Automatic Testbench Build to Reduce Cycle Time and Forster Reuse

Joachim Geishauser, NXP, Munich, Germany (*joachim.geishauser@nxp.com*)

Alexander Schiling, NXP, Munich, Germany (*alexander.schiling@nxp.com*)

***Abstract*—The cycle time to get a new product to market is constantly getting smaller. The typical industry design cycle is anywhere in between nine and 12 months [1], derivative MagniV or Kinetis designs however have a design cycle that is in the range of 3 months from planning to tape-out. This paper shows a concept to automatically create very flexibly a complete testbench in a modular way. The created testbench includes Verilog, C, documentation and tool setup files.**

Keywords—Verification; Automation; Reuse

I. Introduction

The cycle time to get a new product to market is constantly getting smaller. The typical industry design cycle is anywhere between nine and 12 months [1], derivative MagniV or Kinetis designs however have a design cycle that is in the range of 3 months from planning to tape-out.

On the design side this pressure was initially addressed by the introduction of standard IP interfaces like the ARM® Advanced High Performance Bus (AHB) to reduce the effort of assembling the IP modules to the final SoC product. After the successful introduction of standard interfaces the SoC assembly was further accelerated by the introduction of an automated SoC assembly based on the IP-XACT standard for example.

The verification effort is anywhere between 50 and 80% of the overall development effort [2]. Being able to cut down the verification time does have a big impact on the overall time to market. Like the concept of the standard IP interfaces the verification language allows to create verification IPs with standard interfaces which ease the assembly of the testbench [3]. On the testbench side however an automation does require more effort than on the design side. This is because a verification environment is more heterogeneous. For example, it may consist of different programming languages like C/C++ and SystemVerilog, which requires more effort in setting up, getting all the required files and flows in the right places.

Besides the time saving aspect, the automation acts as a catalyst for the modules that are delivered into the automation process in the sense that the code provided will be optimized for as minimal updates as possible to ease the handling.

II. Verification ChallEnge

The automation described hereafter is built on top of the directory structure IPDSS of the NXP Design System called Stingray. This is an implicit requirement, but the implementation is completely independent of the design System. The architecture of the build environment contains a small build core and provides an application programming interface (API) to adopt it to the user requirement. This enables a modular approach, where a verification IP (VIP) provides the required automation files to integrate itself into a testbench. The build flow and the modular approach will be detailed in the next sections.

A. Testbench Bill of Material

Since the components defined by the term testbench vary, this section defines what is meant by testbench Bill of Material (BOM). The testbench BOM that is handled by the testbench automation described in this paper goes beyond the Verilog files for the testbench

- Tool setup files for VCS and Certitude
- Design Environment Metadata Files
- C Stimulus Files
- Linker Files
- Regression Files
- Verification Guide
- X-Filter Handling
- SoC Memory Map Files

III. 3. Testbench Build Role Models

Since building a SoC testbench is not a simple task, multiple parties are involved. There is always some experienced engineer needed that defines the architecture of the testbench. Second, there will be engineers that will work on implementing testcases for identified verification features. In most cases this features are grouped by function or modules of the SoC. And finally, there will be another engineer that will assemble and maintain the SoC testbench and will do the releases.

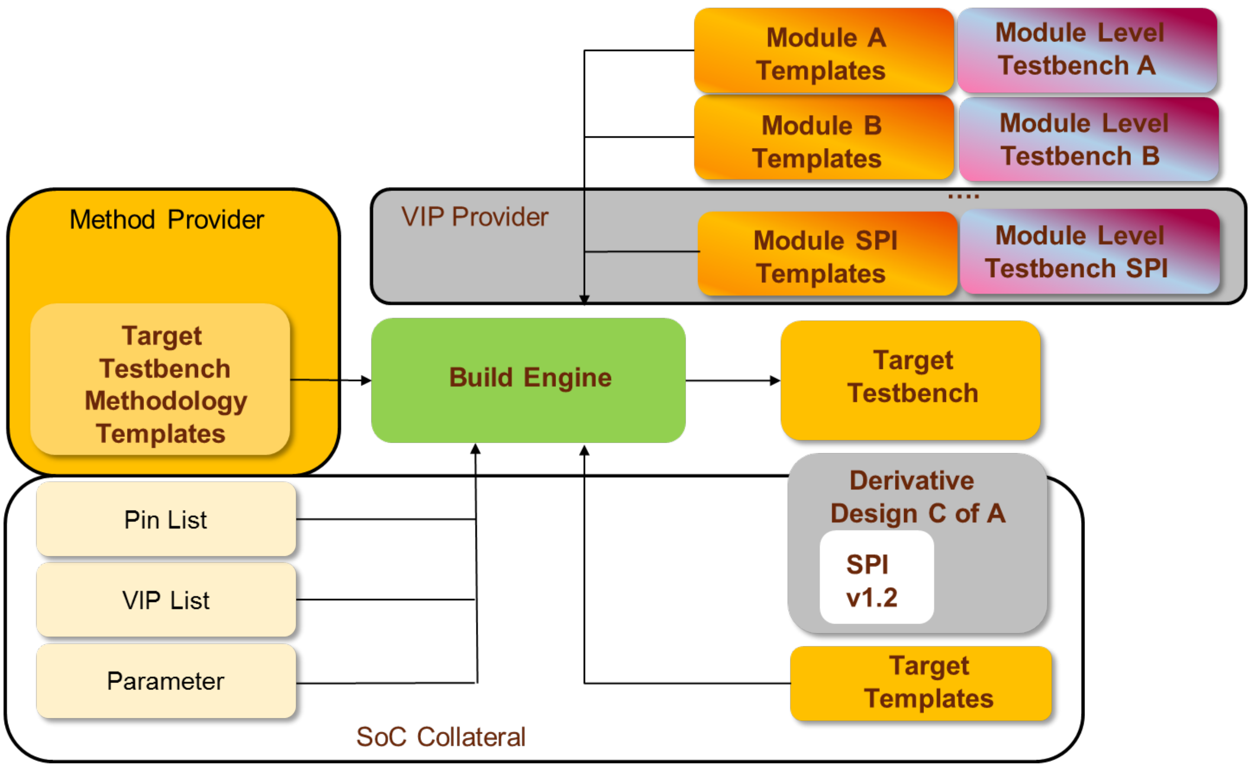

Figure 1. Role Models

In Figure 1 the three roles are shown. The testbench architect will work as the Method Provider. This role defines methodology templates. Most of the resources will work in the role of the VIP provider. In this role, they will provide templates that follow the method provider's templates. The third role is the SoC role. This role provides the SoC collaterals as well as some SoC VIP templates that follow also the method provider's templates.

IV. Automation Flow

The Automation flow automates the traditional flow to copy and update files from an existing testbench. The files that are copied are mainly delivered by the VIP provider following the rules of the method provider.

The files coming with the Verification IP block are based on template files. A small script provides a Unix shell environment that the Verification IP templates can use.

The template files reside in a directory called templates. The structure below the template directory reflects an IPDSS structure and holds the content and location for the files in the target build directory structure.

Figure 2 shows the build flow. The main input file is the VIP list. The VIP list contains all the VIP blocks that shall be used to build the testbench. The first element in the list does have a special meaning. For our case this block defines the build method for MagniV SoC testbenches. This special handling is also reflected in Figure 2. Only elements in the template directory structure from the first block will run through the final callback phase. All the other calls shown in the figure are applied to every file in the template directory structure of the blocks listed in the VIP list file. A VIP List example is shown in Figure 5 VIP List Example.

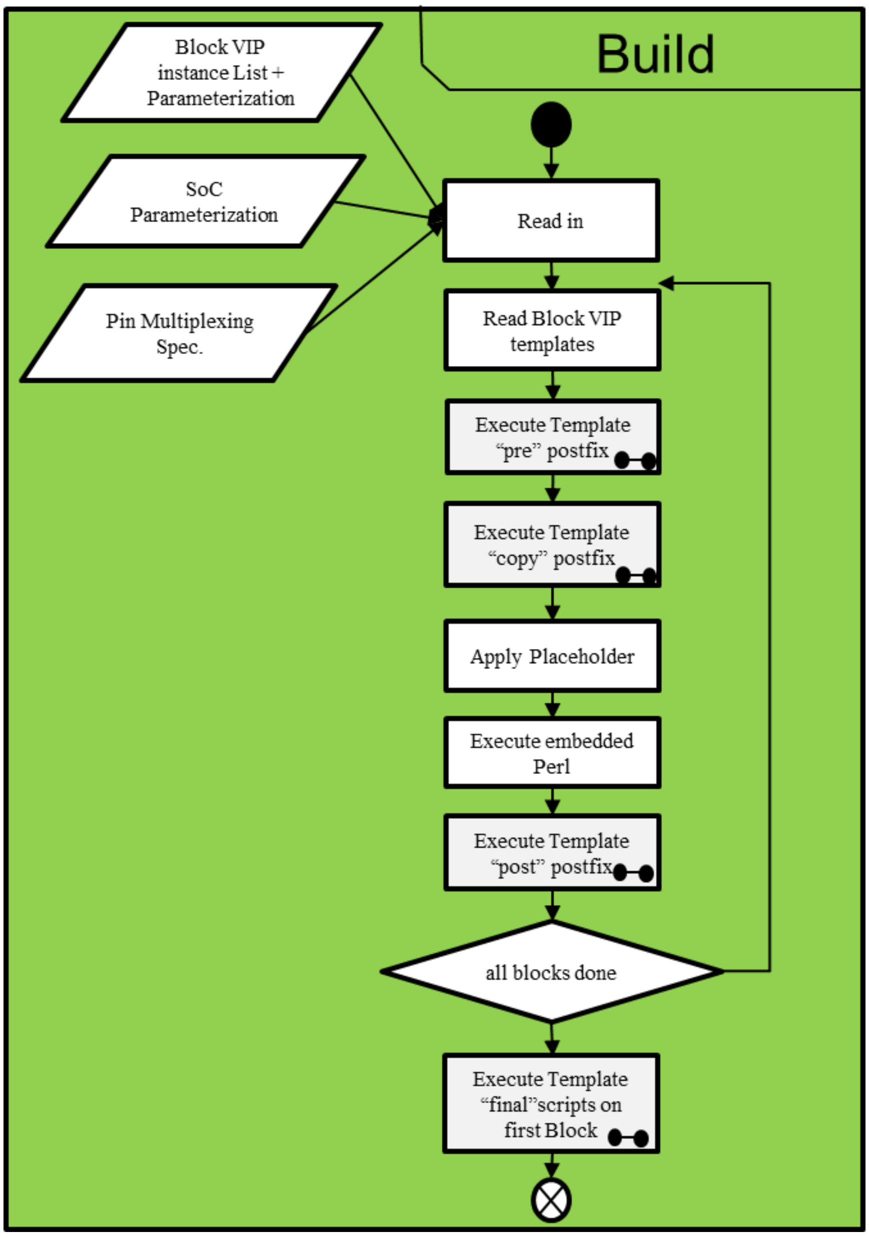
Build
Block VIP instance List + Parameterization
SoC Parameterization
Pin Multiplexing Spec.
Read in
Read Block VIP templates
Execute Template "pre" postfix
Execute Template "copy" postfix
Apply Placeholder
Execute embedded Perl
Execute Template "post" postfix
all blocks done
Execute Template "final"scripts on first Block

Figure 2. The Execution Flow

For a file filename in the template tree the following is looked for and will be executed if supplied

```
filename.pre.sh
filename.copy.sh
filename.post.sh
filename.final.sh
```

The execute template process is shown in the Figure 3.

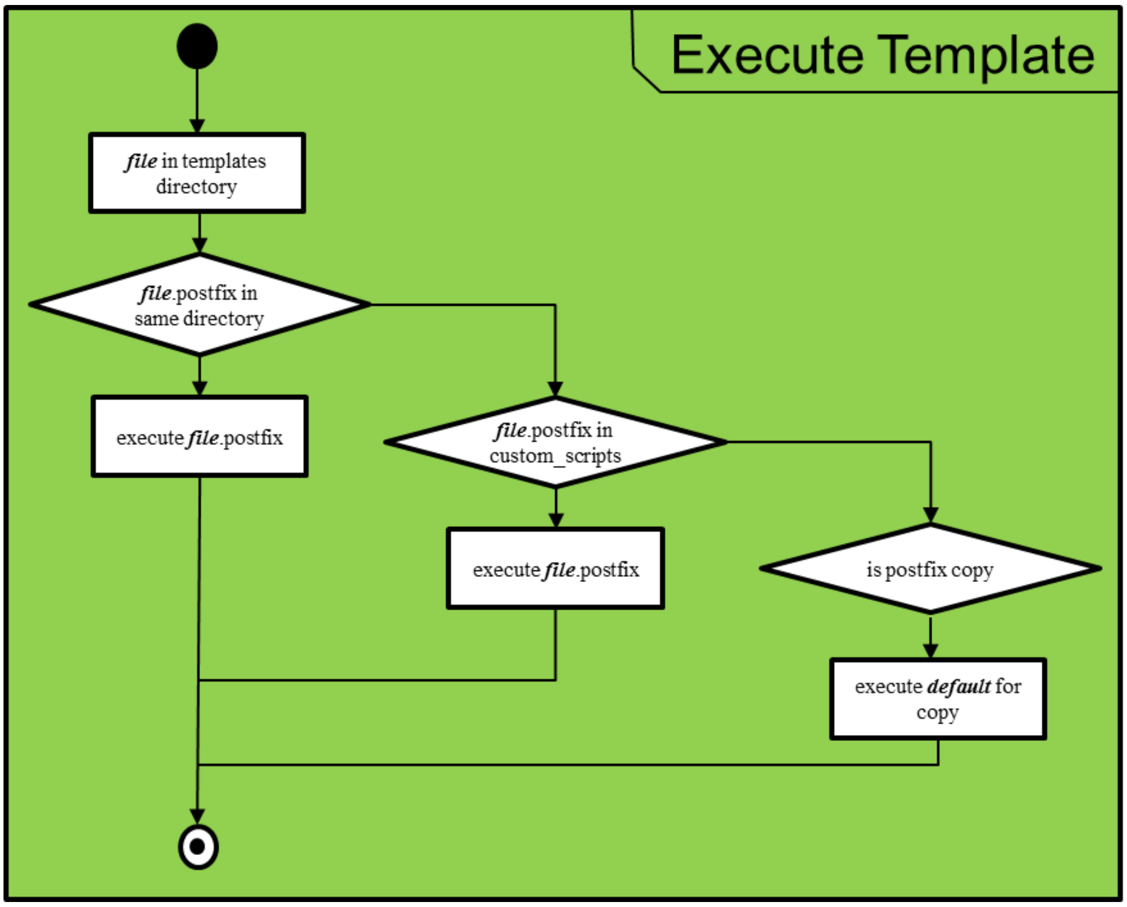
Execute Template
file in templates directory
file.postfix in same directory
execute *file*.postfix
file.postfix in custom_scripts
execute *file*.postfix
is postfix copy
execute ***default*** for copy

Figure 3. Copy Mechanism Flow

Below is an example for the copy postfix

```
spi_vip/templates/testbench/top_packages_v/pin_multiplex_names_pkg.sv
spi_vip/templates/testbench/top_packages_v/pin_multiplex_names_pkg.sv.copy.sh
```

With providing the copy postfix file the copy can be changed to a e.g. concatenation copy.

V. Copy Mechanism

As indicated in the earlier sections the main function of the automation engine is copy and replace. The copy mechanism is outlined in this section. The engine uses the VIP List to look for a matching block. Within the block two directories are elaborated. One is the templates directory and the other is the bin directory.

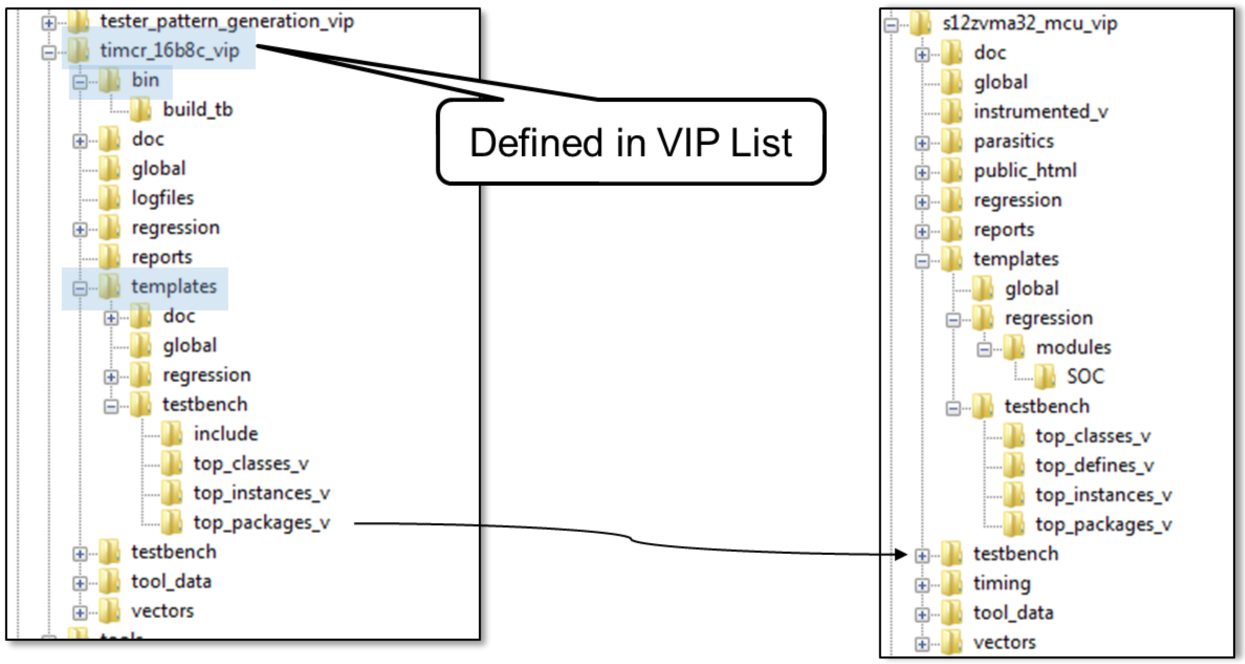

Figure 4. Copy Mechanism Example

On the left side of Figure 4 a VIP block directory structure is shown. The shaded elements show the important directories. The timcr_16b8c_vip is the VIP block that is referenced in the VIP list. Within the VIP block the bin and templates directory is shown. On the right-hand side, the destination directory structure is shown, this is the s12zvma32_mcu_vip block, the SoC testbench. The copy mechanism uses the structure within the template VIP directory as the default and copies it to the destination structure. For example, a file in the templates top_packages_v directory is copied to the testbench/top_package_v directory on the right-hand side directory structure.

A. Parameter Handling

Within the VIP list file also placeholder can be specified. Within the VIP list file the placeholder information is organized line wise. Alternatively, these placeholders can be specified in the SoC parameterization file. In this file the placeholder replacement definitions can be specified line by line, which increases the readability.

Finally, the pin multiplexing file is provided to the flow. This is done in the form of a placeholder replacement. The syntax of the pin multiplex file is a simple csv file.

An example of a VIP list file is shown in Figure 5

```
s12z_mcu_vip, s12zvm256_mcu_vip
s12zvm256_pim_vip,pim
s12zvm256_bist_vip,bist
s12_cpmu_uhv_vip,cpmu
spi_vip,spi0,__BASE_ADDRESS__=32'h780
```

Figure 5 VIP List Example

1) Basic Replacement

In the VIP list file the spi_vip has a placeholder replacement, in this case the placeholder is

```
begin : __INSTANCE__
__instance__StimManager = new (
.parentStimulusManager (this),
.name ("__INSTANCE__"),
…
.spi_rspdr_if_i (__instance___rspdr_if_i),
.base_addr (__BASE_ADDRESS__));
end : __INSTANCE__
```

__BASE_ADDRESS__ and will be replaced in the templates of this block with 32'h780.
The next core snippet shows an example excerpt of an SoC parameterization file

```
### NO WHITESPACES ALLOWED __...__=...
### General
__PART__=s12zvm256
__SOC_NAME__=S12ZVM256
```

Finally, to complete the illustration of the flow, the following shows a template file with simple placeholders.

2) Perl Template Replacement

Sometimes a simple placeholder cannot do the job. For these cases perl template support was added. Why perl, because perl is also very popular in the design community and has its strength in processing text using regular expressions and the like.

The template flow allows perl code to be embedded into the templates for testbench code generation. This is done by using the "Text Template" Perl module [4] which augments the basic placeholder replacement approach.

By embedding the perl code within the template, the static code together with the parts that are generated and how they are generated can be seen in context.

In Figure 6 an example of a template with embedded Perl is listed:

```
package pin_names_pkg;
   typedef enum {
      __[ create_pin_list (@pins); ]__
      NUMBER_OF_PINS} PINS_T;
   } PINS_T;
endpackage : pin_names_pkg
                          .
```

Figure 6 Embedded Perl Code in template

The implementation of the function used in the template is shown in **Fehler! Verweisquelle konnte nicht gefunden werden.**

As last part of the flow, the VIP blocks can provide shell callbacks for every element within the templates directory structure.
In Figure 8 an example of a call back shell is listed:

```
sub create_pin_list {
   my @pins = @_;
   my $code = "\n";
   $code .= "// START: create_pin_list \n";
   foreach my $pin (@pins) {
      $code .= "      PIN_$pin->{pin}, // ";
      foreach my $func (@{$pin->{"func"}}) {
         $code .= "$func / ";
      }
      $code .= "\n";
   }
   $code .= "// END: create_pin_list \n";
   return $code;
```

Figure 7 Example of embedded Perl Code

As last part of the flow, the VIP blocks can provide shell callbacks for every element within the templates directory structure. In Figure 8 an example of a call back shell is listed:

```
#! /bin/sh
# Add module specific routing pim_routing
argv=$*
PROGRAM=`basename $0`
TEMPLATE_DIR=$1
  . $TEMPLATE_DIR/mk_common.sh

capi=`echo ${block_name} |
      /bin/sed s/_vip/_capi/g `
if [ -w  pim_routing.c ] ; then
  /bin/cat pim_routing.c >> ${block_name}/${capi}/pim_routing.c
  apply_placeholder ${block_name}${capi}/pim_routing.c
else
  echo "  Skip cat since there are no write permission target
${block_name}/${capi}//pim_routing.c"
fi
```

Figure 8 Example of a callback shell script

In the MagniV testbench the introduction of the automation did cut down the testbench setup time from 5 to 1 day. On top of the time saving, this automation did allow to move the verification focus away from getting the syntax right, to higher level and more verification centric questions like what base address is defined for this module in the specification.

For example, the generation of different parts of the SoC-testbench based on the information in the "Pin Multiplexing Spec." is done using perl.

B. *Testbench Architecture Impact*

1) Testbench Structure

The SoC-testbench for the MagniV devices is built with a high level of reuse made from the block level environments. Block level VIP components are added and connected in many different places within the SoC-testbench. To do this manually is a relatively repetitive and boring work, which cries for automation. In Figure 9 the SoC-Testbench Structure is depicted. The colored blocks are reused from module level and instantiated in the SoC-testbench.

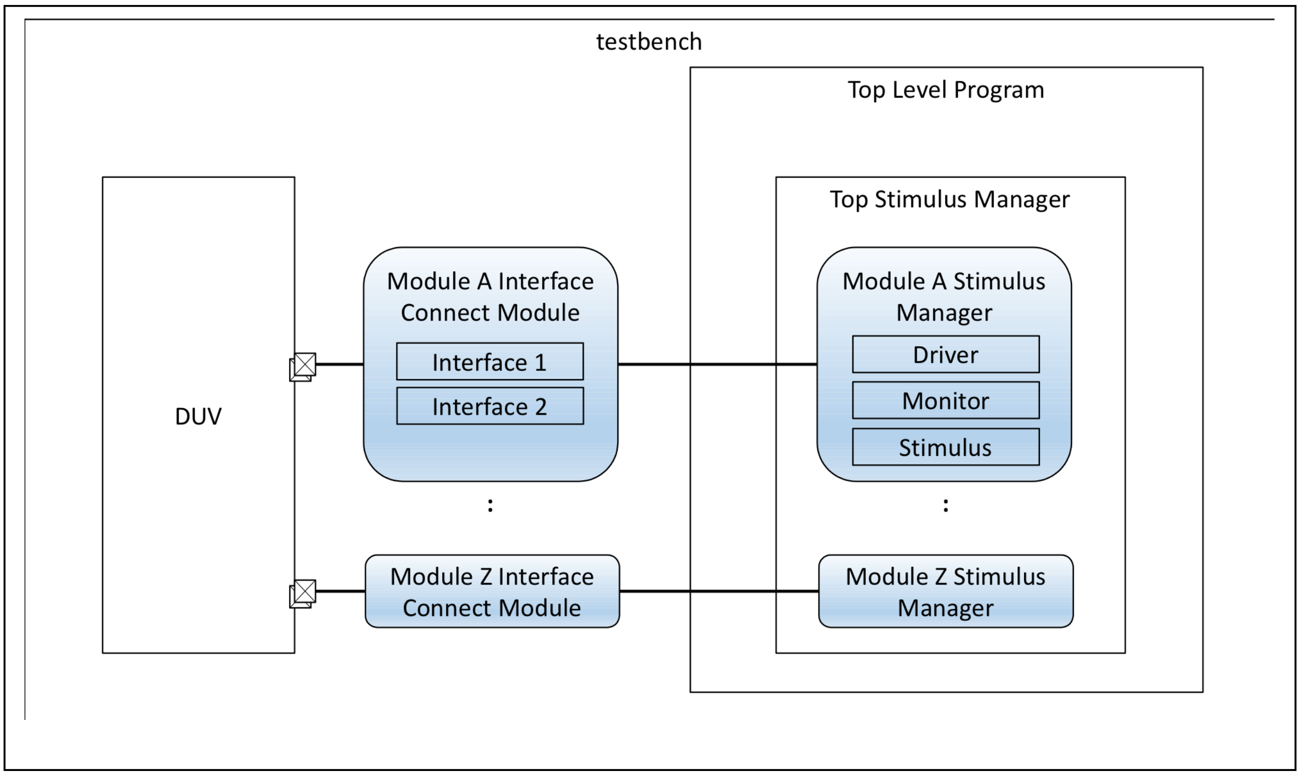
testbench
Top Level Program
Top Stimulus Manager
DUV
Module A Interface Connect Module
Interface 1
Interface 2
Module A Stimulus Manager
Driver
Monitor
Stimulus
Module Z Interface Connect Module
Module Z Stimulus Manager

Figure 9 SoC-Testbench Structure

2) Module to SoC Reuse

To ease reuse from module to SoC-level a common method how to package VIP components and instantiations is used in the MagniV ecosystem. This regular structure also helps when implementing an automatic build process.

Two main components for integration can be mentioned:

- Interface connect module
- Stimulus Manager

The “Interface connect module” instantiates all System Verilog interfaces and connects them to the DUV. The “Stimulus Manager” is a System Verilog class which instantiates all VIP components like Drivers, Monitors and Stimuli and connects them to the interfaces. It is also responsible for registration of the components with the testbench. The “Stimulus Manager” itself is finally instantiated in the “Top Stimulus Manager” in the SoC-testbench.

Instantiating these module VIP components in the SoC-testbench and making the connections is handled by the automatic testbench build.

Next to the structural grouping the concept of API definitions is important to allow seamless reuse from stimulus written on module level on SoC-Level. The following lists some APIs that were defined in the MagniV testbench infrastructure:

- Interrupt API
- Stimulus Manager
- Register Flash Preloading
- Running System Verilog on the Embedded Core - Core Slave Mode [5]
- System Verilog and CAPI Register Access API [6]
- Reusable DMA Stimulus API [7]
- Reusable Clock API – [3]
- Control Loop API – [8]

- Reusable Low Power Stimulus API [9]

VI. Results

The automatic testbench build flow was used on the last 3 tape-outs to generate the sub-testbenches as well as the SoC testbench. An effort reduction for this task from about 5 days to 1 day was demonstrated.

VII. Conclusion

The introduction of the testbench automation did pay off in many ways. The most obvious one is the effort reduction. The other very important observation is that work focus did change from fixing syntax errors in the testbench to fixing verification parameters. Finally, the automation did sharpen the APIs used to assemble the testbench since every unnecessary interface creates now some work on the VIP side and encourages therefore the VIP provider to clean up the APIs. In the past, this additional work ended up in the SoC testbench owner's effort list and did not get fixed on the VIP side.

VIII. References

[1] 24-hour chip design cycle called possible, Richard Goering, http://www.eetimes.com/document.asp?doc_id=1199922.

[2] Verification of SoC designs, J.A. Abraham, UT Austin, ECE Department, http://www.cerc.utexas.edu/~jaa/soc/lectures/14-2.pdf

[3] Reusable Functional Clock Verification, SNUG 2010

[4] Mark Jason Dominus, "Text Template", http://search.cpan.org/~mjd/Text-Template-1.46

[5] Running System Verilog on the Embedded Core - Core Slave Mode, SNUG 2004

[6] System Verilog and CAPI Register Access API – DATE 2002, SNUG 2005

[7] Reusable DMA Stimulus API – SNUG 2006

[8] Control Loop API – SNUG 2014

[9] Reusable Low Power Stimulus API - SNUG 2011