2022 DESIGN AND VERIFICATION™ DVCON CONFERENCE AND EXHIBITION UNITED STATES

SAN JOSE, CA, USA
FEBRUARY 28 - MARCH 3, 2022

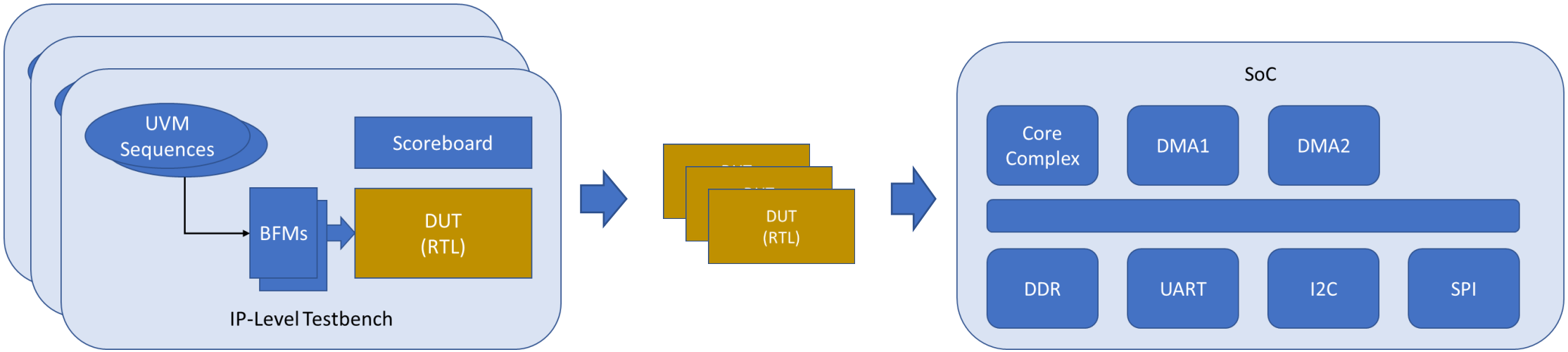
Co-Developing IP and SoC Bring-up Firmware with PSS

Matthew Ballance, Siemens EDA

accellera SYSTEMS INITIATIVE

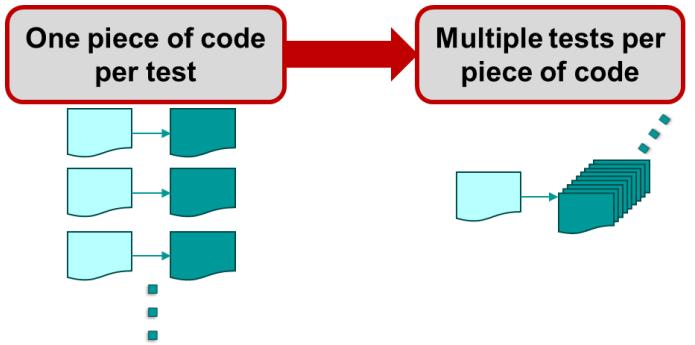
Verification at IP and SoC

- Common to use UVM for IP-level verification
  - Sequences interact with device registers
  - Use directed and constrained-random tests to exercise key cases
- SoC assembled from RTL delivered by IP teams
- Use processor cores to verify SoC integration
  - Need to write low-level firmware (driver) to interact with device registers
  - Need to write embedded-software tests

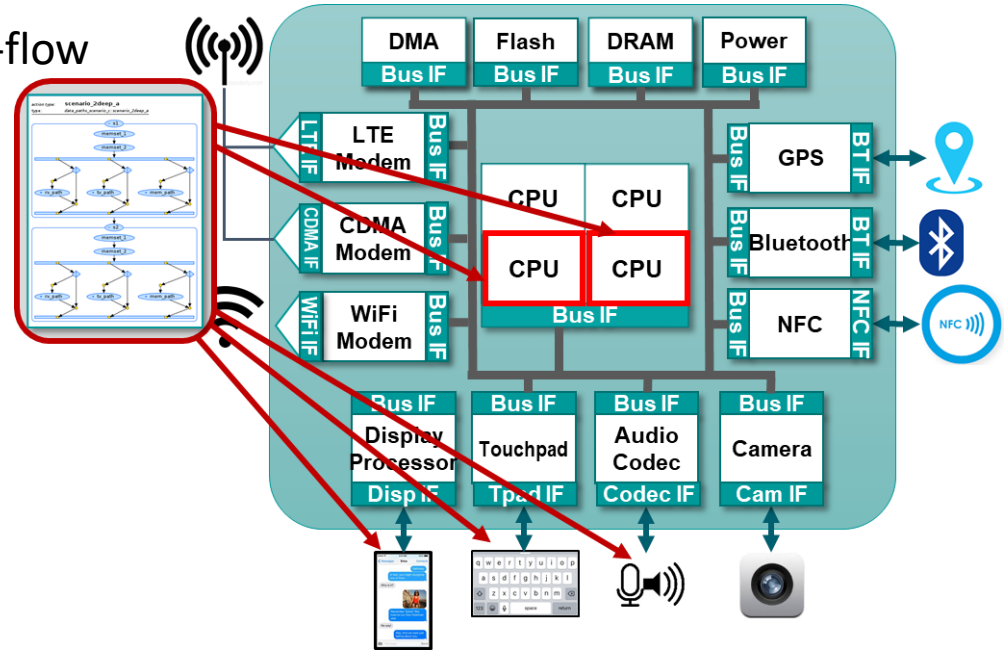


PSS: Constrained-Random Testing for SoC

- Writing SoC-level tests is challenging
  - Coordinating multi-core tests and managing concurrency
  - Large number of cases to cover
  - Directed-test techniques are labor-intensive

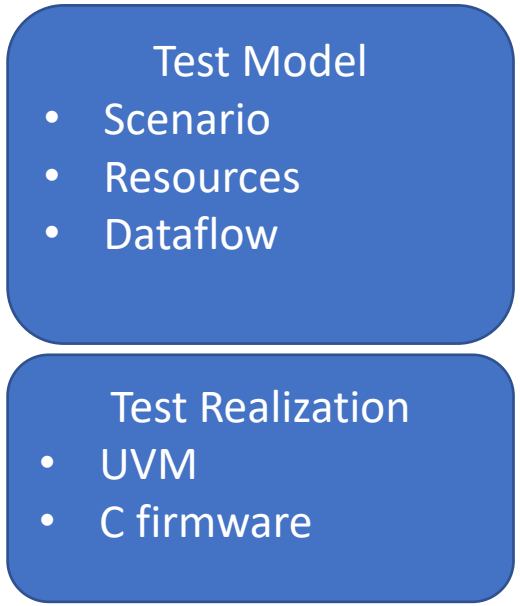


- PSS offers a compelling alternative
  - Model test scenarios in terms of actions, resources, and data-flow
  - Generate multiple randomized test scenarios
  - Constrain model to focus scenarios



Two Pieces of a PSS Model

- Test Intent – *What* is to be tested
  - Models scenarios to be tested
  - Captures rules around resource utilization and dataflow
- Test Realization – *How* that is carried out
  - Interacts with IP
  - Reads/writes registers
  - Waits for interrupts
- PSS helps create more tests more quickly
- But, low-level driver code is still required to implement test realization

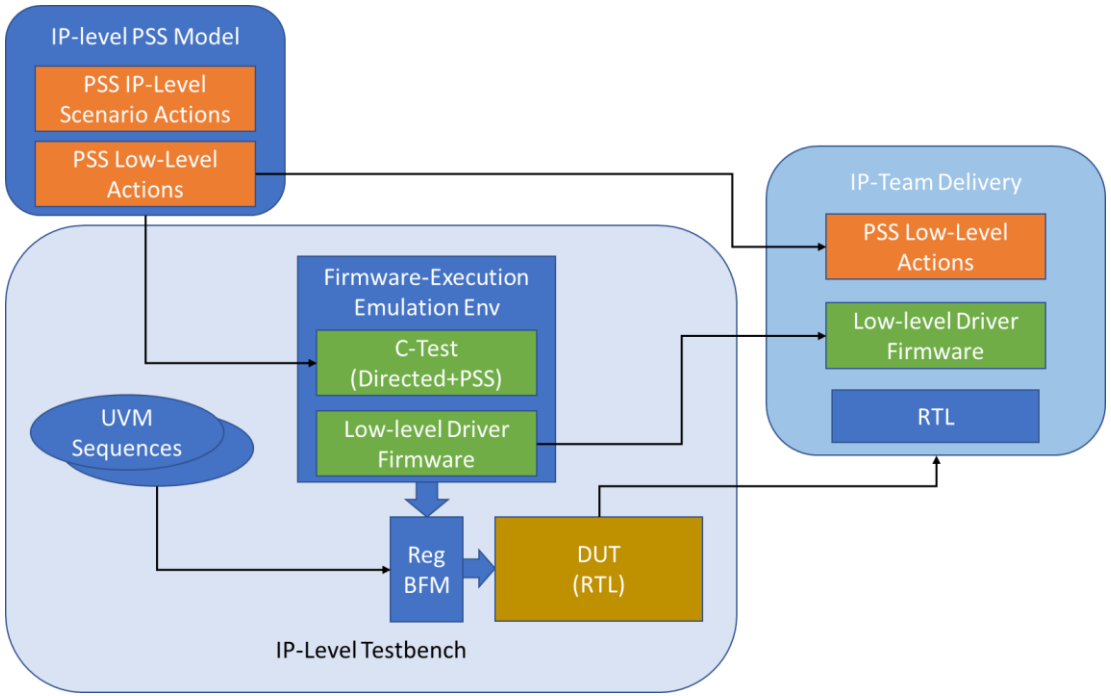


Objective: Expand IP-level Deliverables

- Much of the IP-level verification process remains the same
  - Still use UVM for functional correctness

- Add creation of low-level driver code
  - Leverage design/verification team's knowledge of device programming
- Add creation of leaf-level PSS actions
  - Provide test 'user interface' to low-level driver code

- IP deliverable now includes
  - RTL
  - Low-level driver code
  - PSS reusable test fragments

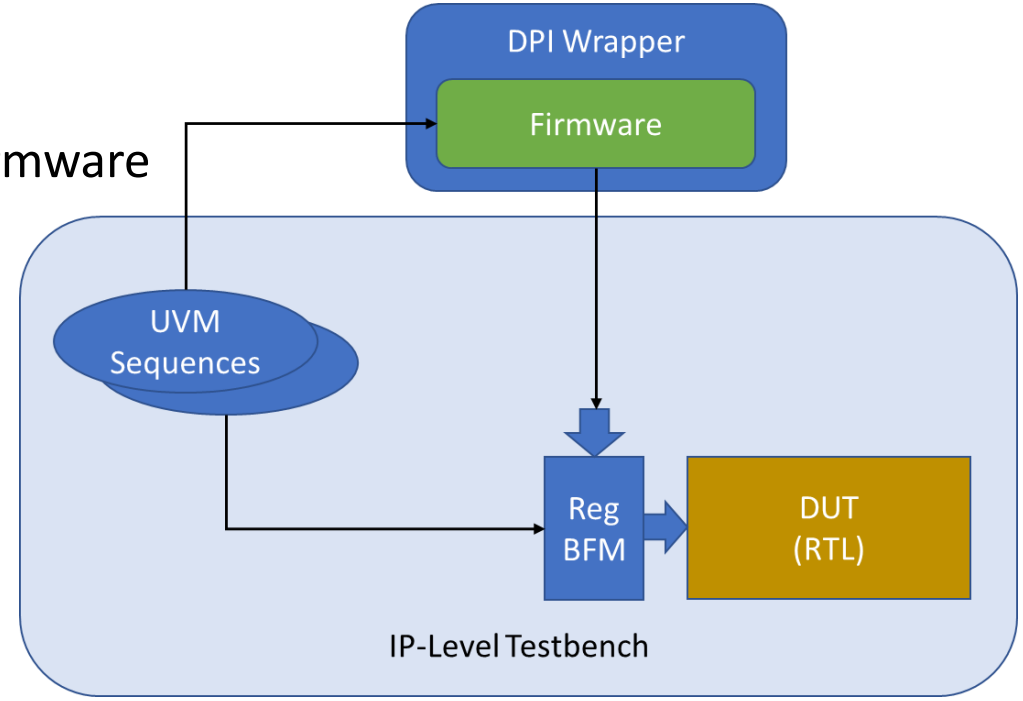


Adding firmware to simulation is simple

- At least relatively so
  - And, there's a fair amount of prior art here

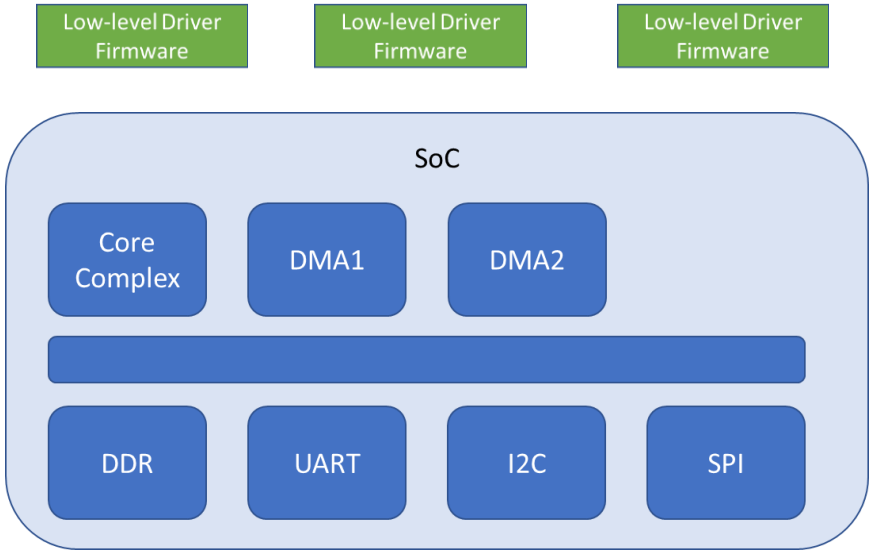
- Define a DPI interface between testbench and firmware
- Wrap firmware as a DPI library
- Call firmware from SystemVerilog, and testbench from firmware

- So, we're done right? Ship it!
- Well, not quite…
- Firmware for our IP must integrate with other firmware



Problem: combining firmware isn't simple

- At SoC level, our low-level driver is just one of many
- It will need to be configured along with other drivers
  - What is the register base address?
  - Which interrupt is it sensitive to?
- It must co-exist with other drivers
- We need an interoperability framework
- How can we avoid rolling our own?



Where have we seen this before?

- An OS has the same driver-integration requirements
  - Support integration of multiple drivers
  - Support multiple device instances
  - Configure key aspects of drivers, such as registers
  - Specify connections, such as the interrupt to be used by a driver

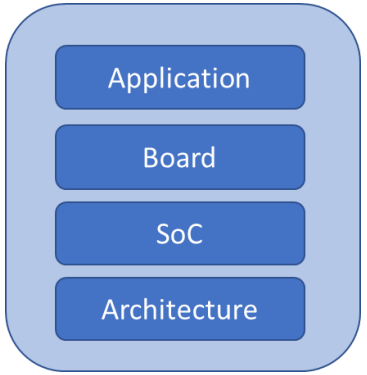
- An OS is a heavy-weight thing – far too much overhead for bare-metal testing
  - Too much memory
  - Too long to start-up
  - Too much complexity

An RTOS, on the other hand…

- An RTOS has many of the same requirements as an OS around driver management
  - But, is designed for low-power, low-resource systems

- In this paper, we look at the Zephyr RTOS, but many others exist

- Characteristics of the Zephyr RTOS
  - Modular and highly configurable
  - More like a library than a traditional OS.
    - OS and application create single exe
  - Can be configured to be very small (~8k)
  - Extremely minimal startup behavior.
  - Very close to bare-metal

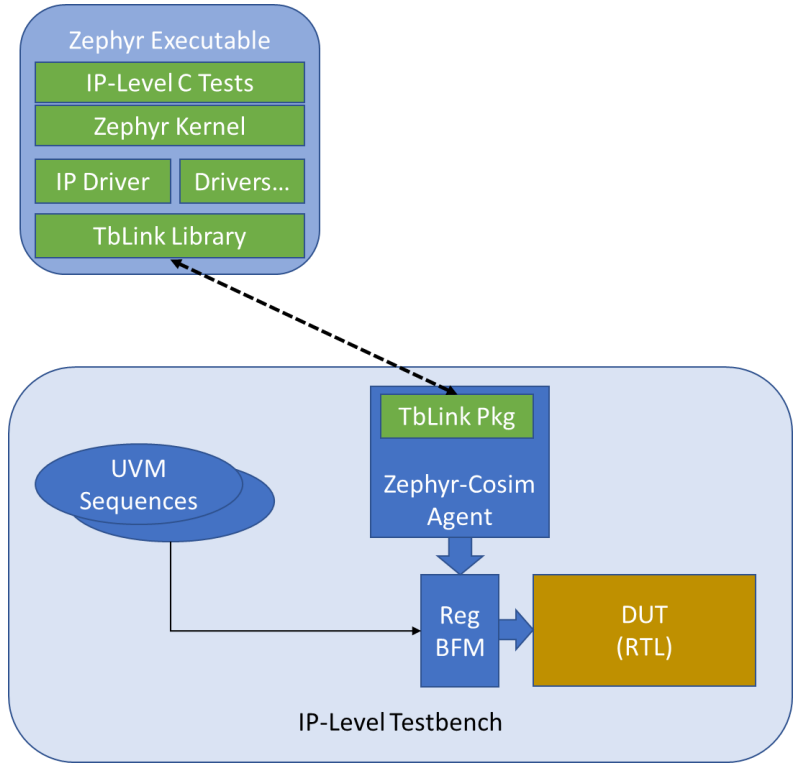


How does an RTOS fit at IP Level?

- Could run the RTOS on a full instruction-set simulator/virtual platform
  - This adds complexity, but doesn't necessarily help us create firmware

- Zephyr RTOS has a mode where it compiles to a host application
  - Primarily intended for application developers
  - But, can be re-purposed to create a co-simulation

- Zephyr integrates with the testbench as a UVM agent
  - Uses the UVM register model to access DUT registers and memory
  - Accepts interrupt-request events from the testbench



Creating a Driver

- Zephyr specifies a format for driver modules
  - Files to specify configurable attributes and requirements
  - Driver source structure
  - Mechanisms for accessing configurable attributes
  - APIs for standard devices (DMA, serial, timer, etc)
- Most effort can focus on implementing behavior
- Drivers are very similar to UVM utility sequences
  - Program device registers
  - React to interrupt events
  - …

```
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;
}
```

Back to PSS – Creating Test Content

- Writing bare-metal C tests – even at IP – isn't easy

- PSS can help us in creating (at least) two types of content
  - 'building blocks' that can be assembled into SoC-level tests
  - IP scenario-level tests using firmware

- Note: We're augmenting, not replacing, standard UVM verification
  - Use coverage-driven constrained-random UVM flow to verify functional correctness
  - Create simple PSS tests focus on verifying firmware and Hw/Sw interation
  - Create more-complex PSS tests exercise scenarios

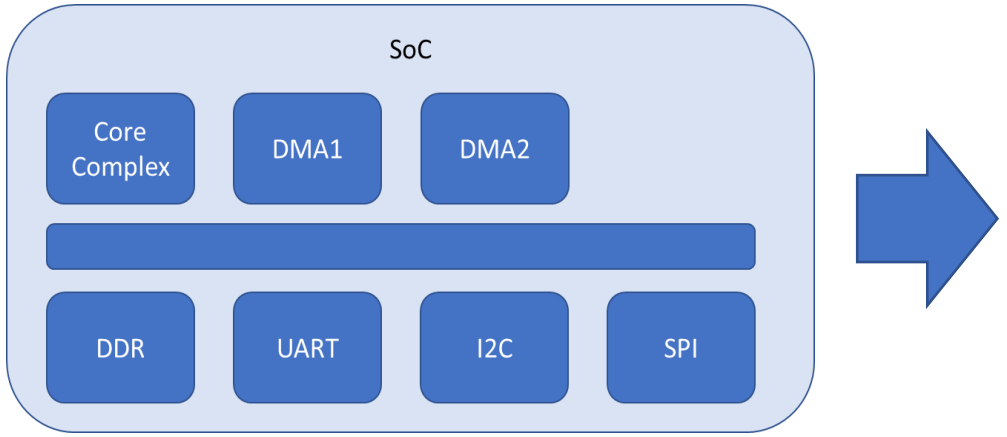
PSS Building Blocks

- Provide a PSS ‘interface’ to IP
  - Capture key behaviors as actions
  - Capture key parameters that can be constrained
  - Capture core validity rules
  - Connect PSS to driver firmware
- Useful at IP level for simple and complex scenarios
- Critical productivity-enabler at SoC level
  - Allows verification engineer to easily compose multi-IP scenarios

```
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 body {
         fwperiph_dma_mem2mem(comp.id, channel.instance_id,
            addr_value(dat_i.data), addr_value(dat_o.data), dat_o.size);
      }
}
```

Moving to SoC

- IP package supports PSS-based test creation
  - Firmware (C source)
  - Device Schema
  - IP-specific actions
- SoC-level RTOS view described using a DeviceTree specification
  - Specifies which IPs are present, and how they're configured
  - Used by Linux, Zephyr RTOS, and others
- Leverage DeviceTree information to support PSS




```
/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>;
    };

    // …
};
```

SoC-Level Scenarios

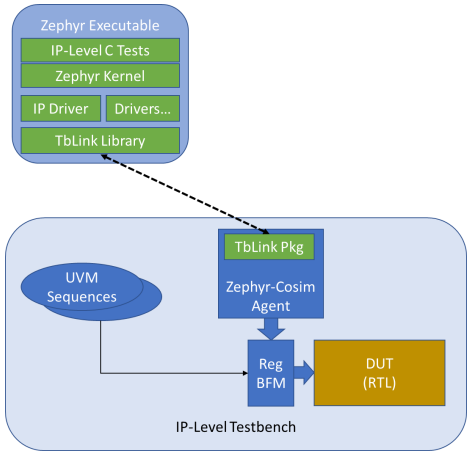
- Data from DeviceTree automates creation of PSS component tree
  - Captures PSS representation of available IPs
- PSS building blocks simplify scenario assembly
- Driver firmware automatically connected to PSS actions
- Verifying firmware at IP level reduces bug sources
- PSS language features simplify creating rich SoC-level scenarios

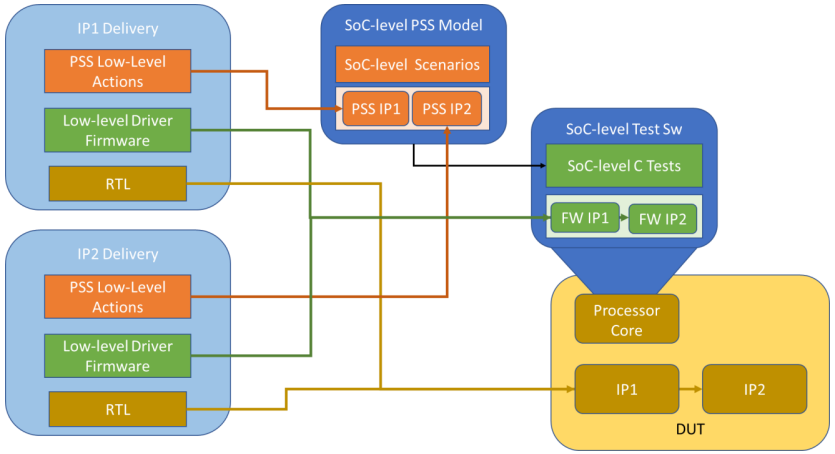
```
component tiny_soc_c {

  // 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;
    }
  }
}
```

Conclusion

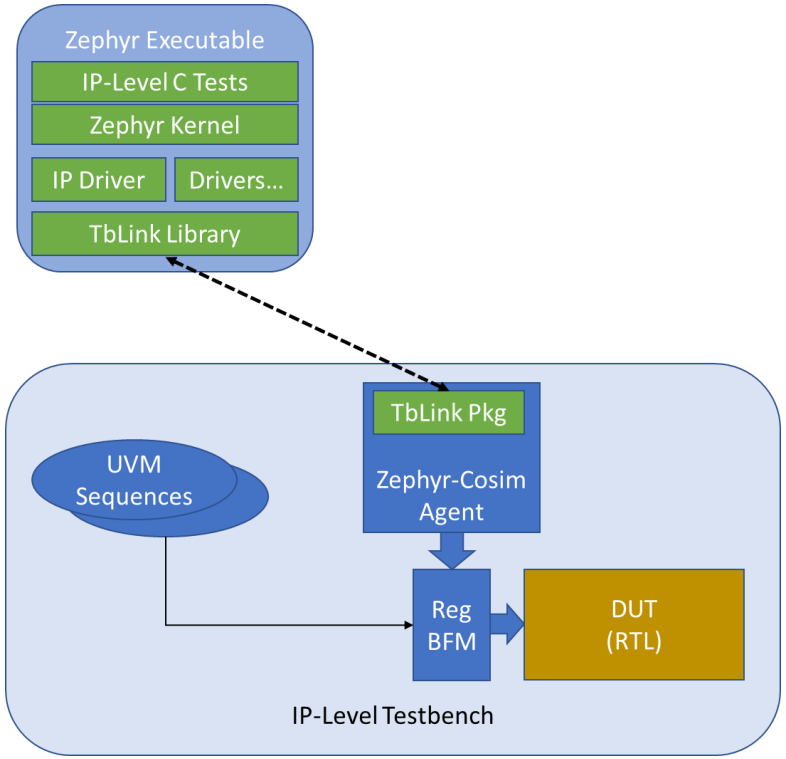
- SoC integration testing is enabled by having test content delivered along with IP

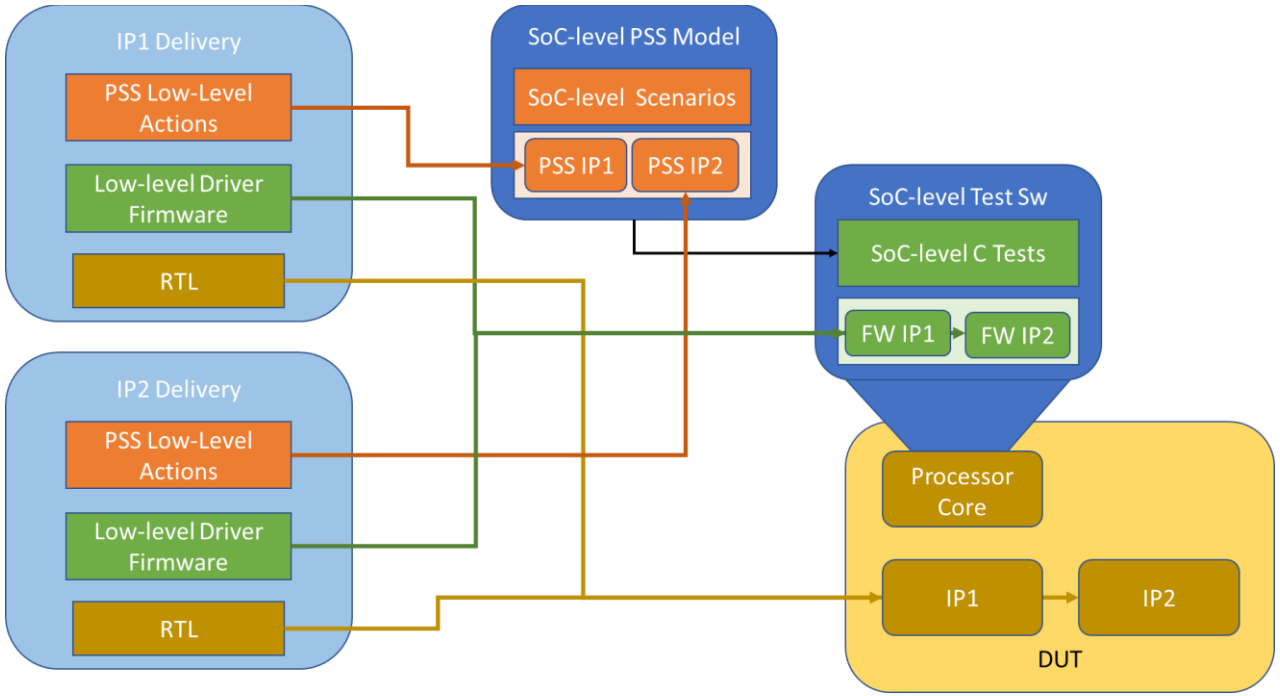
- PSS can provide reusable test content
  - But, PSS test content depends on having firmware as well
- Show a flow in which PSS+firmware developed alongside IP
  - IP can now deliver RTL, firmware, and test content building blocks to SoC
- Show using Zephyr RTOS as the common software env between IP and SoC





Q&A


Zephyr Executable
IP-Level C Tests
Zephyr Kernel
IP Driver
Drivers...
TbLink Library
TbLink Pkg
Zephyr-Cosim Agent
UVM Sequences
Reg BFM
DUT (RTL)
IP-Level Testbench


IP1 Delivery
PSS Low-Level Actions
Low-level Driver Firmware
RTL
IP2 Delivery
PSS Low-Level Actions
Low-level Driver Firmware
RTL
SoC-level PSS Model
SoC-level Scenarios
PSS IP1
PSS IP2
SoC-level Test Sw
SoC-level C Tests
FW IP1
FW IP2
Processor Core
IP1
IP2
DUT