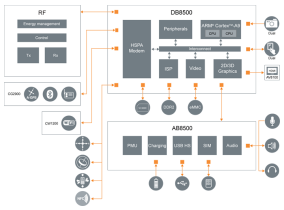
SoC Verification

Mike Benjamin
Associate at TVS

What is SoC level?



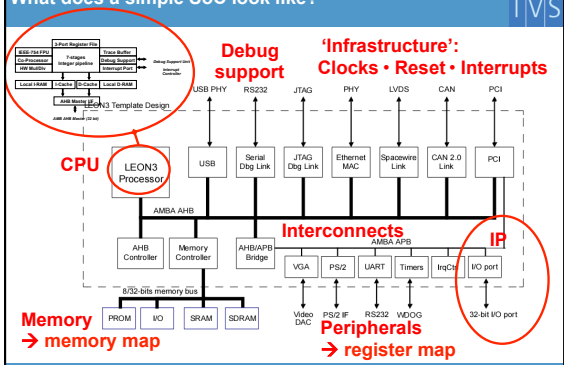
- **Top level**
  Looking at the complete design
- **System Level**
  Putting the complete design in a wider context …

System architecture

Partner IP

Software

What does a simple SoC look like?



Why write SoC level tests?

- **Some top level functionality not visible at unit level**
  - Imported IP
  - Register / address mapping
  - Signal connectivity
  - Performance verification
  - Power management
  - Power on / reset
  - Coherence?
  - Clocking strategy
  - Benchmarking
- **Allows verification to focus on actual use model**
  - Testing restricted to real use model
  - Configurability / parameterized blocks instantiated!
  - Generate typical/worst case waveforms for power analysis!
- **Missing system level functionality & compliance testing**
  - Partner IP
  - Software
  - System architecture

Why bother doing unit level testing?

- **Controllability at top level v unit level?**
  → **REDUCED**
  - **Harder to hit corner case and longer run times**
- **Visibility at top level v unit level?**
  → **REDUCED**
  - **Harder to debug fails**
- **Overhead on testing at top level v unit level?**
  → **INCREASED**
  - **Need working top level integration before testing**
  - **Need to propagate block level fixes/changes to top level before they can be tested**
  - **Need to understand the complete SoC to test and debug a single block**

Barriers to top level testing

- **Barriers to top level verification?**
  - **B1:** Complexity of building the complete top level design
  - **B2:** Late availability of key blocks / functionality
  - **B3:** Difficulty of anyone understanding the complete design
  - **B4:** Size of full top level design
  - **B5:** Limited controllability of the design from outside
  - **B6:** Limited visibility inside design

- **Solutions?**
  - **S1:** Require changes to be co-ordinated between dependent blocks
  - **S2:** Regression testing before changes are committed
  - **S3:** A schedule defining milestones for delivering features
  - **S4:** Ensure major interfaces are stable and well defined
  - **S5:** Black box some components
  - **S6:** Replace components with abstract models or BFMs (eg: CPU, memories)

Reuse from unit level?

- VIP
  - BFMs
  - Monitors and scoreboards
  - Protocol checkers
- Assertions
- Functional coverage points
- Tests
  - Integration tests
    - Connectivity, address mapping
  - Stress tests
    - Cross cutting concerns such as interrupts or power management
    - Shared resources or 'convergence points' (eg: memory synchronisation)
  - Right level of abstraction
    - Transactions and/or bus accesses
    - Relative address map

Need to plan for reuse!

Test and Verification Solutions

What do our top level tests contain?

- Tests are typically C programs running on an SoC CPU
- Loaded into SoC memory
- Component tests
- Register / address map
- Result checking
- Trace and error reporting
- Halt mechanism
- Interrupt handling

```
main(){
    report_start();
    leon3_test(1, 0x80000200, 0);
    irqtest(0x80002001);
    gptimer_test(0x80000300, 8);
    gpio_test(0x80000700);
    report_end();}
```

```
int gpio_test(int addr)
{
pio = (int *) addr;
int mask;
int width;

report_device(0x0101a000);
pio[3] = 0; pio[2] = 0; pio[1] = 0;
pio[2] = 0xFFFFFFFF;

/* determine port width and mask */
mask = 0; width = 0;

while( ((pio[2] >> width) & 1) && (width <= 32)) {
    mask = mask | (1 << width);
    width++;}

pio[2] = mask;
if( (pio[0] & mask) != 0) fail(1);
pio[1] = 0x89ABCDEF;
if( (pio[0] & mask) != (0x89ABCDEF & mask)) fail(2)
pio[2] = 0;

return width;}
```

Test and Verification Solutions

How to check the test results

- Fail causes test to hang
- Dump results to memory and compare to reference results from model
  - mpeg decoder video stream
  - reference simulator
- Explicit checks in the test
  - Observe and count interrupts
  - Check data values
- Trace comparison
  - Compare simulation state to a reference model cycle by cycle during the simulation
- Use of monitors, scoreboards or assertions

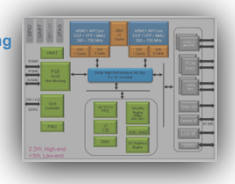
Need error propagated to end of test

Sensitive to accuracy of reference model (especially timing)

Test and Verification Solutions

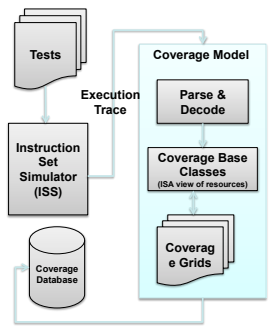
Methodology for top level testing

1. Pipe cleaning flow with regression tests
   → to verify basic functionality is not broken
2. Incremental test set verifying the subsets of functionality
   → scope grows with successive builds
3. Architectural and conformance tests
4. Micro-architectual tests
5. Soak testing
6. Performance testing and benchmarking



Test and Verification Solutions

Adding Coverage



Why add coverage?

- Conformance testing:
  - Need complete coverage of cases
- Targeting specific scenarios:
  - Hitting required corner cases
- Soak testing
  - Ensure testing is not becoming repetitive

Test and Verification Solutions

How to further increase the 'stress'

- Build multiple configurations (set at build time)
  - Increase stress by maximising corner cases
    eg: small memories or FIFOs
  - Increase stress my maximising 'synchronisation points'
    eg: shared resources or coherent memories
- Chicken bits (set at start of test)
  - Turn features on or off (can be verification specific or used to minimise design risk by disabling potentially risky optimisations)
- Hot load (set at start of test)
  - Can force states of part of the design into conditions that maximise chance of hitting corner conditions early (most often hot load caches but can also leave holes or create dirty entries)
- Use of irritators (set during test)
  - Hardware/DMA data transfers/traffic generators and BFMs (bursts of traffic and corner cases for transaction timing)

Test and Verification Solutions

The New Verification Challenges of Low Power Design

- **Why does low power matter?**
  - Battery life eg: mobile devices
  - Operating temperature and cooling requirements eg: automotive, data centres
- **How to achieve low power?**
  - Dynamic power = switching flops
  - Static power = leakage current
- **Minimize Switching by Design**
  - Clock gating: *Inferred* (by synthesis) and *architectural*
- **Turn off units (eg: run fast then stop)**
  - Multiple *power domains* and *power modes* managed by a 'Power Management Unit' (PMU)
  - System level (eg: ARM big.LITTLE architecture)
- **Some of the new Verification challenges**
  - Ensure correct state retention and restoration when switching a power domain.
  - Clamping inactive signals at the boundary of a power domain
  - Ensure the design doesn't try to use a unit that is (being) switched off!
  - Can the design get stuck in a power mode?
    eg: interacting state machines restored to states that cause deadlock or livelock
  - Errors in the sequences of save and restore operations performed by the PMU
  - Interaction of the power modes with chip level power on, off and reset!

Energy efficiency | Power dissipation | Minimise switching | Turn off!

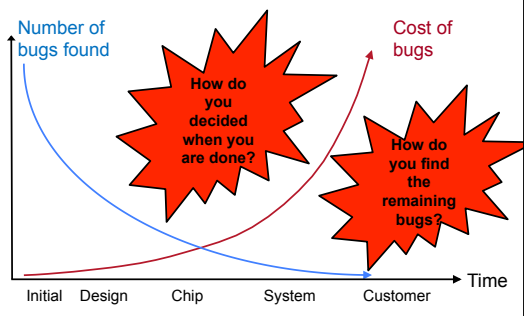
Power Aware Simulation

- **Need to tell your RTL simulations about low power intent**
  - A common description shared between simulation, synthesis and layout
- **Two competing standards: UPF and CPF**
  - Both extend the functional description without changing the existing RTL
- **What they describe…**
  - Power domains, supply rails and switches eg:
    create_power_domain pdA –include_scope moduleA
    Create_supply_net RETENTION –domain pdA
  - State retention and isolation
  - System power states
    … the number of state combintions can be large!
- **For an RTL simulation 'OFF' means**
  - All 'OFF' registers are corrupted
  - Any signals driven by logic that is 'OFF' are corrupted
  - No evaluation of logic that is 'OFF'
- **Can also describe other features…**
  - Multi-voltage designs and level shifters
  - Voltage and frequency scaling

Retention registers have separate supplies

Isolation cells have separate supplies

Cost of bugs over time (revisited)



Being pro-active to improve verification

- **Achieving the best possible test plan**
  - Methodical analysis of design specifications and extraction of features
  - Brainstorming and reviewing within the development team
  - Refinement and maintenance throughout the development process
  - Tracking and sign-off of verification deliverables against the test plan
- **Make the design 'verification friendly' (design for verification)**
  (High quality products are a combination of robust and extensive verification with good design practices)
  - Ensure good visibility of architectural and micro-architectural corner cases
  - Avoid **unnecessary** functional complexity eg: excessive configurability, irregular structures
  - Understand the verification impact of design changes (eg: code churn during optimization)
  - Designers document their intent and assumptions, especially at interface between units
  - Ensure the architecture, specifications and design are as stable as possible

Verification Requirements Specification

Communicate!

(Verification is not just the responsibility of verification Engineers)

- **Engage closely with the designers**
- **Be an active participant in reviews**
- **Take every opportunity to get the widest possible input into verification planning**

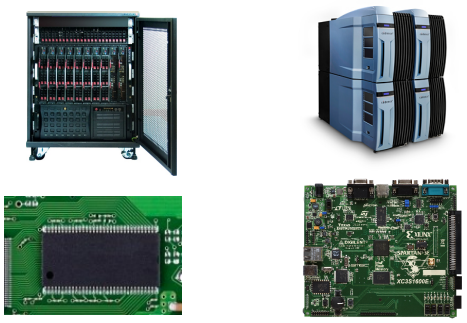
Is block and top level verification sufficient?

- **Is block level and top level verification sufficient?**
  - Verification of IP in System context
  - Verifying correct operation with related IP
  - Verification of complete systems (both HW and SW)
    - Software conformance testing
    - Soak testing
- **Soak testing at system level?**
  - Focus at system level is shared resources
    eg: coherent memory system
  - Running irritator software in parallel on multiple threads or multiple CPUs (minimal OS)
  - Switching CPUs (eg: swapping big/LITTLE)
  - Virtualisation

What goes wrong at system level?

- **Integration bugs**
  - Connecting a big-endian subsystem to a little-endian sub-system
- **Clocks and power**
  - System hangs following mode change
- **Concurrency and shared resources**
  - Concurrent memory gets corrupted
- **Performance**
  - Bus bandwidth and latency is much worse than predicted

How to go faster!
Compute Farm, Emulators, FPGA and test chips



The 'tradeoffs' for different platforms

Favours lots of short tests!

… but also need to load tests and dump test results!

	Compute farm	Emulator	FPGA	Test chip
Speed	10Hz - 100Hz …per machine	1MHz	2MHz – 50MHz	GHz
Observability	Total	Trace window + host debug	Probes + host debug	Host debug
Behavioural testbench?	Yes	Co-emulation (speed penalty)	Co-emulation (speed penalty)	No
Test in 'real world' systems	No	Host debug + ICE with speed bridges	Mostly	Yes
Are fails easily reproducible in simulation?	Yes	Yes	No	No
Bring-up time	Minutes	Weeks → hours	Weeks → Days	Months

Partitioning!

Complex timing dependencies

Depends on process maturity

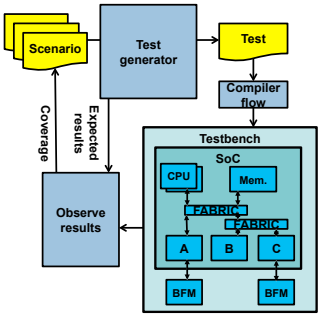
Summary

- What is SoC level verification? (Top v System)
- Looked at structure of a simple SoC
- Why do both 'SoC level' & 'unit level' verification?
- A methodology for SoC level verification
- System level verification

If time permits ….

- RIS (Random Instruction Stream) Test Generators
- Looked at IP-XACT

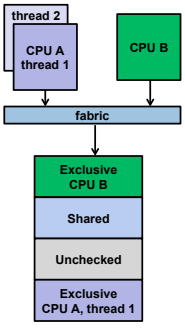
Top Level Test Generation



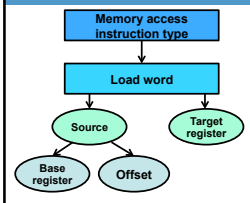
- Bias tests to hit interesting corner cases
  - Scenario interleaving
  - Target shared resources/'points of convergence'
- Non-repetitive useful tests
- There should be an efficient workflow
  - Generation performance
  - Target diverse platforms
  - Ease of use
  - Maintainability
  - Reuse (of testing knowledge)
  - Effective result checking:
    - Propagation of results
    - Trace comparison

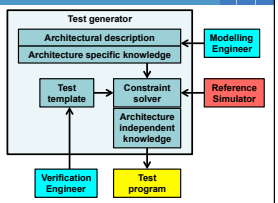
'Point solutions' for test generation
Memory Coherence

- Time sharing a resource (memory)
  - Coherency
  - Memory protection
- Most interesting cases are overlapping accesses
- Colliding access can be:
  Write||Write, Write||Read, Read|| Read
  eg: PowerPC: 'store quadword' || 'load quadword'
- True sharing: same memory
- False sharing: close enough to interfere
  (eg: same cache line)
- MP memory model can have weak ordering (with barriers)
  (W(a,d1) || W(a,d2)) → M(a) = {d1, d2}



A general purpose test generator for CPUs



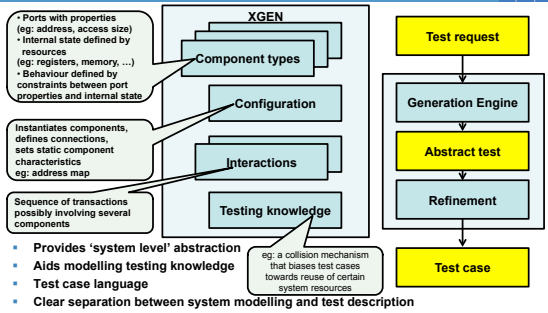


- Constraints (relations between attributes)
  eg: source.address = base.data+displacement.data
  PageCross(source.address)
  - hard or soft?
- Typically several weakly coupled constraints
- Randomize all other parameters and events
  eg: cache event in parallel with load
- Huge domains (eg: 2^64 address and data)
- Randomly sample solution space

- As resources are 'used up' it can become harder to solve constraints. Solutions are:
  - Register reloading
  - Backtrack and retry
- Generating loops is a challenge:
  - Procedure calls
  - Recurring interrupts
  - Self modifying code

  → Prevent random re-entrant code

Model Based Test Generation for SoCs?



- Provides 'system level' abstraction
- Aids modelling testing knowledge
- Test case language
- Clear separation between system modelling and test description
  eg: different SMP clusters will have same interactions and components and only the configuration will change
- Expects a separate checking mechanism

Test and Verification Solutions

IP-XACT

IP-XACT is:

- A standard XML scheme for describing components and connections.
- It describes things like interfaces (<spirit:busInterfaces>) and registers (<spirit:memoryMaps>) rather than function!

Tools can then generate and manipulate the metadata:

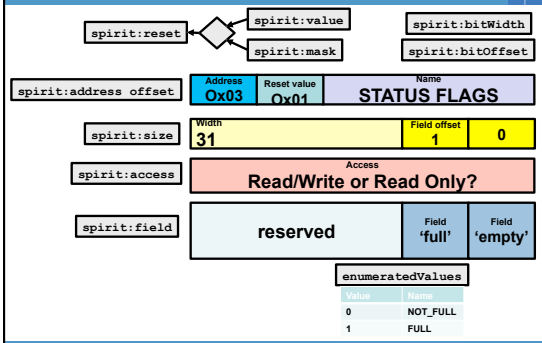
- Packagers: Generate 'sound' meta data for components
- Generators: Configure components where IP blocks and the design may both have generic parameters
- Assemblers & SoC design tools: Create an IP-XACT description of the design that can be used to automatically stitch together the components

Why is it useful to have a standard for documenting IP?

- Provides a common specification that can be shared between: SoC design, verification, software and documentation teams
- Vendor neutral: exchange libraries and combine components from multiple sources
- Allows automation of SoC and test bench assembly … a manual process is very error prone as number of components increases!

Test and Verification Solutions

IP-XACT example: IP register description



Test and Verification Solutions

IP-XACT example: Code

```
<spirit:register>
    <spirit:name> STATUS_FLAGS</spirit:name>
    <spirit:description> Register contains flags to report if FIFO empty or full </spirit:description>
    <spirit:dim>1</spirit:dim>
    <spirit:size>32</spirit:size>
    <spirit:access>read-write</spirit:access>
    <spirit:reset>
          <spirit:value>1</spirit:value>
          <spirit:mask>3</spirit:mask>
    </spirit:reset>
    <spirit:field>
          <spirit:name>EMPTY</spirit:name>
          <spirit:description>FIFO empty flag</spirit:description>
          <spirit:bitOffset>0</spirit:bitOffset>
          <spirit:bitWidth>1</spirit:bitwidth>
          <spirit:access>read-only</spirit:access>
    </spirit:field>
    <spirit:field>
          <spirit:name>FULL</spirit:name>
          <spirit:description>FIFO full flag</spirit:description>
          <spirit:bitOffset>1</spirit:bitOffset>
          <spirit:bitWidth>1</spirit:bitwidth>
          <spirit:access>read-only</spirit:access>
    </spirit:field>
</spirit:register>
```

Test and Verification Solutions