

EE382V: System-on-a-Chip (SoC) Design

Lecture 20 – SoC Verification

*Sources:
Jacob A. Abraham*

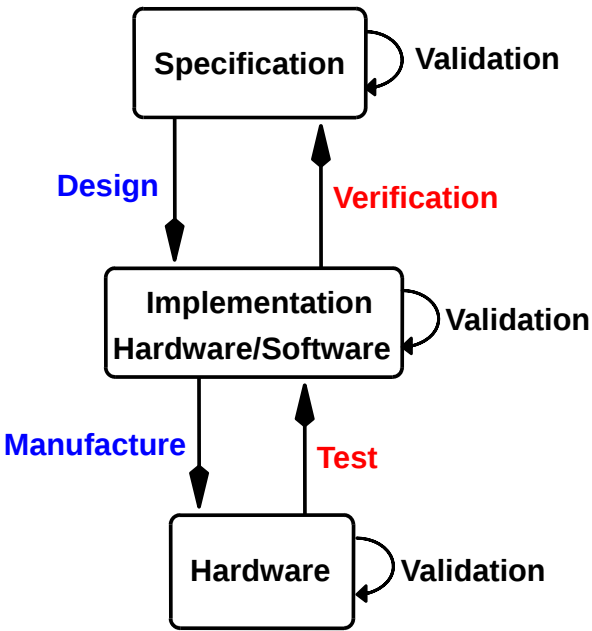
Andreas Gerstlauer
Electrical and Computer Engineering
University of Texas at Austin
gerstl@ece.utexas.edu



Lecture 20: Outline

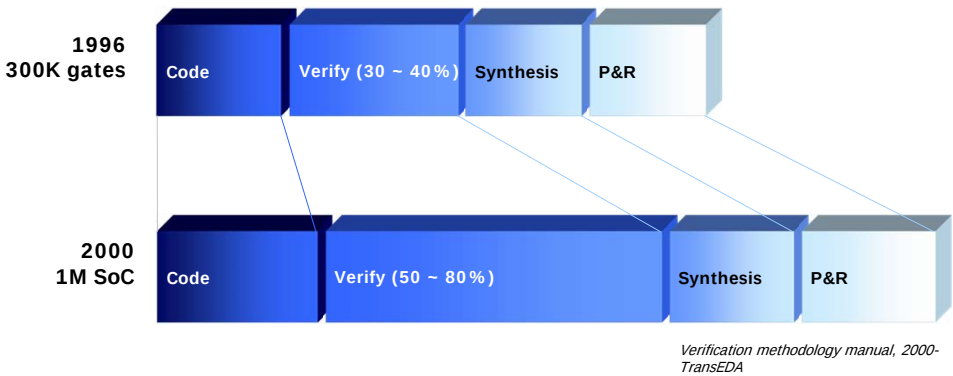
- **Verification**
 - Verification flow
- **Verification methods**
 - Simulation-based techniques
 - Formal analysis
 - Semi-formal approaches
- **Formal verification**
 - Dealing with state explosion
 - Property checking
 - Equivalence checking
 - Software verification

Verification versus Test



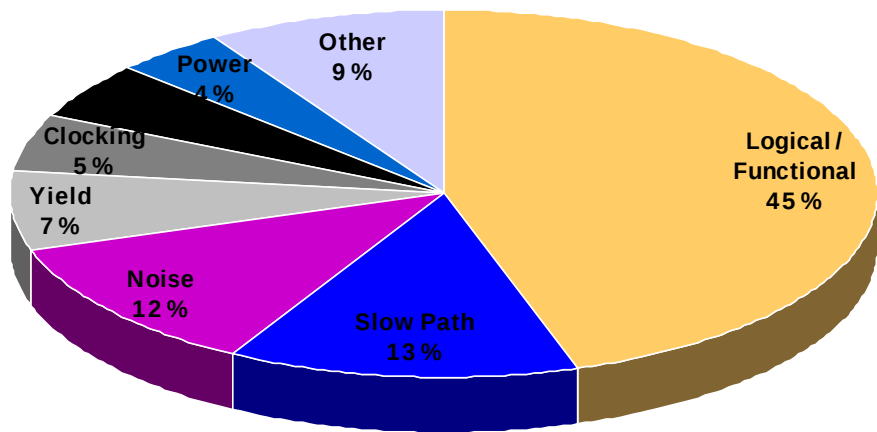
Verification Effort

- **Verification** portion of design increases to anywhere from **50 to 80%** of total **development effort** for the design.



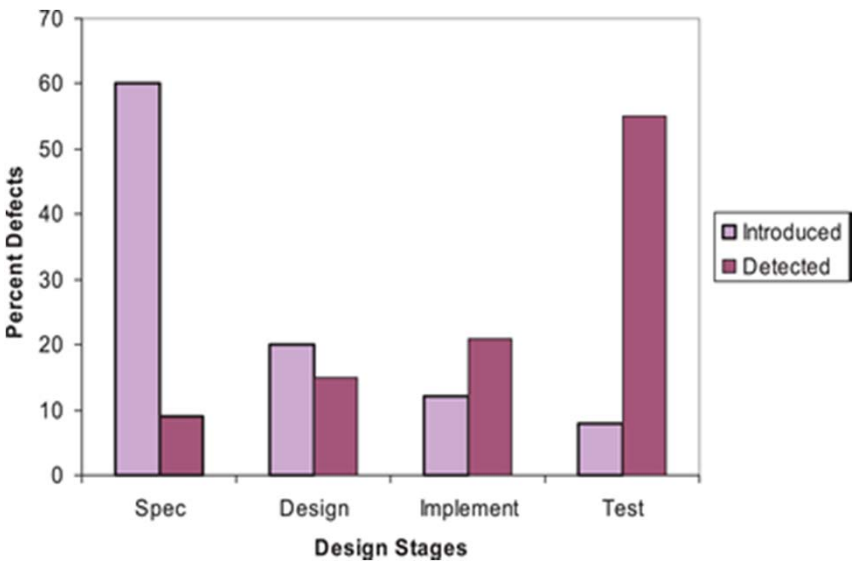
Percentage of Total Flaws

- About 50% of flaws are functional flaws
 - Need verification method to fix logical & functional flaws



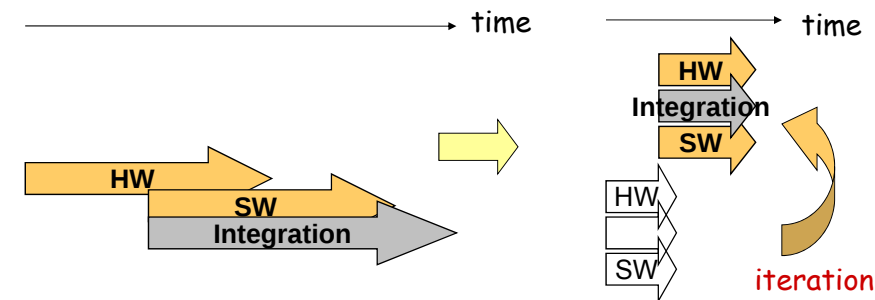
From Mentor presentation material, 2003

“Bug” Introduction and Detection

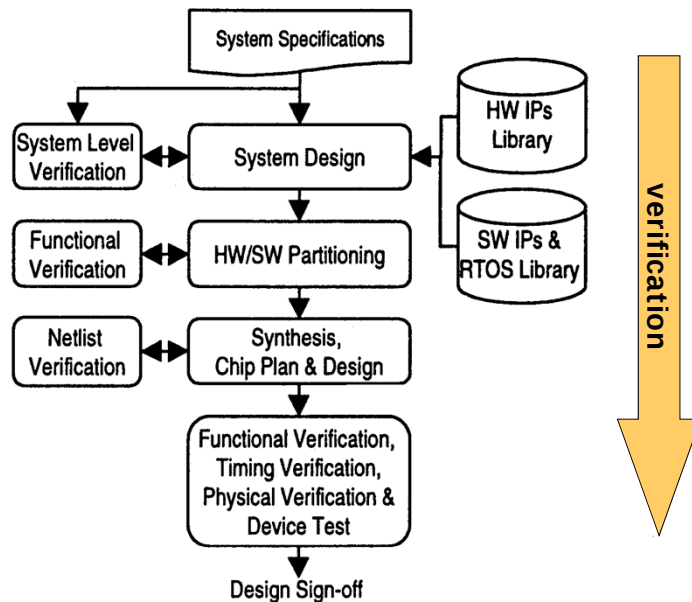


HW/SW Co-Design

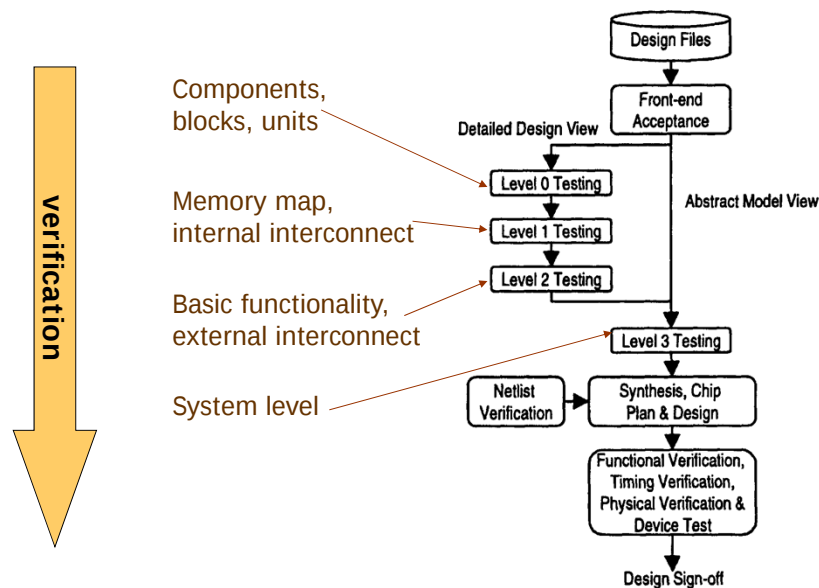
- Concurrent design of HW/SW components
- Evaluate the effect of a design decision at early stage by “virtual prototyping”
- Co-verification



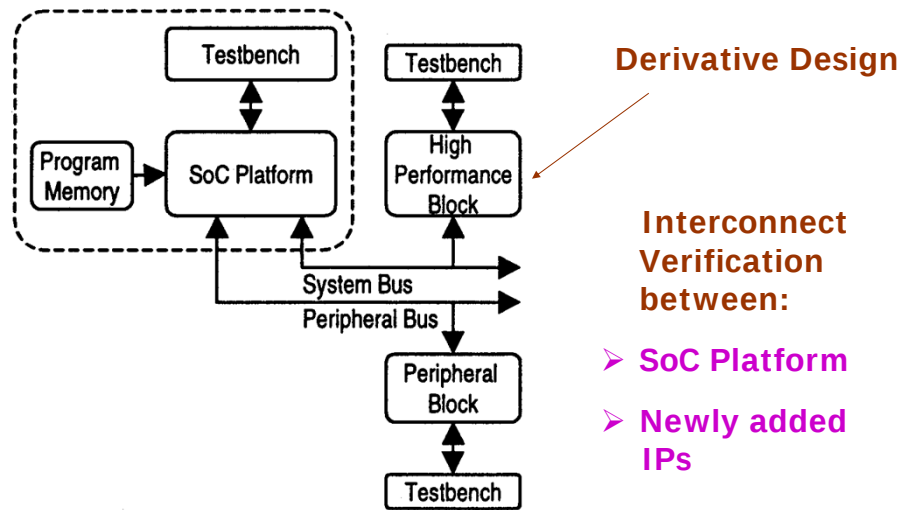
Top-Down SoC Verification



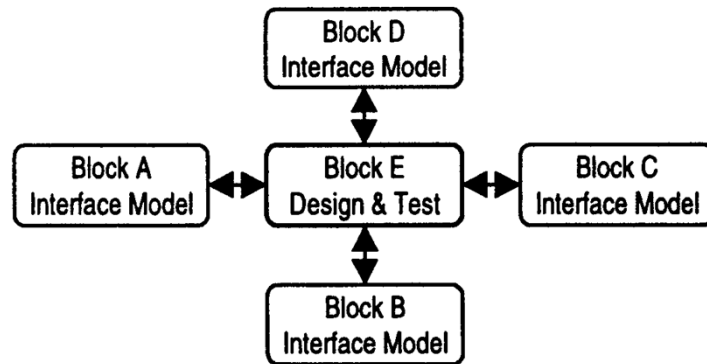
Bottom-Up SoC Verification



Platform Based SoC Verification

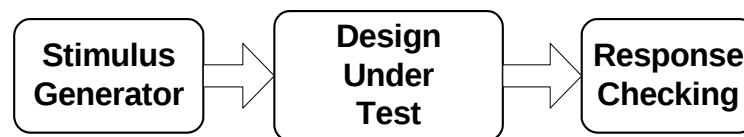


Interface-Driven SoC Verification



**Besides Design-Under-Test,
all others are interface models**

Traditional Specification



• Problems of Traditional Testbench

- Real-World Stimuli
- System-Level Modeling
- High-Level Algorithmic Modeling
- Test Automation
- Source Coverage

Lecture 20: Outline

✓ Verification

✓ Verification flow

• Verification methods

- Simulation-based techniques
- Formal analysis
- Semi-formal approaches

• Formal verification

- Dealing with state explosion
- Property checking
- Equivalence checking
- Software verification

Design Verification Methods

• Simulation based methods

- Specify input test vector, output test vector pair
- Run simulation and compare output against expected output

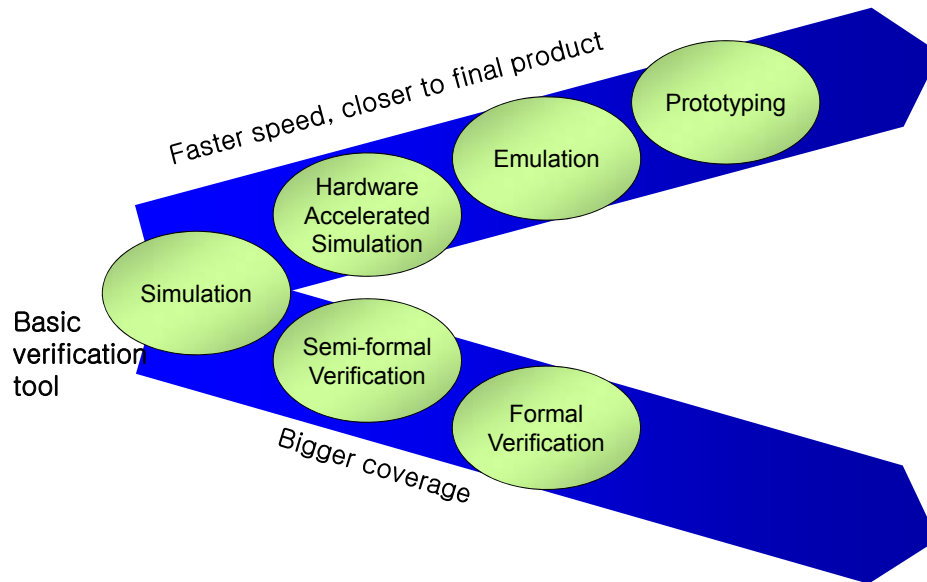
• Formal Methods

- Check equivalence of design models or parts of models
- Check specified properties on models

• Semi-formal Methods

- Specify inputs and outputs as symbolic expressions
- Check simulation output against expected expression

Verification Approaches



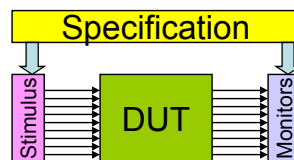
EE382V: SoC Design, Lecture 20

© J. A. Abraham

15

Simulation

- **Create test vectors and simulate model**
 - Simulation, debugging and visualization tools [Synopsys VCS, Mentor ModelSim, Cadence NC-Sim]



- **Inputs**
 - Specification
 - Used to create interesting stimuli and monitors
 - Model of DUT
 - Typically written in HDL or C or both
- **Output**
 - Failed test vectors
 - Pointed out in different design representations by debugging tools

EE382V: SoC Design, Lecture 20

© 2014 A. Gerstlauer

16

Simulation Technologies

- **Different techniques at varying levels of abstraction**
 - Numerical Simulation (MATLAB)
 - AMS Simulation
 - Transaction-based Simulators
 - HW/SW co-simulation
 - Cycle-based Simulators
 - Event-based Simulators
 - Code Coverage
 - Emulation Systems
 - Rapid Prototyping Systems
 - Hardware Accelerators

Static Technologies

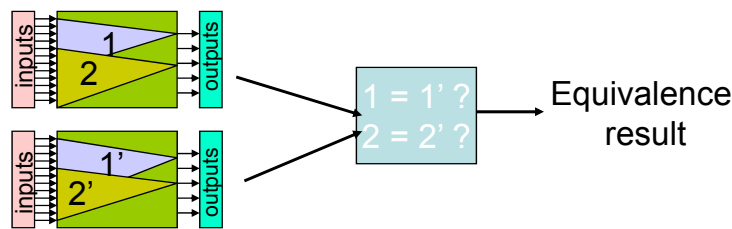
- **“Lint” Checking**
 - Syntactic correctness
 - Identifies simple errors
- **Static Timing Verification**
 - Setup, hold, delay timing requirements
 - Challenging: multiple sources

Formal Techniques

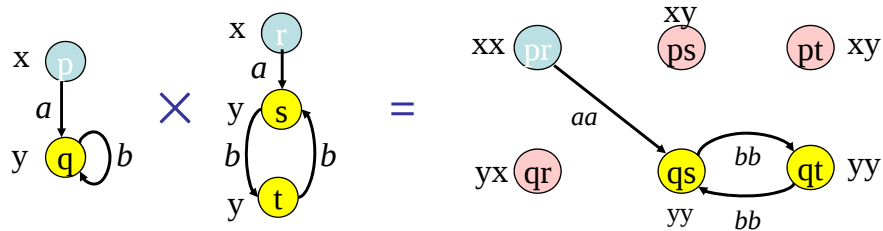
- **Theorem Proving Techniques**
 - Proof-based
 - Not fully automatic
- **Formal Model Checking**
 - Model-based
 - Automatic
- **Formal Equivalence Checking**
 - Reference design ↔ modified design
 - RTL-RTL, RTL-Gate, Gate-Gate implementations
 - No timing verification

Equivalence Checking

- LEC uses boolean algebra to check for logic equivalence

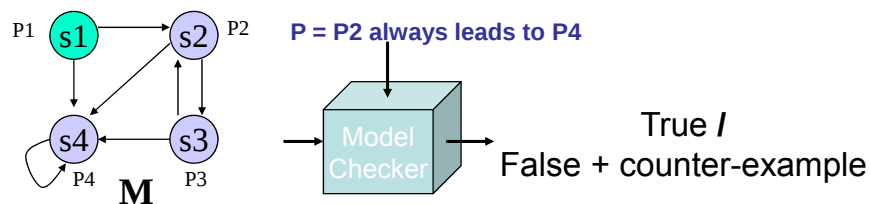


- SEC uses FSMs to check for sequential equivalence



Model Checking

- **Model M satisfies property P ? [Clarke, Emerson '81]**
- **Inputs**
 - State transition system representation of M
 - Temporal property P as formula of state properties
- **Output**
 - True (property holds)
 - False + counter-example (property does not hold)



EE382V: SoC Design, Lecture 20

© 2014 A. Gerstlauer

21

Semi-Formal Methods

- **Executable specification for behavioral modeling**
 - Design Productivity
 - Easy to model complex algorithm
 - Fast execution
 - Simple Testbench
 - Tools
 - Native C/C++ through PLI/FLI
 - Extended C/C++ : SpecC, SystemC
- **Verify it on the fly!**
 - Test vector generation
 - Compare RTL Code with Behavioral Model
 - Coverage Test

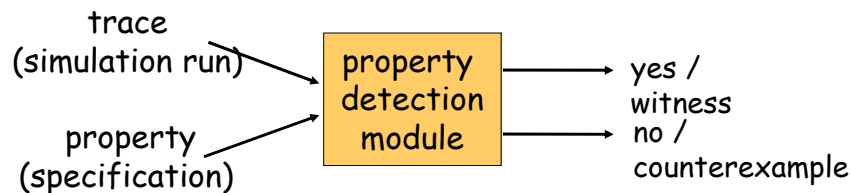
EE382V: SoC Design, Lecture 20

© J. A. Abraham

22

Assertion-Based Verification

- **Property Detection:** To decide whether a simulation run (trace) of a design satisfies a given property (assertion)



e.g., violation of mutual exclusion, $\text{critical}_1 \wedge \text{critical}_2$

➤ Temporal logic

- Example: Properties written in PSL/Sugar

Specifying Properties (Assertions)

- **Open Vera Assertions Language (Synopsys)**
- **Property Specification Language (PSL)** (IBM, based on *Sugar*)
 - Accelera driving consortium
 - IEEE Std. 1850-2005
- **Accelera Open Verification Library (OVL)** provides ready to use assertion functions in the form of VHDL and Verilog HDL libraries
- **SystemVerilog** is a next generation language, added to the core Verilog HDL
 - IEEE Std. 1800-2005

Lecture 20: Outline

✓ Verification

- ✓ Verification flow

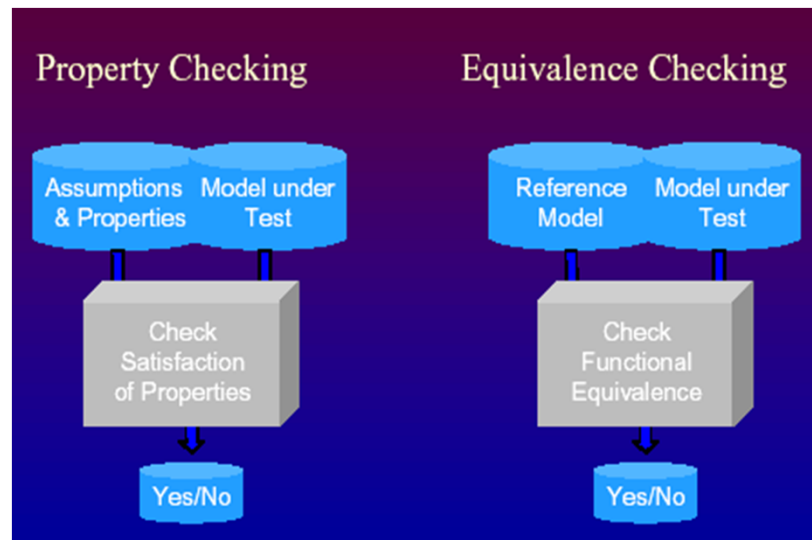
✓ Verification methods

- ✓ Simulation-based techniques
- ✓ Formal analysis
- ✓ Semi-formal approaches

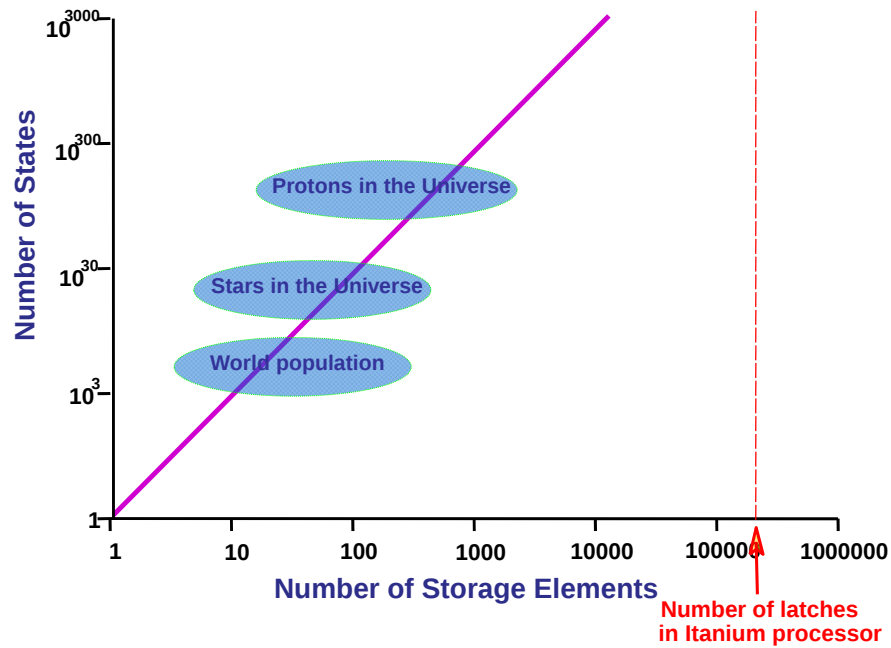
• Formal verification

- Dealing with state explosion
- Property checking
- Equivalence checking
- Software verification

Formal Verification of SoCs



State Explosion!



EE382V: SoC Design, Lecture 20

© J. A. Abraham

27

Abstractions to Deal with Large State Spaces

- Model checking models need to be made smaller
- Problem: **State-Space Explosion**
- Smaller or “reduced” models must retain information
 - Property being checked should yield same result

➤ **Balancing solution: Abstractions**

EE382V: SoC Design, Lecture 20

© J. A. Abraham

28

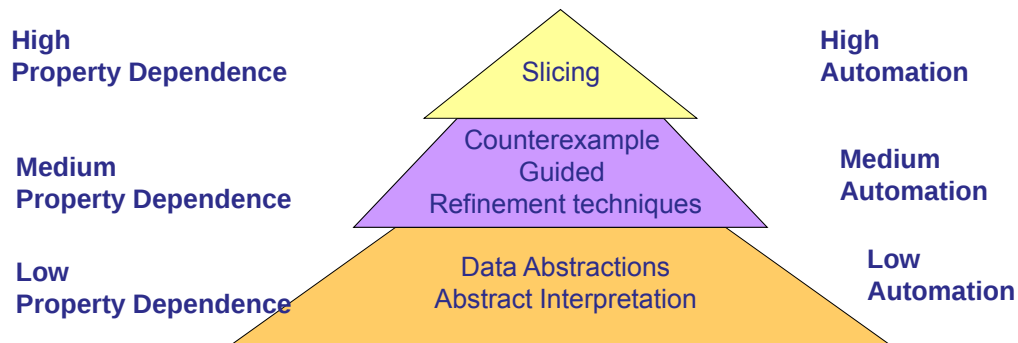
Program Transformation Based Abstractions

- **Abstractions on Kripke structures**
 - Cone of Influence (COI), Symmetry, Partial Order, etc.
 - State transition graphs for even small programs can be very large to build
- **Abstractions on program text**
 - Scale well with program size
 - High economic interest
 - Static program transformation

Types of Abstractions

- **Sound**
 - Property holds in abstraction implies property holds in the original program
- **Complete**
 - Algorithm always finds an abstract program if it exists
- **Exact**
 - Property holds in the abstraction iff property holds in the main program

Abstraction Landscape



EE382V: SoC Design, Lecture 20

© J. A. Abraham

31

Property Checking

- **High level symbolic simulation**
 - Symbolic simulation of antecedent
 - Symbolic simulation of all CFG nodes
- **Domain aware analysis**
 - Function-wise case splitting
- **Decision procedure**
 - Model checker
- **Antecedent conditioned slicing**
 - RTL abstraction technique
 - Applied to LTL formulas $G(a \Rightarrow c)$
 - Theoretically complex, practically effective
 - USB 2.0 protocol verification

EE382V: SoC Design, Lecture 20

© J. A. Abraham

32

Program Slicing

- Program transformation involving statement deletion
- “Relevant statements” determined according to *slicing criterion*
- Slice construction is completely *automatic*
- Correctness is *property specific*
 - Loss of generality
- Abstractions are sound and complete

Specialized Slicing Techniques

- Static slicing produces large slices
 - Has been used for verification
 - Semantically equivalent to COI reductions
- Slicing criterion can be enhanced to produce other types of slices
 - Amorphous Slicing
 - Conditioned Slicing

Conditioned Slicing

- Slices constructed with respect to set of possible input states
- Characterized by first order, predicate logic formula
- Augments static slicing by introducing condition
 - $\langle C, I, V \rangle$
 - Constrains the program according to condition C
- [Canfora et al.]

Example Program

```
begin

1:      read(N);
2:      A = 1;
3:      if (N < 0) {
4:          B = f(A);
5:          C = g(A);
6:      } else if (N > 0) {
7:          B = f'(A);
8:          C = g'(A);
9:      } else {
10:         B = f''(A);
11:         C = g''(A);
12:      }

      print(B);
      print(C);

end
```

Static Slicing wrt <11, B>

```

begin

1:      read(N);
2:      A = 1;
3:      if (N < 0) {
4:          B = f(A);
5:          C = g(A);
6:      } else if (N > 0) {
7:          B = f'(A);
8:          C = g'(A);
9:      } else {
10:         B = f''(A);
11:         C = g''(A);
12:     }

    print(B);
    print(C);

end

```

EE382V: SoC Design, Lecture 20

© J. A. Abraham

37

Conditioned Slicing wrt <(N<0),11, B>

```

begin

1:      read(N);
2:      A = 1;
3:      if (N < 0) {
4:          B = f(A);
5:          C = g(A);
6:      } else if (N > 0) {
7:          B = f'(A);
8:          C = g'(A);
9:      } else {
10:         B = f''(A);
11:         C = g''(A);
12:     }

    print(B);
    print(C);

end

```

EE382V: SoC Design, Lecture 20

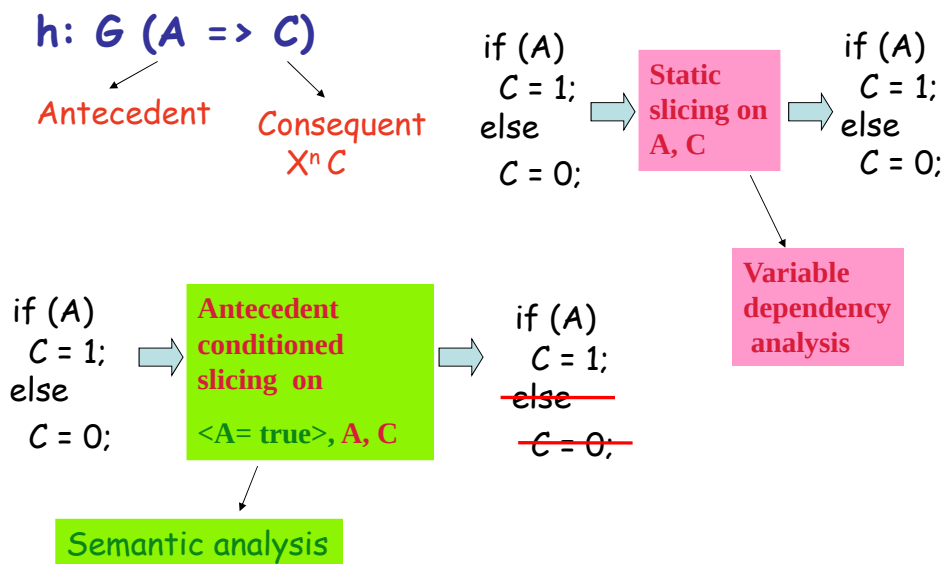
© J. A. Abraham

38

Verification Using Conditioned Slicing

- Slicing part of design irrelevant to property being verified
- Safety Properties of the form
 - $G (\text{antecedent} \Rightarrow \text{consequent})$
- Use *antecedent* to specify states we are interested in
 - We do not need to preserve program executions where the antecedent is false

Antecedent Conditioned Slicing



Example

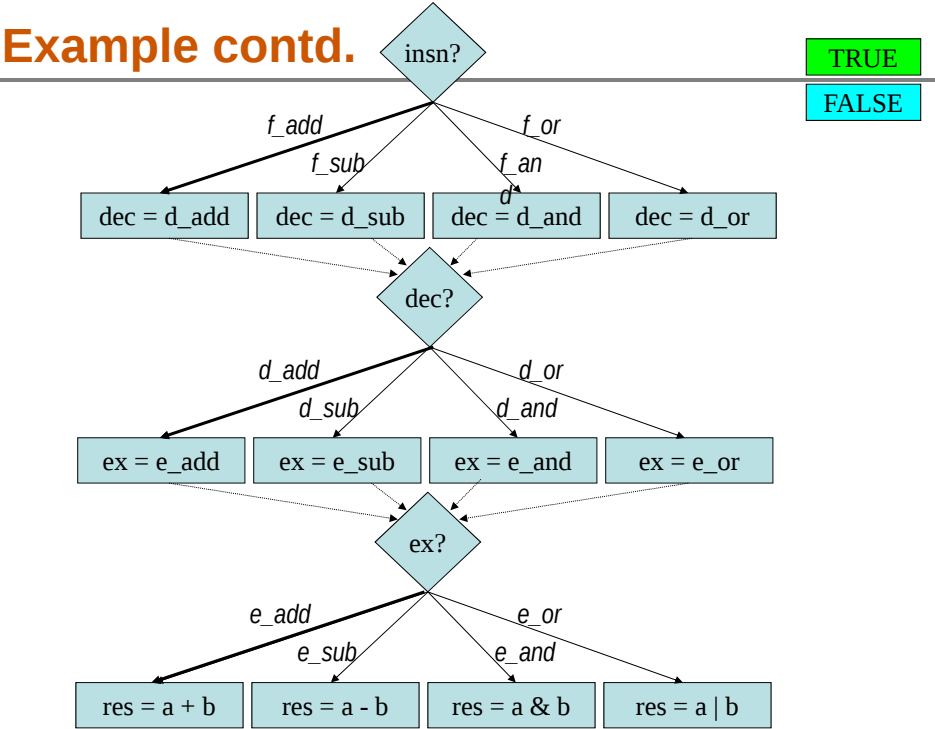
```
always @ (clk) begin
  case(insn)
    f_add: dec = d_add;
    f_sub: dec = d_sub;
    f_and: dec = d_and;
    f_or:  dec = d_or;
  endcase
end
```

```
always @ (clk) begin
  case(dec)
    d_add: ex = e_add;
    d_sub: ex = e_sub;
    d_and: ex = e_and;
    d_or:  ex = e_or;
  endcase
end

always @ (clk) begin
  case(ex)
    e_add: res = a+b;
    e_sub: res = a-b;
    e_and: res = a&b;
    e_or:  res = a|b;
  endcase
end
```

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

Example contd.



Example contd.

```

always @ (clk)    begin
  case(insn)
    f_add: dec = d_add;

  endcase
end

always @ (clk)    begin
  case(ex)
    e_add: res = a+b;

  endcase
end

always @ (clk)    begin
  case(dec)
    d_add: ex = e_add;

  endcase
end

```

Single instruction behavior for f_add instruction

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

Experimental Results

- **Verilog RTL implementation of USB 2.0 function core**
 - USB has many interacting state machines
 - Approximately 10^{33} states
 - Properties taken from specification document
 - Mostly control based, state machine related
- **Temporal property verification**
 - Safety properties of the form (in LTL)
 - $G(a \Rightarrow Xc)$
 - $G(a \Rightarrow a \cup_s c)$
 - Liveness Properties
 - $G(a \Rightarrow Fc)$
- **Used Cadence SMV-BMC**
 - Circuit too big for SMV
 - Used a bound of 24

Example Properties of the USB

- $G((\text{crc5err}) \vee \neg(\text{match}) \Rightarrow \neg(\text{send_token}))$
 - If a packet with a bad CRC5 is received, or there is an endpoint field mismatch, the token is ignored
- $G((\text{state} == \text{SPEED_NEG_FS}) \Rightarrow X((\text{mode_hs}) \wedge (\text{T1_gt_3_0ms}) \Rightarrow (\text{next_state} == \text{RES_SUSPEND})))$
 - 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
- $G((\text{state} == \text{RESUME_WAIT}) \wedge \neg(\text{idle_cnt_clr}) \Rightarrow F(\text{state} == \text{NORMAL}))$
 - If the machine is waiting to resume operation and a counter is set, eventually (after 100 mS) it will return to normal operation

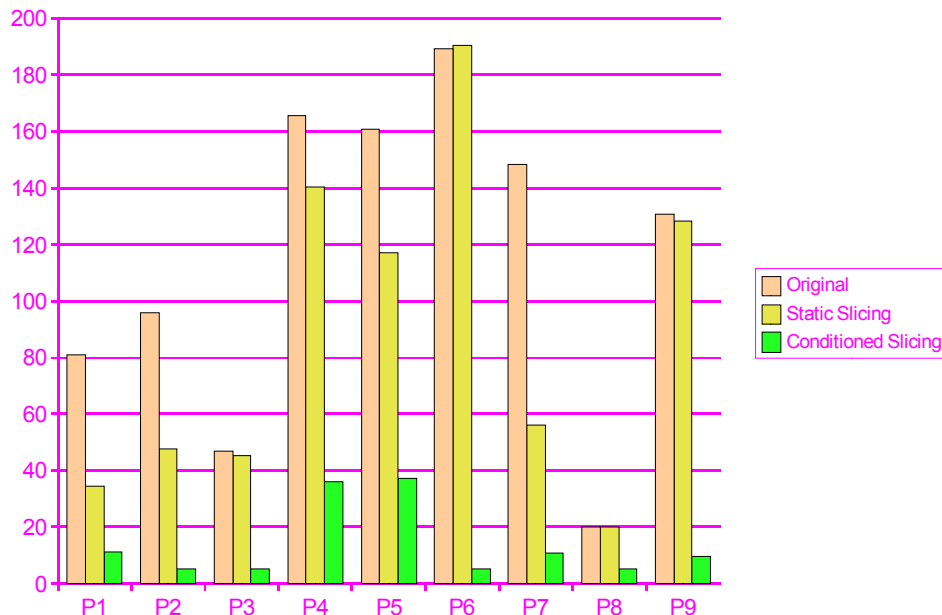
EE382V: SoC Design, Lecture 20

© J. A. Abraham

45

Results on USB $G(a \Rightarrow c)$ Properties

CPU Seconds



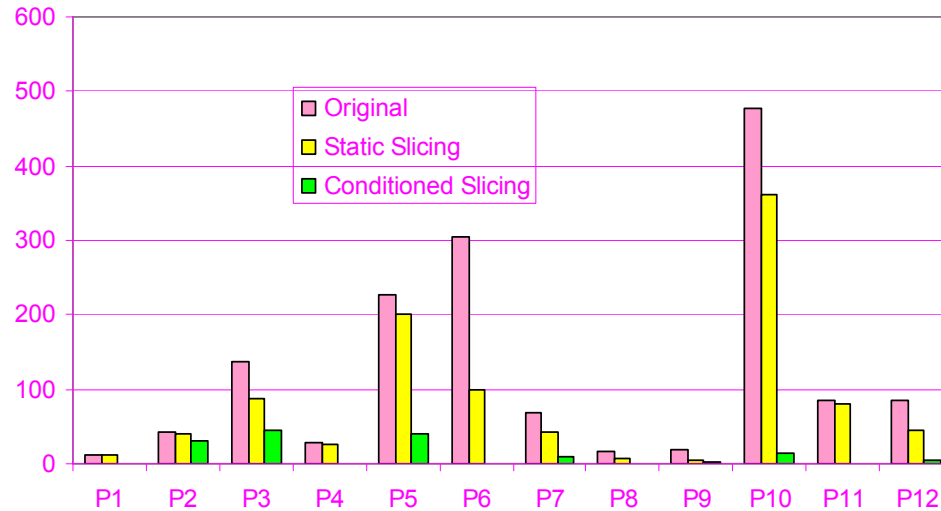
EE382V: SoC Design, Lecture 20

© J. A. Abraham

46

Results on Temporal USB Properties

CPU Seconds



EE382V: SoC Design, Lecture 20

© J. A. Abraham

47

Equivalence Checking

- **Sequential equivalence checking**
 - Verifying two models with different state encodings
- **System specifications as system-level model (SLM)**
 - Higher level of abstraction
 - Timing-aware models
- **Design concept in RTL needs checking**
 - Retiming, power, area modifications
 - Every change requires verification against SLM
- **Simulation of SLM**
 - Tedious to develop
 - Inordinately long running times

EE382V: SoC Design, Lecture 20

© J. A. Abraham

48

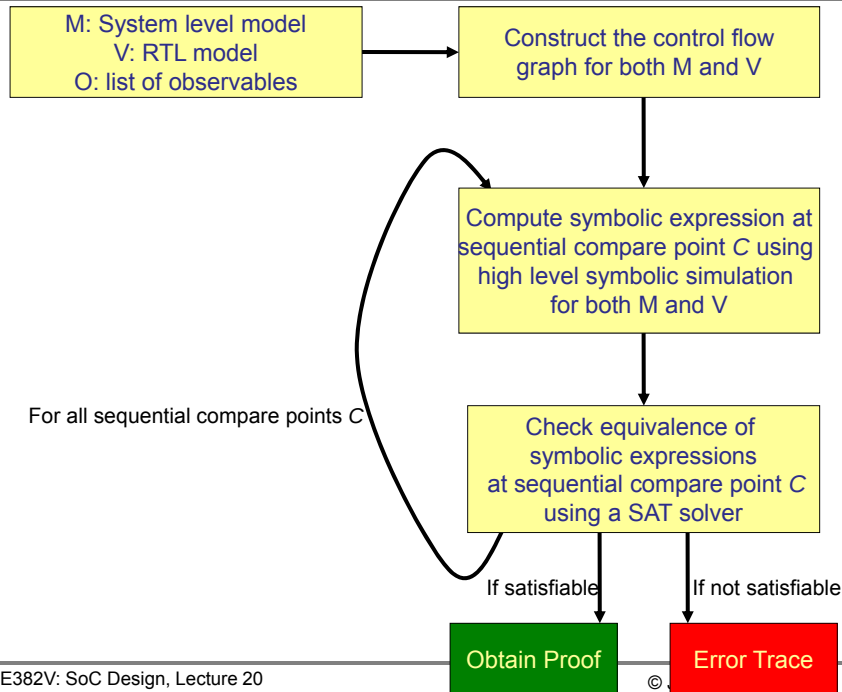
Sequential Equivalence Checking

- **High-level symbolic simulation of RTL implementation**
- **High-level symbolic simulation of system-level spec**
- **Domain aware analysis**
 - Sequential compare points obtained using heuristics
- **Decision procedure**
 - SAT solver

Sequential Compare Points

- **Variables of interest (observables) obtained from user/block diagram**
 - Primary outputs / relevant intermediate variables
- **Symbolic expressions obtained for observables assigned in a given cycle (high level symbolic simulation)**
- **Introduce notion of sequential compare points**
 - Identification with respect to relative position in time
 - Identification with respect to space (data or variables)
- **Symbolic expressions compared at sequential compare points**
 - Comparison using a SAT solver in this work
 - Other Boolean level engines can also be used

Algorithm



EE382V: SoC Design, Lecture 20

©

51

Verifying Embedded Software

- **Software Testing**
 - Execute software for test cases
 - Analogous to simulation in hardware
- **Testing Criteria**
 - Coverage measures
- **Formal analysis of software**
 - Model Checking
 - Theorem Proving

EE382V: SoC Design, Lecture 20

© J. A. Abraham

52

Software Path Testing

- **Assumption: bugs affect the control flow**
- **Execute all possible control flow paths through the program**
 - Attempt 100% path coverage
- **Execute all statements in program at least once**
 - 100% statement coverage
- **Exercise every branch alternative during test**
 - Attempt 100% branch coverage

Software Verification

- **Formal analysis of code**
- **Result, if obtained, is guaranteed for all possible inputs and all possible states**
- **Example of software model checker: SPIN**
- **Problem: applicable only to small modules**
 - State Explosion

Data Abstractions

- **Abstract data information**
 - Typically manual abstractions
- **Infinite behavior of system abstracted**
 - Each variable replaced by abstract domain variable
 - Each operation replaced by abstract domain operation
- **Data independent systems**
 - Data values do not affect computation
 - Datapath entirely abstracted

Data Abstractions: Examples

- **Arithmetic operations**
 - Congruence modulo an integer
 - k replaced by $k \bmod m$
- **High orders of magnitude**
 - Logarithmic values instead of actual data value
- **Bitwise logical operations**
 - Large bit vector to single bit value
 - *Parity generator*
- **Cumbersome enumeration of data values**
 - Symbolic values of data

Abstract Interpretation

- **Abstraction function mapping concrete domain values to abstract domain values**
- **Over-approximation of program behavior**
 - Every execution corresponds to abstract execution
- **Abstract semantics constructed once, manually**

Abstract Interpretation: Examples

- **Sign abstraction**
 - Replace integers by their sign
 - Each integer K replaced by one of $\{> 0, < 0, =0\}$
- **Interval Abstraction**
 - Approximates integers by maximal and minimal values
 - Counter variable i replaced by lower and upper limits of loop
- **Relational Abstraction**
 - Retain relationship between sets of data values
 - Set of integers replaced by their convex hull

Counterexample Guided Refinement

- **Approximation on set of states**
 - Initial state to bad path
- **Successive refinement of approximation**
 - Forward or backward passes
- **Process repeated until fixpoint is reached**
 - Empty resulting set of states implies property proved
 - Otherwise, counterexample is found
- **Counterexample can be spurious because of over-approximations**
 - Heuristics used to determine spuriousness of counterexamples

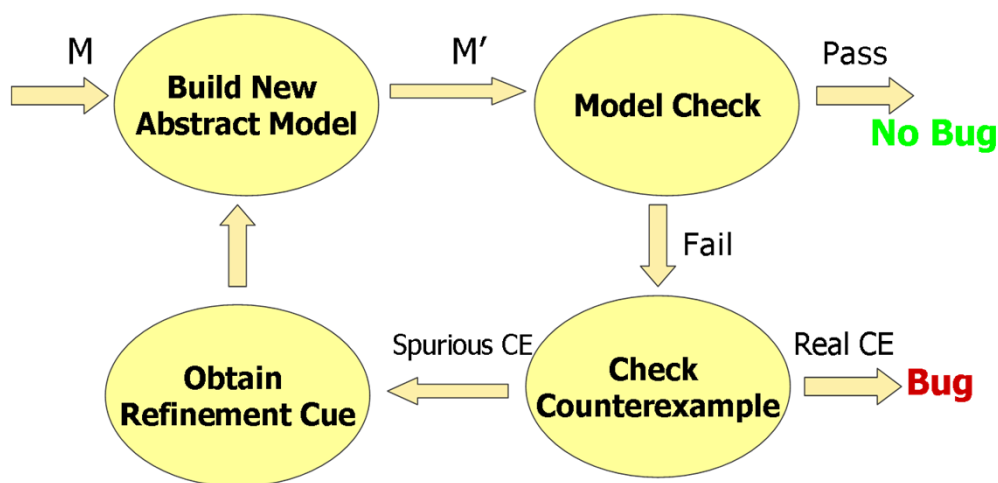
EE382V: SoC Design, Lecture 20

© J. A. Abraham

59

Counterexample Guided Refinement

- **CEGAR tool**



EE382V: SoC Design, Lecture 20

© J. A. Abraham

60

Specialized Slicing for Verification

- **Amorphous Slicing**

- Static slicing preserves syntax of program
- Amorphous Slicing does not follow syntax preservation
- Semantic property of the slice is retained
- Uses rewriting rules for program transformation

Example of Amorphous Slicing

```
begin
  i = start;
  while (i <= (start + num))
  {
    result = K + f(i);
    sum = sum + result;
    i = i + 1;
  }
end
```

LTL Property: $G \text{ sum} > K$

Slicing Criterion: (end, {sum, K})

Example of Amorphous Slicing

Amorphous Slice:

```
begin
  sum = sum + K + f(start);
  sum = sum + K + f(start + num);
end
```

Program Transformation rules applied

- Induction variable elimination
- Dependent assignment removal
- Amorphous Slice takes a fraction of the time as the real slice on SPIN