Assertions for Formal Tools

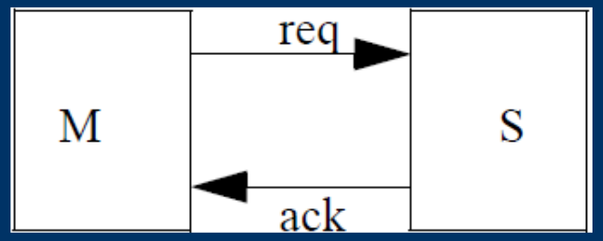
- Model Checking and Assertions
- Assertions on Data

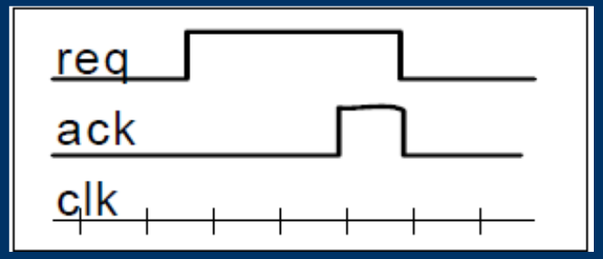
Reference:
VMM by Janick Bergen

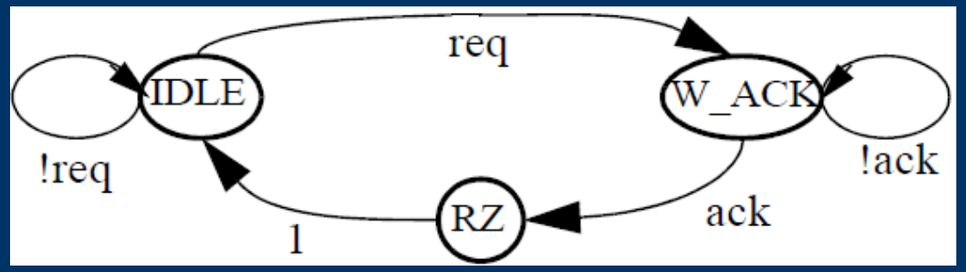
Model Checking and Assertions

- The formal tools used to verify properties specified using a temporal language have been usually called model checkers or property checkers
- The tools may then carry out the following tasks:
  1. Given a set of properties that represent assumptions on the behavior of the environment, prove that the DUT satisfies the assertions on its behavior.
  2. Given a coverage goal specified as a *cover property* statement or a particular signal state of the design, the tool generates a stimulus sequence that drives the design to reach that goal.
  3. Given a set of state variables in the DUT, the tool determines a lower bound on the set of states that cannot ever be reached

Model Checking and Assertions







Master-Slave Handshake Protocol

```
// MasterCheck
  M1: assert property (@(clk)
        state == W_ACK |-> req );
  M2: assert property (@(clk)
        state == RZ |-> !req );
//SlaveCheck
  S1: assert property (@(clk) // bounded liveness
        (state == IDLE) && req
          |->
      ##[min_ack_latency:max_ack_latency] ack );
  S2: assert property (@(clk)
        (state == I) || (state == RZ) |-> !ack );
```

Assertions on Data

- Guidelines to write assertions on data that can be formally verified
- Without Local Variables
  - Without the use of local variables, task calls and non-synthesizable Boolean expressions and functions, all SystemVerilog properties are synthesizable into RTL code.
    - Enumeration should be used for small data value ranges
- With Local Variables
  - Predict the storage requirements for the assertion during the lifetime of evaluation threads and attempts

System-Level Verification

- Extensible Verification Components
- XVC Architecture
- System-Level Verification Environments
- Verifying Transaction-Level Models
- Hardware-Assisted Verification

Reference:

VMM by Janick Bergen

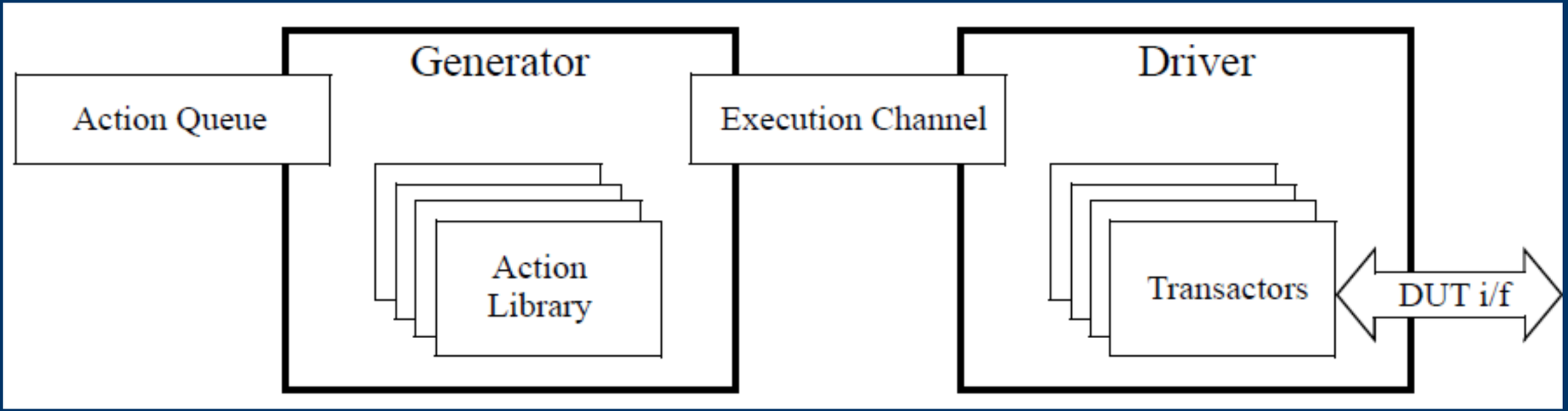
System-Level Verification

- A system is a design that is composed of independently designed and independently verified blocks as well as a block interconnect infrastructure.
- System-level verification is the verification of the correct interaction among these individual blocks.
- Each block, including the block interconnect infrastructure—a bus and associated bridges and other supporting elements—has been verified individually.
- Therefore, system-level verification should focus on the functionality embodied by the combination of the blocks.

Extensible Verification Components (XVC)

- Extend or combine their block-level functionality into system-level functionality
- XVCs provide a foundation for modular, scalable and reusable system-level verification environments, with the aim of minimizing test set-up overhead.
- The purpose of an XVC is to support system-level integration and functional verification using both directed and random testing using a unified methodology approach.
- The structure of an XVC is such that it is highly portable to different system-level designs and across design abstraction levels

XVC Architecture



- The generator layer executes user-defined actions.
- The driver layer interacts with the DUT to execute or monitor the transactions as required by the action.
- The generation layer controls the transactors in the driver layer.
- The action library contains a selection of known test actions that can be executed by the XVC.

XVC Architecture

- XVCs shall be configurable to match the design they are testing
  - Parameters such as bus width, FIFO depth, clock frequency shall be configurable in the XVC
- XVCs shall be configurable to constrain functionality as required
  - When a DUT implements only a subset of a particular interface protocol, a corresponding driver XVC shall be configurable to prevent certain stimulus from being generated
- XVCs should be configurable to allow for error injection
  - Verification requirements may require that invalid protocol or data be applied to test the error detection and recovery mechanisms of the DUT

System-Level Verification Environments

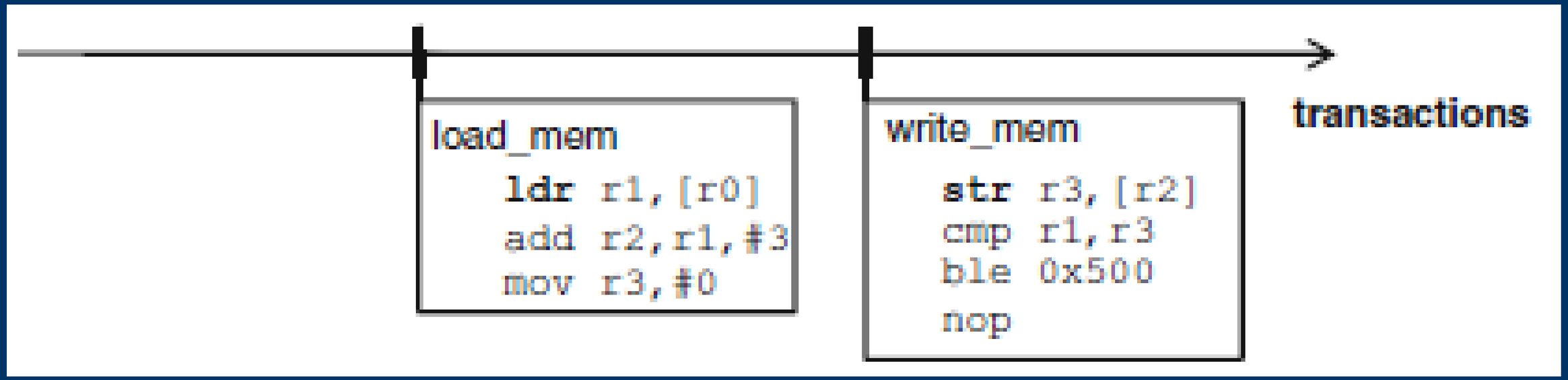
- Unless a standard approach is taken, block-level testbench components or functional coverage elements cannot easily be reused at the system level
- System-level verification should be split across multiple verification environments
- Block interconnect infrastructure environment
  - Pre-verified in its own verification environment
  - Checks the functional correctness of data transfers, protocol rules and bus performance requirements, such as latency and bandwidth
- Basic integration environment
  - Checks the correctness of the connectivity by toggling all I/O ports in a system
- Low-level system functional environment
  - This environment is to cover any functionality that cannot readily be observed
  - Monitoring of control signals, reset mode checking etc
- System validation environment
  - System validation ensures that the overall performance requirements such as latency and bandwidth of the system are met

Verifying Transaction-Level Models

- Levels of abstraction
  - Continuous Time, Discrete-event, Cycle-accurate, Instruction-accurate, Transaction-accurate
- Transaction-accurate
  - Behavior is expressed in terms of the interactions between the components of a system.
  - These interactions are called transactions.
- For example, one could model a system with a disk drive and a user application, and create a simulation that focuses on the commands exchanged between the disk drive and the user application.
- A transaction-accurate model allows considerable simplification of the disk drive and the user application.
- Indeed, in between two transactions, several instructions can be lumped together and simulated as a single, atomic function call.

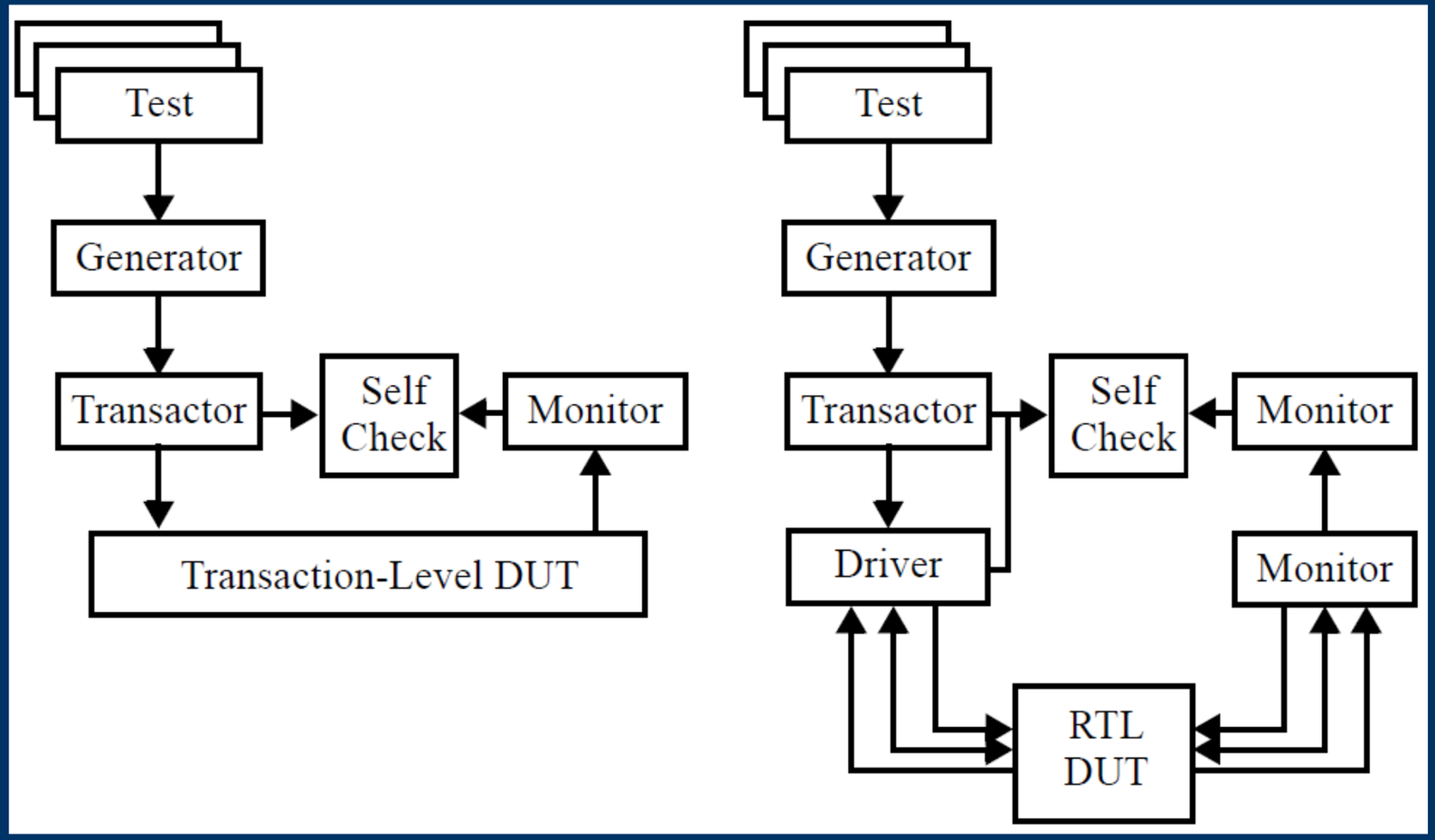
Verifying Transaction-Level Models

- Transaction-accurate



Verifying Transaction-Level Models

- Verification environments should be reusable across different abstraction views of the same DUT.



Reusing a Transaction-Level Verification Environment with an RTL Implementation

- SystemC, SystemVerilog can be used for TLM

Hardware-Assisted Verification

- Used when design must be verified on actual hardware
- Emulation of the design is usually required for runtime performance
- Peripherals integrated into a hardware-assisted verification of the design may have a number of interfaces to consider for verification.
- The bus interface is de-facto handled by the system itself.
- But any external interface needs to be driven or monitored during block- and system-level testing, and under the control of the simulation-based environment manager.
- Stand-alone, synthesizable RTL verification component for an external interface - such a verification component, called a peripheral test block (PTB), To work as
  - Standalone
  - Controlled by an external driver such as a file-reader bus-functional model
  - Controlled by an XVC.

Hardware-Assisted Verification

- Examples of peripherals that a PTB can address:
- Sequential Data Peripherals
  - These peripherals transfer data sequentially, a single word, byte or nibble etc. at a time.
  - The content or structure of the sequential data is not defined.
  - Each transfer is independent of the previous or subsequent transfers.
  - Peripherals of this type include UARTs and GPIO.
- Sequential Block Peripherals
  - These peripherals transfer data sequentially, a block of data at a time (multiple words, bytes or nibbles etc.).
  - The format of the data in the block is defined for individual transfers and blocks are transferred in a fixed order.
  - Peripherals of this type include video controllers and network interfaces (Ethernet, USB etc.).

Hardware-Assisted Verification

- Examples of peripherals that a PTB can address:
- Random Block Peripherals
  - These peripherals transfer data a block at a time.
  - The format of the data in the block is defined for individual transfers and blocks are transferred in an arbitrary order.
  - Peripherals of this type include Serial Peripheral Interface, Memory Stick, SD-Card and Multi-Media-Card reader interfaces.

Hardware-Assisted Verification

- Examples of peripherals that a PTB does not attempt to address include
- Random Data Peripherals
  - These devices transfer the data in a random way for example a single word, byte or nibble etc. at a time.
  - The format for the data is not defined for the transfer larger than the entity size and data can be read out in any order.
  - Peripherals of this type include ROMs and RAMs.

Processor Integration Verification

- Software Test Environments
- Structure of Software Tests
- Test Actions

Reference:

VMM by Janick Bergen

Processor Integration Verification

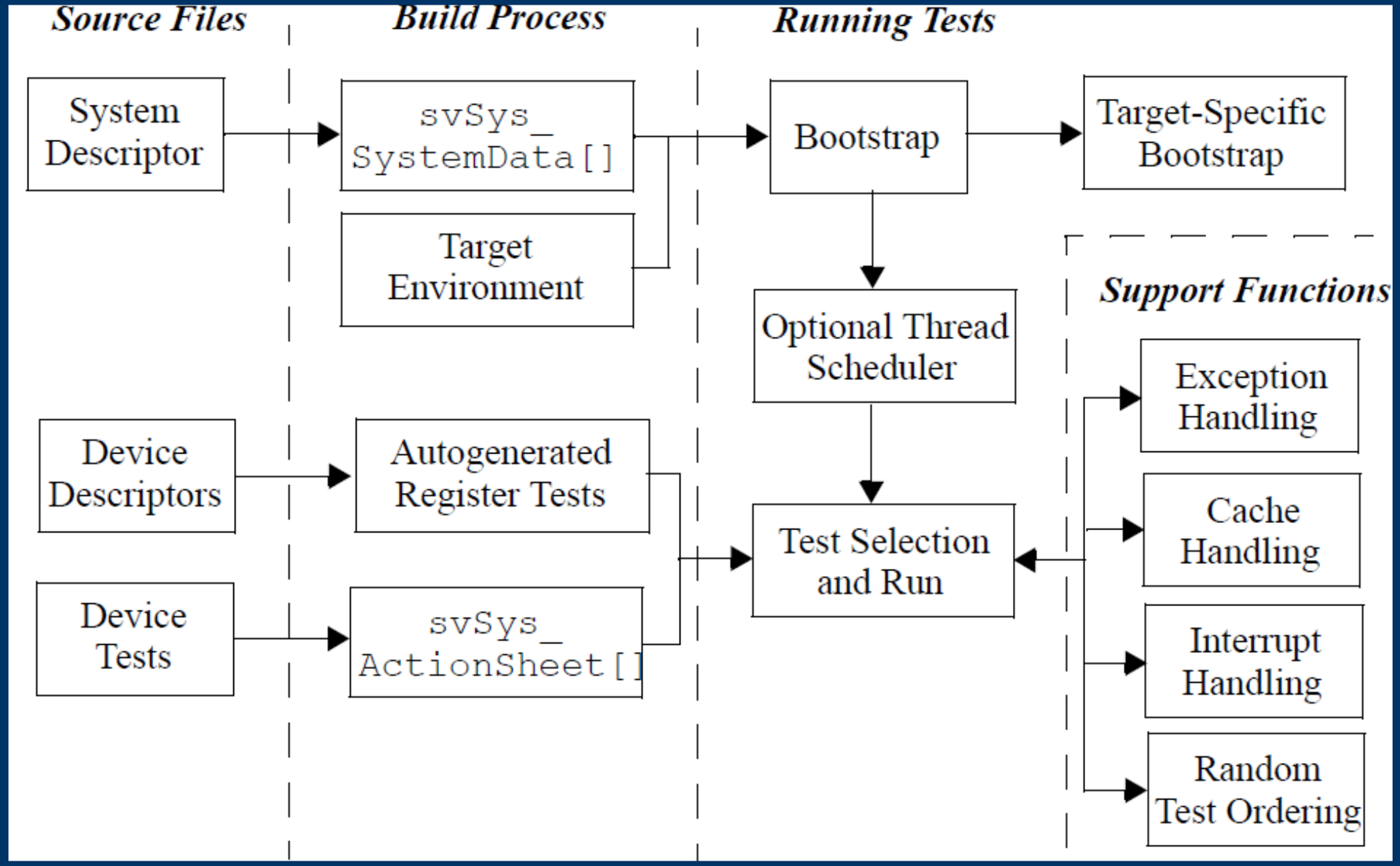
- Verification using embedded software is an important part of any system verification infrastructure
  - Host processor directs application data and controls memories and peripherals.
- Suppose, a system-level testing where a CPU or DSP is part of the system design being tested
  - It is desirable to have a verification environment that supports the execution of test software to demonstrate that the system can successfully support the execution of an operating system, application software or a DSP control algorithm.
- Software test environments to complement the hardware-centric infrastructure
- The environment is used in place of an operating system in a CPU-centric system design.
- System verification can thus be conducted prior to the operating system being available on the system itself.
- XVCs in the verification environment will work in concert with the software test framework such as to meet the hardware/software verification requirements
  - Both external and software-internal stimulus can be generated and synchronized to create interesting and relevant system conditions.

Software Test Environments

- CPU or DSP integration should be tested using software test routines
- Software test routines must be used to direct the processor(s) to address the system peripherals they are responsible for interacting with
- For example, a processor may insert an extra idle cycle due to an internal processing operation.
- The arbitration unit may decide to take away bus ownership during that extra idle cycle and hence delay a slave transaction or even defer it to another master.
- In contrast, a bus-functional model may generate slightly different behavior whilst still driving the bus with the same stimulus.
- Were the bus-functional model not to generate an additional idle cycle in this scenario, the arbitration handover would not take place, leading to a different scenario being played in the system
- In another aspect, unless the CPU/DSP transactor also emulates the instruction fetch and-execute cycle from external memory, it is also unlikely to exhibit the same behavior as the actual CPU or DSP in operation.
- It would be possible to program the transactor to exhibit this behavior, but in practice it is far simpler to program the CPU or DSP to execute instructions to perform this task.

Structure of Software Tests

- Software Test Execution Process



Test Actions

- Compilation Process
- Running Tests
- Bootstrap
- For every peripheral, there shall be one or more test actions.
- Each action is designed to test a specific part of the peripheral’s functionality.
- These actions are written predominantly in C, with assembler being used where necessary.
- Every action should be self verifying by checking the result of the action against an expected result and return a pass or a fail indication.
- The software verification framework will output a message should the action fail.
- Test actions source files should be reusable by the software build process from one system design to another

Test Actions

- Compilation Process
- A top-level script should manage the full compilation process
- The compilation process should managed by a top-level Perl script or makefile.
- A full compilation process will perform the following steps:
  - Visit the entire directory tree under the root directory looking for directories named testcode.
  - In all such directories found, locate all C and ASM files in that directory identified by the .c and .s suffixes, respectively.
  - Compile all C and ASM files to object files in the relevant object code directory if the source is newer than the existing object code file in the object code directory.
- The compilation process can support creating a configurable software test image to be loaded into the verification environment

Test Actions

- Running Tests
- After all source files have been compiled, running a test sequence requires:
  - Specifying the tests and test conditions in a top-level source file.
  - Compiling these top-level source files into object files in the relevant object code directory if the source is newer than the existing object code file in the object code directory.
  - Linking all required object files into an appropriate .elf or binary files.
  - Running the .elf or binary on the chosen target and setting up other binaries as needed.

Test Actions

- Bootstrap
- The bootstrap module performs the following tasks:
  - Do minimal setup.
  - Set up the memory controllers.
  - Do further setup including stacks, heap, caches, MMU
  - Optionally perform the board configuration.
  - Execute the test(s).
  - Perform end-of-test operations.
- After completing the end-of-test operations, the bootstrap module should report a summary of tests.
- The number of tests passed, skipped and failed should be reported.

Post-Silicon SoC Validation

- Introduction
- Validation Activities
- Planning for Post-Silicon Readiness
- Post-Silicon Debug Infrastructure
- Generation of Tests
- Post-Silicon Debug

Reference: Post-Silicon Validation and Debug by Prabhat Mishra & Farimah Farahmandi