Test Scenarios and Coverage

Testing & Verification

Dept. of Computer Science & Engg, IIT Kharagpur

Pallab Dasgupta

Professor, Dept. of Computer Science & Engg.,
Professor-in-charge, AVLSI Design Lab,
Indian Institute of Technology Kharagpur

Agenda

- ❑ Hierarchical Verification
- ❑ Test Plan
- ❑ Pseudorandom Test Generator
- ❑ Verification Coverage

Reference: ***Hardware Design Verification,*** **William K Lam**
Prentice Hall Modern Semiconductor Design Series

Exhaustive Simulation is Infeasible

Consider a sequential circuit having N FFs and M inputs

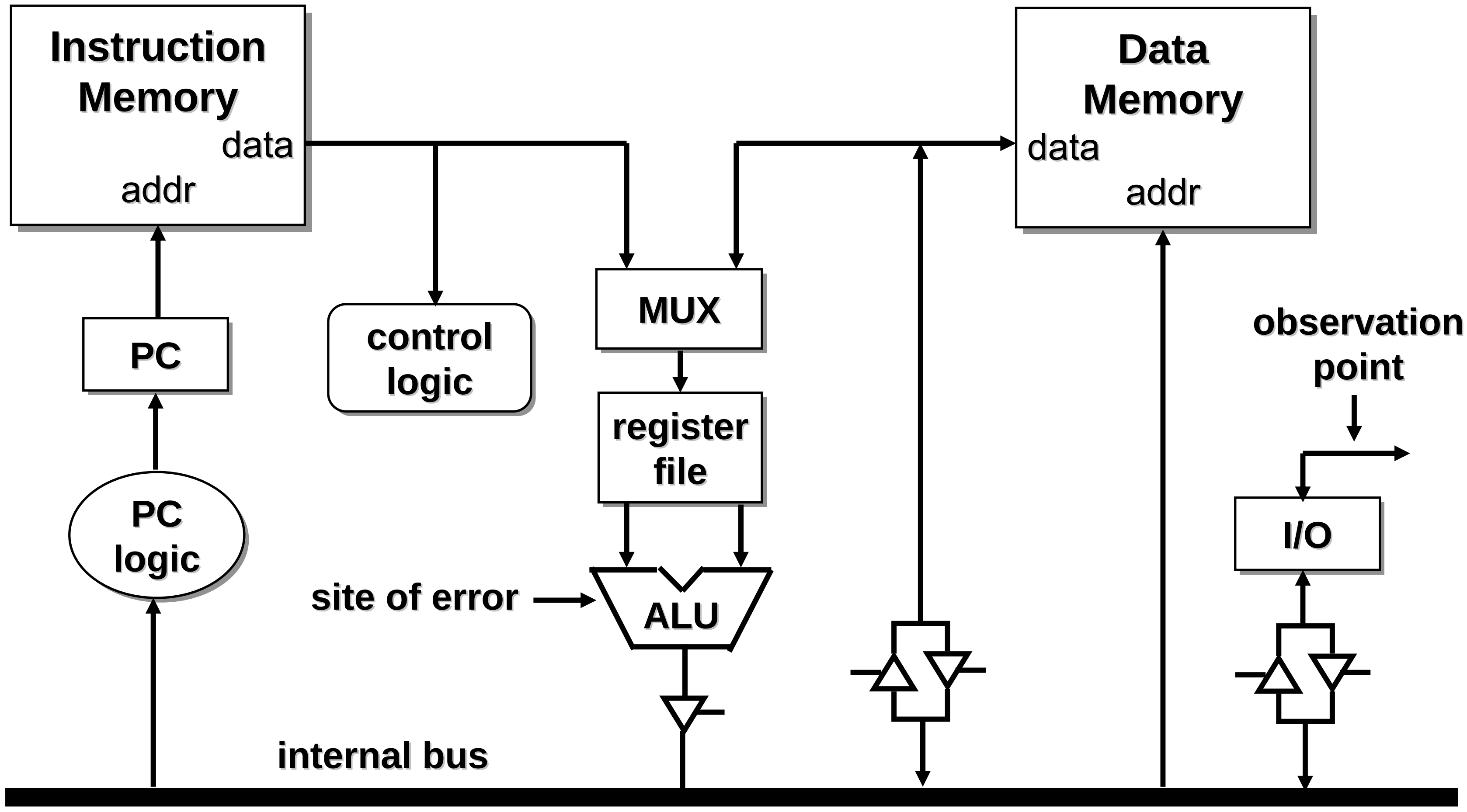
- Exhaustive verification by simulation
  - Reach each state from the initial state
  - At each state verify the behavior for each input vector

- Upper-bound:
  - Number of states: $S = O(2^N)$
  - Number of input vectors at a state: $R = O(2^M)$
  - To reach a state we may have to pass through $O(S)$ states, where each transition requires an input vector
  - Total number of input vectors = $O(S \times R) = O(2^{M+N})$

What's the Alternative?

- The design must be simulated using a well selected subset of input patterns
- Well selected?
  - Test plans and test scenarios
  - Coverage
- What do we observe?
  - Detecting errors by comparing a design's primary outputs with the desired responses may not be the most efficient
    - Internal error may take many cycles to propagate to a primary output to be detected
    - It may not always propagate to the primary output
  - We must carefully choose the signals to be observed

Example

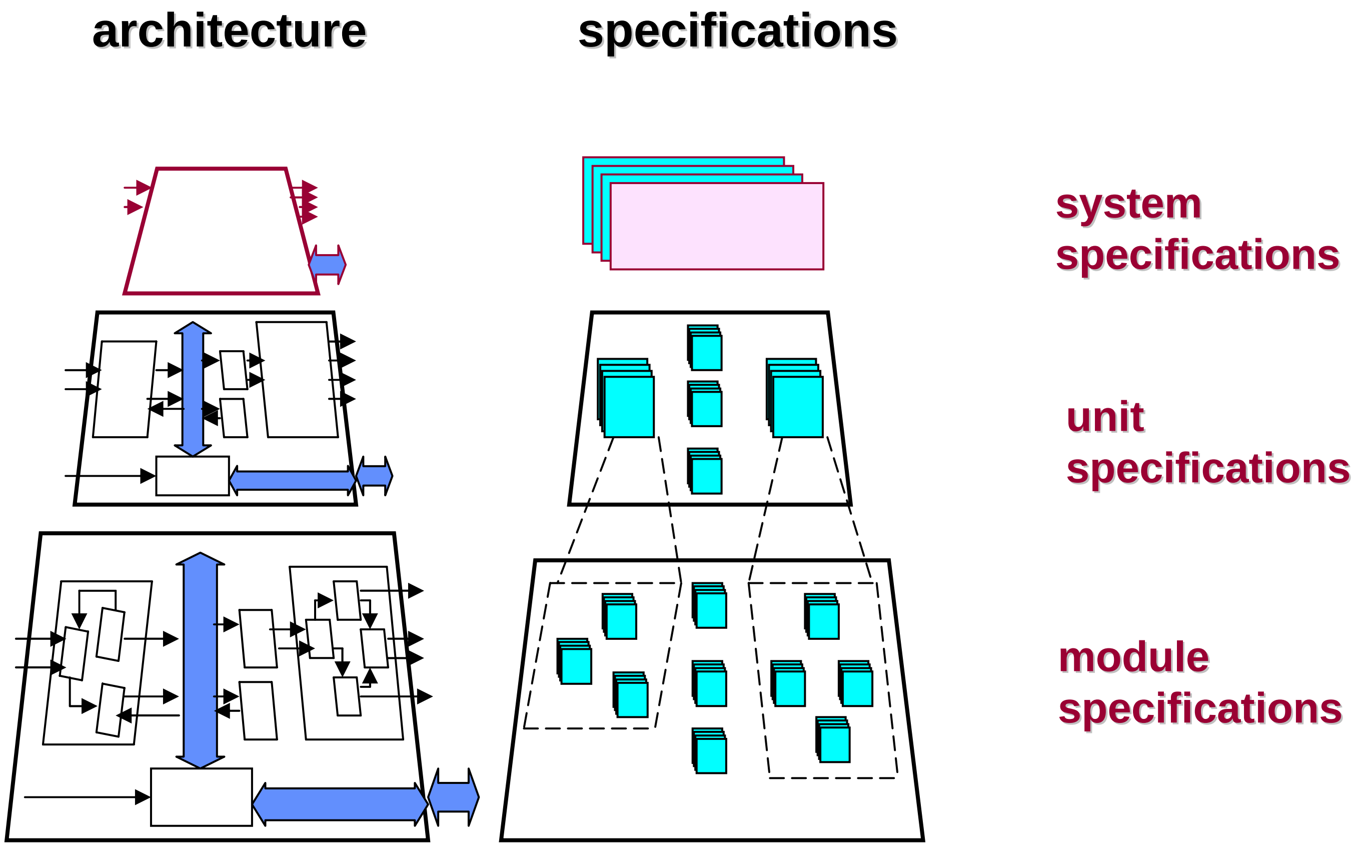


An error of the ALU may be observed many cycles after it is simulated

Bug Hunting: *Main Tasks*

- **Enumerating input possibilities with test plans**
- **Enhancing output observability with assertions**
- **Bridging the gap between exhaustive search and practical execution with coverage measures**

Hierarchical Verification



System Level

- **Problem-1**
  - **Verifying interconnections of the subsystems that make up the system**
    - **Units are assumed to be verified**
    - **Often replaced by abstract functional models**
      - **Example: Instruction Set Simulators for CPUs**

- **Problem-2**
  - **Verifying the complete system at the level of implementation**
    - **Very expensive – high complexity and slow runtime**
    - **Hardware acceleration preferred**
    - **Post silicon debug – emerging strategy**

Unit and Module Levels

- A unit can be a large functional unit
  - Could also be a CPU core
- Each unit can have several modules
  - Possibility of using formal

Test Plan

Steps in creating a test plan:

- Extract functionality from architectural specifications
- Prioritize functionality
- Create test cases
- Track progress

Some Notations

Q_0	: *valid initial states*	Q'_0	: *invalid initial states*
S	: *valid states*	S'	: *invalid states*
I	: *valid inputs*	I'	: *invalid inputs*
O	: *valid outputs*	O'	: *invalid outputs*

$X \rightarrow Y|_I$: *transition from state X to state Y under input I*

$X \Rightarrow Y|_I$: *sequence of transitions from state X leading to state Y under input sequence I*

Ω : *don't care or wild card*

$\Omega \rightarrow \Omega|_\Omega$ ***is the set of all possible input and behavior of the design***

Categories of Test Scenarios

1. $\Omega \rightarrow \Omega|_{I'}$ In this case the design receives an input not specified in the specification. May happen if the design is used in an unintended environment.
2. $S' \rightarrow \Omega|_{\Omega}$ The design is in an invalid state. Verify how the design recovers.
3. $S \rightarrow S'|_{I}$ The design is in a valid state, receives a valid input, and yet reaches an invalid state. This is a bug!!
4. $S \rightarrow S|_{I}$ All is well!!
5. $\Omega \Rightarrow Q_0$ Power-on tests. Start from an arbitrary state and reach a valid initial state.
6. $\Omega \Rightarrow Q'_0$ The design reaches an invalid state after power-on. This is a bug!!

Does this cover everything?

1. $\Omega \Rightarrow \mathbf{Q}_0$ and $\Omega \Rightarrow \mathbf{Q'}_0$ together verify $\Omega \Rightarrow \Omega_0$

2. $\mathbf{S} \rightarrow \mathbf{S'}|_I$ and $\mathbf{S} \rightarrow \mathbf{S}|_I$ together verify $\mathbf{S} \rightarrow \Omega|_I$

3. Combining $\mathbf{S} \rightarrow \Omega|_I$ with $\mathbf{S'} \rightarrow \Omega|_\Omega$ covers $\Omega \rightarrow \Omega|_I$

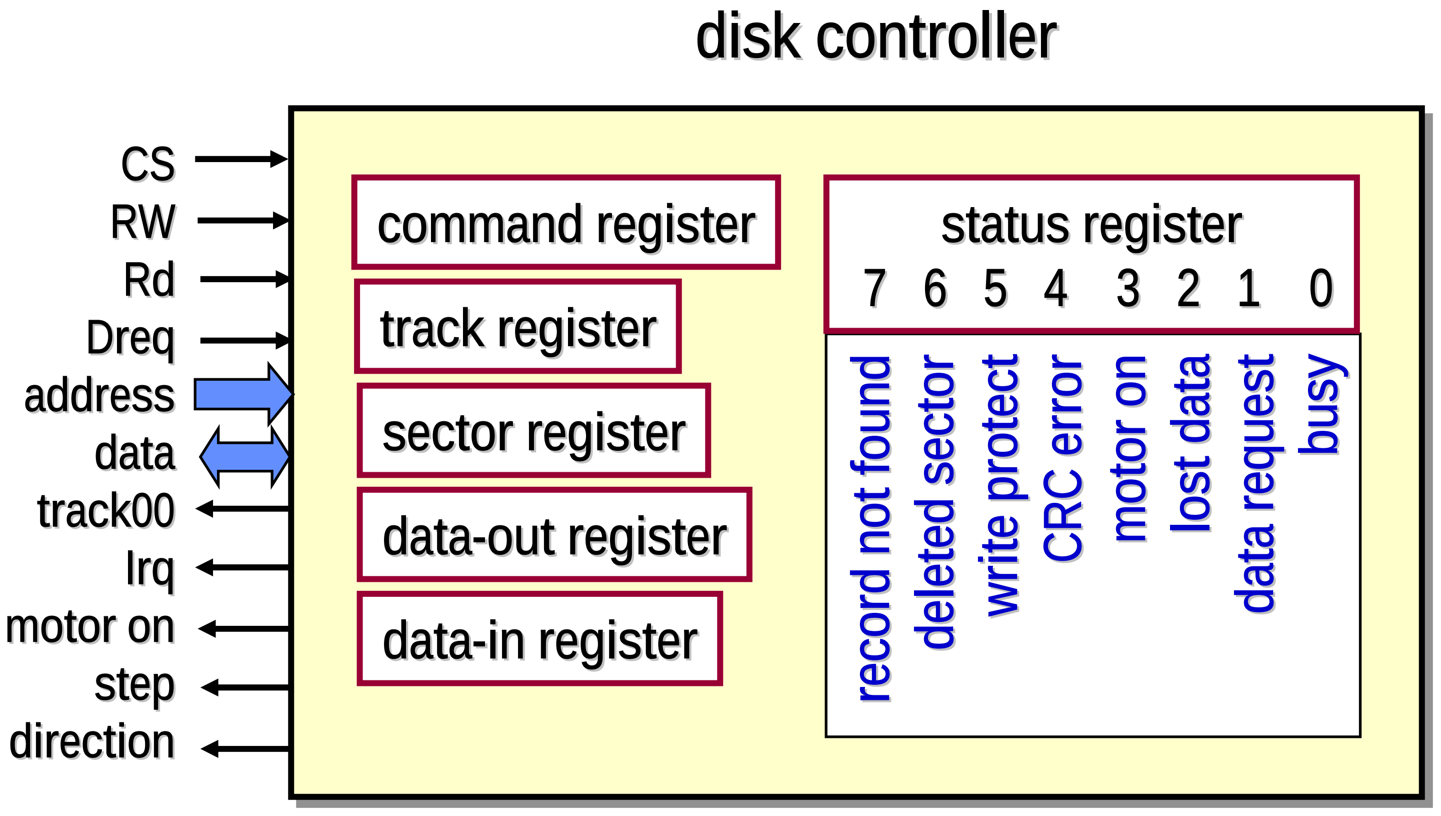
4. Combining $\Omega \rightarrow \Omega|_I$ with $\Omega \rightarrow \Omega|_{I'}$, covers the entire transition space, namely $\Omega \rightarrow \Omega|_\Omega$

Hence, if each of the six categories are thoroughly verified, the design in thoroughly verified.

Example: *Disk Controller*

- **A disk is partitioned into concentric circular strips called *tracks* and each track is partitioned into blocks called *sectors.***
- **Six types of registers**
  - ***Command – contains the command being performed on disk***
  - ***Track – track number***
  - ***Sector – sector number***
  - ***Data-out – data to be written***
  - ***Data-in – data read***
  - ***Status – shows the state of disk and status of the operation***
- **Commands**
  - **RESTORE, SEEK, STEP-IN, STEP-OUT, READ SECTOR, WRITE SECTOR, READ ADDRESS, READ TRACK, WRITE TRACK, FORCE INTERRUPT**
  - ***The RESTORE command determines the head position when the disk drive is first turned on.***

Disk Controller



Test Scenarios

1. $\Omega \rightarrow \Omega|_{I'}$ Test cases send illegal inputs into the controller.

   Example: *address in a forbidden range*

2. $S' \rightarrow \Omega|_{\Omega}$ Set the controller in an illegal state, such as *illegal opcode in command register, out-of-range data in track and sector registers.* Observe whether the controller detects the problem and recovers from it.

3. $S \rightarrow S'|_{I}$ Try to drive it to invalid states. Bug hunting.

   Example: *Given the head position, are there track values that would produce a wrong value from signal* direction?

4. $S \rightarrow S|_{I}$ Try to establish that it reaches only valid states from a valid state.

   Example: *For given values in the track and sector registers, are the outputs from* step *and* direction *correct?*

Test Scenarios

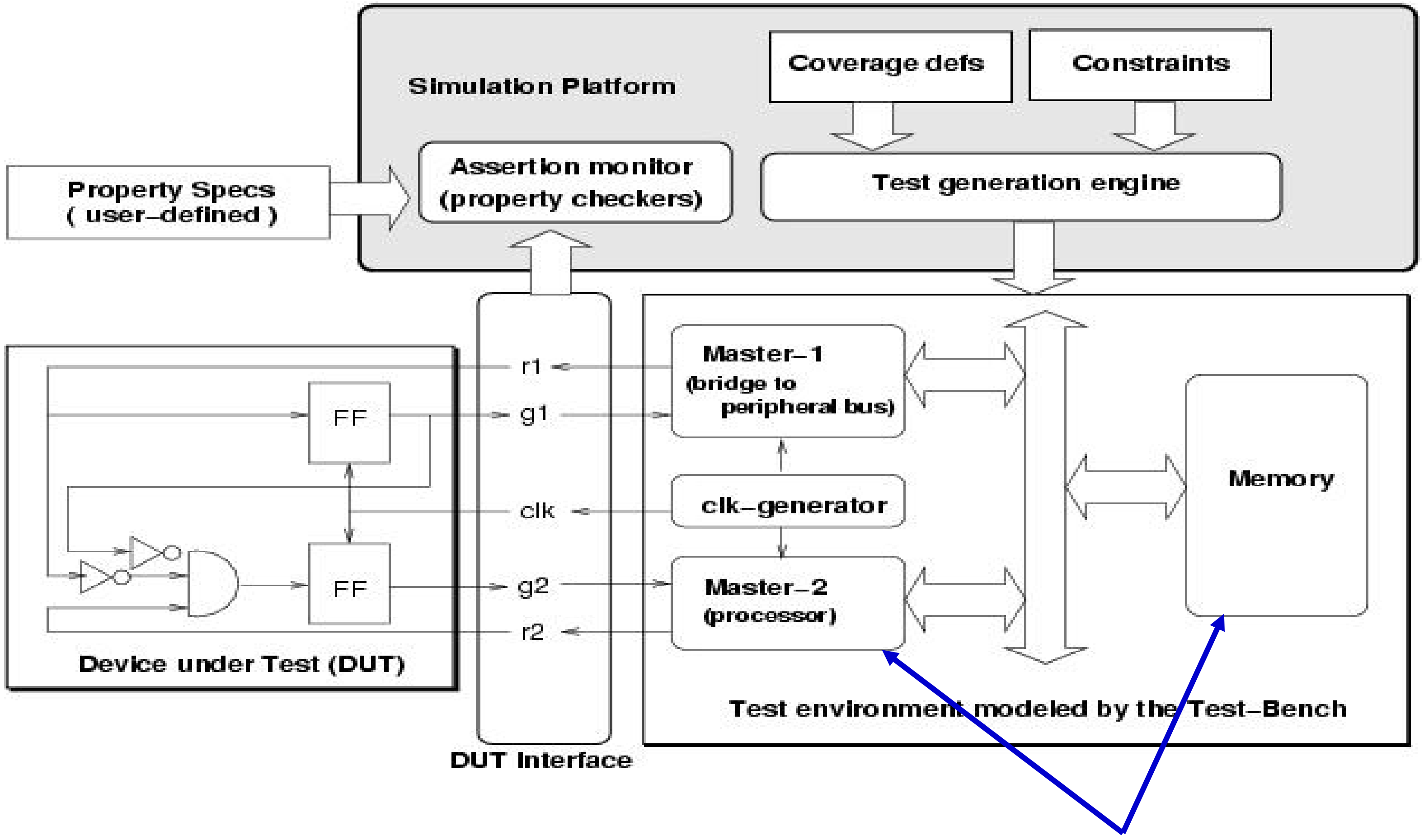
5. $\Omega \Rightarrow Q_0$
   - *What are the values of the registers when the controller is first powered on? Are these values correct or expected?*
   - *Immediately after power-on, do registers* track *and* sector *contain the correct track and sector number of the head position if the* RESTORE *command is executed?*

6. $\Omega \Rightarrow Q'_0$
   - *Under what conditions will the controller enter an illegal state on power-on?*

Pseudo-random Test Generation

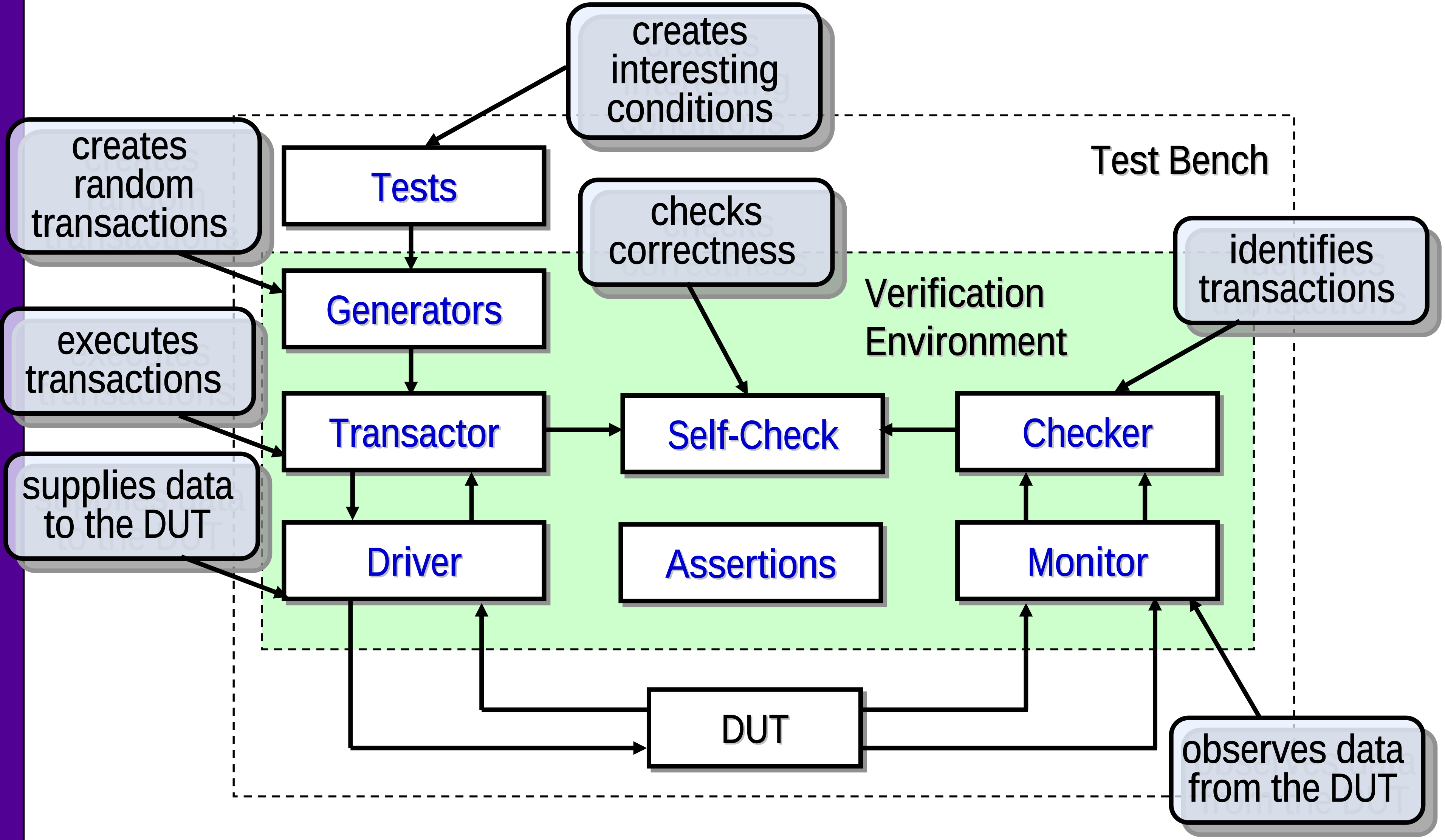
```
program Test (input clk);
class Bus;
   rand bit [15:0] addr;
endclass

Bus B1=new;
initial begin
  B1.randomize() with {addr <= 128;};
  ……
  @(posedge clk);
  B1.randomize () with {addr > 128 && addr <= 1024;};
  ……
end
endprogram
```

Coverage Driven Random Verification

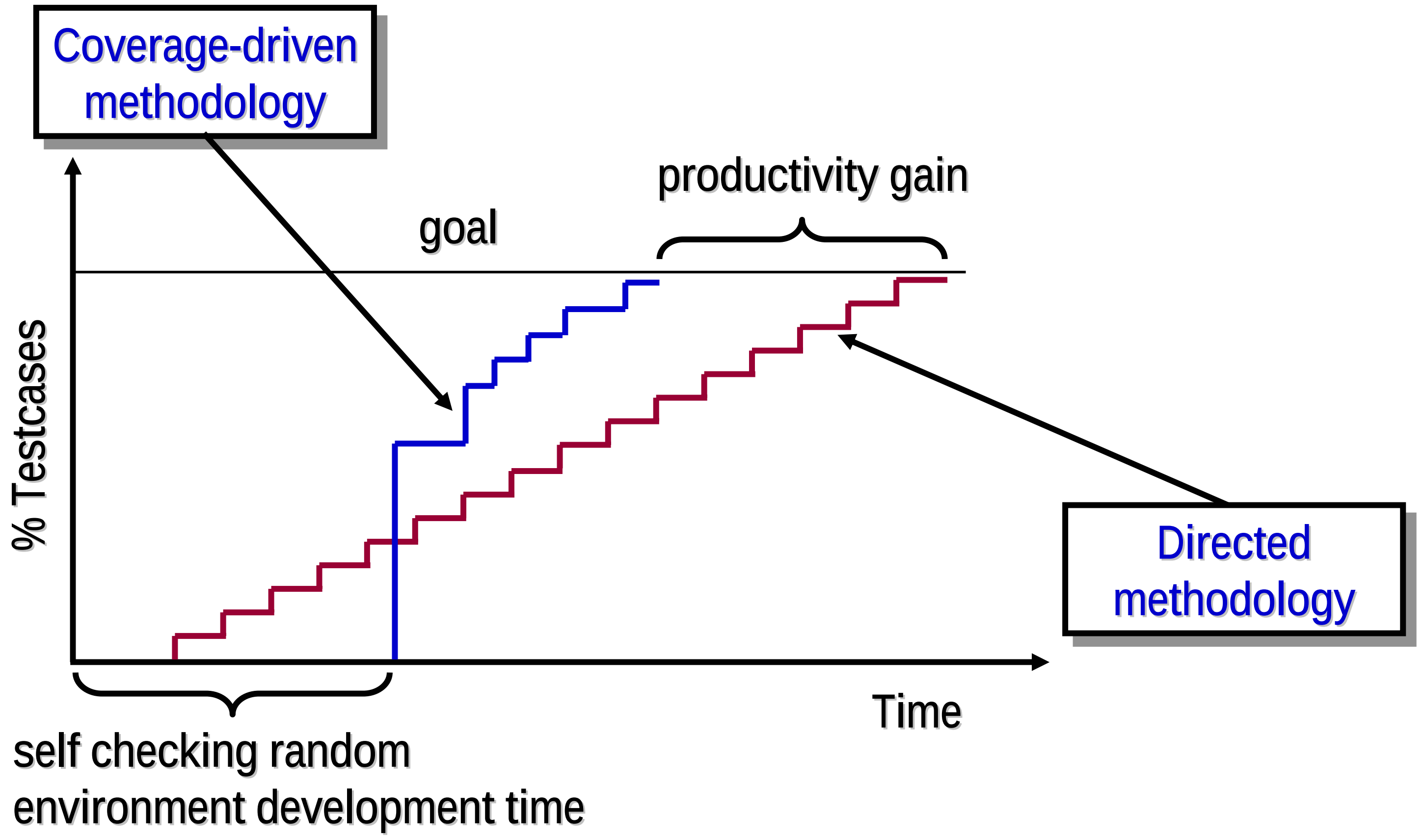


models developed
at the behavioral level

Layered Random Test Architecture



Directed versus Constrained Random



Coverage Metrics

- **Code Coverage**
  - Checks how thoroughly the code of a design is exercised by a suite of simulations
    - For example, *the percentage of RTL code of a FIFO simulated by a simulation suite*
- **Parameter Coverage**
  - Checks the extent that dimensions and parameters in functional units are stressed
    - For example, *the range of depth that a FIFO encounters during a simulation*
- **Functional Coverage**
  - Accounts for the amount of operational features or functions in a design that are exercised
    - For example, *did we check all the operations (push, pop) and all the conditions (full, empty)?*

Code Coverage: *Statement Coverage*

- Collects statistics about statements that are executed

```
always @(posedge clock)
begin
    a = b + c;
    x = (y << 4);
    if (a > x)
    begin
            y = b & a;
            x = (x >> 2);
    end
    else
            b = b^c;
    if (x == y)
            c = y;
    else
            y = a;
end
```

Code covered when a>x and x = y

Code Coverage: *Block Coverage*

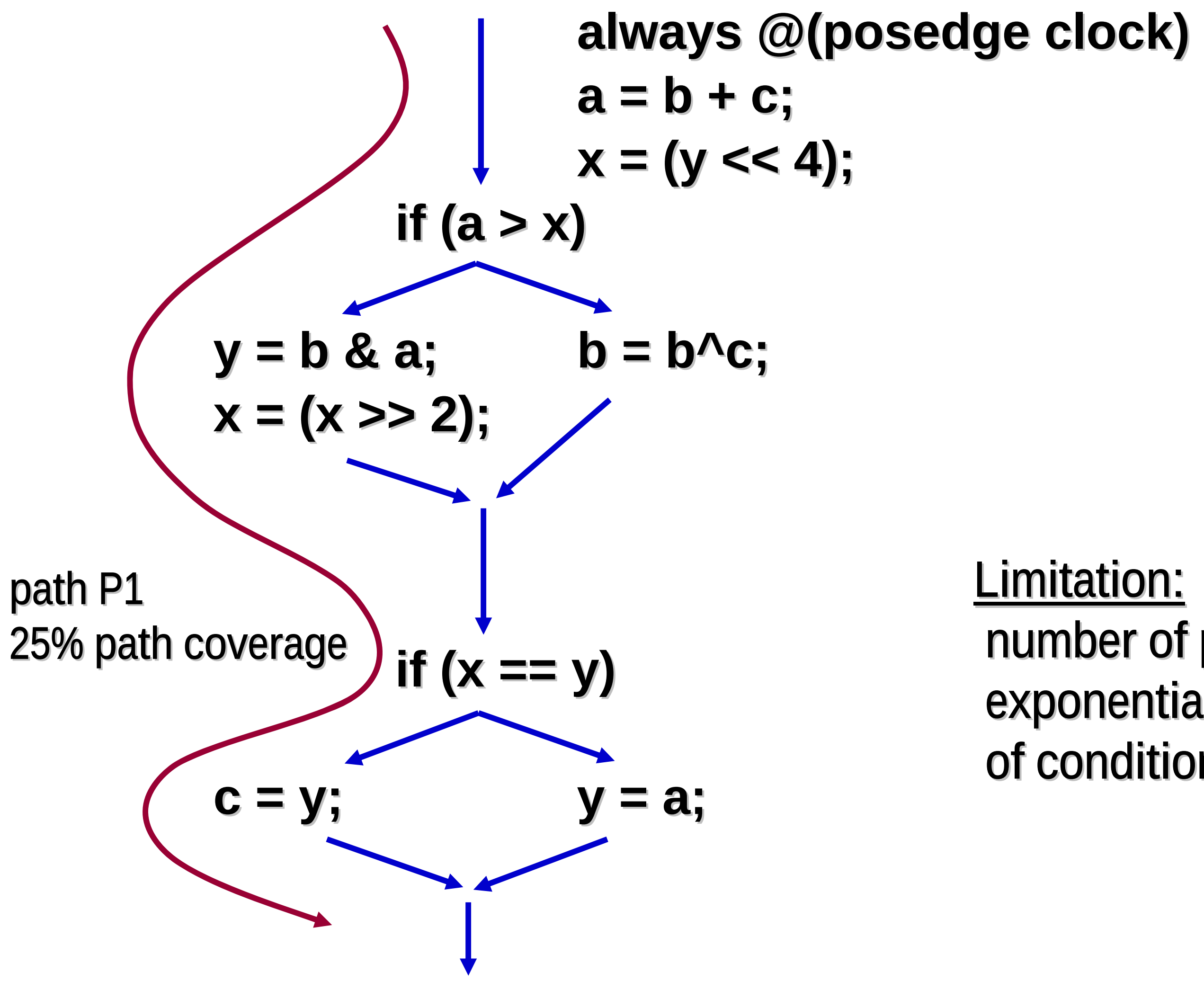
- Collects statistics about blocks that are executed

```
always @(posedge clock)
begin
    a = b + c;
    x = (y << 4);
    if (a > x)
    begin
            y = b & a;
            x = (x >> 2);
    end
    else
            b = b^c;
    if (x == y)
            c = y;
    else
            y = a;
end
```

- A statement in a block is executed if and only if all other statements are executed
- Block coverage is typically preferred over statement coverage

Code Coverage: *Path Coverage*

- Collects statistics about paths that are executed



Limitation:
number of paths grows exponentially with the number of conditional statements

Code Coverage: *Expression Coverage*

- Collects statistics about parts of expressions evaluated
  - For example, $(x_1x_2 + x_3x_4)$ evaluates to 1 can be broken down to whether x_1x_2 or x_3x_4 or bot evaluate to 1
- A *minimum input table* is used
  - A minimum input determining an expression is the fewest number of input variables that must be assigned in order for the expression to have a Boolean value

Examples:

$f = x_1 \& x_2$	x_1	x_2
0	0	X
0	X	0
1	1	1

$f = (y\ ?\ x_1 : x_2)$	y	x_1	x_2
1	1	1	X
0	1	0	X
1	0	X	1
0	0	X	0

Code Coverage: *Expression Coverage*

- For complex expressions we decompose into layers
  - For example, $f = ((x > y) \mid\mid (a \,\&\, b))$, may be decomposed into a top layer having the expression, $f = (f_1 \mid\mid f_2)$, and a bottom layer consisting of $f_1 = (x>y)$ and $f_2 = (a \,\&\, b)$
- Suppose simulation covers the following two cases:

  (x, y, a, b) = { (3, 4, 1, 1), (2, 5, 1, 0) }
  - Coverage of f = 66.67%
  - Coverage of f_1 = 50%, Coverage of f_2 = 66.67%

$f_1 = x>y$	$x>y$
0	0
1	1

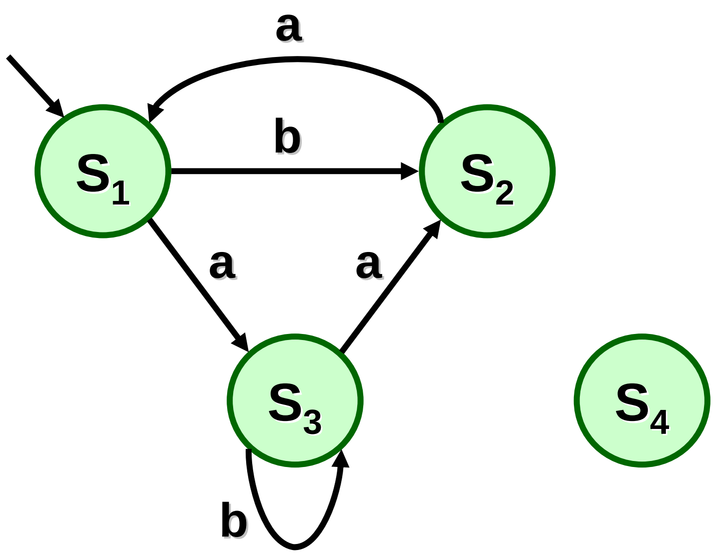
$f_2 = a \,\&\, b$	a	b
0	0	X
0	X	0
1	1	1

$f = f_1 \mid\mid f_2$	f_1	f_2
0	0	0
1	1	X
1	X	1

State Coverage

- State coverage calculates the fraction of states visited during simulation
  - The states are extracted from the RTL code
- Example:

```
initial present_state = `S1;
always @(posedge clock)
   case ({present_state, in})
      {`S1, a} : next_state = `S3;
      {`S2, a} : next_state = `S1;
      {`S3, a} : next_state = `S2;
      {`S1, b} : next_state = `S2;
      {`S3, b} : next_state = `S3;
  endcase
```



Other state-based coverage metrics

- Transition coverage
  - Percentage of transitions of the state machine covered during simulation
- Sequence coverage
  - Percentage of *user-defined* state sequences that are traversed during simulation
- Toggle coverage
  - Measures whether and how many times nets, nodes, significant variables, ports, and buses toggle.

Code Instrumentation

- The RTL code is instrumented to detect statement execution and collect coverage data
  - This is done by adding user-defined PLI tasks to the RTL code
    - Whenever a statement is executed, a PLI task is called to record the activity
  - Simulators support specific types of *callbacks* at specific simulation points

Instrumented RTL Code

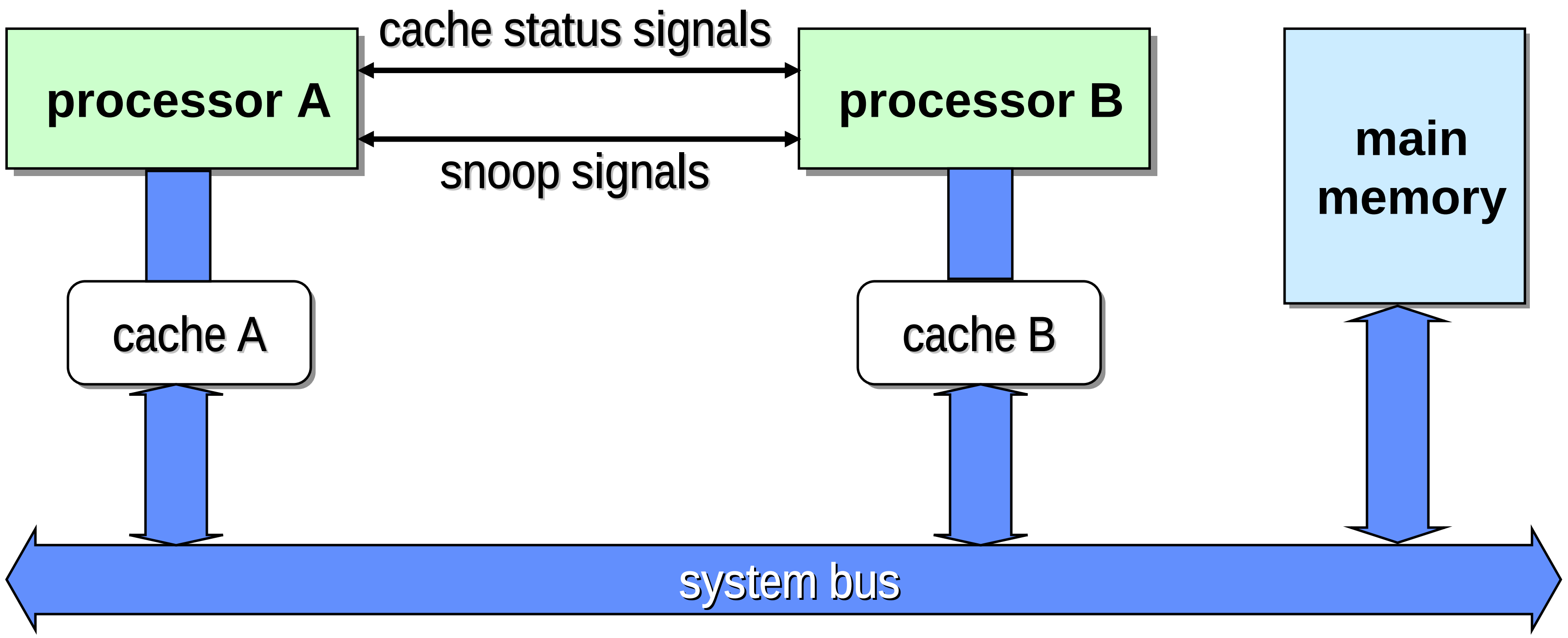
```
always @(posedge clock)
begin
    $C1( );
    a = b + c;
    x = (y << 4);
    if (a > x)
    begin
            $C2( );
            $C_exp(y, b, a);   // y = b&a;
            x = (x >> 2);
    end
    else begin
            $C3( );
            b = b^c;
    end
end
```

Instrumented PLI:

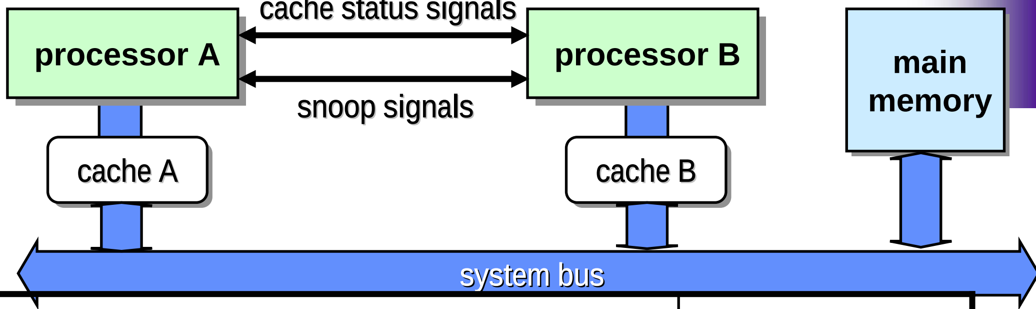
```
void C1( )
{
  ----
  time = $tf_gettime( ); //get sim time
  C1_call_frequency[time].freq++;
  ----
}
```

Instrumented PLI for computing expression coverage

Functional Coverage Example



Functional Cov



Processor	Cache	Action	Result
1. Processor A reads D	Cache A: Cache B:	Read from main memory and mark it exclusive	Cache A:D (exclusive) Cache B:
2. Processor B reads D	Cache A:D (exclusive) Cache B:	Copy D to cache B and make both copies shared	Cache A:D (shared) Cache B:D (shared)
3. Processor A writes D	Cache A:D (shared) Cache B:D (shared)	Invalidate all copies of D, update cache A, and mark it modified	Cache A:D (modified) Cache B: no D
4. Processor B reads D	Cache A:D (modified) Cache B:	Deposit D of cache A to memory, copy D to cache B, mark both shared	Cache A:D (shared) Cache B:D (shared)