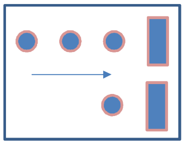
Functional Validation

Simulation/Emulation

Lecture summary

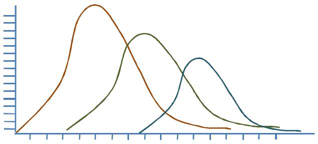
- Background/Environment/Landscape/Setting the stage
  - Simulation Objectives
  - Controllability and Observability
  - Functional Verification within Design Cycle
- Testbench structure
- Verification key concepts
  - Stimulus
  - Checkers
  - Coverage

10,000 foot view



Simulation (performance analysis perspective)

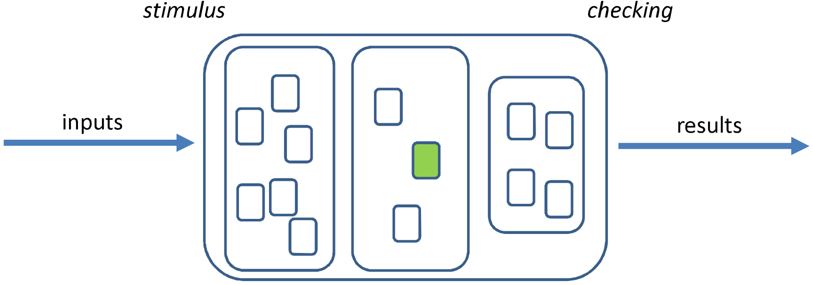
- Objective: to gain insights about a system through “experiments”
- Need a goal! What is being studied?
  - Abstract modeling
    - Some components so complex → approximate behavior by assuming it is random.
      - Replace components with simplified versions that follow statistical laws.
    - Ignore details that have no bearing on what we are trying to measure.
- Example: fast food restaurant
  - Goal: would more cashiers or cooks increase profits?
  - Variables (stimulus): When do customers arrive? What do they order? How long are they willing to wait? How long does it take for an order to be processed?
    - Use statistical distribution to model behavior. E.g. interarrival time
  - Measure: queue lengths, average and maximum waiting times, idle time of staff



Simulation (for Design Verification)

- <u>Functional Correctness Objective</u>: provide evidence that design behaves as specified and find examples of where it does not.
- Goal:
  - Fast food restaurant: all customers serviced and receive their order
  - Design: operations complete, produce desired result, on time, no deadlock, …
- Strategy:
  - All possible scenarios need to be considered
    - don’t want probability of correct behavior
  - How know when a failure occurs?
- Abstract modeling of system?
  - Often limited for cost considerations---just use RTL

Controllability and Observability

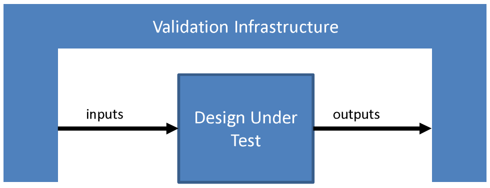


- Adder example:
  - At block level: specify specific inputs, view block outputs
  - At system level: input program must be fetched from memory, decoded, scheduled, result written back to memory, and then view result
- Easier to *wiggle* controls and observe results at implementation boundary
- Lower design level verification

  + Design ready sooner -> more bugs found at this stage

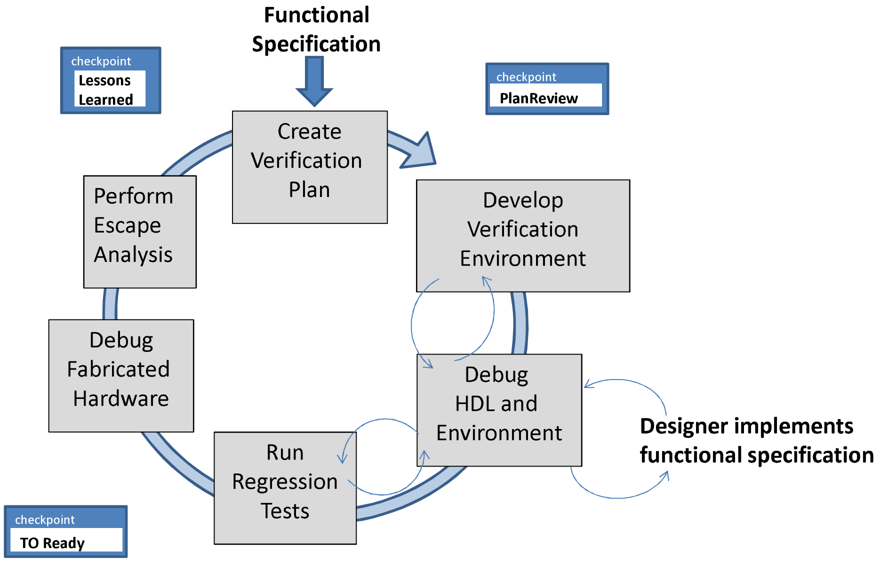
  - Design more likely to change -> requiring test changes

Verification Scope



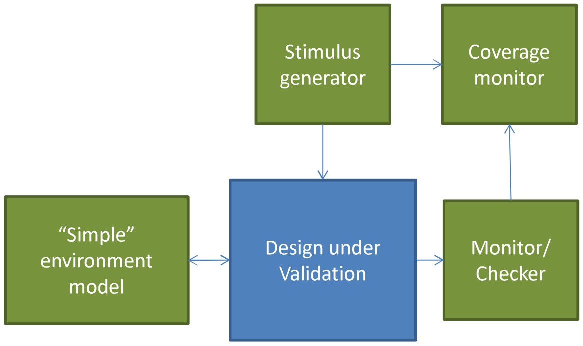
- At what level of design is verification done?
  - unit/cluster/full/system
- How much do we know about the internals ?
  - Do we treat the entity being tested (DUT/DUV) as an unknown
  - blackbox/whitebox/greybox verification
- How does a flaw manifest itself?
  - fault->error->failure
  - How can you detect/debug?
- Quality/Schedule/Cost
  - how long does it take to provide feedback and debug failures?

Functional Verification Cycle



From **Comprehensive Functional Verification**, Wile, Goss, & Roesner
Textbook for ECE510 Fundamentals of PreSi Validation

Basic Test Bench Structure



- Test Bench coded to create, observe, and check a pre-determined input sequence to the design.
- Closed system -- includes environment that responds to all potential DUV requests

All validation techniques need to consider stimulus, checking, and coverage.

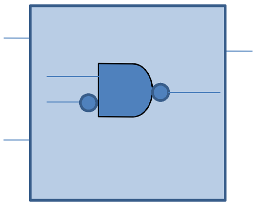
Stimulus

- Need to think about stimulus as sequences over time

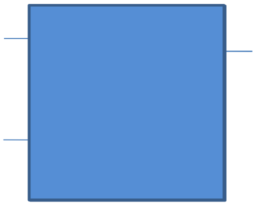
Specification

input		IMP
x	y	out
0	0	1
0	1	1
1	0	0
1	1	1

Implementation



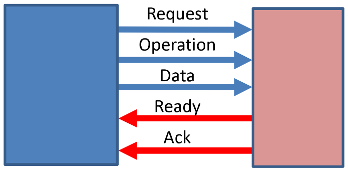
Implementation 2

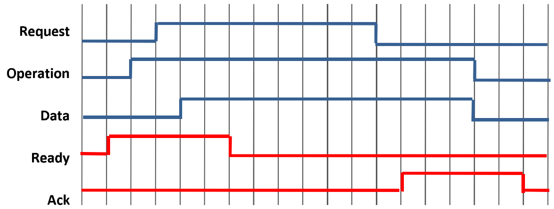


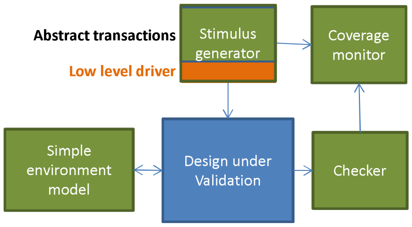
input		IMP	
x	y	out t=1	out t=2
0	0	1	1
0	1	1	1
1	0	0	0
1	1	1	0

Stimulus

- Abstraction essential
  - Consider the tedious effort required for someone to write a collection of test stimulus patterns



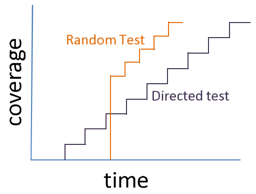




- Reuse important
  - Think abstractly and with an object oriented perspective
  - Macros, templates, functions, generating functions, transactions

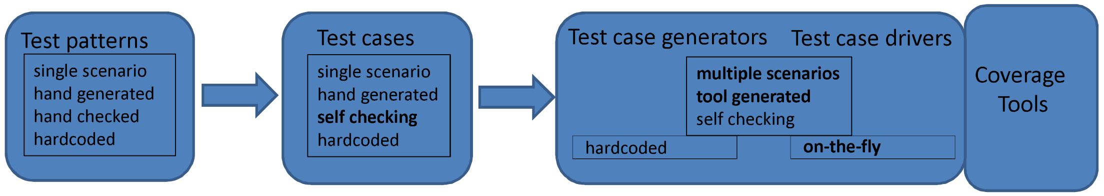
Stimulus

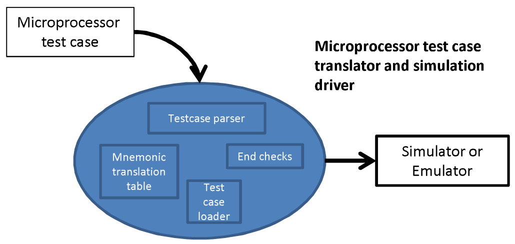
- Directed Tests
  - Define stimulus sequence (valid or invalid!) and possibly expected results (self checking directed test)
  - Need to write many tests
    - Hitting obscure conditions can be hard to construct
    - As features are added, complexity rises exponentially and can’t manually add all new directed tests.
- Test Generators
  - Constrained Random Tests (really pseudo-random)
    - Define the range of valid stimulus for each initiating/responding inputs (or sequence/timing of inputs)
      - Including initial state
    - Generator can create outlandish sequences leading to interesting (and confusing) failures
  - Directed Random Tests
    - Bias provided to the test generator to restrict the randomness
      - Distribution per variable, relationship between variables
      - temporary disabling, over-approximating for robustness
    - Focus attention on critical areas



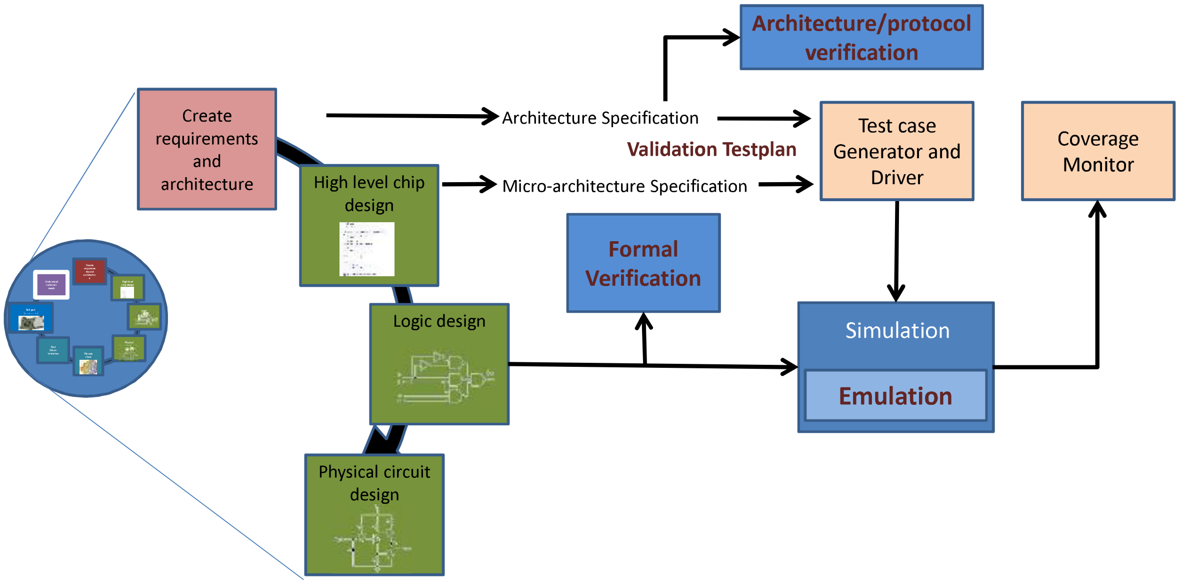
Foreshadowing: Formal Verification automatically considers every possible input sequence

Evolution of Functional Verification





Evolution of Functional Verification


Architecture/protocol verification
Create requirements and architecture
Architecture Specification
Validation Testplan
Test case Generator and Driver
Coverage Monitor
High level chip design
Micro-architecture Specification
Formal Verification
Logic design
Simulation
Emulation
Physical circuit design

Stimulus

Additional considerations

- Test generators an investment for multiple generations---build in quality!
- Use stimulus generators at multiple validation levels
  - User could provide some details, generator fill in the rest
- Constraint problems: false positives and false negatives

	Test Failure	Test Pass
Design Error	real failure	False positive
Design Correct	False negative	good result

- Product life cycle(s) and ROI (how much do you need and when?)
  - Regression
  - Design changes over time
  - Early design exercise (throw away?) vs robust validation (before tapeout?)

Monitors/Checkers

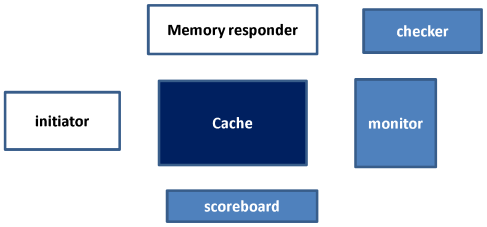
- Driving all possible stimulus won’t help without assessing the results/behavior…
- Goals: need observer to
  - Guard the ultimate quality of the product
  - Detect architecture/microarchitecture problems
  - Accelerate bug detection and root-cause discovery

- Guards can be
  - Self-checking test (tied to specific inputs)
  - Part of the RTL (embedded assertions)
  - A separate module or modules

- What to check
  - Outputs and state (reference model)
    - Visible state only or also internal state (more expensive)
  - Abstract behavior specifications

Monitors/Checkers

- Monitor/Checker/Scoreboard independent components
  - Monitor: self-contained component that observes:
    - Outputs for protocol adherence
    - Inputs for functional coverage and scoreboard updates
    - Internals for events of interest
  - Checker:
    - Special monitor that validates design is functionally correct
  - Scoreboard
    - Temporary holding location for checker (reference model tracker)



Coverage

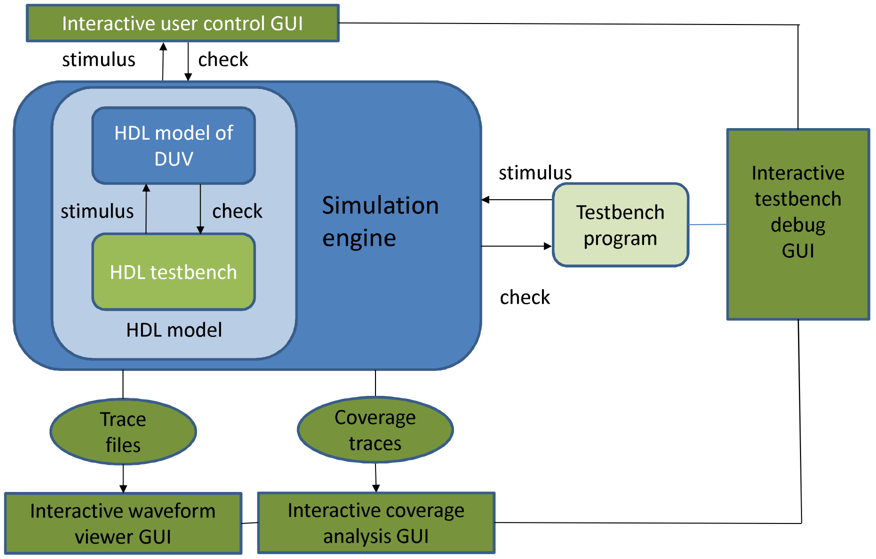
- Constrained random stimulus & checker infrastructure enables discovering unexpected bugs
  - Requires utilization of massive compute farms
  - Potential execution space effectively infinite
  - Can’t run enough test vectors to exhaustively check the whole design or even a significant fraction of it!
- Are we done yet?
  - Define a good set of targets
  - If haven’t hit everything, probably not done
  - Ultimately “done” assessment requires good engineering judgment

Coverage

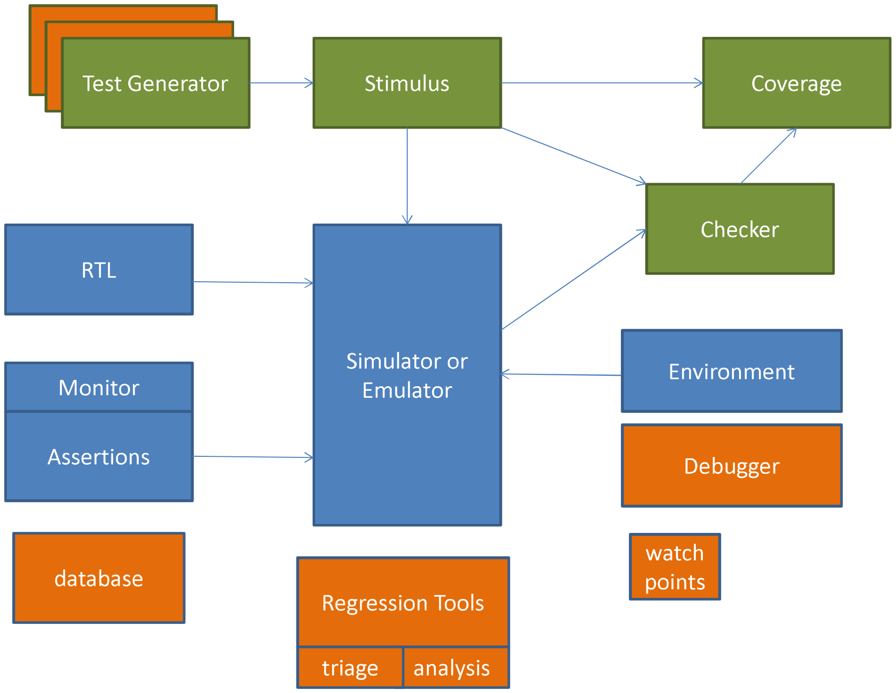
Additional considerations

- Has stimulus hit targeted conditions?
  - On which model and when?
    - Breath and Depth measures
    - What bugs escaped lower level model validation
  - Provides feedback to stimulus engines
- Establish connection back to validation plan
  - Is it OK to miss the identified holes?
- Creating a coverage monitor and subsequent analysis is a significant effort
  - Think OO, reusable, scalable

Simulation tool overview



Inspired from **Verilog HDL: A Guide to Digital Design and Synthesis,** Samir Palnikar


Test Generator
Stimulus
Coverage
Checker
RTL
Simulator or Emulator
Monitor
Assertions
Environment
Debugger
database
Regression Tools
triage
analysis
watch points