COMSM0115 Design Verification:

Tools for Simulation-based Verification

Kerstin Eder

(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

Last Time

- Verification hierarchy
  - Levels of verification
- Fundamentals of Simulation-based Verification:
  - Strategy
    - Driving principles
    - Checking strategies

Achieving Automation

Task of Verification Engineer:

- Ensure product is not a Type II mistake (false positive) - 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

Functional Verification Tools

Tools used for simulation-based verification:

- Linting Tools
- **General purpose HDL Simulators**
  - Event-driven simulation
- **Improving simulation performance**
  - Raising the abstraction level
  - Cycle-based simulation
    - Synchronous design methodology
  - Hardware accelerators
- **Waveform Viewers**
- Third Party Models
- For programming: Verification Languages (Non-HDL!)
- Version Control(!)
- Issue Tracking(!)
- Metrics
- Coverage (later in more detail)

Linting Tools

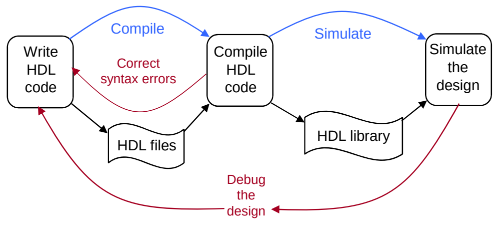
- Linters are **static** checkers.
- Assist in finding common coding mistakes
  - Linters exist for software and also for hardware.
    - `gcc -Wall` (Did you know this existed?)
- Only identify certain classes of problems
  - Many false negatives are reported.
  - Use a **filter** program to reduce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

General HDL Simulators

- Most Popular Simulators in Industry
  - **Mentor Graphics ModelSim - MTI VSIM**
  - **Cadence NCSim - Verilog XL**
  - 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.



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) ..."*

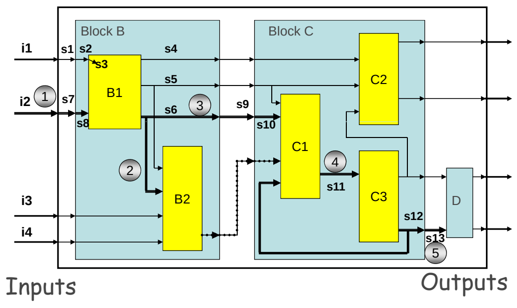
- Simulation requires **stimulus.** It is dynamic, not just static!
- Requires to reproduce environment in which design will be used.
  - **Testbench** (Remember: Verification vs Testing!)

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

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

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

Event Flow Example



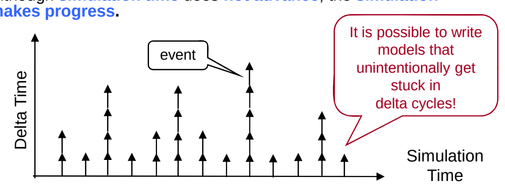
Event-based Simulators

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!**

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**.

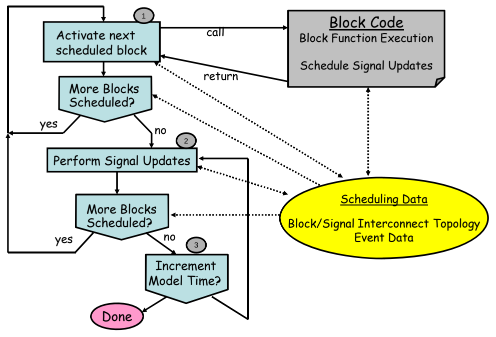


- NOTE: Simulation progress is first along the zero/delta-time axis and then along the simulation time axis.

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.

Core Algorithm of an Event-Driven Simulation Engine



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

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 w/ ~1Million lines of HDL
    - Imagine the compile time for a C-program w/ 1M lines!

Changing the Level of Abstraction

- Common theme:
  - Cut down of number of scheduled events
  - Create larger sections of un-interrupted sequential code
  - Use less fine-grain granularity for 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

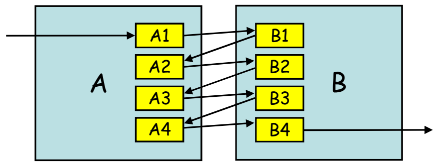
```
s(0) <= a(0) xor b(0);
c(0) <= a(0) and b(0);
s(1) <=  a(1) xor b(1) xor c(0);
c(1) <= (a(1) and b(1)) or (b(1) and c(0)) or (c(0) and a(1));
sum_out(1 to 0) <= s(1 to 0);
carry_out <= c(1);
```

```
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);
```

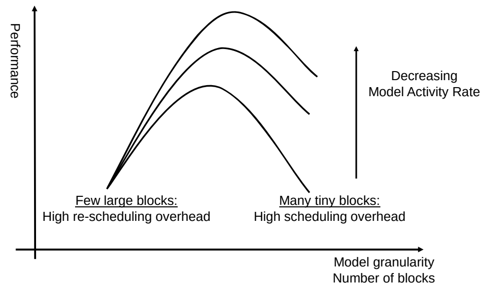
```
process (a, b)
begin
  s(2 to 0) <= ('0' & a (1 to 0)) + ('0' & b(1 to 0) );
  sum_out(1 to 0) <= s(1 to 0);
  carry_out <= s(2);
end process
```

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

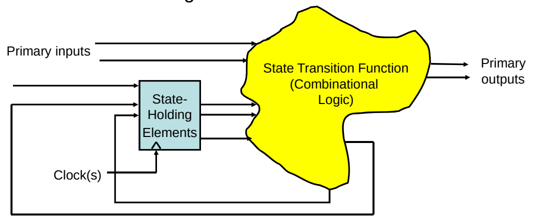


Changing the Level of Abstraction



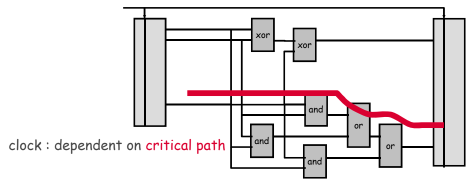
Synchronous Design Methodology

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



Synchronous Design Methodology

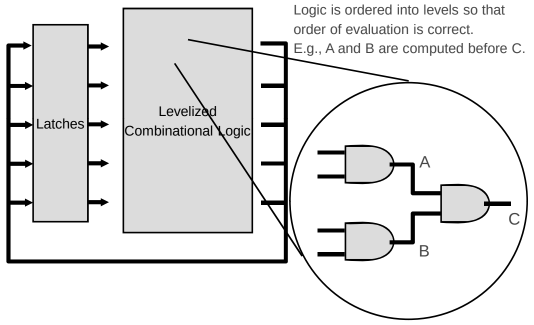
- Clock the design only so fast as the longest possible combinational delay path settles before cycle is over
- Cycle time depends on the **longest topological path**
  - Hazards/Races do not disturb function
  - Longest topological path can be analytically calculated w/o using simulation -> stronger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



Cycle Based Model Build Flow

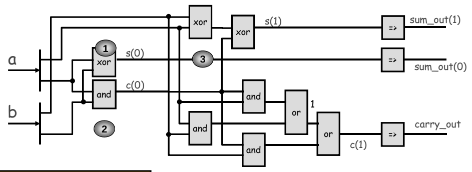
- Language compile – synthesis-like process
  - Logic mapped to a non-cyclic network of Boolean functions
  - Hierarchy is preserved
- Flatten hierarchy – crush design hierarchy to increase optimization potential
- Optimization – minimize the size of the model to increase simulation performance
- Levelize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



```
  Load temp1, a(0)
  Load temp2, b(0)
1 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 cover every Boolean function into a minimal set (~4 or better) instructions

Parallelism in Generated Code Cycle-Sim

- Word-level operations can be easily parallelized

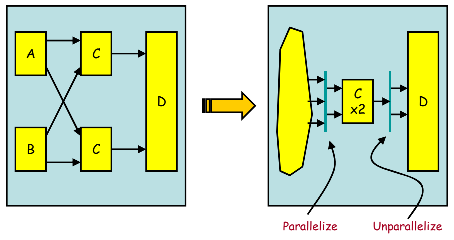
A(0 to 31) <= B(0 to 31) and C(0 to 31) and D(0 to 31)

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)
```

Parallelism in Generated Code Cycle-Sim

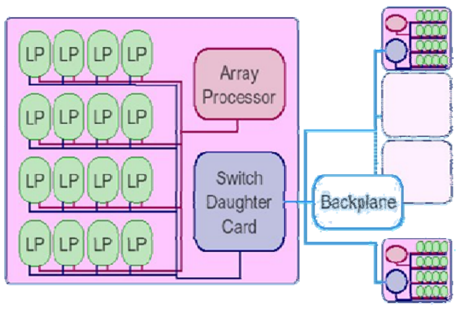
- Across similar blocks in the hierarchy



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

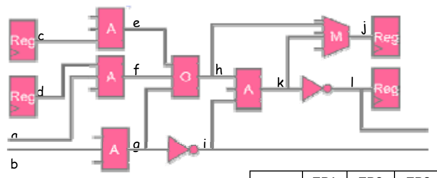
Accelerator Basic Structure

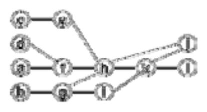


Accelerator: Principle of Operation

- 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

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

Simulation Speed Comparison

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

Why do Event-based Simulators still exist?

What is the cost of the speedup?

Co-Simulation

- Co-simulators are a combination of event, cycle and other tools, e.g. accelerators, emulators.
- **Performance is decreased** due to inter-tool communication overhead.
- **Ambiguities arise** during translation from one simulator to the other.
  - Verilog's 128 possible states to VHDL's 9!
  - Analog current and voltage into digital logic value and strength.

Waveform Viewers

- Often considered 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 design passes or fails a test.

- **Use waveform viewer for debugging.** (Assignment 1)
- Advanced waveform viewers can **compare** waveforms.
  - Problem is how to determine what is the **"golden wave".**
  - Most applicable to redesign, where design must maintain cycle by cycle compatibility. (Assumes previous design was correct!)
  - **Pitfall when doing comparisons:** Comparison fails when...
  - **Absolute value** of signals is not so important - **relative relationship between events** is (often) more important.

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.

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-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):
  - **e-language used for Specman Elite** [IEEE P1647]
  - Synopsys' Vera, System C, SystemVerilog

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, Vera and e, but also: Python, Lisp, mySQL, Java, ... ☺

Version Control Software

- **Concepts for version control:**
  - Files must be centrally managed.
  - It must be easy to get the files. (Simplify access.)
  - Files are owned by team - not by one individual!
  - (Advanced: Automatic notification of changes via e-mail.)
  - History is kept for each file.
  - Examples: Subversion (see http://subversion.tigris.org/) or CVS.

- HDL design/verification is similar to managing a large software project!
- Software Engineers can bring valuable skills to design/verification.

Why is version control considered a verification tool?
- To ensure that what is being verified is actually what is being implemented. Wouldn't you hate to spend a month verifying something that is out of date...?

Issue Tracking

- Another tool often not considered a verification tool.
  - Verification engineer's job is to **find bugs.** :)
  - Issue tracking is used to deal with the bugs that are found – **bugs must be fixed!**
- Two things to consider:
  - **What is an issue?**
  - **How to track it?**

What is an issue?

- The cost of tracking an issue should not be greater than the cost of the issue itself!
- An **issue** is anything that can effect the functionality of the design!
  - Bugs found during execution of testbench.
  - Ambiguities or incompleteness of a specification.
  - Architectural decisions/trade-offs.
  - Errors found at all stages of the design (in the design or the verification environment).
  - New but relevant stimuli (corner cases) that are not part of the **verification plan.**
- Basic principle: **When in doubt - track it!**
- Some bugs might not be worth fixing in the current product, but wewant to capture the issue so that they are fixed in next revision of chip.

How to track an issue?

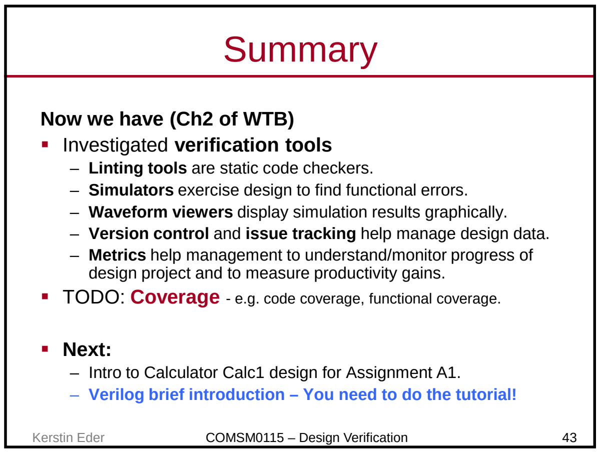
- Different methods - more or less successful...
  - "Grapevine", Post-It, Procedural, …
- **Computerised system:** e.g. track (see http://trac.edgewall.org/)
  - **Use a computer system (usually database) to track issues!**
  - Issues are seen through to resolution.
  - Issues can be assigned and/or reassigned to individuals or small teams.
  - Automatic e-mail notification:
    - of new issues, of status of top issues, etc.
  - **Contains a history.**
    - Provides **lessons-learned database** for others.

Some computerised issue tracking systems fail - why?

Biggest problem: **Ease of use!!!**
– It should not take longer to submit an issue than to fix it!

Metrics

- Not really a verification tool - 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.



Summary

Now we have (Ch2 of WTB)

- Investigated **verification tools**
  - **Linting tools** are static code checkers.
  - **Simulators** exercise design to find functional errors.
  - **Waveform viewers** display simulation results graphically.
  - **Version control** and **issue tracking** help manage design data.
  - **Metrics** help management to understand/monitor progress of design project and to measure productivity gains.
- TODO: **Coverage** - e.g. code coverage, functional coverage.

- **Next:**
  - Intro to Calculator Calc1 design for Assignment A1.
  - **Verilog brief introduction – You need to do the tutorial!**

Kerstin Eder COMSM0115 – Design Verification 43