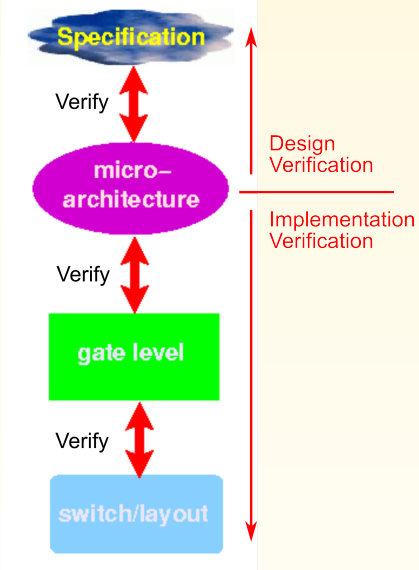
Design Verification

Jacob Abraham

EE 382M.8 VLSI-II

Spring 2015

Verification in the Design Cycle



- **Implementation Verification**: For all feasible inputs the behavior of the circuit is consistent with the behavior required by the specification
- **Design Verification**: For all feasible inputs the design has a number of properties required by the specification

Current verification techniques focused on functional verification

Analyzing Complex Designs

Need to (implicitly) search a very large state space

- Find bugs in a design – verification process
- Generate tests for faults in a manufactured chip

Basic algorithms for analyzing even combinational blocks (SAT, ATPG) are NP-complete

Approaches to deal with real designs

- Exploit hierarchy in the design
- Develop abstractions for parts of a design

Cost of a new mask set can be on the order of $1+ Million for a large chip

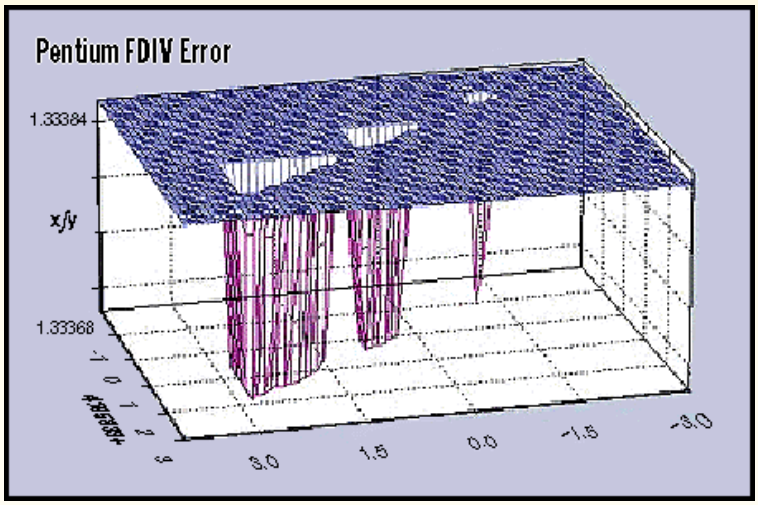
- Cannot afford mistakes
- **Want working "first silicon"**

Many Aspects of Verification

- Verifying the **functional correctness** of the design
- **Performance** verification
  - Architecture level (number of clocks to perform a function)
- **Timing verification**
  - Circuit level (how fast can we clock?)
- Verifying **power consumption**
- Verifying **signal integrity and variation tolerance**
- Checking **correct implementation** of specifications at each level

The (In)Famous Pentium FDIV Problem

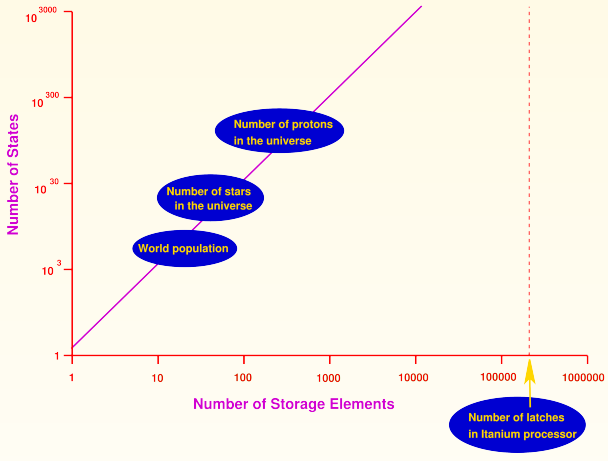
Graph of x, y, x/y in a small region by Larry Hoyle



State-Space Explosion

May need to check a very large number of states to uncover a bug

Problem: the number of protons in the universe is around 10^{80}, which is less than the number of states for a system with 300 storage elements!



What is a "Bug"?

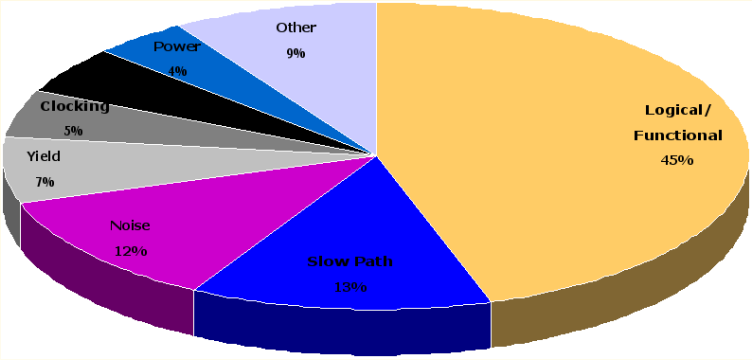
Design does not match the specification

- One problem: complete (and consistent) specifications may not exist for many products
- For example, the difficulty in designing an X86 compatible chip is not in implementing the X-86 instruction set architecture, but in matching the behavior with Intel chips

Something which the customer will complain about

- Marketing: **"It's not a bug, it's a feature"**

Design Flaws



About half of the flaws are functional flaws

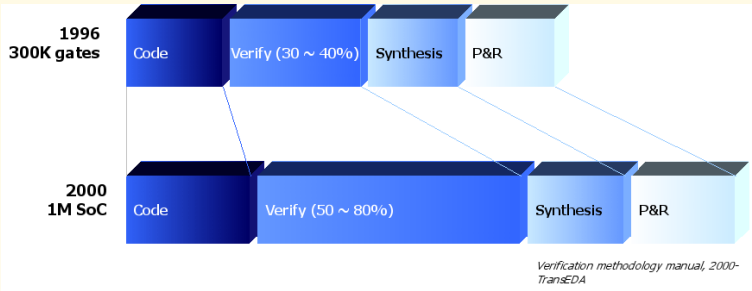
Need verification methods to find and fix logical and functional flaws

Design Bug Distribution in Pentium 4

Type of Bug	%
"Goof"	12.7
Miscommunication	11.4
Microarchitecture	9.3
Logic/Microcode Changes	9.3
Corner Cases	8.0
Power Down	5.7
Documentation	4.4
Complexity	3.9
Initialization	3.4
Incorrect RTL Assertions	2.8
Design Mistake	2.6

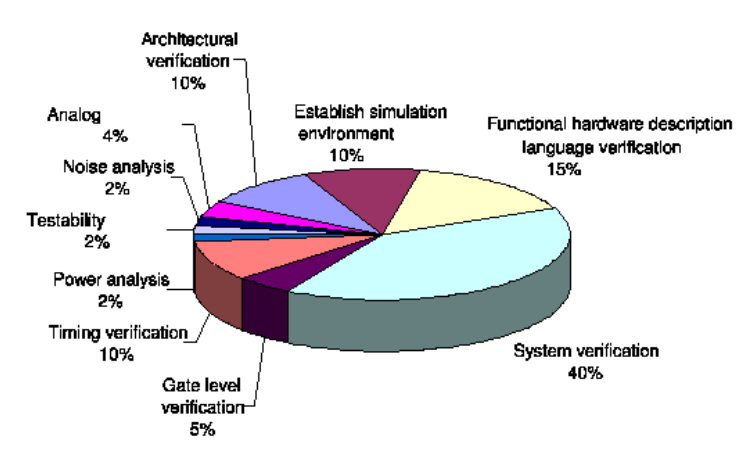
- Source: EE Times, July 4, 2001
- 42 Million Transistors
- High-level description: 1+ million lines of RTL
- 100 high-level bugs found through formal verification

Design Effort



Verification is becoming an increasing part of the design effort

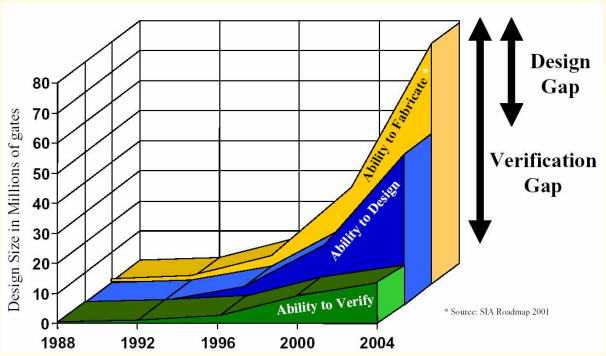
Verification Effort



Source: 1999 ITRS

Over 50% – 80% of design effort is in verification

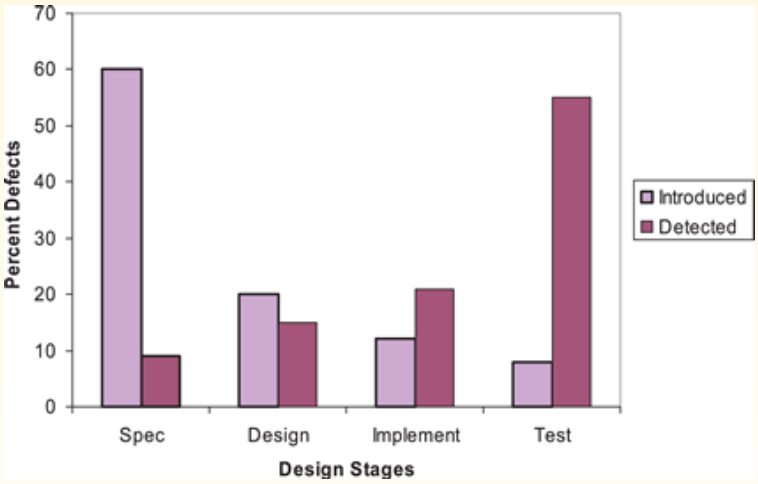
Design and Verification Gap



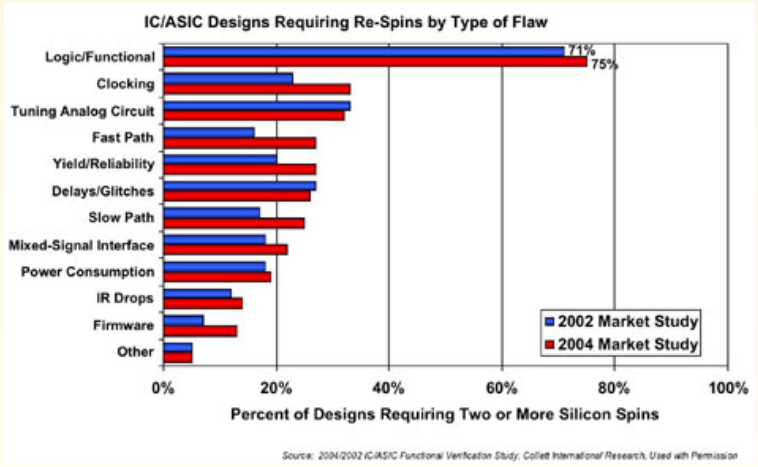
Blame it on Moore's Law

- The number of transistors on a chip is increasing every year (approx. 58%)
- Design productivity, facilitated by EDA tool improvements, grows only about 21% per year
- These numbers have held constant for two decades

"Bug" Introduction and Detection



Re-Spins because of Functional Flaws



Tom Fitzpatrick
EE Times, December 5, 2005

Evaluating the Complete Design

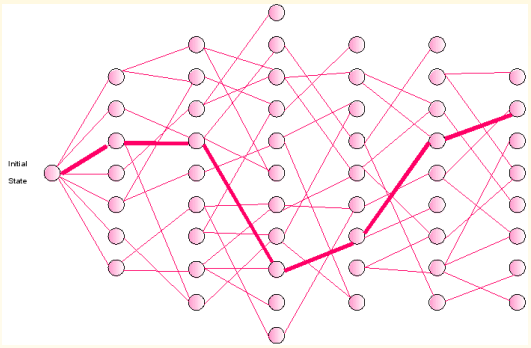
- Is there a verification technique which can be applied to the entire chip?
- Only one approach which scales with the design: **Simulation**
- Most common technique now used in industry
- **Cycle-based simulation** can exercise the design for millions of cycles
  - Unfortunately, the question of when to stop simulation is open
  - No good measures of **coverage**
- **Emulation**
  - Used to verify the first Pentium (windows booted on FPGA system)
  - Developing another accurate model is an issue

When are we Done Simulating?

When do you tape out?

- Motorola criteria (EE Times, July 4, 2001)
- 40 billion random cycles without finding a bug
- Directed tests in verification plan are completed
- Source code and/or functional coverage goals are met
- Diminishing bug rate is observed
- A certain date on the calendar is reached

What is the Fundamental Problem with Simulation?



Traversing States in the Design

- Simulation traverses paths in the state graph
- A bug may be associated with a **specific transition** from a **specific state**
- Cannot guarantee that we will exercise that transition

Coverage-Driven Verification

Attempt to Verify that the Design Meets Verification Goals

- Define all the verification goals up front in terms of "functional coverage points"
  - Each bit of functionality required to be tested in the design is described in terms of events, values and combinations
- Functional coverage points are coded into the HVL (Hardware Verification Language) environment (e.g., Specman 'e')
  - Simulation runs can be measured for the coverage they accomplish
- Focus on tests that will accomplishing the coverage ("coverage driven testing")
  - Then fix bugs, release constraints, improve the test environment
  - Measurable metric for verification effort

Assertions

- Assertions capture knowledge about how a design should behave
- Used in coverage-based verification techniques
- Assertions help to increase observability into a design, as well as the controllability of a design
- Each assertion specifies
  - legal behavior of some part of the design, or
  - illegal behavior of part of the design
- Examples of assertions (will be specified in a formal language)
  - The fifo should not overflow
  - Some set of signals should be "one-hot"
  - If a signal occurs, then ...

Simulation Monitors and Assertions

```
assert_never underflow ( clk, reset_n,
    (q_valid==1'b1) && (q_underflow==1'b1));
```


```
module assert_never (clk, reset_
 input clk, reset_n, test_expr;
 parameter severity_level = 0;
 parameter msg="ASSERT NEVER VIOLATION";
// ASSERT: PRAGMA HERE
 //synopsys translate_off
 `ifdef ASSERT_ON
  integer error_count;
  initial error_count = 0;
  always @(posedge clk) begin
   `ifdef ASSERT_GLOBAL_RESET
    if (`ASSERT_GLOBAL_RESET != 1'b0) begin
   `else
    if (reset_n != 0) begin // active low reset_n
   `endif
     if (test_expr == 1'b1) begin
       error_count = error_count + 1;
       `ifdef ASSERT_MAX_REPORT_ERROR
        if (error_count<=`ASSERT_MAX_REPORT_ERROR)
       `endif
         $display("%s : severity %0d : time %0t : %m", msg, severity_level, $time);
       if (severity_level == 0) $finish;
     end
    end
  end
 `endif
//synopsys translate_on
endmodule
```

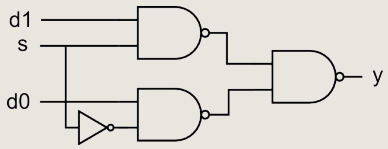
Equivalence Checking

- Validate that the implementation of a module is consistent with the specification
  - Can use simulation or formal techniques
  - Combinational or sequential modules

Example: Specification in RTL

```
module mux(input s, d0, d1,
           output y);
assign y = s ? d1 : d0;
endmodule
```

Example: Implementation at the gate level



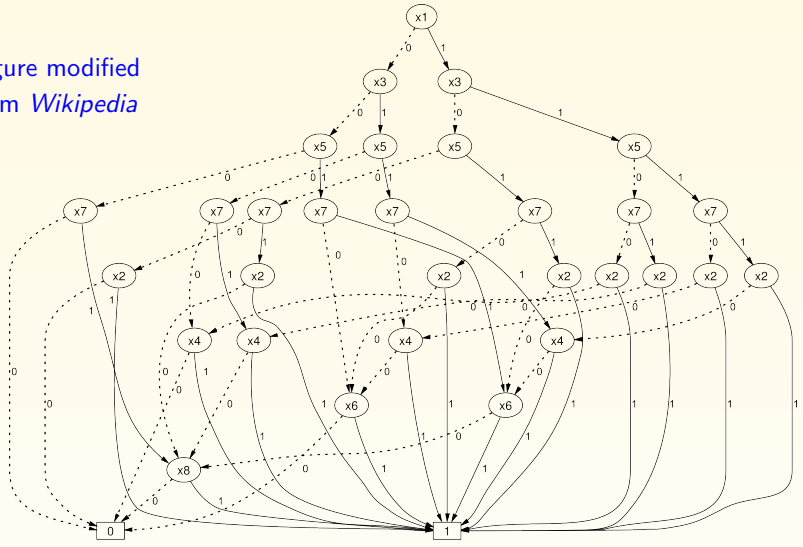
Example tool: Synopsys *Formality*

Impact of BDD Variable Ordering

$$f(x_1, x_2, \ldots, x_8) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 + x_7 \cdot x_8$$

$$Ordering : x_1 < x_3 < x_5 < x_7 < x_2 < x_4 < x_6 < x_8$$

Figure modified from *Wikipedia*

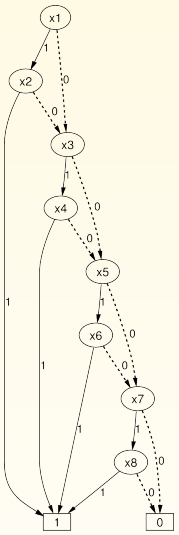


Impact of BDD Variable Ordering, Cont'd

$$f(x_1, x_2, \ldots, x_8) = x_1 \cdot x_2 + x_3 \cdot x_4 + x_5 \cdot x_6 + x_7 \cdot x_8$$

$$Ordering : x_1 < x_2 < x_3 < x_4 < x_5 < x_6 < x_7 < x_8$$

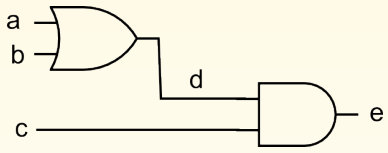
Figure modified from *Wikipedia*



SAT Solvers – Circuit to CNF

$d \equiv (a + b)$

Clauses:
$(a + b + \overline{d})$
$(\overline{a} + d)$
$(\overline{b} + d)$

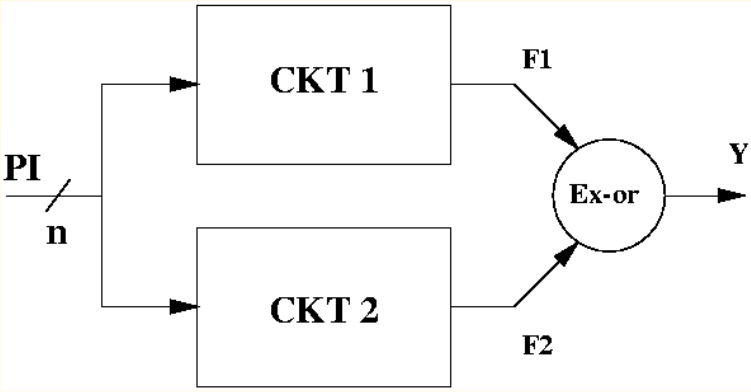


$e \equiv (c.d)$

Clauses:
$(\overline{c} + \overline{d} + e)$
$(d + \overline{e})$
$(c + \overline{e})$

Use of ATPG for Equivalence Checking

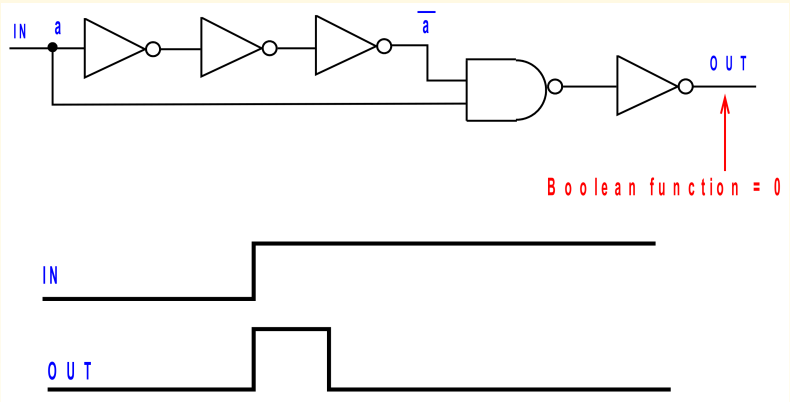
- Use a tool (Automatic Test Pattern Generator) which generates manufacturing tests
- Detecting a "stuck-at-0" fault at Y (requires an input which generates a 1 on Y) will prove inequivalence of the two circuits
- Approach is not memory limited (like BDDs)



Symbolic Simulation

- Equivalence checking between RTL and circuit schematics is difficult for some circuits (e.g., custom arrays)
  - Critical timing and self-timed control logic
  - Large number of bit-cells
  - Inherently complex sequential logic blocks
  - Dynamic logic
- Traditional tools fail on such circuits
  - Very large state space, too many initial state/input sequences for simulation-based tools
  - Boolean equivalence tools only check static cones of logic, do not capture dynamic behavior

Example: Custom Control for Custom Array Structures



- OUT pulse fans out to array READ/WRITE control signals
- Equivalence checking does not work

Scalar Simulation

To prove that the circuit is a NAND gate, exhaustive simulation requires 2^n vectors

Antecedent	Consequent
A = 0 (t0,t1) and B = 0 (t0,t1)	C is 1 (t1,t2)
A = 0 (t0,t1) and B = 1 (t0,t1)	C is 1 (t1,t2)
A = 1 (t0,t1) and B = 0 (t0,t1)	C is 1 (t1,t2)
A = 1 (t0,t1) and B = 1 (t0,t1)	C is 0 (t1,t2)

Table could be viewed as: Antecedent $\Longrightarrow$ Consequent

Ternary Simulation

Using three values (0, 1, X), N-input NAND requires N+1 vectors to verify

Antecedent	Consequent
A = 0 (t0,t1) and B = X	C is 1 (t1,t2)
A = X and B = 0 (t0,t1)	C is 1 (t1,t2)
A = 1 (t0,t1) and B = 0 (t0,t1)	C is 1 (t1,t2)

Symbolic Simulation

Exhaustive Verification: N-input NAND requires 1 vector and N variables

Antecedent: A = "a"(t0,t1) and B = "b"(t0,t1)
("a" and "b" are Boolean variables)

Consequent: C = [¬ (a AND b)](t1,t2)

Symbolic Trajectory Evaluation

- VERSYS symbolic trajectory evaluation tool developed at Motorola/Freescale
  - Based on VOSS (from CMU/UBC)
- Trajectory formulas
  - Boolean expressions with the temporal next-time operator
  - Ternary values states represented by a Boolean encoding
- Properties of type: Antecedent $\Longrightarrow$ Consequent
- Used to verify PowerPC arrays at Motorola/Freescale in 8 – 10% of the design time
- Bugs found during array equivalence checking
  - Incorrect clock regenerators feeding latches
  - Control logic errors in READ/WRITE enables
  - Violation of "one-hot" property assumptions
  - Scan chain hookup errors
  - Potential circuit-related problems such as glitches and races
- Other commercial applications: Intel (*Forte*), etc.

Design Verification

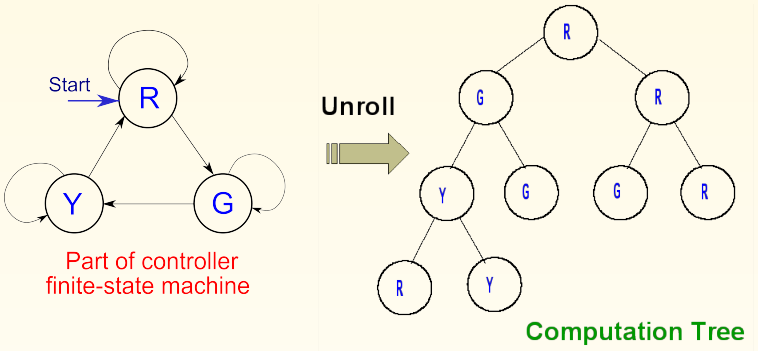
- Digital systems similar to **reactive programs**
- Digital systems receive inputs and produce outputs in a continuous interaction with their environment
- Behavior of digital systems is concurrent because each gate in the system simultaneously evaluating its output as a function of its inputs

Check Properties of Design

- Since specification is usually not formal, check design for properties that would be consistent with the specification
- Safety "something bad will never happen"
- Liveness Property: "something good will eventually happen"
- Temporal Logic and variations commonly used to specify properties
- Example: Linear Temporal Logic (LTL) or Computation Tree Logic (CTL)

Example of Computation Tree

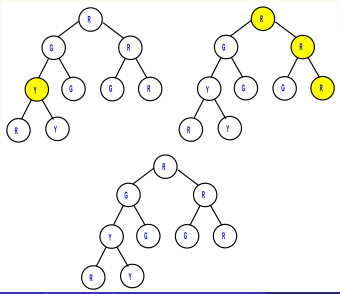
Traffic light controller



Operators

- Referring to paths
  - A: For every path
  - E: There exists a path
- Referring to states on a path
  - G: Globally
  - F: In the future (eventually)
  - X: Next state along the path
- Examples
  - EF p: there is some path on which p is eventually true
  - AG p: for every path, at every state, p is true

EF Y
(True)

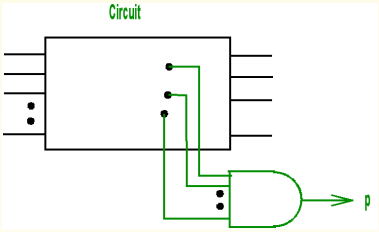


EG R
(True)

AG(R+G)
(False)

Use of ATPG to Check Properties

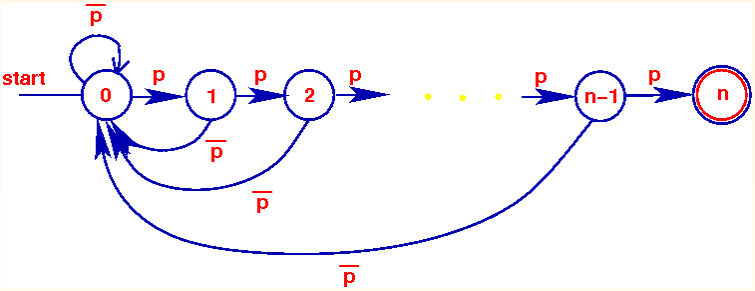
- This moves verification of the design to the same level as the models used to generate manufacturing test of the physical chip
- Using ATPG allows the verification engine to deal with tri-state signals, multiple clocks, etc.



Bounded Model Checking: Prove properties for a limited number of cycles

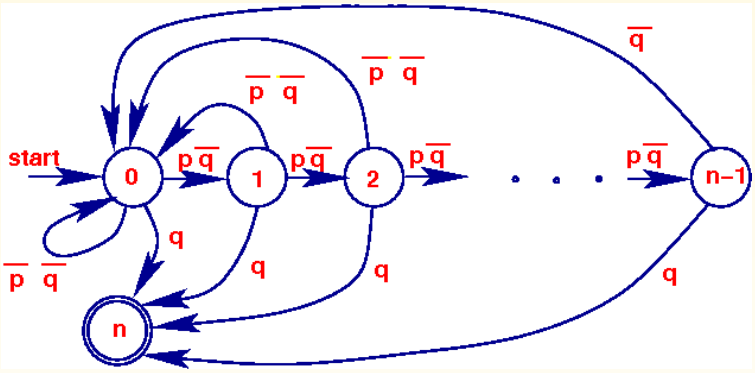
Monitor State Machine for EGp

Find an input sequence of length **n** for which the system will satisfy the property **p**



Monitor State Machine for EpUq

For some path of up to **n** cycles, there is a state where **q** holds and **p** holds in every previous state



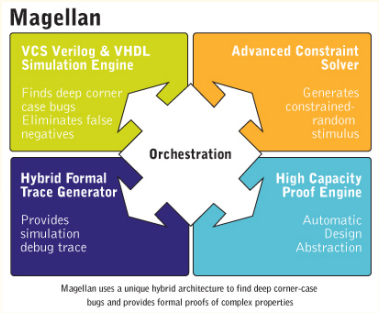
Model Checking on IBM Power 4

- "Functional formal verification" (equivalence checking and model checking) on ≈40 design components (IU, FPU, control, memory, etc.)
- Found more than 200 design flaws at various stages and of varying complexity
- At least one bug was found by almost every application of formal verification
- Estimate: 15% of bugs would have evaded simulation
  - Some of the bugs literally escaped 1-2 years of simulation

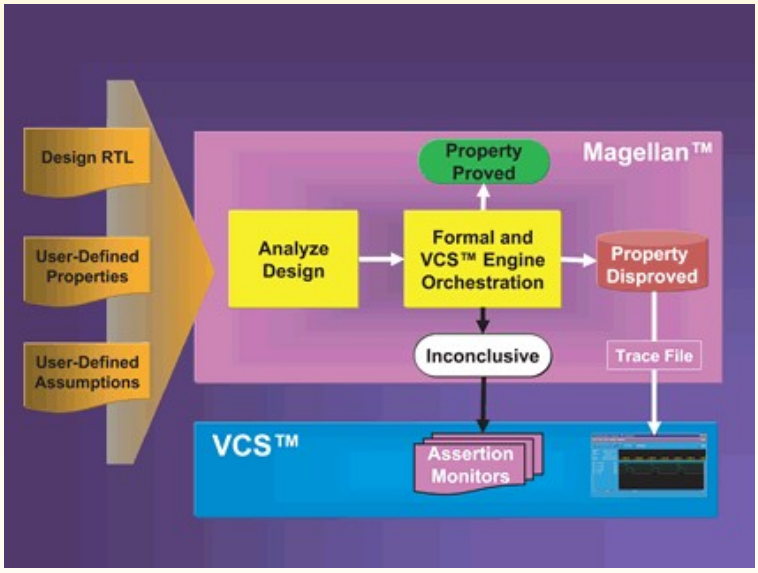
Example Industry Tool: Synopsys Magellan

A "Hybrid" Verification Tool

- Functional verification tool
- Static formal mathematical techniques
- Dynamic random simulation
- Test properties specified in systhesizable System Verilog Assertions



Magellan Flow



Specifying Properties (Assertions) in Industry Tools

- Used for both simulation monitoring and formal verification
- Examples of assertion languages include Vera (Synopsys), Sugar (IBM), Property Specification Language,PSL (Accelera consortium), System Verilog

PSL/Sugar

- Core based on Boolean and Temporal logic
- Layer of user-friendly "syntactic sugar"
- Comes in three flavors
  - Verilog
  - VHDL
  - GDL
- Reference Manual:
  http://www.eda.org/vfv/docs/psl_lrm-1.01.pdf

System Verilog Assertions (SVA)

SVA

- Assertions: Predicates placed in program
- Immediate and Concurrent Assertions
- assert, assume, cover, expect constructs

Immediate Assertions

```
assert (a == b);
```

Concurrent Assertions

```
assert property (@(posedge clk) req |→ ack);
```

Reasoning at a Higher Level of Abstraction

SAT (used by BMC, k-Induction, Unbounded model checking) and BDDs do not scale well; need to use high level models to tackle state explosion

Model Checking based on Satisfiability Modulo Theories (SMT)

- Word level model can be represented as a finite-precision bit-vector formula with arithmetic operators such as addition
- SMT solvers can solve such bit vector formulae using a combination of Boolean SAT and automatic abstraction, hence are more scalable than SAT solvers

Predicate Abstraction for Model Checking

- Abstract data by keeping predicates on variables
- Each predicate is represented by a Boolean variable in the abstract model
- Predicate Abstraction is effective if refinement procedure is able to infer strong word-level predicates

Dealing with State Explosion

Verification is a very difficult problem

- Even combinational equivalence checking problems (ATPG, SAT) are NP-complete
- Checking sequential properties is only possible for small designs
- Additional problem of generating correct "wrappers" for the module being verified

How can we deal with the complexity?

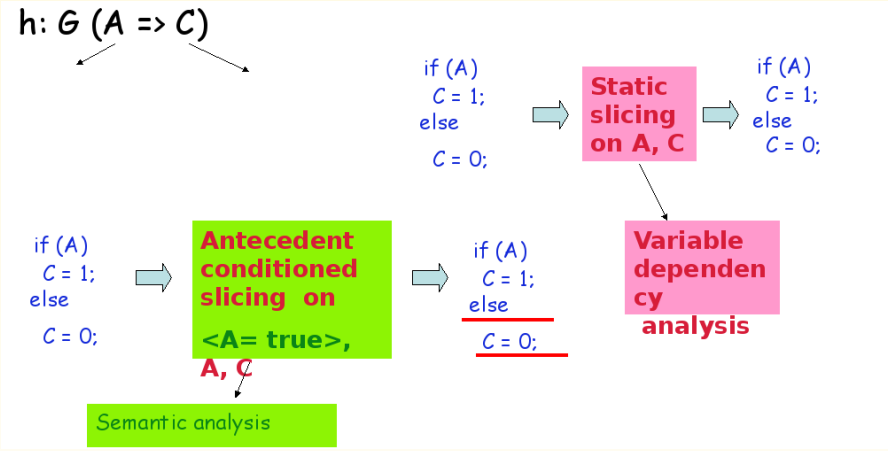
- Use more powerful computers?
  - Computers double in capability (assuming we can program multi-core processors) every couple of years
  - Adding one state variable to a design doubles its states
- Exploit hierarchy in the design
- Develop powerful abstractions

Program Slicing

A Slice of a Design

- Represents behavior of the design with respect to a given set of variables (or slicing criterion)
- Proposed for use in software in 1984 (Weiser)
- Slice generated by a control/data flow analysis of the program code
- Slicing is done on the **structure** of the design, so scales well
- "Static analysis"

Antecedent Conditioned Slicing



Example of Antecedent Conditioned Slicing – I

```
always @ (clk)   begin    always @ (clk)   begin    always @ (clk)   begin
    case(insn)                 case(dec)                  case(ex)
      f_add:  dec = d_add;     d_add:  ex = e_add;        e_add:  res = a+b;
      f_sub:  dec = d_sub;     d_sub:  ex = e_sub;        e_sub:  res = a-b;
      f_and:  dec = d_and;     d_and:  ex = e_and;        e_and:  res = a&b;
      f_or:   dec = d_or;      d_or:   ex = e_or;         e_or:   res = a|b;
    endcase                    endcase                    endcase
end                        end                        end
```

$h = [G((insn == f\_add) \Rightarrow XX(res == a+b))]$

Example of Antecedent Conditioned Slicing – II

```
always @ (clk)   begin    always @ (clk)   begin    always @ (clk)   begin
    case(insn)                 case(dec)                  case(ex)
      f_add:  dec = d_add;     d_add:  ex = e_add;        e_add:  res = a+b;

    endcase                    endcase                    endcase
end                        end                        end
```

Single instruction behavior for f_add instruction

$h = [G((insn == f\_add) \Rightarrow XX(res == a+b))]$

Experiments with Antecedent Conditioned Slicing

USB 2.0 Function Core

- Verilog implementation from www.opencores.org
- Properties from specification document
- Safety properties expressed in LTL $(G(a \implies c))$
- Verification engine: Cadence-BMC (bound of 24–50 steps)

Example USB Properties

$G((crc5err \vee \neg(match)) \implies \neg(send\_token))$

If a packet with a bad CRC5 is received, or there is an endpoint field mismatch, the token is ignored

$G((state == SPEED\_NEG\_FS) \implies X((mode\_hs) \wedge (T1\_gt\_3\_0ms) \implies (next\_state == RES\_SUSPEND))$

If the machine is in the speed negotiation state, then in the next clock cycle, if it is in high speed mode for more than 3 ms, it will go to the suspend state

$G((state == RESUME\_WAIT) \wedge \neg(idle\_cnt\_clr) \implies F(state == NORMAL))$

If the machine is waiting to resume operation and a counter is set, eventually (after 100 mS) it will return to normal operation

Results on Temporal USB Properties

CPU seconds, on a 450 MHz dual UltraSPARC-II with I GB RAM