2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION INDIA

Verifying the I/O peripherals of OpenTitan SOC using the Portable Stimulus Standard

Mahesh R, Bidisha Das and Raj S Mitra, Ph.D.,
Cisma Consultants Pvt Ltd, Bangalore

Loganath Ramachandran Ph.D., Viraphol Chaiyakul Ph.D.
Accelver Systems Inc, Sunnyvale, CA

Abstract: The integration of complex design blocks (IPs) into modern SoCs has increased its verification challenges, thus decreasing the possibility of meeting time to market goals. Verification teams are yearning for more automated solutions to tackle this complexity better. Portable Stimulus Standard (PSS [1]) was introduced by Accellera as a standard language to capture verification intent with the potential to automate test case generation across various platforms. In this paper we explore the potential of PSS to generate "C" testcase scenarios for an open-source complex SOC design, and propose a systematic approach to implement a set of SoC data transfer tests across its peripheral blocks.

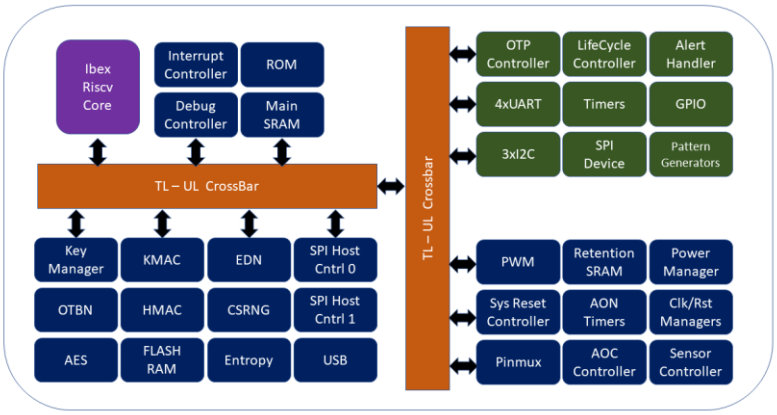
I. INTRODUCTION

Typical SOCs are comprised of many design blocks (or IPs) that communicate with each other over one or more central (and standard) buses. Although each IP is assumed to be verified in terms of functionality, the SoC verification tasks by itself can still be very daunting. Interconnect verification, register validation, data transfer validation, and power validation are some examples of SOC level verification tasks. These tasks require the creation of complex scenarios where each scenario targets a given verification goal. In this paper we propose a systematic approach using PSS to plan and implement a set of SoC data transfer tests across the peripheral blocks. An example of a complex SoC is the OpenTitan [2], an SOC developed by Google with the tagline "OpenTitan is the first open-source project building a transparent high quality reference design and integration guidelines for silicon root-of-trust chips".

Open Titan's Earl Grey chip is a low power secure microcontroller that is designed specifically for hardware security applications. The functional specification and the details of the IP are provided at the OpenTitan website. The block diagram shows the (a) the high-level architecture, (b) the different IPs that are integrated into the SOC and (c) the interconnection architecture.

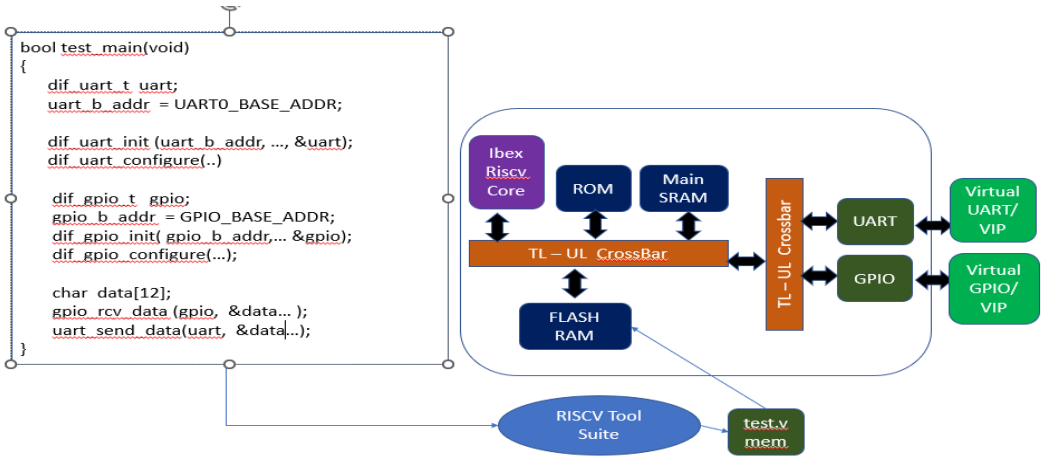
OpenTitan is based on RISC-V Ibex core [3] which has a 3-stage pipeline. The platform includes a RISC-V compliant JTAG debug module and a platform level interrupt controller. The security features of the design include memory scrambling on the icache level. There are 2 banks of FlashRAM and 128kB of sRAM, in addition to a 4KB retention SRAM. The boot code is stored in a 32Kb ROM.

The focus of this paper is the I/O peripherals in the design. The SOC contains (a) 32-bit GPIO, (b) four UART peripherals, (c) three i2C peripherals with host and device modes. (d) SPI device controller and (e) USB interface. We will also focus on bare-metal applications and application tests, although the approach can be easily extended if we have to work with a setup where the operating system is also involved.



2. DATA TRANSFER TESTS

The SOC platform is the first physical platform where different IPs are stitched together. Hence it is the first platform where data transfer across the peripherals can be observed. Let us take a simple scenario where we would like to send a piece of data into the SOC from the GPIO peripheral and output the same data onto a UART peripheral. The overall approach to these tests is summarized in the following figure. Here the "C" testcase initializes the peripheral components and configures them with a specific configuration parameter. For example, the baudrate of the UART can be set in the configuration phase. Then the data transfer begins where the test waits to receive data from the GPIO peripheral and stores it in some location in memory. Then the same data is sent out on the UART peripheral.



3. OVERVIEW OF DATA TRANSFER TEST INTENT

Based on the above simple example we can write several data transfer tests that carry data from one peripheral to another.

1) Test cases are written in "C" and compiled using the RISC-V tool suite. The compiled elf/vmem files will be loaded onto the FlashRAM.
2) Once the simulation is started the CPU starts with the boot code in the ROM and then starts executing the instructions in the FlashRAM.
3) Although these tests can be written from scratch, it is always preferable to create an abstraction layer called Hardware Abstraction Layer which are a set of APIs for each peripheral. For example, dif_gpio_configure() configures the gpio peripheral based on the parameters.
4) Data transfer tests typically would read data coming in on peripheral A, store the incoming data into memory and then write the same data on peripheral B.
5) A VIP or virtual peripheral can be used to generate data on peripheral A and read the output data from peripheral B.
6) A scoreboard or checking mechanism can be connected to the VIPs to ensure that data correctness is met.
7) A large number of testcases can be written by selecting the following
    a. Select peripheral A and select peripheral B from the list of available peripherals. There are several combinations in the OpenTitan itself.
    b. Select the size of the data for the test.
    c. Select the configuration parameters of the peripheral from a list of legal parameters.
    d. Select the memory location to store the incoming data and send it out on the outgoing peripheral B.

In summary a large number of scenarios need to be created to get sufficient confidence in the SOC. However, this is a labor-intensive process as each of these test cases, have to be written by hand, compiled, debugged and run on the SOC [7]. Hence any automation in this area would be extremely beneficial. Enter PSS!

4. PORTABLE STIMULUS STANDARD (PSS)

Portable Stimulus Standard was created by Accellera to capture the verification intent. It is a Domain Specific Language intended to capture the verification intent. The PSS model also generates different outputs based on the platforms that are being targeted. PSS serves as an ideal in this project where a large variety of test cases need to be created. Some of the main advantages of PSS include:

a) The constraint specification in PSS is on par with SystemVerilog [4] . It can be used to generate constrained random scenarios with ease.
b) PSS actions support inheritance (OOP) and extensions (Aspect-Oriented programming). This makes it easy to extend PSS components for newer projects where the peripherals may have additional legal configurations.
c) PSS is the one of the few languages that provides control-flow randomization. Multiple random traversals through an activity graph in PSS produces widely varying scenarios.
d) PSS also provides flow-object (data) binding. This capability allows us to specify constraints such that the output data of one action is bound to the input data of another action, providing data transfer capabilities that can be randomized.
e) PSS provides target language portability. Although we are using it only to generate "C" code for this project we may think about different flavors of "C" (based on the compilers) and different HAL layer functions being used for simulation vs emulation.

We have relied heavily on the Tutorials provided by Accellera [5] [6] when designing this methodology.

5. OUR SOLUTION OUTLINE

Using PSS for SOC level "C" test generation is a very useful methodology and can be applied to any Sock We focus on the OpenTitan project because of two reasons (a) it has a well-defined HAL layer for each of the peripherals and (b) the SOC is available in open-source and it is easy to compile and run our tests on the SOC.

A. Device Interface functions (HAL layer)

Each peripheral in OpenTitan has a good set of well-defined and well documented APIs which makes it easy to build our solution. Some examples of such basic functions are shown here. Using PSS for SOC level "C" test generation is a very useful methodology and can be applied to any SoC. The following figure shows some of the functions available for the UART peripheral.

```
dif_uart_result_t dif_uart_bytes_send              (const dif_uart_t *uart, ..... )
dif_uart_result_t dif_uart_byte_send_polled        (const dif_uart_t *uart, ..... )
dif_uart_result_t dif_uart_bytes_receive           (const dif_uart_t *uart, ..... )
dif_uart_result_t dif_uart_byte_receive_polled     (const dif_uart_t *uart, ..... )
```

B. Creating PSS Actions

The basic building block in PSS is called an action. We group the C-APIs based on functionality and wrap them into actions. For example, the *vk_uart_configure_a* action shown in the following figure demonstrates the action needed to configure the uart. PSS can have properties that are randomizable. For the uart we declare two such random variables (i.e, clock_frequency and baud_rate), When a PSS Action is randomized, these variables will assume new values based on the constraints applied. Here we show two dynamic constraints for both these variables.

```
action vk_uart_configure_a {
    rand int baudrate;
    rand int clk_fz;

    dynamic constraint default_baud { baudrate == 2697;}
    dynamic constraint default_clk  { clk_fz == 240;}

            ..............

    exec body C = “””
            CHECK(dif_uart_configure(&uart, (dif_uart_config_t)
                {.baudrate = {{baudrate}},  .clk_freq_hz = {{clk_fz}},
                 .parity_enable = kDifUartToggleDisabled,
                 .parity = kDifUartParityEven,
                }) == kDifUartConfigOk, "UART config failed!");
            “””;
}
```

PSS actions can have an exec body section which specifies the code to be generated when the action is traversed. The code is written within a moustache notation and contain references to PSS variables which are dynamically solved. The above action generates the uart HAL functions for *dif_uart_init* and *dif_uart_configure*. The baudrate and clock_freq variables are internal PSS variables whose values may be used in the exec_body to customize the generation. These actions are called Atomic Actions in PSS.

C. Creating PSS components

All PSS actions for a given peripheral is encapsulated into a PSS component. The PSS component for UART will contain all the actions necessary for scenario creation with the UART peripherals. At a very high one can view this component like a reusable VIP and can be moved to other UART related projects within the OpenTitan family of SOCs. The following figure illustrates the UART PSS component.

```
component  vk_uart_c {
    action  vk_uart_configure_a {
        exec  body  C  =  “””.....””” ;
    }
    action vk_uart_rand_data_send_a { ..... }
    action vk_uart_rand_data_receive_a { ..... }
    action vk_uart_read_and_send_a {
                        input  addr_buf  buf_i ;
                        .....
     }
     action  vk_uart_receive_and_store_a { ..... }
}
```

D. Top Level SOC component

At the SOC level it is necessary to carry out global actions that may span one or more peripherals. These actions are introduced at the SOC level component. In the following figure we implement a system wide action called *vk_configure_all_peripherals_a*.

```
component vk_top_earlgrey_c {
    vk_uart_c  uart_c_inst ;
    vk_usb_c   usb_c_inst ;
    vk_spi_c    spi_c_inst ;
          ....................
    action vk_configure_all_peripherals_a {
        vk_uart_c::vk_uart_configure_a   uart_conf_inst;
        vk_usb_c::vk_usb_configure_a    usb_conf_inst;
        vk_spi_c::vk_spi_configure_a     spi_conf_inst;
       activity {
            uart_conf_inst with { default_baud; clk_fz == 3625; };
           usb_conf_inst with { ..... };
           spi_conf_inst  with { ..... };
                                .....
       }
}
```

In this action we instantiation all the actions from all the other components and traverse these actions in a given order. This type of action is called a compound action as it invokes one or more other actions using control flow ordering. In these compound actions, customized parameter values for the action instances are provided with inline constraints using the “with” keyword. The default constraints which are dynamic in nature will be enabled by passing the constraint label as inline constraint.

E. Modeling Data Movement with a PSS activity

Data movement functionality across peripherals is represented by a compound PSS action. These actions internally traverse other actions that may move data from one part of the SOC to another. In the figure shown the compound action reads some data from internal memory and transfers it to the TX ports of the UART block.

```
component pss_top {
    vk_mmio_c       mmio_c_inst;
    vk_uart_c       uart_c_inst;

        // scenario of UART sending data loaded in memory
        action uart_mem_data_send {
            vk_uart_c::vk_uart_configure_a          uart_conf_inst;
            vk_mmio_c::vk_load_mmio_a               mem_ld_inst;
            vk_uart_c::vk_uart_mem_data_send_a      uart_send_inst;

            rand int data_addr;
            rand string   data;
            int size;
             activity {
                    uart_conf_inst with {default_baud; default_clk; }
                    mem_ld_inst with { start_addr == data_addr;   info == data; };
                    uart_send_inst with { mem_addr == data_addr; data_size == size};
             }
        }
}
```

The first action instance in the activity does the configuration by considering the default configuration values. An action instance of a memory-management is used to load the provided data into a given address, the values are passed using the (inline) with-constraints. At the last there will be a traversal of the UART action to which reads the data of a specified size from the given memory address. The read data will be written into the UART write-fifo to transmit the same.

F. Creating SOC level scenarios

We construct a PSS model to represent a given scenario. The PSS model is created using the following steps:

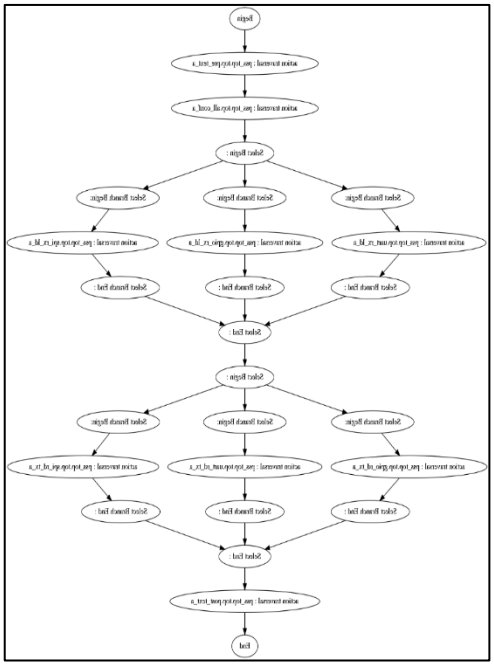
(a) Component instances are created for all the peripherals in the SOC.
(b) A PSS buffer *vk_data_buf* is declared. An object of this type is bound to the send and receive actions.
(c) A PSS pool *data_p* is used to hold the buffer objects. This pool is made visible to all the PSS components and actions.
(d) Action instances are created as required for constructing this scenario.
(e) Initially, a PSS action to configure all the peripherals is traversed. Constraints can be added during this traversal using the “with” statement.
(f) A select statement is used to choose a source peripheral randomly. This peripheral is used for receiving the data from the external ports and storing the data in memory.
(g) Size of the data is determined by randomizing the data-flow object (*vk_data_buf* type). This object is mapped to the output port of the receive action.
(h) A select statement is used to choose a target peripheral randomly. This peripheral is used for sending the data (read from memory) to the external ports. The same data flow object (mentioned in g) is mapped to the port of this action

(i) The *pre_text_a* and *post_text_a* will take care of adding header and trailer codes to each of the generated testcases. This is required to complete the automated scenarios.

```
component pss_top {
    vk_top_earlgrey_c   top_eg_inst ;
    vk_uart_c           uart_c_inst ;
    vk_gpio_c           gpio_c_inst ;
    vk_spi_c            spi_c_inst ;

    buffer  vk_data_buf { rand  int  data_size; }

    pool  vk_data_buf  data_p;
    bind   data_p  * ;
      .............
      .............
    // scenario
    action top {
        vk_top_earlgrey_c::vk_configure_all_peripheral_a    all_conf_a;
        vk_gpio_c::vk_gpio_read_and_send_a                  gpio_rd_tx_a;
        vk_gpio_c::vk_gpio_receive_and_store_a              gpio_rx_ld_a;
        vk_uart_c::vk_uart_read_and_send_a                  uart_rd_tx_a;
        vk_uart_c::vk_uart_receive_and_store_a              uart_rx_ld_a;
        vk_spi_c::vk_spi_read_and_send_a                    spi_rd_tx_a;
        vk_spi_c::vk_spi_receive_and_store_a                spi_rx_ld_a;
        vk_pre_text_generator_a                             pre_text_a;
        vk_post_text_generator_a                            post_text_a;

         activity {
                pre_text_a;
                all_conf_a;
                select {
                      uart_rx_ld_a;
                      gpio_rx_ld_a;
                      spi_rx_ld_a;
                  }
                  select {
                       gpio_rd_tx_a;
                       uart_rd_tx_a;
                       spi_rd_tx_a;
                   }
                  post_text_a;
             }
      }
}
```

For example, UART [0]→mem [2098]→UART [1] could be one of the generated tests from this scenario. Other scenarios can be created using **inference** a unique feature of PSS. By inferring a mem-to-mem transfer action, the following scenario can be automatically created.
UART [0]→mem [2098]→mem [1094] → UART [1]
But for this paper we did not enable the inferencing feature of PSS.

The above PSS scenario was processed by an internal tool to produce a control flow graph shown in the next figure



.

7. SUMMARY/ RESULTS

The scenario generator can generate all combinations of data transfer across peripherals. The select construct enables control flow randomization to produce all possible data transfers making the PSS approach compact. Also, it is scalable as new peripherals are added to the mix. Here is an example of a generated testcase which compiles and runs on the OpentTitan RTL:

```
bool test_main(void) {
    dif_uart_t uart;
    CHECK(  dif_uart_init( (dif_uart_params_t)
    {
                .base_addr = mmio_region_from_addr(TOP_EARLGREY_UART0_BASE_ADDR),
    },  &uart) == kDifUartOk);

    CHECK(dif_uart_configure(&uart,
                    (dif_uart_config_t){
                       .baudrate = kUartBaudrate,
                       .clk_freq_hz = kClockFreqPeripheralHz,
                       .parity_enable = kDifUartToggleDisabled,
                       .parity = kDifUartParityEven,
                    }) == kDifUartConfigOk,
                    "UART config failed!");
    CHECK(dif_uart_fifo_reset(&uart, kDifUartFifoResetAll) == kDifUartOk);

    static const uint32_t kGpioMask = 0x0000FFFF;
    dif_gpio_t gpio;
    CHECK(dif_gpio_init(
          (dif_gpio_params_t){
             .base_addr = mmio_region_from_addr(TOP_EARLGREY_GPIO_BASE_ADDR),
          },
          &gpio) == kDifGpioOk);

    CHECK(dif_gpio_output_set_enabled_all(&gpio, kGpioMask) == kDifGpioOk);

    static uint8_t kData[10];
    for (int i = 0; i < 10; ++i) {
          uint8_t receive_byte;
         CHECK(dif_uart_byte_receive_polled(&uart, &receive_byte) == kDifUartOk);
          kData[i] = receive_byte;
     }

     for (int i = 0; i < 10; ++i) {
         CHECK(dif_gpio_write_all(&gpio, kData[i]) == kDifGpioOk);
     }

     return true;
}
```

7. REFERENCES

1) Portable Test and Stimulus Standard - Accellera Systems Initiative- https://www.accellera.org/downloads/standards/portable-stimulus

2) OpenTitan project, https://www.opentitan.org

3) Ibex core project, https://github.com/lowRISC/ibex

4) IEEE Standard for SystemVerilog – Unified Hardware Design Specification, and Verification Language, IEEE Std 1800-2017, https://ieeexplore.ieee.org/document/8299595

5) Accellera Tutorial presented at DVCON US 2020, Portable Stimulus : What's coming in 1.1 and what it means for you. https:// ww.accellera.org/resources/videos/portable-stimulus-tutorial-2020

6) Accellera Tutorial presented at DVCON US 2018, Portable Test and Stimulus: The next level of verification productivity is here, https:/www.accellera.org/resources/videos/pss-tutorial-2018

7) Mahesh R, Shamanth HK, "Verification of the PULPino SoC platform using UVM" presented at RISC-V Workshop at IIT Chennai, 2018