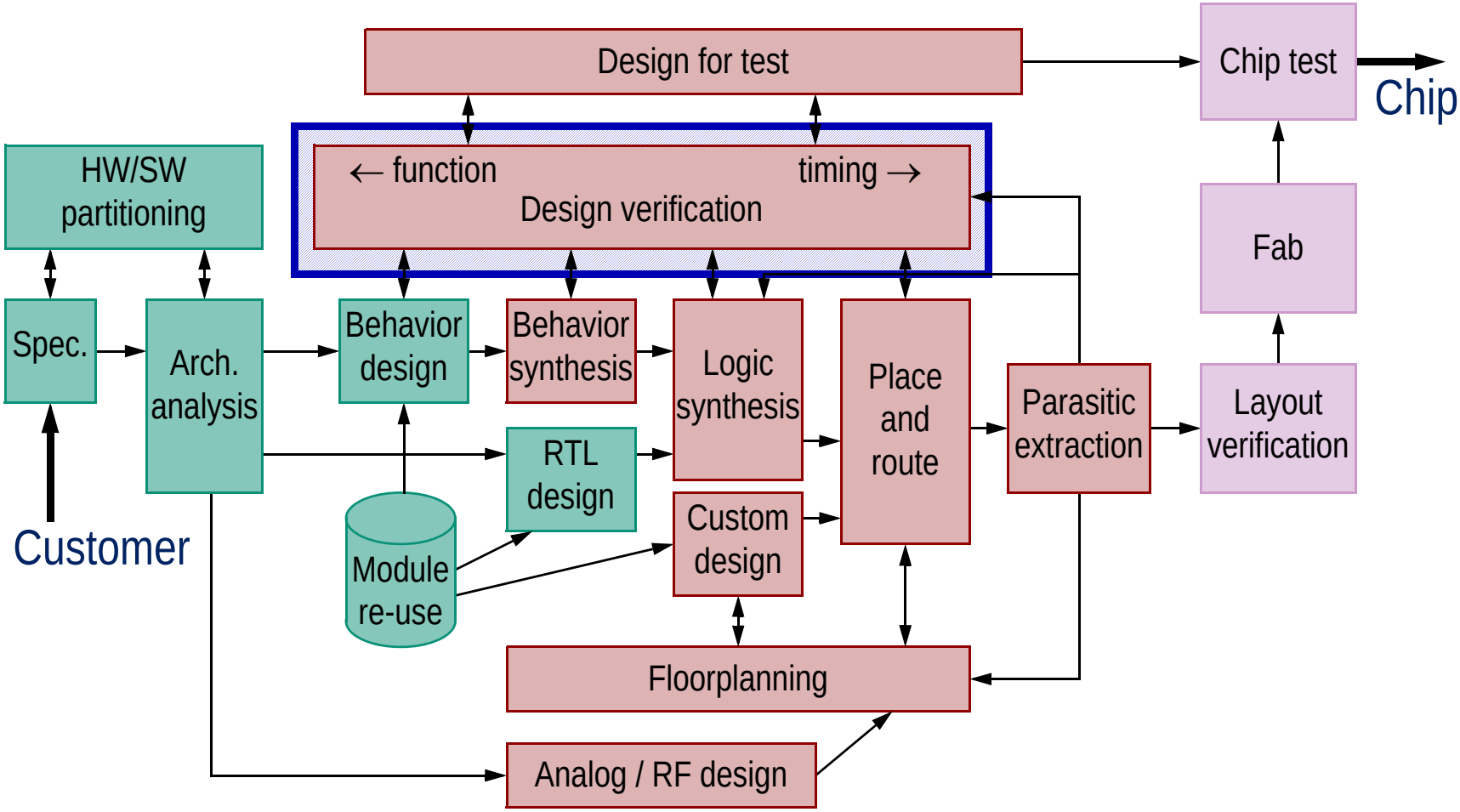
DAT110
Methods for Electronic System Design and Verification

Per Larsson-Edefors
VLSI Research Group

Lecture 2: Functional Verification.

Conventional Verification



Some Types of Verification

- Prototyping: PCB-based test (bread-boarding).
- Equivalence checking:
  check consistency specification $\rightarrow$ implementation.
- Correctness-by-construction:
  prove refinement preserves behavior.
- Simulation at different levels.
- Timing analysis (static) for delay verification.
- Layout verification (DRC, ERC, Extraction, LVS ...).

Verification Options for HDL Code 1(2)

- Logic (digital) simulation.
  - Slow (but not compared to circuit simulation).
  - Not exhaustive (depends on test vector set).
- Acceleration using special purpose hardware.
- Emulation; implementing advanced system (e.g. ASIC) on a simpler platform (e.g. FPGA).
  - High NRE-cost (no reuse possible).
  - Slower than the final hardware, but faster than simulation.

Verification Options for HDL Code 2(2)

- Formal verification.
  - Large state space - practical only for some problems.
  - Non-intuitive - demands expert user.
- ‘Linters’ (from lint).
  - A heuristic static parsing strategy for inspection of code.

Context of Functional Verification

1. Determine intent - specification.
2. Determine actual function - implementation.
3. Compare intended function and actual function.
4. Consider the confidence of comparison results.

Logic Simulation

1. Given a piece of HDL code and a digital stimuli, the test vectors, ...
2. evaluate the HDL code using a testbench ...
3. by, for each stimulus, checking the HDL code output result against “golden” reference.

- Purpose:
  - Find design implementation errors early, before circuit design.
  - Verify that implementation matches specification.

Challenges with Specification Definition

- Even for derivative designs, application use-cases are often used to determine the intended function - the specification.
  - With system complexity and radically new consumer patterns, calling for immature applications, how to establish use-cases?
- Specifications are often first expressed in natural languages.
  - A natural language specification is ambiguous, has gaps and may be full of mistakes/misunderstandings.
  - Specifications in an executable form offer both a conciseness and allow for deliberations on HW/SW partitionings [Lecture 1].

Executable Models

- Clearly it is possible that mistakes are made
  not only during implementation,
  but also during the definition of a specification.
- Striving to express specifications and implementations as some kind of (executable) models helps to eliminate mistakes and ambiguities.

  This is important as an implementation often
  acts as specification for a lower design level.

Challenges with Implementation Process

- How do we implement the design specification correctly?
  - This is the obvious challenge that we focus on in traditional engineering.
- Just as important though, and a bit forgotten is:

  How do we implement the specification into a verification reference correctly?
  - For new designs: How do we create the golden reference?
  - Since the verification reference implementation itself is not targeting a product, it may become a “second-class citizen”. Use separate design and verification teams!

Correct Functional Verification

1. Determine intent - specification.

2a. Determine actual function - design implementation.

2b. Determine function in a different manner - verification implementation.

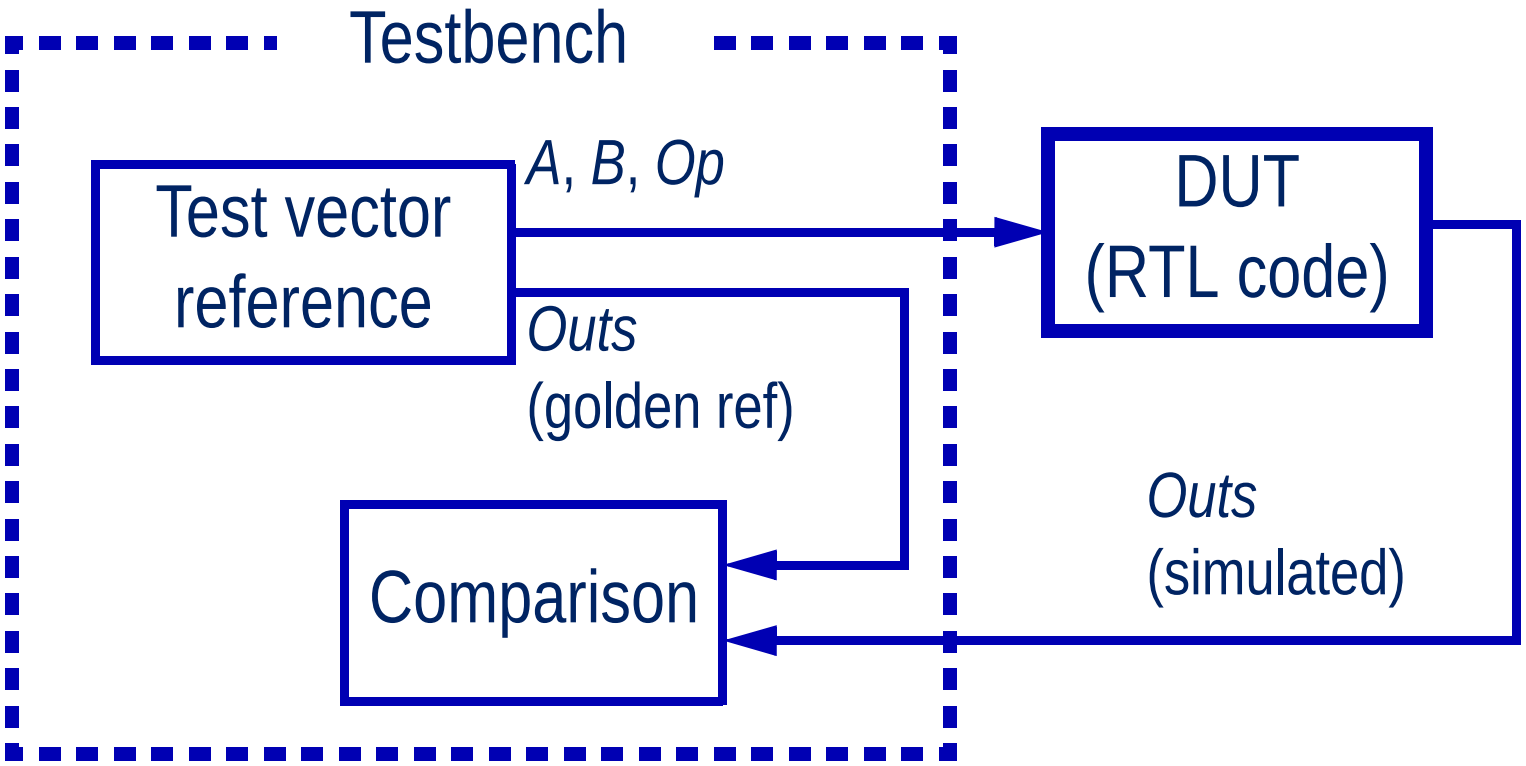
3. Compare design implementation and verification implementation.

4. Consider the confidence of comparison results.

Some Categories of Simulation

- Logic simulation (RTL and gate level) Digital
- Switch-level simulation (bidirectional data flow) Digital
- Circuit-level simulation (SPICE, Spectre) Analog

Logic Simulation Context



Logic Simulation Strategies 1(3)

- Consider the use of assertions inside the testbench.
  - ALU opcode should not be outside the valid range.
  - A half adder should never deliver Sum = 1 *and* Carry = 1.
  - Assertions become an ‘executable databook’ for blocks.
- Stimulate HDL code using test vectors for directed tests.
  - Instruction Set Simulator (ISS) simulation using benchmarks.
  - Test vectors that target blocks that are known to be prone to errors.
- Stimulate HDL code using random test vectors.
  - Pseudo-randomization can uncover errors in unexpected places.

Logic Simulation Strategies 2(3)

- Evaluate the simulation output either by ...
  - waveform eyeballing (use only for debugging !).
  - comparison against a “golden” reference file:
    An expected results file.
  - comparison against a “golden” reference design:
    A behavioral model that we know is correct or
    a previous design with the same functionality and cycle behavior.
  - checking the output log of assertions.

Logic Simulation Strategies 3(3)

- Since we can never exhaustively simulate real systems, we need to decide on an acceptable confidence level.
- The metric of coverage - percentage of items verified out of all possible items:
  - code coverage (syntax): % of RTL code lines simulated (use metric sensibly... 100% code coverage on flawed design?)
  - functional coverage (semantics):% of functional features simulated.
  - parameter coverage (semantics): % of operational ranges simulated, for example, is the range of depth of a FIFO fully simulated?

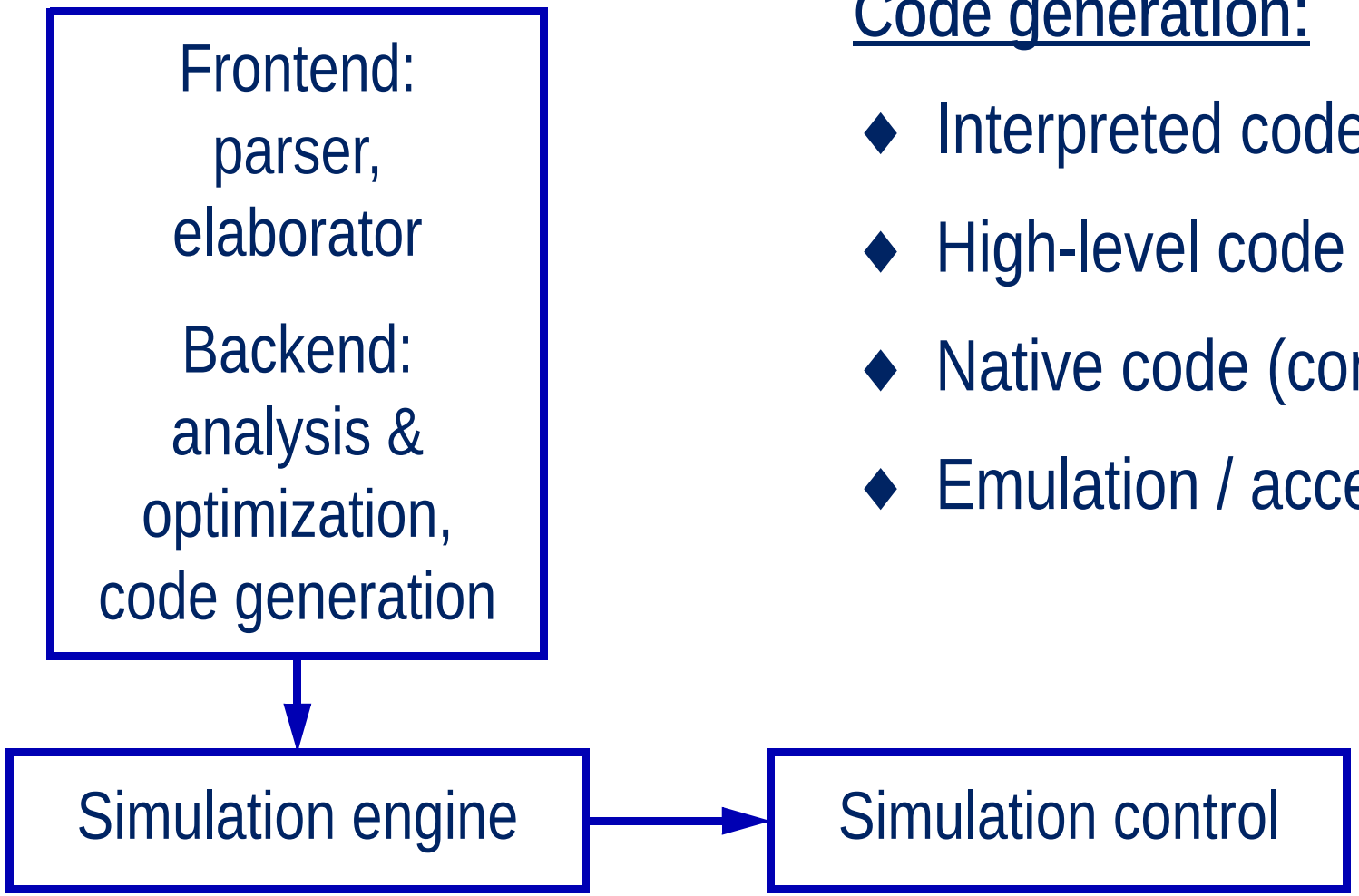
Logic Simulation Flow

- Apply random test vectors.
- Check code coverage [Sec. 9.3 of Ch9_FunctionalVerification].
  - Incisive averages block, expression and toggle coverage [pp. 409- in imc15.20_userguide.pdf].
  - Block coverage: Proportion of blocks (collection of lines after control statements) visited.
  - Expression coverage: Proportion of minterms [Lecture 4] visited.
  - Toggle coverage: Proportion of toggling signals in instance/module.
- Add directed test vectors for corner cases that happened not to be covered.

Bugs!

- Handle bugs!
- The life of a bug ...
  - opened:
    when detected (by design or verification eng), the bug is logged.
  - verified:
    when the designer confirms it is indeed a bug.
  - fixed:
    when the designer has removed the bug.
  - closed:
    when the surrounding code works fine.

Logic Simulators



Code generation:

- Interpreted code.
- High-level code (C etc.).
- Native code (common).
- Emulation / acceleration.

source: Lam

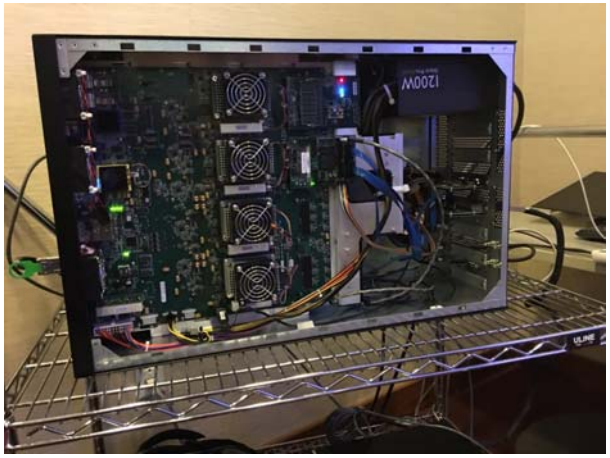
Acceleration and Emulation

- Indus of NVIDIA; emulation based on Palladium from Cadence.

source: NVIDIA

EMULATION

- An emulator represents an exact replica of actual hardware.
  - Fast!
  - Limited observability.
- Once RTL has been developed, use emulation to develop software on top of emulator.
- Example to the right: Protium (1-8X Virtex UltraScale XU440).



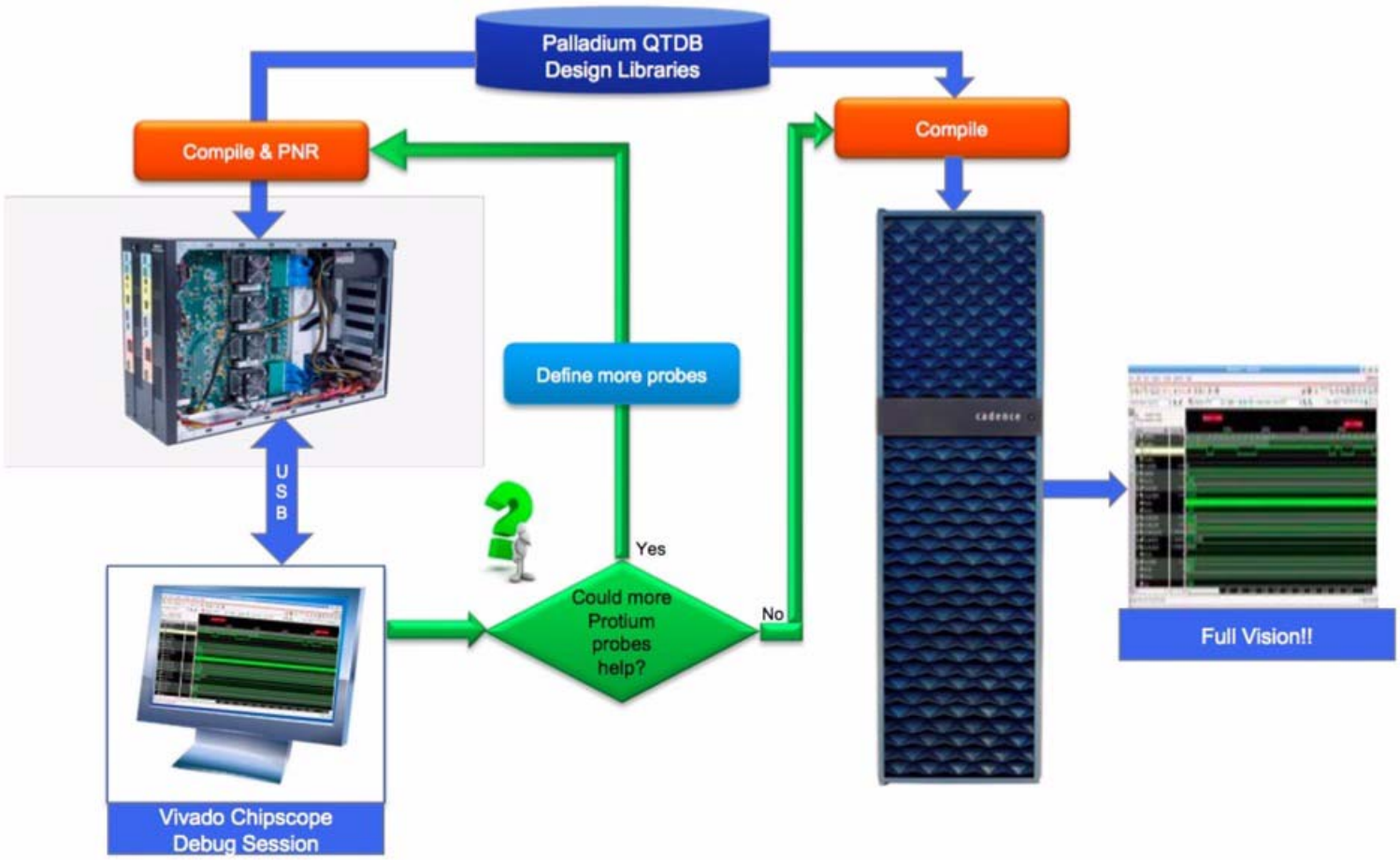
source: Cadence

Acceleration

- Simulation acceleration.
  - Slower than emulation.
  - Faster than simulation on desktop.
  - High, user-defined observability.
- Example to the right: Palladium.

source: Cadence

Mixing Acceleration and Emulation



source: Microsemi

Logic Simulation Engines

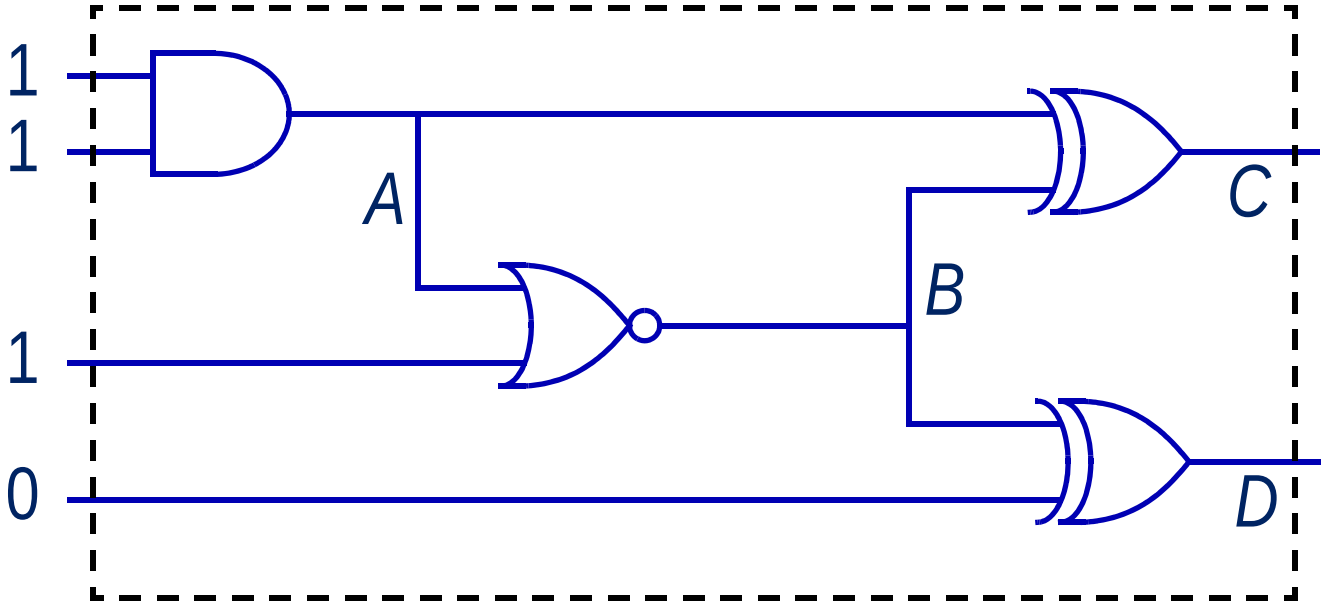
- Cycle-based (time-driven) approaches.
  - Evaluate logic states on all nodes between registers
  - Pursue the steady-state (per cycle) values.
  - New data evaluation every clock cycle.
- Event-driven approaches.
  - Evaluate only those nodes that change state.
  - Can find timing information, if delay models are accurate.
- Cycle-based approaches are faster for signal switching activities above 1%.

Logic Evaluation and Representation

- Unidirectional signal flow. (In contrast to lower-level simulators.)
- When the basic element is a gate, we call it gate-level simulation.
- Two-state representation: 0 and 1.
- Four-state representation: 0, 1, *X* and *Z*.
  - *X* represents either an uninitialized state or that several drivers enforce conflicting values onto the node.
  - *Z* represents a high-impedance (floating) node.
- Four-state representation is used during circuit power-up; then the simulator switches to two states to increase simulation speed.

Cycle-based logic simulation

Cycle-Based Simulation

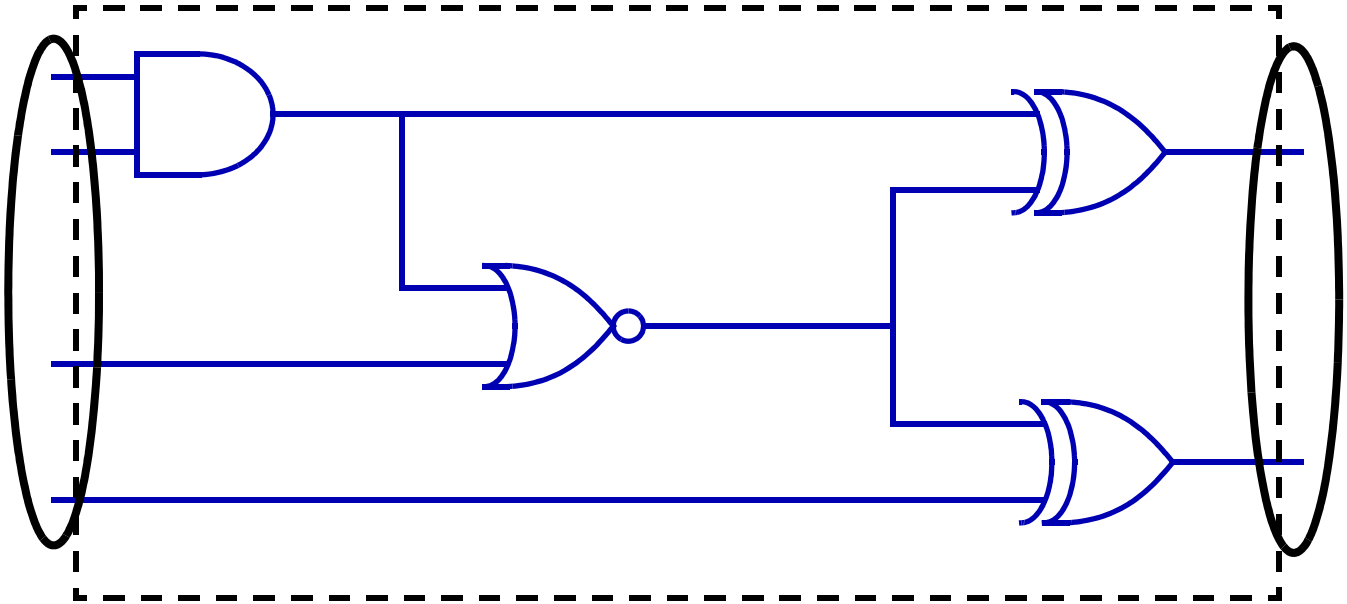


Evaluate all logic between register boundaries

1. A = 1 AND 1 = 1
2. B = A NOR 1 = 1 NOR 1 = 0
3. C = A XOR B = 1 XOR 0 = 1
4. D = B XOR 0 = 0 XOR 0 = 0

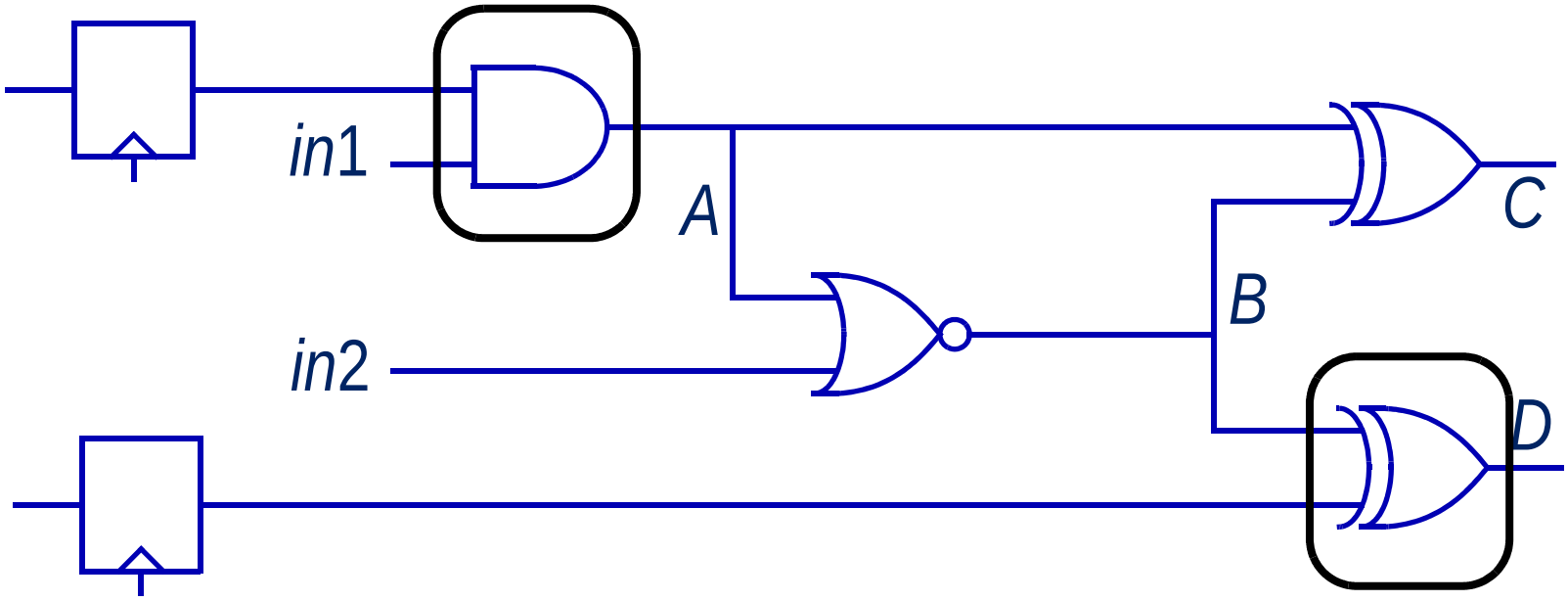
source: Lam

Primary Inputs and Outputs



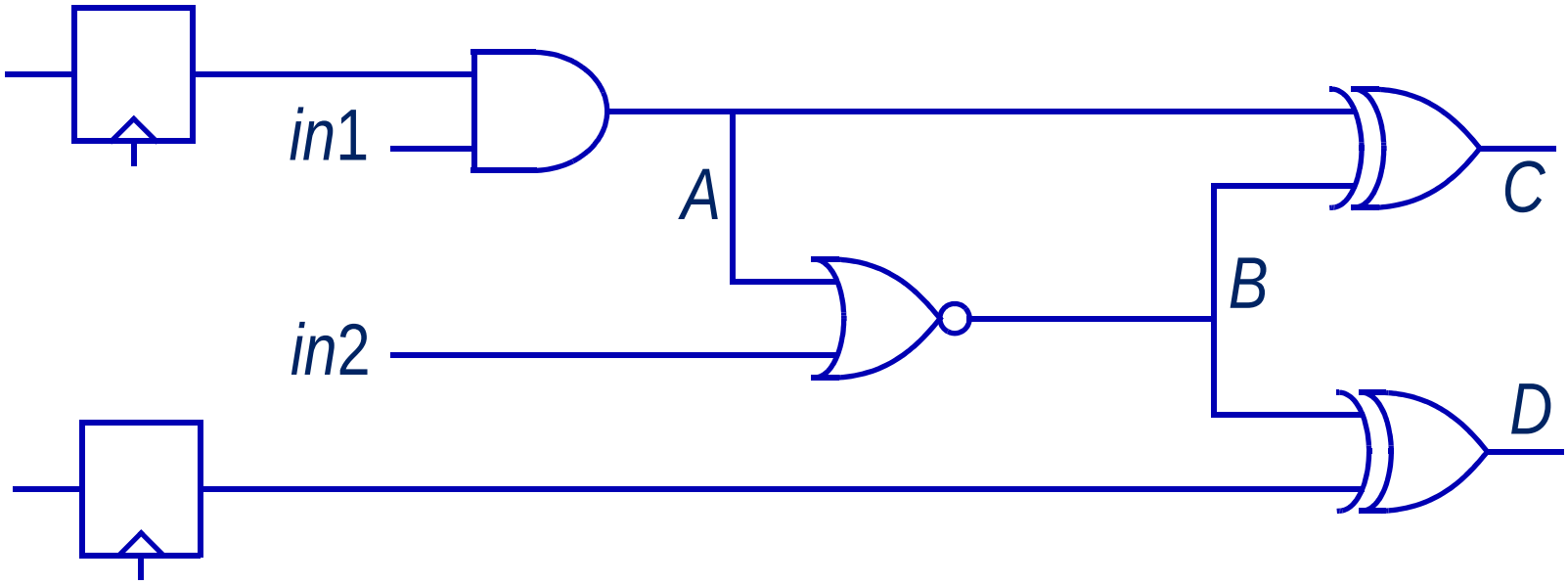
Generally the main inputs and outputs,
those on the boundary of a block,
are called primary inputs and outputs

Order of Evaluation Matters



FF outputs are the first event in an event-driven approach ⇒
gates with inputs from FFs would evaluate first ⇒
an outdated value of *B* would define *D*

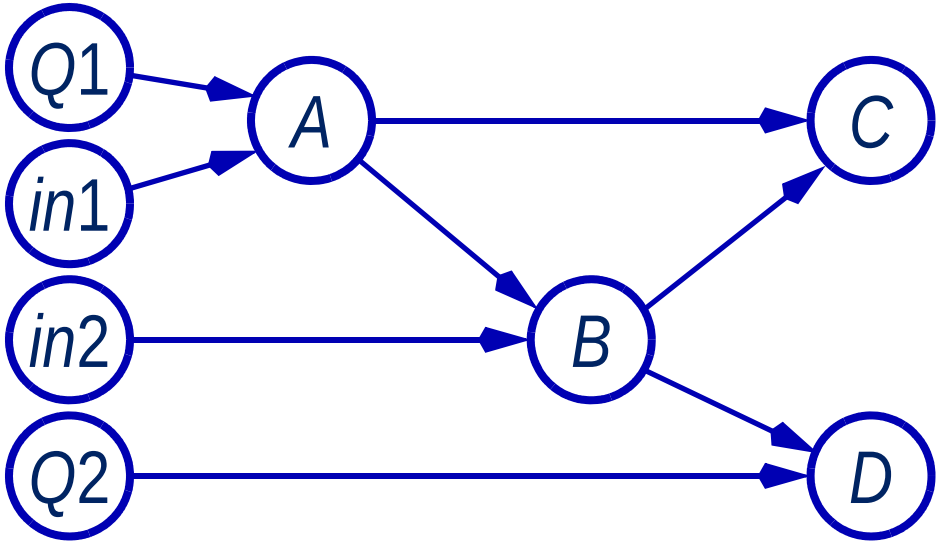
Levelization of Gates



In cycle-based simulation approaches, the criterion of levelization ensures that a gate evaluates its output only after its inputs have been updated.

This is important since each gate only is evaluated once!

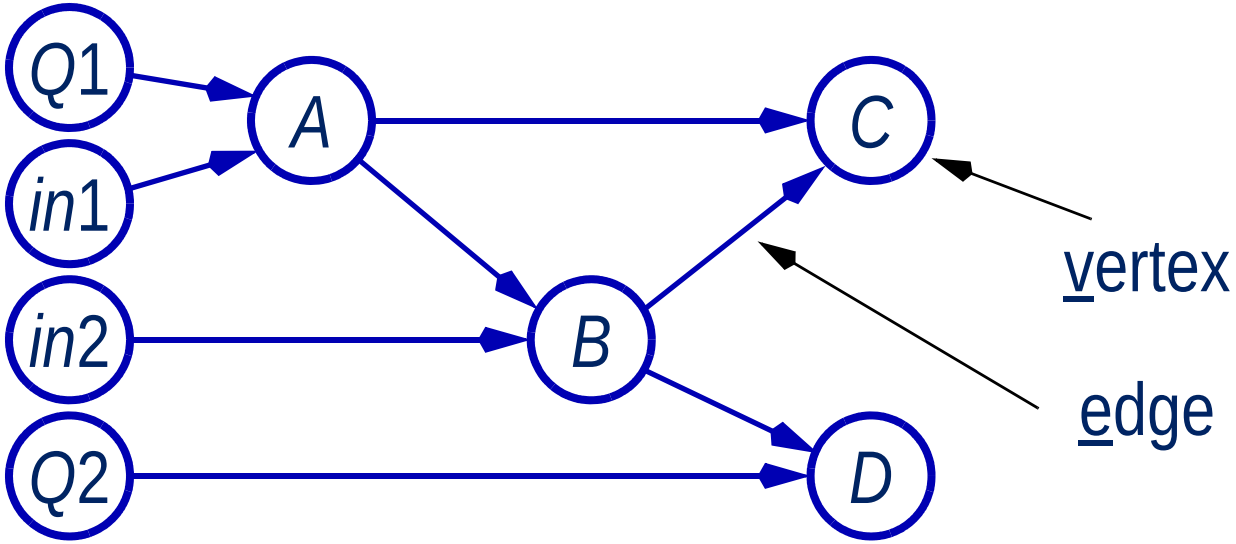
Algorithm for Levelization 1(5)



To establish in which order to evaluate the gates,
we have to use a systematic approach - an algorithm:

Graph theory - a branch of discrete mathematics [see later lecture].

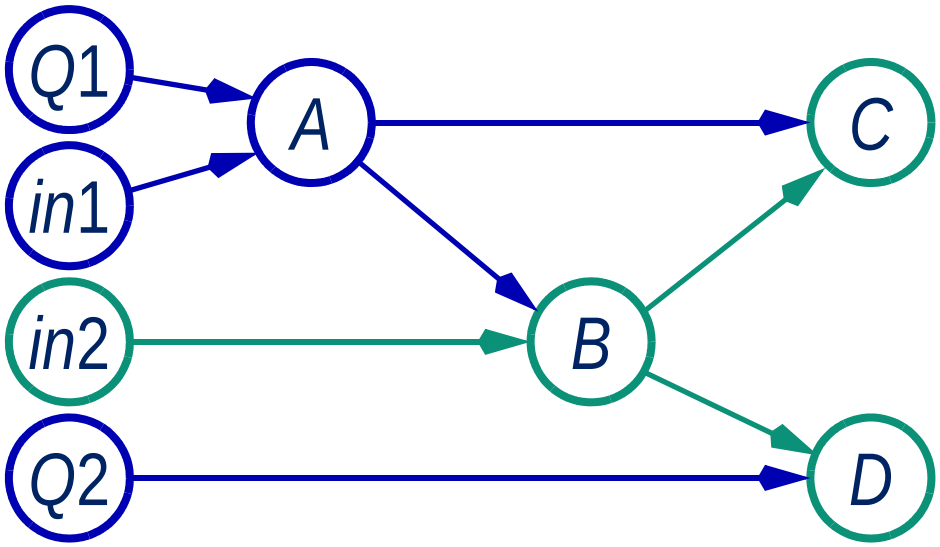
Algorithm for Levelization 2(5)



Use so-called depth-first search, starting at any primary input:

"Trace through graph until no more outgoing edge exists
or until the next vertex has already been visited,
backtrack and insert the visited vertex in an ordered evaluation list."

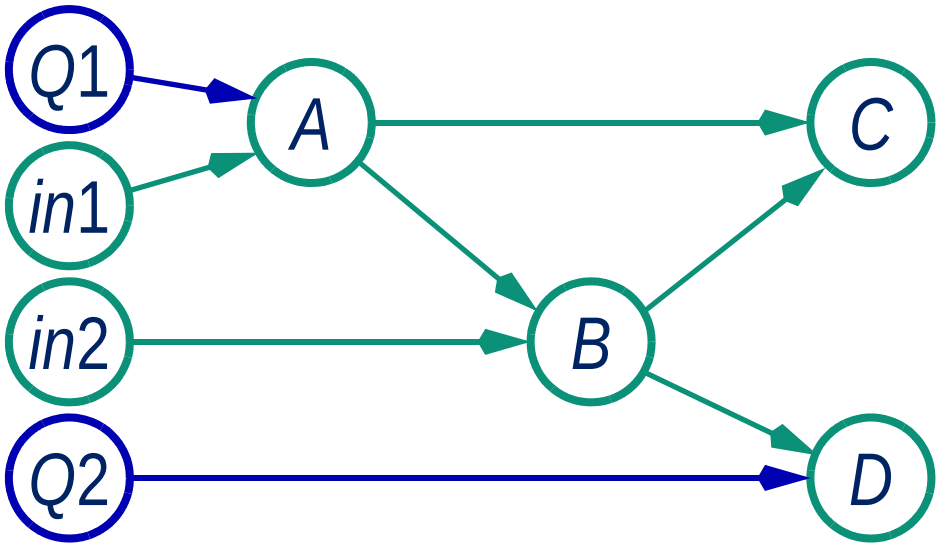
Algorithm for Levelization 3(5)



1. Start for example at *in2* and follow the edge to *B*.
2. Continue to *C* or *D*; here we (arbitrarily) choose *D*.
3. No edge: return and save *D*.
4. Go to *C*.
5. No edge: return and save *C*.
6. Return and save *B*.
7. Back at *in2*, we are done. Thus, save *in2*.

Evaluation list so far: *in2*, *B*, *C*, *D*

Algorithm for Levelization 4(5)

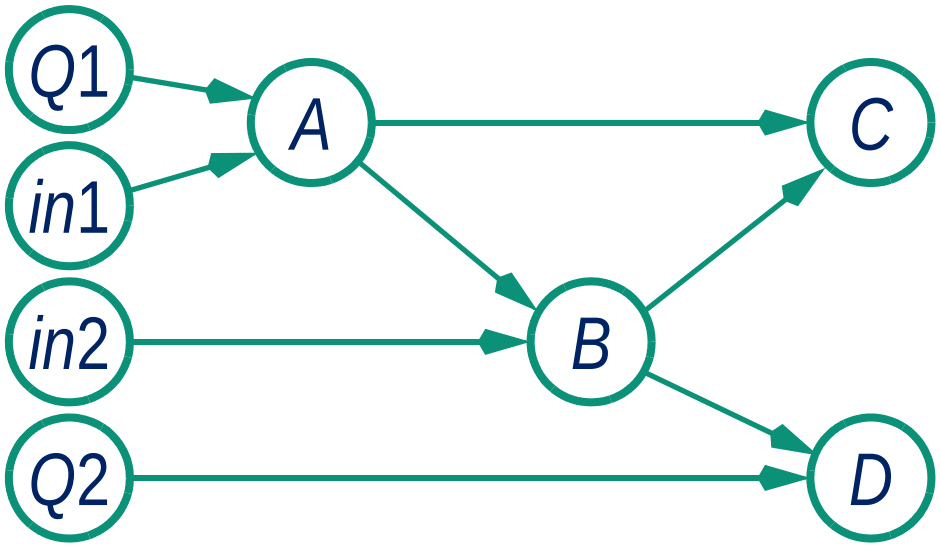


1. Now start with *in*1 and follow the edge to *A*.
2. In trying to continue to *B* or *C*, we discover these vertices have already been visited: Thus, save *A*.
3. Return and save *in*1.

Evaluation list so far:

*in*1, *A*, *in*2, *B*, *C*, *D*

Algorithm for Levelization 5(5)



1. Now start with *Q*1.
2. In trying to continue to *A*, we discover this vertex has been visited: Thus, save *Q*1.
3. After trying *Q*2, we end up saving *Q*2.

Final evaluation list:

*Q*2, *Q*1, *in*1, *A*, *in*2, *B*, *C*, *D*

An evaluation that follows the list guarantees that gate inputs always are evaluated before the gate's output is evaluated.

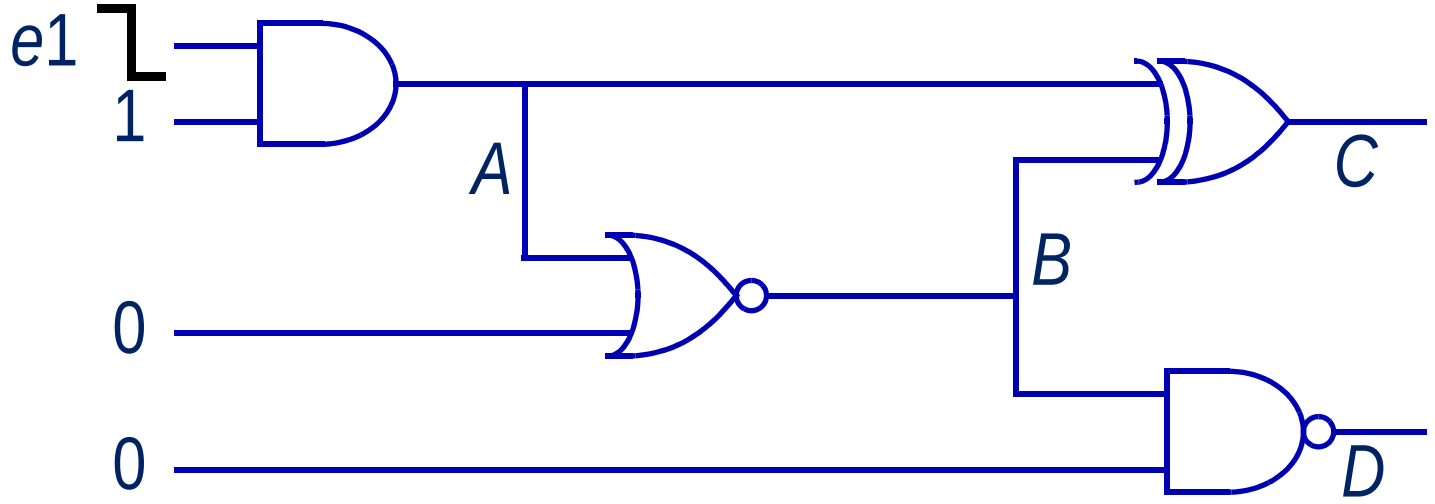
The Language of Algorithms

"Trace through graph until no more outgoing edge exists
or until the next vertex has already been visited,
backtrack and insert the visited vertex in an ordered evaluation list."

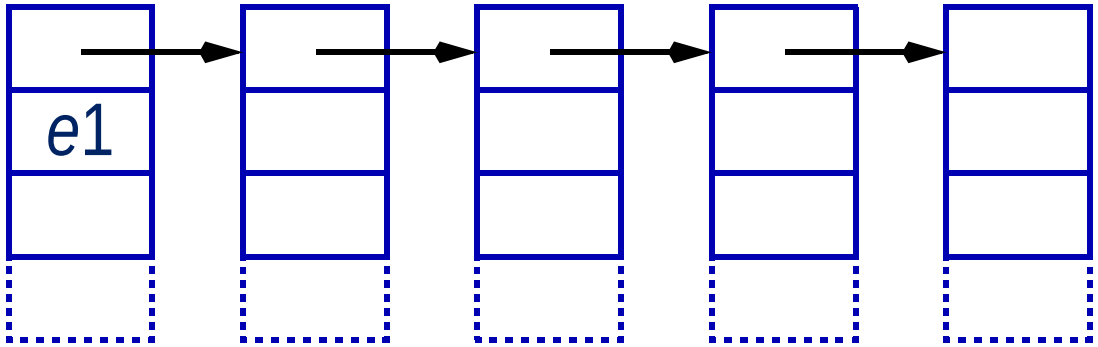
```
Input: G(V,E), Output: List of ordered nodes

TopologicalSort (G)
{while (node v in V is not marked visited) VISIT(v)}

VISIT(v) {
mark v visited;
for each (u taken from the fanout of v)
   if (u is not marked visited) VISIT(u);
insert u in front of List; }
```

Event-driven logic simulation

Event-Driven (Gate-Level) Simulation 1(5)

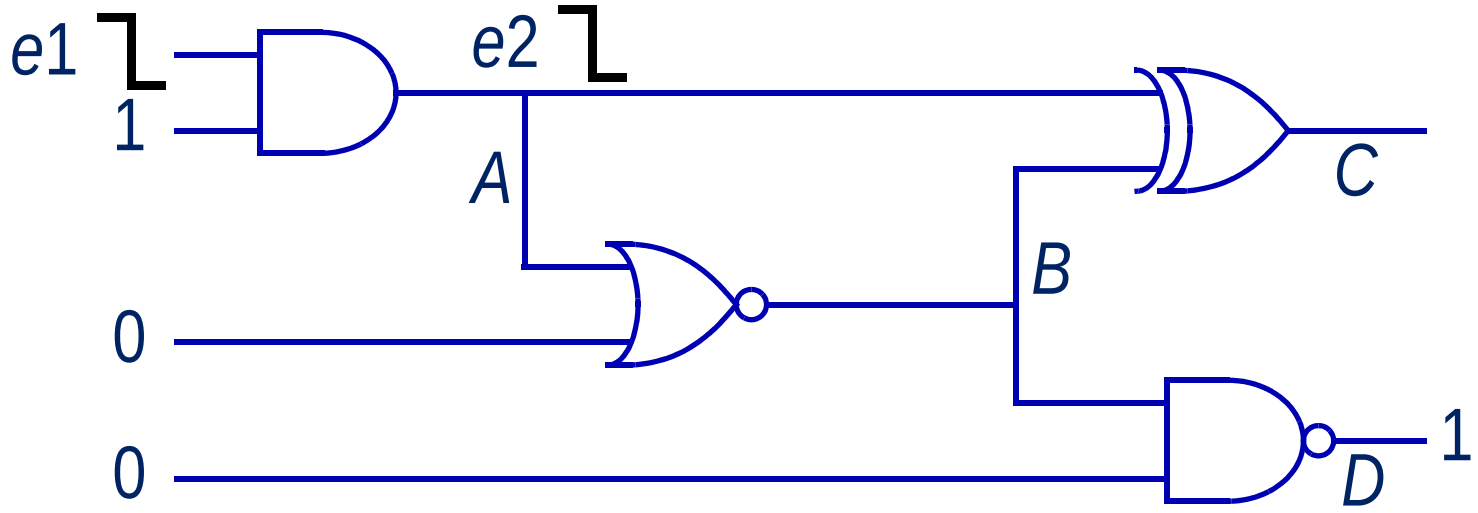


Event queue

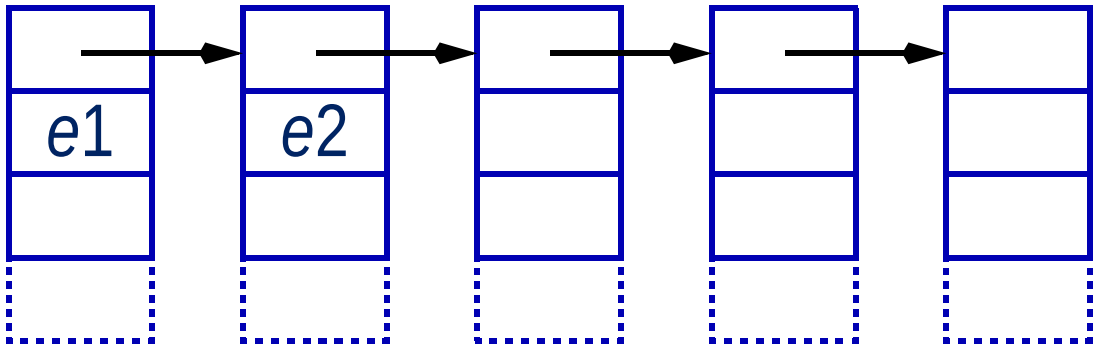


source: Lam

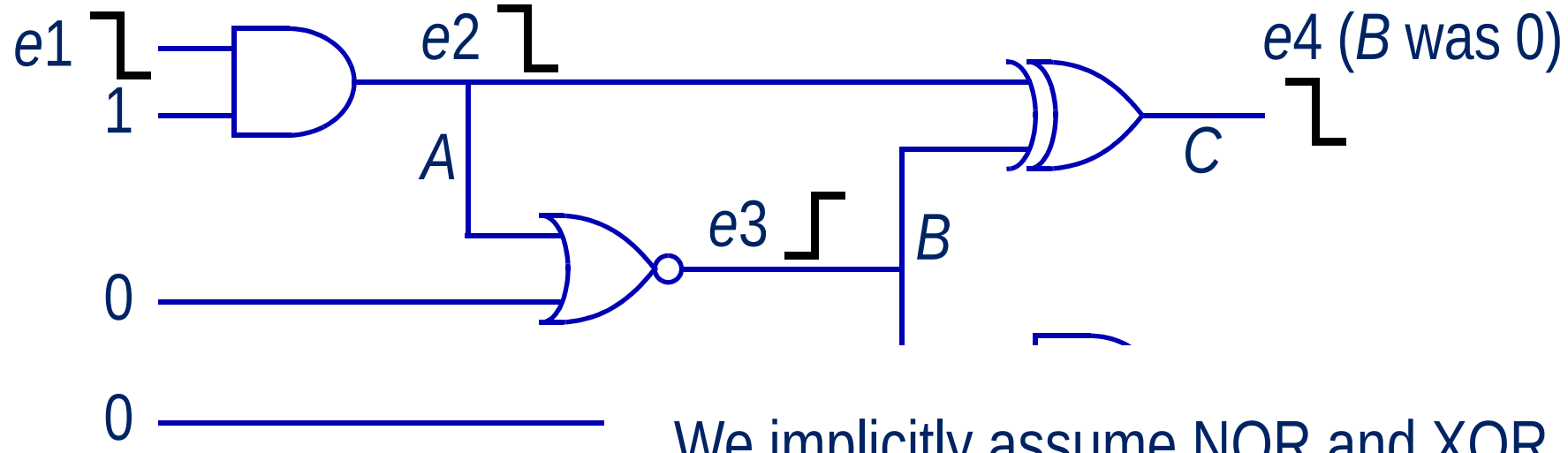
Event-Driven (Gate-Level) Simulation 2(5)



Event queue

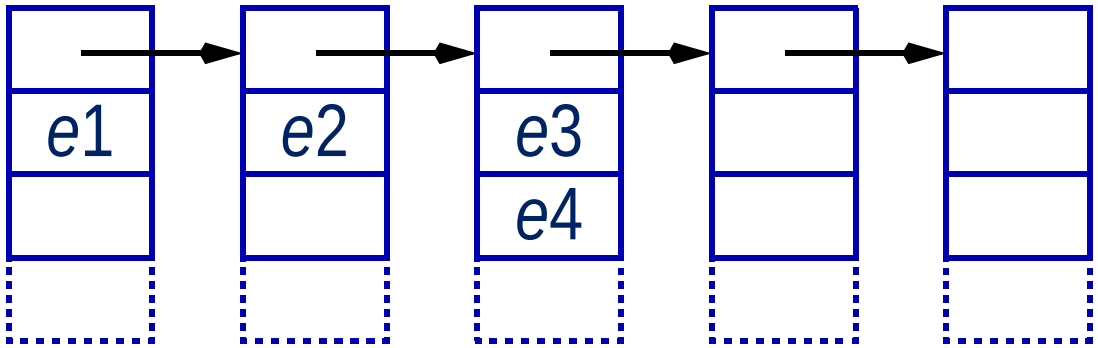


Event-Driven (Gate-Level) Simulation 3(5)

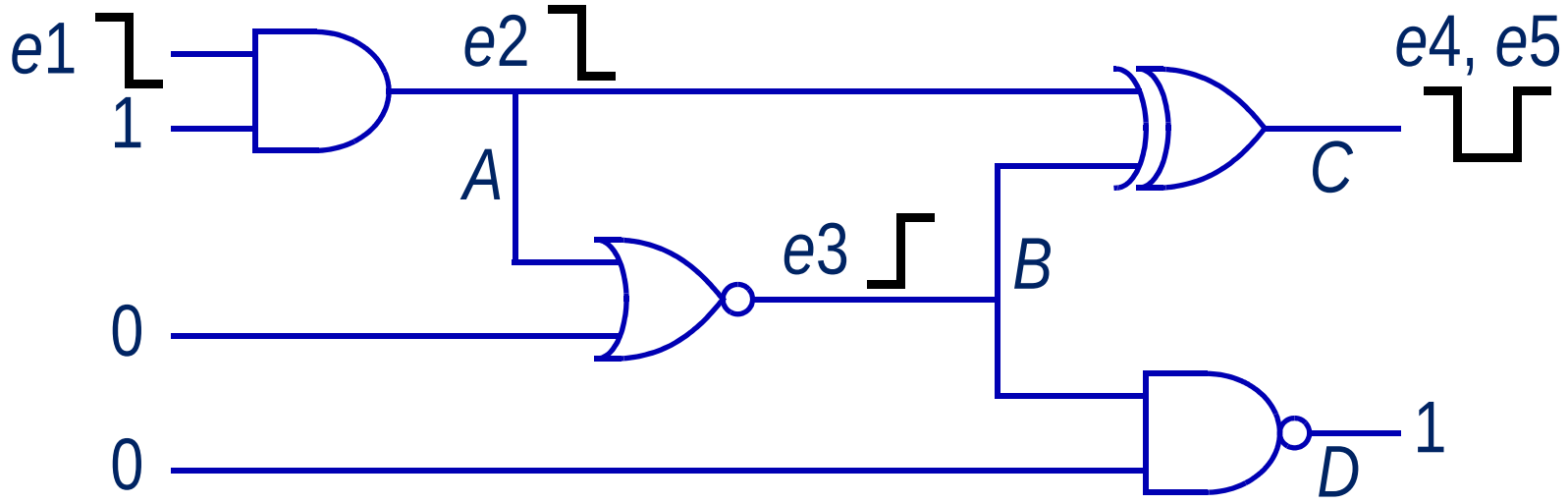


We implicitly assume NOR and XOR have identical delays - a unit delay

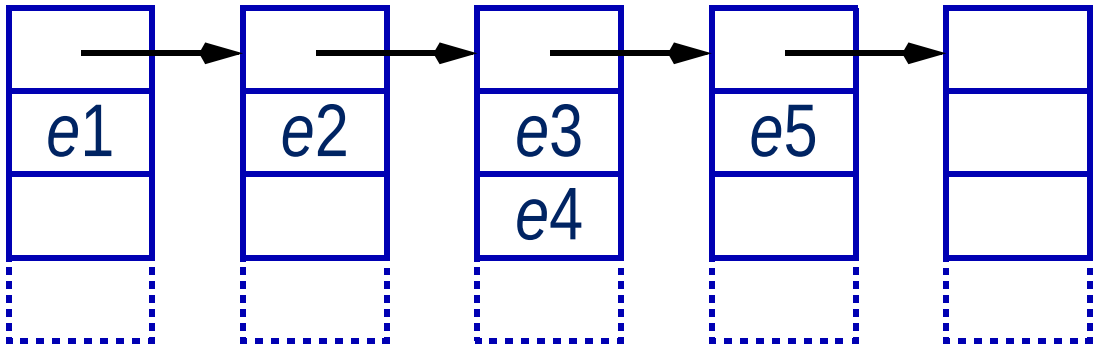
Event queue



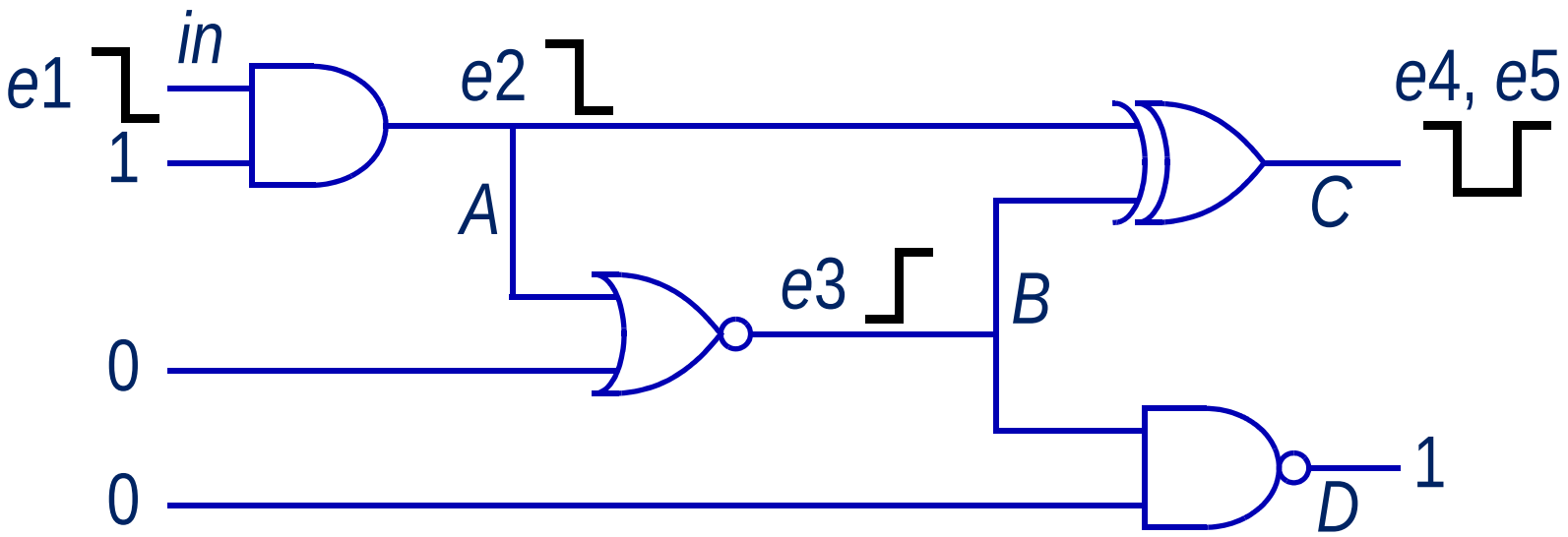
Event-Driven (Gate-Level) Simulation 4(5)

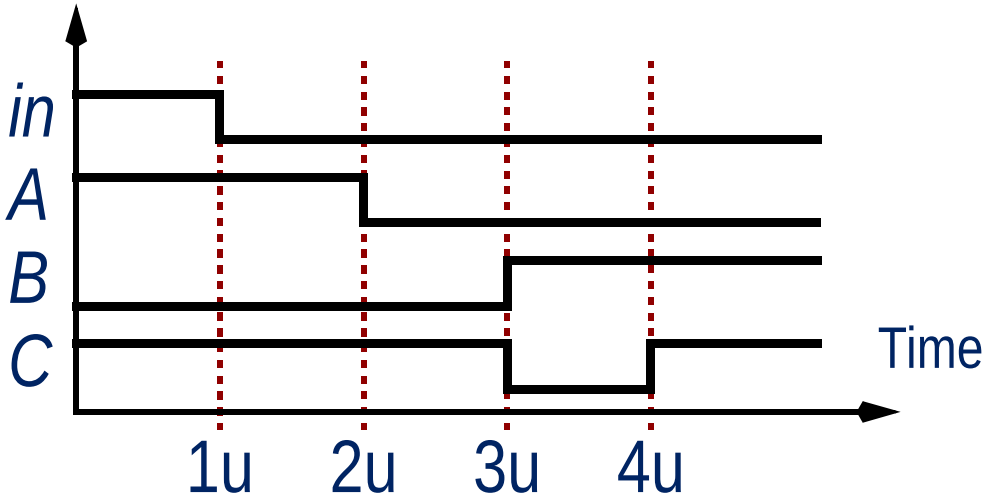


Event queue



Event-Driven (Gate-Level) Simulation 5(5)





Functional vs RTL Signoff

- Functional signoff is not hardware centric (VHDL code in lab 1).
- RTL signoff is hardware centric (synthesized netlist in lab 2) and allows verification of
  - power up.
  - shut down.
  - configuration modes.
  - reset.
  - low power.

Functional Verification: Conclusion

- Electronic systems grow more complex $\Rightarrow$
  we have to take functional verification very seriously,
  to the point where designated verification engineers are appointed.
- Problem of university education:
  Limited complexity of engineering challenges $\Rightarrow$
  verification is relatively easy.
- Knowledge of simulation principles helps
  handle the inevitable EDA tools.
- Make use of testbenches ...
  a topic of the next lecture; see next slide!

Verification Task Breakdown