width )`
`bool verilog_compare_neq_eq_eq( const Number rhs ) const`	`!n.is_case_equal( rhs )`
`bool verilog_Compare_eq_eq_eq( const Number rhs ) const`	`n.is_case_equal( rhs )`

`bool verilog_compare_eq_eq( const Number rhs )`	`n.is_logic_equal( rhs ).`
	
	`Note: is_logic_equal( rhs ) returns StimGen::Bit4State_X whenever there is an X in either side. The definition is the same as Verilog "==". User need to change the code accordingly.`

`Number bnot( );`	`Number & invert( )`
`int number_of_bits_on( )`	`uint32_t count_1s( )`

19   Register Definition

At the heart of SG4 is the register. A register allows us to control or observe internal state of a device. It is defined as a uniquely addressable or accessible group of bitfields or memory elements (flops/latches). Each register the following properties:

1. A unique name

2. An address

3. Access information (read-only, write-only, read-write)

4. Reset value

5. Composed of one or many uniquely named bitfields which may or may not have their own access and reset information.

6. State

One of the requirements of SG4 is vector generation and in order to enable this SG4 registers must model and provide additional information. The following values are modeled within SG4:

1. value – Current value of the register in the DUT

2. load_value – Simulated value that SG4 will load into the DUT when vectors are being produced

3. measure_value – Simulate value that SG4 should measure from the DUT when vectors are applied

4. read_value – Value previously read from register in the DUT when vectors were applied

20 Virtual Registers

IEEE1149.1, IEEE1500 and IEEEP1687 accessible registers by design can be viewed as a single register which dynamically change their definitions growing or shrinking based on the current state of one or more control registers or pins. There are two fundamental types of virtual registers supported by StimGen:

7. Virtual Select Register (VSR)

8. Virtual Enable Register (VER)

There are also a few specialized abstract register functions to enable broadcasting type functionality:

5. Virtual Index Register (VIR)

6. Virtual Serial Broadcast Register (VSBR)

7. Virtual Serial Multicast Register (VSMR)

All virtual registers are implemented as a list of condition/register pairs. Depending on the virtual register type evaluation of the network is handled slightly differently and will be explained.

Registers within any of the virtual registers may be any type of register or other virtual register. When evaluating connectivity evaluation must traverse into each register to get the desired information.

Virtual registers as described in the document represent basic functionality and may not represent how the register structures are implemented in silicon. To correctly represent functionality for StimGen complex conditional expressions may be required to mimic what silicon has implemented. Additionally, the combination of multiple virtual registers may be needed to represent complete functionality.

When StimGen is generating vectors and encounters a Virtual Register whose control registers are not correctly preconditioned, it will automatically schedule the necessary operations to access the target register. StimGen will by default schedule an operation to restore all control registers back to their documented default or preferred state after the target register is accessed. All restore operations will be scheduled when an apply operation is issued. Users have the ability to programmatically disable the restore per register using Register method set_enable_auto_reset(bool). The runtime configuration allows a project to explicitly call out all registers that should not be restored on apply.

20.1 Virtual Select Register

The virtual select register(VSR) allows the register network to select one of many parallel registers to connect between a serial input and output. The VSR is used to model a IEEE1149.1 TIR/TDR network and a IEEE1500 WIR/WDR network.
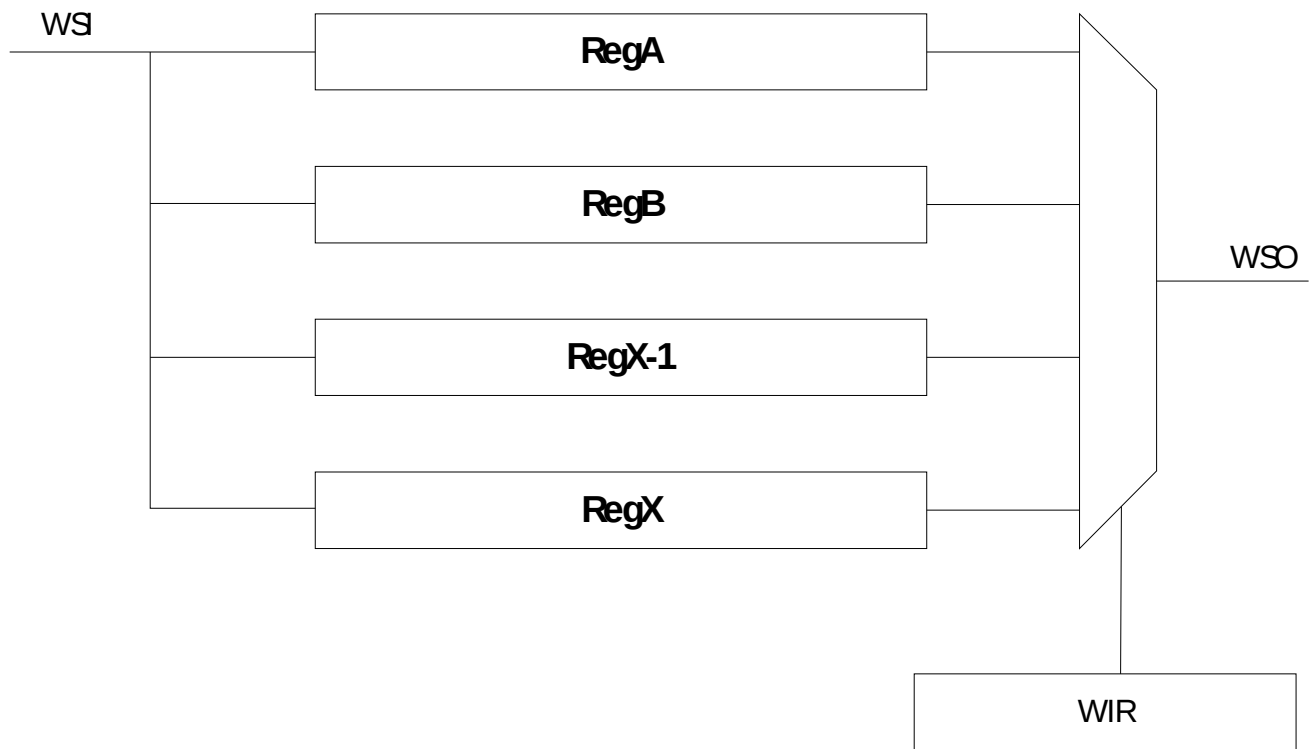
Illustration 6: Sample Virtual Select Register

20.1.1　　StimGen Handling

StimGen is will automatically update the WIR to access the target register. Every VSR should have associated with it a default index which will be selected in the case that the WIR does not select a register. When no decode is found the default register is automatically inserted between the SI and SO.

In the case that a VSR is used to represent a IEEE1149.1 TIR/TDR network for the AMD Modular JTAG networks, if no register is targeted within the TDR StimGen will automatically set the TIR to ~0 to force the BYPASS register between TDI and TDO.

20.1.2　　StimGen Evaluation

```
sub evaluate {
  foreach ( @condition_register_pairs ) {
    if ( condition is true ) {
      return evaluate(register)
    }
  # If we haven't returned no condition was true so return default
  # index if available
  if ( defined default_index ) {
```

```
        return evaluate( @condition_register_pairs[ default_index ] )
    }
    # otherwise return nothing or 0 bits
}
```

20.2   Virtual Enable Register

The virtual enable register(VER) allows the virtual register network to include or bypass a register in the larger register network. The VER is used to model AMD Modular Taps and is used to model IEEEP1687 SIB based register networks.
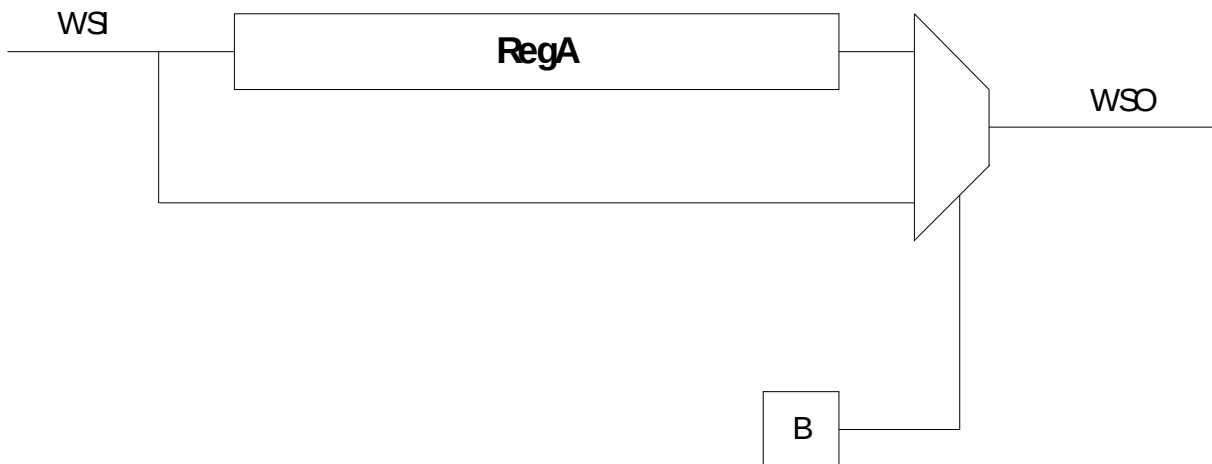
Illustration 7: Sample Virtual Enable Register

20.2.1   StimGen Handling

There are two scenarios related to VER registers where the control register is either a IEEEP1687 SIB or it is not. In the case where it is a SIB StimGen will automatically open the SIB to access the register network or include it between WSI and WSO. StimGen will not automatically close the network unless the register is marked for automatic reset on apply (BOZO). If the control signal is not marked as a SIB the user is completely responsible for managing the state of the control signal.

20.2.2   StimGen Evaluation

```
    sub evaluate {


    }
```

20.3   Virtual Index Register

The virtual index register(VIR) takes an address and data and performs a parallel indirect register read

or write operation. The VIR is currently use only for AMD Fusebox and MBIST_FSM BISTCONFIG and FATALERROR registers access.
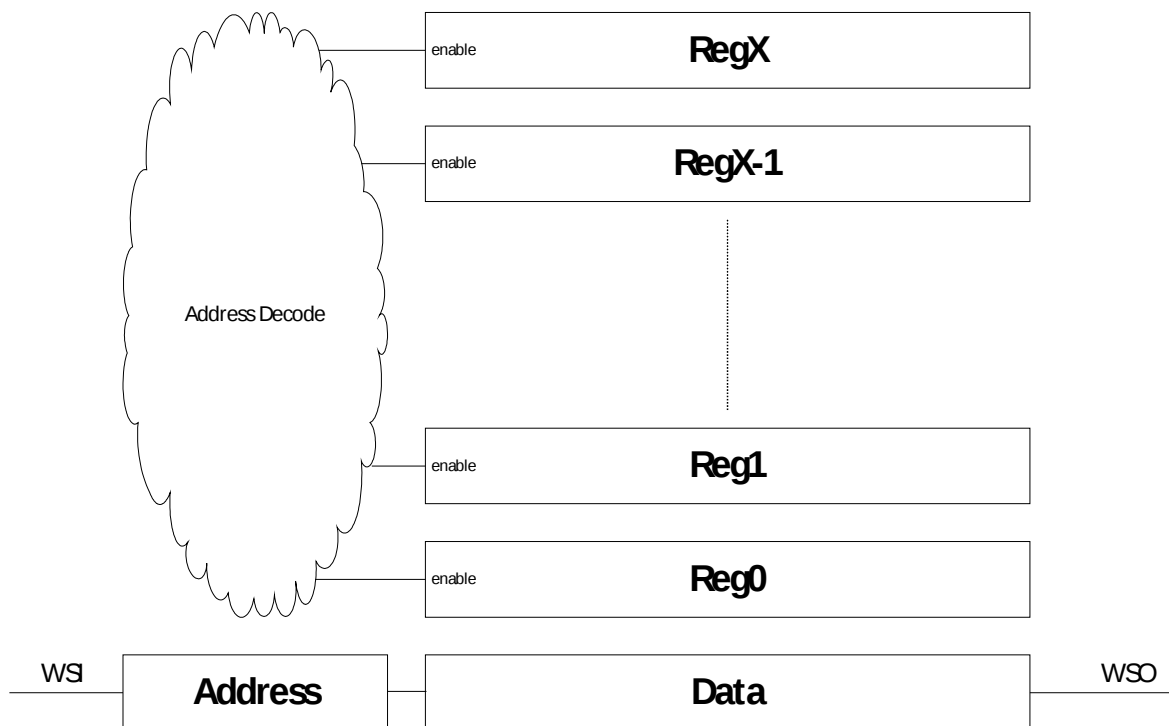
Illustration 8: Sample Virtual Index Register

20.3.1   StimGen Control Register Restore

StimGen will not reset VIR control registers.

20.4   *Virtual Serial Xcast Register*

The virtual serial xcast register or VSXR is being defined to describe basic and common functionality between the virtual serial broadcast and multicast register. Both versions of the VSXR require that all tiles in the network be analyzed to check data integrity to ensure we're broadcasting to registers of equivalent width and with identical values. If either checks fail then StimGen will print an error message to STDOUT if checking is enabled.

20.5   *Virtual Serial Broadcast Register*

The virtual serial broadcast register (VSBR) is a network designed to broadcast write ideally to identical or similar register networks. The VSBR is designed specifically to broadcast to VSR networks where a subset of the encoding have identical register definitions. When reading registers in these register networks the network must be put into serial mode where all register works are daisy

chained together. Write operations can be issued in serial mode as well. This mode of operation is required if you need to write different values into identical registers.

The VSBR also supports the ability to bypass VSBR tile networks which have a BYPASSEN control. Each tile has a unique BYPASSEN control which is typically provided internally from the tile's VSR. In this mode of operation broadcast operations are ignored by network in BYPASSEN mode and in serial mode the BYPASS register is inserted between the WSI and WSO. This functionality must be encoded into the VSR. In both cases the BYPASS register should always be inserted and be written to.

The WSO of the VSBR is always driven by the last VSR in the network. This effectively causes the network to unusable when other TDRs are connected to WSO and the VSR driving WSO is in BYPASS mode or doesn't have a decode for the current WIR encoding.
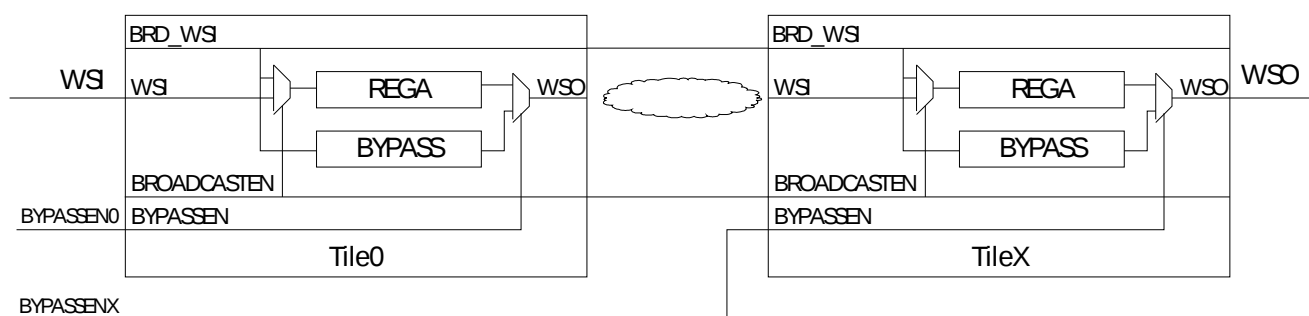
Illustration 9: Sample Virtual Serial Broadcast Register

20.5.1   StimGen Handling

Because VSR in a VSBR are not guaranteed to be identical care has to be taken in evaluating the current state of the network and ensuring that register operations are setup correctly.

20.6   Virtual Serial Multicast Register

The virtual serial multicast register or VSMR is similar to the VSBR but each tile has an ENABLE signal which is similar to NOT BYPASSEN. The VSMR uses a global daisy chain mode instead of a global BROADCASTEN. VSMR has one other difference when compared to the VSBR and when ENABLE=0 the BYPASS bit the is inserted is connected to the serial WSI and not the broadcast WSI.
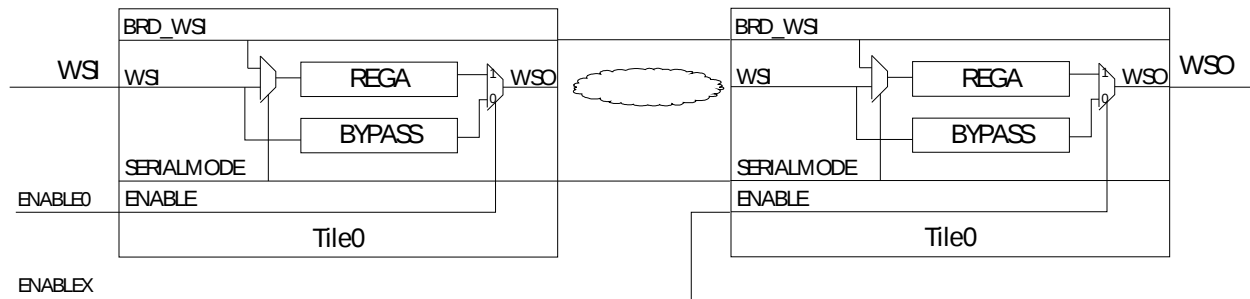
Illustration 10: Sample Virtual Serial Multicast Register

20.6.1   StimGen Handling

Unlike VSBR, the VSMR will not shift invalid data into TDRs connected to its WSO because the registers connected between the WSI and WSO provide exact data.

Each ENABLE signal of the VSMR is typically generated from within the Tile to eliminate the need for global wires.

The complexity of the VSMR is found when attempting to locate the tile that driving WSO which is not disabled and shifting through its BYPASS register.

21   AMD STAC Support / Multiple Levels of XCasting

The STAC architecture and more specifically the STAC router and its broadcasting functionality is complicated. There can be multiple levels of broadcasting to different types of Xcast register networks which can contain a mix of configurations such that some networks may be operating in a serial mode while others are broadcasting.

21.1   DFXIP1 1500 Tile Broadcast Support

<BOZO> Virtual Serial Broadcast: No deterministic WSO.

21.2   DFXIP2 1500 Tile Broadcast Support

<BOZO> Virtual Serial Multicast: Insertion of BYPASS bit when a tile is not active which provides deterministic control of WSO.

21.3   DFXIP3 Multi-Level Xcasting Support

DFXIP3 is implemented using both "1500 Tile Broadcast" at the bottom level and "STAC Broadcast" at the top level or in the BC1500 STAC instruction. The "1500 Tile Broadcast" architecture is different than previous implementations and is similar to the DFXIP2 implementation which inserts the BYPASS bit into the chain if its not enabled. The following three illustrations attempt to show the WSI to WSO connectivity through a mocked up model that contains two BC1500 STAC Broadcast domains that each contains a 1500 broadcast network configured with three tiles in each for simplicity. The logic shown is not 100% accurate and should not be used for implementation but should instead represent a high level view of the WSI $\rightarrow$ WSO connectivity.
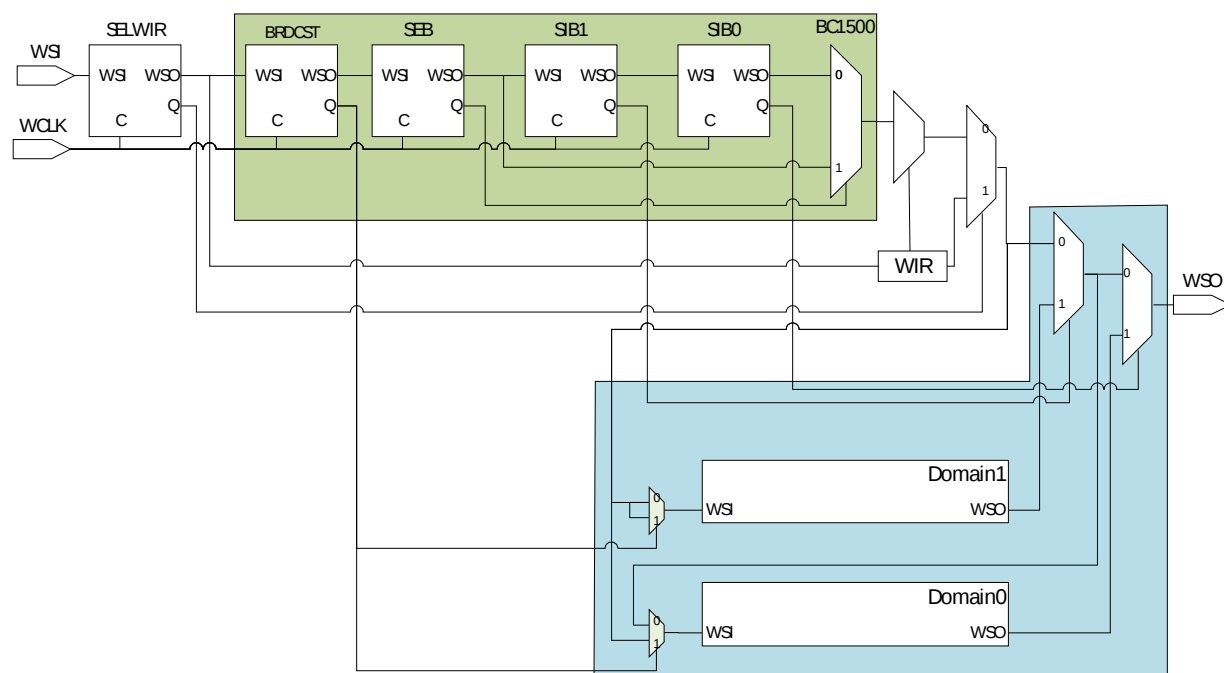
Illustration 11: DFXIP3 BC1500 Domain Broadcasting WSI/WSO Connectivity (VSBR)
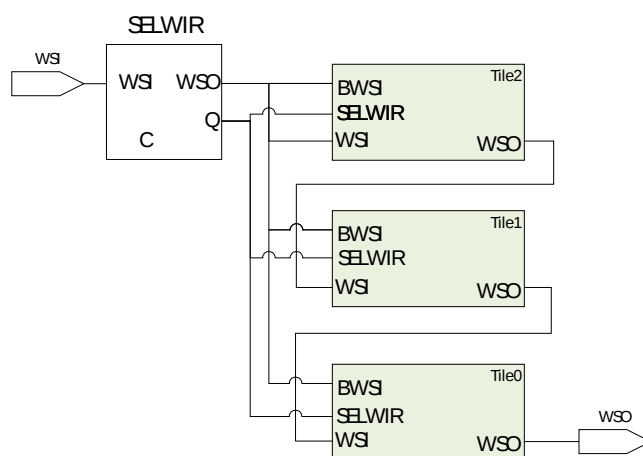
Illustration 12: DFXIP3 Domain Tile 1500 Broadcast WSI/WSO Connectivity
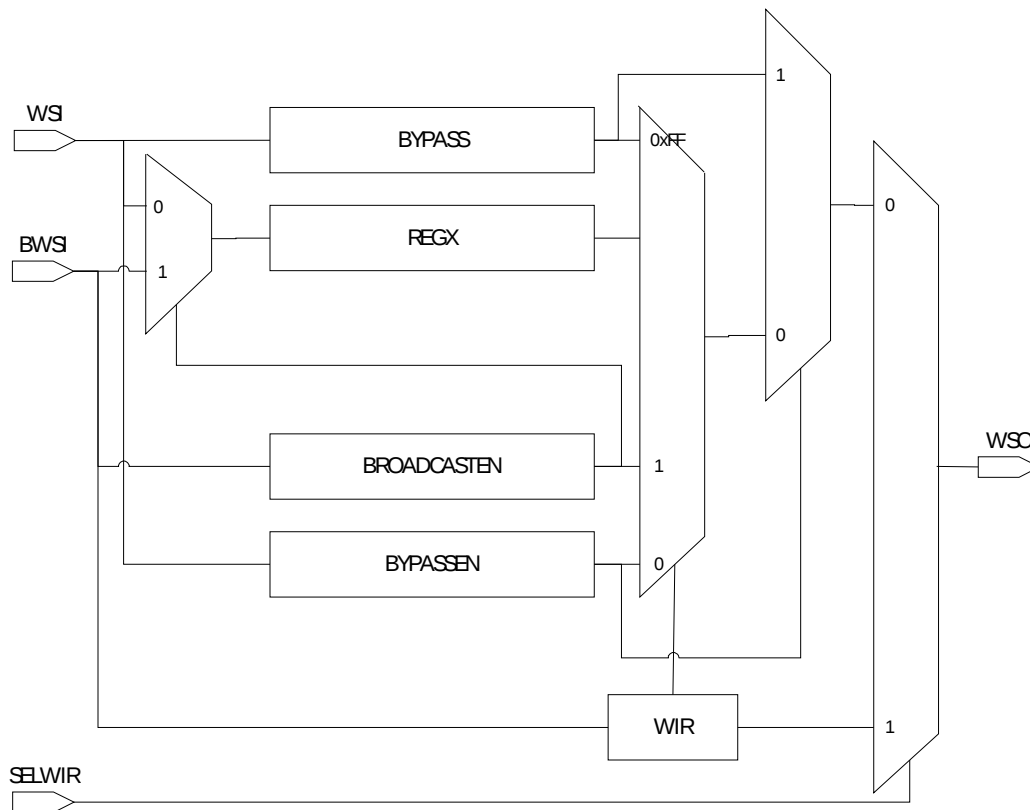
Illustration 13: DFXIP3 Tile 1500 Broadcast WSI/WSO Connectivity (VSMC)

21.3.1   StimGen Programming

Generating vectors for these networks requires a good bit of accounting. It can be done manually but is much easier to have StimGen handle the logistics of things. StimGen currently uses a "relatively" greedy algorithm for generating vectors. Most of the time this produces good vectors but there are times when there are other ways that vectors could have been constructed to produce a more optimal solution.

21.3.1.1   Items to be aware of

When generating vectors for register networks that include multiple levels of broadcasting there are several things going on that users need to be aware of.

21.3.1.1.1   Transition into and out of Broadcast mode

The "1500 Broadcast" network is always broadcasting to its Tile WIR registers. If the parent "STAC Broadcast" is in serial mode you will see two WIR and SELWIR bits and if its in broadcast mode you will see one WIR and SELWIR. Because the "1500 Broadcast" takes more operations to setup you

may see vectors generated that write "1500 Broadcast" tile WIR/SELWIR bits in serial mode but then the write the corresponding tile WDR networks in broadcast mode.

21.3.1.1.2 Use of XCastModule::operator[] and XCastModule::get_register

Where Xcast registers exist, StimGen use a class XcastModule which provides enhanced functionality for the operator[](const std::string &name) and get_register(const std::string &name) methods. In these modules these methods will search all of the Xcast register networks for registers with the specified name and returns a list of registers from all of the XCast register networks. This operator will take into account what networks are enabled and bypassed when building the list of registers.

In order for these methods to work correctly StimGen must use the state that has been applied to the network. Using the simulation state (load_value) may produce undesirable results since operations may be merged together when apply is issued. StimGen therefore requires that you issue an apply() prior to using these operations when you are changing the state of the control registers.

For STAC Broadcast this gets complicated by the fact that SIBs are also being used to determine the correct state. BOZO: Need to talk with Sonia to understand what the POR is here.

21.3.2 Corner Cases

Special care has to be taken to generate vectors for these networks. There are several situations which should be avoided.

9. When there are multiple levels of broadcasting all levels should either be broadcasting or in serial mode. If modes of operations are mixed results maybe undesirable. To control and ensure you correctly get into and out of broadcast modes its advised that a sequence be defined and used across the board to ensure consistency.

10. When STAC broadcasting to multiple "1500 Broadcast" networks which are not identical it may be possible that these networks have bypassen configured such that the network driving WSO is longer/shorter than the other "1500 Broadcast" networks which will result in invalid data being loaded into the registers fed by WSO. There should be no other register networks after an Xcast network enabled when broadcasting to avoid this. It is up to the user to correctly author patterns to avoid this. BOZO we need a diagram and more clarification here

11. "1500 Broadcast" networks are not guaranteed to be identical. StimGen will always check register widths during broadcast operations to ensure register widths match (StimGen will account for BYPASS bit insertion) and print an error message but will not terminate. It is up to the user to broadcast write to registers with the same definition and catch this error.

21.3.3 Broadcast Recipes

The following recipes are written assuming the register network from the previous illustrations. The StimGen sequence will be defined. An initial state of the registers will be documented for the register operations to be built against. This state may or may not be consistent with your environment or patterns and should be considered random. With the sequence and initial state defined a table of operations per 'apply()' will be defined. The order which they appear in the table will be consistent

with how StimGen will queue them up. The order which they appear is not necessarily how they will get merged. The table will list out what operations can be merged together for the various top level operations that are generated, these will be color coded. An asterisk(*) will be used to identify operation that are merged as a result of broadcast operations. Finally each sequence will include the list of registers currently connected between WSI and WSO and the shift value through that configuration.

21.3.3.1   Enable STAC and 1500 Broadcast

21.3.3.1.1   Sequence

gnb["BRDCST"] (1).write();

gnb["BROADCASTEN"] (1).write().apply();

21.3.3.1.1.1   Assumed Initial Register State

Register	Current State
SELWIR	1
BC1500[BRDCST]	0
BC1500[SEB]	0
BC1500[SIB1]	0
BC1500[SIB0]	0
WIR	0xFF
Domain[1:0].SELWIR	1
Domain[1:0].BYPASS	0
Domain[1:0].Tile[2:0].REGX	X
Domain[1:0].Tile[2:0].BROADCASTEN	0
Domain[1:0].Tile[2:0].BYPASSEN	0
Domain[1:0].Tile[2:0].WIR	0xFF

21.3.3.1.1.2   Event Queue & Event Merging

ID	Execution Order	Operation	Description
1	1	WIR=BC1500	Select BC1500 register
2	1	SELWIR=0	Select BC1500 register
3	2	BC1500[BRDCST]=1	Enable top level broadcast

ID	Execution Order	Operation	Description
4	2	BC1500[SIB1]=1	Open Domain1 for operations
5	3	Domain[1].Tile[2:0].WIR=1	Select BROADCASTEN register
6	3	Domain[1].SELWIR=0	Select WDR
7	4	Domain[1].Tile[2:0].BROADCASTEN=1	Enable broadcast
8	2	BC1500[SIB0]=1	Open Domain 0 for operations
9	3*	Domain[0].Tile[2:0].WIR=1	Select BROADCASTEN register
10	3*	Domain[0].SELWIR=0	Select WDR
11	4*	Domain[0].Tile[2:0].BROADCASTEN=1	Enable broadcast
12	4	Domain[0].SELWIR=1	Restore SELWIR
13	4*	Domain[1].SELWIR=1	Restore SELWIR
14	4	BC1500[SIB0]=0	Restore SIB
15	4	BC1500[SIB1]=0	Restore SIB
16	4	SELWIR=1	Restore SELWIR

21.3.3.1.1.3 Events

8. WSI → SELWIR → WIR → WSO (9'b0_00000001)

9. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → WSO (5'b0_1011)

10. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].WIR → WSO (14'b0_1011_0_00000001)

11. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].BROADCASTEN → WSO (7'b1_1000_1_1)

21.3.3.2 *Disable Broadcast STAC and 1500 and Enter Serial mode*

21.3.3.2.1 Sequence (Option 1)

gnb["BROADCASTEN"] (0).write().apply();

gnb["BRDCST"] (0).write().apply();

21.3.3.2.1.1 Assumed Initial Register State

Register	Current State
SELWIR	1
BC1500[BRDCST]	1

Register	Current State
BC1500[SEB]	0
BC1500[SIB1]	1
BC1500[SIB0]	1
WIR	X
Domain[1:0].SELWIR	1
Domain[1:0].BYPASS	0
Domain[1:0].Tile[2:0].REGX	X
Domain[1:0].Tile[2:0].BROADCASTEN	1
Domain[1:0].Tile[2:0].BYPASSEN	0
Domain[1:0].Tile[2:0].WIR	X

21.3.3.2.1.2   Event Queue & Event Merging

Table 6: gnb["BROADCASTEN"] (0).write().apply();

ID	Execution Order	Operation	Description
1	1	WIR=BC1500	Select BC1500 register
2	1	SELWIR=0	Select BC1500 register
3	2	Domain[1].Tile[2:0].WIR=1	Select BROADCASTEN register
4	2	Domain[1].SELWIR=0	Select WDR
5	3	Domain[1].Tile[2:0].BROADCASTEN=0	Enable broadcast
6	2*	Domain[0].Tile[2:0].WIR=1	Select BROADCASTEN register
7	2*	Domain[0].SELWIR=0	Select WDR
8	3*	Domain[0].Tile[2:0].BROADCASTEN=0	Enable broadcast
9	3	Domain[0].SELWIR=1	Restore SELWIR
10	3	Domain[1].SELWIR=1	Restore SELWIR
11	3	BC1500[SIB0]=0	Restore SIB
12	3	BC1500[SIB1]=0	Restore SIB
13	3	SELWIR=1	Restore SELWIR

21.3.3.2.1.3   Events

1. WSI → SELWIR → WIR → WSO (9'b0_00000001)

2. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].WIR → WSO (14'b0_1011_0_00000001)

3. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].BROADCASTEN → WSO (7'b1_1000_1_0)

Table 7: gnb["BRDCST"] (0).write().apply();

ID	Execution Order	Operation	Description
1	1	SELWIR=0	Select BC1500 register
2	2	BC1500[BRDCST]=0	Disable top level broadcast
3	2	SELWIR=1	Restore SELWIR

21.3.3.2.1.4   Events

1. WSI → SELWIR → WIR → WSO (9'b0_00000001)

2. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → WSO (5'b1_0000)

21.3.3.2.2   Sequence (Option 2)

gnb["BROADCASTEN"] (0).write(); // No apply

gnb["BRDCST"] (0).write().apply();

21.3.3.2.2.1   Register State

Register	Current State
SELWIR	1
BC1500[BRDCST]	1
BC1500[SEB]	0
BC1500[SIB1]	1
BC1500[SIB0]	1
WIR	X
Domain[1:0].SELWIR	1
Domain[1:0].BYPASS	0
Domain[1:0].Tile[2:0].REGX	X
Domain[1:0].Tile[2:0].BROADCASTEN	1
Domain[1:0].Tile[2:0].BYPASSEN	0

Register	Current State
Domain[1:0].Tile[2:0].WIR	X

21.3.3.2.2.2 Event Queue & Event Merging

Table 8: First apply();

ID	Execution Order	Operation	Description
1	1	WIR=BC1500	Select BC1500 register
2	1	SELWIR=0	Select BC1500 register
3	2	Domain[1].Tile[2:0].WIR=1	Select BROADCASTEN register
4	2	Domain[1].SELWIR=0	Select WDR
5	3	Domain[1].Tile[2:0].BROADCASTEN=0	Enable broadcast
6	2*	Domain[0].Tile[2:0].WIR=1	Select BROADCASTEN register
7	2*	Domain[0].SELWIR=0	Select WDR
8	3*	Domain[0].Tile[2:0].BROADCASTEN=0	Enable broadcast
9	3	BC1500[BRDCST]=0	Disable top level broadcast
10	3	Domain[0].SELWIR=1	Restore SELWIR
11	3*	Domain[1].SELWIR=1	Restore SELWIR
12	3	BC1500[SIB0]=0	Restore SIB
13	3	BC1500[SIB1]=0	Restore SIB
14	3	SELWIR=1	Restore SELWIR

21.3.3.2.2.3 Events

1. WSI → SELWIR → WIR → WSO (9'b0_00000001)

2. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].WIR → WSO (14'b0_1011_0_00000001)

3. WSI → SELWIR → BC1500[BRDCST, SEB, SIB1, SIB0] → Domain[0].SELWIR → Domain[0].Tile[2:0].BROADCASTEN → WSO (7'b1_0000_1_0)

21.4 STAC Routers

22 ConfigManager

The ConfigManager provides a single interface for all SG4 run time configuration information and SG4 configuration switches. The ConfigManager also provides a facility for retrieving and storing command line arguments should the main method make the necessary calls. The ConfigManager is implemented as a singleton so that there is one instance.

The SG4 configuration switches are used to enable certain types of functionality or turn on/off debug information. These switches can be directly accessed in patterns or loaded from a SGXML run time configuration file. The run time configuration information provides standard setup information for the DUT and visitor patterns and can be loaded from a SGXML run time configuration file.

To simplify access to the configuration manager the following macro has been created

> #define SGCM StimGen::ConfigManager::Instance()

and will allow you to access the configuration manager via SGCM. The sections that follow will provide more details on specific usage.

22.1 SG4 Run Time Configuration Information

The SG4 run time configuration information is a data structure to store configuration information for the DUT and visitor patterns. It is up to the user to use this information correctly. All data is imported from a SGXML file with run time configuration information. To load the information issue the command

> SGCM.import("PATH_TO_SGXML");

The SGXML file that is loaded must be compliant to the SGXML schema. Use the following command to validate the SGXML (NOTE: this is the same command to validate all other SGXML)

> xmllint \
>
> > --schema $SG4COREBASE/../sigtools/schema/SGXML.xsd \
> > --noout \
> > --xinclude \
> > PATH_TO_SGXML

If you attempt to load non-compliant SGXML, data may not load and SG4 might segfault.

22.1.1 SGXML Run Time Configuration File

The easiest way to describe the run time configuration file is to provide a sample and explain.

```
52. <?xml version="1.0" encoding="UTF-8"?>
53. <runtime_configuration xmlns="http://mpdwww.amd.com/sigdata"
54.  xmlns:xs="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xi="http://www.w3.org/2001/XInclude"
55.  xs:schemaLocation="http://mpdwww.amd.com/sigdata
    http://mpdwww.amd.com/sigdata/SGXML.xsd">
56.
57.  <!-- ConfigManager SG4 configuration Switches -->
```

```
58.    <options>
59.     <enable_reset_network_control_on_apply>false</enable_reset_network_control_on_apply>
60.     <report_reset_registers>true</report_reset_registers>
61.     <report_queue_events>false</report_queue_events>
62.     <enable_read_auto_broadcast_to_serial>true</enable_read_auto_broadcast_to_serial>
63.     <enable_read_auto_serial_to_broadcast_restore>false</enable_read_auto_serial_to_broadca
    st_restore>
64.     <check_broadcast_load_value_consistency>true</check_broadcast_load_value_consistency>
65.     <use_verilog_four_state_compare>false</use_verilog_four_state_compare>
66.    </options>
67.
68.    <!-- Generic Parameters for user use -->
69.    <parameters>
70.     <parameter>
71.      <key>Barney</key>
72.      <value>Rubble</value>
73.     </parameter>
74.     <parameter>
75.      <key>Fred</key>
76.      <value>Flinstone</value>
77.     </parameter>
78.    </parameters>
79.
80.    <event_visitors>
81.    <!-- Add hook for generic user visitor patterns -->
82.    <generic type="YAPP">
83.     <configuration name="default">
84.      <parameters>
85.       <parameter>
86.        <key>key</key>
87.        <value>value</value>
88.       </parameter>
89.      </parameters>
90.     </configuration>
91.    </generic>
92.
93.    <!-- STIL specific configuration information -->
94.    <stil_credence>
95.     <!-- Default Configuration -->
96.     <configuration name="default">
97.      <parameters>
98.       <parameter>
99.        <key>key</key>
100.        <value>value</value>
101.       </parameter>
102.      <!-- parameters -->
103.      </parameters>
104.     <compress_vectors>true</compress_vectors>
105.     <include_path>/proj/sigdata/15h/CZ/pins/stil_inc/bump</include_path>
106.     <includes>
107.        /a/b/c/d/e /a/d/e/f/g
108.        /a/f/g/h/i
109.     </includes>
110.     <keywords> KW0 KW1
111.             KW2
112.             KW3
113.     </keywords>
114.     <output_path>/proj/cz_debug/SG4/user/sfoster/sfCZworkdir/STIL</output_path>
115.     <package_name>CZ_PEO</package_name>
116.     <signal_group_include>15h_CZ_PEO_SIGGRP.inc</signal_group_include>
117.     <signal_include>15h_CZ_PEO_SIG.inc</signal_include>
```

```
118.      <timing_name>timing_nstm</timing_name>
119.      <waveform_table>func_200MHz</waveform_table>
120.      <wft_include>15h_CZ_PEO_WFT.inc</wft_include>
121.    </configuration>
122.
123.    <!-- A second configuration variant -->
124.    <configuration name="Package_FM2R">
125.      <compress_vectors>true</compress_vectors>
126.      <include_path>/proj/sigdata/15h/CZ/pins/stil_inc/bump</include_path>
127.      <output_path>/proj/cz_debug/SG4/user/sfoster/sfCZworkdir/STIL</output_path>
128.      <package_name>FM2R</package_name>
129.      <signal_group_include>15h_CZ_PEO_SIGGRP.inc</signal_group_include>
130.      <signal_include>15h_CZ_PEO_SIG.inc</signal_include>
131.      <timing_name>timing_nstm</timing_name>
132.      <waveform_table>func_200MHz</waveform_table>
133.      <wft_include>15h_CZ_PEO_WFT.inc</wft_include>
134.    </configuration>
135.   </stil_credence>
136.  </event_visitors>
137.
138.  <!-- DUT Configuration -->
139.
140.  <!-- Define initial pin states -->
141.  <pin_init_definitions>
142.   <pin_init_definition name="cold_reset">
143.    <pin name="BP_PWROK" value="0" />
144.    <pin name="BP_RESET_L" value="0" />
145.   </pin_init_definition>
146.   <pin_init_definition name="warm_reset" input="1"
147.    output="Z" inout="0" supply1="0" supply0="1">
148.    <pin name="BP_PWROK" value="1" />
149.    <pin name="BP_RESET_L" value="0" />
150.   </pin_init_definition>
151.   <pin_init_definition name="running" input="0"
152.    output="X" inout="Z" supply1="1" supply0="0">
153.    <pin name="BP_PWROK" value="1" />
154.    <pin name="BP_RESET_L" value="1" />
155.   </pin_init_definition>
156.  </pin_init_definitions>
157.
158.  <!-- Define multi-cycle clocks -->
159.  <clock_definitions>
160.   <clock_definition name="default">
161.    <clock name="BP_X48M_X1" vectors_per_cycle="2" clock_up_vector="0"
162.     clock_down_vector="1" />      <!-- 50MHz XTAL clock -->
163.    <clock name="BP_X48M_X2" vectors_per_cycle="2" clock_up_vector="1"
164.     clock_down_vector="0" />      <!-- 50MHz XTAL clock -->
165.    <clock name="BP_CLKIN_H" vectors_per_cycle="1" clock_up_vector="0"
166.     clock_down_vector="1" />       <!-- 100MHz refclk -->
167.    <clock name="BP_CLKIN_L" vectors_per_cycle="1" clock_up_vector="1"
168.     clock_down_vector="0" />       <!-- 100MHz refclk -->
169.   </clock_definition>
170.
171.   <clock_definition name="Orochi_default">
172.    <clock name="BP_BYPASSCLK_B_H" vectors_per_cycle="2"
173.     clock_up_vector="0" clock_down_vector="1" />     <!-- 50MHz XTAL clock -->
174.    <clock name="BP_BYPASSCLK_T_H" vectors_per_cycle="2"
175.     clock_up_vector="0" clock_down_vector="1" />     <!-- 50MHz XTAL clock -->
176.    <clock name="BP_BYPASSCLK_B_L" vectors_per_cycle="2"
177.     clock_up_vector="1" clock_down_vector="0" />     <!-- 50MHz XTAL clock -->
178.    <clock name="BP_BYPASSCLK_T_L" vectors_per_cycle="2"
```

```
179.        clock_up_vector="1" clock_down_vector="0" />    <!-- 50MHz XTAL clock -->
180.    </clock_definition>
181.  </clock_definitions>
182.
183.  <!-- Define Protocol Clock and Measure cycles defintions -->
184.  <protocol_definitions>
185.
186.    <!-- JTAG cycle definitions -->
187.    <IEEE1149_1>
188.     <cycle_definition name="default" vectors_per_cycle="4"
189.      clock_up_vector="1" clock_down_vector="3" measure_vector="3" />
190.     <cycle_definition name="tms" vectors_per_cycle="4"
191.      clock_up_vector="1" clock_down_vector="3" measure_vector="3" />
192.     <cycle_definition name="shift" vectors_per_cycle="4"
193.      clock_up_vector="1" clock_down_vector="3" measure_vector="3" />
194.     <cycle_definition name="sms" vectors_per_cycle="24"
195.      clock_up_vector="10" clock_down_vector="20" measure_vector="21" />
196.    </IEEE1149_1>
197.
198.    <!-- MDIO cycle definitions -->
199.    <IEEE802_3>
200.     <cycle_definition name="default" vectors_per_cycle="4"
201.      clock_up_vector="1" clock_down_vector="3" measure_vector="3" />
202.     <cycle_definition name="MDIO_B" vectors_per_cycle="8"
203.      clock_up_vector="2" clock_down_vector="5" measure_vector="5" />
204.    </IEEE802_3>
205.  </protocol_definitions>
206.
207. </runtime_configuration>
```

Lines 7-15:   Setup SG4 Configuration Switches

Lines 17-27:   User defined parameters. These can be retrieved using
                       bool SGCM.has_parameter (const std::string &parameter_name);
                       std::string SGCM.get_parameter (const std::string &parameter_name);

Lines 29-85:   SG4 visitor pattern configuration information. Every visitor pattern can insert setup
                information into the SGXML. Some visitor patterns, such as stil_credence, have
                extra checking in place while others are limited to generic structures. Each visitor
                pattern or the instantiating environment is responsible for loading visitor pattern
                information. Review visitor pattern naturalDocs documentation for specific details.
                In general all visitor patterns provide hooks so that you should be able to use this
                command to load information
                       SGCM.load_visitor_pattern_configuration (Event_Visitor* vp,
                                  const std::string configuration_name);
                where:
                       vp is a pointer to the visitor pattern that will load configuration information.
                       configuration_name is the name of the configuration information to load
                defined in the SGXML. It is up to the visitor pattern to know which type of
                information to load from the SGXML if it uses generic information.

                Every visitor pattern can have multiple sets of named configuration information.
                Each set must provide all information and not rely on loading multiple

configurations.

Lines 89-105:   Define a variety of initial pin state configurations each with a unique name. Each
pin_init_definition provides default values to assign to pins based on pin type: input,
output, inout, supply0, supply1. These values will be applied to all pins that aren't
explicitly called out to be assigned some other value.

To apply this a pin state issue the following command
       SGCM.load_int_definition(Module* dut, const std::string &pin_init_name);
where:
       dut, is the toplevel module where pins are defined.
       pin_init_name, is the pin configuration name from the SGXML.

Lines 108-130:   Define a variety of initial clock configurations each with a unique name. Each
configuration defines clock period and edges where the clock is multiple time units.
Only one clock definition can be loaded at a time. When a clock definition is loaded
previously added clocks will be removed. To load clocks issue the following
command
       SGCM.load_clocks_definition (Module* dut,
                                         const std::string &clocks_config_name);
where:
       dut, is the toplevel module where clocks pins are defined.
       clocks_config_name, is the clocks definition name from the SGXML

Lines 133-154:   Define a protocol clock & measure definitions for each type of protocol supported.
Each protocol can have any number of cycle definitions each with its own name. In
some cases a protocol may support multiple cycle definitions, e.g. JTAG, and you
must specify which definition to load into what cycle definition.   See naturalDocs
information for the specific details on loading cycle definitions. In general one of the
following two commands are used
       // When multiple cycles types are available
       SGCM.load_protocol_cycle_definition (
              Protocol* protocol,
              <CYCLE_TYPE> type,
              const std::string &cycle_definition_name
       );

       // or when there is only a single cycle definition
       SGCM.load_protocol_cycle_definition (
              Protocol* protocol,
              const std::string &cycle_definition_name
       );
where:
       protocol is a pointer to the protocol object
       type is the cycle type to load the definition into
       cycle_definition_name is the name specified in the SGXML

22.2   SG4 Configuration Switches

To set the configManager's variables issue commands of the form (substituting for *VARIABLE* and *boolean* as needed)

> SGCM.*VARIABLE* = *boolean*;

If you need to query or use the variable you can access the variable with commands of the form (substituting for *VARIABLE* as needed)

> boolean my_var = SGCM.*VARIABLE*;

22.2.1   ConfigManager Functionality Variables

22.2.1.1   General Functionality Variables

22.2.1.1.1   enable_reset_network_control_on_apply

Enable StimGen to automatically restore "control" registers back to their default state after all events within the apply are processed if and only if the control register was programmed within the current apply. If the control register was not modified within the current apply the pattern must manually set its value to re-enable auto reset.

22.2.1.1.2   use_verilog_four_state_compare

By default StimGen::Number treats 'X' as a don't care for 2-state comparisons(0 or 1). This doesn't fit all use models especially those in verification. There are use models where values need to be compared using verilog 4-state comparisons (0, 1, X, Z). When enabled, this option cause StimGen to perform 4-state compares of number vs the default 2-state.

22.2.1.2   Broadcast Specific Variables

22.2.1.2.1   enable_read_auto_broadcast_to_serial

BOZO is this used?

22.2.1.2.2   enable_read_auto_serial_to_broadcast_restore

BOZO is this used?

22.2.2   ConfigManager Debug Variables

22.2.2.1   report_queued_events

This option will report all of the events that are queued and remaining the queue prior to producing the

next set of protocol operations. This report is produced stimGen is merging events for the next protocol operation. The first event in the queue is always processed first and then other events are merged with it. This report will appear before the event is reported. Here is an example

```
0  Top.IPGPU0.GNB_STAC_SELWIR  1'h0
1  Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SIB  1'h1
2  Top.IPGPU0.usblk_gck.WIR  8'h02
3  Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SELWIR  1'h0
4  Top.IPGPU0.usblk_gck.TESTCTRL  22'h066666
5  Top.IPGPU0.usblk_dft.TESTCTRL  22'h066666
```

22.2.2.2 report_reset_registers

The report_reset_registers will report to you what registers that control virtual registers are automatically getting reset to their default state when apply() is complete.

```
Event::IEEE1149_1_DR  Top.tdr  load=24'h002087
meas=24'bXXXXXXXXXXXXXXXXXXXXXXXX
Resetting control network Top.IPGPU0.SEL_SUN_SMU_TILES_P1500_SIB load_value
back to 1'h0 currently=1'h1
Resetting control network Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SIB load_value
back to 1'h0 currently=1'h1
Resetting control network Top.IPGPU0.SEL_DFP_VDDGFX_TILES_P1500_SIB
load_value back to 1'h0 currently=1'h1
Event::IEEE1149_1_DR  Top.tdr  load=52'h0020018000000
meas=52'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Use this option when StimGen is not providing you with the desired patterns and you are working with 1500, 1687 or STAC type networks. This may help identify the source of the problem. BOZO include see section for enabling and disabling automatic register reset per register.

22.2.2.3 check_broadcast_load_value_consistency

Working with broadcast networks adds a new level of complexity. By default this option is enabled. It will cause stimGen to throw an exception when a broadcasting broadcast register network has been incorrectly conditioned to write conflicting data to the target registers. This typically occurs when the network should be in serial mode to write unique data to the target registers but is currently configured for broadcasting.

22.3 Disable Register Auto Reset on Apply

StimGen will by default schedule all registers that control a virtual register network to be restored to its documented default state on apply. Users have the ability to selectively disable and enable this restore on apply. Doing this in code is dangerous and introduces variability across users and environments. The runtime configuration file allows projects to explicitly call out all registers that should not be reset on an apply. Use the following example SGXML content to set the list.

```
<disable_register_auto_resets>
  <register>Top.ATECONFIG</register>
  <register>Top.Functional</register>
  <register>Top.IPCONFIG</register>
```

```
        <register>Top.IR</register>
        <register>Top.SCANCONFIG</register>
        <register>Top.HT3.IR</register>
        <register>Top.HT2.IR</register>
        <register>Top.HT1.IR</register>
        <register>Top.HT0.IR</register>
        <register>Top.FuseBox.CHAIN_CONFIG</register>
        <register>Top.D0.IR</register>
        <register>Top.CPU3.IR</register>
        <register>Top.CPU3.SCANCONFIG</register>
        <register>Top.CPU2.IR</register>
        <register>Top.CPU2.SCANCONFIG</register>
        <register>Top.CPU1.IR</register>
        <register>Top.CPU1.SCANCONFIG</register>
        <register>Top.CPU0.IR</register>
        <register>Top.CPU0.SCANCONFIG</register>
    </disable_register_auto_resets>
```

22.4　　SG4 ConfigManager Command Line Options

There are many use cases for SG4 where a common test pattern needs to generate a variety of stimulus based on some set of arguments. These use cases are common when static files, such as STIL or ATPG initialization sequences. To cover this use case, the ConfigManager provides a facility to extract, store and retrieve specific command line arguments. These command line arguments will persist throughout execution.

To parse command line options the following SGCM.load_command_line_arguments should be executed from the main method

```
1.      if ( ! SGCM.load_command_line_arguments( &argc, argv ) ) {
2.        std::cout << "Error loading StimGen command line arguments.  Aborting!";
3.        return 5;
4.      }
```

The load_command_line_arguments takes a pointer to the number of arguments in on the command line and that list of arguments. The command will look for all command line arguments of the form:

<div align="center">--sg_KEY=VALUE</div>

KEY may be any identifier the user would like to pass in and VALUE is the value to associate with that key. All arguments found in *argv* that take this form will be removed from argv and stored in the ConfigManager. When the argument is removed from *argv*, *argc* will be decremented. All unrecognized arguments will be ignored and left in *argv* for the main method to handle.

Two methods are provided to access these command line arguments:

- bool SGCM.has_command_line_argument (const std::string&)

- const std::string& SGCM.get_command_line_argument (const std::string &)

Both methods take a single argument which is the desired KEY to locate from the command line. The has method checks if the KEY command line argument exists and the get method returns the associated value. If the KEY is not specified on the command line, get will throw an exception so you should

always use has and get methods together.

23   Debugging

StimGen contains different internal mechanisms to assist with debugging test patterns and stimGen operation. There is a debug visitor pattern which can be added to the DUT to simply report generated events to STDOUT. There are also global switches for controlling both how events are processed and dumping of debug information that are available through a singleton ConfigManager.

23.1   Debug Visitor Pattern

The debug visitor pattern is available in sg_Event.h. If added to the DUT it will write to STDOUT all of the events being processed. Recall that StimGen supports a variable number of visitor patterns so the debug visitor pattern can be added in addition to the primary visitor patterns.

The default report will contain information similar to

```
    Event::IEEE1149_1_DR  Top.tdr  load=16'h0020  meas=16'bXXXXXXXXXXXXXXXX
```

Verbosity can be increased by turning on bitfield reporting by calling method Event_Visitor_Debug::set_report_field(bool) and passing in a true value. When enabled the bitfields will be included in the output and will look similar to

```
    Event::IEEE1149_1_DR  Top.tdr  load=16'h0020  meas=16'bXXXXXXXXXXXXXXXX
      Register Bitfields:
    TDI -->
      15:15 Top.SOC_P1687A_SEB.Q = 1'h0    1'h0
      14:14 Top.DUMMY14_STAC_SIB.Q = 1'h0   1'h0
      13:13 Top.DUMMY13_STAC_SIB.Q = 1'h0   1'h0
      12:12 Top.DUMMY12_STAC_SIB.Q = 1'h0   1'h0
      11:11 Top.DUMMY11_STAC_SIB.Q = 1'h0   1'h0
      10:10 Top.DUMMY10_STAC_SIB.Q = 1'h0   1'h0
      9:9 Top.DUMMY9_STAC_SIB.Q = 1'h0   1'h0
      8:8 Top.DUMMY8_STAC_SIB.Q = 1'h0   1'h0
      7:7 Top.DUMMY7_STAC_SIB.Q = 1'h0   1'h0
      6:6 Top.GPA_STAC_SIB.Q = 1'h0   1'h0
      5:5 Top.GNB_STAC_SIB.Q = 1'h1   1'h0
      4:4 Top.PCIE0_STAC_SIB.Q = 1'h0   1'h0
      3:3 Top.PCIE1_STAC_SIB.Q = 1'h0   1'h0
      2:2 Top.PCIE2_STAC_SIB.Q = 1'h0   1'h0
      1:1 Top.PCIE3_STAC_SIB.Q = 1'h0   1'h0
      0:0 Top.PCIE4_STAC_SIB.Q = 1'h0   1'h0
    --> TDO
```

23.1.1   Using the Debug Visitor Pattern

To use the debug visitor pattern create an instance of it where you create your DUT and add it using Module::add_event_visitor(Event_Visitor&)

```
    StimGen::ModuleFactory factory ( "sgxml", "." );
    Module *dut = factory.import( "orochi_A0.sgxml", "" );
```

```
Event_Visitor_Debug debug_visitor;

// Turn on reporting of individual bitfield information
debug_visitor.set_report_field( true );
dut->add_event_visitor( debug_visitor );
```

23.2   Debug Switches

SG4 contains various switches to assist with debug. These switches are all accessible through singleton StimGen::ConfigManager. To get the instance of the singleton include "sg_ConfigManager.h" and then add to your code:

StimGen::ConfigManager::Instance().*<VARIABLE_NAME>* = *<boolean>*;

23.2.1   ConfigManager Debug Variables

23.2.1.1   report_queued_events

This option will report all of the events that are queued and remaining the queue prior to producing the next set of protocol operations. This report is produced stimGen is merging events for the next protocol operation. The first event in the queue is always processed first and then other events are merged with it. This report will appear before the event is reported. Here is an example

```
0  Top.IPGPU0.GNB_STAC_SELWIR  1'h0
1  Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SIB  1'h1
2  Top.IPGPU0.usblk_gck.WIR  8'h02
3  Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SELWIR  1'h0
4  Top.IPGPU0.usblk_gck.TESTCTRL  22'h066666
5  Top.IPGPU0.usblk_dft.TESTCTRL  22'h066666
```

23.2.1.2   report_reset_registers

The report_reset_registers will report to you what registers that control virtual registers are automatically getting reset to their default state when apply() is complete.

```
Event::IEEE1149_1_DR  Top.tdr  load=24'h002087
meas=24'bXXXXXXXXXXXXXXXXXXXXXXXX
Resetting control network Top.IPGPU0.SEL_SUN_SMU_TILES_P1500_SIB load_value
back to 1'h0 currently=1'h1
Resetting control network Top.IPGPU0.SEL_DFP_VDDNB_TILES_P1500_SIB load_value
back to 1'h0 currently=1'h1
Resetting control network Top.IPGPU0.SEL_DFP_VDDGFX_TILES_P1500_SIB
load_value back to 1'h0 currently=1'h1
Event::IEEE1149_1_DR  Top.tdr  load=52'h0020018000000
meas=52'bXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXXX
```

Use this option when StimGen is not providing you with the desired patterns and you are working with 1500, 1687 or STAC type networks. This may help identify the source of the problem. BOZO include see section for enabling and disabling automatic register reset per register.

23.2.1.3   *check_broadcast_load_value_consistency*

Working with broadcast networks adds a new level of complexity. By default this option is enabled. It will cause stimGen to throw an exception when a broadcasting broadcast register network has been incorrectly conditioned to write conflicting data to the target registers. This typically occurs when the network should be in serial mode to write unique data to the target registers but is currently configured for broadcasting.

23.3   *Run-Time Logging*

StimGen4 includes a run-time logging system which can be useful for debugging. See `sg_Logger.h` for implementation details; note that `sg_Logger.h` is included by `StimGen.h`.

23.3.1.1   *Basic Usage*

Two macros are provided which evaluate to a `std::ostream` object. Log messages are sent to this object using the stream insertion operator. A simple log message looks like this:

```
SG_LOG(SG_LOG_INFO) << "Log message here" << std::endl;
```

A "component-filtered" log message (see below for details) looks like this:

```
SG_LOG_COMPONENT(SG_LOG_INFO, "module", 3) << "Log msg here" << std::endl;
```
With either of these calls, any object that supports the stream-insertion operator may be inserted into the logging call.

23.3.1.2   *Setting Log Output*

The log output can be sent to any object derived from `std::ostream,` or to a file. By default, output is sent to `std::cerr.` The following calls can be used to specify the destination for log messages:

```
SG_LOGGER.set_log_output(std::cout);          // Send to an ostream
SG_LOGGER.set_log_output("/tmp/stimgen.log");  // Send to a file
```

23.3.1.3   *Compile-Time Log Level Filtering*

Log messages go through several levels of filtering. The first level is compile-time filtering, which allows compilers to completely optimize out any overhead associated with the filtered statements. To set the maximum compile-time log level, include an argument in the compiler command line such as the following (GCC syntax):

```
-DSG_COMPILE_TIME_MAX_LOG_LEVEL=SG_LOG_INFO
```
Alternatively, you could edit the definition in `sg_Logger.h.`

The log levels are as follows, in increasing levels of verbosity:

```
SG_LOG_ERROR
SG_LOG_WARNING
SG_LOG_INFO
SG_LOG_VERBOSE
SG_LOG_DEBUG
```

Specifying a maximum log level will include messages at all levels up to and including the specified level.

23.3.1.4   Run-Time Log Level Filtering

The next level of filtering is run-time level filtering. This is set using a call such as:

```
SG_LOGGER.set_log_level(SG_LOG_VERBOSE);
```

Any log message sent at levels above the run-time level will be ignored. Unlike the compile-time level filtering, this filtering requires some run-time checks to be performed.

23.3.1.5   Component Filtering

The third level of filtering is component filtering, and applies only to the messages sent using the `SG_LOG_COMPONENT` macro. This macro allows you to specify components (arbitrary string) and flags (integer, 0-31) to further categorize your log messages. The component and flag are provided as the second and third arguments, respectively, to the `SG_LOG_COMPONENT` macro. Only enabled component/flag combinations are included in the log output. This filtering occurs at the time that log messages are generated, so the component/flag must be enabled prior to the execution of the logging call. Component flags can be turned on or off using the following calls:

```
SG_LOGGER.set_component_flag("core", 0); // Turn on for core/0
SG_LOGGER.set_component_flags("hierarchy", 0xF); // Turn on hierarchy/0,1,2,3
SG_LOGGER.clr_component_flag("hierarchy", 2); // Turn off hierarchy/2
```

24 Custom Register Access Procedures

Most of the test registers participate in register networks that are directly accessible via IEEE1149.1 or IEEE1500 protocols which are supported in SG4 by default. There are other registers which do not fall into this category but are accessible by performing multiple IEEE1149.1 or IEEE1500 operations or attach to other protocols such as I2C. SG4 provides the necessary hooks to attach procedures to groups of registers or modules anywhere in the design hierarchy to make this connections.

In SG4 all register belong to a module and modules can contain instances of other modules with their own registers. Internal to each module, registers are grouped into RegisterBlocks which are grouped into RegisterMaps. Typically these blocks and maps organize the registers by access mechanism and as a result provide hooks to attach custom write and read procedures. Unique procedures can be attached to different instances of a Module (or its RegisterBlocks or RegisterMaps).

The RegisterBlock and RegisterMap objects will attach an Interface::RegisterAccess derived class. This base class defines several read and write methods to process an argument of type const StimGen::shared-ptr<Event_Register>&.

The const StimGen::shared_ptr<Event_Register>& argument provides the details about the target register and the value to read or write from it. Given the target register you can easily access to the SG4 model and access the design hierarchy and other registers in the design. With access to these objects you can access other registers and perform the necessary read and write register operations.

24.1 Sample Custom Register Access Procedure

In the GNB dft tile the DSM is instantiated and can be accessed using registers DSM_ADDR, DSM_CTRL and DSM_DATA. When a user writes to a DSM register, 'dsm_control_status' for example, a new Event_Register will be created and will contain a reference to the target register('dsm_control_status') and the corresponding write value. Here is a sample albeit lengthy write procedure.

```
1.  namespace StimGen {
2.  namespace fuse_details {
3.    /*****************************************************************************
4.     * Class:  fuse_details::FuseConfigRegAccessHandler
5.     *
6.     *  Implement custom read/write methods to access fuse configuration registers.
7.     *
8.     *  Map register operations into appropriate FUSE_Config and FUSE_Read
9.     *  operations.
10.    *
11.    *****************************************************************************/
12.     class STIMGEN4MODEL_API FuseConfigRegAccessHandler
13.       : public StimGen::Interface::RegisterAccess {
14.      public:
15.       FuseConfigRegAccessHandler( StimGen::Register& fuse_config,
16.                                   StimGen::Register& fuse_read )
17.        : StimGen::Interface::RegisterAccess( 0, 1 ),
18.          fuse_config(fuse_config),
19.          fuse_read(fuse_read)
20.       {}
21.
22.       void write( const StimGen::shared_ptr<Event_Register>& e ) {
23.         RegisterAccessOptions options( e->get_options() );
```

```
24.           options.add_child_event( e );
25.
26.       fuse_config(0)
27.                  ["ConfigSel"](Number::One)
28.                  ["OpCode"](Number::One)
29.                  ["Send"](Number::One)
30.                  ["WriteData"]( e->get_load_value() )
31.                  ["Addr"]( e->get_register()->get_offset() )
32.                  .write( options );
33.       }
34.
35.     void read ( const StimGen::shared_ptr<Event_Register>& e ) {
36.       fuse_config(0)
37.                  ["ConfigSel"](Number::One)
38.                  ["OpCode"](Number::Zero)
39.                  ["Send"](Number::One)
40.                  ["WriteData"]( e->get_load_value() )
41.                  ["Addr"]( e->get_register()->get_offset() )
42.                  .write();
43.
44.       RegisterAccessOptions options( e->get_options() );
45.       options.add_child_event( e, &fuse_read["ReadData"] );
46.
47.       static const Number sv_32h0("32'h0");
48.       fuse_read["ReadValid"]( Number::Zero, Number::One ) // Write 0, Measure 1
49.              ["ReadData"]( sv_32h0, e->get_measure_value() )
50.              .read( options );
51.       }
52.
53.     private:
54.      StimGen::Register &fuse_config;
55.      StimGen::Register &fuse_read;
56.
57.      // Methods required by Interface for generic usage which we are not using
58.      // make them private and do nothing
59.      void write( Register* /*reg*/,
60.                  const RegisterAccessOptions &/*options*/ ){}
61.      void read ( Register* /*reg*/,
62.                  const RegisterAccessOptions &/*options*/ ){}
63.      void write( const Number& /*offset*/,
64.                  const Number& /*data*/,
65.                  const RegisterAccessOptions &/*options*/ ){}
66.      void read ( const Number& /*offset*/,
67.                  const MeasureNumber& /*expect*/,
68.                  const RegisterAccessOptions &/*options*/ ){}
69.
70.    };
71.   }
72. } // end of namespace StimGen
```

Lines 12,13:	Derive a new class from Interface::RegisterAccess
Lines 15-20:	This interface provides a custom constructor to pass in registers that will be used for reading and writing target registers.
Line 22 & 35	Override and implement the write and read register access operations. When Interfaces are attached to a register block and or register map, these methods will be used. Each takes a single const StimGen::shared_ptr<Event_Register>& argument.
Line 23, 24,	Create a new copy of the incoming event argument options and attach to it the new event. This will allow SG4 to update the originating register with the write data when the new write event is executed. If the event is not added, you will not see the load_value moved into the register value.

Lines 26-32   Get the appropriate information from the source event and create a new register write operation as required by the IP. Note that the options are passed into the write method.

Lines 36-42, 48-50   When reading registers in this IP, it requires two register operations. The first sets up the address to read from. This is the write on line 42. Note that we do not pass along any options to this write. Options are only passed into the final read operation because it is the operation that provides the data that needs to be propagated back to the original register initiating the the read.

Lines 44   Create a new copy of the incoming event argument options.

Lines 45   Add to the options the original event and pass in a pointer to the register or bitfield to extract read data from that will then be used to set the read data of the register that initiated the read operation.

Lines 59-69   Interface::RegisterAccess provides several interfaces for reading and writing registers. This example illustrates how to attach it to a registerMap and registerBlock where it operates behind the scenes. These other interfaces can be setup so that you can use the interface directly to read and write register or addresses directly. These will be discussed at a later time.

24.2   Attaching an Access Procedure to a RegisterMap or RegisterBlock

Once you've defined the Interface::RegisterAccess it needs to be attached to a specific register block or map. The first item is to determine if the interface should be attached to the block or to the map. Items to consider:

- Attach to the RegisterBlock if the access procedure applies only to the registers in this block

- Attach to the RegisterMap if the access procedure applies to all RegisterBlock's and their Registers.

- When SG4 is looking for an access procedure it will look for a procedure in the order RegisterBlock, RegisterMap, Module

Sample code to register custom register access handler in the personalize method.

```cpp
void StimGen::fuse::personalize() {

  // Install register access procedures to the Configuration Registers if they
  // are loaded/available
  if ( ! config_reg_access_handler ) {
    if ( has_register_map( "Config" )
      && get_register_map( "Config" ).has_register_block( "Reg" )
    ) {
      config_reg_access_handler.reset( new
          StimGen::fuse_details::FuseConfigRegAccessHandler( *get_register("FUSE_Config"),
                                                            *get_register("FUSE_Read")
                                                           )
                                      );
      get_register_map( "Config" ).get_register_block( "Reg" )
                                  .set_register_access_handler( config_reg_access_handler );
    } else {
      SG_LOG ( SG_LOG_ERROR )
        << "Fuse module is missing configuration registers.  Access/support is limited."
```

```
                << std::endl;
        }
      }
    }
```

SG4 provides a utility method install_register_access_interface to register access handler to specific IP register map or blocks.

```
/**************************************************************************************
 * Function: install_register_access_interface
 *
 * Install the register access interface into all instances with the specified
 * register map and block name.  If the block name is empty the interface will
 * be installed into the map
 *
 * Parameters:
 *   register_access_interface - The register access interface to install.
 *   design                    - Top level or IP to start insertion from
 *   map_name                  - Register Map name.  If no block interface is installed into map
 *   block_name                - Register Block name within the map to add the interface to.
 *
 **************************************************************************************/
STIMGEN_API void install_register_access_interface(
  StimGen::shared_ptr<StimGen::Interface::RegisterAccess> register_access_interface,
  StimGen::Module* design,
  const std::string &map_name,
  const std::string &block_name = ""
);

/**************************************************************************************
 * Function: install_register_access_interface
 *
 * Install the register access interface into instances of a specific type with the specified
 * register map and block name.  If the block name is empty the interface will
 * be installed into the map
 *
 * Parameters:
 *   register_access_interface - The register access interface to install
 *   design                    - Top level or IP to start insertion from
 *   module                    - Module type to install the method into (EG MBIST_FSM)
 *   rev                       - Specific revision of the module to install in.
 *   map_name                  - Register Map name.  If no block interface is installed into map
 *   block_name                - Register Block name within the map to add the interface to.
 *
 **************************************************************************************/
STIMGEN_API void install_register_access_interface(
  StimGen::shared_ptr<StimGen::Interface::RegisterAccess> register_access_interface,
  StimGen::Module* design,
  const std::string &module,
  const std::string &rev,
  const std::string &map_name,
  const std::string &block_name = ""
);
```

25   Custom Module

There are several occasions where an IP or SOC requires extra functionality or it may need to override default SG4 functionality. Extra functionality could be required to provide reusable sequences that are specific to an IP or SOC or to a specific revision of an IP or SOC. Most IPs will need to override default register read and write functionality.

25.1   Use Models

25.1.1   Common Sequences

Common sequences are used to implement and verify predefined sequences that multiple users would normally have to re-implement and validate in other environments. Every common sequence that can be defined and reused up front will all multiple other users to leverage and focus on other tasks and responsibilities. Common sequences must attached to the derived StimGen::Module for the appropriate IP/SOC

25.1.2   Override StimGen Default Functionality

There may be occasions where stimGen's default read and write methods must be overridden on a per instance basis. As an example we may require this functionality when stimGen's virtual register networks can't be used to describe the register network connectivity. There may be other occasions where we want the IP/SOC to intercept register operations and do something interesting with the data.

25.1.3   Extend Inheritance

AMD has several IPs that have a common basic functionality but different versions have slight differences. Custom modules can be defined for these to add extra levels of inheritance. This allows common functionality to be in a base class derived from stimGen::Module from which the different versions of the IP are derived to override as necessary. There are SOCs that have different versions of an IP instantiated on it multiple times. Virtual methods and this extended inheritance will make this seamless and the deltas a don't care for end users.

25.2   Requirements

All custom modules must utilize methods available in StimGen only. They must not utilize other libraries specific to an environment. These will not necessarily be available in all environments where StimGen is intended to be used and will cause compilation failures.

Custom modules are owned by the IP and SOC teams. All files must be stored in the <BOZO: Chris what's the directory structure?> CDS under $STEM/src/meta/sigdata/dft/SG4. All custom sequences should be accompanied by regression tests which at a minimum calls all methods to ensure that end user models and compilations can use them.

Custom modules will be loaded dynamically at runtime through a shared library. In order to access the

modules and associated sequences standard conventions must be adhered to in order to gain access to the module implementations.

25.2.1   General Requirements

73. Include only stimGen header file "StimGen.h". Don't call out other stimGen related header files with the exception of sg_VerifBase.h if you are using that one since its not included in StimGen.h. (Of course include other header files as needed)

74. Top level module must inherit from StimGen::Module somewhere in the inheritance tree.

75. The class must provide a virtual destructor if it needs to clean up any local data otherwise do not define a destructor and allow the compiler to create it for us.

76. You must provide a header and cpp implementation file

77. Cpp implementation file must include an 'STIMGEN4MODEL_EXTERN_C' for the create method (described below)

78. All header files must include Natural Docs based comments to build documentation for users. This is key to enabling other users of your modules and sequences.

25.2.2   Sample Custom SOC Module

SOC's will more than likely always have a custom module associated with them to implement various sequences. As an example lets consider reading and writing fuses via JTAG chains.

```
#ifndef CARRIZO_H
#define CARRIZO_H

#include "StimGen.h"

namespace StimGen {

  class STIMGEN4MODEL_API carrizo: public Module {
    public:

      carrizo (
         const std::string &name,
         const std::string &rev,
         const std::string &instance_name,
         Module *_parent = NULL );

       void read_fuse_chain ( Register* fuse_reg );
       void write_fuse_chain ( Register* fuse_reg );
       void read_fuse_chain ( const std::string &n );
       void write_fuse_chain ( const std::string &n );

    private:
      void config_fusechain ( const std::string &name );

  }; // End of class carrizo
}  // End of namespace StimGen
#endif // CARRIZO_H
```

Line 8: Declare carrizo class and in this case derive directly from Module.

Lines 11-15:   carrizo constructor that is setup to service all revisions of carrizo.

Lines 17-20:   Define sequences for reading and writing fuse changes. User can pass in either the fuse
                chain name or the Register object

Line 23:   Private method to handle configuring the fuse box

The implementation (BOZO: DON'T USE THIS it is TRINITY's with CARRIZO swapped in. Get it
in place for CARRIZO and then update accordingly)

```cpp
#include "carrizo.h"

STIMGEN4MODEL_EXTERN_C {
  StimGen::Module* create_carrizo( const std::string& rev ) {
    return new StimGen::carrizo ( "carrizo", rev, "", NULL );
  }
}

StimGen::carrizo::carrizo ( const std::string &name,
                            const std::string &rev,
                            const std::string &instance_name,
                            Module* parent
) : Module(name, rev, instance_name, parent) {}

void StimGen::carrizo::read_fuse_chain( const std::string &_name ) {
  read_fuse_chain( get_register(_name) );
}

void StimGen::carrizo::write_fuse_chain( const std::string &_name ) {
  write_fuse_chain( get_register(_name) );
}

void StimGen::carrizo::read_fuse_chain ( Register* fuse_reg ) {
  config_fusechain(fuse_reg->get_name());
  fuse_reg->ready();
}

void StimGen::carrizo::write_fuse_chain ( Register* fuse_reg ) {
  fuse_reg->write();
  config_fusechain(fuse_reg->get_name());
}

void StimGen::carrizo::config_fusechain ( const std::string &name ) {
  unsigned int extended_chain_config = 0x0;
  unsigned int chain_config = 0x0;

  if ( name == "Lock" ) {
    chain_config = 0x01;
  } else if ( name == "SerialNumber" ) {
    chain_config = 0x02;
  } else if ( name == "UVDSecurity" ) {
    chain_config = 0x04;
  } else if ( name == "SFP" ) {
    chain_config = 0x08;
  } else if ( name == "GNB" ) {
    chain_config = 0x10;
  } else if ( name == "Redundancy_CACandSTPC" ) {
    chain_config = 0x20;
  } else if ( name == "Reprogrammable" ) {
    chain_config = 0x40;
  } else if ( name == "Functional" ) {
    chain_config = 0x80;
  } else if ( name == "FuseRedundancy" ) {
    extended_chain_config = 0x1;
  }

  //set the extended chain config
```

```
Register *config = get_register("FuseBox.EXTENDED_CHAIN_CONFIG");
config->reset();
(*config)("ChainCapture", extended_chain_config);
config->write();

//set the chain config
config = get_register("FuseBox.CHAIN_CONFIG");
config->reset();
(*config)("ChainCapture", chain_config);
config->write();
}
```

25.2.3 Sample Custom IP Module

The greyhound_bist is a good example of an IP that requires one implementation that can be utilized across all revisions. Its register definitions are slightly different but the interface under the covers is the same.

```
#ifndef greyhound_bist_H
#define greyhound_bist_H

#include "StimGen.h"

namespace StimGen {

  class STIMGEN4MODEL_API greyhound_bist: public Module {
    public:

      greyhound_bist ( const std::string &name,
                       const std::string &rev,
                       const std::string &instance_name,
                       Module *_parent = NULL
      );

      virtual void write ( Register* r );
      virtual void read  ( Register* r );
  };
} // End of namespace StimGen

#endif // greyhound_bist_B0_H
```

Lines 8:   Declare constructor for generic instance of greyhound_bist

Lines 11-15: Constructor

Lines 17-18: Redefine the default StimGen methods write and read. This module is going to intercept calls to the method and if the target register is BISTCONFIG or FATALERROR the operation will be redefined in terms of the necessary BISTCONFIG_LOAD and FATALERROR_LOAD register operations. In affect we are creating multiple read/write operations for a single higher level operation.

The Implementation: (BOZO THIS NEEDS TO BE COMPLETED and verifiied)

```
#include "greyhound_bist.h"

STIMGEN4MODEL_EXTERN_C {
  StimGen::Module* create_greyhound_bist ( const std::string &rev ) {
    return new StimGen::greyhound_bist("greyhound_bist", rev, "" );
  }
}

StimGen::greyhound_bist::greyhound_bist (
```

```cpp
    const std::string& name,
    const std::string& rev,
    const std::string& instance_name
    Module* _parent
) : Module ( name, rev, instance_name, _parent ) {}

void StimGen::greyhound_bist::write ( Register *r ) {
  Register *bistconfig_load = get_register("BISTCONFIG_LOAD");
  bool has_request = false;

  if ( r->get_name() == "BISTCONFIG" ) {
    Number load_value = r->get_load_value();
    Number      value = r->get_value();

    for ( int offset = 1; offset <= 8; offset++ ) {
      Number lv(load_value.get_interval( (offset-1) * 32, 32 ) );
      Number v (     value.get_interval( (offset-1) * 32, 32 ) );

      // If the load value and the current value are different
      // Then write to the BISTCONFIG_LOAD register
      if ( ! lv.value_eq(v) ) {
        (*bistconfig_load)["ADDR"](offset)
                          ["DATA"](lv)
                          ["REQUEST"](1)
                          ["VALID"](0, has_request )
                          ;
        Module::write( bistconfig_load );
        has_request = true;
      }
    }

  } else if ( r->get_name() == "BISTCFG" ) {
    (*bistconfig_load)["ADDR"](0)
                      ["DATA"](r->get_load_value())
                      ["VALID"](0, has_request )
                      ["REQUEST"](1);
    Module::write( bistconfig_load );
    has_request = true;

  } else if ( r->get_name() == "BACKGROUND" ) {
    Number load_value = r->get_load_value();
    Number      value = r->get_value();

    for ( int offset = 12; offset <= 15; offset++ ) {
      Number lv(load_value.get_interval( (offset-12) * 32, 32 ) );
      Number v (     value.get_interval( (offset-12) * 32, 32 ) );

      // If the load value and the current value are different
      // Then write to the BISTCONFIG_LOAD register
      if ( ! lv.value_eq(v) ) {
        (*bistconfig_load)["ADDR"](offset)
                          ["DATA"](lv)
                          ["VALID"](0, has_request )
                          ["REQUEST"](1);
        Module::write( bistconfig_load );
        has_request = true;
      }
    }

  } else {
    Module::write( r );
    return;
  }

  // Make sure last operation completed correctly
  // and reset request to 0 for next time we come through
  (*bistconfig_load)["VALID"](0, has_request )
                    ["REQUEST"](0);
```

```
    Module::read( bistconfig_load );

  }

  void StimGen::greyhound_bist::read ( Register *r ) {
    Module::read(r);
  }
```

A third example is for a the DSM which has different functionality for specific versions. All version derive from the same base class to provide common default functionality which revisions can redefine as required

```
        #ifndef DSM_B0_H
        #define DSM_B0_H

        #include "StimGen.h"
        #include "DSM.h"

        namespace StimGen {

        class STIMGEN4MODEL_API DSM_B0: public DSM {
          public:
            int number_of_states () {
              return 4;
            }
            int number_of_vectors () {
              return 5;
            }
            int number_of_counters () {
              return 6;
            }

            DSM_B0 ( const std::string &name,
                const std::string &rev,
                const std::string &instance_name,
                Module *_parent = NULL ) :
                DSM(name, rev, instance_name, _parent)
            {

            DSM_B0 ( StimGen::Module *_parent = NULL ) :
                DSM("DSM", "B0", "", _parent)
            {}

            virtual ~DSM_B0 () {
              for (int i = 0; i < number_of_states(); i++) {
                for (int j = 0; j < number_of_vectors(); j++) {
                  Vectors[i][j] = NULL;
                }
                delete Vectors[i];
              }
              delete Vectors;

              for (int i = 0; i < number_of_counters(); i++) {
                GenCounter[i] = NULL;
              }
              delete GenCounter;
            }

          public:
            virtual void configure() {
              Vectors = new Register**[number_of_states()];
              for (int i = 0; i < number_of_states(); i++) {
                Vectors[i] = new Register*[number_of_vectors()];
              }

              Vectors[0][0] = get_register("dsm_sm_vec_0_0");
              Vectors[0][1] = get_register("dsm_sm_vec_0_1");
```

```
        Vectors[0][2] = get_register("dsm_sm_vec_0_2");
        Vectors[0][3] = get_register("dsm_sm_vec_0_3");
        Vectors[0][4] = get_register("dsm_sm_vec_0_4");

        Vectors[1][0] = get_register("dsm_sm_vec_1_0");
        Vectors[1][1] = get_register("dsm_sm_vec_1_1");
        Vectors[1][2] = get_register("dsm_sm_vec_1_2");
        Vectors[1][3] = get_register("dsm_sm_vec_1_3");
        Vectors[1][4] = get_register("dsm_sm_vec_1_4");

        Vectors[2][0] = get_register("dsm_sm_vec_2_0");
        Vectors[2][1] = get_register("dsm_sm_vec_2_1");
        Vectors[2][2] = get_register("dsm_sm_vec_2_2");
        Vectors[2][3] = get_register("dsm_sm_vec_2_3");
        Vectors[2][4] = get_register("dsm_sm_vec_2_4");

        Vectors[3][0] = get_register("dsm_sm_vec_3_0");
        Vectors[3][1] = get_register("dsm_sm_vec_3_1");
        Vectors[3][2] = get_register("dsm_sm_vec_3_2");
        Vectors[3][3] = get_register("dsm_sm_vec_3_3");
        Vectors[3][4] = get_register("dsm_sm_vec_3_4");

        GenCounter = new Register*[number_of_counters()];

        GenCounter[0] = get_register("dsm_gen_cnt_0");
        GenCounter[1] = get_register("dsm_gen_cnt_1");
        GenCounter[2] = get_register("dsm_gen_cnt_2");
        GenCounter[3] = get_register("dsm_gen_cnt_3");
        GenCounter[4] = get_register("dsm_gen_cnt_4");
        GenCounter[5] = get_register("dsm_gen_cnt_5");
      };
    };
  }  // End of namespace StimGen

    #endif // DSM_B0_H
```

Lines: sdfsdf

The implementation

```
    #include "DSM_B0.h"

    STIMGEN4MODEL_EXTERN_C {
      StimGen::Module* create_DSM_B0() {
        return new StimGen::DSM_B0();
      }
    }
```

25.2.4   Library Module Constructor Hooks

The source file must contain an 'STIMGEN4MODEL_EXTERN_C' method for the ModuleFactory to locate. It must be in the source file so that we don't have duplicate definitions. There are two type of methods that you can create which are dependent on SIGAPI IP/SOC names and revision names as well as the functionality required. The two method types are

- IP/SOC Revision specific method

- IP/SOC Generic method

Use IP/SOC Revision specific methods when you need to provide custom functionality or sequences for a specific version of an IP/SOC. The method hook into the shared library for StimGen to create an instance of this type is:

StimGen::Module* create_<NAME>_<REVISION> ()

If you have generic sequences or methods that work for all revisions of an IP/SOC use the hook

> StimGen::Module* create_<NAME> (const std::string&)

This method requires that you pass in the version of the IP/SOC to create. This is important for SOCs or IPs that are using multiple revisions of an IP. For example on carrizo there are multiple and different versions of the DSM being used in the GNB container vs Excavator core vs UNB. When the base StimGen::Module is created it must know what revision it is processing so that it correctly loads register information.

25.2.4.1   Sample Revision Specific Module Constructor Hook

The DSM has multiple revisions, each with different functionality. The extern hook for the B0 version is in SG4/src/DSM_B0_utl_dsm_model.cpp

```
#include "DSM_B0.h"

STIMGEN4MODEL_EXTERN_C {
  StimGen::Module* create_DSM_B0() {
    return new StimGen::DSM_B0();
  }
}
```
The extern hook for the A2 version is in SG4/src/DSM_A2_utl_dsm_model.cpp

```
#include "DSM_A2.h"

STIMGEN4MODEL_EXTERN_C {
  StimGen::Module* create_DSM_A2() {
    return new StimGen::DSM_A2();
  }
}
```

25.2.4.1.1.1   Sample Generic Module Constructor Hook

The greyhound_bist has multiple revisions whose functionality only differs in register definitions. As a result we can use a generic greyhound_bist module to cover the functionality of all revisions. To do this we use the following extern hook located in SG4/src/greyhound_bist_register_access.cpp

```
#include "greyhound_bist.h"

STIMGEN4MODEL_EXTERN_C {
  StimGen::Module* create_greyhound_bist ( const std::string &rev ) {
    return new StimGen::greyhound_bist("greyhound_bist", rev, "" );
  }
}
```

25.2.5   Library Search Order

There are several options for locating custom module implementation libraries. The simplest is to create one dynamic library which contains all custom sequences for the entire SOC called libSG4MODEL.so or on Windows libSG4MODEL.dll.

The first library to be looked for is lib<NAME>_<REVISION>.so or on Windows lib<NAME>_<REVISION>.dll. This library must have the create_<NAME>_<REVISION>() hook.

The second library to locate is lib<NAME>.so or on Windows lib<NAME>.dll. This library can contain all revision specific implementations or the generic ones. StimGen will first look for hook

create_<NAME>_<REVISION>() and then create_<NAME>(const std::string&)

The final library to look for hooks in is the libSG4MODEL.so or on Windows libSG4MODEL.dll. This library can contain all revision specific implementations or the generic ones. StimGen will first look for hook create_<NAME>_<REVISION>() and then create_<NAME>(const std::string&)

25.3   Using Custom Sequences

To use sequences defined in the various custom modules users must know what type of class corresponds to the module and then down cast it accordingly. StimGen provides the following Module method to assist with this

> template<typename T>
> T* Module::get_typed_instance(const std::string &instance_name)

Example Code:

```
#include <IPGPU0.h>

// Code removed for simplicity

StimGen::ModuleFactory factory ( "sgxml", "." );
Module *dut = factory.import( "SOC.sgxml", "" );
IPGPU0 *gnb = dut->get_typed_instance<IPGPU0>("IPGPU0");

// Now use custom GNB sequences
gnb->disableBroadcast();
gnb->apply();

gnb->setBC1500SIBAutoReset(false);
gnb->setSelWirAutoReset(false);
```

Line 1:   You must include the header file that defines the IP/SOC class you are casting to

Lines 5-6:   Load the SGXML

Line 7:   Get this instance called 'IPGPU0' and cast it to type 'IPGPU0'

Lines 10-14:   Now use the sequences available for that object

25.3.1   Things to be aware of

25.3.1.1   Cygwin (g++)

Cygwin uses dll extensions and locates dlls by searching the PATH environment variable. It does not support ld directive -rpath or environment variable LD_LIBRARY_PATH. This option will be passed in but is ignored.

25.3.1.2   Windows (Mingw g++)

Mingw on Windows uses dll extensions and locates dlls by searching the PATH environment variable. Mingw g++ does not support ld directive -rpath or environment variable LD_LIBRARY_PATH. This

option will be passed in but is ignored.

25.3.1.3   Linux (g++)

Different versions of g++/ld command appear to handle shared libraries differently. We've observed compiles where the executable will look for shared libraries in the current directory and others where it will not. If you can shed some light on this please share. To account for this -rpath='.' should be added to the compile/linking of the toplevel executable or you should set LD_LIBRARY_PATH prior to execution.

25.3.2   Example

26   External Sequences

One of the challenges we will encounter is integrating new sequences in a timely manner. Design owns the sequences and is responsible for implementing and making them available. To enable teams to produce their own custom sequences and use them prior to availability within the stimGen model we will define a standard interface or method authoring rules to make the sequences usable and consistent across all Ips and SOCs.

Instead of using C++ objects to call methods against

```
        instance->foo( arg0, …, argX );
```
we will utilize C style methods where the first argument will be a reference to the Module or derived Module that the method should be called against.

```
        foo ( instance, arg0, …, argX );
```

26.1   Requirements

The custom sequences must check that the instance passed into the is of the correct type and version and should throw an exception if they are not.

C methods do not support polymorphism so a unique method name must be defined for each version type of a Module or the method must be smart enough to deal with different versions of an IP.

27   Custom Regression Tests

StimGen provides the ability to provide custom IP/SOC sequences and access procedures. To ensure that these are working correctly and more generally ensure that changes to StimGen core code don't change expected vectors, StimGen provides a methodology to add IP/SOC custom regression tests. StimGen utilizes google's gtest for all unit and regression tests.

27.1   Google gtest

Background information about gtest can be found here
https://code.google.com/p/googletest/wiki/Documentation

27.2   Requirements

All gtest regression test 'test_case_name' and 'test_name' names must not include underscores '_'.
https://code.google.com/p/googletest/wiki/FAQ#Why_should_not_test_case_names_and_test_names_contain_underscore

27.3   Synchronizing with SIGDATA

This section defines how to synchronize SIGDATA with RTL releases where SIGDATA documentation is not stored with RTL. If all information is stored in the CDS repository with the RTL this synchronization process should not be needed.

SIGDATA should be viewed as the latest and greatest version of the documentation and should match the desired end state of the design. This doesn't entirely align with the needs of front end design teams (who own the documentation) that must have the ability to work with documentation that is synced with previous RTL releases. As an example consider ATPG teams which are always working with previously released data and the latest version of the documentation may not necessarily correctly reflect registers defined in the RTL.

To synchronize RTL and documentation the design teams must coordinate RTL and documentation development and ensure that they align at sync points. They must then manually label the SIGDATA client view with a meaningful label.

To manually add a label to the SIGDATA perforce repository we must first identify the changelist number you currently have checked out. The assumption being made is that you have a local checkout with all updates submitted and you have validated that RTL and SIGDATA are synchronized. To get the changelist number issue p4 command

        p4 -scverpf01:1671 -c <LOCAL_CLIENT> \
            changes -s submitted -m 1 //<LOCAL_CLIENT>/...#have

This will write to the terminal output for the form

        Change 61246 on 2013/08/23 by pjakobse@pjakobse_pc_sigdata 'Look for .sgxml and .xml'

In this case the local checkout (#have) contains changelist number 61246. To create a new label for

this release issue the command

    p4 –atlvp4s04:1671 label <LABEL_NAME>

    This will open an editor. Add to the file:

        Release: <RELEASE_FROM_CHANGES>

    Replace the View with
        View:
            //depot/sigdata/...

    This will label the entire sigdata repository.

When updates are complete save the file and exit. Fix any errors and resubmit until complete.

This will create a label with the name <LABEL_NAME> and should be standardized and the following format used:

    <LABEL_NAME> := <SIGAPI_PROJ>_<QUALIFIER>_<CHANGELIST#>

For a Carrizo RTL sync SIGAPI_PROJ='15h_CZ_A0' and QUALIFIER='RTL_SYNC'

    15h_CZ_A0_RTL_SYNC_<RTL_CHANGELIST#>

To the use this label you will need to create a local check out of CZ SIGDATA and sync to this label

    79. Create a new directory and cd into it. This will be where CZ documentation will be checked out

    80. Create a new client

```
p4 -atlvp4s04:1671 \
   -c<USERID>_<LABEL_NAME> \
   client –t  <SIGAPI_PROJ>
```

    Example:

```
p4 -atlvp4s04:1671 \
   -cpjakobse_15h_CZ_A0_RTL_SYNC_1234567 \
   client –t  15h_CZ_A0
```

    81. Check out the sync point

```
p4 -atlvp4s04:1671 \
 -c<USERID>_<LABEL_NAME> \
 sync @<LABEL_NAME>
```

    Example:

```
p4 -atlvp4s04:1671 \
```

```
     -cpjakobse_15h_CZ_A0_RTL_SYNC_1234567 \
     sync @15h_CZ_A0_RTL_SYNC_1234567
```

82. Use the synchronized data with SIGAPI use SIGAPI option '-c <CLIENT_NAME>'
    and insert the appropriate <CLIENT_NAME> created with the client command

    Example: Run genSG4.pl to create the SG4 model (SIGAPI/genSG4 will locate your
    sigdata documentation from the p4 client definition created in step 2

```
$SG4COREBASE/bin/genSG4.pl -proj 15h_CZ_A0 \
     -c pjakobse_15h_CZ_A0_RTL_SYNC_1234567  -gen_sgxml
```