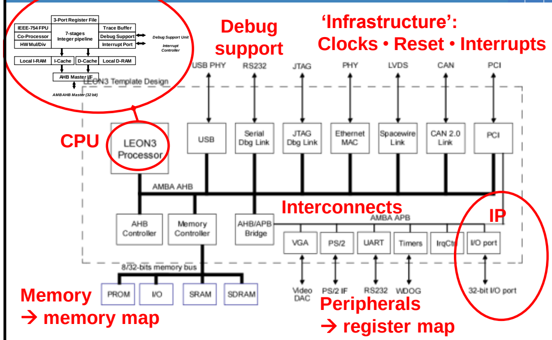
SoC Verification

Mike Benjamin
Associate at TVS

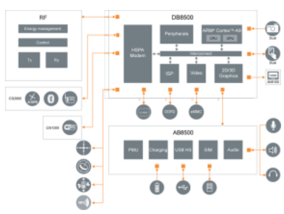
Test and Verification Solutions

What does a simple SoC look like?



Test and Verification Solutions

What does SoC level verification involve?



- **Top level**
  Verify the SoC as a stand alone device
- **System Level**
  Putting the complete design in a wider context …
  - ***System architecture***
  - ***Partner IP***
  - ***Software***

Test and Verification Solutions

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

Test and Verification Solutions

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**

Test and Verification Solutions

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)

Test and Verification Solutions

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(0x80000200);
    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

Coverage

Why add coverage?

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

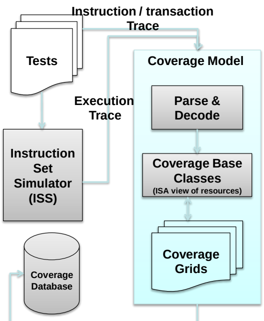
Test and Verification Solutions

Coverage Metrics

- Code Coverage
  - Toggle coverage on interconnects and top level signals
  - Have all the blocks been wired up correctly?
- Functional Coverage
  - Behaviour that can only be fully tested at top level
  - For example: power management
- Test case coverage
  - Quality of architectural, conformance or soak testing
  - Measure by parsing tests or looking at simulation trace

Test and Verification Solutions

Adding Coverage by Parsing the Tests

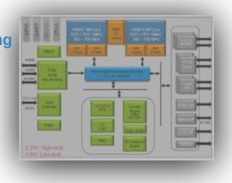


Test and Verification Solutions

Methodology for top level testing

TVS

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**



How to further increase the 'stress'

TVS

- **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)

Being pro-active to improve verification

TVS

- **Achieving the best possible test plan** — Verification Requirements Specification
  – 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

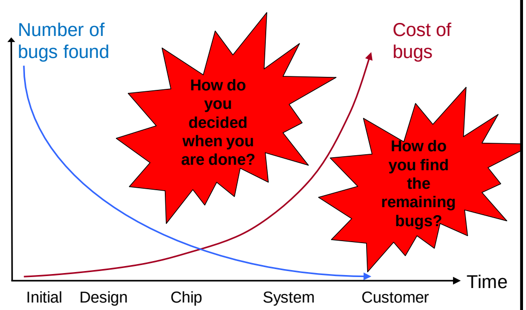
Communicate!

(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**

Cost of bugs over time (revisited)

TVS



Is block and top level verification sufficient?

TVS

- **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?

TVS

- **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

Test and Verification Solutions

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

Test and Verification Solutions

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

Test and Verification Solutions