COMS30026 Design Verification

# Verification Tools

Kerstin Eder

Trustworthy Systems Laboratory

University of
BRISTOL

# 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 and, thus, increase productivity.

- Automation reduces the human factor, improves efficiency and reliability.

**Verification TOOLS** are used to achieve **automation**.

- – Tool providers: Electronic Design Automation (EDA) industry

# Annex A
## (informative)

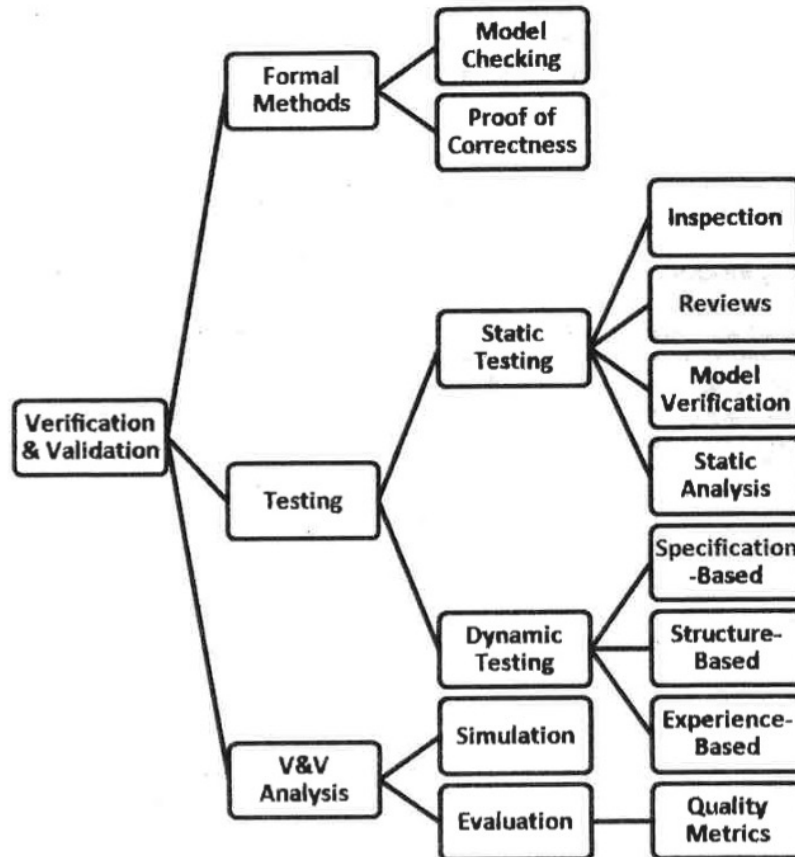# The Role of Testing in Verification and Validation



**Figure 11 — Hierarchy of Verification and Validation activities**

Figure 11 defines the complete nature of verification and validation (V&V) activities. V&V can be done on system, hardware, and software products. These activities and planning are defined and refined in IEEE 1012 and ISO/IEC 12207. Much of V&V is accomplished by testing. The ISO/IEC 29119 standard addresses the Dynamic and Static software testing (directly or via reference), thus covering parts of this verification and validation model. ISO/IEC 29119 is not intended to address all the elements of the V&V model, but it is important for a tester to understand where they fit within this model.

# 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**
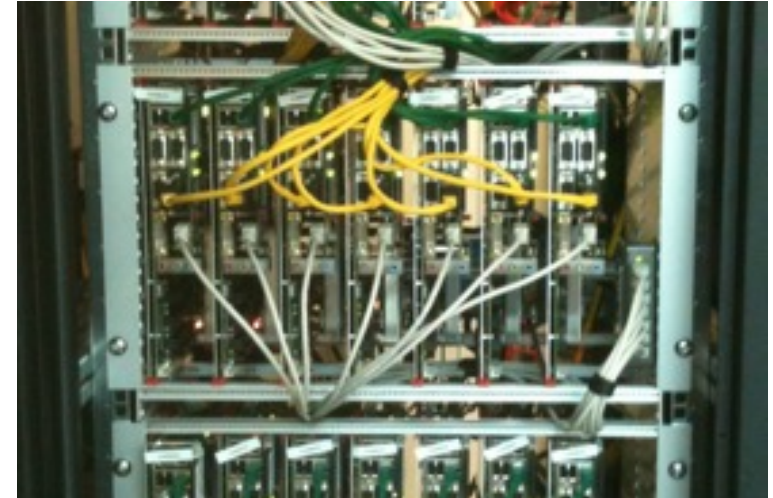
5
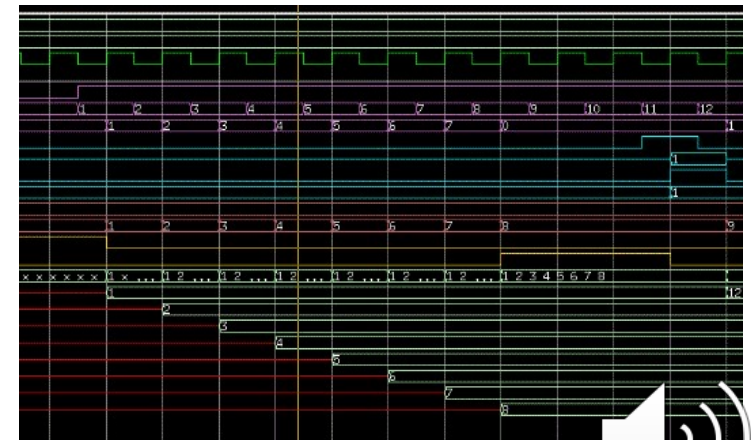
# Functional Verification Approaches
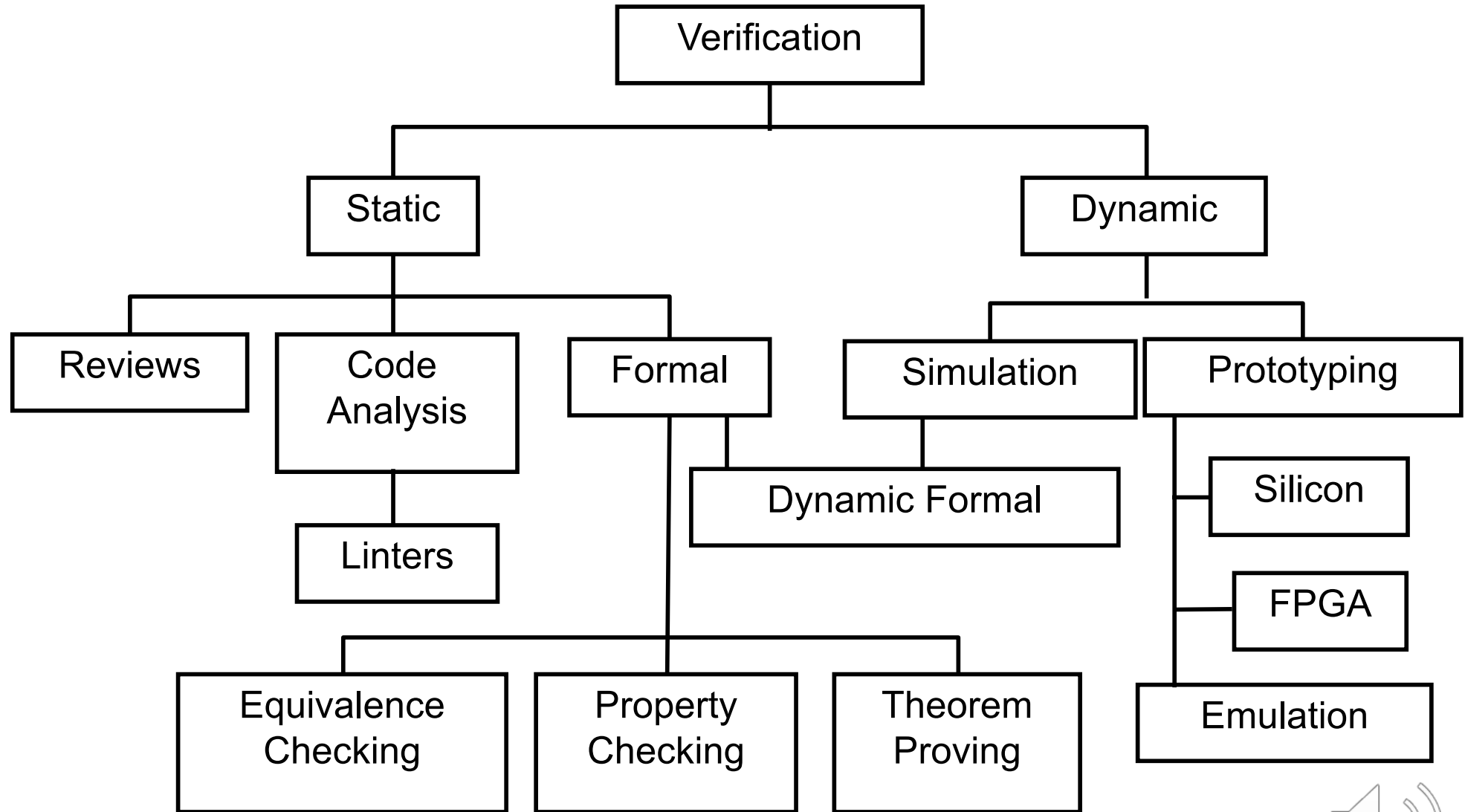
# Linting Tools

- Linters are **static** checkers.
- Assist in finding common coding mistakes in syntactically correct code
  - Linters exist for software and also for hardware design.
    `gcc –Wall / clang-tidy`     (When do you use these?)
- Linters only identify certain classes of problems
  - Many false positives (i.e. false alarms) are reported.
  - Use a **filter** program to reduce these.
    - Careful - don't filter true positives though!
- Linters can assist in enforcing **coding guidelines!**
  - Rules for coding guidelines can be added to linter.

*Is there any merit in using a linter alongside simulation-based testing?*

# Tools for Simulation-Based Verification

## Simulators

# 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

Checker

Driver
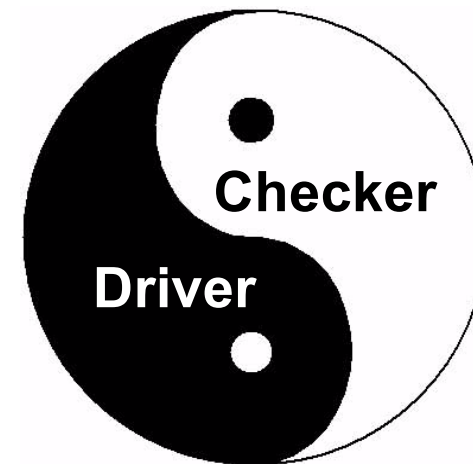
# What is a Testbench?

*"**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.
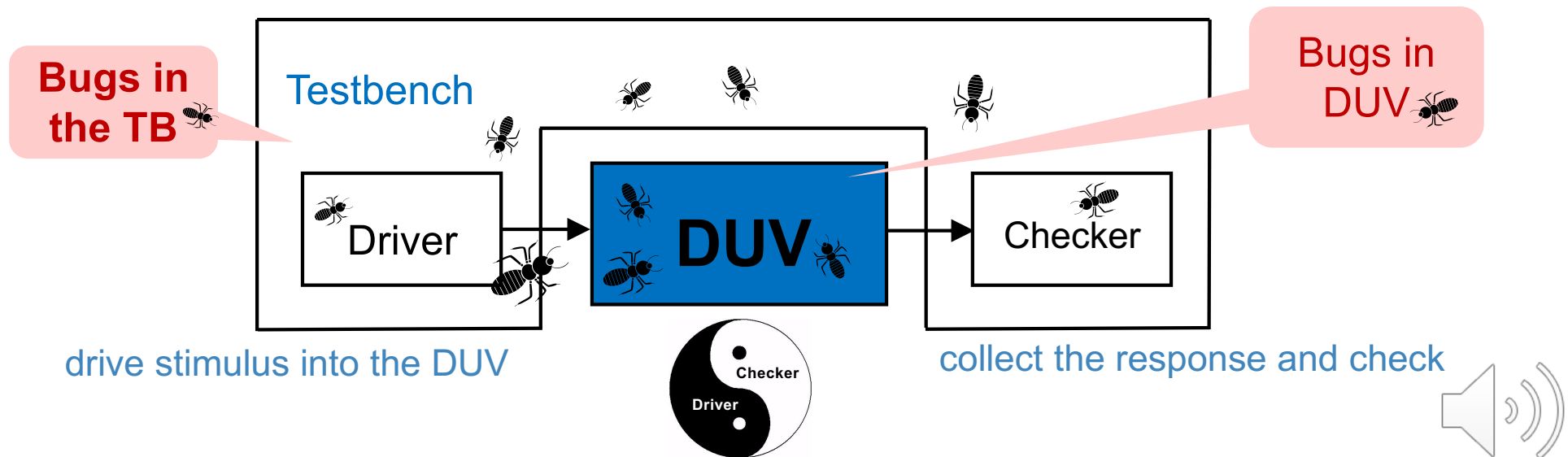


drive stimulus into the DUV

collect the response and check

# 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 the environment in which the DUV will be used.
  - *Test*bench (Remember: Verification vs Testing!)

# 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 the environment in which the DUV will be used.
  - *Test*bench (Remember: Verification vs Testing!)

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

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

    *"Testing shows the presence, not the absence of bugs."*

    [Edsger W. Dijkstra]
    see also the famous EWD manuscript
(e.g. *"The Humble Programmer"* https://www.cs.utexas.edu/users/EWD/ewd03xx/EWD340...F)

# General HDL Simulators

- **Most Popular Simulators in Industry**
  - **Siemens ModelSim/Questa**
  - **Cadence NCSim**
  - Synopsys VCS

- **Provide full language support**
  - "EVENT DRIVEN" algorithms

- **VHDL's execution model is defined in detail in the IEEE LRM (Language Reference Manual)**

- **Verilog's execution model was for a long time defined by Cadence's Verilog-XL simulator ("reference implementation");**
  **it is now an IEEE standard, e.g. Verilog 2005**

# Simulation based on Compiled Code

- To simulate with **ModelSim/Questa:**
  - Compile HDL source code into a library.
  - Compiled design can be simulated.
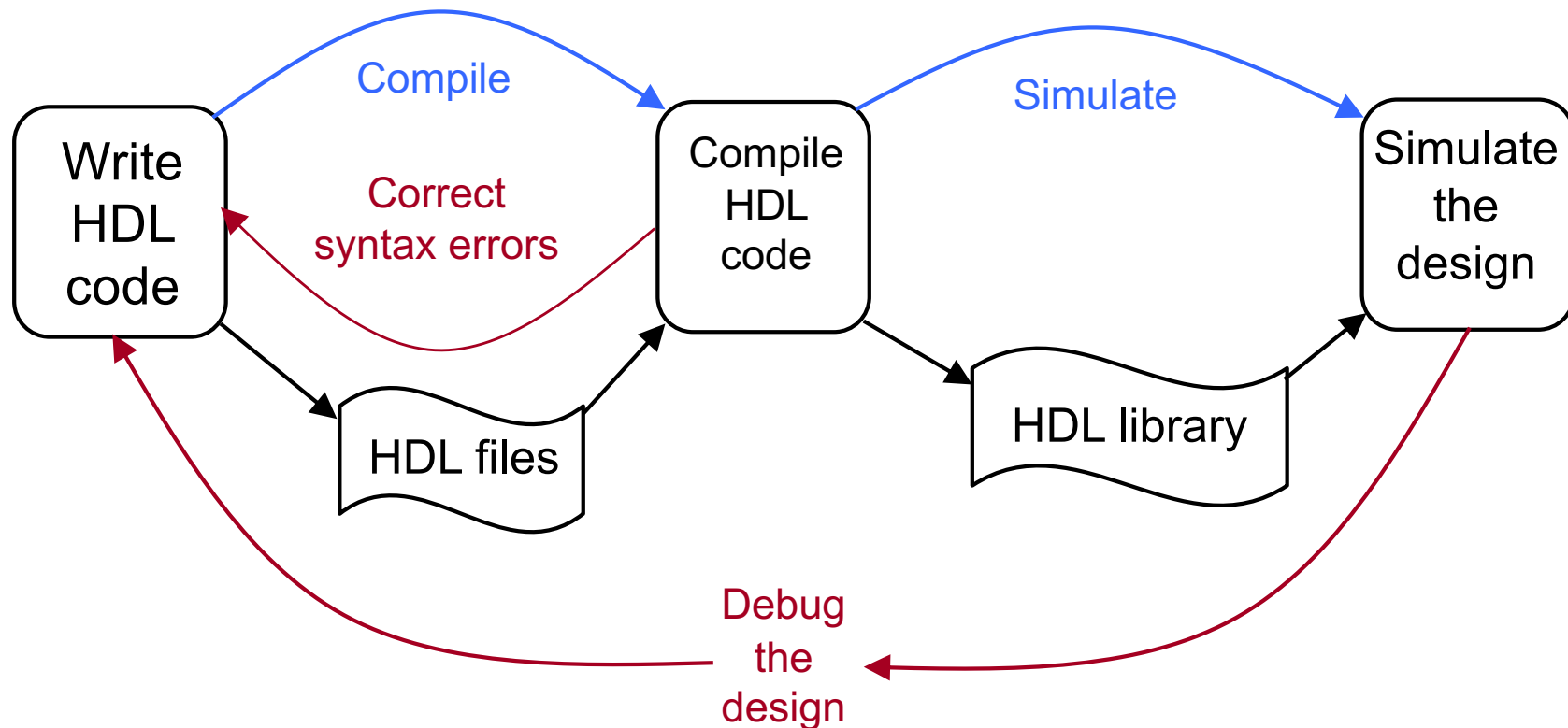
# Two types of simulators:

event-based and
cycle-based

# 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) ...

# Event Flow Example
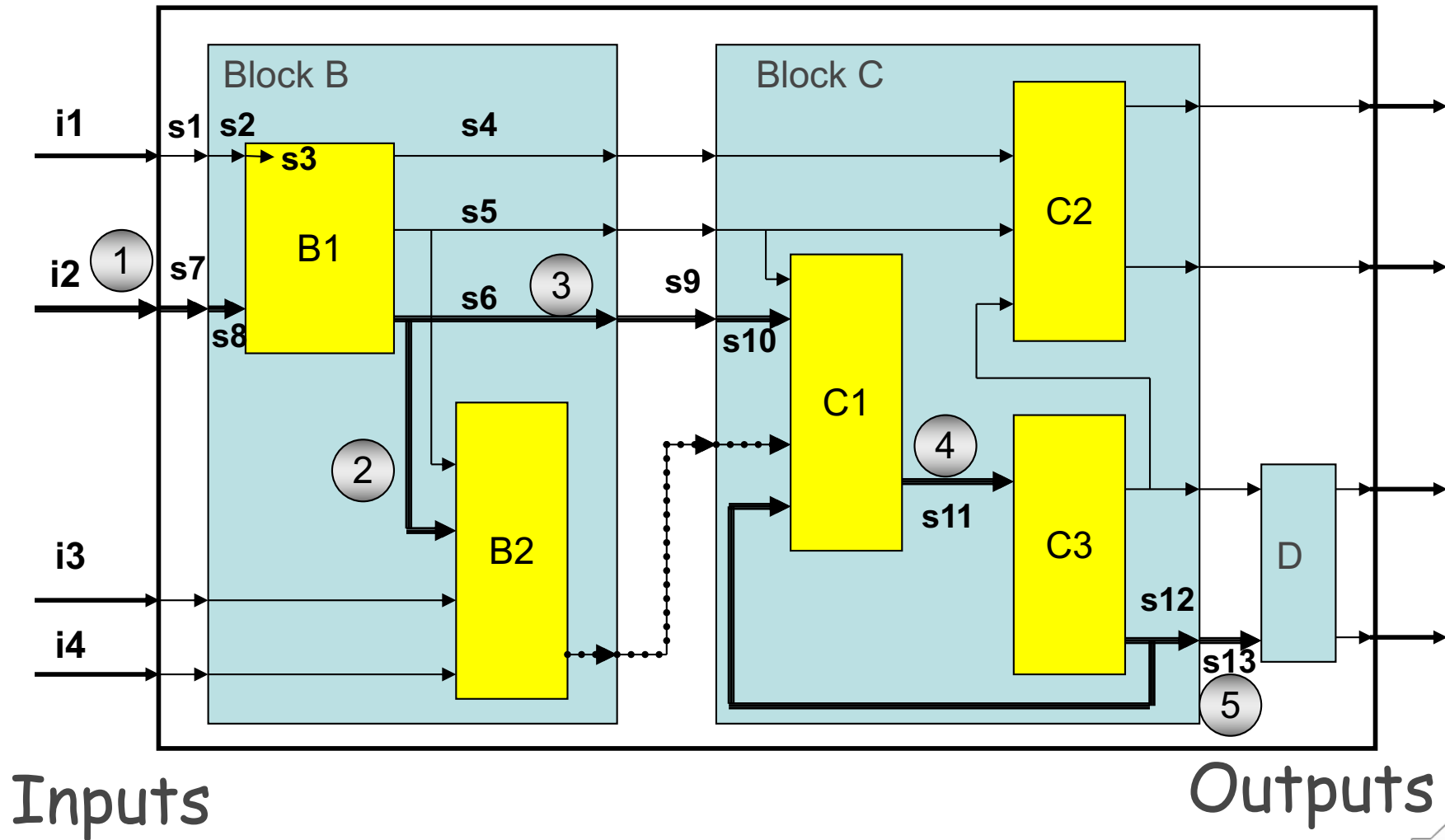
# Event-based Simulators

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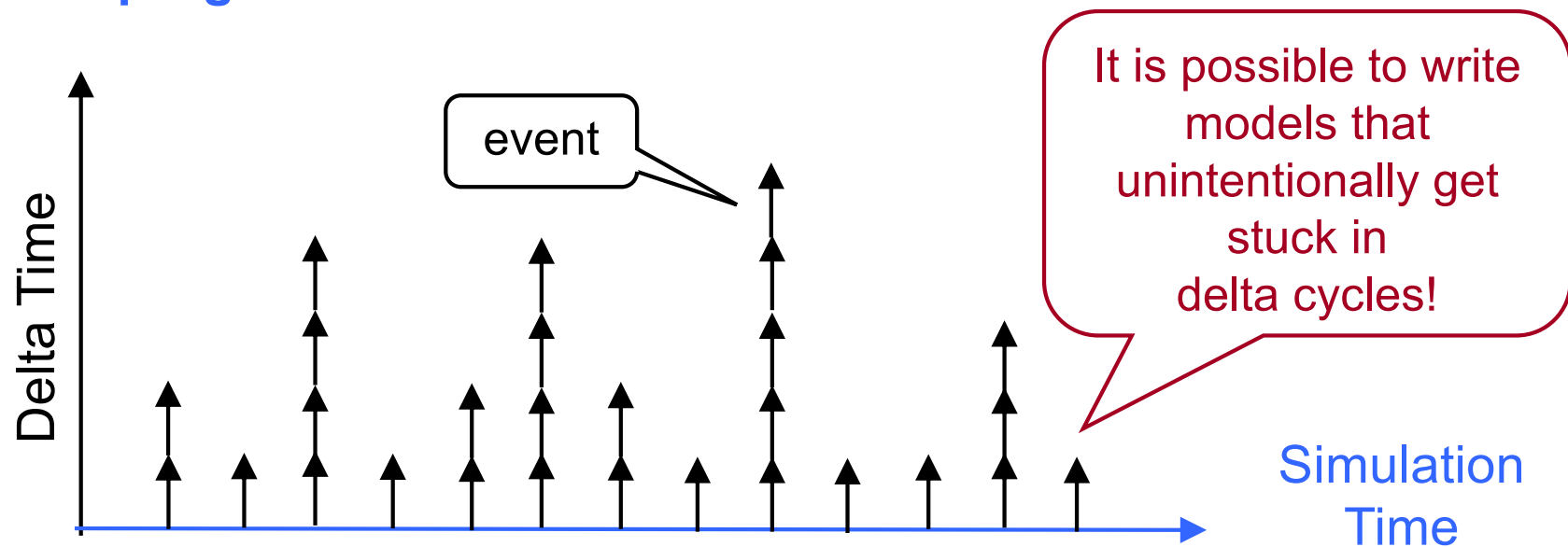


- NOTE: Simulation progress is first along the zero/delta-time axis and then along the simulation time axis.

# Event-Driven Simulation 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**
    - records the points in time when events happen
  - **Event queue**
    - 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
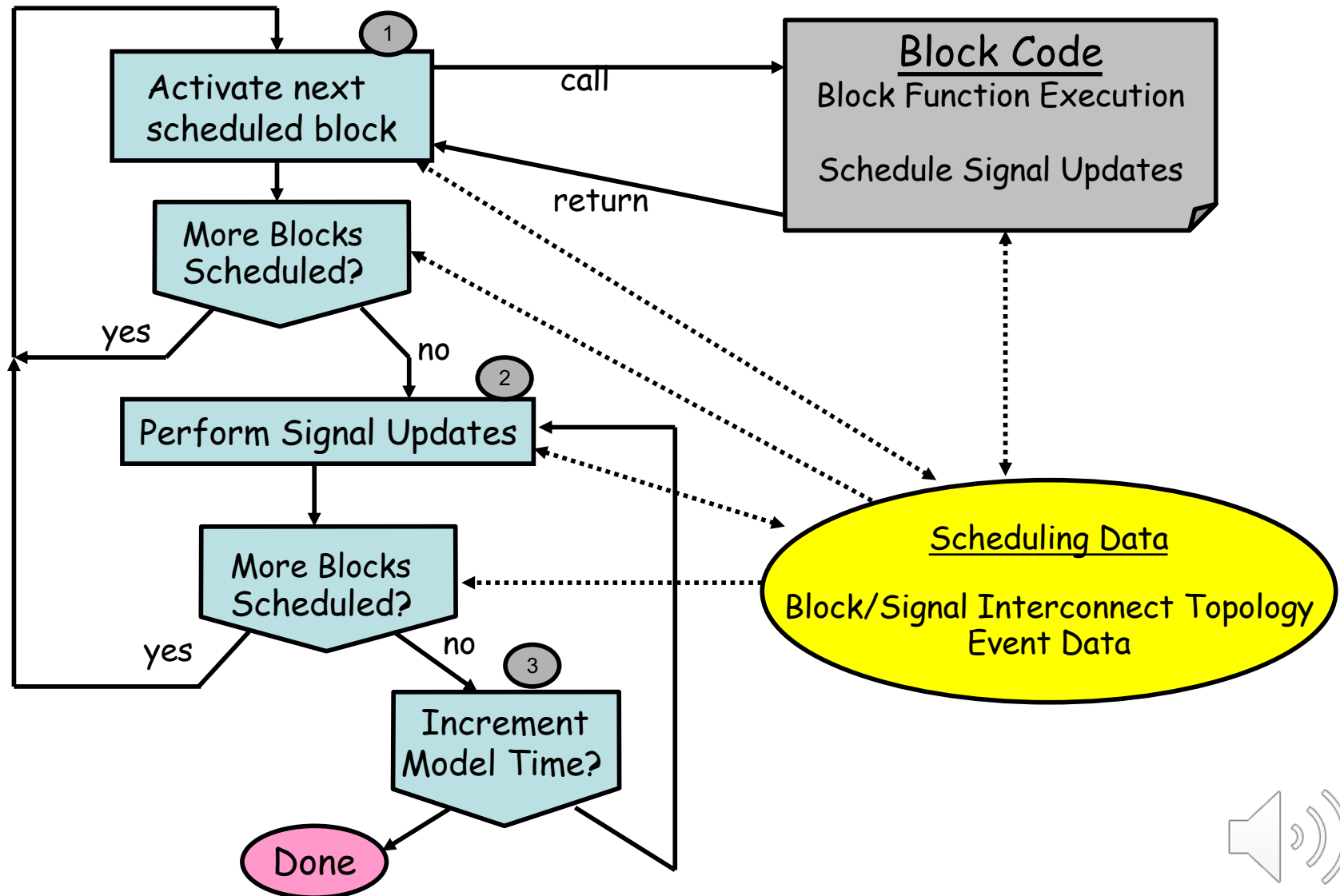
# Simulation Speed

## 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

  **SSE: further information on the above three methods is provided in the slides and explained in more detail in the pre-recording on BB unit page**

  - Design 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 delivered a breakthrough
  - Hard to compete against "trivial parallelization"
- **Simple solution (brute force) – run independent testcases on separate machines**
  - Workstation clusters called "Simulation Farms"
  - 100s - 1000s of engineer's workstations run simulations 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

- **Cut down the number of scheduled events to reduce simulation overhead**
  - 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



Few large blocks:
High re-scheduling overhead

Many tiny blocks:
High scheduling overhead

Performance

Model granularity
Number of blocks

Decreasing
Model Activity
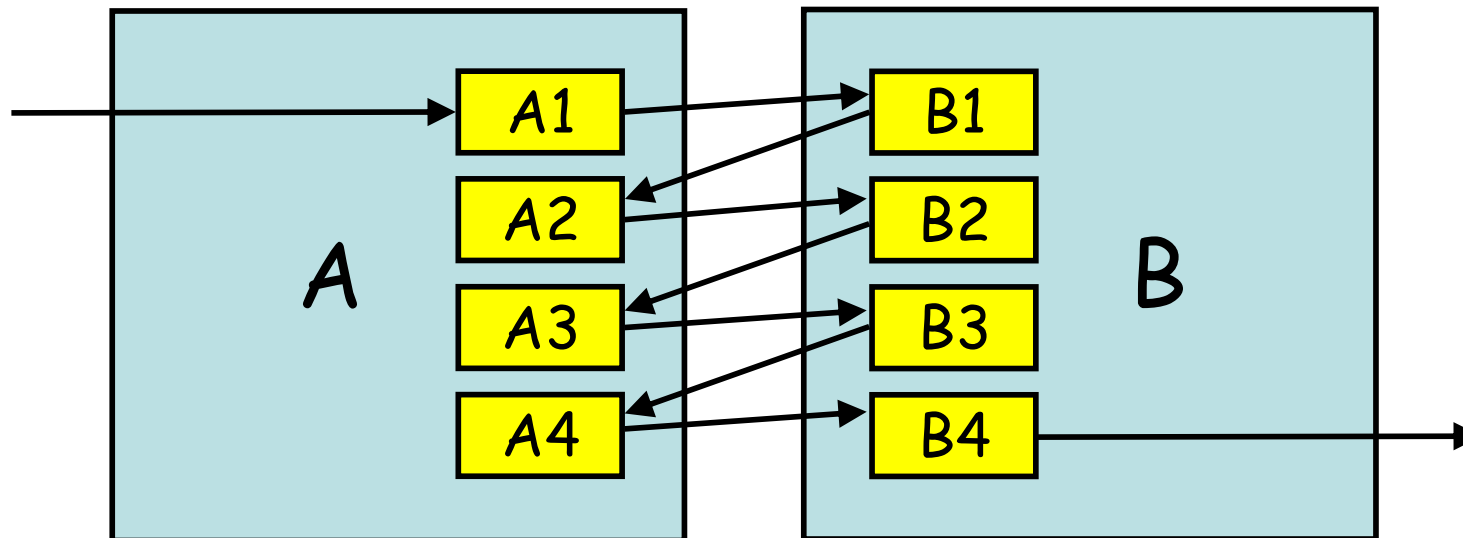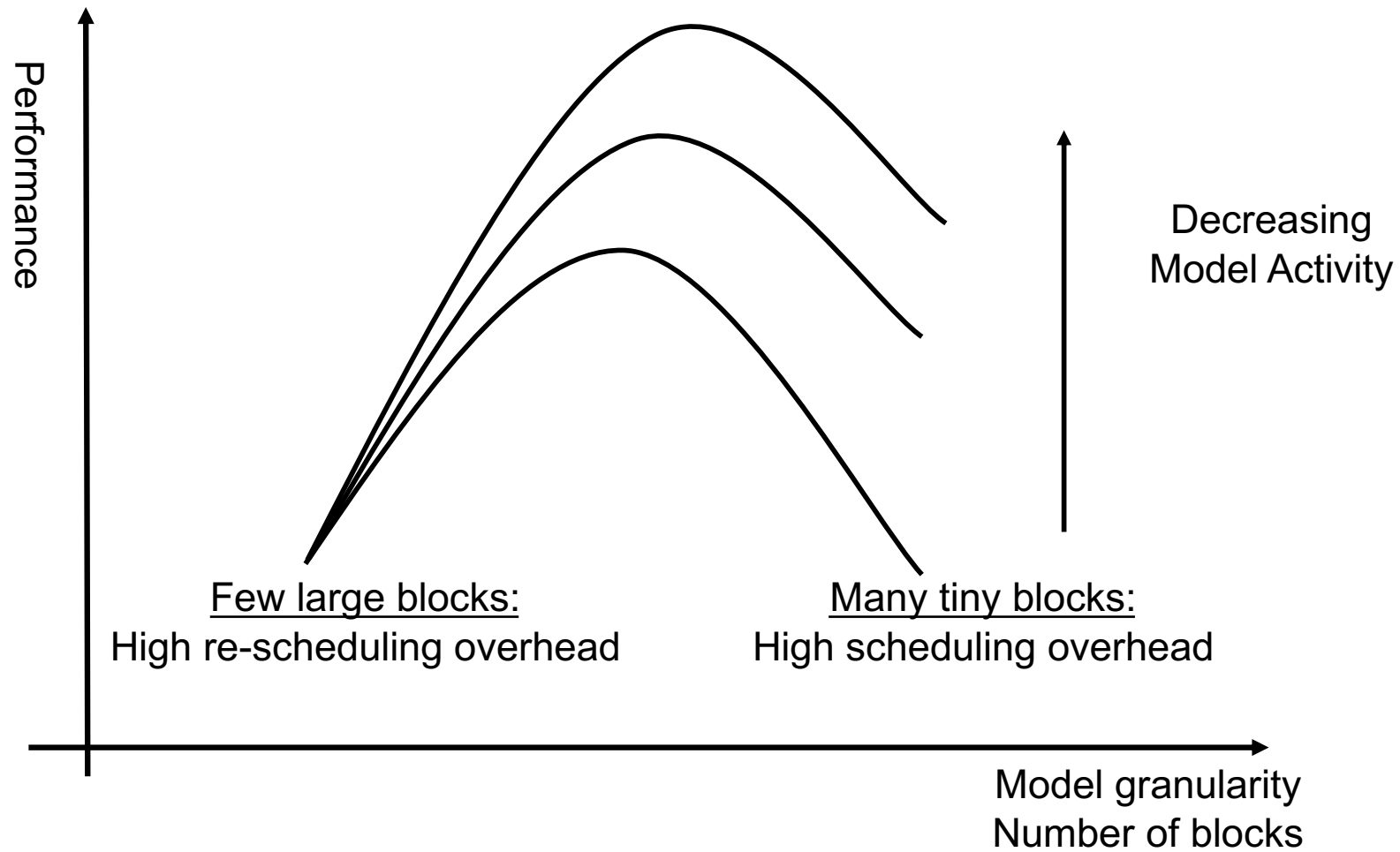
# Two types of simulators:

event-based and
cycle-based

# Two types of simulators:

event-based and
cycle-based

# Synchronous Design Methodology

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



Primary inputs

State Transition Function
(Combinational Logic)

Primary outputs

State-Holding Elements

Clock(s)
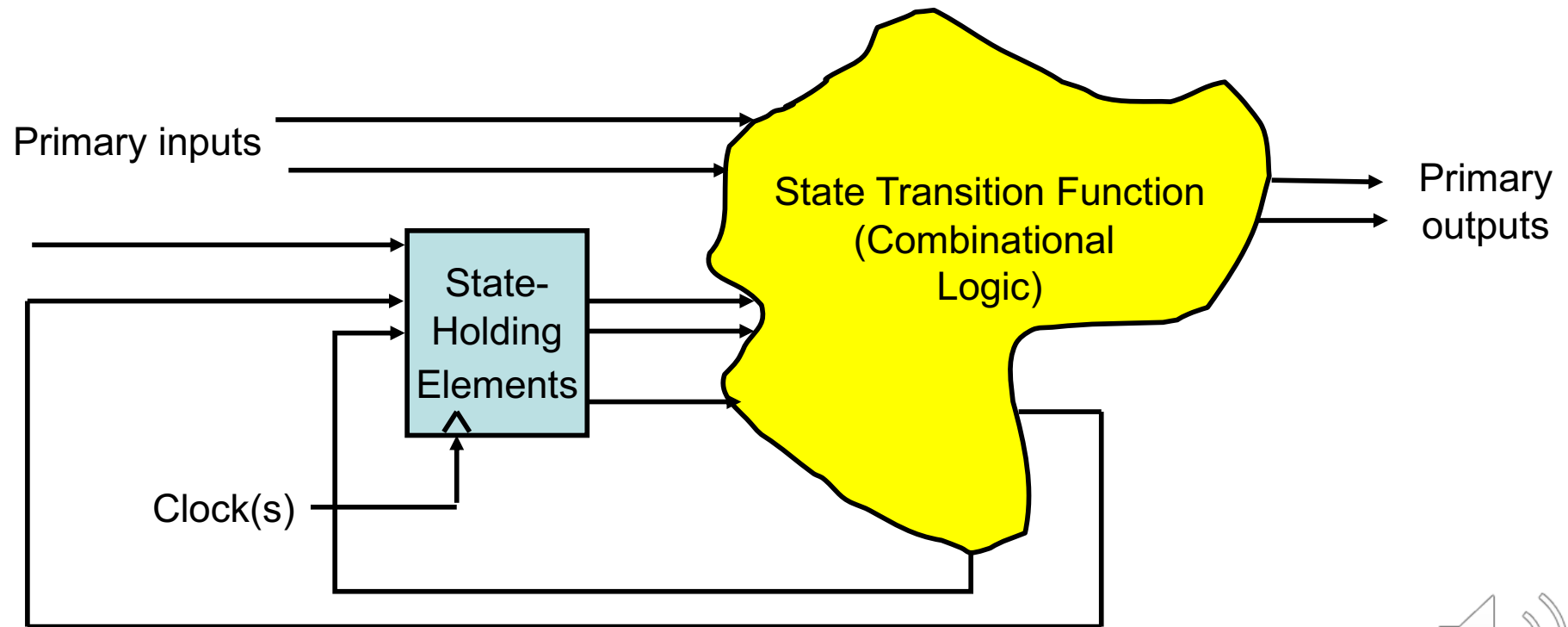
# Cycle-Based Model Build Flow

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

# Model Build – Levelization

Logic is ordered into levels so that the order of evaluation is correct.

e.g., A and B are computed before C.
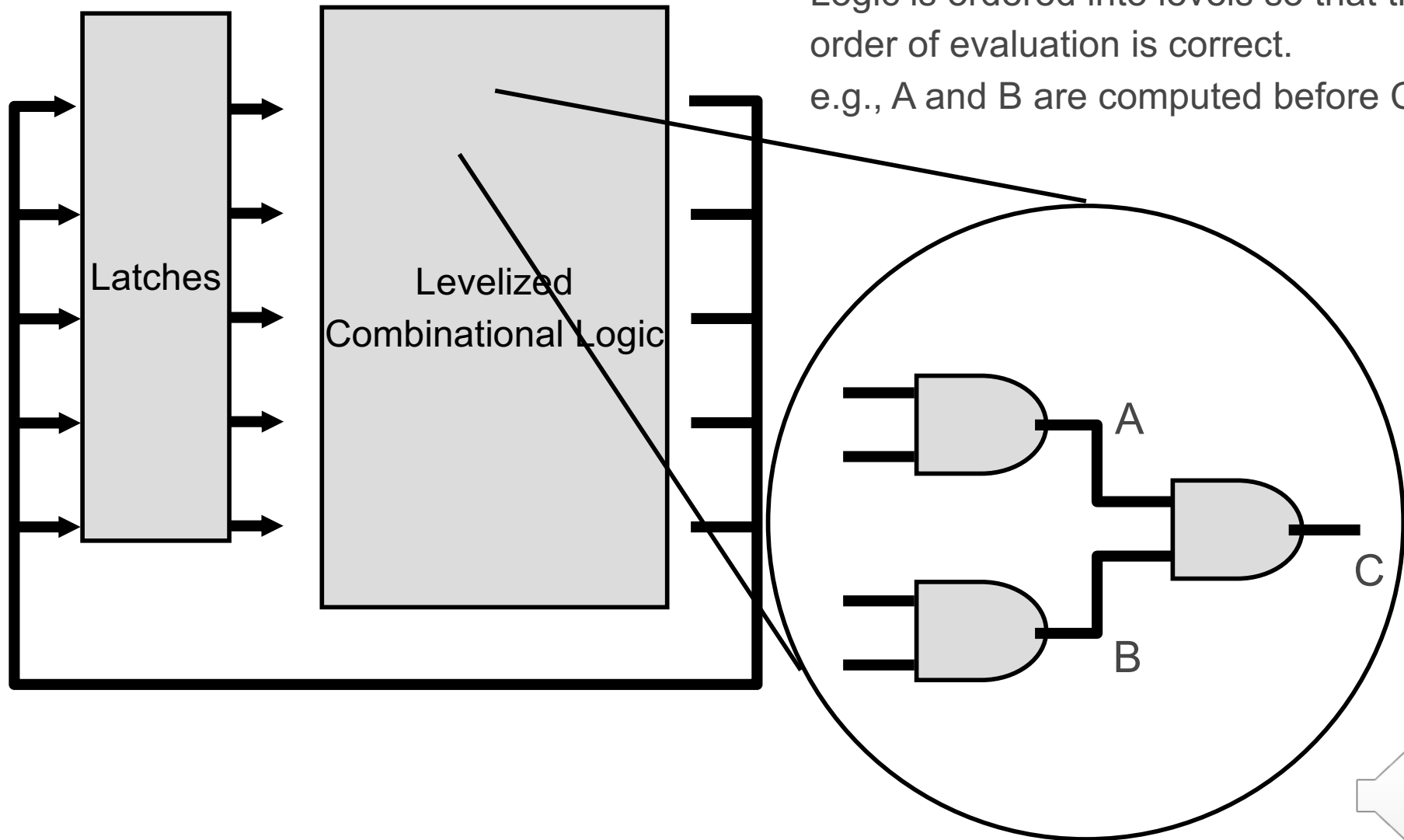
# Translate to Instructions
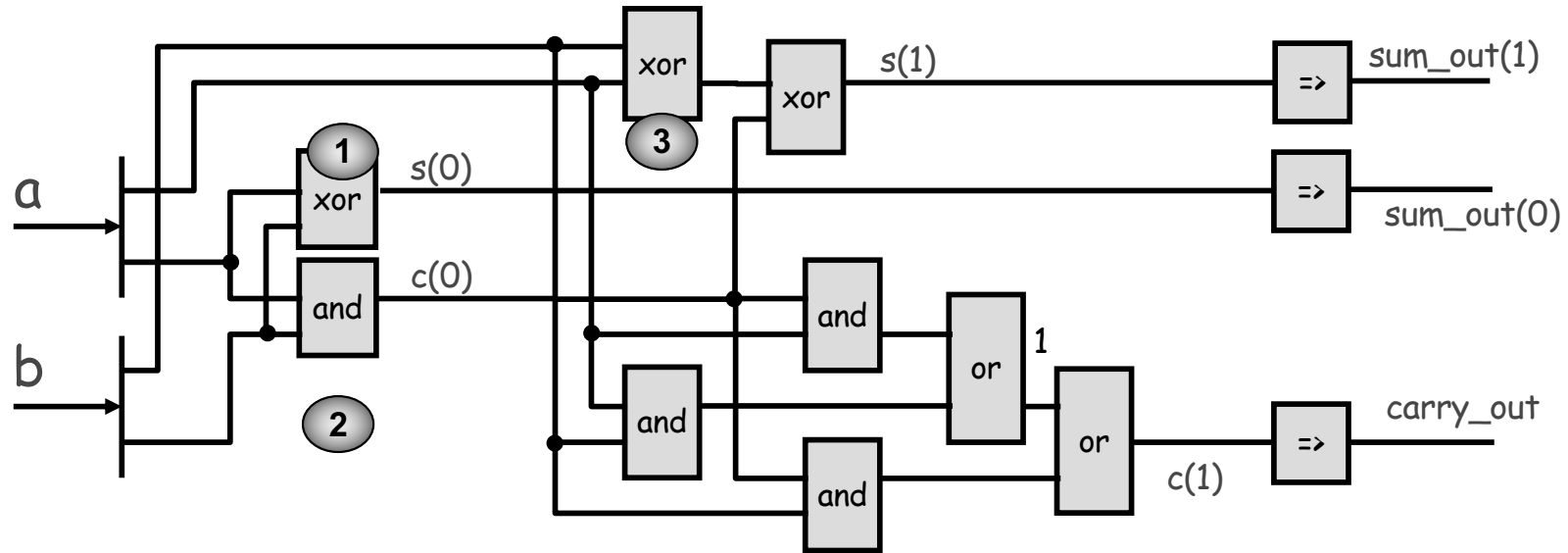


Load temp1, a(0)
Load temp2, b(0)
1  Xor temp1, temp2, temp3
Store temp3, s(0)
2  And temp1, temp2, temp3
Store temp3, c(0)
Load temp1, a(1)
Load temp2, b(1)
3  Xor temp1, temp2, temp3
...

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

35

# Parallelism in Generated Code

- 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)
```

# Speed of Cycle-Based Simulation

- Clock the design only as fast as the longest possible combinational delay path settles before the cycle is over

- Cycle time depends on the longest topological path
  - Longest topological path can be calculated using static analysis w/o using simulation -> stronger result w/o simulation patterns
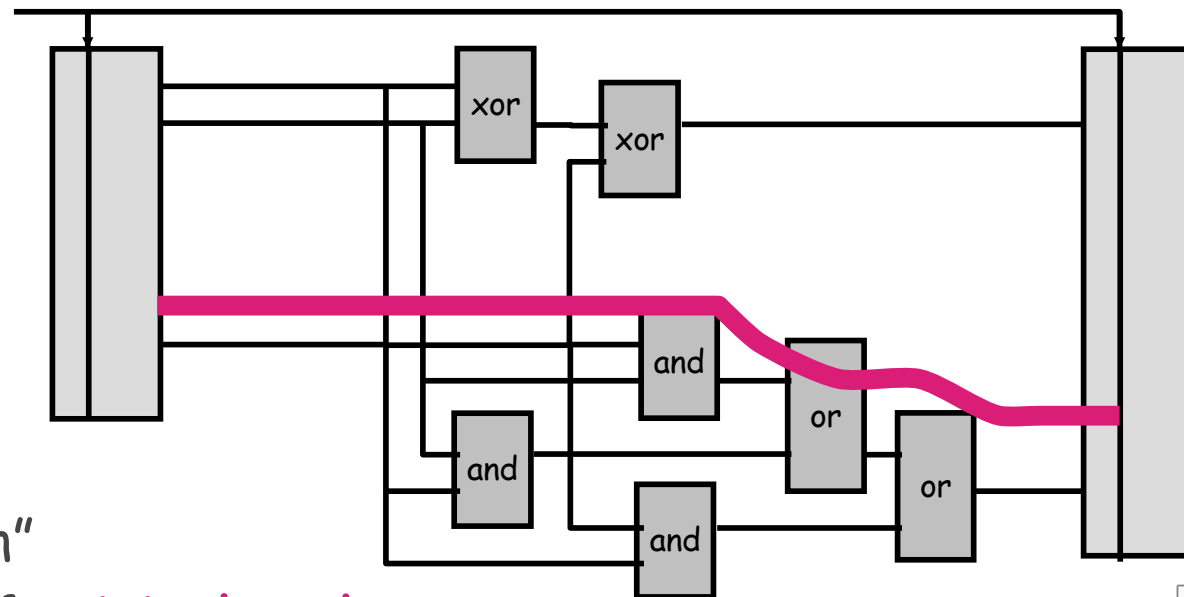


clock cycle "duration"
depends on length of **critical path**

# Hardware Acceleration

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

# Scheduling Example
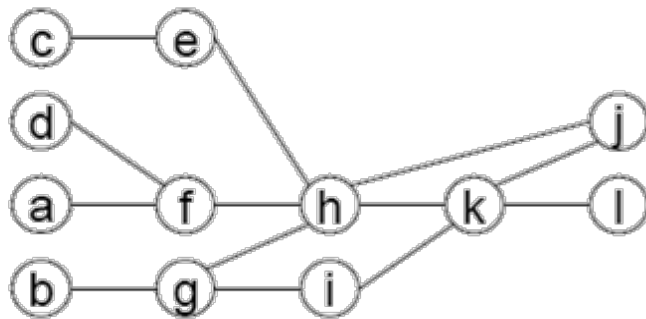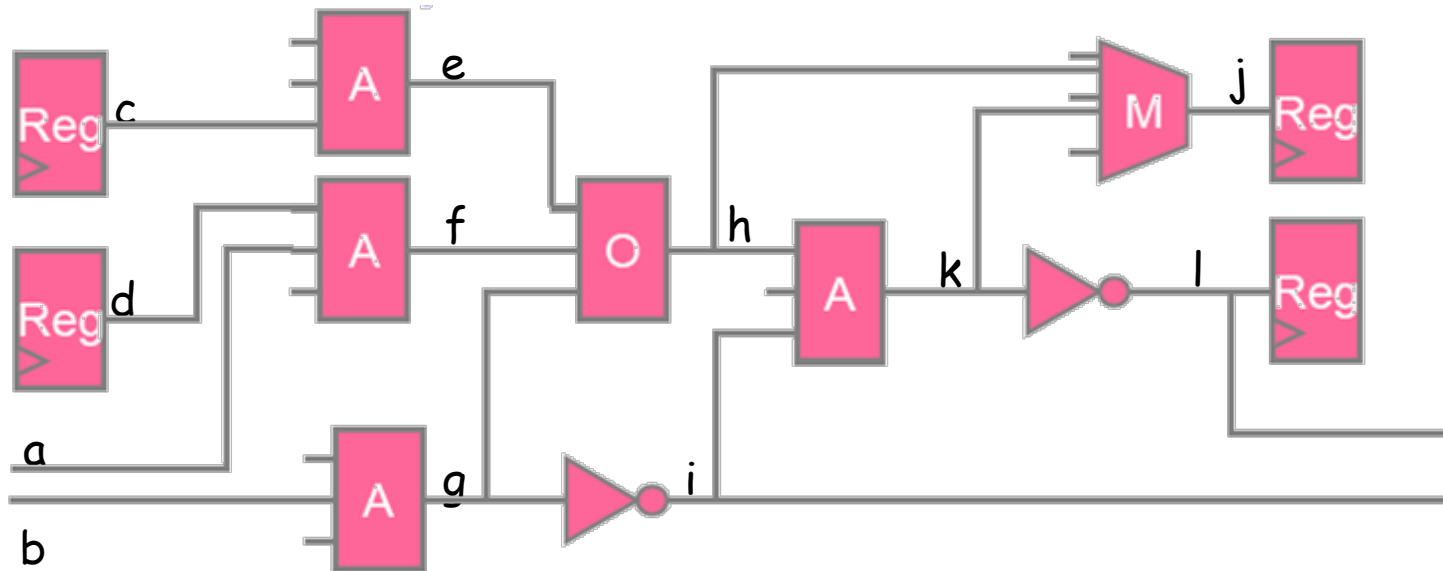


|       | EP1 | EP2 | EP3 | EP4 |
|-------|-----|-----|-----|-----|
| Step1 | b   | a   | d   | c   |
| Step2 | g   | f   |     | e   |
| Step3 | i   | h   |     |     |
| Step4 | k   |     |     |     |
| Step5 | j   | l   |     |     |

12 steps serial, 5 steps parallel

39

# Accelerator Basic Structure
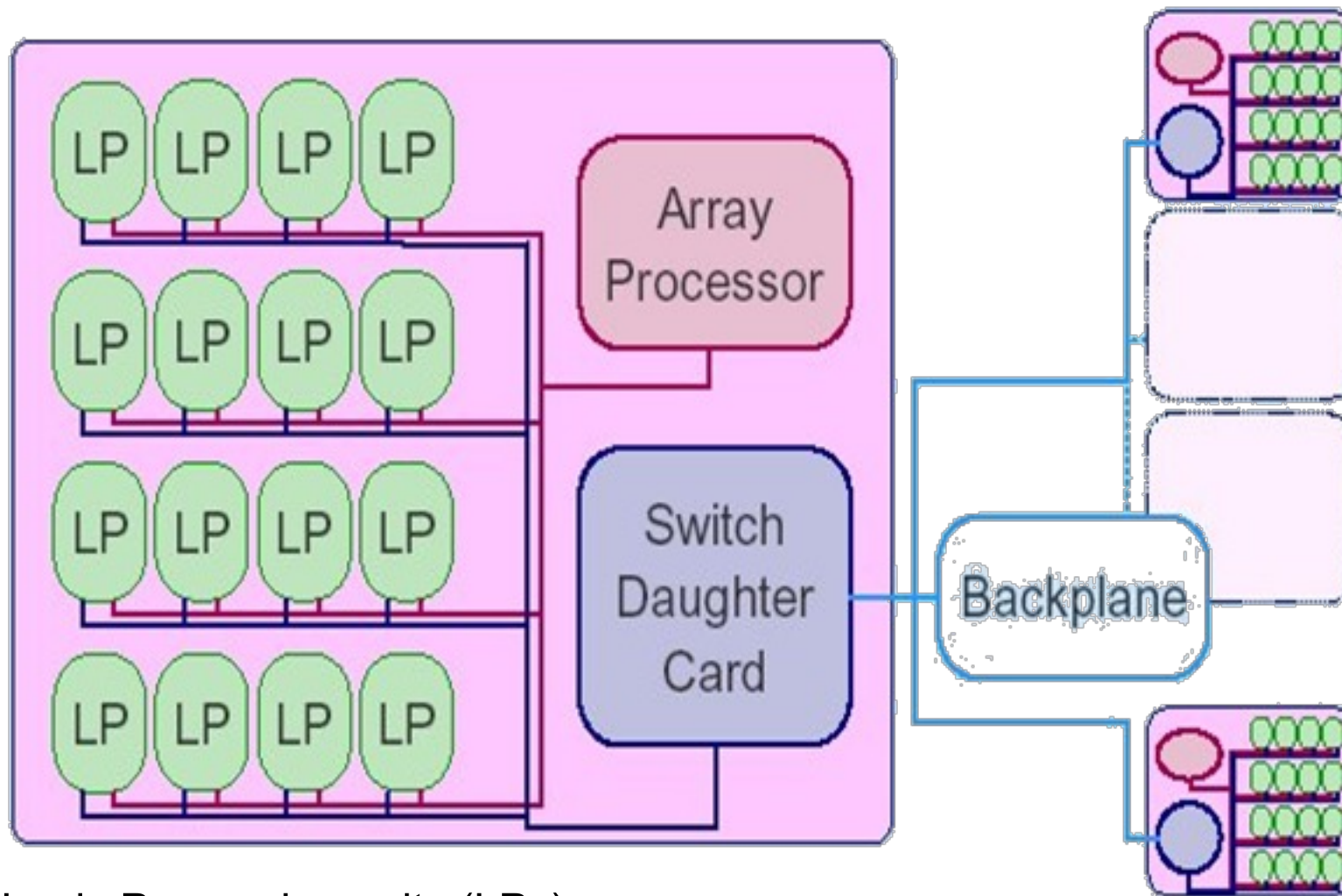


Logic Processing units (LPs):
These are special purpose processing units, that are much faster than CPUs.

# Principle of Operation

- **Compiler transforms combinational logic into Boolean operations**

- **Compiler schedules inter-processor communications using a fast broadcast technique**

- **Performance dictated by**
  - Number of processors (Logic Processing units)
  - Number of levels in the design

# Simulation Speed Comparison

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

# Simulation Speed Comparison

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

# Putting processor verification into context....

## 1,000,000,000,000,000

The number of verification cycles Arm uses when verifying an Arm core

- SystemVerilog simulator executes 2,000 cycles / second
  - => 15,000 SystemVerilog simulators running for 1 year
- HW emulator or FPGA runs at 1,000,000 cycles / second
  - ⇒ 30 years of running needed...

- OK – so this is for a high performance OoO, MP, VM core (full apps processor)
  - Embedded processors will be an order of magnitude less...

Verification Futures UK 2023

**RISC-V verification and implications of the 5:1 ratio of DV to design engineers**
https://youtu.be/clL97C7jmUE

# 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 of HDLs 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)

46

# 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
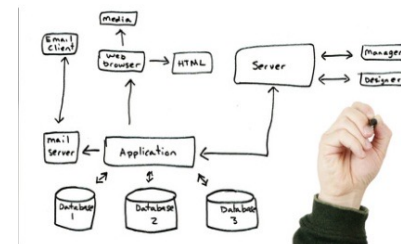  - Language constructs to implement functional coverage models

# 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, ... ☺

# (other)
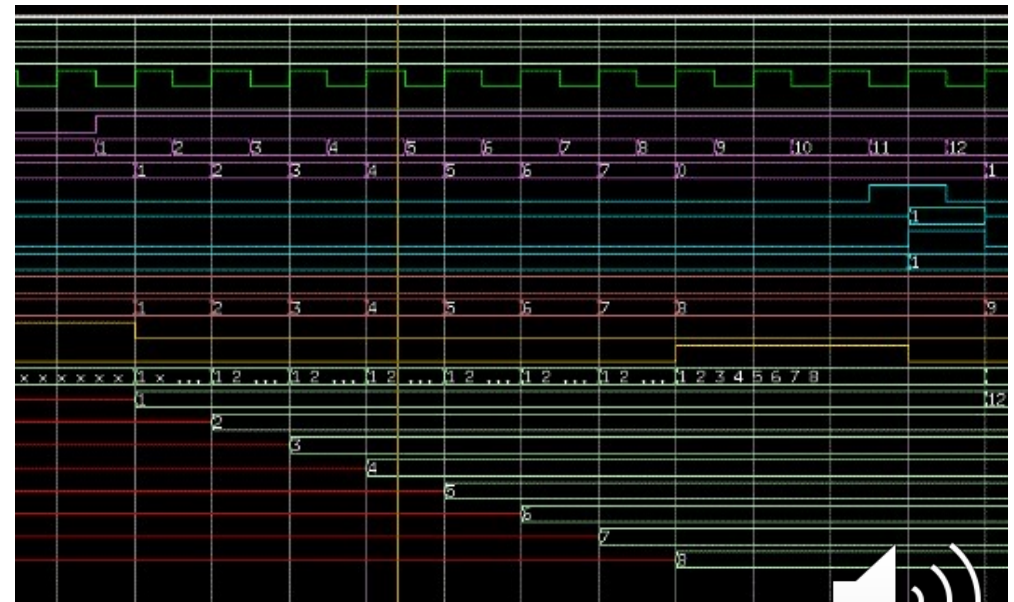# Verification Tools

## Waveform Viewers
## Third Party Models
## Metrics

# Waveform Viewers

- **Waveform viewers** 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 waveform viewers to determine whether the DUV passes or fails a test.**
  - **Why not?**

- **Can use waveform viewer for debugging.**

  - Sophisticated debuggers link from the wave directly to the source code and highlight the related logic.

# 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.

# 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.

# 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.
- **Other metrics include, e.g.:**
  - Number of lines of code
  - Ratio of lines of code
    (between design and verification)
  - Drop of source code changes
  - Number of outstanding issues

# Summary

- Basic introduction to testbenches

**We have studied**

- Verification Tools
  - including an outlook on Verification Languages

- Linting tools – static checkers

- Simulators: event-based and cycle-based
  - How to improve the performance of simulation-based verification
  - Compared the speed of different simulation techniques
  - Dangers of using waveform viewers as checkers

# Become an SVA Expert in One Hour
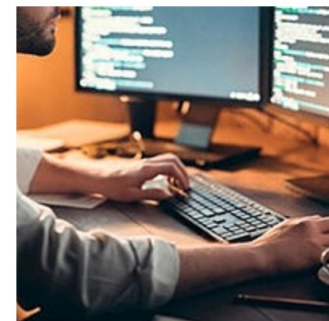
Wed, Sep 25, 2024 10:00 AM - 11:00 AM BST

Show in My Time Zone

In this webinar, Doulos Co-Founder & Technical Fellow John Aynsley will teach the core principles necessary to understand and use SystemVerilog Assertions, focusing on the aspects of SVA that are applicable to both formal verification and simulation.

Particular emphasis will be placed on the core semantics of temporal logic so that you will be able to write your own assertions, understand what you are doing, and avoid the many pitfalls that trap beginners. SVA is really not hard if you approach it properly!

This training webinar will consist of a one-hour session and will be interactive with Q&A participation from delegates.

Attendance is free of charge.

*Required field

First Name*

Last Name*

https://register.gotowebinar.com/register/816225097157355863?source=lastchance

55