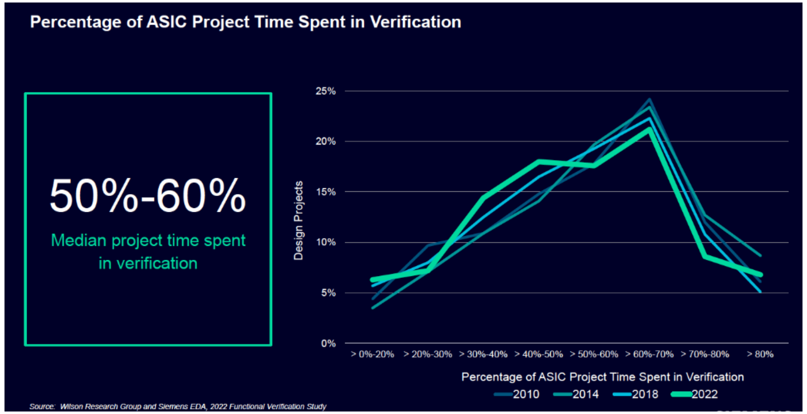
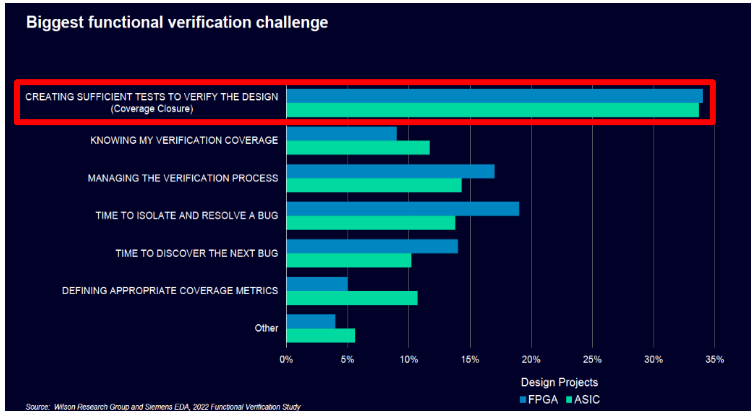
AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

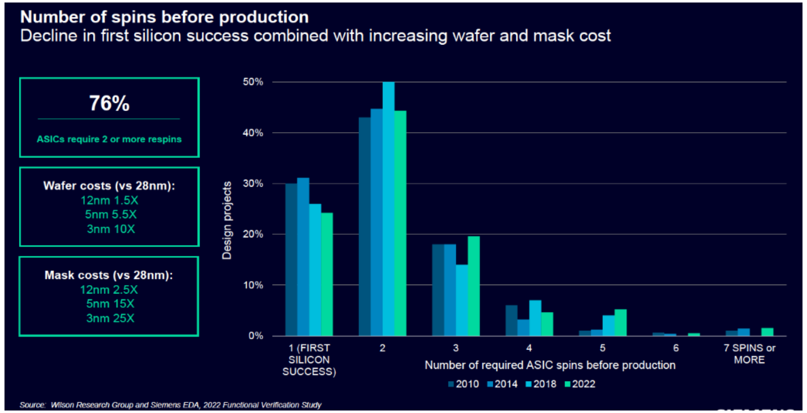
Auto-Generation of Verification Infrastructure for IP / SoC

Presenter : Nikita Gulliya
Product Engineer- Agnisys

The State of Verification in 2023







- Most development time spent in verification 60%-70%
- However, respins per project still increasing
- Greatest verification challenge (by far)…

  creating sufficient tests

Source: Wilson Research Group 2022, courtesy Siemens EDA

Verification Challenges

High Cost	● 70% of development effort is verification. ● Tools, simulation licenses, cost of computing power.
Difficulty	● Lack of qualified, experienced verification resources. ● Lot of manual work goes into translating designer intent into test code – error prone, tedious, not good use of time.
Horizontal Reuse	● Verification is not the end, there is also firmware, prototype, and validation that is needed. ● Typically these teams do not share code. ● Different environment, language, focus,etc.
Vertical Reuse	● Test sequences, register specification created at block or IP level can be run from subsystem or system level. ○ Changes in bus protocol. ○ Differences in configuration.

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Solution : Auto Generation of Verification infrastructure

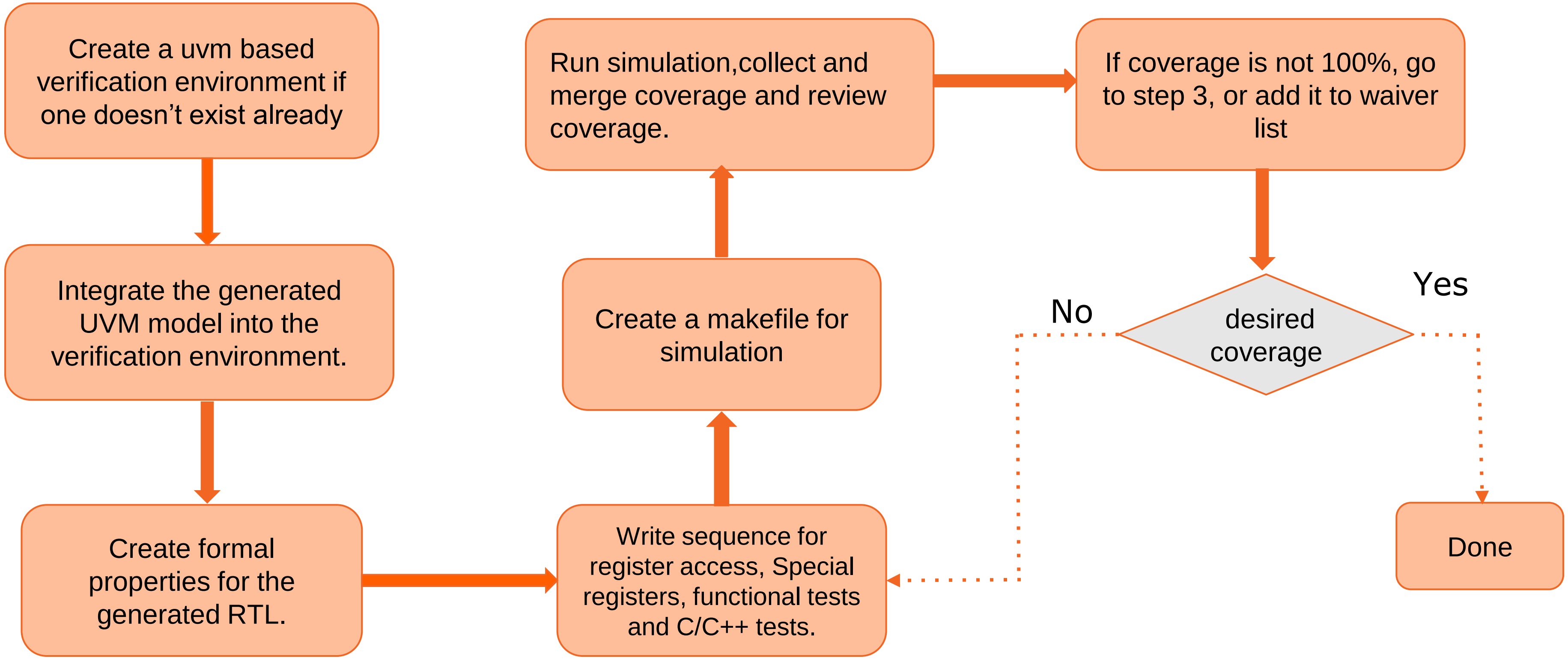
Generate Verification Infra from what?

- Single Golden Specification
  - Register Specification
    - SystemRDL, IP-XACT, …
  - Sequences Specification
    - PSS, …
  - Design Specification
    - FSM, Datapath, …
  - Connection Specification
    - Tcl, Excel, …

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Register Specification Automation

What's next after UVM RAL Model generation?


Create a uvm based verification environment if one doesn't exist already
Integrate the generated UVM model into the verification environment.
Create formal properties for the generated RTL.
Write sequence for register access, Special registers, functional tests and C/C++ tests.
Create a makefile for simulation
Run simulation,collect and merge coverage and review coverage.
If coverage is not 100%, go to step 3, or add it to waiver list
desired coverage
No
Yes
Done

Outputs from Register specification

Test Generation for registers

- Register model corresponding to the DUT specification with register focused **cover groups**
- Positive Test Sequences including tests for special registers and wide variety of register accesses.
- Reset Sequences
- Negative Test Sequences
- UVM Verification Env

Gets you close to 100% coverage

Assertion Generation for registers

- SV Properties and Assertions for register **access policies** and compliance to **bus protocols**
- Top-level file to bind the DUT RTL as well as third-party design IP with the assertions.
- Makefile or Tcl command scripts

Independent Verification required by Functional Safety

Verification Env for registers

- Generates the design components and corresponding RAL
- Generates **standard sequences** for registers
- Generates **custom test sequence** and user checker events
- Complete and fully-connected UVM test environment including components, hdl_paths, covergroups, constraints and illegal bins
- “Makefile” for various simulators

High-level test creation which runs in the UVM environment

More Features and Capabilities

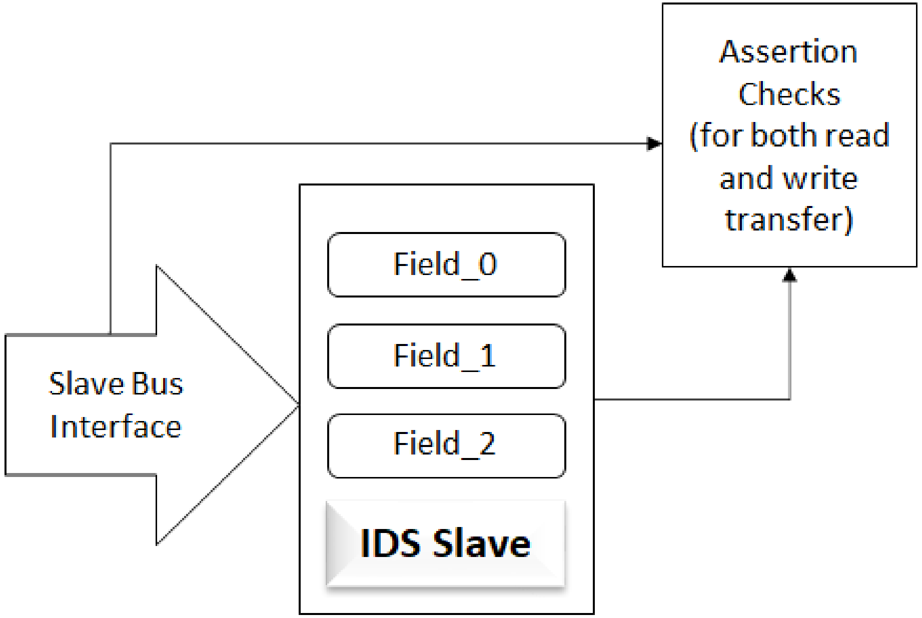
- A complete solution for
  - Complete UVM test bench environment
  - Automatic tests sequence for registers
  - Custom test sequences and checker events for user tests
  - Formal assertions based on design intent and registers
- Generate complete UVM infrastructure for a **push-button verification** with integrated RAL model and DUT
- Support for **multiple buses** such as AXI4Full, AXILite, AHB, APB etc.
- Out-of-the box ~**100% coverage** with register focused cover groups
- **Automatic test sequences** for special registers and wide variety of register accesses
- **Automatic assertions** for registers (including special registers), register access policies and bus compliance
- Generate "**Makefile**" (for Cadence Xcelium, Mentor Questa, and Aldec Riviera Pro) to run simulations and collect results from the database for different widely used simulators
- Provide **auto mirroring** of registers to continuously monitor the hardware events occurring on the registers
- Reduce the required verification effort by automatically generating **~30% of the verification code** making it an efficient and error free flow

Automatic Assertion Generation

What type of assertions are generated?

Assertions based on specification for different RTL behavior

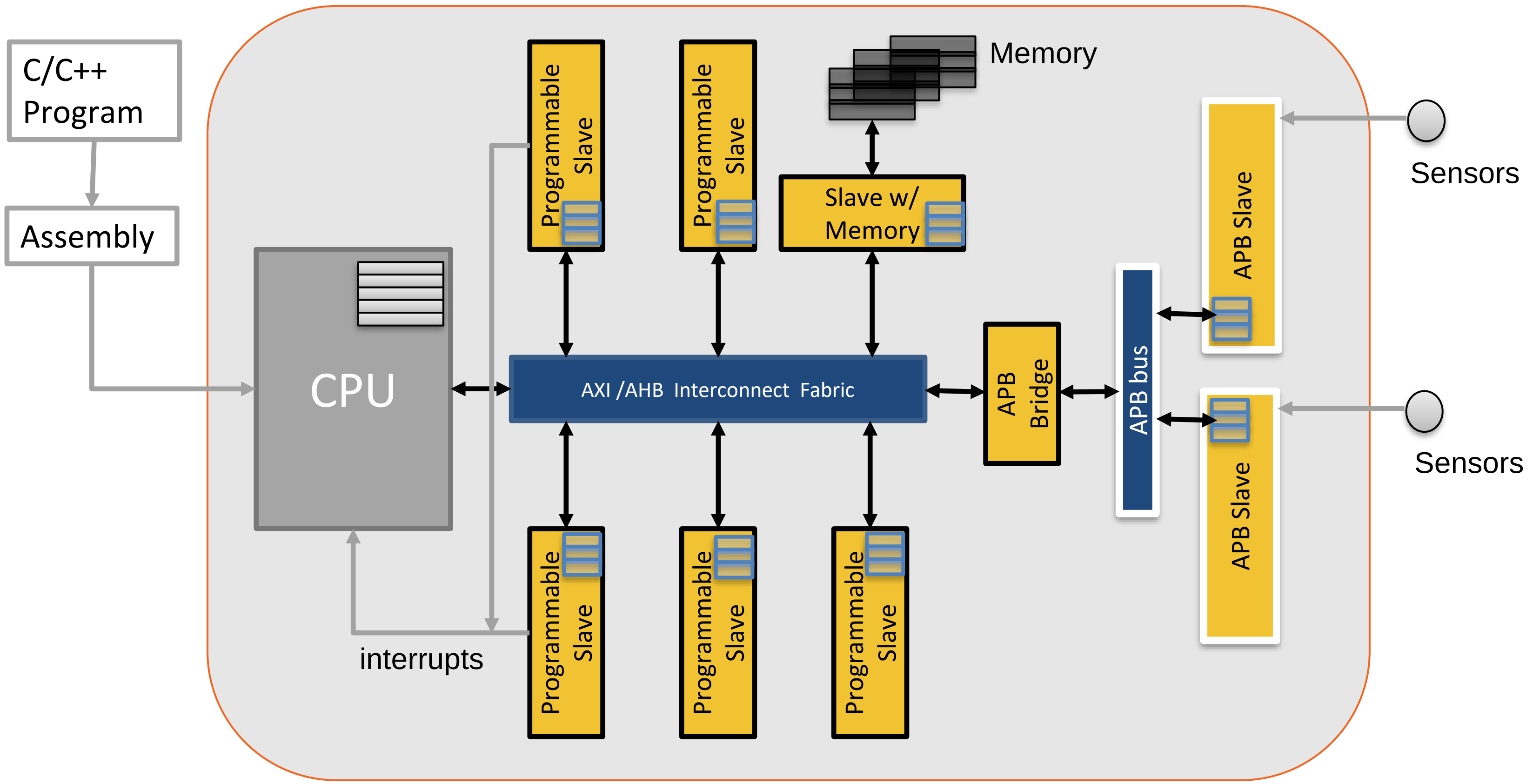
- Assertions for register accesses
- Assertions for external RTL interface
- Assertions for special registers
- Assertions for special IDS associated UDPs
- Assertions for clock domain crossing (CDC)
- Assertions for functional safety RTL design
- System level assertions
    - Assertions to check RTL ports connections
    - Topology assertions
    - Multi-master design assertions



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Design & Connection Specification Automation

A Typical SoC

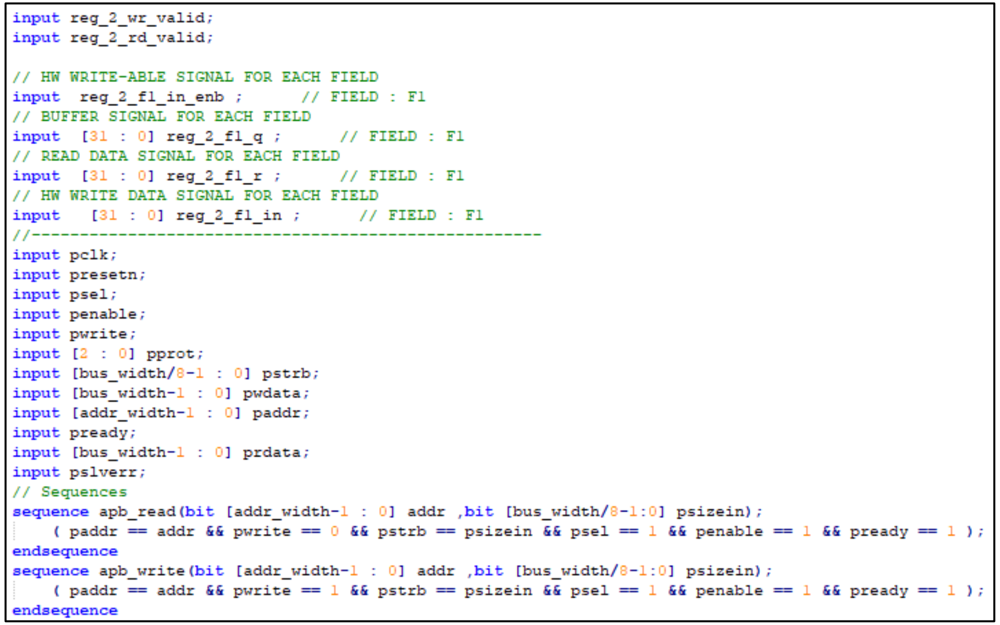


Assertions for IDS generated Register RTL

Input RDL to generate assertions

```
addrmap block_1 {
    reg reg_1 {
        field {
            hw = rw;
            sw = rw;
        } f1[31:0] = 32'h5;
    };
    reg_1 reg_1;
    reg reg_2 {
        field {
            hw = rw;
            sw = rw;
        } f1[31:0] = 32'h8;
    };
    reg_2 reg_2;
};
```

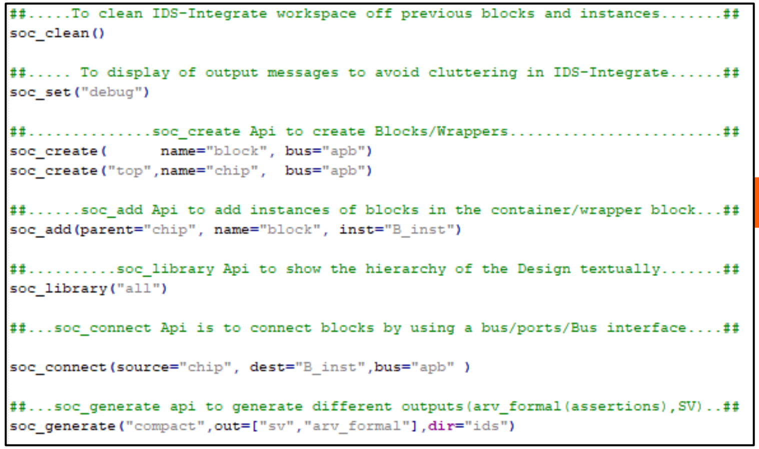
Generated RTL for Input RDL



```
input reg_2_wr_valid;
input reg_2_rd_valid;

// HW WRITE-ABLE SIGNAL FOR EACH FIELD
input  reg_2_f1_in_enb ;      // FIELD : F1
// BUFFER SIGNAL FOR EACH FIELD
input  [31 : 0] reg_2_f1_q ;      // FIELD : F1
// READ DATA SIGNAL FOR EACH FIELD
input  [31 : 0] reg_2_f1_r ;      // FIELD : F1
// HW WRITE DATA SIGNAL FOR EACH FIELD
input   [31 : 0] reg_2_f1_in ;      // FIELD : F1
//-----------------------------------------------------
input pclk;
input presetn;
input psel;
input penable;
input pwrite;
input [2 : 0] pprot;
input [bus_width/8-1 : 0] pstrb;
input [bus_width-1 : 0] pwdata;
input [addr_width-1 : 0] paddr;
input pready;
input [bus_width-1 : 0] prdata;
input pslverr;
// Sequences
sequence apb_read(bit [addr_width-1 : 0] addr ,bit [bus_width/8-1:0] psizein);
    ( paddr == addr && pwrite == 0 && pstrb == psizein && psel == 1 && penable == 1 && pready == 1 );
endsequence
sequence apb_write(bit [addr_width-1 : 0] addr ,bit [bus_width/8-1:0] psizein);
    ( paddr == addr && pwrite == 1 && pstrb == psizein && psel == 1 && penable == 1 && pready == 1 );
endsequence
```

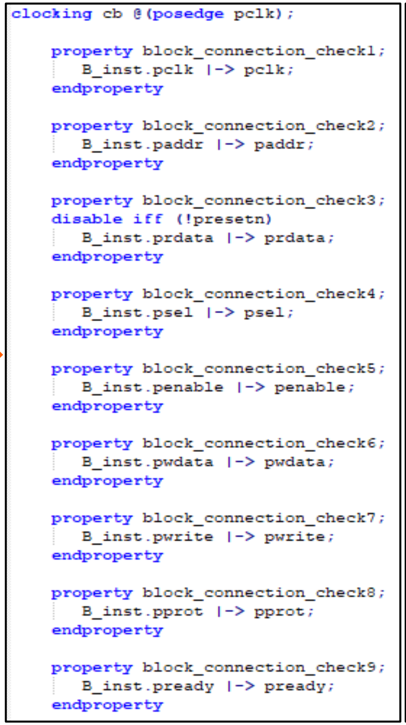
Assertion Generation from Connectivity Spec

Input python script to generate assertions

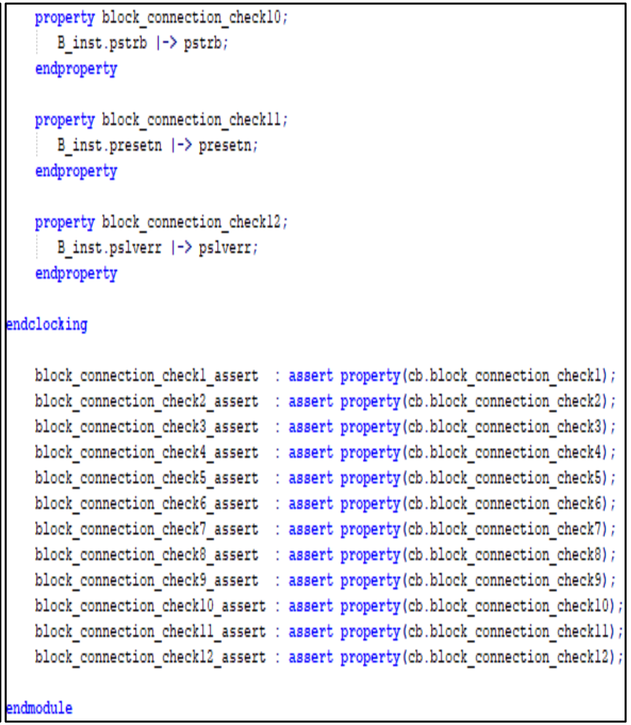


```
##.....To clean IDS-Integrate workspace off previous blocks and instances.......##
soc_clean()

##..... To display of output messages to avoid cluttering in IDS-Integrate......##
soc_set("debug")

##.............soc_create Api to create Blocks/Wrappers........................##
soc_create(      name="block", bus="apb")
soc_create("top",name="chip",  bus="apb")

##......soc_add Api to add instances of blocks in the container/wrapper block...##
soc_add(parent="chip", name="block", inst="B_inst")

##..........soc_library Api to show the hierarchy of the Design textually.......##
soc_library("all")

##...soc_connect Api is to connect blocks by using a bus/ports/Bus interface....##

soc_connect(source="chip", dest="B_inst",bus="apb" )

##...soc_generate api to generate different outputs(arv_formal(assertions),SV)..##
soc_generate("compact",out=["sv","arv_formal"],dir="ids")
```

Generated output assertions by IDS-Integrate



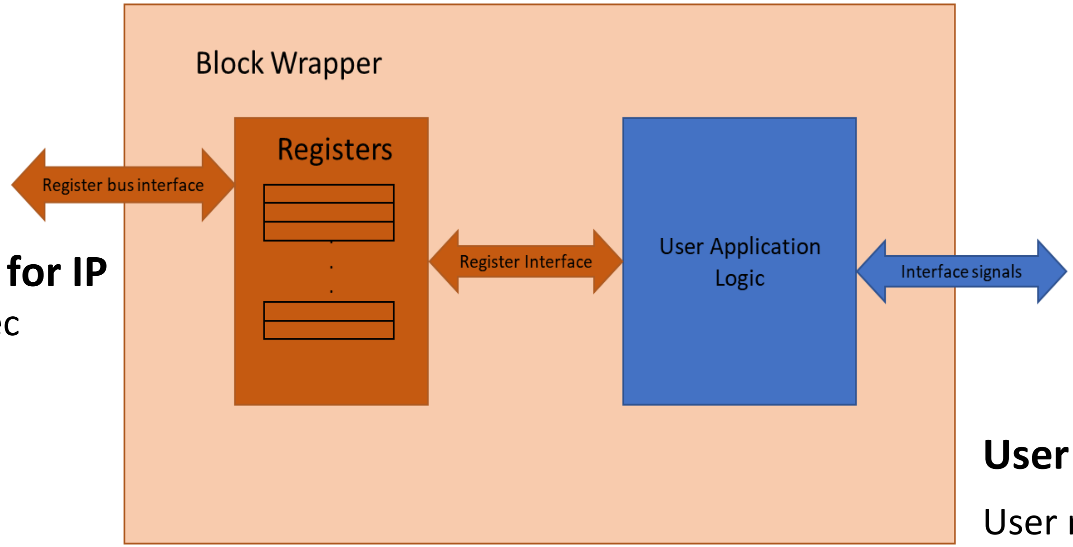
```
clocking cb @(posedge pclk);

    property block_connection_check1;
      B_inst.pclk  |-> pclk;
    endproperty

    property block_connection_check2;
      B_inst.paddr  |-> paddr;
    endproperty

    property block_connection_check3;
    disable iff  (!presetn)
      B_inst.prdata  |-> prdata;
    endproperty

    property block_connection_check4;
      B_inst.psel  |-> psel;
    endproperty

    property block_connection_check5;
      B_inst.penable  |-> penable;
    endproperty

    property block_connection_check6;
      B_inst.pwdata  |-> pwdata;
    endproperty

    property block_connection_check7;
      B_inst.pwrite  |-> pwrite;
    endproperty

    property block_connection_check8;
      B_inst.pprot  |-> pprot;
    endproperty

    property block_connection_check9;
      B_inst.pready  |-> pready;
    endproperty
```



```
    property block_connection_check10;
      B_inst.pstrb |-> pstrb;
    endproperty

    property block_connection_check11;
      B_inst.presetn |-> presetn;
    endproperty

    property block_connection_check12;
      B_inst.pslverr |-> pslverr;
    endproperty

endclocking

    block_connection_check1_assert  : assert property(cb.block_connection_check1);
    block_connection_check2_assert  : assert property(cb.block_connection_check2);
    block_connection_check3_assert  : assert property(cb.block_connection_check3);
    block_connection_check4_assert  : assert property(cb.block_connection_check4);
    block_connection_check5_assert  : assert property(cb.block_connection_check5);
    block_connection_check6_assert  : assert property(cb.block_connection_check6);
    block_connection_check7_assert  : assert property(cb.block_connection_check7);
    block_connection_check8_assert  : assert property(cb.block_connection_check8);
    block_connection_check9_assert  : assert property(cb.block_connection_check9);
    block_connection_check10_assert : assert property(cb.block_connection_check10);
    block_connection_check11_assert : assert property(cb.block_connection_check11);
    block_connection_check12_assert : assert property(cb.block_connection_check12);

endmodule
```

AGNISYS SYSTEM DEVELOPMENT WITH CERTAINTY

User application logic connected to register map



Addressable registers for IP

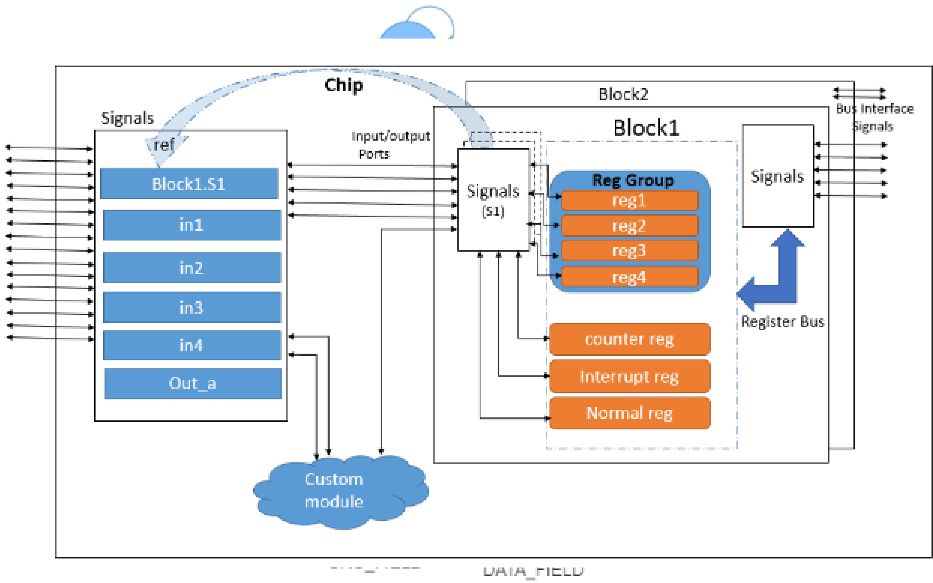
Generated by IDesignSpec

User application logic

User needs to create manually

User Application Logic Constructs

- User logic constitutes mainly of:
  - State Machine
  - Assigns with combinatorial logic
  - Signals input/output
  - Sequential Elements/Data Path



Additional User Specification in form of Templates

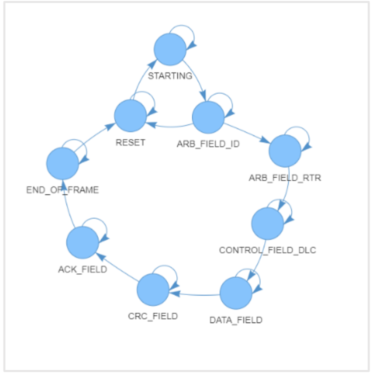
- Create a formal specification of the requirements
- Automatically generates:
    - UVM prediction model
    - UVM test sequences for complete coverage
    - Other collaterals are possible
        - Deadlock detection
        - Documentation
    - In addition, RTL and SystemC models can be generated

Tackling Verification using AI

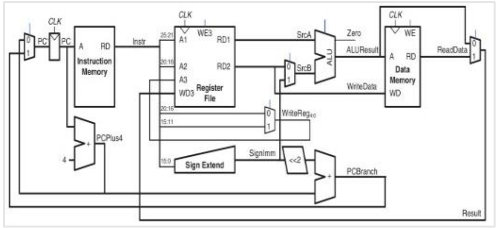
- Alleviates significant verification burden: automatic test generation using *DRL (uniquely enabled by Agnisys reward function) for registers, FSM, datapath & interconnect
- 100% Code / Functional Coverage based on Coverage Goals
- Automatic Output generation: RTL, UVM RAL, UVM Tests

IP Specification

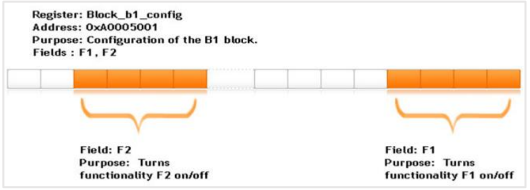
FSM



Datapaths



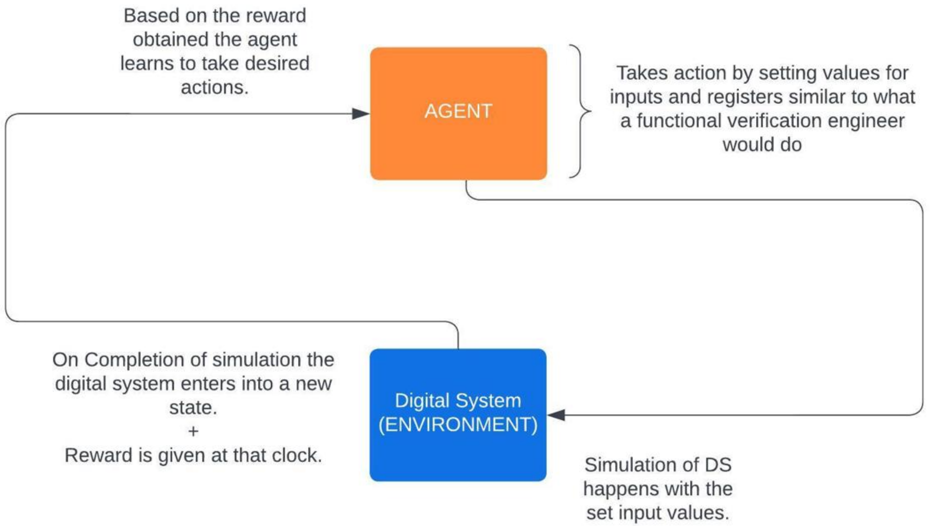
Registers



Register: Block_b1_config
Address: 0xA0005001
Purpose: Configuration of the B1 block.
Fields : F1, F2

Field: F2
Purpose: Turns functionality F2 on/off

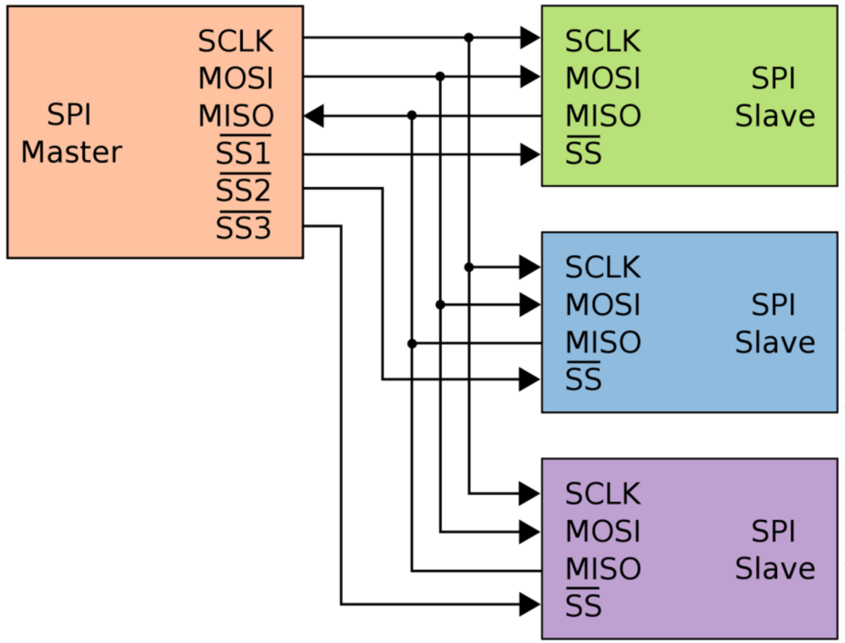
Field: F1
Purpose: Turns functionality F1 on/off

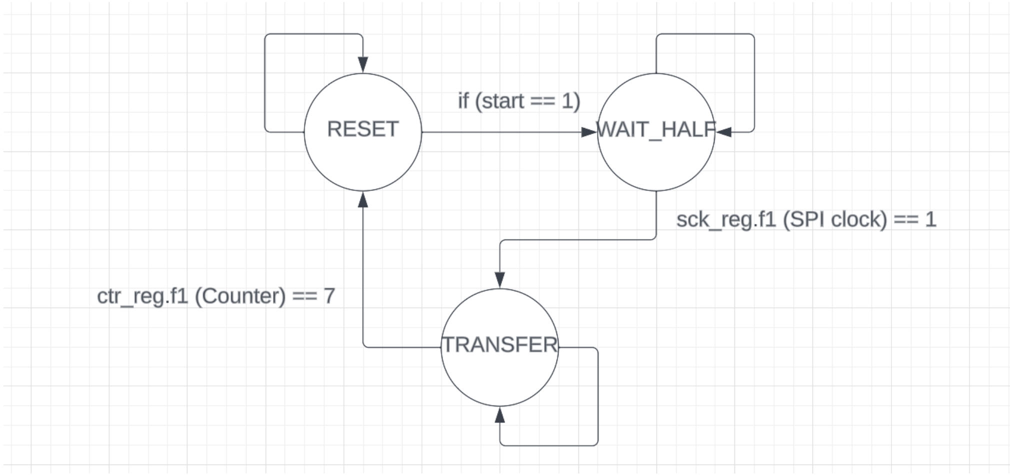

Based on the reward obtained the agent learns to take desired actions.
AGENT
Takes action by setting values for inputs and registers similar to what a functional verification engineer would do
On Completion of simulation the digital system enters into a new state.
+
Reward is given at that clock.
Digital System (ENVIRONMENT)
Simulation of DS happens with the set input values.

*Deep Reinforcement Learning (DRL)

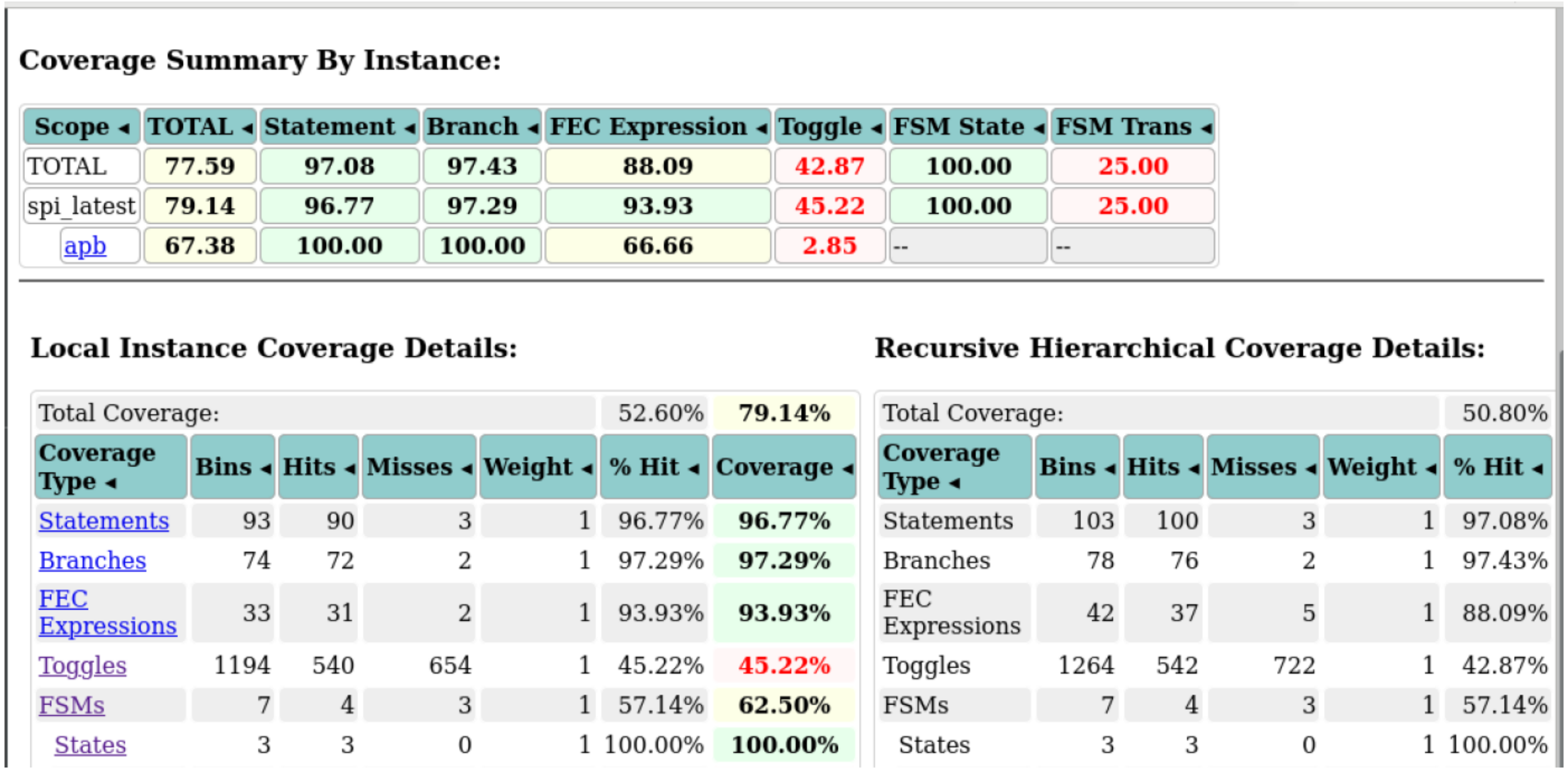
- Combines deep learning and unique reinforcement learning
- Efficient exploration of complex digital designs
- Significant increases in code coverage using fewer test cases

SPI Design FSM and Block Diagram





SPI Design Coverage Results



Coverage Summary By Instance:

Scope ◂	TOTAL ◂	Statement ◂	Branch ◂	FEC Expression ◂	Toggle ◂	FSM State ◂	FSM Trans ◂
TOTAL	77.59	97.08	97.43	88.09	42.87	100.00	25.00
spi_latest	79.14	96.77	97.29	93.93	45.22	100.00	25.00
apb	67.38	100.00	100.00	66.66	2.85	--	--

Local Instance Coverage Details:

Total Coverage:					52.60%	79.14%
Coverage Type ◂	Bins ◂	Hits ◂	Misses ◂	Weight ◂	% Hit ◂	Coverage ◂
Statements	93	90	3	1	96.77%	96.77%
Branches	74	72	2	1	97.29%	97.29%
FEC Expressions	33	31	2	1	93.93%	93.93%
Toggles	1194	540	654	1	45.22%	45.22%
FSMs	7	4	3	1	57.14%	62.50%
States	3	3	0	1	100.00%	100.00%

Recursive Hierarchical Coverage Details:

Total Coverage:					50.80%
Coverage Type ◂	Bins ◂	Hits ◂	Misses ◂	Weight ◂	% Hit ◂
Statements	103	100	3	1	97.08%
Branches	78	76	2	1	97.43%
FEC Expressions	42	37	5	1	88.09%
Toggles	1264	542	722	1	42.87%
FSMs	7	4	3	1	57.14%
States	3	3	0	1	100.00%

AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Sequence Specification Automation

What is Verification?

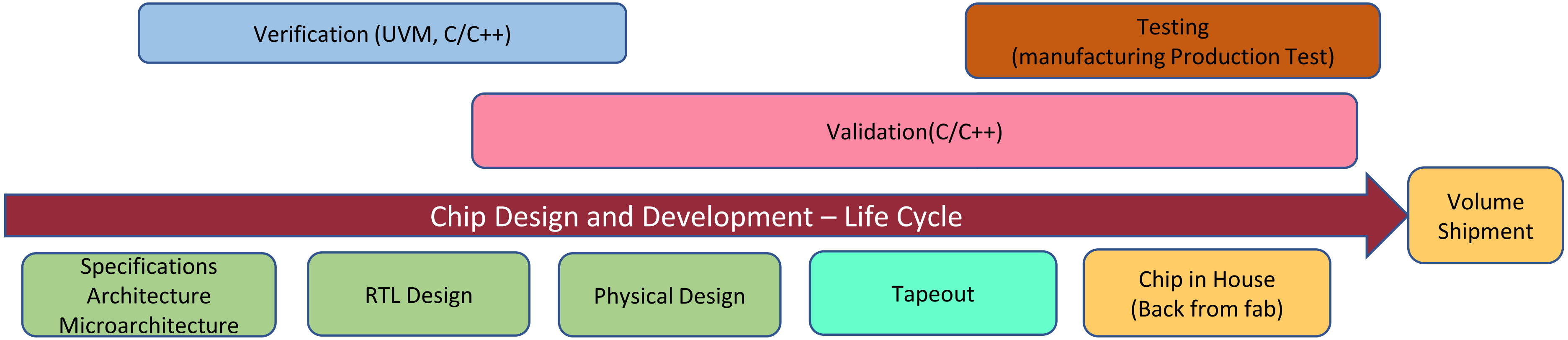
- A process to test a design is against a given specification to ensure its functional correctness
- Functional coverage using cover-points, constraints and randomization
- Includes Simulation along with Formal, Emulation and Prototyping
- A typical SV-UVM based environment is needed

What is Validation?

- A process in which the manufactured design (chip) is tested for all functional correctness in a lab setup
- Bare-Metal Environment with emulation of real processor and IP's
- Software test mainly c-tests including :
  - Automated C-tests
  - Custom C-tests

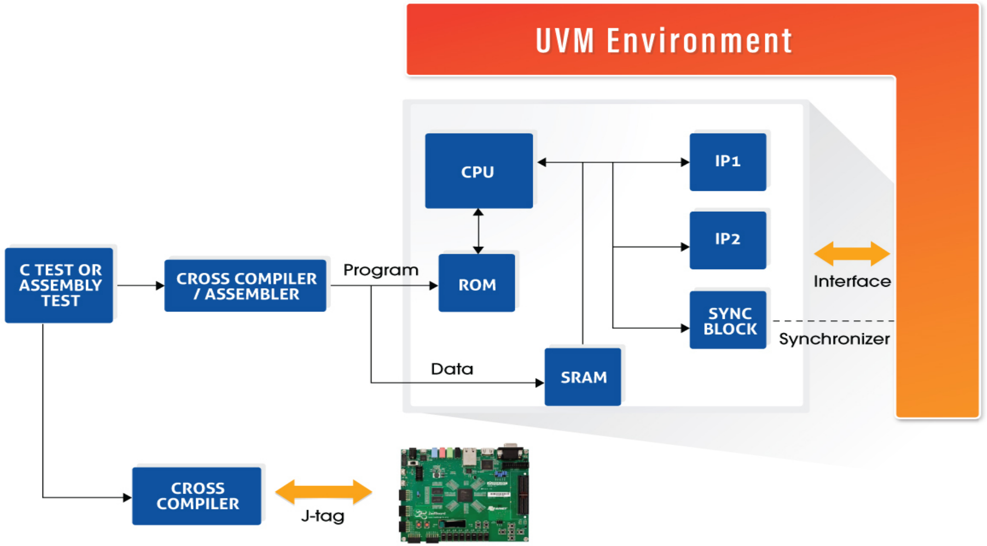
SoC Verification and Validation

- Is the testing/ verification of a Design against a given specification before actual tape-out to ensure functional correctness
- Can be during Design development or start from early Design architecture/ microarchitecture definition
- Involves validating the Chip-in-a-System level environment with real software running on the hardware


Verification (UVM, C/C++)
Testing (manufacturing Production Test)
Validation(C/C++)
Chip Design and Development – Life Cycle
Volume Shipment
Specifications Architecture Microarchitecture
RTL Design
Physical Design
Tapeout
Chip in House (Back from fab)

Bare-Metal Verification

- The “bare metal environment” include **not only** the IPs but also the **CPU** in the simulation
- Bare-metal programming means writing an application directly on your hardware without using an external application programming interface or any operating system
- The tests are written in C language and are compiled and run on the CPU
- **Salient features**
  - System level UVM-C mixed verification environment built around RISC-V VeeR core
  - Automatic generation of C based tests from Register Layer
  - C tests synced with UVM tests
  - Custom C/UVM tests



Challenges Development Teams face with Sequences

Has this ever happened to you?

- Sequence is not clear or well documented
- A sequence works on one platform and not on other
- No way to create the same debug environment on multiple platforms

Inconsistent definition of Sequences

- In-exact definition
- Inconsistent interpretation
- Incorrect implementation

Sequences contain Register data that can be in any format:

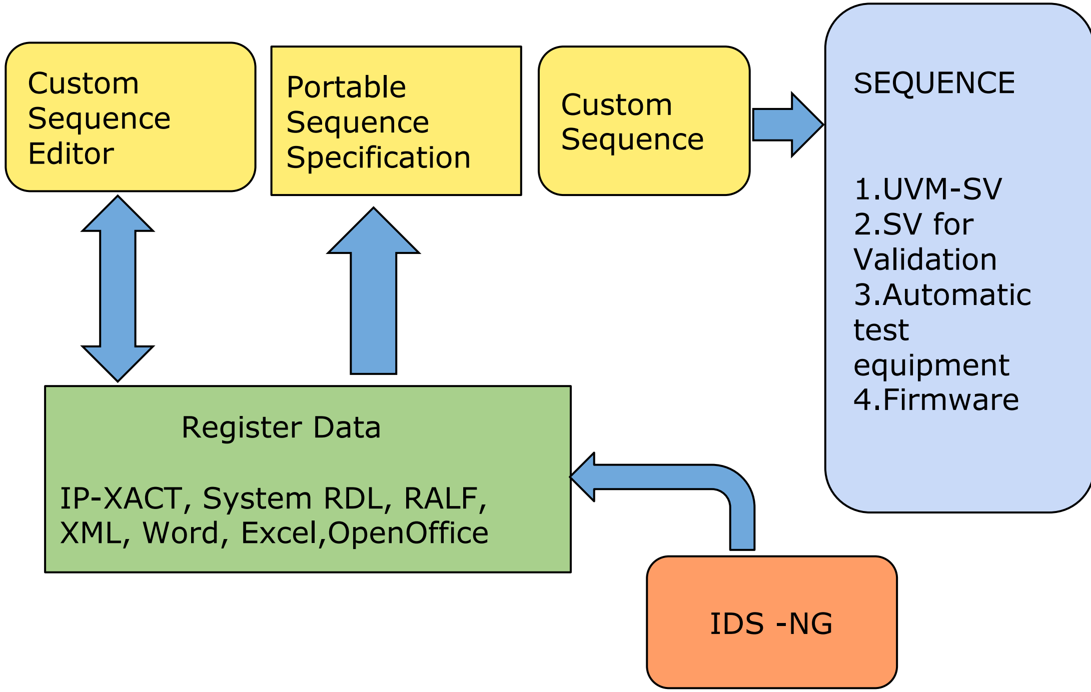
- Industry Standards – IP-XACT, SystemRDL, RALF
- Custom formats – CSV, Excel, XML
- IDesignSpec formats – IDS-NG, IDSWord, IDSExcel

Sequences are everywhere

- Architects/designers plan them
- Design engineers encode Verilog functionality
- Verification engineers write them in UVM or PSS

Custom Sequence Generation

- IDS-Validate™ enables users to describe the programming and test sequences of a device and automatically generate sequences ready to use from an early design and verification stage to post silicon validation
- Centralize creation of sequences from a single specification and generate various output formats for multiple SoC teams.
  - UVM, System Verilog, C, TCL, CSV or MATLAB
  - HTML
  - Platform
- Specify portable sequences for multiple IPs at a higher level in-sync with the register specification.
- Use register descriptions in standard formats such as IP-XACT, SystemRDL, RALF or leverage IDesignSpec integrated flow to use the register data.


Custom Sequence Editor
Portable Sequence Specification
Custom Sequence
SEQUENCE
1.UVM-SV
2.SV for Validation
3.Automatic test equipment
4.Firmware
Register Data
IP-XACT, System RDL, RALF, XML, Word, Excel,OpenOffice
IDS -NG

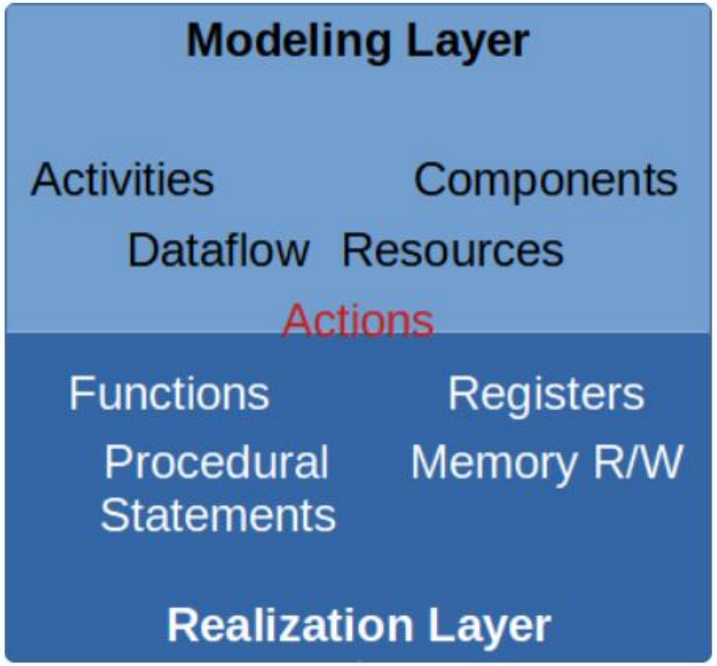
PSS Support

- PSS 2.0 is a new* industry standard created by Accellera
- Agnisys has been a member and contributed to it

We are an expert in creating the Realization layer
  Widest/Most comprehensive Register/Memory definition
  Pioneer in Sequence/Functions for IP/SoC

What we offer
- Use PSS (or Excel, Python, GUI (NG)) to create Golden Spec for Sequences
- Generate C functions and UVM Sequences

Benefits
- Single Golden Source for Registers and Sequences reduces Time to Market, improves quality



AGNISYS
SYSTEM DEVELOPMENT WITH CERTAINTY

Conclusion:

Auto Generation of 50%-60% of Verification Collateral is possible today.
This number will only grow with Specification Automation and AI.