Co-Developing Firmware and IP with PSS

M. Ballance
Siemens Digital Industries Software
8005 SW Boeckman Rd
Wilsonville, OR, 97070

***Abstract*- Runtime-configuration and operation of design IP is increasingly dependent on firmware. However, firmware for those IPs is often created too late, and in an unsuitable form, to be helpful in SoC-bringup tests. This presents an obstacle to creating SoC integration tests, and often results in late discovery of hardware/software interaction issues. This paper proposes an Accellera Portable Test and Stimulus (PSS) -enabled flow for co-developing and co-verifying design IP and firmware.**

I. Introduction

Runtime-configuration and operation of design IP is increasingly dependent on firmware (low-level software). A side effect of this operational dependency is that SoC integration testing also depends on the availability of firmware for the IPs within the SoC.

Accellera Portable Test and Stimulus (PSS) [1] is a language used for defining model-based tests, with specific features that target the concerns of creating SoC integration tests. Using PSS to create tests boosts test-creation productivity, but a PSS-based test description is still reliant on the existence of low-level firmware to interact with IPs within the system.

Unfortunately, this lowest-level firmware for IPs is often developed independently of the IP development and verification process. This means that it is often not ready at the point when, ideally, it could be leveraged along with PSS in creating SoC-integration tests. Firmware is often developed as production software targeting a production operating system. This means that it is even challenging to leverage production firmware created for similar (eg older) IPs, since the software environment used for SoC-integration testing is more resource constrained than the production software environment.

There are several unfortunate consequences of the current situation. Hardware/software integration issues are discovered late when they could be discovered while the IP was being designed and verified. SoC-integration testing is impeded by a lack of readily-available low-level driver firmware. This often leads to the SoC-bringup team writing their own low-level drivers which is both wasteful and error-prone, and delays when PSS can productively be deployed to create more-complex test scenarios. Ideally, of course, we want the SoC firmware platform to be ready and stable as soon as the SoC IP is integrated, and be ready to deploy PSS for expanded test coverage.

II. Goals and Requirements

Before describing the elements of an approach that will enable device driver firmware to be developed, verified, and delivered along with IP, it is well worth delving into overall goals and requirements. As with many things, let's look at the flow starting with the end in mind.

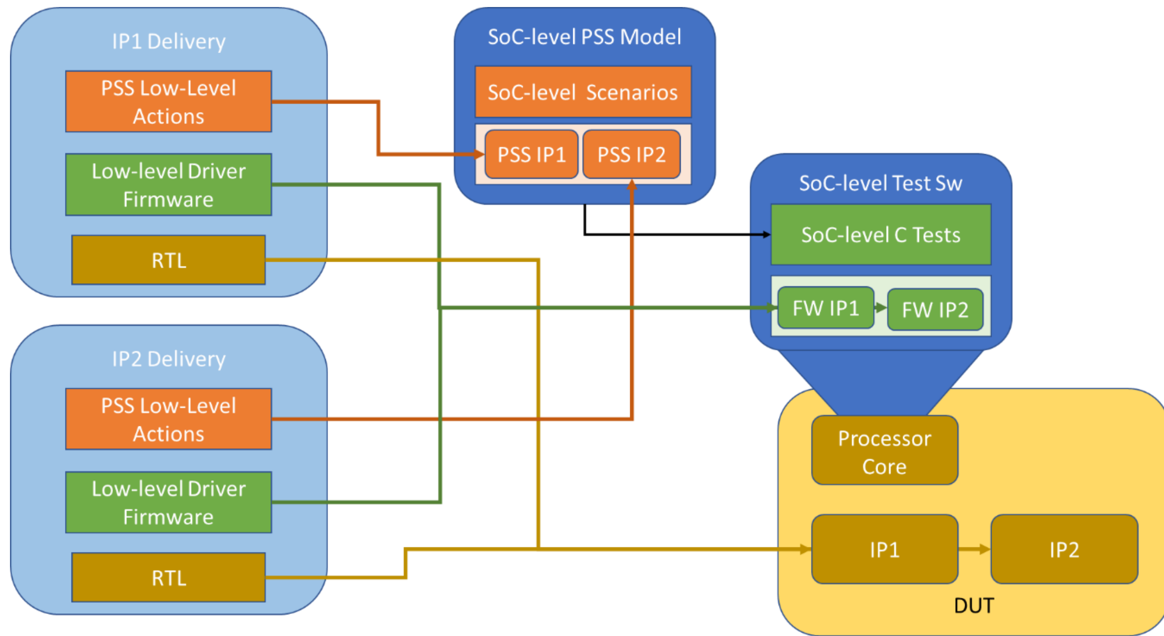

Figure 1 - SoC-level verification environment

Figure 1 shows an SoC-level verification environment and the reusable elements that form the design and verification environment in our idealized early-firmware-development flow. Our goal is to enable the IP design and verification teams to not only deliver fully-verified RTL that can be incorporated into the SoC, but to also deliver test content and firmware that can be rapidly assembled and form the basis for SoC-level tests. The flow shown above is PSS-enabled, which means that each IP requires some IP-specific PSS model content that enables scenarios involving that IP to be created. Our goal is for IP-specific driver firmware and PSS test content to be developed and delivered as part of the full IP DV-team's deliverable. It is also important that the driver firmware and PSS test content easily integrate at the SoC level.

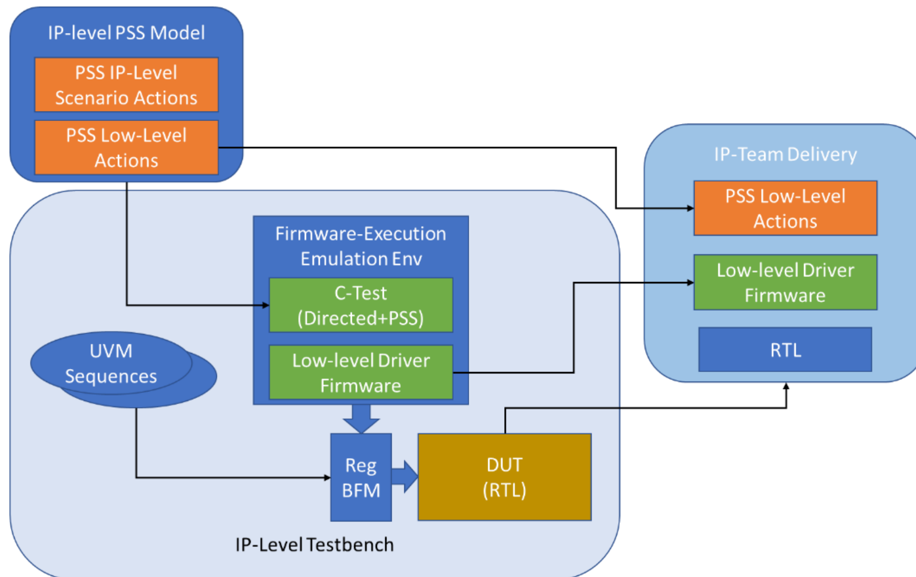

Figure 2 - IP-Level Verification Environment

Figure 2 shows the IP-level verification environment from our idealized early-firmware development flow. In many respects, there is little change from a standard UVM-based environment. Verification IP is used to interact with the design RTL via its interfaces. UVM sequences drive stimulus using the verification IP, scoreboards are used to check

results, and functional coverage is used to ensure that key cases are exercised. These aspects of IP verification remain in place, and remain key to achieving high-quality design RTL.

What changes is the introduction of software-driven tests that interact with the environment via low-level driver firmware, which will be provided as part of the comprehensive IP deliverable. Also added is PSS low-level test content that will be provided along with the IP, and PSS test content that is specific to the IP level and will not be delivered with the IP. A key goal in this process is to simplify the process of developing and testing low-level driver firmware such that the IP design and verification team is easily able to capture their knowledge of how to interact with the IP.

III. Key Challenges and Design Decisions

One of the larger challenges in realizing the solution shown above isn't connecting software behavior into an IP-level verification environment. It is encapsulating and packaging driver firmware and test content such that it can easily be incorporated, along with driver firmware and test content for other IPs, at the SoC level. There is a not-insignificant history [2] [3] of bringing some aspects of low-level software into verification environments via simulation interfaces, such as the SystemVerilog DPI. These solutions deal with how firmware for a single device is incorporated into an IP-level simulation environment, but do not deal combining firmware for multiple devices at the SoC level. At the SoC level, firmware modules for multiple IPs and for multiple instances of a given IP must be integrated and configured. For example, different driver instances must be configured to use different base addresses and must be connected to different interrupts. And, of course, the resulting software image must run with sufficient speed on the RTL implementation of the design cores to enable integration and performance verification in a simulation or emulation environment.

Bare-metal software is most-commonly used for SoC-level integration tests. Bare-metal software provides very few services, such as preemptive thread scheduling or memory allocation, to the software author. Its key advantage is that it provides full control over the hardware and imposes little memory, performance, and structural overhead. In order to enable driver code to easily integrate and interoperate, we will need to impose some structure and conventions on the driver code. This paper proposes the use of an existing RTOS (the Zephyr RTOS) instead of a new bare-metal driver-integration framework because it provides the required driver integration structure, while retaining many of the memory and performance characteristics of bare-metal software. The Zephyr Project [4] develops a highly-configurable Real-Time Operating System (RTOS) kernel targeting resource-constrained devices. Zephyr is available under a permissive open-source license, and the project is used and sponsored by a wide variety of non-profit and commercial entities. As a side-effect of its focus on resource-constrained devices, Zephyr imposes very little overhead compared to pure bare-metal software running without an OS or RTOS. This is important for SoC-level testing, since SoC-level tests must be able to have a high degree of control over interactions in the system – something that full-OS kernels may interfere with. Zephyr can also be configured to have a very small footprint (~8KB [5]) and has a very minimal boot process. Due to these factors, using Zephyr during the SoC bring-up test process imposes very little overhead on the verification process.

A huge benefit of reusing an existing RTOS for verification vs designing a custom framework for software-driven verification is an ability to leverage the existing RTOS ecosystem. Zephyr has a sizable collection of member companies and organizations, well-maintained documentation, tutorials by third-party developers, and existing developers experienced with the RTOS.

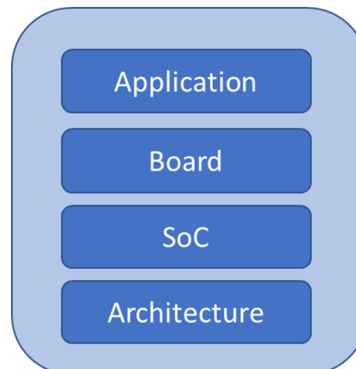

Figure 3 - Zephyr Modular Architecture

One very attractive aspect of the Zephyr RTOS from a verification perspective is that it is very modular and configurable. Due to its requirements to support a wide variety of processor architectures and devices, the Zephyr RTOS has a layered architecture (Figure 3) that enables the RTOS implementation and included device drivers to be easily changed as needed without impacting other aspects of the RTOS stack. For example, we can easily change the targeted processor architecture without impacting how device drivers are included in the OS.

Due to its focus on deeply-embedded systems, the Zephyr RTOS has a different application structure than what one might typically think of when thinking about an operating system. When compiling application code for a traditional operating system such as Linux, the output is an executable image that is independent of the operating system image. Specifically, the application executable is executed by the running operating system. When compiling application code for Zephyr, the output is a single executable image that includes the application, operating system kernel, and driver code. This structure works well for deeply-embedded systems that typically execute a single application from non-volatile storage, and do not have an external filesystem from which to load applications. This structure also works well for bare-metal SoC-bringup tests due to its simplicity. The monolithic-executable model enables Zephyr to perform nearly all initialization activity, such as creating driver instances, statically at compile time, which contributes significantly to Zephyr's small memory footprint and fast startup time.

Targeting Zephyr OS to an IP-level Environment

Zephyr supports a target architecture named *posix* that implements the architecture layer using POSIX API calls. Using the *posix* architecture, all the RTOS layers shown in Figure 3 are compiled into a single executable that runs on the host workstation. As the documentation states [6], this architecture is primarily intended for testing by application developers. However, we can leverage Zephyr's support for the *posix* architecture to run a test program (the application) that uses driver code to interact with an IP in the context of an IP-level verification environment.

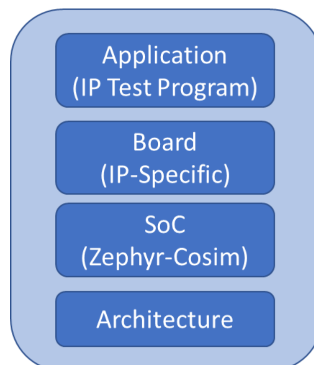

Figure 4 - IP-Level Zephyr Image

Due to its modularity, configuring the Zephyr RTOS to co-simulate with an IP-level testbench environment requires no changes to the core of the RTOS. Figure 4 shows the components of the Zephyr executable that can co-simulate with a UVM testbench environment. The application, in this case, is the test code that interacts with the IP via the low-level driver being developed. The *board* layer is created using a template provided by the Zephyr-Cosim package [7], and properly configures the driver-under-development and causes it to be included in the Zephyr executable image. The *SoC* layer intercepts calls to the Zephyr functions used by the driver code to access IP registers, and redirects them to the UVM testbench. The *SoC* layer is entirely provided by the Zephyr-Cosim package. The *Architecture* layer is provided by the Zephyr *posix* architecture code.

Connecting Zephyr OS to a UVM Environment

One of the most-popular methods for incorporating native-compiled code into a SystemVerilog simulation is to compile the code as a shared library that can be loaded by the simulator and accessed via the SystemVerilog Direct Programming Interface (DPI). Due to the structure of the Zephyr executable, and some of the link-time optimizations that enable small code size and static initialization, taking this approach is not feasible. Zephyr-Cosim, instead, integrates with the simulator via the *tblink-rpc* package [8]. *tblink-rpc* is a generic co-simulation library that allows remote procedure call (cross-calling) between simulation-external (ie non-SystemVerilog) code and testbench code

running in the simulator. The *tblink-rpc* package supports co-simulation with standalone executables, which is how we will connect the Zephyr-based test and driver code to our simulation.

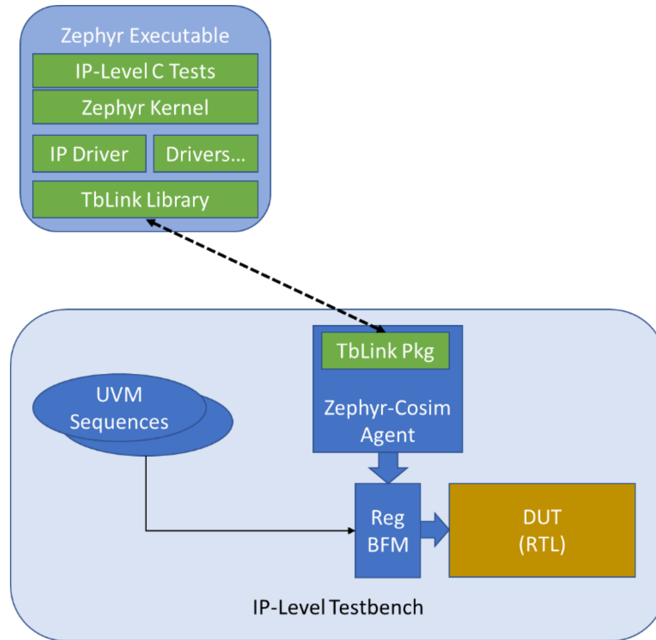

Figure 5 - Zephyr-Cosim Simulation Integration

Figure 5 shows a diagram of the co-simulation between a UVM testbench and the Zephyr-based software test application. The Zephyr-Cosim package provides a UVM agent that manages the Zephyr software executable, and interacts with the testbench using a UVM register adapter [9] connected to the UVM agent connected to the register interface.

One benefit of running software compiled for the host workstation instead of emulating execution of target-architecture code is that host-platform debug and analysis tools can be used on the driver code under development. For example, the memory-checking tool Valgrind [10] can be used to detect memory-access bugs. Familiar software-development tools, such as the GNU Debugger (GDB), Eclipse, and VSCode may all be used to debug logic bugs.

IV. The Device Driver Package

The Zephyr RTOS supports developing device drivers and other OS components outside the main OS tree. This capability is very helpful in accomplishing our goal of co-developing and co-locating driver firmware with the RTL IP. There are three elements that we will group together as part of the firmware and test-content package: driver meta-data, driver firmware, and PSS test content.

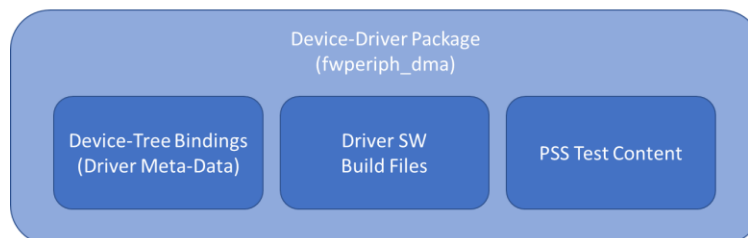

Figure 6 – Device Driver Package Elements

The Zephyr RTOS uses the Devicetree Specification [11] to specify what devices exist in a given system, and to configure the drivers used to access those devices. The Device-Tree Bindings portion of the device driver package captures configurable aspects of the device, such as the interrupts (if any) that it generates. The core of the driver is

the driver software and build files that enable the driver software to be incorporated in a Zephyr image. Reusable PSS elements are provided to simplify SoC test creation.

Device Driver Structure

At the IP level, we will focus on a driver for a DMA engine. The Zephyr RTOS defines several standard driver-driver APIs, but also allows device drivers to define their own custom APIs. When writing a device driver to use in production software, there will often be little reason to use a custom API. It's almost always better to make the device look, and the driver behave, like something familiar. However, when writing device drivers for the purpose of SoC level testing, it can be beneficial to leverage custom APIs to exercise operational details that may not be relevant to production software.

```
/**
 * @brief Configure individual channel for DMA transfer.
 *
 * @param dev     Pointer to the device structure for the driver instance.
 * @param channel Numeric identification of the channel to configure
 * @param config  Data structure containing the intended configuration for the
 *              selected channel
 *
 * @retval 0 if successful.
 * @retval Negative errno code if failure.
 */
int dma_config(const struct device *dev, uint32_t channel,
                                        struct dma_config *config);

/**
 * @brief Reload buffer(s) for a DMA channel
 *
 * @param dev     Pointer to the device structure for the driver instance.
 * @param channel Numeric identification of the channel to configure
 *              selected channel
 * @param src     source address for the DMA transfer
 * @param dst     destination address for the DMA transfer
 * @param size    size of DMA transfer
 *
 * @retval 0 if successful.
 * @retval Negative errno code if failure.
 */
int dma_reload(const struct device *dev, uint32_t channel,
                        uint32_t src, uint32_t dst, size_t size);
```

Figure 7 - DMA Device Driver API

Zephyr defines a standard API for DMA devices, which is sufficient for our verification purposes. Figure 7 shows two of the public-interface functions along with documentation. Our task in developing a driver is to implement the device-specific 'back-end' of these public functions by accessing the registers of our specific IP. Figure 8 shows our driver's implementation of the *reload* function that configures a DMA channel to carry out a transfer.

One thing to observe is that the driver-code complexity is on-par with the corresponding code that we would expect to see in a UVM testbench to configure the DMA. A design and verification team that has implemented UVM sequences to program the DMA engine in the testbench has the device knowledge required to create such low-level driver firmware.

```
static int fw_periph_dma_reload(
        const struct device     *dev,
        uint32_t                channel,
        uint32_t                src,
        uint32_t                dst,
        size_t                  size) {
    const fw_periph_dma_cfg_t *const dma_cfg =
                                (const fw_periph_dma_cfg_t const*)dev->config;
    uint32_t sz;
    // Updates the source/dest/size for a transfer, while leaving
    // the rest of the configuration as-is

    // Configure source and destination addresses
    sys_write32(src, &dma_cfg->regs->channels[channel].src);
    sys_write32(dst, &dma_cfg->regs->channels[channel].dst);
    sz = sys_read32(&dma_cfg->regs->channels[channel].size);

    // Configure the transfer size in the channel-specific registers
    sz &= ~(0xFFFF);
    sz |= (size & 0xFFFF);
    sys_write32(sz, &dma_cfg->regs->channels[channel].size);

    return 0;
}
```

Figure 8 - DMA 'reload' function implementation

Connecting PSS to the Device Driver

The Zephyr driver and software-module framework manages interoperability between embedded software modules. The Portable Test and Stimulus Standard supports interoperability between the various IP-specific pieces of test intent. A useful total solution must also address interoperability between the RTOS device driver framework and reusable PSS test intent.

The key requirement for connecting PSS to embedded software is that the PSS description be properly-configured to use the appropriate instance of the device driver. An observation to make from the public driver API (Figure 7) and the implementation functions (Figure 8) is that both receive their context data via a *struct device* pointer. We will need a way to ensure that PSS-created test code passes the correct Zephyr-created device handle to the driver functions.

There are two nice-to-have aspects of connecting PSS to the device driver: collecting the set of PSS files from the IP-specific drivers, and creating the PSS *component tree* that corresponds to the available devices in the system. These aspects can be handled manually, but some of the same features of the Zephyr device driver framework that make it easy to package and assemble device drivers for a system can help us to automate these nice-to-have aspects of PSS test content assembly.

PSS Test Content Structure

Figure 9 shows the PSS description that describes how to interact with the DMA engine. The key element is the action used to carry out a memory-to-memory DMA transfer on a channel within the DMA engine. The PSS component also contains PSS *resources, resource pools,* and *resource claims* that ensure that only a single action can use one of the DMA engine’s channels at any point in the test.

```
component fwperiph_dma_c {
    // Bring in memory-claim types
    import addr_reg_pkg::*;

    resource channel_r { }

    pool[4] channel_r  channels;
    bind channels *;

    // id points to the appropriate driver instance
    int id;

    action mem2mem_a {
        lock channel_r          channel;
        input data_b            dat_i;
        output data_b           dat_o;
        rand addr_claim_s<>     dst;


        constraint dat_i.size > 0;
        constraint dat_o.size > 0 && dat_o.size <= dat_i.size;
        constraint dst.size == dat_o.size;

        exec post_solve {
            dat_o.data = make_handle_from_claim(dst);
        }

        exec body {
            fwperiph_dma_mem2mem(
                comp.id,
                channel.instance_id,
                addr_value(dat_i.data), // src
                addr_value(dat_o.data), // dst
                dat_o.size);
        }
    }
}

import function void fwperiph_dma_mem2mem(
    int             dev_id,
    int             channel,
    bit[64]         src,
    bit[64]         dst,
    int             size);
```

Figure 9 - DMA PSS Component

The component also includes a plain-data field called *id* that we will use to correlate an instance of the DMA component in the PSS model with an instance of the device driver configured within the Zephyr RTOS. The *fwperiph_dma_mem2mem* function shown at the bottom of the PSS file and called from the *mem2mem_a* action is test utility function. Test utility functions are typically written at a higher level of abstraction than driver functions, and exist to connect the PSS test intent to the lower-level software.

```
void fwperiph_dma_mem2mem(
                int32_t             dev_id,
                int32_t             channel,
                intptr_t            src,
                intptr_t            dst,
                int32_t             size) {
  // Obtain the device handle corresponding to the device id
  const struct device *dev = id2dev(dev_id);

  // Call the driver function to configure the channel for operation
  dma_reload(dev, channel, src, dst, size);

  // Start the channel performing the transfer
  dma_start(dev, channel);

  // Wait for the transfer-complete interrupt
  dma_wait(dev, channel);
}
```

Figure 10 - Test Utility Function for mem2mem_a Action

Figure 10 shows the implementation of this test-utility function. The first thing this function does is to map the *device id* passed from the PSS model to a device-struct handle within the Zephyr RTOS. We will leverage some of the meta-data required by the Zephyr RTOS to automate creation of this mapping.

Zephyr Board Definition

The Zephyr RTOS requires a *board* definition to be provided that specifies what devices exist in the system and the configuration for those devices – for example, base addresses, interrupt mappings, etc. Zephyr accepts this information in the form of a Devicetree Specification (.dts) file [11]. In order to incorporate our DMA device driver into the Zephyr image, we need two things: a Devicetree Specification file, and a schema file that specifies the configurable attributes of our IP.

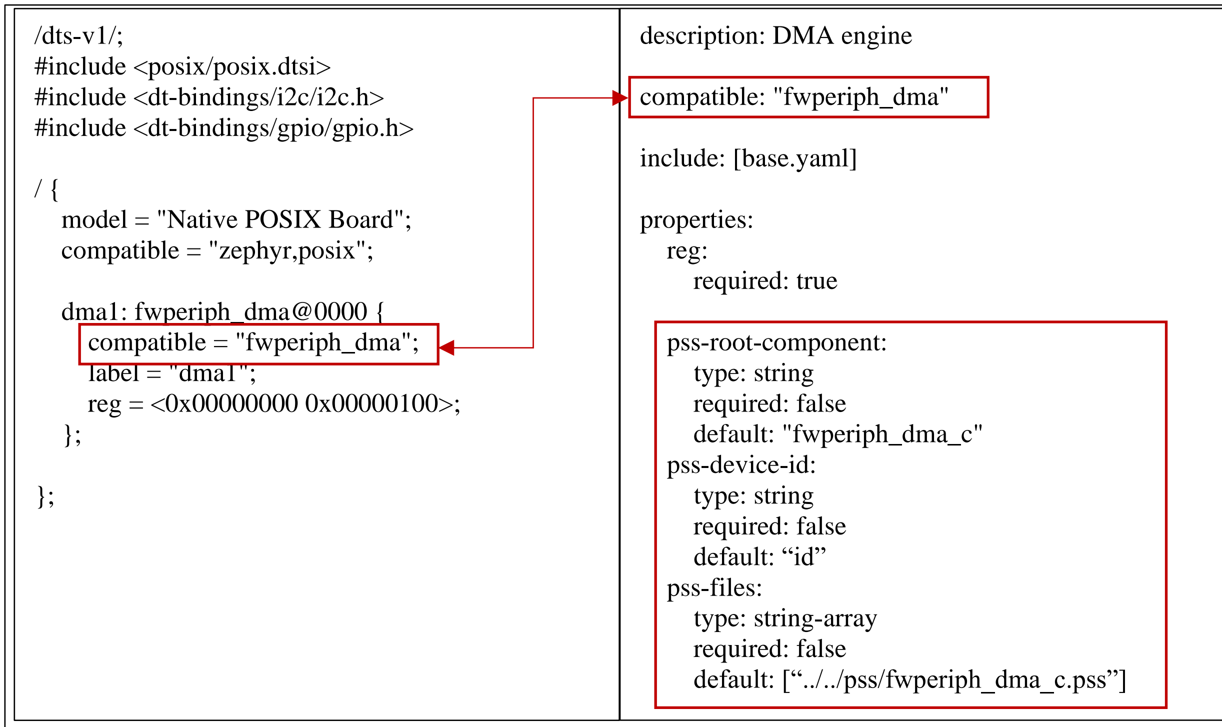

Figure 11 - Devicetree Specification and Device Schema

Figure 11 shows the Devicetree Specification (left) and Device Schema for our IP. The *compatible* entries in the two files enable the Zephyr RTOS build process to build and link the correct device driver software, and to create a device driver instance to support the *dma1* device that we have specified exists at base-address 0x00000000.

Fortunately, the Device Schema specification is extensible. We have added several special *pss* entries where meta-data about the PSS test content can be stored and later retried by an automation script. We have captured three important pieces of information:

- The name of the PSS top component for this device. An instance of this component must exist in the full PSS component tree used at the IP and SoC level.
- The name of the field in the component that holds the device ID (*id* in this case). The *id* field in each instance of this component must be properly-set to allow the test-utility code to find the proper device handle.
- The path to the PSS source files that implement the PSS model for this device.

Configuration entries in the Device Schema file are typically used to expose user-configurable settings. In this case, we are using the configuration entries to embed meta-data about the PSS model in a way that can be easily used by downstream automation that will also need to extract data from the Devicetree Specification.

PSS-Specific Generated Files

Our IP-level board definition for Zephyr is quite simple, because there is just a single device. Consequently, the generated PSS-integration files will also be quite simple. In our IP-level environment, our Zephyr *board* is named *fwperiph_dma_brd*.

```
component fwperiph_dma_brd_c {

    fwperiph_dma_c              dma1;

    exec init_down {
        dma1.id = 0;
    }

}
```

Figure 12 - Generated Board-Specific Component Tree

Figure 12 shows the PSS component generated from the Devicetree Specification file and the meta-data in the DMA Device Schema file. This PSS component will be used in the PSS model for our IP-level test scenarios.

```
#include <zephyr.h>
#include <device.h>

static struct device *fwperiph_dma_brd_devices[] = {
    DEVICE_DT_GET(DT_NODELABEL(dma1))
};

struct device *id2dev(int32_t id) {
    return fwperiph_dma_brd_devices[id];
}
```

Figure 13 - Generated Board-Specific Device-Accessor Function

Figure 13 shows the generated implementation of the previously-mentioned *id2dev* function. This function accepts an integer ID from the PSS code and returns a handle to the Zephyr device driver. Zephyr provides compile-time mechanisms for referencing devices based on their identifier in the Devicetree Specification (dma1 in this example). These mechanisms are very appropriate when writing application software, but don't lend themselves to automatically assembling test content. The generated device-accessor function bridges between a variable device identifier that the PSS code understands and the compile-time reference to a device driver instances.

At the IP level automation is nice to have, but it is feasible to manually create all the required PSS content. At the SoC level, automation is much more important given the number of IPs and the overhead of debugging mistakes made when changing the devices and device configurations within the SoC.

V. Developing IP-Level Scenarios

We will, of course, need to create tests at the IP level for the low-level driver code and PSS test content that the SoC team will use. These tests exist to verify the hardware-software interface implemented by the driver, but are not a replacement for the detailed hardware verification implemented by the UVM tests and testbench.

It is also worth noting that the IP-level scenarios will not be used by the SoC team, except as examples and starting points for SoC-level scenarios. The low-level driver firmware and PSS test content, such as what is shown in Figure 9, are the primary reusable assets that the SoC team will leverage.

So, what should we test at the IP level? We need to ensure that all the PSS reusable test-content actions work properly. Concurrent scenarios, where multiple behaviors happen simultaneously, are a key focus of SoC-level testing, so we should ensure that our PSS actions and driver code work properly under these circumstances.

```
component fwperiph_dma_uvm_c {
    fwperiph_dma_brd_c              board;

    activity entry_a {

        repeat (20) {
            parallel {
                do fwperiph_dma_c::mem2mem;
                do fwperiph_dma_c::mem2mem;
                do fwperiph_dma_c::mem2mem;
                do fwperiph_dma_c::mem2mem;
            }
        }
    }
}
```

Figure 14 - IP-Level Test Scenario

Figure 14 shows a typical IP-level scenario where four parallel transfers are carried out in a loop. This scenario will randomly run overlapping differently-sized transfers on different DMA channels. In addition to testing the basics of whether the driver code properly programs the DMA registers, a test like this confirms that the interrupt routine properly deals with multiple pending transfers. While our PSS development at the IP level is primarily for the benefit of the SoC team at this point, it can also be used as a laboratory in which to begin identifying the types of IP-level verification problems where PSS can help.

VI. Moving to SoC Level

Including PSS test content and low-level firmware drivers targeted to a common RTOS as part of the IP deliverable, along with design RTL, allows us to achieve reuse with firmware and test content in addition to RTL reuse. Assembly automation enabled by the Zephyr RTOS driver framework and our PSS-based additions to the driver modules help to simplify the assembly process for the test-software image.

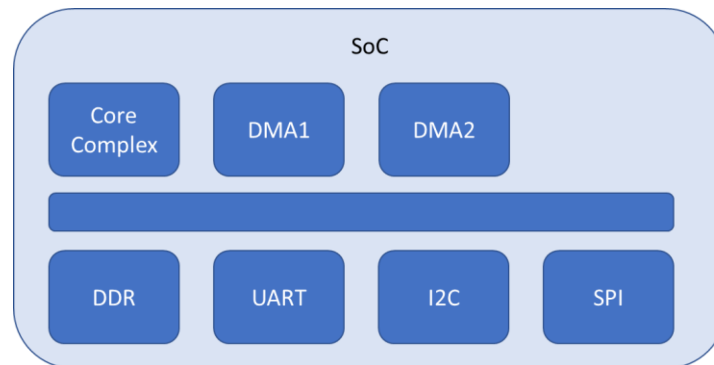

Figure 15 - SoC Example Design

At the IP level, we focused on a developing driver firmware for a single DMA engine. At the SoC level (Figure 15), we have more IPs – two instances of our DMA IP, along with some communication controllers.

```
/dts-v1/;
#include <posix/posix.dtsi>
#include <dt-bindings/i2c/i2c.h>
#include <dt-bindings/gpio/gpio.h>

/ {
    model = "Tiny SoC";
    compatible = "zephyr,riscv";

    dma1: fwperiph_dma@80000000 {
        compatible = "fwperiph_dma";
        label = "dma1";
        reg = <0x80000000 0x00000100>;
    };
    dma2: fwperiph_dma@80000100 {
        compatible = "fwperiph_dma";
        label = "dma2";
        reg = <0x80000100 0x00000100>;
    };
    uart : fwuart@80000200 {
        compatible = "fwuart";
        label = "uart";
        reg = <0x80000200 0x00000100>;
    };
    i2c : fwi2c@80000300 {
        compatible = "fwi2c";
        label = "i2c";
        reg = <0x80000300 0x00000100>;
    };
    spi : fwspi@80000400 {
        compatible = "fwspi";
        label = "spi";
        reg = <0x80000400 0x00000100>;
    };
};
```

Figure 16 - Soc-Level Devicetree Specification

The corresponding SoC-level Devicetree Specification is shown in Figure 16, and drives creation of the SoC-level firmware image (via Zephyr) and the SoC-level PSS infrastructure. Having a single location from which to specify required drivers and their configurations (ie base address) simplifies the process of creating the base firmware image.

```
// Note: generated from DTS file tiny_soc_brd.dts

component tiny_soc_brd_c {

    fwperiph_dma_c      dma1;
    fwperiph_dma_c      dma2;
    fwuart_c            uart;
    fwi2c_c             i2c;
    fwspi_c             spi;

    exec init_down {
        dma1.id = 0;
        dma2.id = 1;
        uart.id = 2;
        i2c.id  = 3;
        spi.id  = 4;
    }
}
```

Figure 17 - Generated PSS Component Tree

This same description can be leveraged to create the PSS component tree and to connect the PSS components to the corresponding device driver instances. There is value in automating this process even when it doesn't involve much code, as is the case with our small example SoC (Figure 17). Deriving the PSS integration from a single specification still allows us to avoid keeping the content within three different files in sync.

VII. SoC-Level Scenarios

At the SoC level, our key verification concern is ensuring that the integrated IPs work together under a variety of conditions. Consequently, we won't seek to reuse the specific test scenarios we developed at the IP level. Rather, we will combine the IP-level PSS test content, delivered along with the IP RTL, to compose scenarios that exercise the IPs in combination.

```
component tiny_soc_c {

    // IP-specific components
    tiny_soc_brd_c              board;

    // Send data to SPI and UART simultaneously
    action mem2spi_uart_c {
        fwuart_c::xmit_a uart_xmit;
        fwspi_c::xmit_a   spi_xmit;

        parallel {
            uart_xmit;
            spi_xmit;
        }
    }
}
```

Figure 18 - SoC-level scenario to exercise SPI and UART

Figure 18 shows a simple SoC-level scenario that makes use of actions delivered along with the UART and SPI IPs to exercise transmitting traffic via both simultaneously. Note the instance of the PSS component containing PSS component instances (tiny_soc_brd_c) corresponding to each IP, which generated from the DTS file.

VIII. CONCLUSION

This paper has shown how the Zephyr RTOS can be used as a framework for developing modular, interoperable, low-level driver firmware at the IP level, which enables a complete firmware image to be assembled from IP-delivered drivers at the SoC level. Developing firmware along with IP design and verification captures the IP team's knowledge in an executable form that is useful to the SoC team. Developing the driver software on a host workstation rather than on a simulated embedded core enables the team to gain development efficiencies using native software-development tools.

As more techniques (eg portable stimulus and embedded software) are included in the verification process, automation and methodology are increasingly necessary to deal with the additional complexity. This paper has shown how PSS can be grafted onto an existing RTOS-driver framework at the IP level to automate construction of key PSS structures at the SoC level. While this paper has chosen to use the Zephyr RTOS, it is believed that the concepts should translate to other OS and RTOS frameworks. Having driver firmware available for key IP at the beginning of the SoC verification process is a key enabler for achieving optimal value from applying PSS for SoC-level test creation.

REFERENCES

[1] Accellera, "Portable Test and Stimulus Standard Version 2.0", https://www.accellera.org/downloads/standards/portable-stimulus

[2] D. Rich "Easy Steps Towards Virtual Prototyping using the SystemVerilog DPI", DVCon 2013

[3] A. Freitas, "Hardware/Software Co-Verification Using the SystemVerilog DPI", https://picture.iczhiku.com/resource/paper/WhIRKfhLkpfqOXVV.pdf

[4] Wikipedia, "Zephyr (operating system)", https://en.wikipedia.org/wiki/Zephyr_(operating_system)

[5] Zephyr Project, "Minimal footprint", https://docs.zephyrproject.org/latest/samples/basic/minimal/README.html

[6] Zephyr Project, "Rationale for this port" (POSIX architecture), https://docs.zephyrproject.org/latest/boards/posix/native_posix/doc/index.html#rationale-for-this-port

[7] M. Ballance, "Zephyr Cosim", https://github.com/zephyr-dv/zephyr-cosim

[8] M. Ballance, "TbLink Project", https://github.com/tblink-rpc

[9] Verification Academy, "Classes for Adapting Between Register and Bus Operations", https://verificationacademy.com/verification-methodology-reference/uvm/docs_1.2/html/files/reg/uvm_reg_adapter-svh.html

[10] Valgrind Project, "Memcheck: a memory error detector", https://valgrind.org/docs/manual/mc-manual.html

[11] Devicetree Project, "The Devicetree Specification", https://www.devicetree.org/specifications/