Adding Coverage to Your Code

CORRECT designs

- Sample flow of developing a coverage model
  - DUT Specification
  - Developing a Verification Plan
- Using *e* coverage constructs
- Using SystemVerilog coverage constructs

DUT Specification

CORRECT designs

Instruction format:

15	14 ... 12	11 ... 10	9 ... 8	7 ... 0
mode	opcode	trg	src	data

	mode	
opcode	**IMMEDIATE** (0x0)	**REGISTER** (0x1)
ADD (0x0)	`(src) + data -> (trg)`	`(src) + (trg) -> (trg)`
AND (0x1)	`(src) & data -> (trg)`	`(src) & (trg) -> (trg)`
OR (0x2)	`(src) | data -> (trg)`	`(src) | (trg) -> (trg)`
NOT (0x3)	*N/A*	`~(src) -> (trg)`
LD (0x4)	`mem[data] -> (trg)`	`mem[(src)] -> (trg)`
ST (0x5)	`data -> mem[(trg)]`	`(src) -> mem[(trg)]`

There are 4 registers, any of which can be used as the source (**`src`**) or target (**`trg`**)

Developing a Verification Plan

CORRECT designs

"Black Box" Analysis

Examining the instruction set specification suggests we should:

1. Verify all opcodes and modes
2. Include interesting values for immediate data:
   - "edge values": *minimum* and *maximum*
3. Test all *valid* mode/opcode combinations

"Grey Box" Analysis

Looking at the CPU implementation indicates we should:

1. Assure an instruction in the pipeline never "interferes" with the next one
2. Assure the target register of each instruction is updated before being used as the source register of a subsequent instruction

Now write coverage code and measure progress towards these verification goals

CORRECT designs

Adding Coverage to *e* Code

CORRECT designs

- Stimulus and Coverage
- Demonstrate per_instance coverage
- Stimulus Independent Coverage

Coverage on Stimulus Struct

CORRECT designs

```
type e_opcode:[ ADD, AND, OR, NOT, LD, ST ] (bits:3);
type e_mode  :[ IMMEDIATE, REGISTER      ] (bits:1);
type e_reg   :[ REG0, REG1, REG2, REG3   ] (bits:2);
struct Instruction {
   opcode : e_opcode;
   mode   : e_mode;
   src    : e_reg;   trg: e_reg;
   data   : byte;
   keep opcode == NOT => mode != IMMEDIATE;
```

	mode	
opcode	**IMMEDIATE** (0x0)	**REGISTER** (0x1)
ADD (0x0)	(src) + data -> (trg)	(src) + (trg) -> (trg)
AND (0x1)	(src) & data -> (trg)	(src) & (trg) -> (trg)
OR (0x2)	(src) | data -> (trg)	(src) | (trg) -> (trg)
NOT (0x3)	*N/A*	~(src) -> (trg)
LD (0x4)	mem[data] -> (trg)	mem[(src)] -> (trg)
ST (0x5)	data -> mem[(trg)]	(src) -> mem[(trg)]

```
};
```

Coverage on Stimulus Struct

CORRECT designs

```
type e_opcode:[ ADD, AND, OR, NOT, LD, ST ] (bits:3);
type e_mode  :[ IMMEDIATE, REGISTER       ] (bits:1);
type e_reg   :[ REG0, REG1, REG2, REG3    ] (bits:2);
struct Instruction {
   opcode : e_opcode;
   mode   : e_mode;
   src    : e_reg;    trg: e_reg;
   data   : byte;
   keep opcode == NOT => mode != IMMEDIATE;
   event cover_instr;
   cover cover_instr is {
      item opcode;
      item mode;

      item data using when=(mode==IMMEDIATE), ranges={
        range([0], "EDGE0");  range([255], "EDGE255");
        range([1..254], "MIDDLE")
      };
      //We'll add more coverage constructs in the next slide
      ...

};
```

Cover group embedded in stimulus struct

A.1 Verify all opcodes and modes

A.2 Include interesting values for immediate data

Instrumenting Per the Test Plan

CORRECT designs

Collect measurements per the verification plan

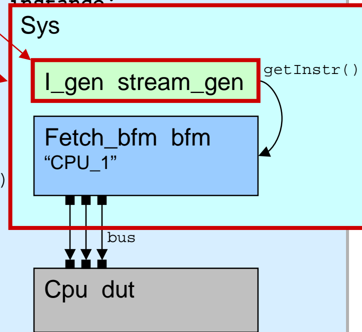
```
struct Instruction {
   ...
   cover cover_instr is {
      ...
      cross opcode,mode using
         ignore=(opcode == NOT
              && mode   == IMMEDIATE);

      transition opcode using name=every_other;

      ...
   };

};
```

A.3 Test all *valid* mode/opcode combinations

B.1 Assure an instruction in the pipeline never “interferes” with the next one

Testbench Architecture

```
unit I_gen {
   getInstr(callerId: string): Instruction is undefined;
};
unit I_gen_01a like I_gen {
   getInstr(callerId: string): Instruction is only {
        gen result;
   };
};
extend sys {
   bus : out simple_port of uint(bits:16) is instance;
   clk :  in event_port  is instance;
   stream_gen: I_gen_01a is instance;
   bfm : Fetch_bfm       is instance;
   keep bfm.bus == read_only(bus);
   keep bfm.clk == read_only(clk);
   run() is also {
     start bfm.startUp("CPU_1",stream_gen,10);
   };

   keep bus.hdl_path() == "bus";
   keep clk.hdl_path() == "clk";
   keep clk.edge()     == rise;
};
```



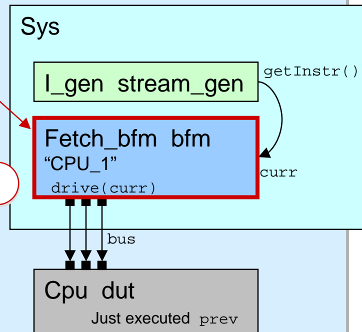
Coverage In a Unit

Use coverage throughout testbench

B.2 Assure the target register of each instruction is updated before being used as the source register of a subsequent instruction

```
unit Fetch_bfm {
   bus : out simple_port of uint(bits:16);
   !previous: Instruction;  !curr: Instruction;
   event op_history;
   cover op_history is {
      item prev_opcode: e_opcode = previous.opcode using
                        when=(curr.src == previous.trg);
   };
   startUp(name: string, stream_gen: I_gen, count: uint) @clk$ is {
      for n from 0 to (count-1) {
        curr = stream_gen.getInstr(name);
        drive(curr);
        emit curr.cover_instr;
        if (n>0) {emit op_history;};
        previous = deep_copy(curr);
      };
   };
   clk : in event_port;
   drive(i: Instruction) @clk$ is {
      //DRIVE i ONTO THE DUT FETCH INTERFACE
   };
}; //Fetch_bfm
```

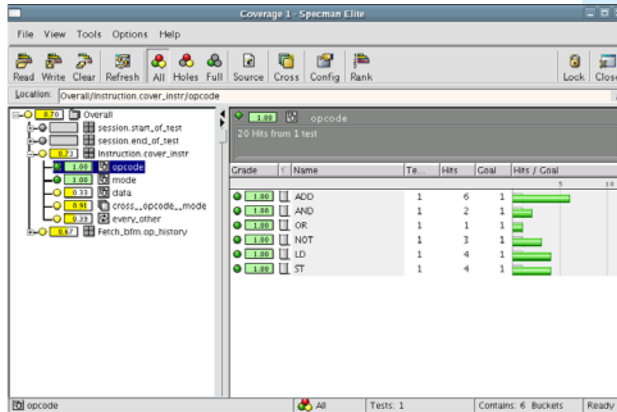
Trigger coverage recording



Struct Coverage

Thousands of `Instruction` stimulus objects may be created and randomized on each simulation run

```
struct Instruction {
   cover cover_instr is {
      opcode : e_opcode;
      mode   : e_mode;
      ...
```



All cover instances (objects) of the struct contribute to common coverage data

Stimulus Values and Coverage

```
unit Fetch_bfm {
   ...
   cover op_history is {
      item prev_opcode: e_opcode = previous.opcode using
                         when=(curr.src == previous.trg);
```

Affect the distribution of stimulus values to increase the odds of hitting your coverage goals

```
unit I_gen_01a like I_gen;
   !i: Instruction;
   src_equal_trg: bool;
   keep soft src_equal_trg == select {2: FALSE; 1: TRUE;};
   !prev_trg: e_reg;
   getInstr(callerId: string): Instruction is only {
      prev_trg = i.trg;
      gen src_equal_trg;
      if (src_equal_trg) {
               gen i keeping {.src == prev_trg;};
      } else { gen i; };
      return i;
   };
```

>33% chance source register will equal previous target register

Controlling Stimulus for Coverage

CORRECT designs

```
unit I_gen_02 like I_gen {
   !i: Instruction;  !i1: Instruction;  !i2: Instruction;
   getInstr(callerId: string): Instruction is only {
      // There would be value in coordinating the instruction (streams)
      // sent to CPU_1 and CPU_2 to hit interesting LD/ST dependency cases.
      // We could write code here to create those cases and cover points
      case (callerId) {
         "CPU_1": {gen i1 keeping {...}; i = i1;}; //return instruction for CPU_1
         "CPU_2": {gen i2 keeping {...}; i = i2;}; //return instruction for CPU_2
      };
      return i;
   };
extend sys {
   bus1: out simple_port of uint(bits:16) is inst
   bus2: out simple_port of uint(bits:16) is inst
   clk :  in event_port  is instance;
   stream_gen: I_gen_02  is instance;
   bfm1: Fetch_bfm       is instance;
   bfm2: Fetch_bfm       is instance;
   run() is also {
     start bfm1.startUp("CPU_1", stream_gen, 10)
     start bfm2.startUp("CPU_2", stream_gen, 10)
   };
   ...
```

Effective stimulus is required to achieve coverage goals

What coverage should be added?

I_gen stream_gen
getInstr()
Fetch_bfm bfm1 "CPU_1"
Fetch_bfm bfm2 "CPU_2"
bus1
bus2
Cpu cpu1
Cpu cpu2

Per Instance Coverage

CORRECT designs

```
type e_cpu: [CPU_1, CPU_2];
...

struct Instruction {
   cpu_num: e_cpu;
   ...

   cover cover_instr is {
      item opcode;
      item mode;
      ...
      item cpu_num using per_instance;
   };
};
unit I_gen_02 like I_gen {
 ...
 case (callerId) {
   "CPU_1": {gen i1 keeping {.cpu_num == callerId.as_a(e_cpu); ...
   ...
```

CPU_1 + CPU_2

CPU_1

CPU_2

Track coverage results separately based on the instance

0.60 Overall
session.start_of_test
session.end_of_test
0.77 Instruction.cover_instr
1.00 opcode
1.00 mode
1.00 cpu_num
0.33 data
0.91 cross__opcode__mode
0.39 every_other
0.60 Instruction.cover_instr(cpu_num==CPU_1)
0.83 opcode
1.00 mode
0.33 data
0.64 cross__opcode__mode
0.19 every_other
0.59 Instruction.cover_instr(cpu_num==CPU_2)
0.83 opcode
1.00 mode
0.33 data
0.55 cross__opcode__mode
0.22 every_other
0.83 Fetch_bfm.op_history

0.83 opcode
10 Hits from 1 test

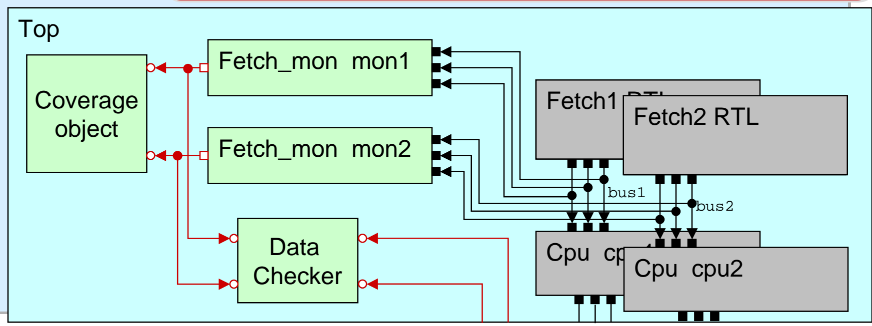
Grade	Name
1.00	ADD
1.00	AND
1.00	OR
1.00	NOT
0	LD
1.00	ST

Stimulus-Independent Coverage

```
extend sys {
   bfm1: Fetch_bfm   is instance;
   bfm2: Fetch_bfm   is instance;
   mon1: Fetch_mon   is instance;
   mon2: Fetch_mon   is instance;
   covInst: Coverage is instance;
   run() is also {
     mon1.startUp("CPU_1", covInst);
     mon2.startUp("CPU_2", covInst);
     ...
};
```

Improved, more portable approach shown

Add coverage at required hierarchy levels in the testbench



Adding Coverage to SystemVerilog Code

- Stimulus and Coverage
- Demonstrate per_instance coverage
- Stimulus Independent Coverage

Coverage on Stimulus Class

```
package InstrCPU;
   typedef enum bit [2:0] { ADD, AND, OR, NOT, LD, ST } e_opcode;
   typedef enum bit       { IMMEDIATE, REGISTER       } e_mode;
   typedef enum bit [1:0] { REG0, REG1, REG2, REG3    } e_reg;
   class Instruction;
      rand e_opcode  opcode;
      rand e_mode    mode;
      rand e_reg     src, trg;
      rand bit [7:0] data;
      constraint valid_op { opcode == NOT -> mode != IMMEDIATE; }
```

	mode	
opcode	**IMMEDIATE** (0x0)	**REGISTER** (0x1)
ADD (0x0)	`(src) + data -> (trg)`	`(src) + (trg) -> (trg)`
AND (0x1)	`(src) & data -> (trg)`	`(src) & (trg) -> (trg)`
OR (0x2)	`(src) | data -> (trg)`	`(src) | (trg) -> (trg)`
NOT (0x3)	*N/A*	`~(src) -> (trg)`
LD (0x4)	`mem[data] -> (trg)`	`mem[(src)] -> (trg)`
ST (0x5)	`data -> mem[(trg)]`	`(src) -> mem[(trg)]`

```
   endclass : Instruction
endpackage
```

CORRECT designs

Coverage on Stimulus Class

```
package InstrCPU;
   typedef enum bit [2:0] { ADD, AND, OR, NOT, LD, ST } e_opcode;
   typedef enum bit       { IMMEDIATE, REGISTER      } e_mode;
   typedef enum bit [1:0] { REG0, REG1, REG2, REG3   } e_reg;
   class Instruction;
      rand e_opcode  opcode;
      rand e_mode    mode;
      rand e_reg     src, trg;
      rand bit [7:0] data;
      constraint valid_op { opcode == NOT -> mode != IMMEDIATE; }
      covergroup cover_instr;
         coverpoint opcode;
         coverpoint mode;
         coverpoint data iff (mode==IMMEDIATE) {
            bins edges[] = {0, 255};
            bins middle  = {[1:254]};
         }
         ... //We'll add more coverage constructs in the next slide
      endgroup
      function new();
         cover_instr = new;
      endfunction
   endclass : Instruction
endpackage
```

covergroup embedded in stimulus class

A.1 Verify all opcodes and modes

A.2 Include interesting values for immediate data

create covergroup instance

CORRECT designs

Instrumenting Per the Test Plan

```
package InstrCPU;
   ...
   class Instruction;
      ...
      covergroup cover_instr;
         ...
         cross opcode,mode {
            ignore_bins irrelevant = binsof(opcode) intersect {NOT}
                                  && binsof(mode)   intersect {IMMEDIATE};
         }

         every_other: coverpoint opcode {
            bins opTrans[] = ([ADD:ST] => [ADD:ST]);
         }
      endgroup

      function Instruction copy();
         copy = new this;
      endfunction
   endclass : Instruction
endpackage
```

Collect measurements per the verification plan

A.3 Test all *valid* mode/opcode combinations

B.1 Assure an instruction in the pipeline never “interferes” with the next one

Analysis

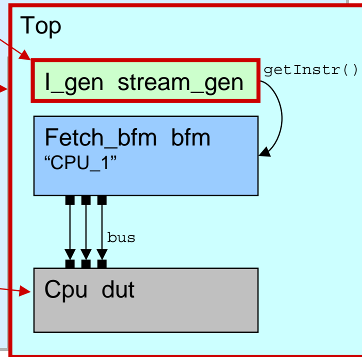
Name	Coverage	Goal	% of Goal	Status
/InstrCPU/Instruction				
TYPE cover_instr	81.1%	100	81.1%	
instr::opcode	100.0%	100	100.0%	
nstr::mode	100.0%	100	100.0%	
_instr::data	33.3%	100	33.3%	
CVP cover_instr::every_other	72.2%	100	72.2%	
CROSS cover_instr::#cross__0#	100.0%	100	100.0%	
bin <auto[ADD],auto[IMMEDIATE]>	2	1	200.0%	
bin <auto[AND],auto[IMMEDIATE]>	5	1	500.0%	
bin <auto[OR],auto[IMMEDIATE]>	2	1	200.0%	
bin <auto[LD],auto[IMMEDIATE]>	4	1	400.0%	

Analysis

Name	Coverage	Goal	% of Goal	Status
/InstrCPU/Instruction				
TYPE cover_instr	81.1%	100	81.1%	
CVP cover_instr::opcode	100.0%	100	100.0%	
CVP cover_instr::mode	100.0%	100	100.0%	
CVP cover_instr::data	33.3%	100	33.3%	
CVP cover_instr::every_other	72.2%	100	72.2%	
bin opTrans[ST=>ST]	2	1	200.0%	
bin opTrans[ST=>LD]	2	1	200.0%	
bin opTrans[ST=>NOT]	1	1	100.0%	
bin opTrans[ST=>OR]	0	1	0.0%	

Testbench Architecture

```
import InstrCPU::*;
class I_gen;
   virtual function Instruction getInstr(string callerId); endfunction
endclass

class I_gen_01 extends I_gen;
   Instruction i = new;
   virtual function Instruction getInstr(string callerId);
      if (i.randomize()) return i;
   endfunction
endclass

module Top;
   logic clk; logic [15:0] bus;
   I_gen_01  stream_gen = new;
   Fetch_bfm bfm(clk, bus);
   Cpu       dut(clk, bus);
   initial bfm.startUp("CPU_1",stream_gen,20);
endmodule

module Cpu(input clk, [15:0] bus);
   ...
endmodule
```



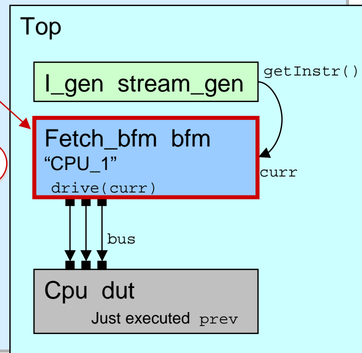
Coverage In a Module

```
module Fetch_bfm(input clk, output [15:0] bus);
   Instruction curr, prev;

   covergroup op_history;
      coverpoint prev.opcode iff (curr.src == prev.trg);
   endgroup
   op_history cg = new;

   task startUp(string name, I_gen stream_gen, int count);
      for (int n=0; n<count; n++) begin
         curr = stream_gen.getInstr(name);
         drive(curr);
         curr.cover_instr.sample();
         if (n>0) cg.sample();
         prev = curr.copy();
      end
   endtask

   task drive(Instruction i);
      //DRIVE i ONTO THE DUT FETCH INTERFACE
   endtask
endmodule : Fetch_bfm
```

Use coverage throughout testbench

B.2 Assure the target register of each instruction is updated before being used as the source register of a subsequent instruction

Trigger coverage recording



Class and Module Coverage

Thousands of `Instruction` stimulus objects may be created and randomized on each simulation run

- Each **class** implicitly has a covergroup variable
- A **module** covergroup variable must be declared

```
class Instruction;
  covergroup cover_instr;
    ...
  function new;
    cover_instr = new;
  endfunction
```

```
module Fetch_bfm(...)
  covergroup op_history;
    ...
  op_history cg = new;

  task startUp(...);
   //for loop
     //get "curr" Instruction
    drive(curr);
    curr.cover_instr.sample();
    if (n>0) cg.sample();
   ...
```

Analysis

Name	Coverage	Goal	% of Goal	Status
/InstrCPU/Instruction				
TYPE cover_instr	73.2%	100	73.2%	
CVP cover_instr::opcode	100.0%	100	100.0%	
CVP cover_instr::mode	100.0%	100	100.0%	
CVP cover_instr::data	33.3%	100	33.3%	
CVP cover_instr::every_other	41.7%	100	41.7%	
CROSS cover_instr::#cross__0#	90.9%	100	90.9%	
/Top/bfm				
TYPE op_history	66.7%	100	66.7%	
CVP op_history::#coverpoint__0#	66.7%	100	66.7%	
bin auto[ADD]	1	1	100.0%	
bin auto[AND]	0	1	0.0%	
bin auto[OR]	1	1	100.0%	
bin auto[NOT]	1	1	100.0%	
bin auto[LD]	2	1	200.0%	
bin auto[ST]	0	1	0.0%	

All covergroup instances (objects) of the class contribute to common coverage data

Stimulus Values and Coverage

```
module Fetch_bfm(...);
   ...
   covergroup op_history;
      coverpoint prev.opcode iff (curr.src == prev.trg);
   endgroup
```

Affect the distribution of stimulus values to increase the odds of hitting your coverage goals

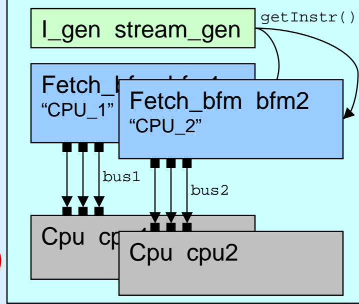
```
class I_gen_01a extends I_gen;
   Instruction i = new;
   rand bit src_equal_trg, ok;
   constraint c1 {src_equal_trg dist {0 := 2, 1 := 1};}
   e_reg prev_trg;
   virtual function Instruction getInstr(string callerId);
      prev_trg = i.trg;
      ok = randomize(src_equal_trg);
      if (src_equal_trg)
                    ok = i.randomize() with {i.src == prev_trg;};
      else          ok = i.randomize();
      return i;
   endfunction
endclass
```

>33% chance source register will equal previous target register

Controlling Stimulus for Coverage

```
class I_gen_02 extends I_gen;
   Instruction i1=new, i2=new;
   virtual function Instruction getInstr(string callerId);
      // There would be value in coordinating the instruction (streams)
      // sent to CPU_1 and CPU_2 to hit interesting LD/ST dependency cases.
      // We could write code here to create those cases and cover points
      case (callerId)
         "CPU_1":  ... //randomize an instruction for CPU_1
         "CPU_2":  ... //randomize an instruction for CPU_2
      endcase
   endfunction
endclass
module Top;
   logic clk; logic [15:0] bus1, bus2;
   I_gen_02 stream_gen = new;
   Fetch_bfm bfm1(clk, bus1), bfm2(clk, bus2);
   Cpu       cpu1(clk, bus1), cpu2(clk, bus2);
   initial fork
     bfm1.startUp("CPU_1", stream_gen, 20);
     bfm2.startUp("CPU_2", stream_gen, 20);
   join
endmodule
```

Effective stimulus is required to achieve coverage goals



What coverage should be added?

Per Instance Coverage

```
package InstrCPU;
  ...

  class Instruction;
     ...

     covergroup cover_instr;
        coverpoint opcode;
        coverpoint mode;
        ...
        option.per_instance=1;
     endgroup

       ...
  endclass : Instruction
endpackage
```

CPU_1 + CPU_2

CPU_1

CPU_2

Name	Coverage
/InstrCPU/Instruction	
TYPE cover_instr	73.2%
CROSS cover_instr::#cross__0#	90.9%
CVP cover_instr::data	33.3%
CVP cover_instr::every_other	41.7%
CVP cover_instr::mode	100.0%
CVP cover_instr::opcode	100.0%
INST VInstrCPU::Instruction::cover_instr	63.8%
CVP opcode	100.0%
CVP mode	100.0%
CVP data	33.3%
CVP every_other	22.2%
CROSS #cross__0#	63.6%
INST VInstrCPU::Instruction::cover_instr#2	61.1%
CVP opcode	83.3%
CVP mode	100.0%
CVP data	33.3%
CVP every_other	25.0%
CROSS #cross__0#	63.6%
/Top2/bfm1	
TYPE op_history	16.7%
/Top2/bfm2	
TYPE op_history	33.3%

Track coverage results separately based on the instance

CORRECT designs

Stimulus-Independent Coverage

```
module Top;
   Fetch_intfc intfc1(clk),         intfc2(clk);
   Fetch_bfm   bfm1(intfc1),        bfm2(intfc2);
   Cpu         cpu1(intfc1),        cpu2(intfc2);
   Fetch_mon   mon1 = new(intfc1), mon2 = new(intfc2);
   Coverage    covInst = new;
   initial fork
     mon1.startUp("CPU_1", covInst);
     mon2.startUp("CPU_2", covInst);
     ...
endmodule
```

Improved, more portable approach shown

Add coverage at required hierarchy levels in the testbench