COMS31700 Design Verification:

Verification Tools

Directed Testing with Manual Checking

Kerstin Eder

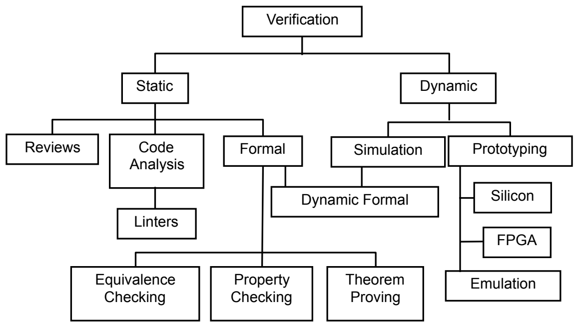
Design Automation and Verification

(Acknowledgement: Avi Ziv from the IBM Research Labs in Haifa has kindly permitted the re-use of some of his slides.)

University of BRISTOL

Department of COMPUTER SCIENCE

Functional Verification Approaches



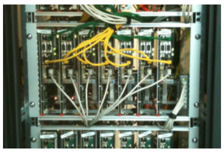
Achieving Automation

Task of Verification Engineer:

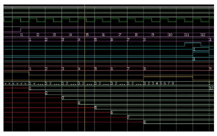
- Ensure product does not contain bugs - as fast and as cost-effective as possible.

(... and of Verification Team Leader):
  - Select/Provide appropriate tools.
  - Select a verification team.
  - Decide when cost of finding next bug violates **law of diminishing returns.**
- Parallelism, Abstraction and **Automation** can reduce the duration of verification. (Automation is currently the least applicable!)
- Automation reduces human factor, improves efficiency and reliability.
- **Verification TOOLS** are used to achieve automation.
  - Tool providers: Electronic Design Automation (EDA) industry

Tools used for Verification



- Dynamic Verification:
  - Hardware Verification Languages (HVL)
  - Testbench automation
  - Test generators
  - Coverage collection and analysis
  - General purpose HDL Simulators
    - Event-driven simulation
    - Cycle-based simulation (improved performance)
    - Waveform viewers (for debug)
  - Hardware accelerators/emulators, FPGAs
- Static Analysis / Verification Methods (Formal Methods):
  - Linting Tools
  - Equivalence checkers
  - Model checkers
    - Property Specification Languages (ABV)
  - Theorem provers
- Administration:
  - Version Control and Issue Tracking
  - Metrics
  - Data Management and Data Mining related to Metrics
- Third Party Models



Linting Tools

- Linters are **static** checkers.
- Assist in finding common coding mistakes
  - Linters exist for software and also for hardware.
    - `gcc -Wall` (When do you use this?)
- Only identify certain classes of problems
  - Many false negatives are reported.
  - Use a **filter** program to reduce false negatives.
    - Careful - don't filter true negatives though!
- Does assist in enforcing **coding guidelines!**
- Rules for coding guidelines can be added to linter.

Simulation-Based Verification

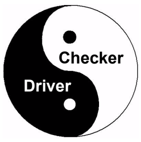
Directed testing with manual checking

Fundamentals of Simulation-based Verification

- Verification can be divided into two separate tasks
  1. Driving the design - Controllability
  2. Checking its behavior - Observability
- Basic questions a verification engineer must ask
  1. Am I driving *all possible* input scenarios?
  2. How will I know when a failure has occurred?

How do I know when I'm done?

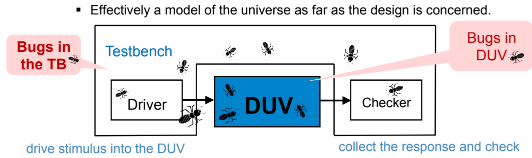
- Driving and checking are the yin and yang of verification
  - We cannot find bugs without creating the failing conditions
    - Drivers
  - We cannot find bugs without detecting the incorrect behavior
    - Checkers



What is a Testbench?

*"**Code** used to create a predetermined **input sequence** to a design, and to then observe the **response**."*

- Generic term used differently across the industry.
- Always refers to a test case/scenario.
- Traditionally, a testbench refers to code **written in a Hardware Description Language (VHDL, Verilog) at the top level of the design hierarchy**.
- **A testbench is a "completely closed" system:**
  - No inputs or outputs.
  - Effectively a model of the universe as far as the design is concerned.



Simulation-based Design Verification

- Simulate the design (not the implementation) **before** fabrication.
- **Simulating the design relies on simplifications:**
  - Functional correctness/accuracy can be a problem.

Verification Challenge: *"What input patterns to supply to the Design Under Verification (DUV) ..."*

- Simulation requires **stimulus.** It is dynamic, not just static!
- Requires to reproduce environment in which design will be used.
  - **Testbench** (Remember: Verification vs Testing!)

Verification Challenge: *"... and knowing what is expected for the output for a properly working design."*

- **Simulation outputs are checked externally** against design intent (specification)
  - Errors cannot be proven not to exist!

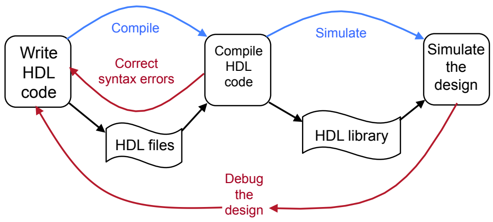
Two types of simulators: **event-based** and **cycle-based**

General HDL Simulators

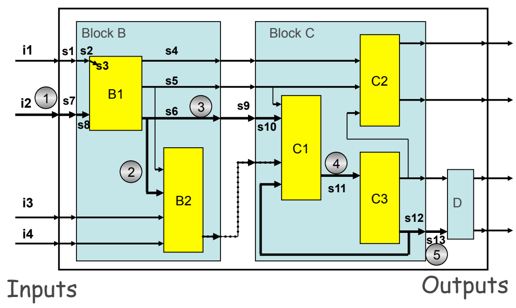
- Most Popular Simulators in Industry
  - **Mentor Graphics ModelSim/Questa**
  - **Cadence NCSim**
  - Synopsys VCS
- Support for full language coverage
  - "EVENT DRIVEN" algorithms
- VHDL's execution model is defined in detail in the IEEE LRM (Language Reference Manual)
- Verilog's execution model is defined by Cadence's Verilog-XL simulator ("reference implementation")

Simulation based on Compiled Code

- To simulate with **ModelSim:**
  - Compile HDL source code into a library.
  - Compiled design can be simulated.



Event Flow Example



Event-based Simulators

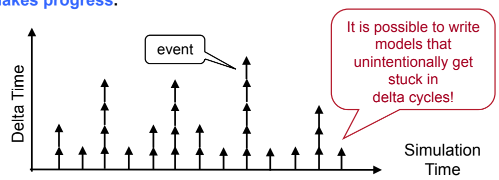
Event-based simulators are driven based on **events.** ☺

- Outputs are a function of inputs:
  - The outputs change only when the inputs do.
  - **The event is the input changing.**
  - An event causes the simulator to re-evaluate and calculate new output.
- Outputs (of one block) may be used as inputs (of another) ...
- **Re-evaluation happens until no more changes propagate through the design.**
- Zero delay cycles are called **delta cycles!**

13

Delta Cycles

- **Event propagation** may cause new values being assigned after **zero** delays.
  - (Remember, this is only a **model** of the physical circuit.)
- Although **simulation time** does **not advance**, the **simulation makes progress.**



- NOTE: Simulation progress is first along the zero/delta-time axis and then along the simulation time axis.

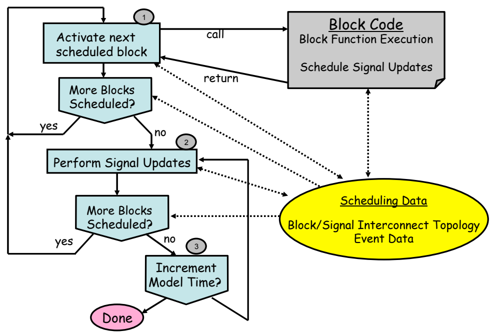
14

Event Driven Principles

- The event simulator maintains many lists:
  - A list of all **atomic** executable blocks
  - Fanout lists: A data structure that represents the interconnect of the blocks via signals
  - **A time queue** – points in time when events happen
  - **Event queues** – one queue pair for each entry in the time queue
    - Signal update queue
    - Computation queue
- The simulator needs to process all these queues at simulation time.

15

Core Algorithm of an Event-Driven Simulation Engine



16

Simulation Speed

What is holding us back?

Speedup strategies

Improving Simulation Speed

- The most obvious bottle-neck for functional verification is simulation throughput
- There are several ways to improve throughput
  - Parallelization
  - Compiler optimization techniques
  - Changing the level of abstraction
  - Methodology-based subsets of HDL
    - Cycle-based simulation
  - Special simulation machines

18

Parallelization

- Efficient parallel simulation algorithms are hard to develop
  - Much parallel event-driven simulation research
  - Has not yielded a breakthrough
  - Hard to compete against "trivial parallelization"
- Simple solution – run independent testcases on separate machines
  - Workstation "SimFarms"
  - 100s - 1000s of engineer's workstations run simulation in the background

Compiler Optimization Techniques

- Treat sequential code constructs like general programming language
- All optimizations for language compilers apply:
  - Data/control-flow analysis
  - Global optimizations
  - Local optimizations (loop unrolling, constant propagation)
  - Register allocation
  - Pipeline optimizations
  - etc.
- Global optimizations are limited because of model-build turn-around time requirements
  - Example: modern microprocessor is designed with ~1Million lines of HDL
    - Imagine the compile time for a C-program with ~1M lines!

Changing the Level of Abstraction

- Common theme:
  - Cut down the number of scheduled events
  - Create larger sections of un-interrupted sequential code
  - Use less fine-grained model structure
    - →Smaller number of schedulable blocks
  - Use higher-level operators
  - Use zero-delay wherever possible
- Data abstractions
  - Use binary over multi-value bit values
  - Use word-level operations over bit-level operations

Changing the Level of Abstraction

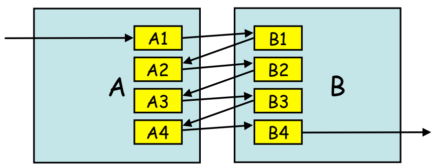
```
s(0) <= a(0) xor b(0);
c(0) <= a(0) and b(0);
s(1) <=  a(1) xor b(1) xor c(0);
c(1) <= (a(1) and b(1)) or (b(1) and c(0)) or (c(0) and a(1));
sum_out(1 to 0) <= s(1 to 0);
carry_out <= c(1);
```

```
s(2 to 0) <= ('0' & a (1 to 0)) +  ('0' & b(1 to 0) );
sum_out(1 to 0) <= s(1 to 0);
carry_out <= s(2);
```

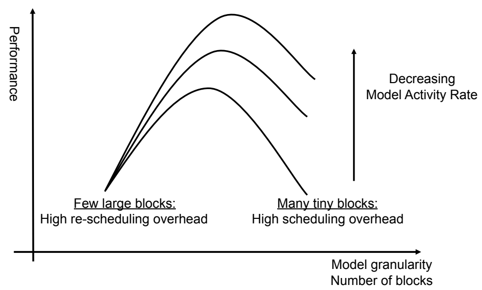
```
process (a, b)
begin
   s(2 to 0) <= ('0' & a (1 to 0)) +  ('0' & b(1 to 0) );
   sum_out(1 to 0) <= s(1 to 0);
   carry_out <= s(2);
end process
```


Changing the Level of Abstraction

- Scheduling the small blocks
  - {A1, B1, A2, B2, A3, B3, A4, B4}
  - Each small block is executed once
- Scheduling the big blocks
  - {A, B, A, B, A, B, A, B}
  - A = A1 and A2 and A3 and A4
  - Each small block is executed 4 times

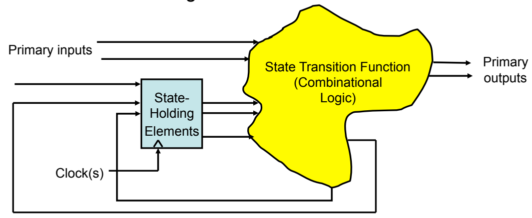


Changing the Level of Abstraction



Synchronous Design Methodology

- The design is comprised of
  - State-holding (storage) elements
  - Combinational logic for state transition function

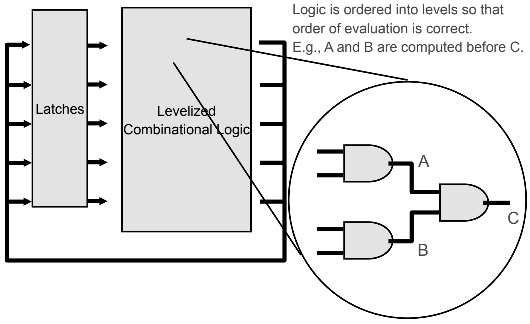


Cycle Based Model Build Flow

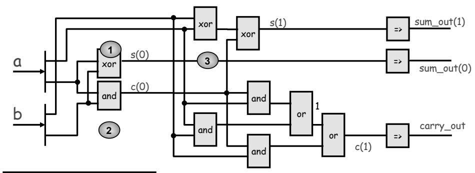
- Language compile – synthesis-like process
  - Simpler because of missing physical constraints
  - Logic mapped to a non-cyclic network of Boolean functions
  - Hierarchy is preserved
- Flatten hierarchy – crush design hierarchy to increase optimization potential
- Optimization – minimize the size of the model to increase simulation performance
- Levelize logic
- Translate to instructions

Model Build – Levelization

Logic is ordered into levels so that order of evaluation is correct.
E.g., A and B are computed before C.



Translate to Instructions




```
1  Load temp1, a(0)
   Load temp2, b(0)
   Xor temp1, temp2, temp3
   Store temp3, s(0)
2  And temp1, temp2, temp3
   Store temp3, c(0)
   Load temp1, a(1)
   Load temp2, b(1)
3  Xor temp1, temp2, temp3
   ...
```

We can convert every Boolean function into a minimal set (~4 or better) of instructions

Parallelism in Generated Code Cycle-Sim

- Word-level operations can be easily parallelized

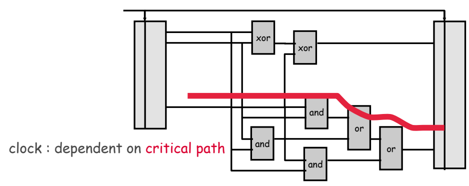
A(0 to 31) <= B(0 to 31) and C(0 to 31) and D(0 to 31)

Is translated into

```
LoadWord R1,  B(0 to 31)
LoadWord R2,  C(0 to 31)
AND R1, R1, R2
LoadWord R2,  D(0 to 31)
AND R1, R1, R2
StoreWord R1, A(0 to 31)
```


Synchronous Design Methodology

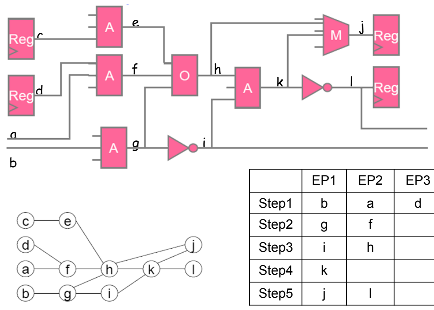
- Clock the design only so fast as the longest possible combinational delay path settles before cycle is over
- Cycle time depends on the longest topological path
  - Hazards/Races do not disturb function
  - Longest topological path can be analytically calculated w/o using simulation -> stronger result w/o sim patterns



Hardware Acceleration

- Programs created for cycle simulation are very simple
  – Small set of instructions
  – Simple control – no branches, loops, functions
- Operations at the same level can be executed in parallel
- Hardware acceleration uses these facts for fast simulation by utilizing
  – Very large number of small and simple special-purpose processors
  – Efficient communication and scheduling

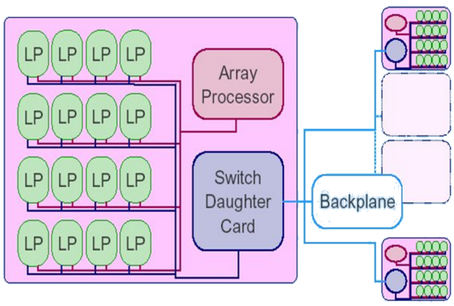
Scheduling Example



	EP1	EP2	EP3	EP4
Step1	b	a	d	c
Step2	g	f		e
Step3	i	h		
Step4	k			
Step5	j	l		

12 steps serial, 5 steps parallel

Accelerator Basic Structure



Principle of Operation

- Compiler transforms combinational logic into Boolean operations
- Compiler schedules interprocessor communications using a fast broadcast technique
- Emulation performance dictated by
  – Number of processors
  – Number of levels in the design

Simulation Speed Comparison

Event Simulator	1
Cycle Simulator	20
Event driven cycle Simulator	50
Acceleration	1000
Emulation	100000

Verification Languages

Raising the level of abstraction

Verification Languages

- Need to be designed to address **verification** principles.
- Deficiencies in RTL languages (HDLs such as Verilog and VHDL):
  - **Verilog** was designed with focus on describing low-level hardware structures.
    - No support for **data structures** (records, linked lists, etc).
    - Not object/aspect-oriented.
      - Useful when several team members develop testbenches.
  - VHDL was designed for large design teams.
- Limitations inhibit **efficient** implementation of verification strategy.
- High-level verification languages are (currently):
  - **System Verilog**
    - IEEE 1800 [2005] Standard for System Verilog- Unified Hardware Design, Specification, and Verification Language
  - e-language used for Cadence's Specman Elite [IEEE P1647]
  - (Synopsys' Vera, System C)

Features of High-Level Verification Languages

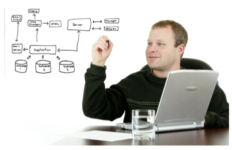
- Raising the level of abstraction:
  - From bits/vectors to high-level data types/structures
    - lists, structs, scoreboards including ready made functions to access these
- Support for building the verification environment
  - Enable testbench automation
  - Modularity
    - Object/aspect oriented languages
    - Libraries (VIP) to enable re-use
- Support for test generation
  - Constrained random test generation features
    - Control over randomization to achieve the target values
    - Advanced: Connection to DUV to generate stimulus depending on DUV state
- Support for coverage
  - Language constructs to implement functional coverage models

Any other *verification* Languages?

Tommy Kelly, CEO of Verilab:

"Above all else, the Ideal Verification Engineer will know how to construct software."

- Toolkit contains not only Verilog, VHDL, SystemVerilog and e, but also Python, Lisp, mySQL, Java, ... ☺



Directed Testing

Focus on checking

The Importance of Driving and **Checking**

- Drivers activate the bug.
- The observable effects of the bug then need to propagate to a checker.
- A checker needs to be in place to detect the incorrect behaviour.

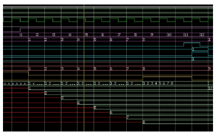
All three are needed to find bugs!

Checking: How to predict expected results

- Methods for checking:
  - **Directed testing:**
    - Because we know what will be driven, a checker can be developed for each test case individually.
- Sources for checking:
  - Understanding of the inputs, outputs and the transfer function of the DUV.
  - Understanding of the design context.
  - Understanding of the internal structures and algorithms (uarch).
  - Understanding of the top-level design description (arch).
  - **Understanding of the specification.**
- **Beware:**
  - Often, all outputs of the design must be checked at every clock cycle!
    - However, if the outputs are not specified clock-cycle for clock-cycle, then verification should not be done clock-cycle for clock cycle!
  - Response verification should not enforce, expect, nor rely on an output being produced at a specific clock cycle.

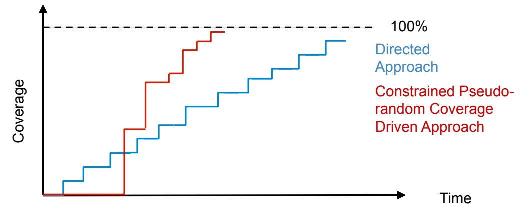
Limitations of Using Waveform Viewers as Checkers

- Often come as part of a simulator.
- **Most common verification tools used...**
  - Used to **visually inspect** design/testbench/verification environment.
  - Recording waves decreases performance of simulator. (Why?)
- **Don't use viewer to determine if DUV passes/fails a test.**
  - **Why not?**
- **Can use waveform viewer for debugging.**
  - Consider costs and alternatives.
  - Benefits of automation.
  - Need to increase productivity.



43

Limitations of Directed Testing: Coverage



Criteria:
- Effectiveness
- Efficiency
- Maintainability
- Re-usability

Directed testing has many shortfalls wrt these criteria.

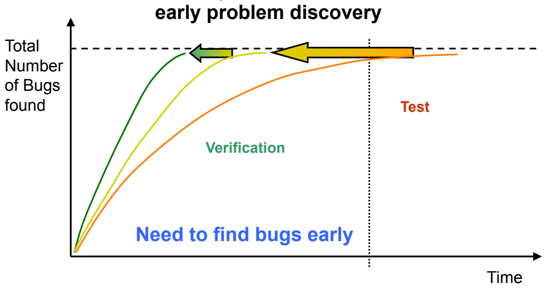
Why would one use Directed Testing?

Need to increase productivity!

44

Impact of Increasing Verification Productivity

Productivity improvements drive early problem discovery



45

Verification Tools

Third Party Models

Metrics

Third Party Models

- Chip needs to be verified in its **target environment.**
  - Board/SoC Verification
- Do you develop or purchase behavioural models (specs) for board parts?
  - Buying them may seem expensive!
  - Ask yourself:
    "If it was not worth designing on your own to begin with, why is writing your own model now justified?"
  - The model you develop is not as reliable as the one you buy.
  - The one you buy is used by many others - not just yourself.
- Remember: In practice, it is often more expensive to develop your own model to the **same degree of confidence** than licensing one.

47

Metrics

- Not really verification tools - but managers love metrics and measurements!
  - Managers often have little time to personally assess progress.
  - They want something measurable.
- **Coverage** is one metric - will be introduced later.
- **Others metrics include:**
  - Number of lines of code
  - Ratio of lines of code (between design and verifier)
  - Drop of source code changes
  - Number of outstanding issues

48

Summary

We have covered:

- Verification Tools & Languages
- Basic testbench components
- Writing directed tests
- The importance of Driving and Checking
- Checking when we use directed testing
- Limitations of directed testing
- Cost of debug using waveforms

49