Coverage-Driven Functional Verification

Prof. Chien-Nan Liu
Institute of Electronics
National Chiao-Tung Univ.

教育部
前瞻晶片系統設計人才培育先導型計畫
DAT 學程核心課程精進計畫
DAT 聯盟

Outline

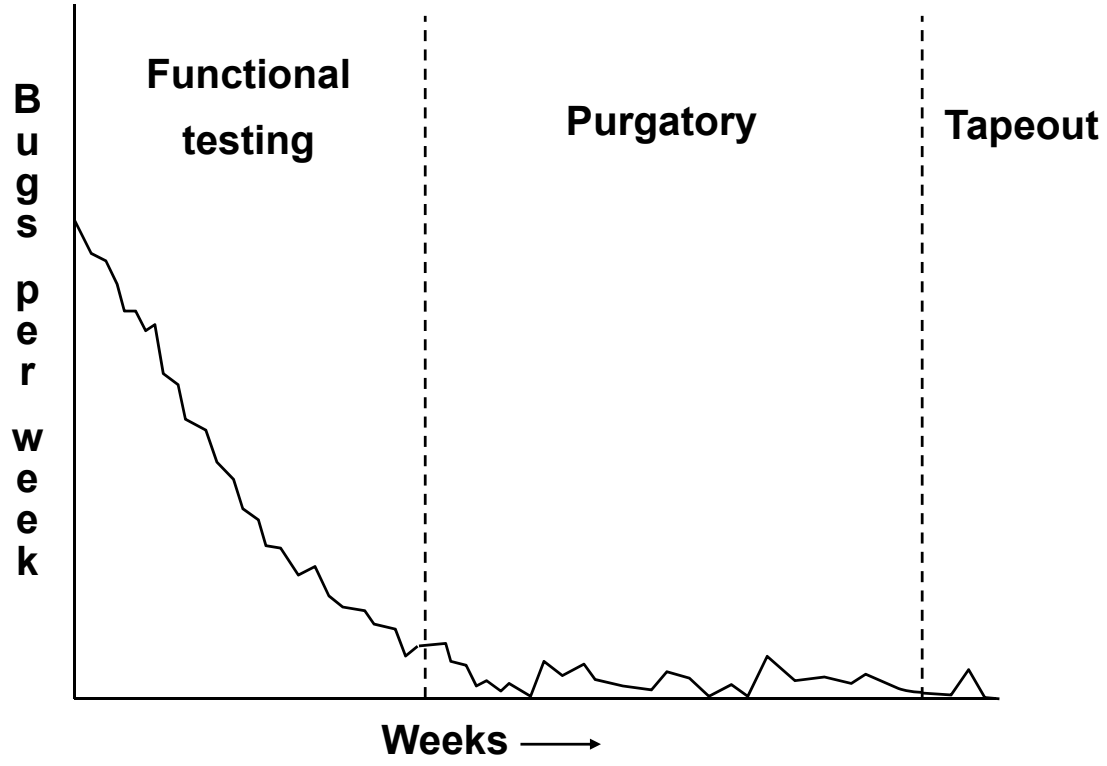
- Introduction
- Functional Coverage Metrics
- Testbench Generation
- Coverage-Assisted Debugging

Verification Complexity

- For a single flip-flop:
  - Number of states = 2
  - Number of test patterns required = 4
- For a Z80 microprocessor (~5K gates)
  - Has 208 register bits and 13 primary inputs
  - Possible state transitions = $2^{\text{bits+inputs}} = 2^{221}$
  - At 1M IPS would take 10^{53} years to simulate all transitions
- For a chip with 20M gates
  - ??????

*IPS = Instruction Per Second

Typical Verification Experience



When is Verification Complete ?

- Some answers from real designers:
  - When we run out of time or money
  - When we need to ship the product
  - When we have exercised each line of the HDL code
  - When we have tested for a week and not found a new bug
  - ***We have no idea!!***
- Designs are often too complex to ensure full functional coverage
  - The number of possible vectors greatly exceeds the time available for test

The Famous Pentium Bug

- Summary of the Pentium division bug
  - Pentium was Intel's mainstream microprocessor
    - 3.3 Million transistors
  - Early versions all had error in floating point division hardware
    - 5 missing transistors, fixed with change to single mask
  - Disclosed largely via Internet
  - Intel ultimately offered replacements to everyone
    - $475 Million charge from 4Q94 revenue

Why Did not Intel Discover it ?

- Standard steps in verifying design
  - Simulate many cases on high-level software model
  - Simulate/emulate final logic design
    - Hardware emulators costing $Millions
    - Run complete chip model at ~ 100Hz (< 10^{-6} X real time)
  - Run tests on initial production chips
    - Feasible to run billions of tests
    - Should have caught error here
  - 1 trillion (10^{12}) test vectors for Pentium chip
- Observations
  - Hard to test all aspects of such a complex system

Simulation-Based Verification

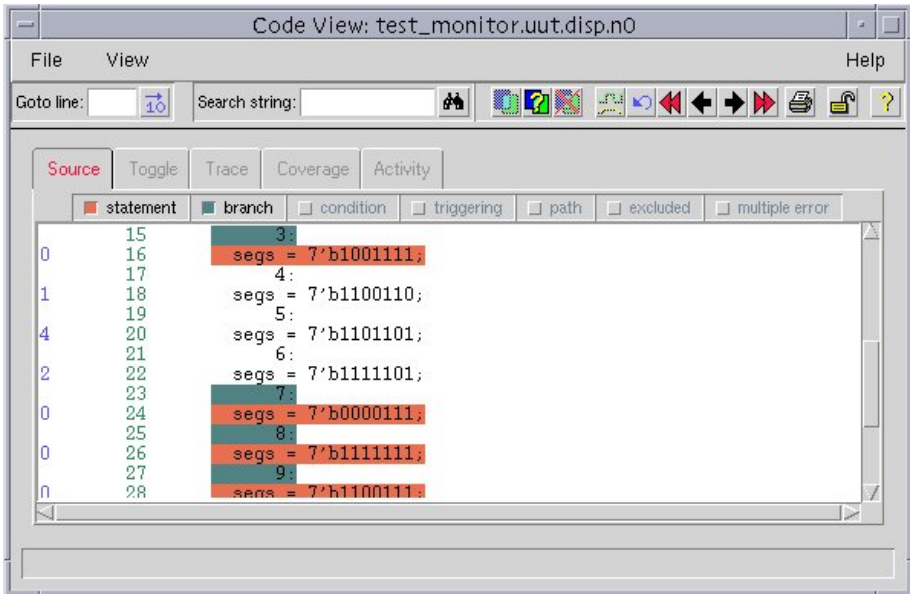
- Still the primary approach for functional verification
  - In both gate-level and register-transfer level (RTL)
- Test cases
  - User-provided (often)
  - Randomly generated
- Hard to gauge how well a design has been tested
  - Often results in a huge test bench to test large designs
- Near-term improvements
  - Faster simulators
    - Compiled code, cycle-based, emulation, …
  - Testbench tools
    - Make the generation of pseudo-random patterns better/easier
- Incremental improvements won't be enough

Coverage-Driven Verification

- Coverage reports can indicate how much of the design has been exercised
  - Point out what areas need additional verification
- Optimize regression suite runs
  - Redundancy removal (to minimize the test suites)
  - Minimizes the use of simulation resources
- Quantitative sign-off criterion
  - A good guidance but cannot guarantee 100% error-free
- Easy to use, low-complexity
- **Verify more but simulate less**

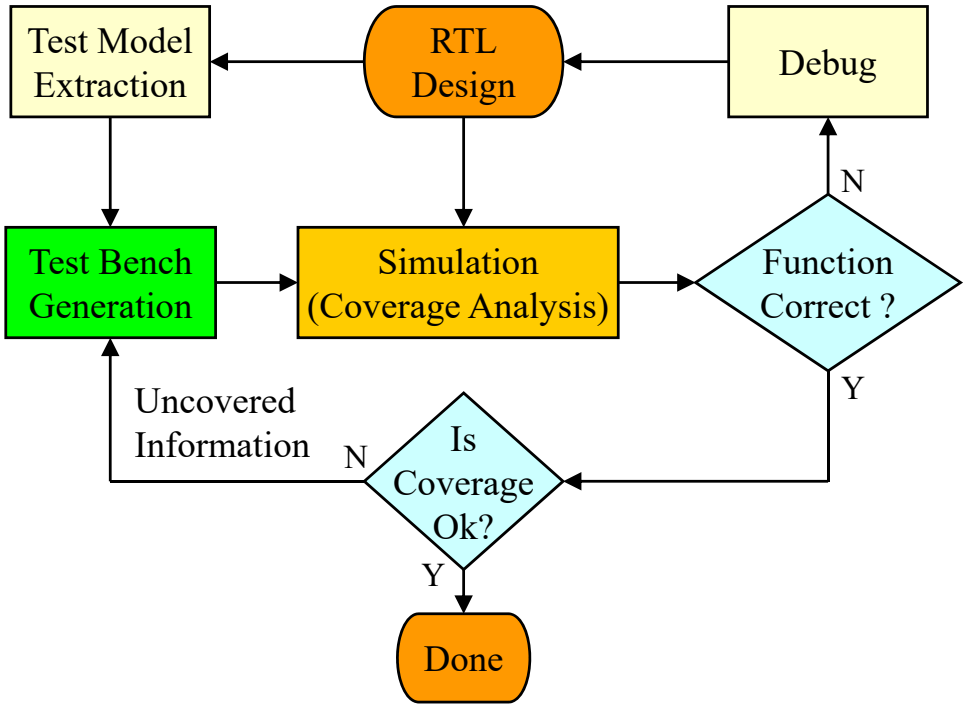
Coverage Analysis Results

- Verification Navigator (TransEDA)

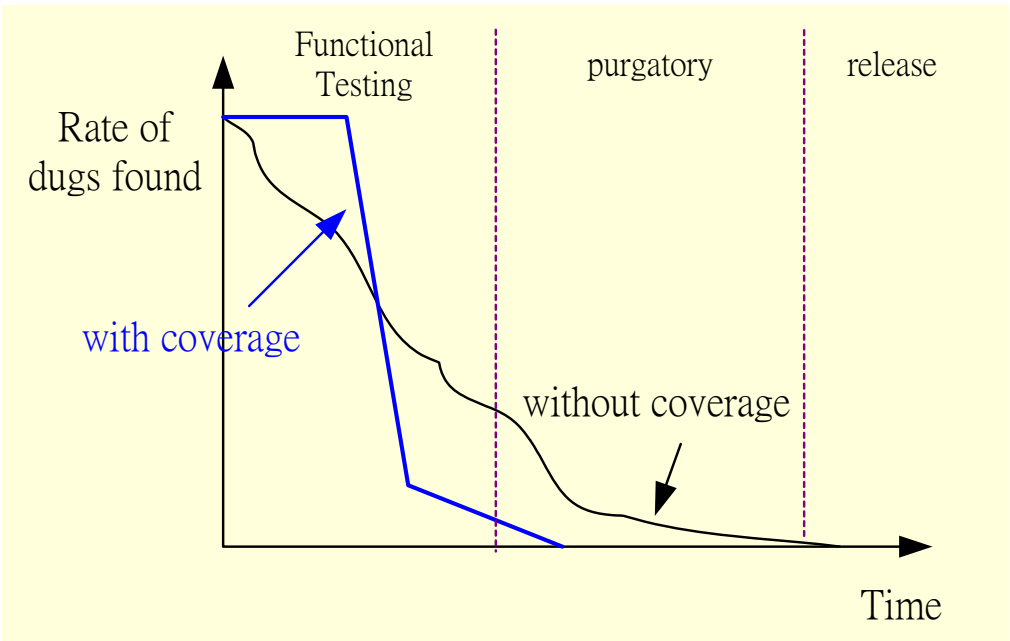


Untested code line will be highlighted !!

Typical Coverage-Driven Verification

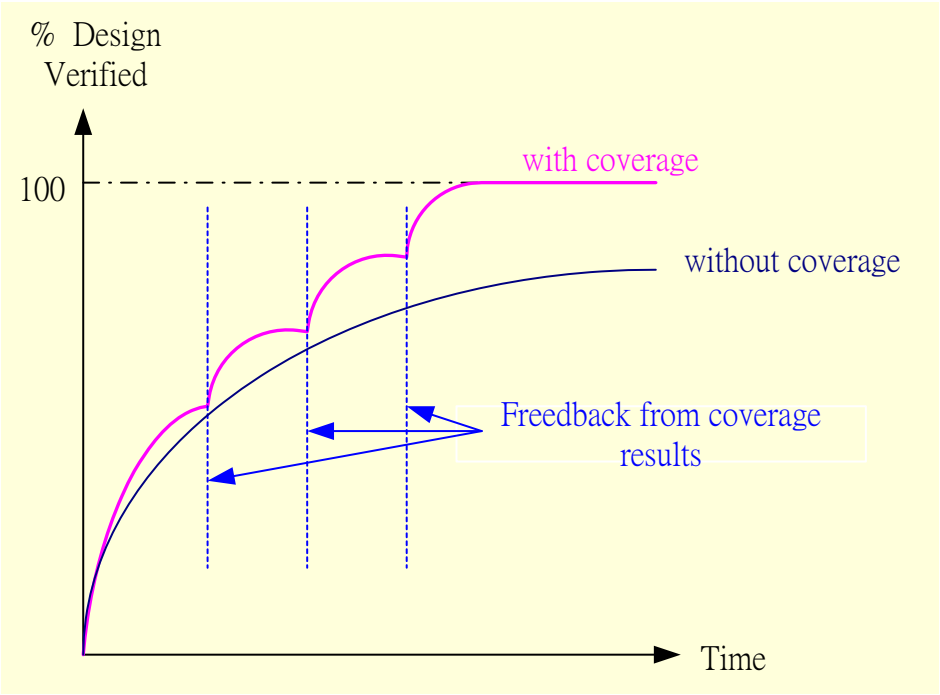


The Rate of Bug Detection



source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

Improvements on Verification



source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

Outline

- Introduction
- Functional Coverage Metrics
- Testbench Generation
- Coverage-Assisted Debugging

Functional Coverage Metrics

- Code coverage
  - Statement coverage
  - Block coverage
  - Decision coverage
  - Path coverage
  - Expression coverage
  - Event coverage
  - Toggle coverage
  - Variable coverage
- FSM coverage
  - Conventional FSM coverage
  - SFSM coverage
- Other coverage
  - Observability-based code coverage
  - Assertion coverage
  - User-defined functional coverage

Statement Coverage

- Measuring how many statements have been exercised in the simulation

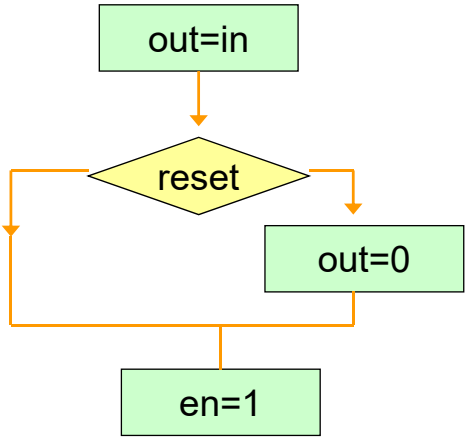
```
always @ ( posedge clk ) begin
❶ out = in;
❷ if ( reset ) ❸ out = 0;
❹ en = 1;
end
```

There are ***4*** independent statements.

Decision Coverage

- Measure the coverage of each branch in the *if* and *case* statements

```
always @ ( posedge clk ) begin
    out = in;
    if ( reset )    out = 0;    else ?
    en = 1;
end
```



Implied *else* is also measured.

Path Coverage

- Measuring the coverage of all possible paths through the HDL code
- Similar to decision coverage
- Handle multiple sequential decisions

```
if ( a )  begin
        …
end
if ( b )  begin
        …
end
```

There are ***4*** paths through the code fragment:

(a, b) = 00, 01, 10, 11

Two cases (00, 11) can reach 100% decision coverage

Expression Coverage

- Measuring how the variables or sub-expressions in conditional statements are evaluated
- Only examines the variables or sub-expressions combined by logical operators
  - a & b (boolean operator)
  - a && b (logical operator)
- Three major categories:
  - Multiple sub-condition coverage
  - Basic sub-condition coverage
  - Focused expression coverage

Multiple Sub-Condition Coverage

- Multiple Sub-Condition coverage is the most popular analysis method
- There are 2^N cases must be checked

x = (a==1) || (b & c) && (~d)

a==1	b&c	~d
0	0	0
0	0	1
0	1	0
0	1	1
1	0	0
1	0	1
1	1	0
1	1	1

0 : True
1 : False

8 cases will be checked

Basic Sub-Condition Coverage

- Each term in the sub-expression must be checked for both true and false

```
x = (a==1) || ( b==1) && ( ~c)
```

There are 6 cases will be checked :
(a==1) is true
(a==1) is false
(b==1) is true
(b==1) is false
(~c) is true
(~c) is false

Focused Expression Coverage

- Only N+1 patterns (N is input number)
- Set other inputs to pass the focused value
  - For pass: AND, NAND → set to 1; OR, NOR → set to 0

```
x = (a && b) || c
```

Check points:
1. input a set to 0 : [0, 1, 0] =>0
2. input a set to 1 : [1, 1, 0] =>1
3. input b set to 0 : [1, 0, 0] =>0
4. input b set to 1 : [1, 1, 0] =>1
5. input c set to 0 : [0, 0, 0] =>0 or [0, 1, 0] =>0 or [1, 0, 0] =>0
6. input c set to 1 : [0, 0, 1] =>1 or [0, 1, 1] =>1 or [1, 0, 1] =>1

For 100% coverage , only 4 patterns is required
(0,1,0) (1,1,0) (1,0,0) and (0,0,1) or (0,1,1)or (1,0,1)

Toggle Coverage

- One of the typical coverage measurement in hardware design
- Measures the bits of logic being toggled during simulation
- A rough measurement for functional verification
  - Often used in gate-level simulation only

Variables	0 → 1	1 → 0
A	Toggled	Toggled
B	Not yet	Toggled
C	Not yet	Not yet
…	…	…

Observability-Based Code Coverage

- Take the observability issue into consideration
- Put a tag Δ on each assignment to represent possible errors
- Watch the outputs for those tags during simulation
  - Need to define the tag propagation rules

```
always @ ( posedge clk ) begin
Δ out = in;
   if ( reset ) Δ out = 0;
Δ en = 1;
end
```

[Devadas 96, Fallah 98]

Probabilistic Observability Measure

- The error on *counter* can be observed only when *counter* < 2
  - Not occurred very often
- Tag will treat it as "observable" even the likelihood is low
  - Use a probability between 0~1 to provide more precise estimation

```
    input [2:0] PI1; input [3:0] PI2;
    input reset, clk;
    output [4:0] PO1;
    reg [3:0] counter;  wire [4:0] a;

1:  assign a = PI1*4;
    always@(counter or PI1 or PI2)
2:    if( counter < PI2 )
3:      PO1 = 1;
      else
4:      PO1 = a;
    always@(posedge clk or posedge reset)
5:    if(reset)
6:      counter = 1;
      else
7:      counter = counter + 10;
```

(a) HDL Code

```
#1 reset=1; PI1=4; PI2=2;
#1 reset=0;
#5 finish;
```

(b) Input stimulus

```
Time = 1
counter=1,  "counter<PI2",PO1=1
Time = 5
counter=11, "counter>PI2",PO1=16
```

2

3

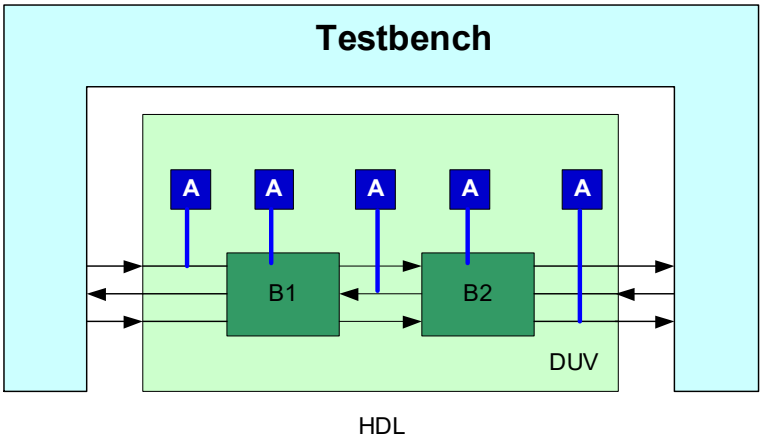
not changed ?

(c) Simulation result

Ref: T. Jiang, C. Liu, and J. Jou, "Observability Analysis on HDL Descriptions for Effective Functional Verification", *IEEE TCAD*, pp. 1509-1521, Aug. 2007.

Assertions Coverage

- Assertions are often added in the designs to observe the internal signals and watch for forbidden behavior
  - Popular in modern design flow
  - Can be viewed as the concerned behavior of designers
- Check if all assertions have been evaluated in the simulation
  - If all important behaviors are exercised
  - More related to real functionality



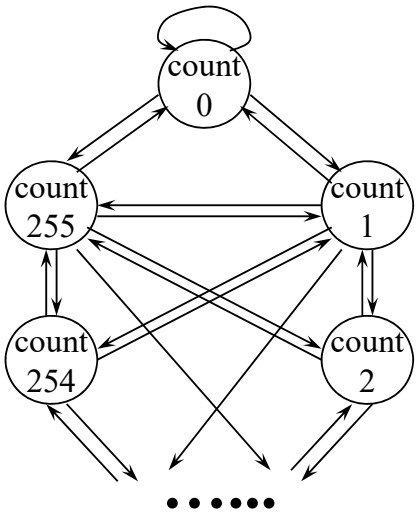
Functional Coverage

- The language-based code coverage
  - Easy to use because it traverses the language structures only
  - Some design errors may not be uncovered
- Need metrics that can measure the hardware behavior of a design
  - Hard to know the functionality it has through code analysis only
  - Most commercial tools allow users to define the functionality and count the coverage of those behaviors
- FSM coverage is another option
  - The sequences of actions can also be verified
  - State explosion problem may occur in FSMs
  - SFSM coverage is proposed [Liu 01]

Conventional FSM Coverage

- The measurement of state visitation and state transitions

```
module counter (clk, rst, load, in, count) ;
input         clk, rst, load ;
input   [7:0] in ;
output [7:0] count ;
reg     [7:0] count ;

always @(posedge clk) begin
    if (rst) count = 0 ;
    else if (load) count = in ;
    else if (count == 255) count = 0 ;
    else count = count + 1 ;
end
endmodule
```

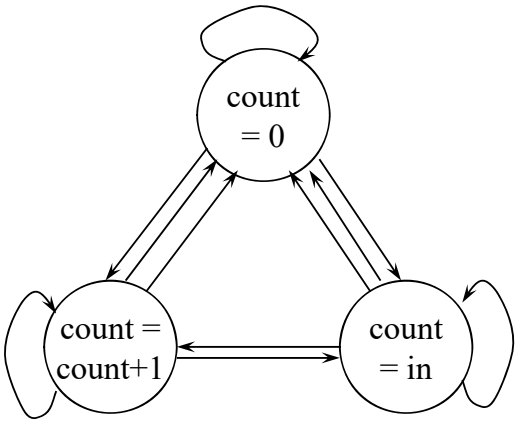


256 states 66047 transitions

Semantic FSM Coverage

- Merge the states with same behavior into one semantic state to reduce the complexity

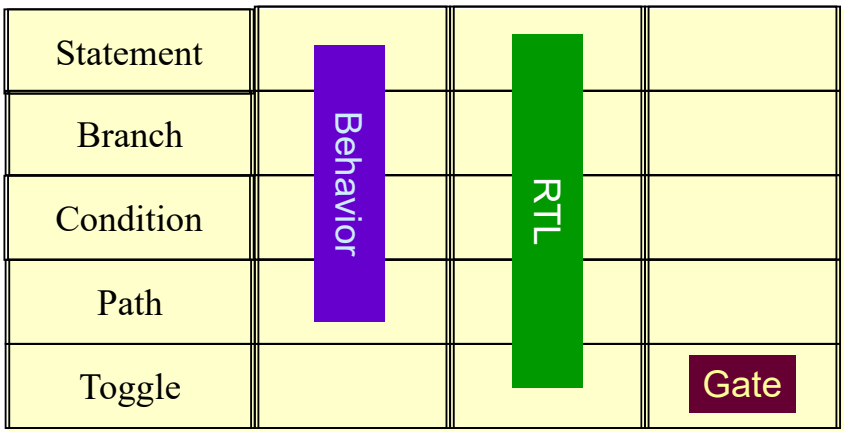
```
module counter (clk, rst, load, in, count) ;
input          clk, rst, load ;
input   [7:0] in ;
output [7:0] count ;
reg      [7:0] count ;

always @(posedge clk) begin
    if (rst) count = 0 ;
    else if (load) count = in ;
    else if (count == 255) count = 0 ;
    else count = count + 1 ;
end
endmodule
```



3 states 11 transitions

Code Coverage Guidelines (1/2)

- Coverage measurements should be used at each stage in design process



source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

Code Coverage Guidelines (2/2)

- Coverage measurements should be used at each stage of the RTL design

	Module Design	Sub-System Integration	System Integration
Statement	ˇ	ˇ	ˇ
Branch	ˇ	ˇ	ˇ
Condition	ˇ	ˇ	ˇ
Path	ˇ		
Toggle	ˇ	ˇ	ˇ

source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

Typical Coverage Targets

- Choose appropriate coverage measurement targets

Measurement	Coverage Test (%)
Statement	100
Branch	100
Condition	60~100 *
Path	> 50
Toggle	100

* Depending on coverage tool

source : "Verification Methodology Manual For Code Coverage In HDL Designs" by Dempster and Stuart

References

- Srinivas Devadas, Abhijit Ghosh, and Kurt Keutzer, “An Observability-Based Code Coverage Metric for Functional Simulation”, Int’l Conf. on Computer Aided Design, Nov. 1996.
- Farzan Fallah, Srinivas Devadas, and Kurt Keutzer, “OCCOM: Efficient Computation of Observability-Based Code Coverage Metric for Functional Simulation”, 35th Design Automation Conf., June 1998.
- Dean Drako and Paul Cohen, “HDL Verification Coverage”, Integrated Systems Design Magazine, June 1998.
- Jing-Yang Jou and Chien-Nan Jimmy Liu, "Coverage Analysis Techniques for HDL Design Validation", the 6th Asia Pacific Conf. on cHip Design Languages (APCHDL'99), October 1999. (Invited Paper)
- C.-N. Liu and J.-Y. Jou, “Efficient Coverage Analysis Metric for HDL Design Validation”, IEE Proceedings - Computers and Digital Techniques, vol. 148, no. 1, pp. 1-6, Jan. 2001.
- David Dempster and Michael Stuart, “Verification Methodology Manual for Code Coverage in HDL Designs”, Teamwork International, Hampshire, United Kingdom, May 2000.
- Reference manuals of Verification Navigator™, TransEDA PLC.
- T. Jiang, C. Liu, and J. Jou, “Observability Analysis on HDL Descriptions for Effective Functional Verification”, *IEEE TCAD*, pp. 1509-1521, Aug. 2007.

Outline

- Introduction
- Functional Coverage Metrics
- Testbench Generation
- Coverage-Assisted Debugging

Testbench Design

- Test bench generation is time-consuming and mostly done manually
  - The user-provided functional vectors are often used
- Auto or semi-auto stimulus generator is preferred
- **Automatic response checking** is highly recommended
  - Hard to understand the meanings of the automatically generated input patterns
- Generating functional vectors automatically for HDL designs is a difficult problem
  - Because of the various descriptions in HDL
  - Pure random data is useless
- Only semi-auto approaches are available now

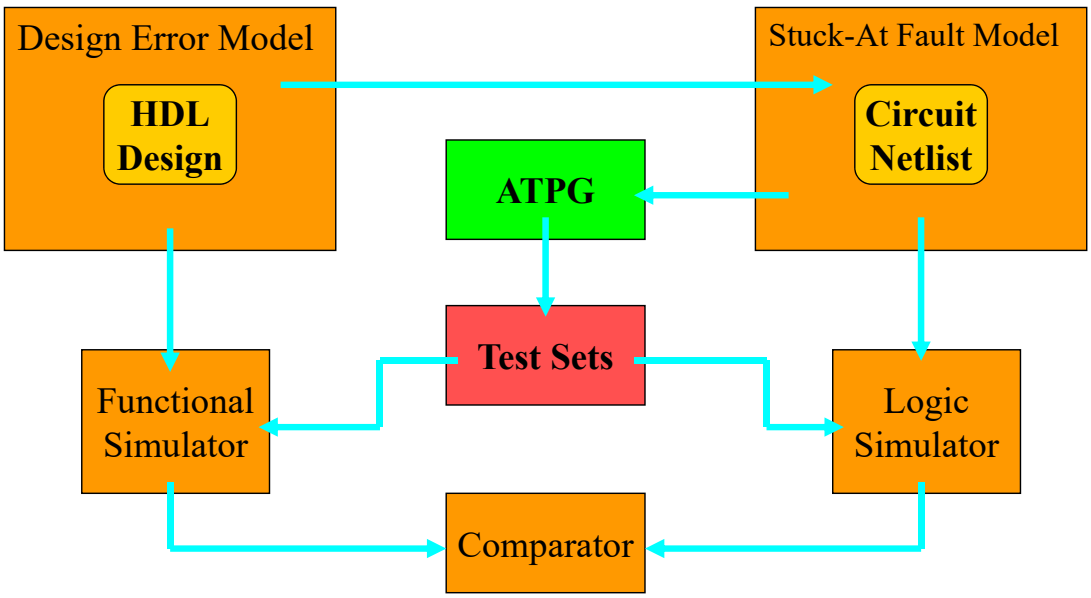
Existing Approaches

- Most techniques are *ad hoc* for specific applications
  - Especially for processor designs [Chandra 95, Bhagwati 94, Aharon 95]
- For general cases, only semi-auto tools are available
  - A platform (language) providing powerful constructs for generating stimulus and checking response
  - VERA, Specman Elite, …
- General approaches in academia can be roughly classified into 3 categories
  - ATPG-based approaches [Chung 93, Kang 94, Assad 95]
  - Code-coverage-based approaches [Cheng 93, Fallah 98]
  - FSM-based approaches [Cheng 92, Cabodi 97, Liu 01]

ATPG-Based Approaches

- Some specific design error models are defined
- The translations from the design error model to the stuck-at fault model are proposed
- Use manufacturing ATPG tools to generate patterns for those translated stuck-at faults
- Take the "observability" issues of those design errors into consideration
- The relationship between the design error models and the RTL functionality is weak
  - Hard to define all possible design errors

Flows for ATPG-Based Approaches

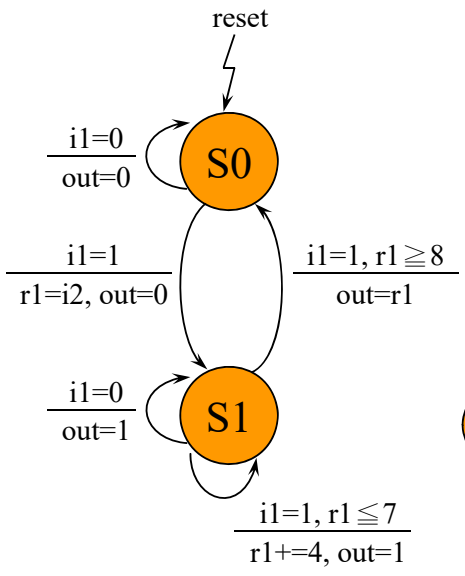


Pattern Generation with EFSM

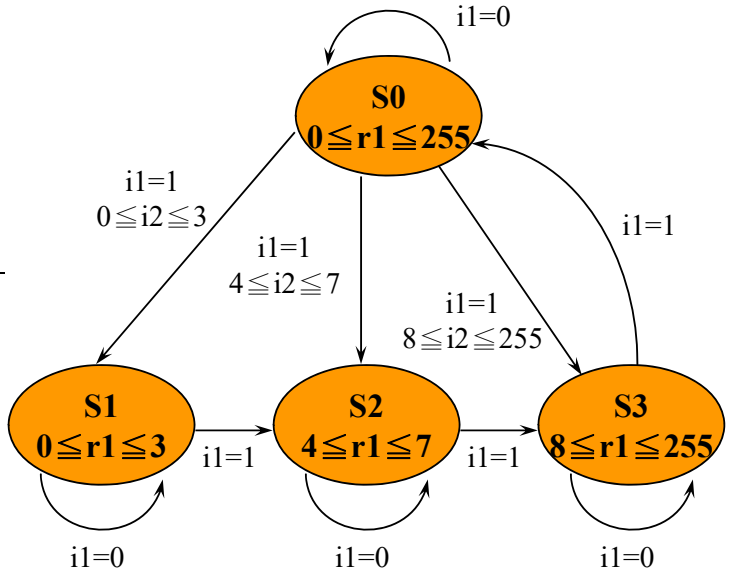
- All statements of a HDL design totally appear in EFSM model
  - Traversing all states and transitions in EFSM model can cover all statements
- A simple tour generator for traversing EFSMs completely can generate the desired patterns
  - Target **statement coverage**
- Cannot be extended to more complex metrics
  - Statement coverage is only the most basic requirement

EFSM Example [Cheng 93]

Initial STG



Stable Block Transition Graph



*Partition all possible values of data registers into groups (states)

Generate Patterns by HSAT Solver

[Fallah 98]

- HSAT (Hybrid-SAT) method
  = SAT (Bit level) + ILP (Word level)
- SAT (Boolean Satisfiability) problem
  - Find an input assignment that produces appropriate signal value for a boolean equation
  - Satisfy a conjunctive normal form (CNF) expression

ex: $Z = X \cdot Y \Rightarrow (Z+\overline{X})(Z+\overline{Y})(X+\overline{Y}+\overline{Z})$ ← CNF expr.

In order to produce a 1 at output
$\Rightarrow$ replace Z with 1
$\Rightarrow$ satisfy (X) (Y) (1)
$\Rightarrow$ X = 1, Y = 1

Generate Patterns by HSAT Solver

- ILP (Integer Linear Programming) problem
  = solving a set of linear arithmetic constraints (LAC)

ex: $C = A+B \Rightarrow A+B-C \le 0$ *and* $A+B-C \ge 0$
$C = A*k \Rightarrow C-kA \le 0$ *and* $C-kA \ge 0$
$z = A>B \Rightarrow A-B+Uz \le 0$ *and* $A-B+U(1-z) \ge 0$
(z is a boolean variable, $U = 2^n$ where n = max. number of bits)

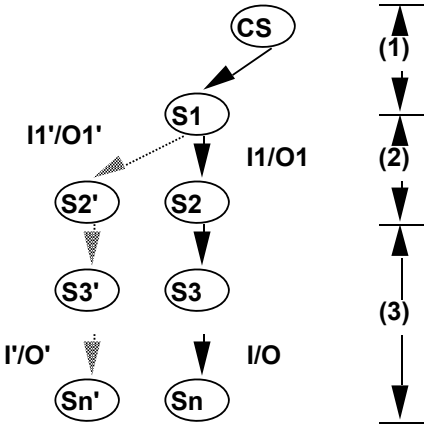
- Generate the required patterns by solving those constraints
  - Logic expressions → CNF constraints
  - Arithmetic expressions → LAC constraints
  - Existing software packages are available for SAT and LP
- Solving SAT and LP is a time-consuming process

FSM-Based Approaches

- Target on the state transitions of a FSM
  - State transition fault (STF) model [Cheng 92]
- Closer to verify the real functionality
  - State transition graph ≈ functionality representation
- The STG size is a big problem
  - Often too large to be traversed exhaustively
- A BDD-based technique with automatic partitioning is proposed to cope memory issues
  - Interacting FSM (IFSM) model [Liu 01]

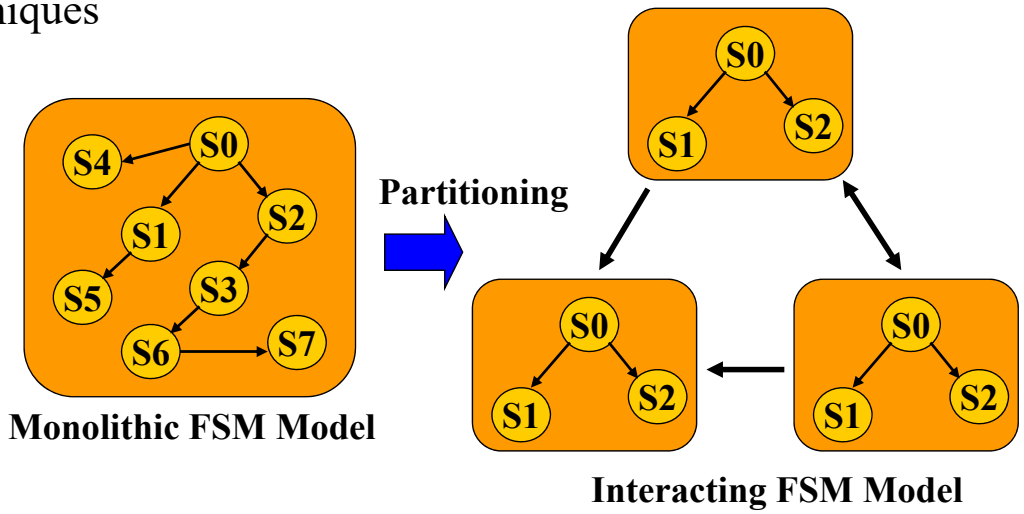
Test Sequence for a Single STF

- A test sequence is composed of three subsequences:
  1) Initialization sequence
  2) Input pattern causing the faulty transition
  3) State-pair differentiating sequence between good and faulty states
- If the output label of the target transition is corrupted (O1 ≠ O1'), subsequence (3) is not needed

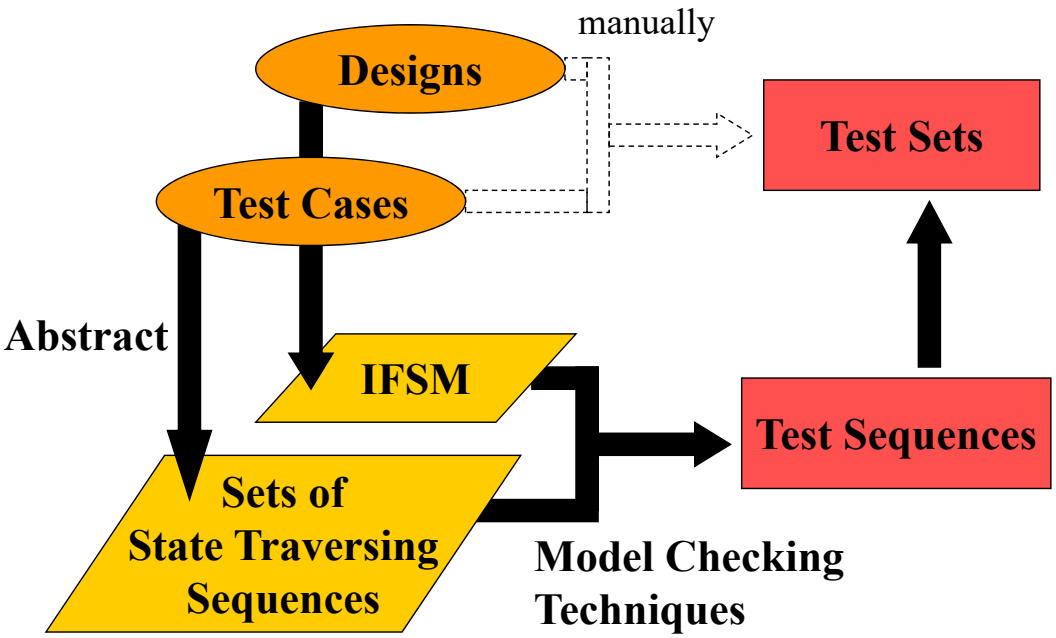


Interacting FSM Model

- Use model checking techniques (e.g. BDD, image computation) to generate input patterns for triggering specific state transition
- Instead of the monolithic FSM model, the Interacting FSM (IFSM) model is used to solve the memory issues in formal techniques



Scheme of IFSM-Based Approach



Semi-Auto Approaches

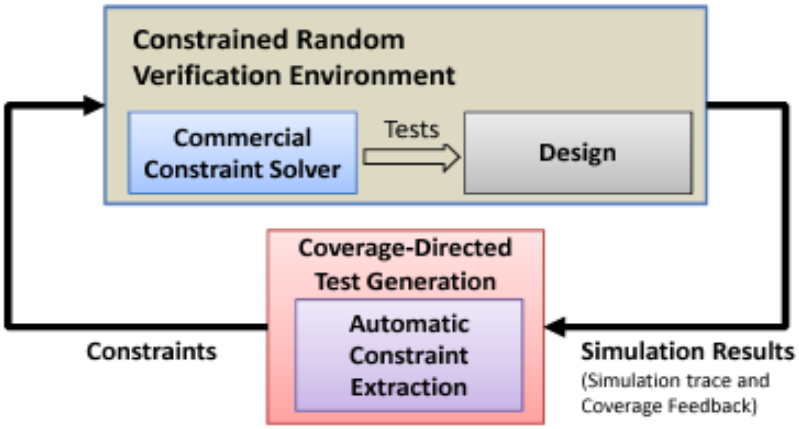
- Key idea: constrained random patterns
- Generator: only generate meaningful patterns
  - Ex: keep A in [10 … 100]; keep A + B == 120;
  - Variations can be directed by weighting options
  - Ex: 60% fetch, 30% data read, 10% write
- Predictor: generate the estimated outputs
  - Require a behavioral model to check the answers automatically
  - Not designed by same designers to avoid the same errors
- HVL (Hardware Verification Language) is proposed to help describe the required testbench
  - e-language, VERA, SystemVerilog, …
- Widely used on industrial cases now

Challenges of Semi-Auto Approach

- The total time needed for constrained random verification depends on both the **performance** of the constraint solver and the stimulus **distribution**
- Inefficient constraint solver → increase time per cycle
  - A fundamental topic in both formal verification and constrained random simulation
- Mismatched distributions → increase number of cycles
  - If coverage distribution is unknown → maximize entropy (uniform)
  - If feedback from coverage analysis is available → dynamically adjust input constraints

Coverage-Directed Test Generation

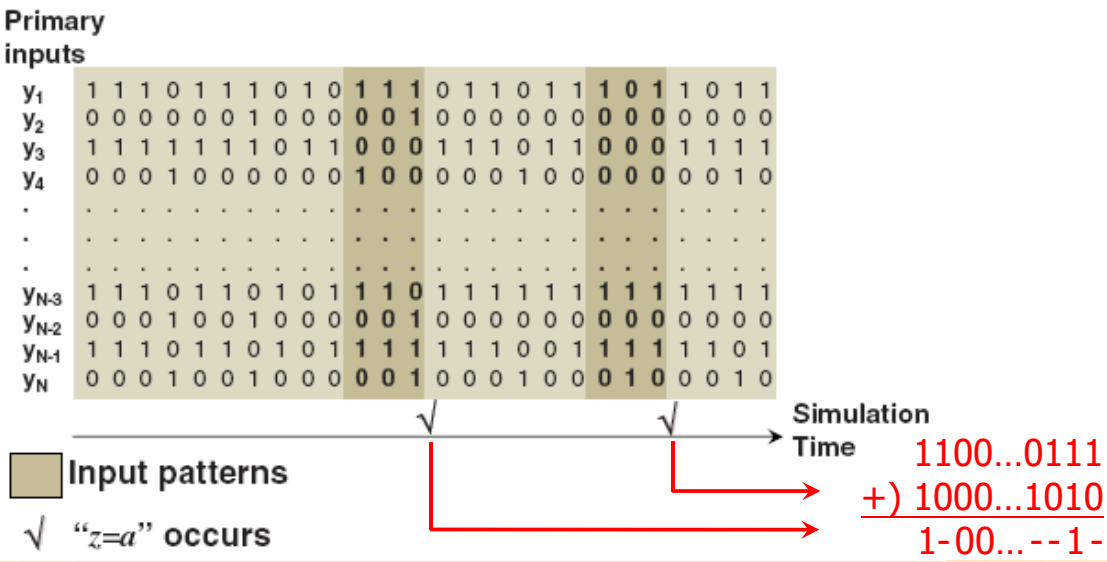
- Random patterns are hard to cover all functionality
  - Although constrained patterns have been more meaningful
- Dynamically adjust the input constraints for the random generator to cover the untested part more quickly



Ref: Onur Guzey, Li-C. Wang, "Coverage-directed test generation through automatic constraint extraction", *IEEE High Level Design Validation and Test Workshop*, 2007.

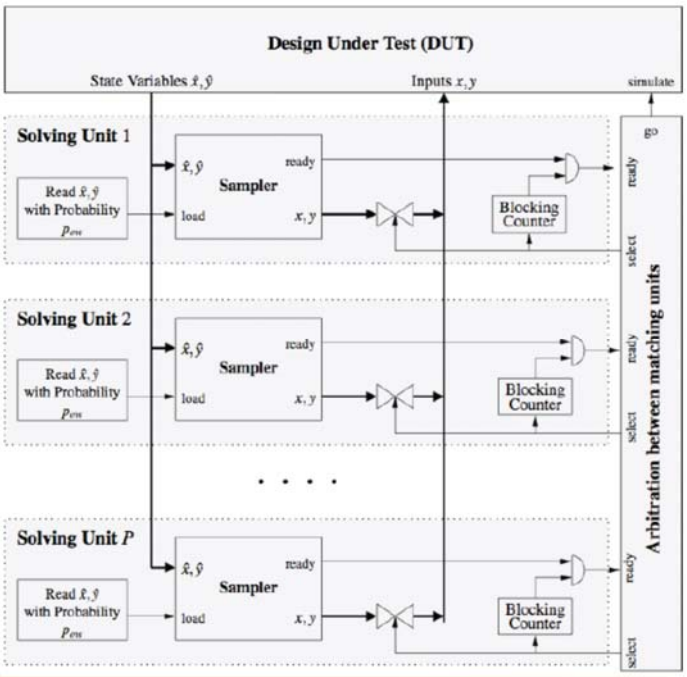
Coverage-Directed Test Generation

- Key idea: increase signal controllability
  - Coverage targets are translated to the internal signals that need to be controlled
  - Constraints are automatically extracted from simulation data



Hardware Implementation

- For FPGA-based and emulation-based simulation, constraint solver can be built in the hardware, too
  - Better efficiency
  - Can simulate more patterns due to faster speed
  - Suitable for fixed input constraints



References (1/2)

- A. Chandra, V. Iyengar, D. Jameson, R. Jawalekar, I. Nair, B. Rosen, M. Mullen, J. Yoon, R. Armoni, D. Geist, and Y. Wolfsthal, "AVPGEN – A Test Generator for Architecture Verification", IEEE Trans. on VLSI, vol. 3, no. 2, pp. 188-200, June 1995.
- Vishal Bhagwati and Srinivas Devadas, "Automatic Verification of Pipelined Microprocessors", 31st Design Automation Conf., June 1994.
- Aharon Aharon, Dave Goodman, Moshe Levinger, Yossi Lichtenstein, Yossi Malka, Charlotte Metzger, Moshe Molcho, and Gil Shurek, "Test Program Generation for Functional Verification of PowerPC Processor in IBM", 32nd Design Automation Conf., June 1995.
- P. Chung, Y. Wang, and I. Hajj, "Diagnosis and Correction of Logic Design Errors in Digital Circuits", 30th Design Automation Conf., June 1993.
- Sungho Kang and Stephen A. Szygenda, "Design Validation: Comparing Theoretical and Empirical Results of Design Error Modeling", IEEE Design & Test of Computers, vol. 11, no. 1, pp. 18-26, Mar. 1994.
- Hussain Al-Asaad and John P. Hayes, "Design Verification via Simulation and Automatic Test Pattern Generation", Int'l Conf. on Computer Aided Design, Nov. 1995.
- K.-T. Cheng and A. S. Krishnakumar, "Automatic Functional Test Generation Using the Extend Finite State Machine Model", 30th Design Automation Conf., June 1993.
- Farzan Fallah, Srinivas Devadas, and Kurt Keutzer, "Functional Vector Generation for HDL models Using Linear Programming and 3-Satisfiability", 35th Design Automation Conf., June 1998.

References (2/2)

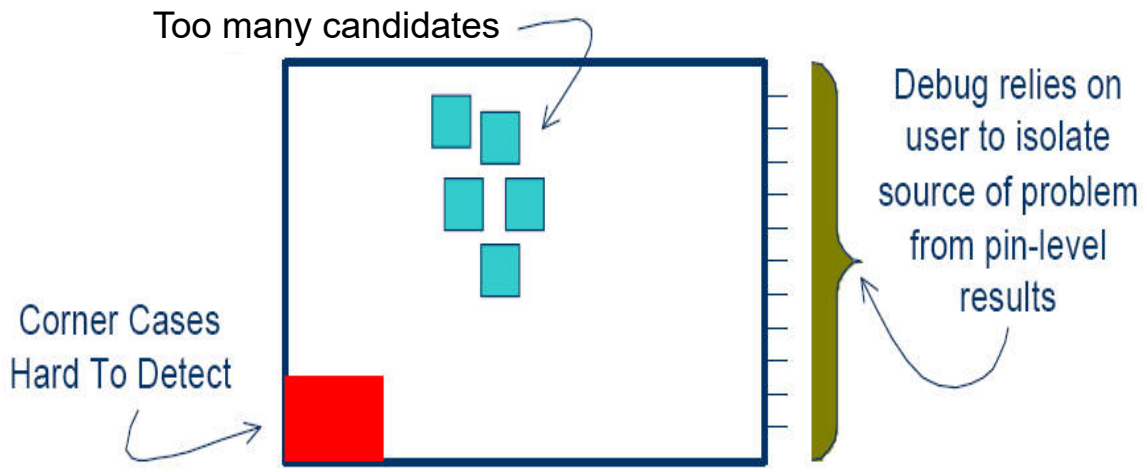
- K.-T. Cheng, and Jing-Yang Jou, "A Functional Fault Model for Sequential Machines", IEEE Transactions on CAD of Integrated Circuits and Systems, September 1992.
- Gianpiero Cabodi and Paolo Camurati, “Symbolic FSM Traversals Based on the Transition Relation”, IEEE Trans. on Computer-Aided Design, vol. 16, no. 5, pp. 448-457, May 1997.
- Chien-Nan Jimmy Liu, Chia-Chih Yen, and Jing-Yang Jou, “Automatic Functional Vector Generation Using the Interacting FSM Model”, Int’l Symposium on Quality Electronic Design, Mar. 2001.
- Braun, M.; Fine, S.; Ziv, A., ”Enhancing the Efficiency of Bayesian Network Based Coverage Directed Test Generation”, IEEE High-Level Design Validation and Test Workshop, 2004.
- Samarah, A.; Habibi, A.; Tahar, S.; Kharma, N., “Automated Coverage Directed Test Generation Using a Cell-Based Genetic Algorithm”, IEEE High-Level Design Validation and Test Workshop, 2006.
- Hsiou-Wen Hsueh; Eder, K., “Test Directive Generation for Functional Coverage Closure Using Inductive Logic Programming”, IEEE High-Level Design Validation and Test Workshop, 2006.
- Onur Guzey, Li-C. Wang, “Coverage-Directed Test Generation through Automatic Constraint Extraction”, IEEE High Level Design Validation and Test Workshop, 2007.

Outline

- Introduction
- Functional Coverage Metrics
- Testbench Generation
- Coverage-Assisted Debugging

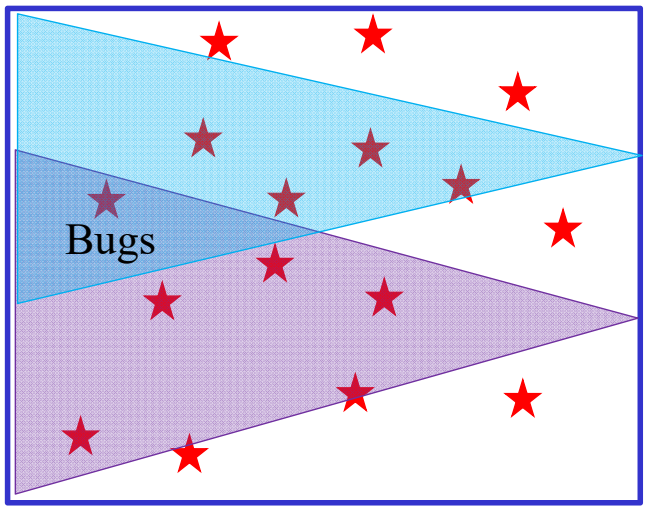
Debugging in Simulation

Use the simulation results only is hard to locate the design bugs from thousands of codes



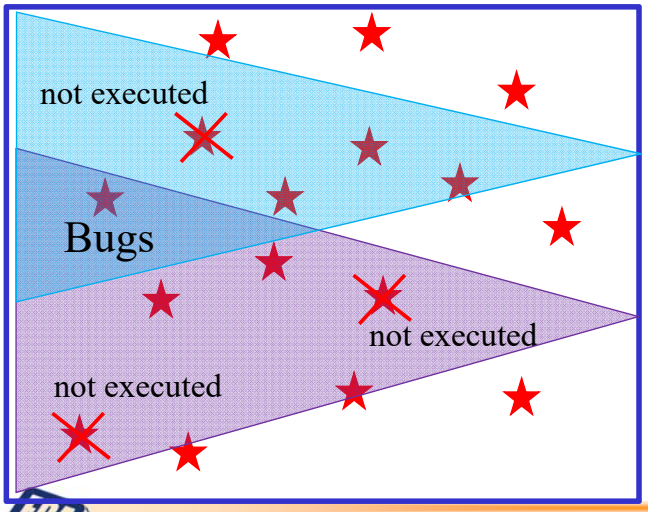
Debugging Priority

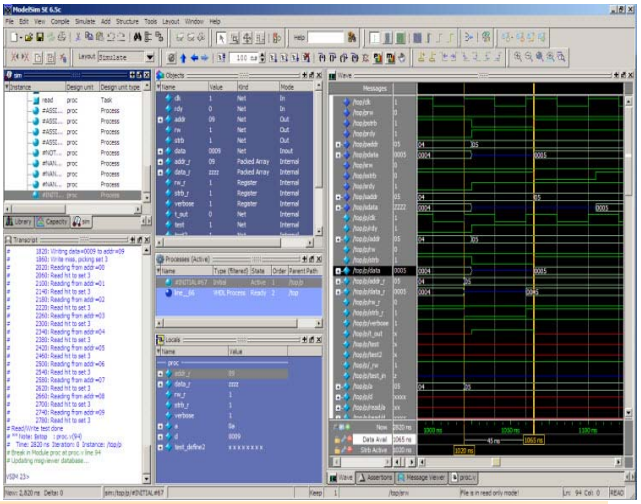
- By tracing the fanin cones of erroneous outputs, a debugging priority list can be provided to find the real error faster
  - Non-related candidates will be screened out
  - The error candidates that appear in more fanin cones are more possible to be the error sources
    → higher priority
- Not very accurate; just a rough estimation to provide some "hints" for users
  - Some commercial tools have provide such capability (Debussy of SpringSoft, …)



Coverage-Assisted Debugging [Jiang 02]

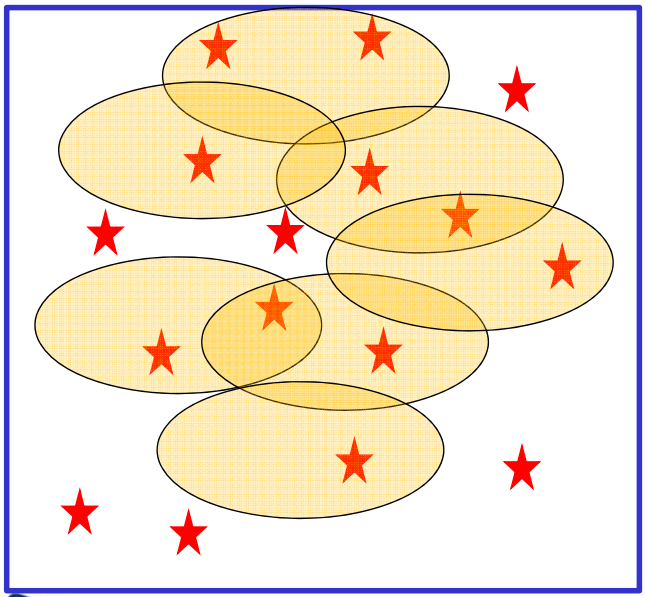
- Although the codes in the same logic cone are correlated to the outputs, not all of them are executed at that time
  - Coverage reports provide the execution status of each candidate
  - Non-executed candidates will be screened out to further reduce the possible error space





Assertions can also help …

Fired assertions can help to further locate the error location



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

RTL Design

Assertion Monitor Library

```
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
```

```
Debussy Release 5.2v14 (SOLARIS) Verilog-XL - 10/23/2002 compile
FSDB file verilog.fsdb is created
Start dumping the top scope(assert_change), layer(0),
Start dumping the top scope(test), layer(0),
End of dumping
OVL_ERROR : ASSERT_ALWAYS : The Counter out of range : : severity 9 : time 35 : test.traff.c1.counter1.ovl_error
OVL_ERROR : ASSERT_ALWAYS : The Counter out of range : : severity 9 : time 37 : test.traff.c1.counter1.ovl_error
OVL_ERROR : ASSERT_ALWAYS : The Counter out of range : : severity 9 : time 39 : test.traff.c1.counter1.ovl_error
OVL_ERROR : ASSERT_ALWAYS : The Counter out of range : : severity 9 : time 41 : test.traff.c1.counter1.ovl_error
OVL_ERROR : ASSERT_ALWAYS : The Counter out of range : : severity 9 : time 43 : test.traff.c1.counter1.ovl_error
OVL_ERROR : ASSERT_TRANSITION : The State Error 2 : : severity 9 : time 59 : test.traff.c1.green1.ovl_error
L31 "test.v": $finish at simulation time 70
0 simulation events (use +profile or +listcounts option to count)
CPU time: 0.0 secs to compile + 0.0 secs to link + 0.1 secs in simulation
End of VERILOG-XL 4.00.p002  Apr 28, 2004  22:35:39
cae12% /usr2/grad91/hbz/OVL/traffic/Ovl ( 58 ))
```

Observability Consideration [Jiang 09]

- During simulation, sometimes executed ≠ observed
  - Observability should be considered in debugging also
- Using probabilistic observability (PCS) as the sorting metric, the effective size reduction (ESR) is larger
  - Most design errors appear in the top 20% of the error space

design name	#line	#var	Confidence Score (CS)					Probabilistic Confidence Score (PCS)					ESR Ratio
			#cases_CS			Avg_ ESR_CS	t(s)	#case_PCS			Avg_ ESR_PCS	t(s)	
			0~0.2	0.2~0.5	0.5~1.0			0~0.2	0.2~0.5	0.5~1.0			
B01	110	7	40	10	0	0.11	0.3	49	1	0	0.07	0.5	0.64
B02	70	5	38	12	0	0.16	0.3	50	0	0	0.11	0.5	0.69
B03	141	21	35	15	0	0.18	0.4	45	5	0	0.09	0.5	0.50
B04	102	19	32	17	1	0.23	0.3	45	5	0	0.11	0.4	0.48
B05	332	25	24	23	3	0.26	1.3	43	7	0	0.10	1.7	0.38
B07	92	11	37	13	0	0.21	0.4	46	4	0	0.09	0.6	0.43
B08	89	23	32	17	1	0.24	0.6	44	6	0	0.10	0.9	0.42
B14	509	27	17	26	7	0.36	3.8	37	13	0	0.15	5.2	0.42
B21	1089	65	14	28	8	0.42	6.7	31	19	0	0.17	9.7	0.40
pcpu	952	54	15	30	5	0.37	4.1	33	17	0	0.16	6.1	0.43
div16	235	11	23	24	3	0.25	0.7	42	8	0	0.12	1.0	0.48
mtrx	80	11	37	13	0	0.19	0.4	50	0	0	0.11	0.6	0.58
rankf	656	48	18	27	5	0.29	3.1	33	17	0	0.17	4.6	0.59

References

- M. Khalil, Y. Le Traon, and C. Robach, "Towards an Automatic Diagnosis for High-level Validation", Int'l. Test Conference, pp. 1010-1018, 1998.
- S.Y. Huang, and K. T. Cheng, "Error tracer: design error diagnosis based on fault simulation techniques", IEEE Trans. on Computer-Aided Design, vol. 18, pp. 1341-1352, Sep. 1999.
- V. Boppana, I. Ghosh, R. Mukherjee, J. Jain, and M. Fujita, "Hierarchical error diagnosis targeting RTL circuit", Int'l Conf. on VLSI Design, pp.436-441, Jan. 2000.
- T.-Y. Jiang, C.-N. Liu and J.-Y. Jou, "Error Diagnosis for RTL Designs in HDLs", The 11th IEEE Asia Test Symposium (ATS), pp. 362-367, Nov. 2002.
- Y.C. Hsu, B. Tabbara, Y.A. Chen, and F. Tsai, "Advanced technique for RTL debugging," IEEE/ACM Design Automation Conference, pp.362-367, Jun. 2003.
- B.-R. .Huang, T.-J. Tsai, and C.-N. Liu, "On Debugging Assistance in Assertion-Based Verification", the 12th Workshop on Synthesis and System Integration of Mixed Information Technologies (SASIMI), pp. 290-295, Oct. 2004.
- B. Peischl and F. Wotawa, "Automated Source-Level Error Localization in Hardware Designs," IEEE Design&Test of Computers, pp.8-19, Jan. 2006.
- T.-Y. Jiang, C.-N. Liu and J.-Y. Jou, "Accurate Rank Ordering of Error Candidates for Efficient HDL Design Debugging", IEEE Trans. on Computer-Aided Design, vol. 28, no. 2, pp. 272-284, Feb. 2009.
- Reference manuals of Debussy™, SpringSoft.