Design Verification

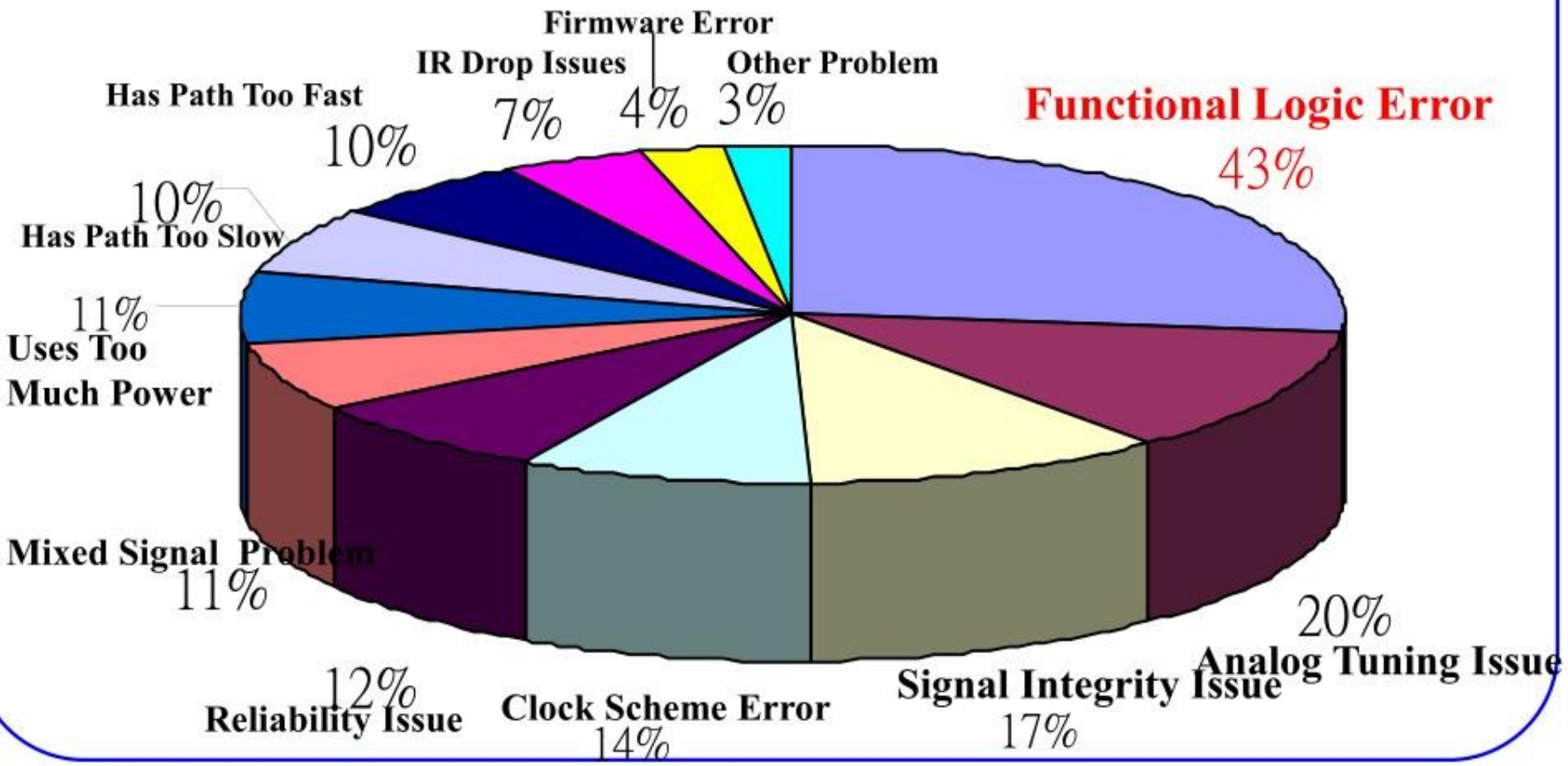
Verification Costs

- ~ 70% of project development cycle: design verification
  - Every approach to reduce this time has a considerable influence on economic success of a product.

- Not unusual for ~complex chip to go through multiple tape-outs before release.

Problems found on 1st spin ICs/ASICs

- Overall 61% of New ICs/ASICs Require At Least One Re-Spin
- %43 due to functional error

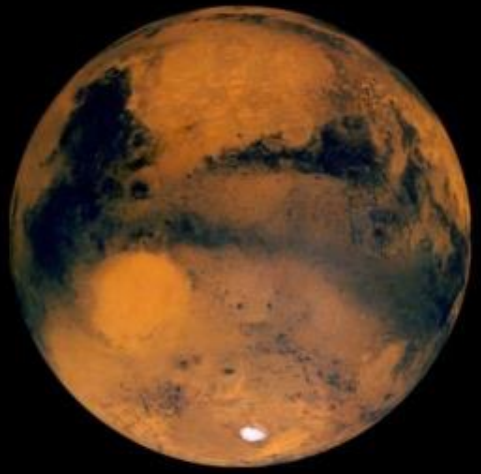


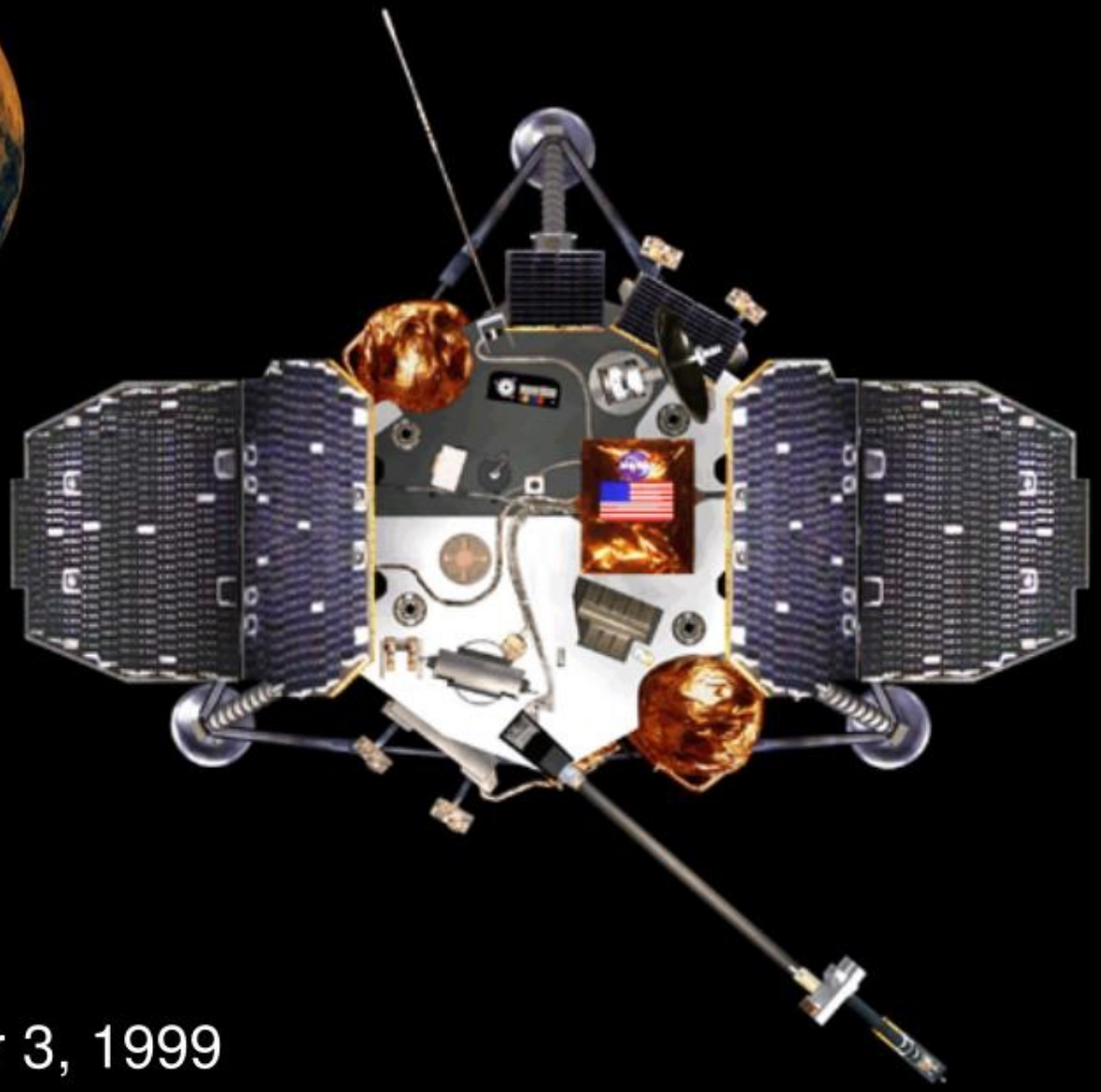
Source: DeepChip, Aart de Geus, Chairman & CEO of Synopsys used it during Wed, 10 Sep 2003 Boston SNUG keynote address

Importance of Verification

- **Verification is a bottleneck in the design process:**
  - High cost of design debug
    - designers (sometimes verification engineers = 2 * design engineers)
    - time-to-market
  - High cost of faulty designs (loss of life, product recall)


French Guyana, June 4, 1996
$800 million software failure





Mars, December 3, 1999
Crashed due to uninitialized variable

$4 billion development effort

> 50% system integration & validation cost


400 horses
100 microprocessors
M NH 2519

Types of Errors

1. **Error in specification**
   - a) Unspecified functionality,
   - b) Conflicting requirements,
   - c) Unrealized features.
   - The Only Solution: Redundancy
     - because specification is at the top of abstraction hierarchy
     - → No reference model to check

2. **Error in implementation**

Types of Errors

2. **Error in implementation:**
   - e.g. human error in interpreting design functionality.
- **Solution 1:**
   - Use software program to synthesize an implementation directly from specification.
     - Eliminates most human errors,
     - Errors remain from
       1. Bugs in synthesis program,
       2. Usage errors of synthesis program.

Error in implementation

- **Solution 1 (cont'd):**
  - Use of synthesis tools are limited because:
    1. Many specifications are in conversational language
       - Automatic synthesis not possible yet,
       - No high level formal language specifies both functionality and timing (or other requirements) yet.
    2. Even if specifications are written in precise mathematical language, few software can produce implementations that meet all requirements.

Error in implementation

- **Solution 2 (more widely used): Uncover through redundancy:**
  - Implement two or more times using different approaches and compare.
  - In theory: the more times and more different ways: → Higher confidence.
  - In practice: rarely > 2, because of
    - cost,
    - time,
    - more errors can be introduced in each alternative.
  - To make the two approaches different:
    - Use different languages:
      - Specification Languages: VHDL, Verilog, SystemC
      - Verification Languages: Vera, C/C++, e (no need to be synthesizable).

Error in implementation

- **In solution 2:**
  - Sometimes comparison is hard:
    - E.g. Compare two networks with arrival packets that may be out of order.
    - Solution: Sort the packets in a predefined way.
  - Solution 2 is double-edge sword:
    - Verification engineers have to debug more errors (in design and verification language).
      - → high cost.
    - Verification engineers may involve with the differences inherent to the languages (e.g parallelism in C)
    - → Hint: Be aware of these differences.

Functional Verification

- **Two Categories:**
  1. Simulation-Based Verification,
  2. Formal Method-Based Verification

  - Difference in the existence or absence of input vectors.

Functional Verification

- **Simulation** - dynamic
  - + The most widely used
  - + The only solution for system verification
  - – Develop testbench and tests
  - – Hard to understand coverage/completeness
- **Formal** - static
  - + Exhaustive – good for module/block
  - – Limited capacity
  - – Limited applicability

Verification and Test

- **Verification**
  - Ensuring that the design matches the designer's intent/specification
  - Detection of design errors accidentally made by the designer(s)
  - Usually includes the debugging task as well
- **(Manufacturing) Test**
  - Detection of physical defects in the implementation
  - Defects appear as a result of manufacturing or of wear mechanisms
  - Tests the fabricated product, not the design

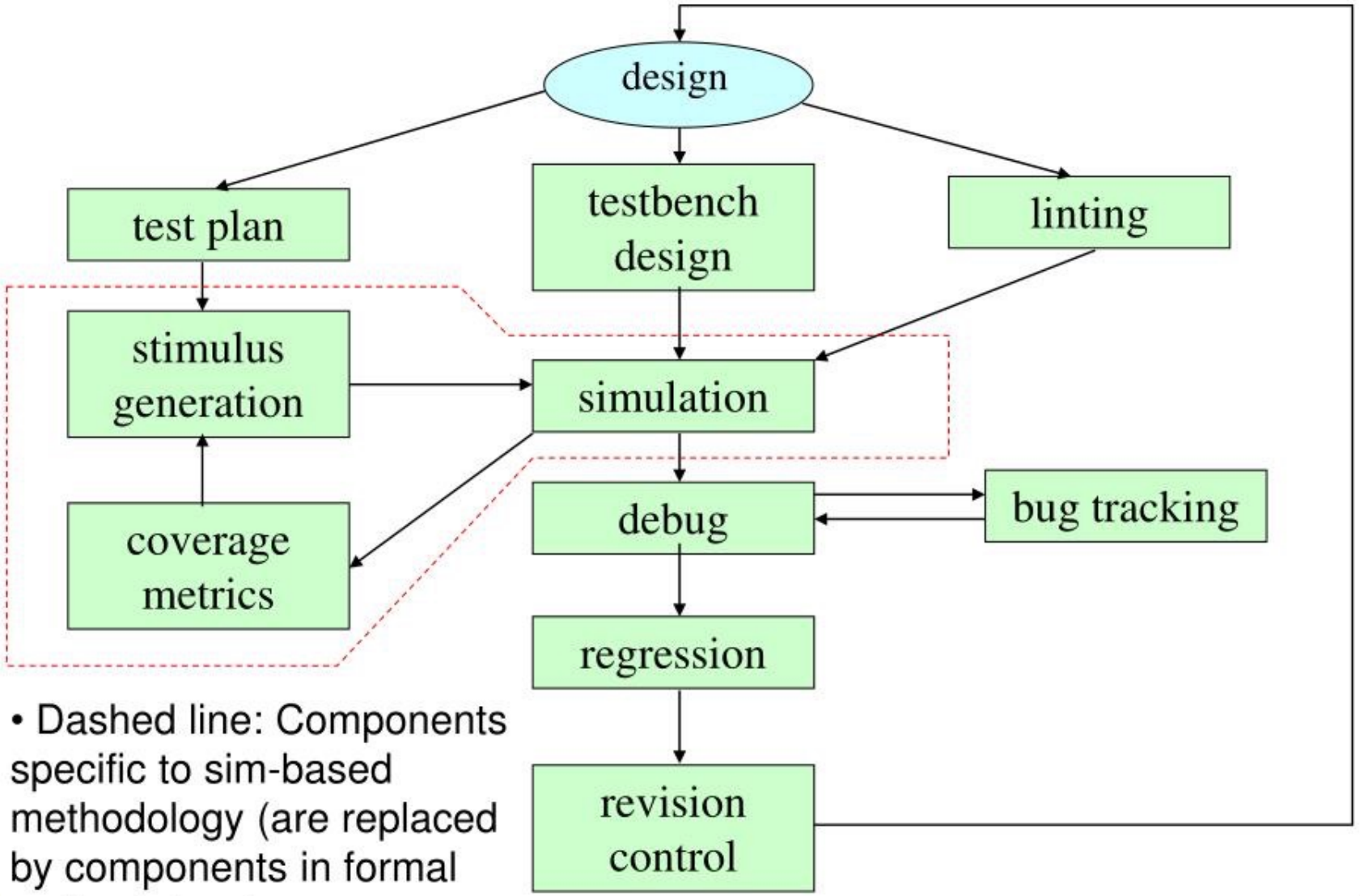
Functional Verification

- **Simulation-Based Verification**
  - Directed
  - Random
- **Formal Verification**
  - Model checking
  - Equivalence checking
  - Theorem proving
- **Assertion-Based Verification**
  - Dynamic formal verification

Verification Metrics

- **Quality of a simulation on a design:**
  - **Functional Coverage**: % of functionality verified
  - **Code Coverage**: % of code simulated
    - % of statements,
    - % of branches taken.

Flow of Simulation-Based Verification



- Dashed line: Components specific to sim-based methodology (are replaced by components in formal methodology)

Flow of Simulation-Based Verification

- **Linter:**
  - Checks for static errors or potential errors and coding style guideline violations.
    - Static errors: Errors that do not require input vectors.
    - E.g.
      - A bus without driver,
      - mismatch of port width in module definition and instantiation.
      - dangling input of a gate.

Flow of Simulation-Based Verification

- **Bug Tracking System:**
  - When a bug found, it is logged into a bug tracking system
  - It sends a notification to the designer.
  - Four Stages:
    1. Opened:
       - Bug is opened when it is filed.
    2. Verified:
       - when designer confirms that it is a bug.
    3. Fixed:
       - when it is destroyed.
    4. Closed:
       - when everything works with the fix.
  - BTS allows the project manager to prioritize bugs and estimate project progress better.

Flow of Simulation-Based Verification

- **Regression:**
  - Return to the normal state.
    - New features + bug fixes are made available to the team.

Flow of Simulation-Based Verification

- **Revision Control:**
  - When multiple users accessing the same data, data loss may result.
    - e.g. trying to write to the same file simultaneously.
  - Prevent multiple writes.

Simulation-Based Verification
- classified by input pattern

- **Deterministic/direct testing**
  - traditional, simple test input
- **Random pattern generation**
  - within intelligent testbench automation
- **Dynamic (semi-formal)**
  - with assertion check
- **Pattern generated from ATPG (Auto. Test Pattern Generation)**
  - academic only

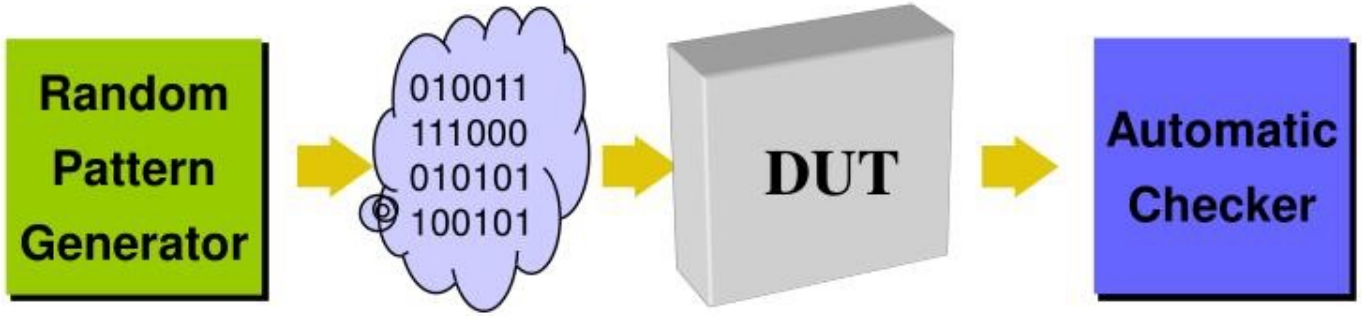
Deterministic testing

- Most common test methodology used today
- Developed manually and normally correspond directly to the functional test plan.
- Specific sequences that cause the device to enter extreme operational modes.
- Drawbacks:
  - Time-consuming, manual programming effort.
  - Difficult to think of all required verification cases.
  - maintenance effort is high.

=> use random…

Automated Verification

- Automatic random generation of inputs
- Automatic checking of the results (a must)



- Saves manual effort of choosing values
- Improves test coverage

Functional verification approaches
- Black Box , White Box ?

- **Black box:**
  - Without any idea of internal implementation of design
  - Without direct access to the internal signals
- **White box:**
  - Has full controllability and visibility on internal structure
  - E.g. Observe internal registers in a microprocessor
- **Gray box:**
  - Compromise in between

Black Box Testing

- Testing without knowledge of the internals of the design.
- Only know system inputs/outputs
- Testing the functionality without considering implementation
- Inherently top-down

Black Box Testing Examples

- **Examples:**
  - Test a multiplier by supplying random numbers to multiply
  - Test a braking system by hitting the brakes at different speeds
  - Test a traffic light controller by simulating pre-recorded traffic patterns

Black Box, White Box?

- **Recommendation by experience:**

- **Start with a black box approach**
  - Good for early testing with behavioral models and architectural exploration
  - May develop the testbench scheme earlier
  - More reusable (flexible) in the future
- **Later, add more white-box pieces**
  - Require to involve the designers!
- **Extra tests using "white box" approach is necessary**
  - "black box" test spec are insufficient to reach the structural coverage target
  - looking at the coverage holes and specifying specific transaction sequences that need to be performed in order to hit the previously uncovered holes.
- **Gray box is usually suggested**

Coding for Verification

Verilog

- **Continuous Assignment:**
  - Propagates inputs to outputs whenever inputs change (concurrent assignment).

```
assign v = x + y + z;
```

- **Procedural Block:**

```
always@(posedge c or d)
   begin
      v = c + d + e;
      w = m - n;
   …
   end;
```

Verilog

- **Non-Blocking Assignment:**
  - As VHDL signal assignment.

```
always@(posedge clk)
   begin
      x <= #20 1'b0;
   …
   end;
```

- **Blocking Assignment:**
  - Blocks the execution, until it is assigned.

```
always@(posedge clk)
   begin
      a = #20 b;
   …
   end;
```

Coding for Verification

- The best way to reduce bugs in a design is to minimize the opportunities that bugs can be introduced (i.e. design with verification in mind)
  - Once this is achieved, the "next" step may be to maximize the simulation speed.
    - because the benefits of minimizing bug-introducing opportunities far outweigh the potential lost code efficiency.

Coding Guidelines

- **Coding Guidelines Categories:**

1. Functional correctness rules:
   - State explicitly the coding intent → eliminate the hidden side effects of some constructs.
     - E.g. Matching width for all operands
2. Timing correctness rules:
   - Examine code that may cause race, glitches and other timing problems.
3. Portability and maintainability rules:
   - Enforce partition and structure, naming convention, comments, file organization, ….
4. Simulation performance rules:
   - Recommend styles that yield faster simulation.
5. Tool compatibility rules:
   - Ensure the code is acceptable by other tools (e.g. synthesizability, debugability)

Coding Guidelines

- **Linter:**
  - Used to check syntactical errors,
  - But as people realized that many design errors can be checked statically, their role has expanded to coding styles
    - Need no input vectors
    - VHDL linters are more powerful (Verilog allows more freedom to designer).

Functional Correctness

- **Two categories:**
  - Syntactical checks:
    - Can be analyzed locally.
      - E.g. unequal width operands.
  - Structural checks:
    - Must be done globally (i.e. the structure of circuit must be checked)
      - E.g. combinational loops.

Syntactical Checks

- **Common syntactical rules:**
  1. Operands of unequal width:
     - Most HDLs pad the shorter operand with zeros.
  2. Implicitly embedded sequential state:
     - In the example, only two (out of 4 possible) values are specified
       - → code has memory (although looks like a MUX)

```
case (X)        // a 2-bit variable
   2'b00: Q = 1'b1;
   2'b11: Q = 1'b0;
endcase;
```

     - Example 2: in a clocked process, use a variable value before it is assigned.
     - Example 3: read a variable value which is not in the sensitivity list.

Syntactical Checks

- **Common syntactical rules (continued):**

3. Overlapping condition cases:

- Synthesis tool may produce a priority encoder (with higher priority to MSB)
- But designer may not mean that (e.g. wants 0 to be assigned if S = 101)

```
case (S)
   3'b1??: Q = 1'b1;
   3'b??1: Q = 1'b0;
endcase;
```

- → Avoid overlapping conditions.

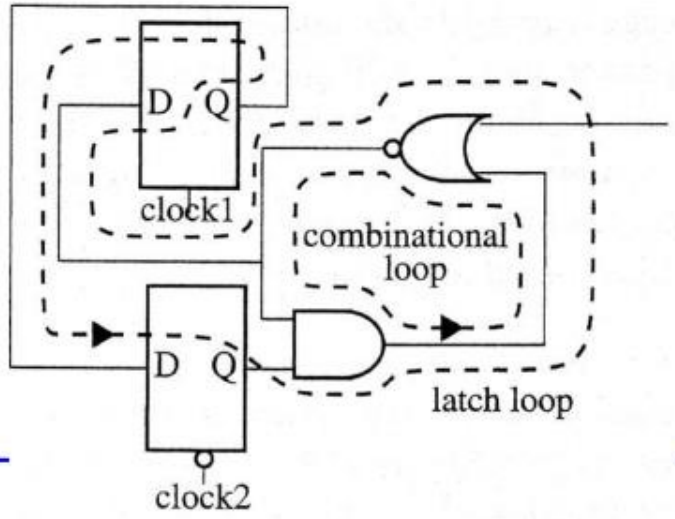
Syntactical Checks

- **Common syntactical rules (continued):**

  4. Connection rules:
     - Use explicit connections (i.e. named association).

Structural Checks

- **Common structural rules:**
  1. Avoid combination loops
  2. Avoid latch loops (Fig 2.6)
     - Harder to discover because needs to determine whether all latches in the loop can become transparent at the same time:
       - Computationally intensive.
     - → designer assist the checker by telling him/her about the phases of latch clocks.



Structural Checks

- **Common structural rules (continued):**

3. Bus Operation:

- Drivers must be checked for mutual exclusion of their enable signals.
  - FV method: computationally expensive
  - Simulation: partial validation (data dependent)

4. FF and Latch Configuration:

- Active-high latches should not drive active-high latches
- Positive-edge trigger FFs should not drive active-high latches
  - Because the driven latches simply pass whatever the driving latches/FFs

5. Sequential Element Reset:

- All sequential elements should be able to be reset or driven to a known state.

Timing Correctness

- Contrary to common belief, timing correctness, to a certain degree, can be verified statically in the RTL without physical design parameters (e.g. gate/interconnect delays).

Timing Correctness

- **Race Problem:**
  - When several operations operate on the same entity (e.g. variable or net) at the same time.
  - Extremely hard to trace from I/O behavior (because non-deterministic)
  - Example 1:

```
always @(posedge clock)
  x = 1'b1;

always @(posedge clock)
  x = 1'b0;
```

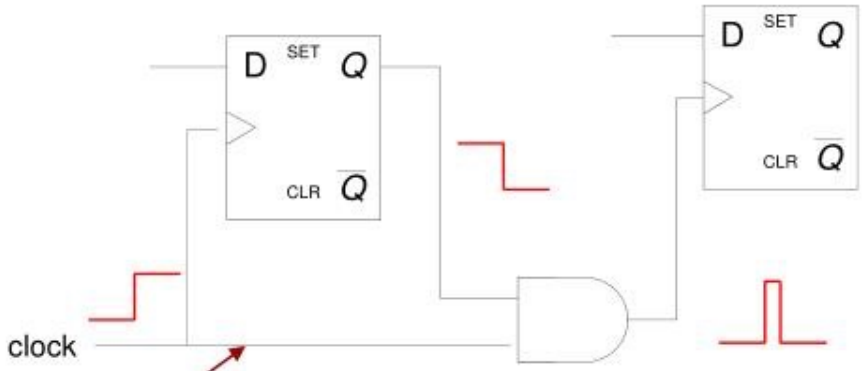
  - Example 2 (event counting):

```
always @(x or y)
   event_number = event_number + 1;
```

    - If x and y have transitions at the same time, increment by one or two?
      - Simulator-dependent

Timing Correctness

- **Clock Gaing:**
  - Can produce glitches on clock tree
    - → trigger latches and FFs falsely.



  - Solution 1:
    - Add delay to here
  - Problems:
    1. Hard to control the delay (depends on physical design).
    2. Zero-delay simulations still produce the glitch.
  - Better Solution:
    - Use OR as the gating device:
      - → clock goes to high first and stabilizes the output of OR.
    - If the gating FF is negative-edge triggered, should use AND.

Timing Correctness

- **Time Zero Glitches:**
  - Question: At the start of simulation, when a clock is assigned to 1, should the transition from X (unknown) to 1 be counted as a transition?
    - Answer: depends on the simulator.
  - Solution: Delay assignment to clock to avoid transitions at time zero.

Simulation Performance

1. **Higher Level of Abstraction:**
   - Behavioral level is much faster than structural gate level.
2. **Simulator Recognizable Components:**
   - Many simulators attempt to recognize some standard components (e.g. FFs, latches, memory arrays) to make internal optimization for performance.
   - Depends on the simulator
     - → Refer to user manual to conform to the recommended style.
   - If no such recommendation is provided:
     - Code in a style as simple and as close to the "common" style as possible:

Simulation Performance

- Common styles:

```
// positive-edge-triggered DFF
always @(posedge clock)
 q <= d;
```

```
// DFF with asynch. Active high reset.
always @(posedge clock or posedge reset)
 if (reset)
      q <= 1'b0;
  else
      q <= d;
```

```
// Active high latch
always @(clock or data)
 if (reset)
       q <= data;
```

Simulation Performance

3. **FSMs:**
   - Particular styles are preferred by the simulator.
   - General rule of thumb:
     - Separate as much combinational logic from the states as possible.
4. **Vector vs. Scalar:**
   - Whenever possible, use the entire bus as an operand (instead of bits)
     - Because simulators take more internal operations to access bits or ranges of bits.
   - Convert bit operations into bus operations by:
     - Concatenation, reduction, and masking.

Simulation Performance

- Convert scalar operations to vectored operations:
  - Scalar Operation:

```
assign bus[15:0]] = a & b;
assign bus[31:16] = x | y;
```

  - Vectored Operation:

```
assign bus = {x | y, a & b}; // {} is concat
```

  - Scalar Operation:

```
assign output = (bus[1] & bus[2]) | (bus[3] ^ bus[0]);
```

  - Vectored Operation:

```
assign output = (&(4'b1001 | bus)) | (^(4'b1001 & bus))
```

Simulation Performance

- Useful application: Error Correction Code
  - Scalar Operation:

```
C = A[0] ^ A[1] ^ A[2] ^ A[9] ^ A[10] ^ A[15];
```

  - Vectored Operation:

```
C = ^(A & 16'b1000011000000111);
```

- Especially effective if used in loops.

```
FOR (i=0; i<=31; i=i+1)
   assign A[i] = B[i] ^ C[i];
```

  - Vectored Operation:

```
assign A = B ^ C;
```

Simulation Performance

- Vectorization in instantiation of arrays of gates
  - Simulator recognizes the inherent bus structure.

```
FlipFlop FFs [31:0] (.Q(output), .D(input), .clock(clock));
```

  - Generates 32 FFs with inputs that are connected to bus D, output bus Q and a clock.

Simulation Performance

5. **Minimize interface to other simulation systems:**
   - Avoid displaying or dumping too much data on the host during run-time
     - → speedup by a factor of 10 or more.
   - → Display or dump code should be able to turn on/off
   - Should be turned on only during debug mode.

Simulation Performance

6. Code Profiler:

- A program attached to simulator which collects about the distribution of simulation time
- → User determines the bottlenecks of the simulation:
- E.g. the total time spent on the simulation of
  - a particular instance,
  - a particular bock (such as `always`),
  - a particular function,
  - ….
- ModelSim: Performance Analyzer

Portability and Maintainability

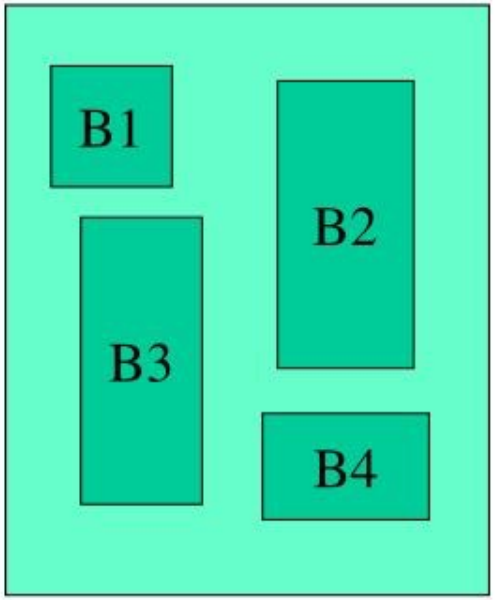
- A design team must have a uniform style guideline so that the code is easy to maintain and reuse .
  - Note: Code may range from thousands to millions.

- **Top down approach:**
  - Project-wide file structure,
  - Common code resources,
  - Individual file format.

Portability and Maintainability

1. Project Code Layout

- Correspondence between file structure and top-down functional blocks:



Source Directory:

- Folder B1
- Folder B2
- Folder B3
- Folder B4

- Each file should contain only one module
  - Except for cell library.

Portability and Maintainability

- Top-level module should consist only of module instantiation and interconnects
  - No logic
  - Reason: top module represents a partition of the design → all low-level logic should belong to one of the functional blocks.
- Minimize the number of models of a module (behavioral for fast simulation, structural for easy synthesis, …):
  - If there are more than one model coexists in a file, equivalence must be ensured.
  - Maintaining multiple-models and their equivalence has a high cost later in the project.
  - Other models should exist only in the cell library or macro library (not in the RTL files).

Portability and Maintainability

- Hierarchical path access should be permitted only in testbenches (all accesses must be done through ports)
  - Hierarchical path access enables reading/writing to a signal directly over a design hierarchy w/o going through ports.
  - Sometimes is necessary in testbenches because the signal monitored may not be accessible through ports.

Portability and Maintainability

2. Centralized Resources:

- A project should have a minimum set of common sources:
  1. a cell library and
  2. a set of macro definitions

- All designers must instantiate gates from the project library (instead of creating their own)
- All designers must derive memory arrays from the macro library.
- No global variables are allowed.

Portability and Maintainability

3. RTL Design File Format:

- Each file should contain only one module
- The filename should match the module name.
- The beginning is about the designer and the design:
  - Name,
  - Date of creation,
  - A description of the module,
  - Revision history.
- Next: header file inclusion.
- In port declaration, brief description about the ports,
- Large blocks should have comments about their functionality and why they are coded as such.
  - Comments are not simply (P)English translation of the VHDL/Verilog code but should contain the “intention” and “operation” of the code.

Portability and Maintainability

- Each begin-and-end pair should have markers to identify the pair.

```
begin  // start of search
...
  begin // found it
  ...
  end  // end of found it
...
end  // end of search
```

- Naming convention:
  - c_buswidth, v_index, s_sig1, ….

Portability and Maintainability

- Each begin-and-end pair should have markers to identify the pair.

```
begin  // start of search
...
  begin // found it
  ...
  end  // end of found it
...
end  // end of search
```

- Naming convention:
  - c_buswidth, v_index, s_sig1, ….

References

1. D. Lam, “Hardware Design Verification: Simulation and Formal Method-Based Approaches,” Prentice-Hall, 2005.

2. D. Gajski and S. Abdi, "System Debugging and Verification: A New Challenge," Verify 2003, Tokyo, Japan, November 20, 2003.

3. Slides from “High Level Design Verification - Current Verification Techniques & Tools” by Chia-Yuan Uang