A Configurable Test Infrastructure using a Mixed-Language and Mixed-Level IP Integration IP-XACT Flow

Erwin de Kock
NXP Semiconductors
High Tech Campus 46
5656 AE Eindhoven
The Netherlands
erwin.de.kock@nxp.com

Jos Verhaegh
NXP Semiconductors
High Tech Campus 46
5656 AE Eindhoven
The Netherlands
jos.verhaegh@nxp.com

Serge Amougou
Magillem Design Services
4 rue de la pierre levée
75011 Paris
France
amougou@magillem.com

ABSTRACT
We present a reusable test infrastructure for RTL designs as an application of mixed-level and mixed-language integration using the IP-XACT standard. The test infrastructure is configurable, meaning that specific configurations can be generated from a template. The main part of the components in the test infrastructure is implemented using the SystemC TLM2 standard. A small part is implemented in VHDL. The design technology to generate configurations uses the IP-XACT standard. We present an application of the reusable test infrastructure for randomized IC verification. To this end, a specific test configuration is integrated with additional stubs and a DUT which are described in Verilog and VHDL. Software images are executed both on the test bench and the DUT. Our results demonstrate an efficient integration flow for mixed-language and mixed-level IPs through flow automation.

Categories and Subject Descriptors
B.7.2 [**Design aids**]: Simulation and Verification.

General Terms
Design; standardization; languages; verification.

Keywords
IP-XACT; SystemC; TLM; FastModel; RTL; mixed-language; mixed-level; randomized verification.

1. INTRODUCTION
The goal of this paper is to demonstrate application of a mixed-language and mixed-level IP integration flow using IP-XACT-based design technology. Furthermore, we demonstrate IP and subsystem configurability using a configurable Test Bench. We show that a large part of this flow can be automated using the IP-XACT standard and IP-XACT tooling. Another contribution of this paper is the use of software for system-level verification.

A key issue for system integration companies is the ability to separate IP models from tools that work with IP models. IP models should work in a variety of tools and tools should work with a variety of IP models. The IP-XACT standard [10] enables this by standardizing IP meta-data descriptions and tool interfaces to operate on meta-data. IP-XACT 1.4 and later versions support both RTL and TLM IP models. There are few integration and simulation tools that support this combination in an IP neutral way. We have used Magillem IP-XACT tools [7] for integration and Cadence Incisive simulator [3] for simulation. Furthermore, we have used the ARM FastModel tools [1] to generate an instruction-accurate Cortex-M3 processor model. These technologies have been used to generate a system-level test bench. On top of this test bench we developed software for randomized verification.

Related work in the area of randomized verification is the Universal Verification Methodology (UVM) [16]. This approach has proven itself for IP verification, also in earlier incarnations such as OVM and *e*. We argue that our approach can be combined with existing IP verification methodologies such as UVM. This is enabled by Software-Driven Verification IP. Related work in the area of IP integration often splits in RTL IP integration and TLM IP integration. For RTL IP integration there are tools such as Mentor HDL Designer [8] and Synopsys Core Assembler [14]. For TLM IP integration there are tools such as ARM FastModels [1], Carbon SOC Designer [4], Mentor Vista [9], and Synopsys Virtualizer [15]. Some tools support co-simulation, for instance in Synopsys Virtualizer, but a true mix of RTL and TLM is rare.

The outline of this paper is as follows. In Section 2 we provide a short introduction to IP-XACT technology. In Section 3 we present our Reusable Test Infrastructure. In Section 4 we apply this infrastructure for randomized verification of a microcontroller IC. In Section 5 we present our results. We conclude in Section 6.

2. IP-XACT Design Technology
We make use of design technology that is based on the IP-XACT standard IEEE 1685-2009. This standard provides a way to describe meta-data of hardware IP components and designs as well as a tool interface to operate on the meta-data. IP component meta-data that can be described are hardware interfaces, software interfaces, and file sets. Hardware interface descriptions include pinning, bus interfaces, and instantiation information. Software interface descriptions include address spaces, memory maps, registers, and bit fields. File set descriptions contain files and options needed for compilation. Design meta-data that can be described are component instances, connections between

CODES+ISSS'12, October 7–12, 2012, Tampere, Finland.

component instances, and connections between component instances and design boundaries.

Our design technology includes commercial tooling from Magillem Design Services. This tooling includes support for IP packaging, platform assembly, netlisting, and generator development and execution. IP packaging is the generation of meta-data for components. Platform assembly is the generation of meta-data for designs. Netlisting is the generation of code for design structures in SystemC, VHDL, or Verilog. Generator development is the development of software tools that operate on the meta-data database in order to produce meaningful output or to generate additional meta-data. We have developed in-house generators for HDL build flow generation, SystemC IP template model generation, and hardware-dependent software generation. These generators have been used for DUT integration in earlier projects. In this paper, we re-use this flow for test bench integration.

3. Reusable Test Infrastructure

Our Reusable Test Infrastructure (RTI) is a configurable SystemC-based Test Bench. In Figure 1 the template for this infrastructure is shown.

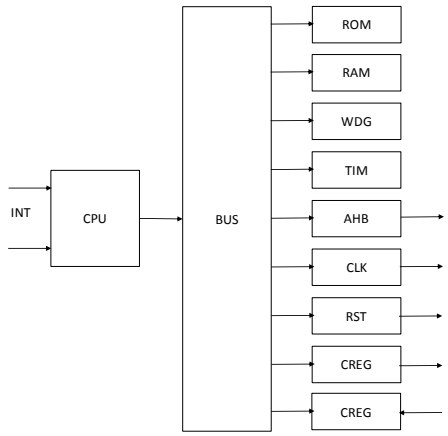

Figure 1. Reusable Test Infrastructure template.

3.1 RTI Components

We use an ARM FastModel Cortex-M3 processor model (CPU) for software execution in the RTI. This is an instruction-accurate model. Furthermore, we make use of the Nested Vector Interrupt Controller (NVIC) in the processor model to support interrupts (INT) in RTI. We wrap the FastModel in a SystemC TLM2 [13] wrapper using generic payloads.

The processor is connected to a generic SystemC TLM2 bus (BUS). The number of masters and slaves that can be connected to the bus are C++ template parameters of the bus model.

Furthermore, we have a generic TLM2 memory model that we use to model ROM and RAM. This model has a special member function that can be called from the Cadence IUS Tcl command line for preloading binary files in memory instances.

Next, we have peripheral models. All these models have a TLM2 target interface that is connected a register set. To model the registers, we make use of the SystemC Modeling Library version 2 (SCML) [11]. There is timer model that we use both as watchdog timer (WDG) and normal timer (TIM). It can be used to implement a software wait function, to abort simulation in case of time outs, and to get the actual simulation time in software. There are models for clock generation (CLK) and reset generation (RST). The shape of clock and reset pulse can be controlled via register settings. Also, there are peripheral models for register input and output (CREG). The initial values of all peripheral registers can be set via SCML properties.

Finally, we have a TLM2-to-AHBlite signal-level transactor (AHB). This transactor translates TLM2 generic payload transactions into AHBlite slave signal-level transactions. This translation is implemented in Esterel [6] from which synthesizable VHDL code is generated. The transactor can be used to connect an AHBlite slave component to a TLM2 bus. This enables us to connect RTI instances to signal-level DUTs.

All RTI components have been described in IP-XACT XML documents, except for the TLM2 bus. The bus is configurable in terms of the number of masters and slaves that can be connected. This type of configurability cannot be described in IP-XACT documents. For this reason, we generate IP-XACT XML documents for specific TLM2 bus configurations when needed, as explained in the next section.

For the RTI peripheral components, part of the implementation can be generated automatically from the IP-XACT description. First, SCML register descriptions can be generated from the IP-XACT register descriptions. They are part of a self-contained SystemC module from which we inherit to implement callback functions on the register accesses. Next, software register access functions can be generated from the IP-XACT register description. To this end, first a register hardware abstraction layer is generated, called vHAL, based on C macros. The vHAL format is described in [5]. A vHAL has the following content.

- A set of macros that define the offset of the registers relative to the instance base address.
- A set of macros that define register characteristics regarding read, write and POR (Power-On Reset).
- A set of bit field macros that define location identifiers of register bit fields.

Finally, get and set functions are generated which are implemented on top of the vHAL. On top of these get and set functions we have implemented a software API similar to the vHAPI functions described in [5] manually. Here we list some examples of such functions.

- Generating clock and reset pulses.
- Registering of interrupt service routines.
- Printing strings and values.
- Sleeping of CPU.
- Resetting of RTI configuration.

3.2 Configurability

3.2.1 RTI Configuration Options

The RTI is configurable. Configuration options include the number of interrupts, the number of AHB interfaces, the number of clock generators, the number of reset generators, and the number of register inputs and outputs. For each AHB interface,

the address offset and the address range can be specified such that the user can control the AHB addresses that are generated from the RTI configuration.

Additional configuration options are the ROM and RAM sizes, the address offset of the heap in the RAM, and the size of the heap and stack. Also the watchdog has configuration options to control the time out functionality. Finally, there are configuration options for the ARM FastModel processor including options for the processor frequency, the quantum value, and the enabling and disabling of the CADI interface for debugger connections.

3.2.2 RTI Configuration Generation

The above-mentioned configuration options are supplied by users in a text file. We parse this text file using a bash script and we configuration options in the IP-XACT database as generator parameter values using the command line interface of the commercial IP-XACT tools. Next, the commercial netlister and in-house generators are called to generate the RTI configuration. These generators make use of a software interface to operate on the IP-XACT database that is similar to IP-XACT Tight Generator Interface, but this software interface adds more functionality. Specifically, we make use of IP-XACT component editing functionality, for instance, to generate specific IP-XACT TLM2 bus descriptions. The RTI configuration generation process contains three steps.

In Step 1, additional meta-data is generated to describe the TLM2 bus configuration and RTI configuration. The TLM2 bus configuration depends on the number of interfaces specified in the configuration options. In the meta-data, the correct number of transactional ports, bus interfaces, and corresponding address spaces with offsets and ranges is generated. Subsequently, an IP-XACT design is generated containing the desired component instances and interconnect. This design is encapsulated as a hierarchical view in an IP-XACT component that reflects the RTI configuration. All relevant signals are exported from the RTI design to the RTI component.

In Step 2, files are generated that are needed at build time of the RTI configuration. For hardware, these files include a SystemC netlist for the implementation of the RTI configuration and a VHDL wrapper for the usage of the RTI configuration in a signal-level environment. Furthermore, a list of files for compilation is generated that can be used as input for the Cadence irun tool in order to build a mixed-language and mixed-level simulation. For software, these files include a C/C++ description of the system memory map similar to the chip context format that is described in [5]. For each bus interface of each component instance, a base address is generated that reflects the position of that interface in the memory map.

In Step 3, an SCML property file is generated which is needed at run-time. This file contains configuration options that can be set at start of simulation.

Users can control which steps in the RTI configuration are executed. This avoids re-building their system if they only change values of run-time configuration options.

4. Case Study

We have applied RTI for randomized verification of a microcontroller IC. The IC contains an ARM Cortex-M3 processor and it has I2C, SPI, and UART interfaces. Our goal was to verify correct functioning of the IC when streaming data over these interfaces. Also DMA capabilities and power-down modes of the IC had to be verified. The acceptance criteria for randomized verification have been captured by 15 test cases which had to run continuously with different seeds for a period of at least two weeks without failures. Randomized verification is the last part in a hardware integration verification flow including formal interconnection verification using PSL assertions and directed tests for basic interoperability using verification software as targeted in the Hardware Verification Software Standard [5].

Figure 2 shows an overview of the Test Bench and DUT integration setup. The DUT is the IC. It is connected to the Test Bench via Test Bench pads. These pads are connected to I2C, SPI, and UART subsystems. These subsystems are connected to an RTI configuration via an AHB connection. For I2C there are two pads, namely serial clock SCL and serial data SDA. For SPI there are four pads, namely master-out-slave-in MOSI, master-in-slave-out MISO, serial clock SCK, and slave select (SSEL). For UART there are three pads, namely transmit data TXD, receive data RXD, and serial clock SCK for synchronous mode. The Test Bench contains four SPI subsystems but for simplicity these are not shown in Figure 2. Each SPI subsystem has its own Test Bench pads which are directly connected to the IC SPI pads. The SPI slave select signal is routed from the IC via one additional GPIO pad per SPI subsystem. These are also not shown in Figure 2. The DUT contains a single SPI interface. In SPI master mode, the DUT selects the corresponding SPI slave via the additional GPIO slave select connections.

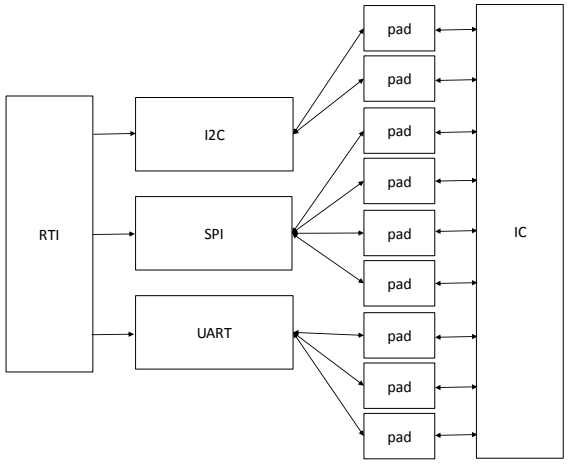

Figure 2. Test bench and DUT integration.

This system has been integrated using an IP-XACT-based flow. The IC has been handled as a black box IP. The Test Bench top-level and the I2C, SPI, and UART sub-systems have been integrated manually using the IP-XACT-based flow. The RTI subsystem has been generated automatically.

4.1 RTI Configuration

Table 1 shows the RTI configuration values for our case study which are input for Step 1. Row 1 and 2 specify which environment variables should be used.

Row 3 specifies the location of the RTI configuration that will be generated.

Row 4 specifies the HDL path of the RTI configuration instance. This path is needed to generate the SCML property file content.

Table 1. RTI configuration values for case study.

	Configuration option	Configuration value
1	projectWork	PROJECT_WORK
2	projectName	PROJECT_RTIGEN
3	cell	$PROJECT_WORK/data/ssd_chip_tb_lib/ssd_chip_tb_rti
4	hdlPath	.u_rti
5	vendor	nxp.com
6	library	ssd
7	name	ssd_chip_tb_rti
8	version	1.0
9	enable_debug	false
10	arm_frequency	100000000
11	quantum_value	0.00001
12	app_file_name	../VERIFICATION/apps/rtiSsd_clock_gen_basic/rtiSsd_clock_gen_basic.axf
13	timeOutEnable	true
14	timeOutUnit	1
15	timeOut	20000
16	romSize	0x20000000
17	ramSize	0x20000000
18	heapBaseAddress	0xFF00000
19	heapSize	0xE0000
20	stackSize	0x20000
21	numberOfInterrupts	26
22	numberOfAhbs	6
23	numberOfClocks	5
24	numberOfResets	2
25	numberOfCregOuts	10
26	numberOfCregIns	6
27	baseAddressesAhbs	"0x40000000 0x50000000 0x60000000 0x50000100 0x50000200 0xB0000000"
28	baseAddressesClocks	"0x70000000 0x70000200 0x70000400 0x70000600 0x70000800 0xC0000000 0x80000000 0x80000200 0x80000400 0x80000600 0x80000800"
29	baseAddressesResets	"0x70000100 0x70000300 0x70000500 0x70000700 0x70000900 0xC0000100 0x80000100 0x80000300"
30	baseAddressesCregOuts	"0x90000000 0x90000100 0x90000200 0x90000300 0x90000400 0x90000500 0x90000600 0x90000700 0x90000800 0x90000900"
31	baseAddressesCregIns	"0xA0000000 0xA0000100 0xA0000200 0xA0000300 0xA0000400 0xA0000500"
32	addressSpacesAhbs	"0x0 0x100 0x0 0x100 0x0 0x100 0x0 0x100 0x0 0x100 0x0 0x1000"

Rows 5-8 specify the IP-XACT Vendor, Library, Name, and Version of the IP-XACT component that will be generated for the RTI configuration.

Rows 9-12 specify the settings for the ARM FastModel Cortex-M3 model.

Rows 13-15 specify settings for the watchdog. The time out unit indicates the unit of the time out value (1 means nanoseconds). The given time out value is small to support license check out failures. In the software running on the processor we reprogram the watchdog for a larger time out value. If the processor does not start executing software because of a license check out failure, then the simulation is stopped early because of the small time out value.

Rows 16-20 specify the memory configuration. These values are used to configure the TLM memories and to generate a scatter file for the ARM linker.

Rows 21-26 specify the interfaces of the RTI configuration. For each AHB interface also a clock and reset component is generated. So the total number of clock and reset components in this example is 6+5=11 and 6+2=8, respectively.

Rows 27-31 specify the base addresses for the RTI component instances. Users can control these explicitly as in this example or they can leave that implicit in which case the RTI configuration process will automatically assign base addresses.

Finally, Row 32 specifies the AHB address offsets and ranges for each of the AHB interfaces. In this example, an access at address 0x40000000 by the Cortex-M3 model will result in an AHB address of 0x0 on the first AHB interface of the RTI configuration.

The configuration options are parsed and the information is used to generate an IP-XACT database. To this end, we have developed in-house generators. An IP-XACT TLM2 bus component is generated and an IP-XACT component, design, and design configuration are generated. The IP-XACT design configuration contains generator chain configurations with parameter values that are derived from the RTI configuration options. The IP-XACT database also contains IP-XACT component descriptions of the other RTI components. This database is the starting point for the source code generation in Steps 2 and 3 as described in Section 3.2.2.

4.2 Test Bench Integration

The next phase was to integrate the complete Test Bench. The RTI configuration from the previous section has become available as a VHDL signal-level IP block. We integrated it with the peripheral I2C, SPI, and UART RTL IP blocks to form the complete Test Bench show in Figure 2**.** Again, we used an IP-XACT approach. We used IP-XACT descriptions of the peripheral IPs which were either supplied by the IP provider or generated using our commercial packaging tool. IP-XACT design integration is a straightforward job using our commercial assembly tool. Figure 3 shows a schematic view of the top-level Test Bench. The I2C, SPI, and UART blocks are sub-systems containing the respective IP blocks and AHB-to-APB bridges. The DUT was handled as a black box meaning that we only described the pin interface and file set in IP-XACT. After IP-XACT design integration, the VHDL netlist generation is again straightforward using a commercial netlister.

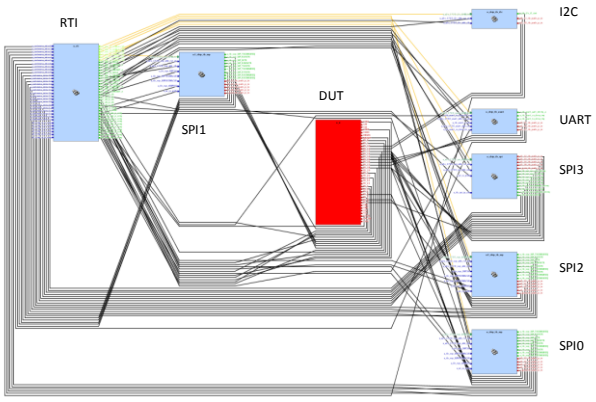

Figure 3. Schematic view of top-level Test Bench IP-XACT design.

4.3 Embedded Software Development

The final phase was to develop software for the Test Bench and for the DUT. Both contain a Cortex-M3 processor so we can re-use software on both ends. In the setup we ran independent software images. There is one Test Bench image and one DUT image.

The software architecture is based on queues. Each peripheral IP has a software TX queue and a software RX queue. Each queue element can contain a message which is a pointer to a stream of bytes. A stream of bytes has a certain structure: there is a header and a payload. The header contains the following elements:

- size (Byte 0), i.e., the number of bytes in the message,
- id (Byte 1), i.e., the id of the message,
- command1 (Byte 2), i.e., the source (Bit 0-2) and destination (Bit 3-5) IP in the Test Bench,
- command2 (Byte 3), i.e., the commands to be executed by the IC which are DMA memory copies to be done (Bit 0-1), DMA channels to be used (Bit 2-4), and power-down modes to be entered (Bit 5-7).
- source address (Byte 4-7), i.e., the start address of the message in the memory space of the DUT, and
- destination address (Byte 8-11), i.e., the start address of the message in the memory space of the DUT after a memory copy of the message by the DUT DMA.

The payload of the message is a sequence of upper case character byte values of random length. Also message headers are randomized. In this way, sequences of messages are streamed in and out of the DUT, verifying peripheral operations, processor operations, DMA operations, memory operations, and power-down modes. In the Test Bench, a shadow memory of the DUT memory is maintained such that it knows which memory locations are in use at any time. This is needed to constrain the randomization of the source and destination addresses. The Test Bench transmits and receives messages from and to the DUT and it verifies correctness by comparing transmitted and received messages.

The processors in the Test Bench and in the DUT execute read and write operations as well as full and empty checks on the queues of their peripherals. A key issue is that these queue operations are protected in critical regions such that the queue operations are atomic and the queue administration remains consistent. To this end, we disable and enable the relevant interrupts for each queue before and after each queue operation by means of the CMSIS NVIC disable and enable operations [2]. On the Test Bench side, the processor adds random messages to the transmit queues and removes messages from the receive queues. On the DUT side, the processor moves messages from receive queues to transmit queues after processing the commands in the messages.

The software on the Test Bench side has been developed on top of the register get and set functions that have been generated automatically from the IP-XACT register description.

4.4 Randomized Verification

The randomized verification setup consisted of 15 test setups which match with the 15 test cases defined in the acceptance criteria. Test cases specify amongst others the following items.

- The interfaces used to stream data to the DUT.
- The interfaces used to stream data from the DUT.
- The amount of DMA copies to be executed by the DUT.
- The power-down and wake-up sequence of the DUT.

On the Test Bench side we developed 15 software images to constrain the randomization process according the specifications of the 15 test setups. On the DUT side we developed 6 software images to program the DUT in different modes. The DUT images control whether the I2C and SPI interfaces run in master or slave mode (resulting in 4 different images). In 2 of these 4 images, we needed to control the activation of the serial interfaces from the Test Bench using GPIO events for some test setups. This results in a total of 6 images. Both on the Test Bench side and on the DUT side a large part of the code base is shared between the setups. The randomization only occurs on the Test Bench side. The DUT side is deterministic. The random seed is loaded into a designated ROM address before simulation start and it is pickup by the software to initialize the random number generator.

The random verification setup has run continuously for 4 weeks. In this period, 45000 seeds have been simulated while running the 15 test setups in parallel jobs. During these weeks, all tests have been developed further continuously. We spent 3 weeks developing the complete setup, without the RTI development time, before we were able to start running seeds continuously.

Figure 4 shows simulation results for one of the test setups. In this setup messages are streamed from the Test Bench UART into the DUT. After a DMA memory copy, the messages are streamed back to the Test Bench via SPI. One of the four SPI slaves is randomly selected for each message via four separate connections. The Test Bench compares the incoming messages with the outgoing messages to implement automated functional pass/fail checking.

Furthermore, the simulations provide performance numbers on throughput and latency of the messages. Simulations have shown that message streaming across the serial interfaces continues to function correctly with acceptable throughput and latency when DMA activity consumes an increasing amount of memory bandwidth inside the DUT.

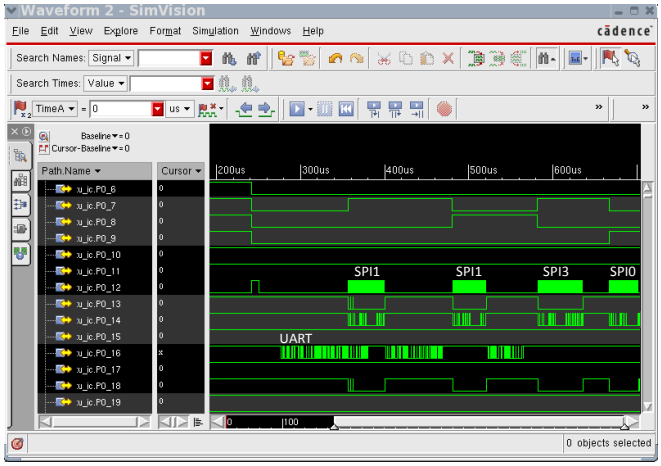

Figure 4. Waveforms showing streaming from TB UART (P0_16) via DUT back to TB SPI blocks (P0_14). The TB contains 4 SPI blocks numbered SPI0 to SPI3. Pins P0_6 to P0_9 are used as slave select for these blocks, respectively.

4.5 Summary

Here we summarize the flow that we used in our case study. First, we have generated an RTI configuration using Table 1 as input. For this RTI configuration the following data was generated automatically.

Meta-data

- IP-XACT component for the SystemC TLM2 bus, and
- IP-XACT component, design, and design configuration for the RTI configuration.

Hardware integration

- SystemC netlist for the RTI configuration,
- VHDL wrapper for the RTI configuration,
- SCML property file for the RTI configuration,
- Address decoding file for the SystemC TLM2 bus, and
- File set with SystemC and VHDL files for hardware compilation using irun.

Hardware component

- SystemC module with SCML register description for RTI clock generator, reset generator, timer, and register input and output.

Software integration

- C/C++ files describing the RTI system memory map,
- ARM scatter file, and
- Makefile for software compilation.

Software component

- C/C++ macros in vHAL format for register and bit field access on RTI clock generator, reset generator, timer, and register input and output, and
- C/C++ get and set functions for register and bit field access on RTI clock generator, reset generator, timer, and register input and output.

Documentation

- Data sheets for the RTI configuration and RTI components.

Second, we have integrated the test bench using IP-XACT technology, i.e., we have used our commercial GUI to generate IP-XACT components, designs, and design configurations for the test bench and the peripheral subsystems. From this IP-XACT database the following files were generated automatically.

Hardware integration

- VHDL entity, architecture, package, and configuration for the test bench top-level and peripheral subsystems.
- File set with SystemC, VHDL, and Verilog files for compilation of the complete test bench using irun.

Software integration

- C/C++ files describing the test bench system memory map including peripheral I2C, SPI, and UART base addresses.
- ARM scatter file for the complete test bench, and
- Makefile for software compilation.

Software component

- C/C++ macros in vHAL format for register and bit field access on I2C, SPI, and UART peripherals.
- C/C++ get and set functions for register and bit field access on I2C, SPI, and UART peripherals.

Documentation

- Data sheets for the I2C, SPI, and UART peripherals.

Third, we have developed embedded software for randomized verification on top of the automatically generated software. This includes drivers for I2C, SPI, and UART which are programmed on top of the generated register get and set functions.

5. RESULTS

One of the benefits of using IP-XACT is that all views for RTI configurations are generated using our existing IP integration flow. The work for generating specific RTI configurations was to script the generation of meta-data. All other views are generated using our existing IP integration flow. The development of the RTI components was additional work. Another benefit of using IP-XACT is that this standard is supported by EDA vendor tools which increase the productivity of flow automation as presented in this paper a lot.

The case study presented in this paper has taken 7 weeks by 2 persons. After 1 week, first results were presented showing streaming between UART and I2C. After 3 weeks, 15 test setups were started running continuously for 4 weeks. In these four weeks, the test setups have been refined further and have been debugged. A large part of the time has been spent on developing and debugging the message streaming software and programming of the peripherals because this was not available from earlier work. Test bench integration has been very efficient. As a result of this case study we can add I2C, SPI, and UART interfaces to the RTI infrastructure, such that they can be used immediately in future test benches. In 4 weeks, more than 45000 seeds have been simulated using 15 parallel jobs. This averages to 15 minutes per seed. Simulation speeds are identical to RTL simulation speeds.

Another observation is that the presented test infrastructure easily accommodates integration of software-driven verification IP. Such verification IPs typically have register interfaces via bus connections. The ARM FastModel can be used to run software that drives the verification IPs. This approach combines IP and system-level verification.

6. CONCLUSIONS

In this paper, we have demonstrated efficient test bench integration that can be achieved using IP integration and flow automation using IP-XACT technology. Also we have applied IP-XACT technology for configuration of test benches. The same technology can be applied to configure systems and subsystems. The configurability has been implemented at the IP-XACT meta-data level. The configuration process produces a configured IP-XACT design. On this configured design, a "traditional" IP-XACT flow is executed producing the relevant implementation views.

7. REFERENCES

[1] ARM FastModels, http://www.arm.com/products/tools/models/fast-models.php

[2] ARM Cortex Microcontroller Software Interface Standard, http://www.arm.com/products/processors/cortex-m/cortex-microcontroller-software-interface-standard.php

[3] Cadence Incisive Enterprise Simulator, www.cadence.com/products/sd/enterprise_simulator

[4] Carbon SOC Designer, http://www.carbondesignsystems.com/soc-designer-plus/

[5] CoReUse 4.6, Hardware Verification Software Standard, http://www.ip-extreme.com/MediaFiles/ebooks/26.pdf

[6] Esterel, http://www.esterel-technologies.com/

[7] Magillem Design Services, http://www.magillem.com/eda/

[8] Mentor HDL Designer, http://www.mentor.com/products/fpga/hdl_design/hdl_designer_series/

[9] Mentor Vista, http://www.mentor.com/esl/vista/upload/vista-architect-ds.pdf

[10] IP-XACT, Standard Structure for Packaging, Integratimg, and Reusing IP within Tool Flows, IEEE 1685-2009, http://standards.ieee.org/getieee/1685/download/1685-2009.pdf

[11] SystemC Modeling Library, http://www.synopsys.com/cgi-bin/slcw/kits/reg.cgi

[12] SystemC, Open SystemC Language Reference Manual, IEEE 1666-2011, http://standards.ieee.org/getieee/1666/download/1666-2011.pdf

[13] SystemC TLM2, http://www.accellera.org/downloads/standards/systemc/tlm

[14] Synopsys Core Assembler, http://www.synopsys.com/dw/ipdir.php?ds=core_assembler

[15] Synopsys Virtualizer, http://www.synopsys.com/systems/virtualprototyping/Pages/default.aspx

[16] Universal Verification Methodology, http://www.uvmworld.org/