Lecture 6: Design verification (basics)

Marian Verhelst (@kuleuven.be)

2021-2022

Acknowledgements to slides from N. Mentens

KU LEUVEN

KU LEUVEN micas

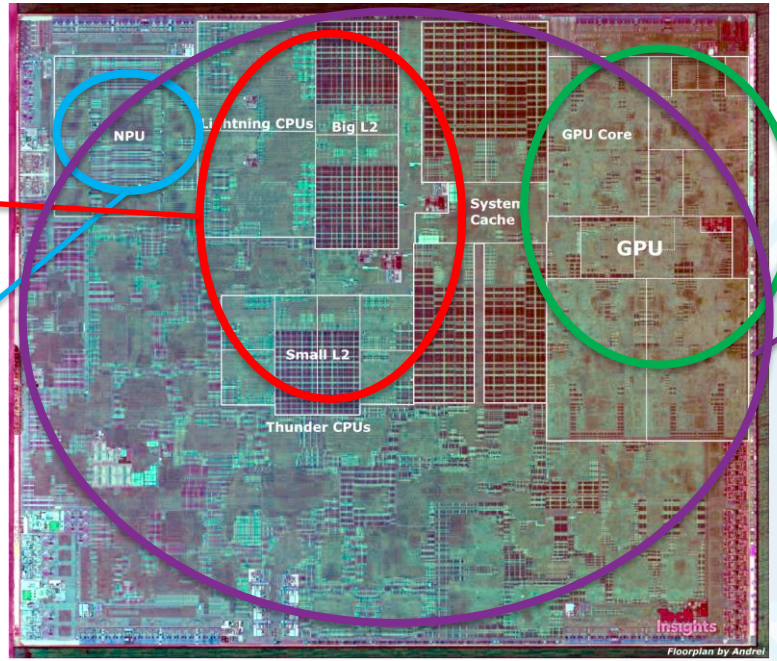
About this class

- Study advanced heterogeneous computer architectures enabling intelligent systems

Towards multi-core CPU: Why? How? (L1)

AI accelerators & HW-algorithm co-design (L3-4-5)

Industry? (L9)



iPhone 11Pro -> A13 chip

ISP, GPU & AI application (L2)

How to verify? (L6 & exercises)

Heterogeneous processor co-operation (L7)

Power/performance management (L8)

KU LEUVEN

micas KU LEUVEN

Outline lecture 6-7-8

Back to the SoC level!

- L7: Integrating heterogeneous SoCs
  - How to efficiently build and program such heterogeneous cores
  - "**Design time**"
- L6: Verifying heterogeneous SoCs
  - How to verify correct functionality of such heterogeneous cores
  - "**Verification time**"

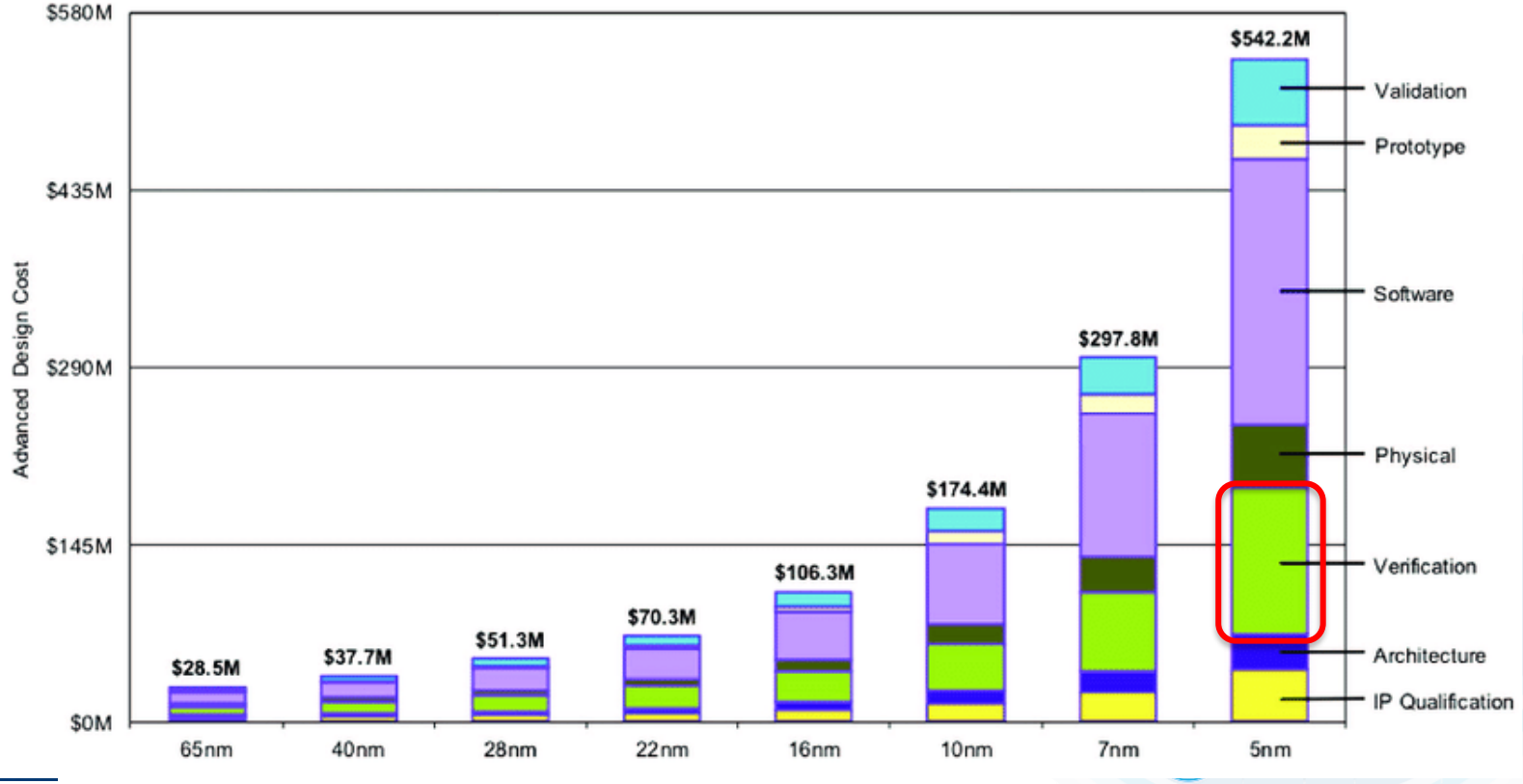
Tightly linked to exercise sessions/project

- L8: Operating heterogeneous SoCs
  - How to (adaptively) operate / manage such heterogeneous cores
  - "**Run time**"

KU LEUVEN

micas KU LEUVEN

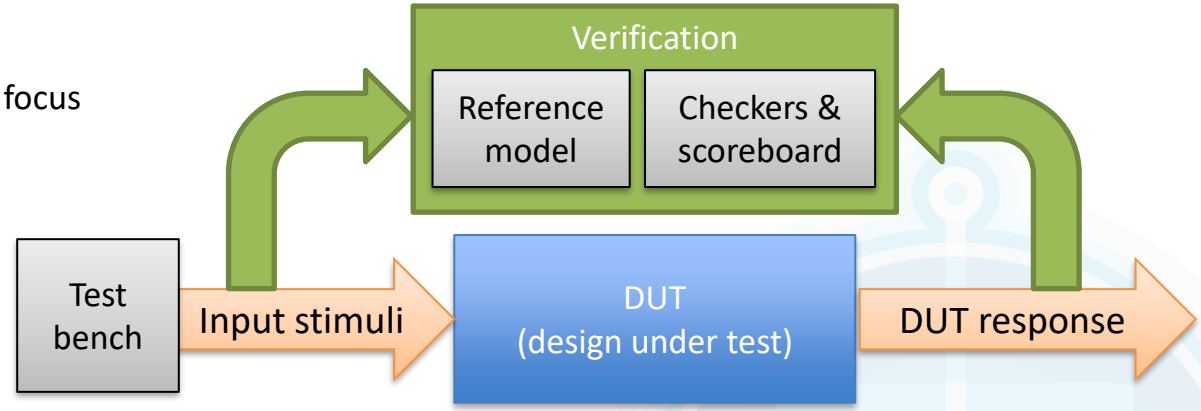
HW is only a fraction of the problem…



KU LEU

What is functional verification?

- Verifications for ASIC
  - Functional Verification ➔ our focus
  - Timing Verification
  - Formal Verification
  - Physical Verification

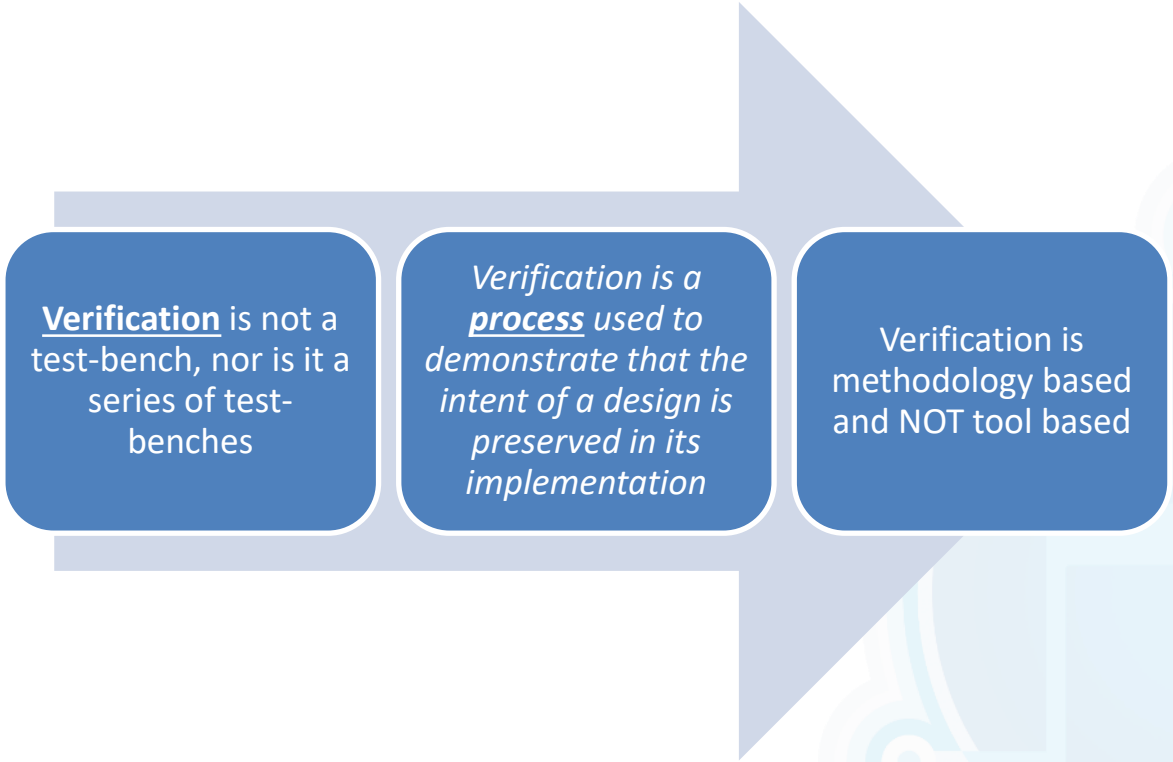


- Functional verification: Verify **correct behavior** of design according to **design intent,** before shipping to customer.
- Necessary along the complete design flow
  - Often separate test strategy at every level of design

KU LEUVEN

micas KU LEUVEN

Verification = testbench?



KU LEUVEN

KU LEUVEN micas

In dire need of a Verification Plan!

- **Need structured approach of verification**
  - Manageable in many-person team
  - Reflection of verification plan

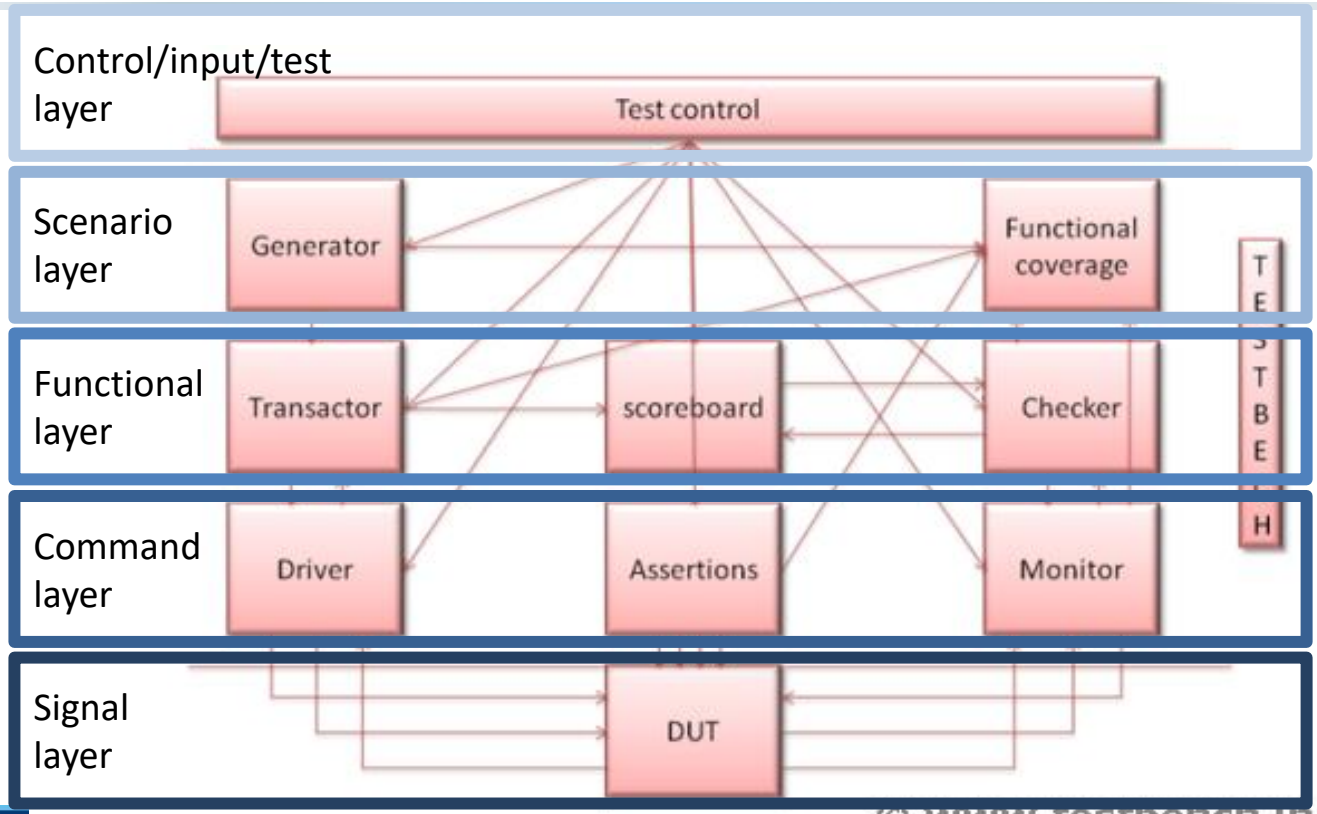
- **Reuse same test infrastructure where possible**
  - Stimuli generators
  - Reference model checkers
  - Scoreboard
  - Test coverage monitor

➔ **Layered test bench!** highly reusable across levels and designs!

Outline

- **Layered testbench structure**
- System Verilog to the rescue
  - System Verilog constructs
  - Programs, functions and tasks
  - Interfaces, modports and clocking blocks
  - Classes & mailboxes
  - Coverage and randomization
  - Outlook

Layered test bench



© www.testbench.in

KU LEUVEN

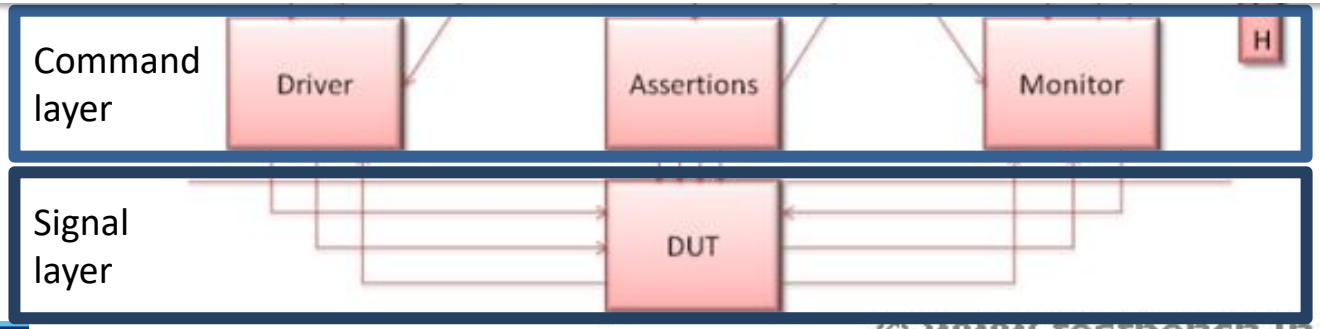
Layered test bench: Signal layer

- Contains the DUT and provides Signal-Level Connectivity in the physical representation of the DUT.

© www.testbench.in

Layered test bench: Command layer

- Provides a consistent, Low-Level Transaction Interface to the DUT, regardless of how the DUT is modeled.
- Contains physical drivers, using e.g. bus-functional models (R/W commands)
- Assertions for monitoring internals of DUT ➔ out of scope for us

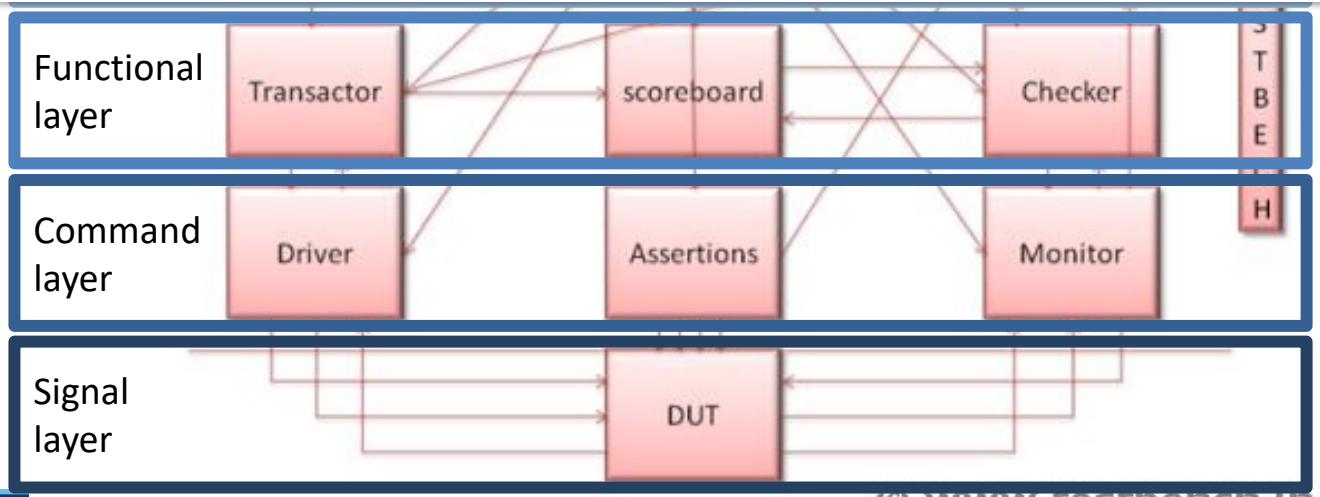


© www.testbench.in

KU LEUVEN

Layered test bench: Functional layer

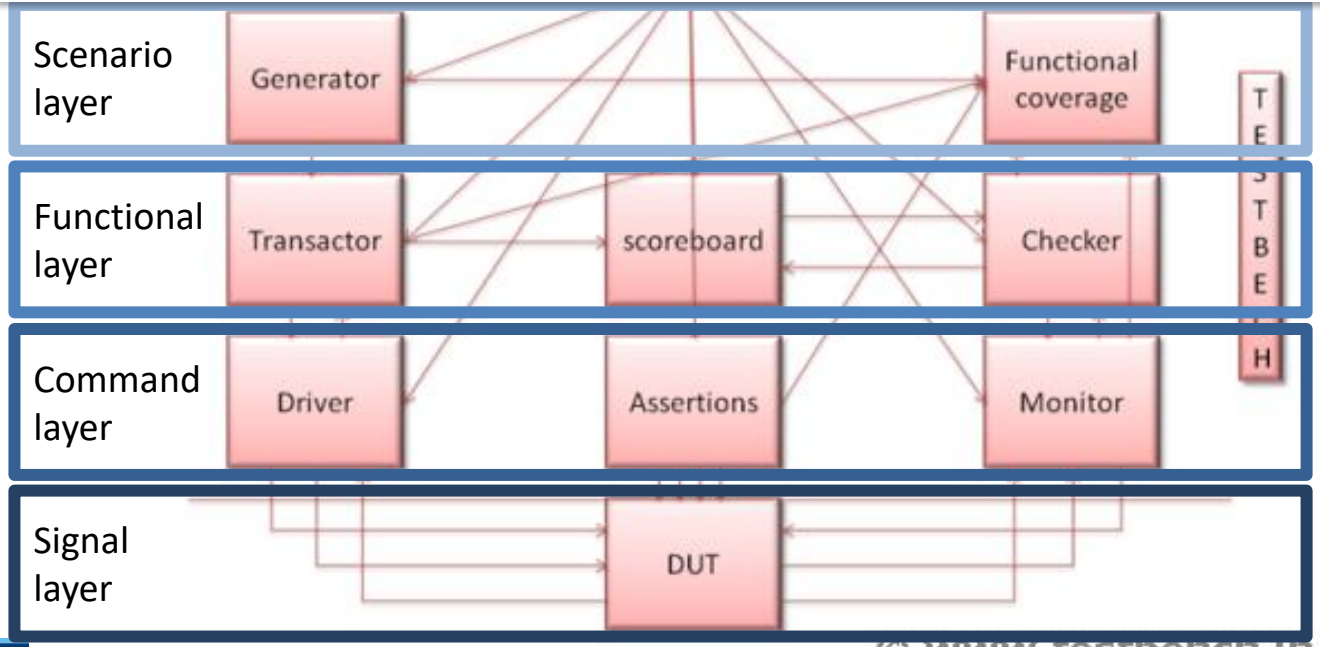
- Provides the necessary abstraction to process application-level transactions (transactor) and verify the correctness of the DUT (checker).
- Can have multiple parallel “agents”, each responsible for sub-task (e.g. agent for USB interface, agent for PCIE interface, …)
- Scoreboard uses transactor and checker info to measure how much of the original design specification have been exercised.



© www.testbench.in

Layered test bench: Scenario layer

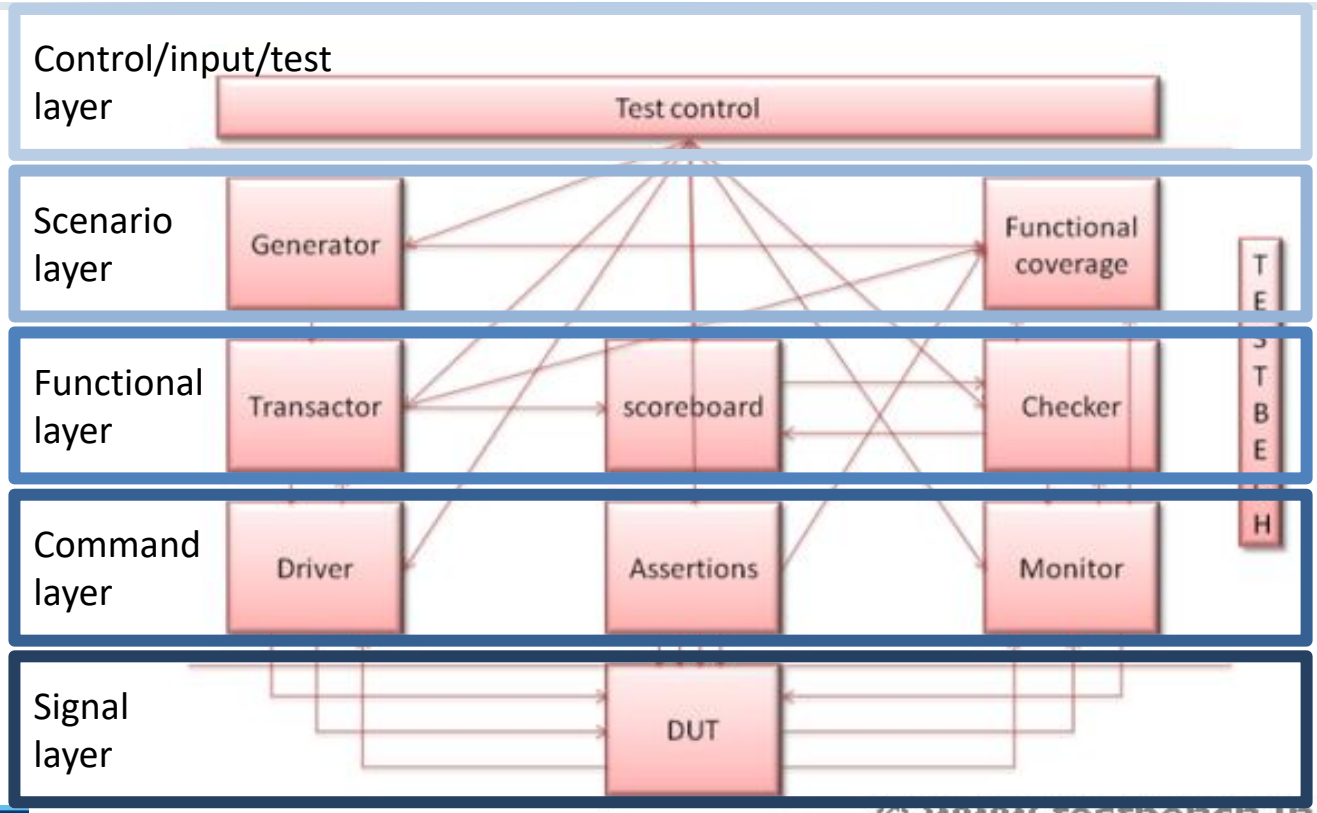
- generates the Stimulus Packets, typically constraint-driven randomized raw data that is passed to the Transactor. Check and influence coverage



© www.testbench.in

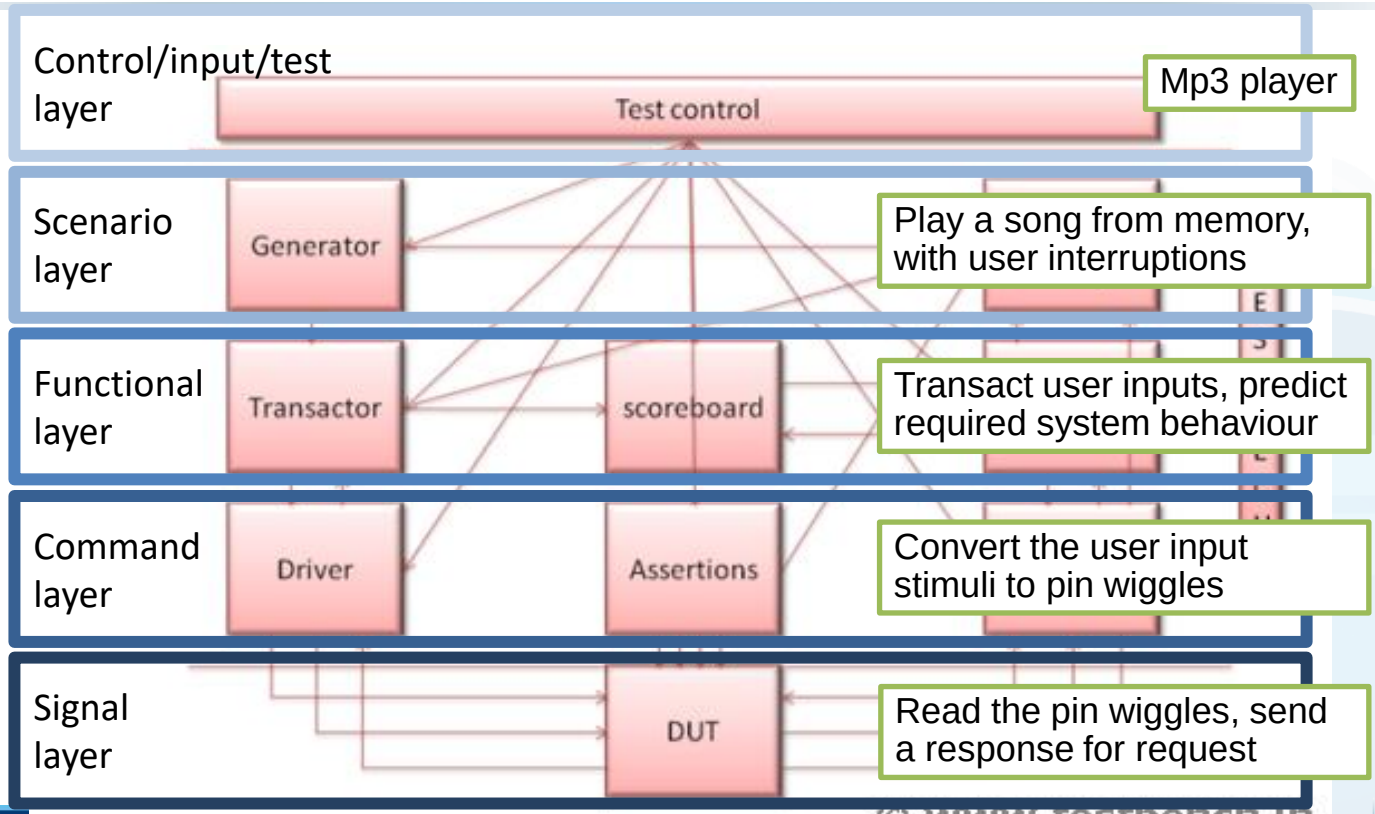
KU LEUVEN

Layered test bench: Control/input/test layer



© www.testbench.in

Layered test bench: Example



© www.testbench.in

Practical verification challenges

- How to maximize verification environment reusability?
  - Work with layered, reusable test benches
  - Work with constrained random stimuli
  - Use standard verification language

- **System Verilog based verification!**

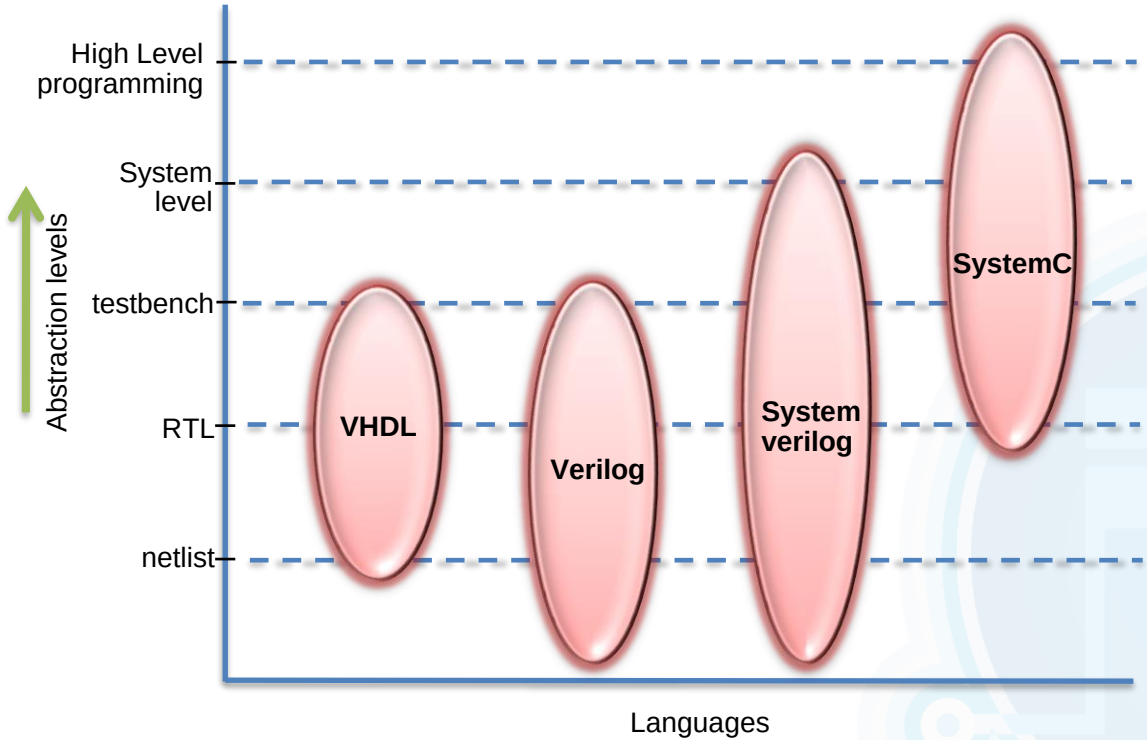
Outline

- Layered testbench structure
- **System Verilog to the rescue**
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - Coverage and randomization
  - Outlook

SystemVerilog

- Superset of Verilog, with many more features
  - Synthesizable extensions (example: interfaces, structures, etc.)
    - ➔More compact / readable HDL code
  - Non-synthesizable extensions (example: mailboxes, classes, assertions, etc.)
    - ➔Enabling highly reusable verification environments
  - Joint language for both design and verification
    - ➔ IEEE Standard,
    - ➔ Take it, run it on any SystemVerilog simulator
    - ➔ Can co-exist smoothly with Verilog, VHDL and C

Abstract levels of HDL / HVL



KU LEUVEN

micas KU LEUVEN

Outline

- The pain of verification
- Layered testbench structure
- System Verilog to the rescue
  - **Basic system Verilog constructs**
    - Example testbench
  - Programs, functions and tasks; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - DPI
  - Coverage and randomization
  - Outlook

Module

- Module = unit that can be instantiated
- Has inputs, outputs, inouts
- Internally has
  - `Initial` assignments (start of simulation)
  - Event-driven assignments
    (e.g. sequential `@(posedge clk)`)
  - Conditional event-driven assignments
    (if-and-only-if)
    (e.g. `@(posedge clk iff(valid==1))`)
  - Continuous (combinatorial) assignments
    (e.g. `assign`)

```
module adder(
  input             clk   ,
  input             reset,
  input  [3:0] a          ,
  input  [3:0] b          ,
  input        valid,
  output [6:0] c                );

logic[6:0] tmp_c;
initial   tmp_c <= 0;

  always @(posedge reset)
    tmp_c <= 0;

  // addition operation
  always @(posedge clk)
    if (valid)     tmp_c <= a + b;

  assign c = tmp_c;

endmodule;
```

Data types

- Systemverilog `logic` replaces `reg` and `wire` in Verilog (4-state – 0,1,X,Z)
  - `integer`: 32 bit (4-state)
- Systemverilog also has 2-state datatypes (0 or 1):
  - `bit`: 1 bit (2-state)
  - `byte`: 8 bit (cfr. char in C)
  - `shortint`: 16 cfr. short in C)
  - `int`: 32 bit, (cfr. int in C)
  - `longint`: 64 bit, (cfr. longlong in C)
- Default = `logic`

```
module adder(
  input             clk   ,
  input             reset,
  input  [3:0] a          ,
  input  [3:0] b          ,
  input        valid,
  output [6:0] c                  );

logic[6:0] tmp_c;
initial   tmp_c <= 0;

  always @(posedge reset)
    tmp_c <= 0;

  // addition operation
  always @(posedge clk)
    if (valid)     tmp_c <= a + b;

  assign c = tmp_c;

endmodule;
```


KU LEUVEN

micas KU LEUVEN

Blocking vs Non-Blocking assignments !

- **Blocking** (=) → immediately assigned within thread-of-execution
  - temp := '1' ; //VHDL syntax
- **Non-Blocking** (<=) → value assigned after thread suspends
  - temp <= '1' ; // VHDL syntax
  - The assignment to the left-hand side is postponed until other evaluations in the current time step are completed

```
module top1;
   int x,y,z;
   initial begin
      x = 99;     // blocking
      y = x;      // blocking
      z = y;      // blocking
      $display ("Z is ", z, "at time ", $time);
   end
endmodule
```

Z is 99 at time 0

```
module top2;
   int x,y,z;

   initial begin
      x = 99;          // blocking
      y <= x;          // Non-blocking
      z = y;           // blocking
      // # 1 z = y ;   // blocking
      $display ("Z is ", z, "at time ", $time);
   end
endmodule
```

Z is 0 at time 0

Z is 99 at time 1

KU LEUVEN

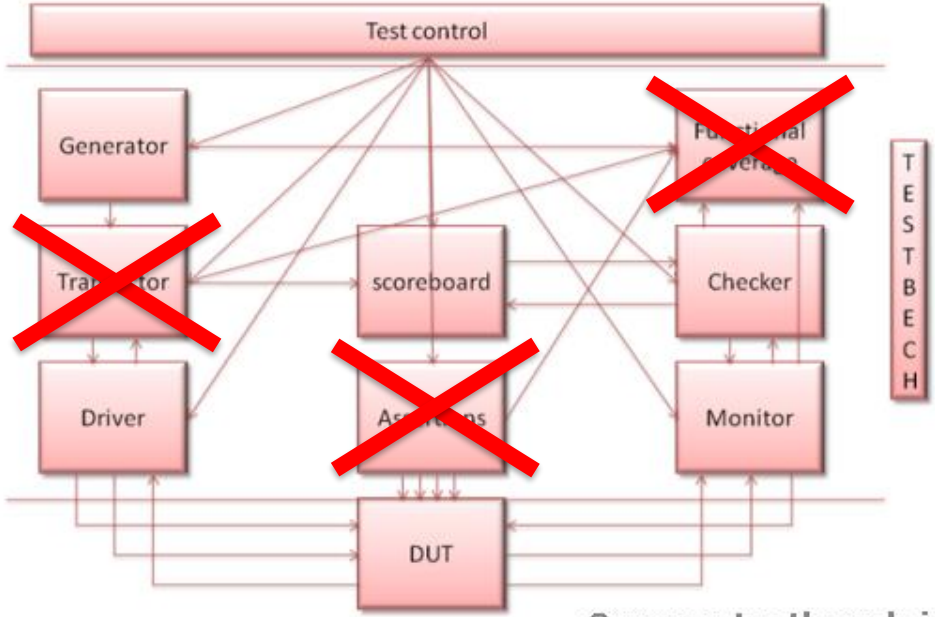
micas KU LEUVEN

Outline

- Layered testbench structure
- System Verilog to the rescue
  - Basic system Verilog constructs
    - **Example testbench**
  - Programs, functions and tasks; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - Coverage and randomization
  - Outlook

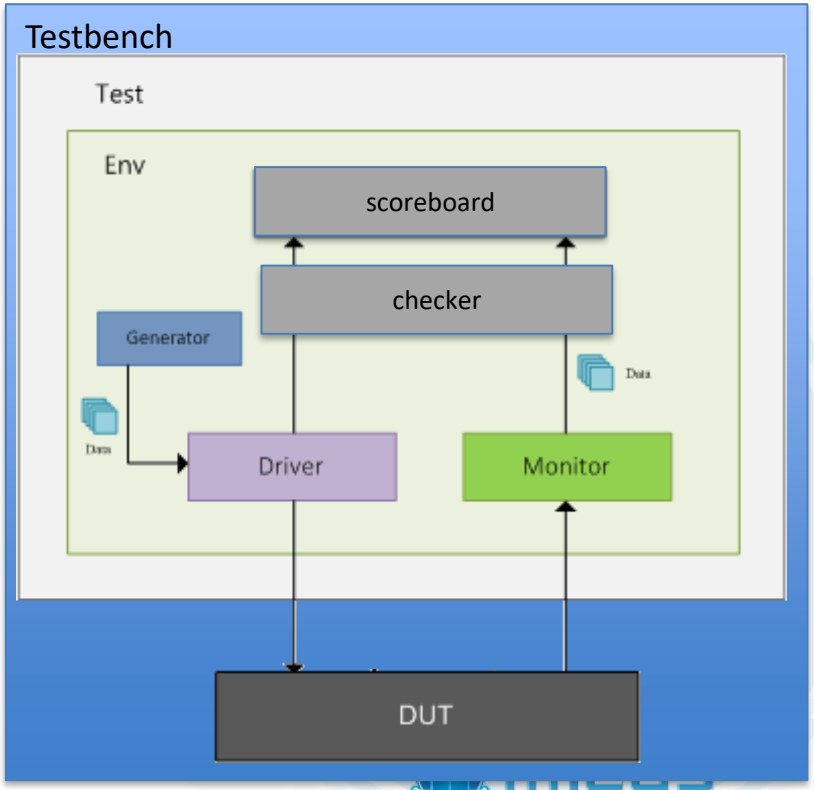
A first test environment

- Remember from start of class
- Will make (simplified version of) this, for an ADDER DUT



A first test environment

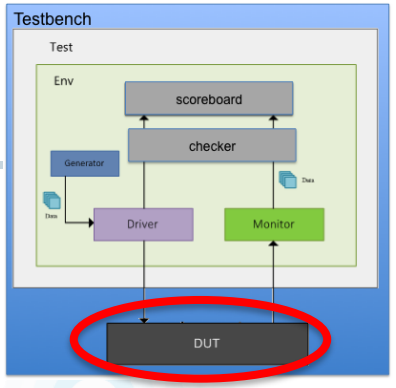
- Following components will be generated:
  - Some still quite empty…
  - Stay tuned for later…

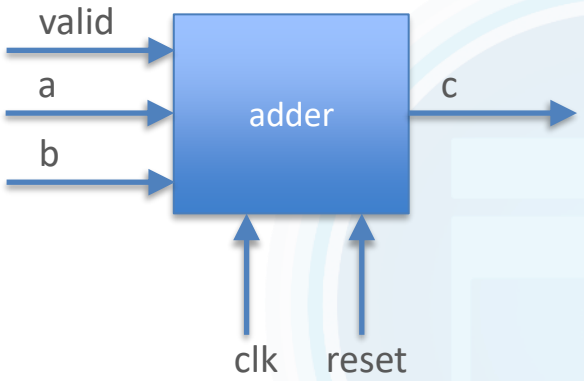


KU LEUVEN

DUT (adder with ‘valid’ signal)

```
module adder(
  input             clk  ,
  input             reset,
  input  [3:0] a         ,
  input  [3:0] b         ,
  input        valid,
  output [6:0] c                 );

 logic[6:0] tmp_c;

  //Reset
  always @(posedge reset)
    tmp_c <= 0;

  // addition operation
  always @(posedge clk)
    if (valid)     tmp_c <= a + b;

  assign c = tmp_c;

endmodule;
```



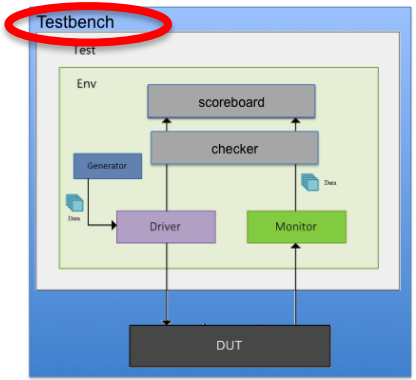


KU LEUVEN

micas KU LEUVEN

Testbench top

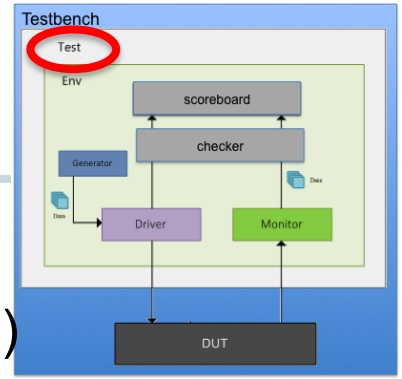
- This is the top most file, in which DUT(Design Under Test) and Verification environment are connected




```
module tbench_top;

  //clock and reset signal declaration
  bit clk;
  bit reset;
  bit  [3:0] a;
  bit  [3:0] b;
  bit  valid;
  bit [6:0] c;

  //clock generation
  always #5 clk = ~clk;

  //reset Generation
  initial begin
    reset = 1;
    #5 reset =0;
  end


  //Testcase instance
  test t1(.clk(clk),.reset(reset),
.a(a),.b(b),.valid(valid),.c(c));

  //DUT instance, interface signals are connected to the DUT ports
  adder DUT (.clk(clk),.reset(reset),
.a(a),.b(b),.valid(valid),.c(c));


  //enabling the wave dump
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule;
```


KU LEUVEN

Test (verification environment)

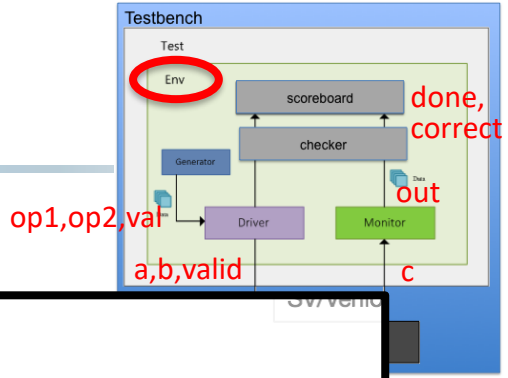


- Instantiates the right test environment
  - A VERY empty box right now (will be of more use later…)
  - Will be possible to here create different tests for random testing, for directed tests,…

```
// could have different test /environment files for random testing, directed testing,... (see later)

module test(
  input            clk  ,
  input            reset,
  input [6:0] c  ,
  output  [3:0] a       ,
  output  [3:0] b       ,
  output        valid
  );

   environment env1(.clk(clk),.reset(reset),
.a(a),.b(b),.valid(valid),.c(c));

endmodule;
```

Test environment

- Contains the instantiations of the generator, driver, monitor and scoreboard
- Signals to the DUT = physical signals
- Signals internal in the TB: can be any type…

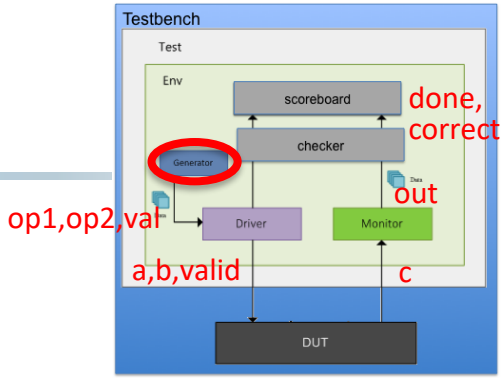



```
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "checker.sv"
`include "scoreboard.sv"

module environment (  input bit clk ,
  input bit reset,
  input bit[6:0] c  ,
  output  bit[3:0] a    ,
  output  bit[3:0] b    ,
  output     bit   valid
  );

 int op1, op2, out, val,done, correct;

  //generator and driver instance
  generator     gen(op1, op2, val);
  driver        driv(op1, op2, val, a, b, valid);
  monitor       mon(c, out);
  checker       che(op1,op2,val, out, done, correct);
  scoreboard    scb(done, correct);

endmodule;
```


KU LEUVEN

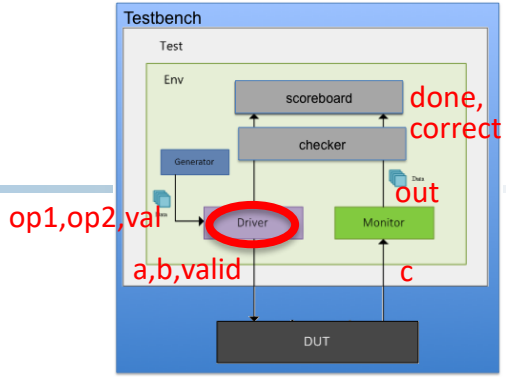
Generator

```
module generator (output int op1, op2, val);

initial begin

    op1 = 0; op2=0; val=0;
    #10ns op1=3; op2=7; val=1;

    $display("-------------------------");
    $display("- op1 = %0d, op2 = %0d",op1,op2);
    $display("-------------------------");

    #10ns val=0;

    #10ns op1=6; op2=2;val=1;

    $display("-------------------------");
    $display("- op1 = %0d, op2 = %0d",op1,op2);
    $display("-------------------------");

    #10ns val=0;

    #10ns op1=7; op2=5;val=1;

    $display("-------------------------");
    $display("- op1 = %0d, op2 = %0d",op1,op2);
    $display("-------------------------");

    #10ns val=0;

end;

endmodule;
```



- Generates the test stimuli

Driver



- Translates the test stimuli to the interface for the DUT (physical signals)
- Still very empty box now.
  - More magic will be added later!

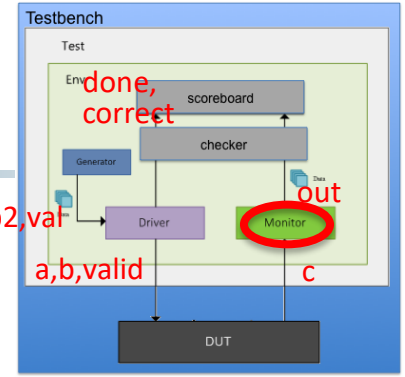
```
module driver (  input  int op1, op2, val, output bit[3:0] a, b, bit valid);

          assign a = op1;
          assign b = op2;
        assign valid = val;

endmodule;
```


KU LEUVEN

micas KU LEUVEN

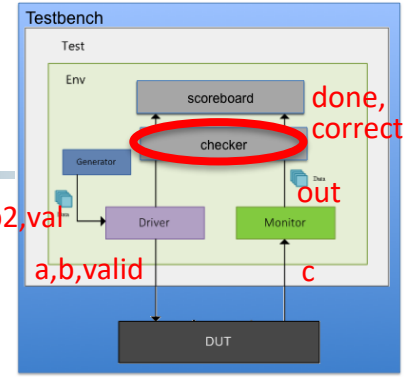
Monitor

- Captures the DUT response and translates the DUT output to the (physical signals) signals for the scoreboard
- Still very empty box now.
  - More magic will be added later!




```
module monitor(input bit[6:0] c, output  int out);

    assign out = int'(c);

endmodule
```

Checker



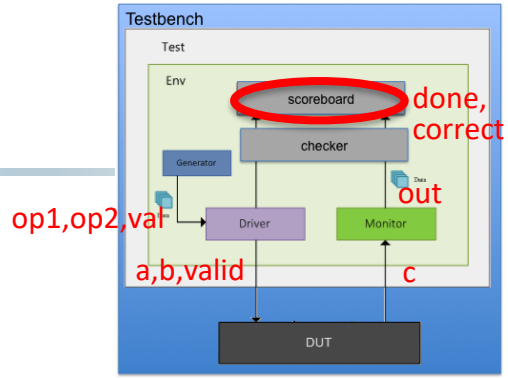
- Checks the output of the monitor, vs expected results based on the driver inputs
  - Can print warnings for errors
  - Or…

```
module checker (input int op1, op2, val, res, output int done,
correct);

  always @(res iff (val==1)) begin
      if(res == op1+op2) begin
          done = 1;
          correct = 1;
          # 1
          done = 0;
          correct = 0;
        $display("Result is as Expected"); end
        else begin
          $error("Wrong Result.\n\tExpeced: %0d Actual: %0d",
(op1+op2),res);
          done = 1;
          correct = 0;
          # 1
          done = 0; end
      end

endmodule
```


KU LEUVEN

Scoreboard

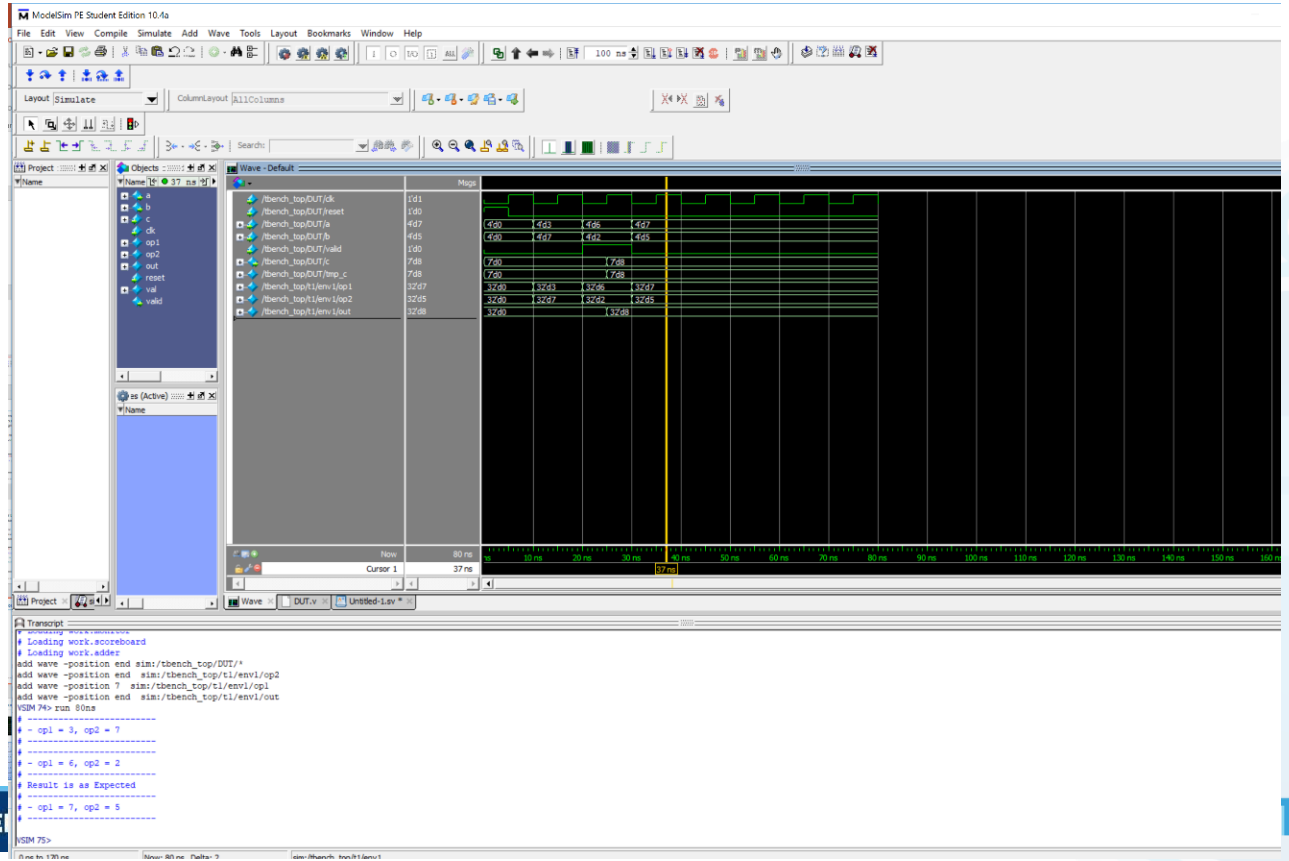
- Keeps track of
  - what part of the tests is already checked off,
  - keeps a scoring of the successful tests
  - can terminate the testbench when done



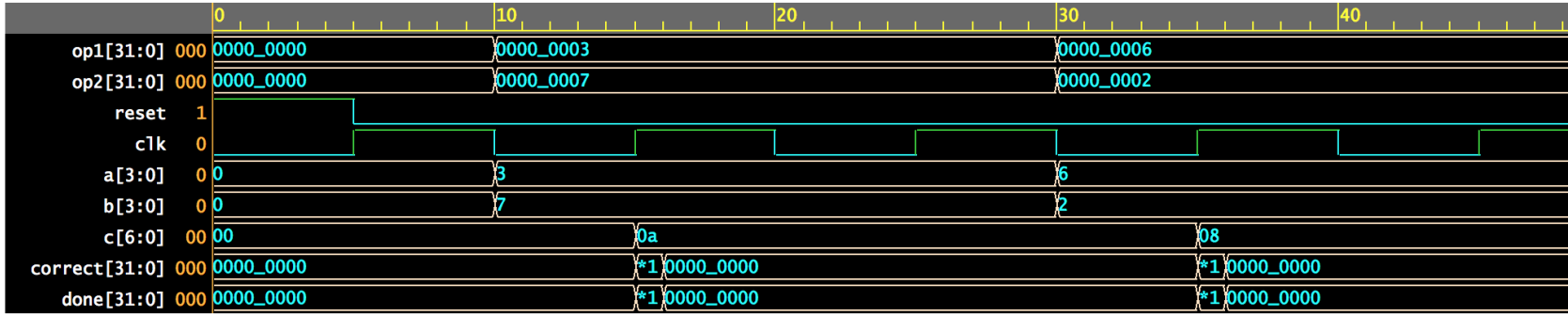

```
module scoreboard (input int done, correct);

  int total_tests = 0;
  int correct_tests=0;

  always @(posedge  done) begin
        total_tests = total_tests+1;
    if (correct==1)
            correct_tests = correct_tests + 1;
    $display("Total nb of tests = %0d ; correct nb of tests =
%0d", total_tests, correct_tests);
     end

endmodule
```


KU LEUVEN

Running simulation: tools



Running simulation: results



- Play with it yourself at: https://www.edaplayground.com/x/2DCG

```
>> run 80ns

-------------------------
- op1 = 3, op2 = 7
-------------------------
Total nb of tests = 1 ; correct nb of tests = 1
Result is as Expected
-------------------------
- op1 = 6, op2 = 2
-------------------------
Total nb of tests = 2 ; correct nb of tests = 2
Result is as Expected
-------------------------
- op1 = 7, op2 = 5
-------------------------
Total nb of tests = 3 ; correct nb of tests = 3
Result is as Expected
```


KU LEUVEN

micas KU LEUVEN

Outline

- Layered testbench structure
- System Verilog to the rescue
  - Basic system Verilog constructs
    - Example testbench
  - **Program, functions and tasks; process control**
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - Coverage and randomization
  - Outlook

Program

- **Module** describes hardware
- SystemVerilog introduces **program**, which can function like a module in testbenches
  - A program serves the purpose of providing an entry point to the execution of testbenches.
  - System task $exit terminates the program. The simulation will terminate when all program instances have exited.
- A program is similar to a module
  - it can contain ports, interfaces, final and initial statements
  - **Yet**, it can not contain an always block
- Not really necessary to use programs: can often be replaced with a module as well (unless you want to dynamically instantiate a class object)

```
program simple_program(input wire clk,
  output logic reset, enable, input logic [3:0] count);
  initial begin
    reset = 1;
    enable = 0;
     #20  reset = 0;
    @ (posedge clk);
    enable = 1;
    repeat (5) @ (posedge clk);
    enable = 0;
  end
endprogram
```

```
module simple_module();
logic clk  = 0;
always  #1  clk ++;
logic [3:0] count;
wire reset,enable;

always @ (posedge clk)
 if (reset) count <= 0;
 else if (enable) count ++;  // Simple up counter

 simple_program prg_simple(clk,reset,enable,count);  //
Program is connected like a module
 endmodule
```


KU LEUVEN

Functions

- Can be used in a module and in a program
- Can have any number of inputs, outputs, inouts, refs,…
  - default direction is input (if no direction specified)
- Returns single value – vector, real or integer
  - Return by assigning value to function name, or through `return` statement
  - Unless return type = void (no return value)
- No timing and delay allowed
  - executes in Zero time
- Synthesizable!

```
byte val, clk, temp ;
… …
function  [7:0] add ( byte a , b , c);
    add = a + b ;
    $display (“Result ", add); // 5
endfunction

always @(posedge clk)
begin
     val = add(2,3,temp);
     $display (“Add value is”, val); // 5
end
```

```
byte val, clk, temp ;
… …
function int add (byte a , b );
      return  a + b ;
endfunction

always @(posedge clk) begin
      temp = add (2,3);
      $display (temp); // 5
end
```

Tasks

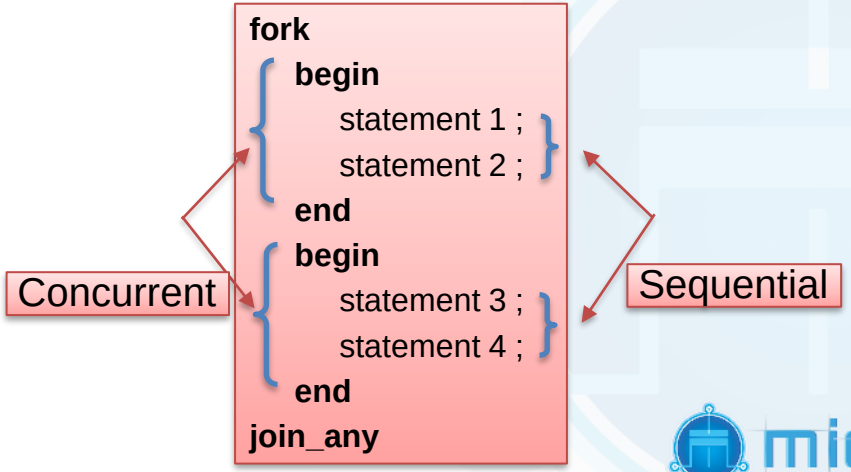
- Can be used in a module and in a program
- tasks can have any number of input, outputs, inouts, refs,…
  - default direction is `input` (if no direction specified)
  - default date type is `logic`
- tasks don't have a `return` type; a `return` statement exits the task
- tasks may contain sequential and concurrent blocks
- tasks may declare local variables
- tasks can have timing and delay (So non-synthesizable!!!)

```
initial begin
int aa, bb = 3 ; int cc;
example_task (aa,bb,cc);
end
```

```
task example_task ( a, b, output c);
#1 c = a + b ;
endtask
```

Process control

- In a program for modeling/verification purposes we often need to start concurrent processes:
- Individual statements within a **fork … join** block state at the same time and execute concurrently with each other

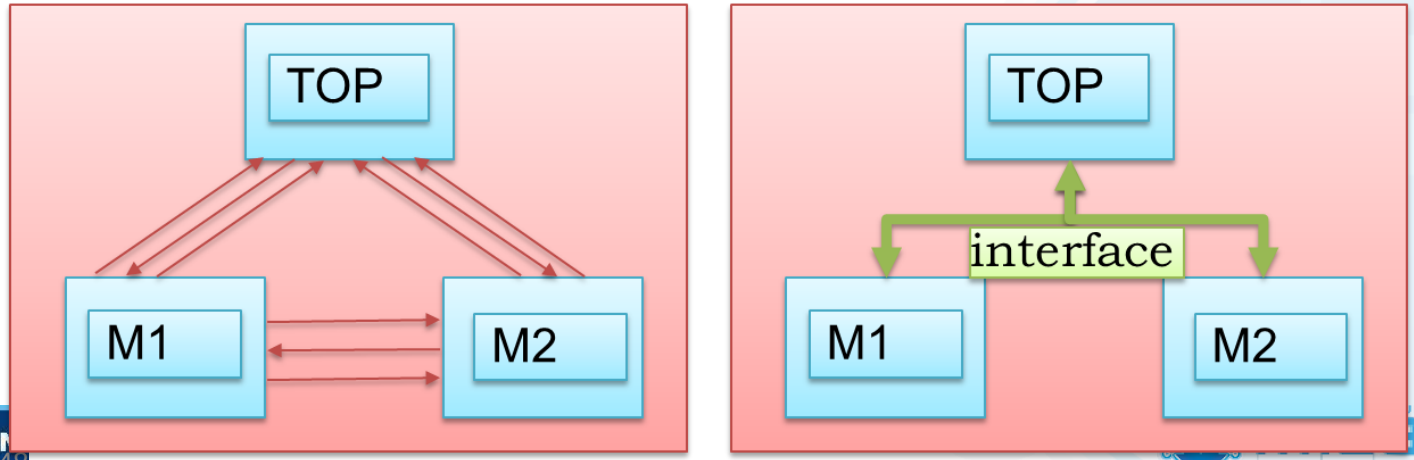


Outline

- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Program, functions and tasks; process control
  - **Interfaces, modports and clocking blocks**
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - Coverage and randomization
  - Outlook

Interfaces: Why?

- Manually connecting hundreds of ports may lead to errors
- Detailed knowledge of all the port is required
- Difficult to change if the design changes
- More time consuming
- Most port declaration work is duplicated in many modules



Interfaces: What?

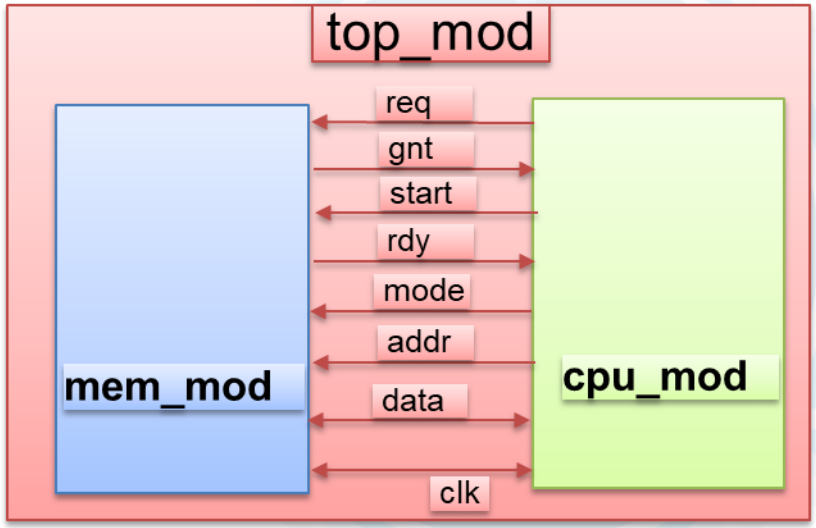
- Special construct in SystemVerilog
  - For system-level modeling and testbenches
  - Also synthesizable!
- Created to encapsulate the communication between blocks (abstraction)
- A more effective/modular/reusable way of communication between blocks of a large complex digital system
- Can be used at different abstraction levels (RTL to System Level)
- Can have task/function, processes (initial/always) and continuous assignments

Example without interfaces

```
module mem_mod ( input bit req,clk,start,
        logic [1:0] mode,
        logic [7:0] addr,
        inout wire [7:0] data,
        output bit gnt,rdy);
    logic sel_mem;
endmodule
```

```
module cpu_mod ( input bit clk,gnt,rdy,
        inout wire [7:0] data,
        output bit req,start,
        logic [1:0] mode,
        logic [7:0] addr );
    logic sel_cpu;
endmodule
```

```
module top_mod;

logic req,gnt,start,rdy;
logic clk = 0;
logic [1:0] mode;
logic [7:0] addr;
wire [7:0] data;

mem_mod mem (req,clk,start, mode, addr, data, gnt,rdy);
cpu_mod cpu (clk,gnt,rdy, data, req,start,mode,addr);
endmodule
```



Example with interfaces

Interface members accessed with dot operator

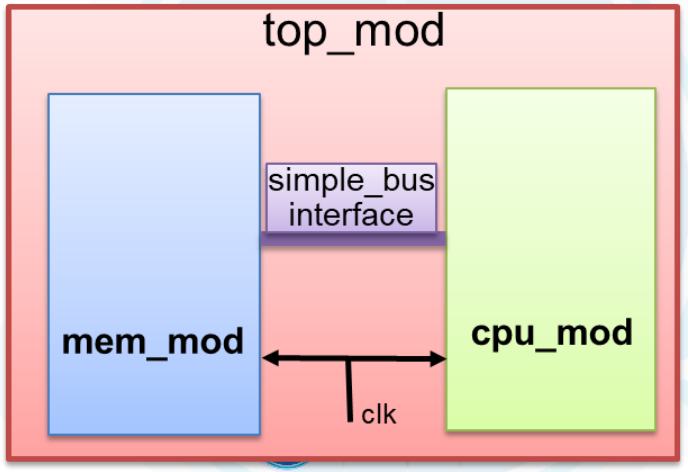
```
interface simple_bus;
    logic req,gnt,start,rdy ;
    logic [1:0] mode;
    logic [7:0] addr,data;
endinterface
```

Interface declared here is directionless

```
module mem_mod ( input bit clk,
simple_bus busa);
    logic sel_mem = 1;
    always @(posedge clk) begin
        busa.gnt <= busa.req & sel_mem;
    end
endmodule
```

```
module cpu_mod ( input bit clk,
simple_bus busa);
    logic sel_cpu;
    always @(posedge clk) begin
        busa.mode <= {busa.gnt, busa.gnt};
        busa.req <= 1 ;
    end
endmodule
```

```
module top_mod1;
    logic clk = 0;
    simple_bus sb( );

    always #5 clk = ~clk;

    cpu_mod1 cpu (clk, sb);
    mem_mod1 mem (clk, sb);

endmodule
```



Ports in Interfaces

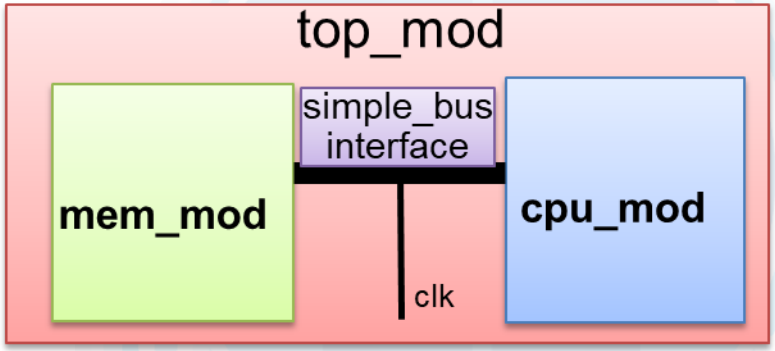
```
interface simple_bus ( input bit clk );
   logic req,gnt,start,rdy ;
   logic [1:0] mode;
   logic [7:0] addr,data;
endinterface : simple_bus
```

```
module cpu_mod1 ( interface b );
   logic sel_cpu;
   always @(posedge b.clk) begin
      b.mode <= {b.gnt,b.gnt} ;
      b.req <= 1 ; end
endmodule
```

```
module mem_mod (simple_bus a );
   logic sel_mem = 1;
   always @(posedge a.clk) begin
      a.gnt <= a.req & sel_mem;
   end
endmodule
```

- Interface can have its own ports
- Connects similar to module port
- Share an external signal
- Modules connected to it can access the member through

```
module top_mod1;
   logic clk = 0;
   simple_bus bus (clk);

   mem_mod1 mem (.a(bus));
   cpu_mod1 cpu (bus);

endmodule
```



Set interface directions ➔ Modports

- Restricts interface access within a module
  - Modport specifies accessibility and/or direction of signals w.r.t modules using them
  - Code also works without this, but prevents errors
  - Modport can be specified in module definition (e.g. master here) or in instantiation (e.g. slave here))

```
module cpu_mod1 ( simple_bus b );
   int yy;
   always @(posedge b.clk) begin
      b.gnt    <= b.start;
      b.req    <= ~b.start ;
      b.addr   <= yy++;    end
endmodule
```

```
module mem_mod1 ( simple_bus.master i );
   int cnt;
   always @(posedge i.clk) begin
      i.start    <= i.gnt & i.req;
      i.data     <= cnt++ ; end
endmodule
```

```
module top_mod1;
   logic clk = 0; simple_bus  bus(clk);
   always #5 clk = ~clk;
   mem_mod1 mem (.i(bus));
   cpu_mod1 cpu (.b(bus.slave));
endmodule
```

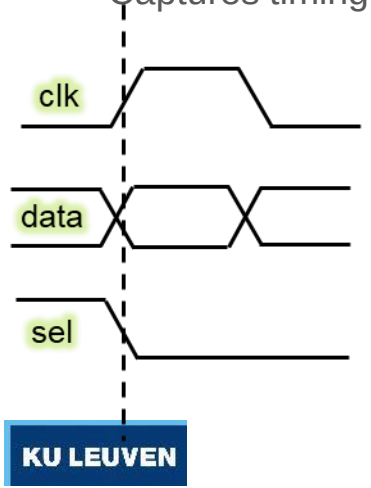
```
interface simple_bus (input bit clk) ;
   logic req,gnt,start,rdy ;
   logic [1:0] mode;
   logic [7:0] addr;
   logic [16:0] data;
   modport master ( input req,gnt,clk,mode,addr,output data, start, rdy);
   modport slave  (output req,gnt,mode,addr,input data, clk, start, rdy);
endinterface
```

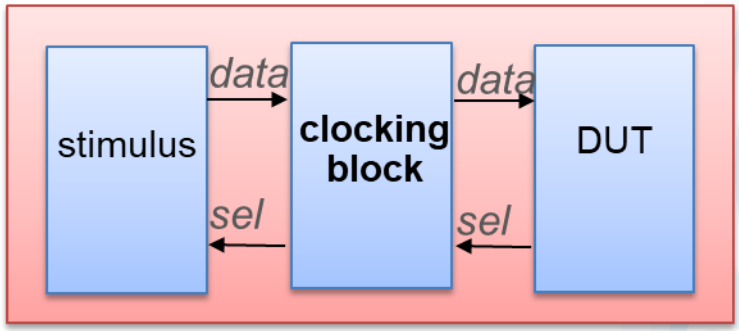
Clocking Block

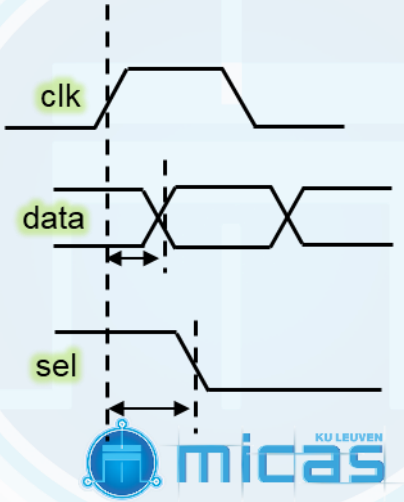
- An interface does not specify timing, synchronization requirements or clocking paradigms
- What is TB generates stimuli @clk and slave reads them @clk?

➔ RACE!

- SystemVerilog adds clocking block
  - Identifies clock signals
  - Allows to insert delays on input and outputs of interface… (cfr. setup and hold time)
  - Captures timing and synchronization requirements`





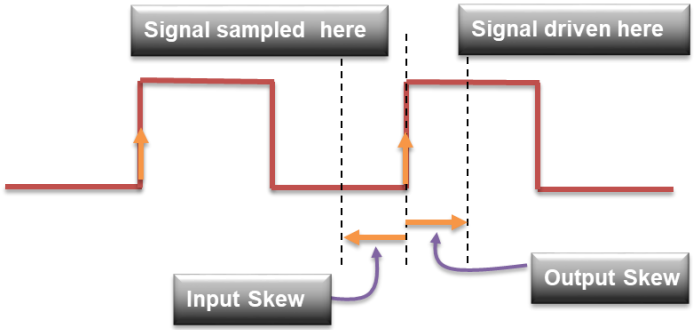


KU LEUVEN

micas KU LEUVEN

Clocking block

- Clocking blocks add drive timing and sample timing to signals
  - relative to a specific clock
  - clocked and unclocked outputs possible
  - default input skew is 1step and output skew is 0
- Helps to avoid, clock/data race (setup/hold time issues)
- Can be integrated in an interface for arrange its timing ➔ see next testbench!
- Allows statements using @(simple_bus.cb); in module using interface




```
interface simple_bus (input bit clk) ;
    logic req,gnt,start,rdy ;
    logic [1:0] mode; logic [7:0] addr; logic [16:0] data;

  clocking cb @(posedge clk)); // clocking block with clocking event
    default input#1ns output#2ns; // default timing skew for input/output
                                  (can be overwritten)
    input data, start, rdy;   output req,gnt,clk,mode;
    output #3ns addr;          // overwrite default output delay
  endclocking

    modport master ( input req,gnt,clk,mode,addr,output data, start, rdy);
    modport slave  (clocking cb);
endinterface
```

Outline

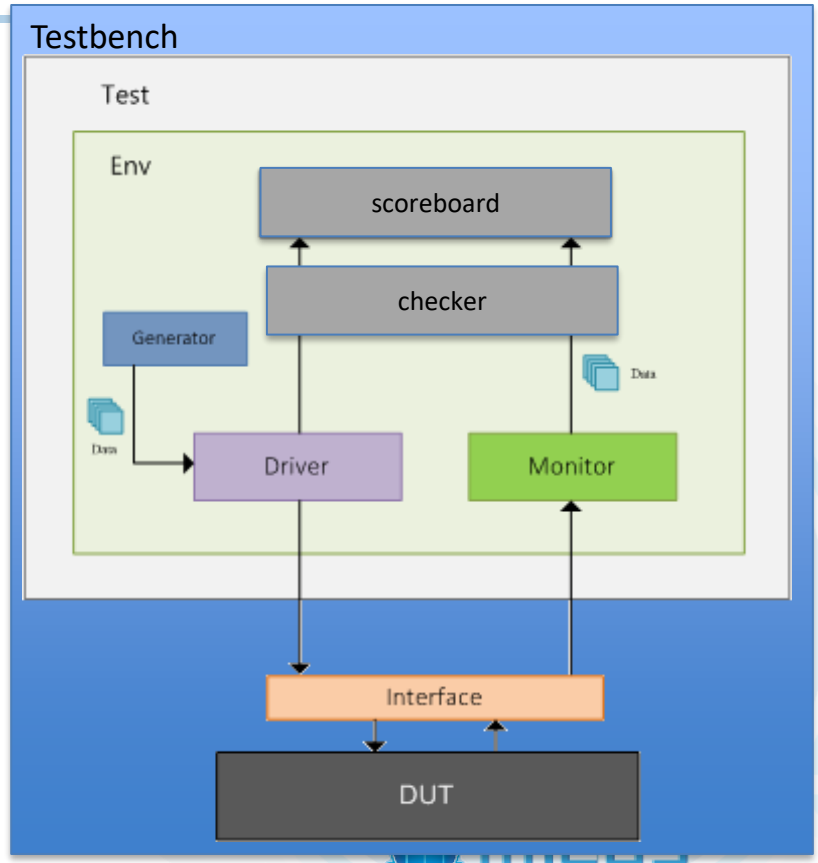
- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks ; process control
  - Interfaces, modports and clocking blocks
    - **Example testbench**
  - Classes & Mailboxes
    - Example testbench
  - Coverage and randomization
  - Outlook

KU LEUVEN

KU LEUVEN micas

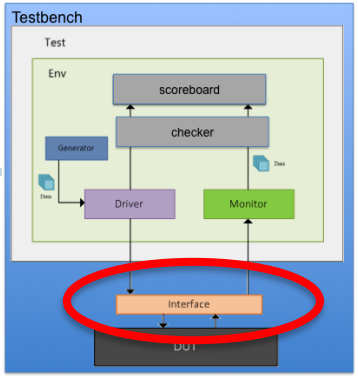
More advanced test environment

- Same DUT (adder)
- Adding:
  - Tasks
  - Interface
  - Clocking block



KU LEUVEN

Interface



- Interface between DUT and testbench
  - Clocking block to generate realistic signals
  - Modport for testbench

```
interface intf(input logic clk,reset);

  //declaring the signals
  logic       valid;
  logic [3:0] a;
  logic [3:0] b;
  logic [6:0] c;

// clocking block
  default clocking cb @(posedge clk);
    default input #3ns output #2ns;
    output a,b,valid;
    input negedge c;
  endclocking ;

 // modport dut (input a,b,valid, clk, reset, output c );
  modport tb (clocking cb); // synchronous testbench

endinterface;
```


KU LEUVEN

micas KU LEUVEN

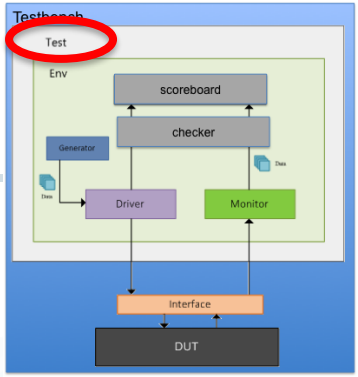
Testbench top

- Instantiates interface between DUT and test!
  - Indicates that testbench uses modport with clocking block

```
//including interface file
`include "interface.sv"

//including testcase file
//Particular testcase can be run by uncommenting, and commenting the rest
`include "test.sv"
//`include "directed_test.sv"
//------------------------------------------------------------------

module tbench_top;

  //clock and reset signal declaration
  bit clk;
  bit reset;

  //clock generation
  always #5 clk = ~clk;

  //reset Generation
  initial begin
    reset = 1;
    #5 reset =0;
  end

  //creatinng instance of interface, inorder to connect DUT and testcase
  intf i_intf(clk,reset);

  //Testcase instance, interface handle is passed to test as an argument
  test t1(i_intf.tb);

  //DUT instance, interface signals are connected to the DUT ports
  adder DUT (
    .clk(i_intf.clk),
    .reset(i_intf.reset),
    .a(i_intf.a),
    .b(i_intf.b),
    .valid(i_intf.valid),
    .c(i_intf.c)
   );

  //enabling the wave dump
  initial begin
    $dumpfile("dump.vcd"); $dumpvars;
  end
endmodule;
```


KU LEUVEN

Test (verification environment)



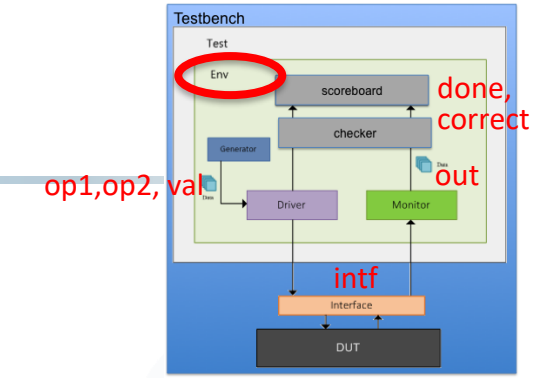
- Instantiates the right test environment
- Start the desired test **tasks** (“run”, see next slide)

```
//including interfcae file
`include "interface.sv"

// could have different test /environment files for random testing, directed testing,... (see later)

module test(intf i_intf);

environment env1(i_intf);

  initial begin env1.run(); end
endmodule;
```


KU LEUVEN

```
`include "generator.sv"
`include "driver.sv"
`include "monitor.sv"
`include "checker.sv"
`include "scoreboard.sv"

module environment (intf i_intf);
 int op1, op2, val, out, done, correct;

  //generator and driver instance
  generator     gen(op1, op2, val);
  driver        driv(op1, op2, val, i_intf.tb);
  monitor       mon(i_intf.tb,out);
  checker       che(op1,op2,val,out,done,correct);
  scoreboard    scb(done,correct);

  task pre_test();
    driv.reset();
  endtask

  task test();
    fork
      gen.main();
      driv.main();
      mon.main();
      che.main();
      scb.main();
    join_any
  endtask

  task post_test();
    wait(gen.ended.triggered);
  endtask

  //run task
  task run;
    pre_test();
    test();
    post_test();
    $finish;
  endtask
endmodule;
```

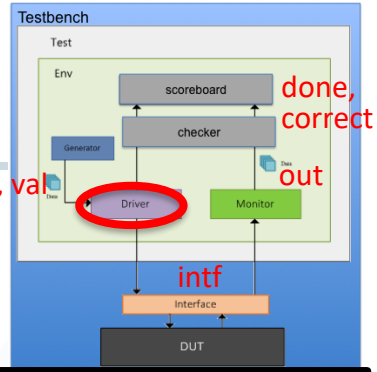
Test environment



- Contains the instantiations of the generator, driver, monitor, checker and scoreboard
- Note: using tasks for starting them up!

Driver

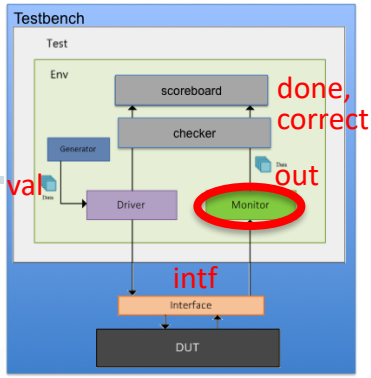
- Generator: no change
- Translates the test stimuli to the interface for the DUT (physical signals)
- Sends these to **clocking block** in interface! ➔ create synchronous signals for the DUT




```
module driver (  input  int op1, op2, val, intf intf_out);


  //Reset task, Reset the Interface signals to default/initial values
  task reset;
    $display("[ DRIVER ] ----- Reset Started -----");
  intf_out.cb.valid <= 0;
  intf_out.cb.a <= 0;
  intf_out.cb.b <= 0;repeat (1) @(intf_out.cb);
    $display("[ DRIVER ] ----- Reset Ended   -----");
  endtask

  //drivers the data items items to interface signals
  task main;
      forever begain
        @  (intf_out.cb)    begin
     intf_out.cb.a <= op1;
     intf_out.cb.b <= op2;
     intf_out.cb.valid <= val;
    end
    end
  endtask

endmodule;
```


KU LEUVEN

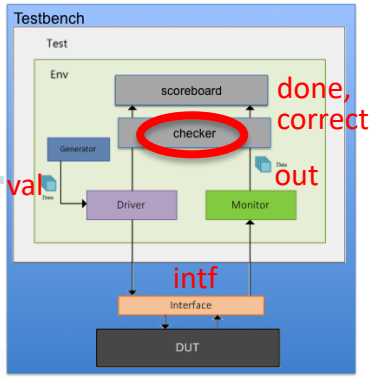
Monitor



- Captures the DUT response and translates the DUT output to the (physical signals) signals for the scoreboard
- Gets these from **clocking block** in interface!
  ➔ read synchronous signals for the DUT

```
module monitor(intf intf_in, output int out);


  //Samples the interface signal and send the sample packet to checker
  task main;
    forever begin
      @ (intf_in.cb.c)
              out<=intf_in.cb.c;
    end
  endtask

endmodule
```

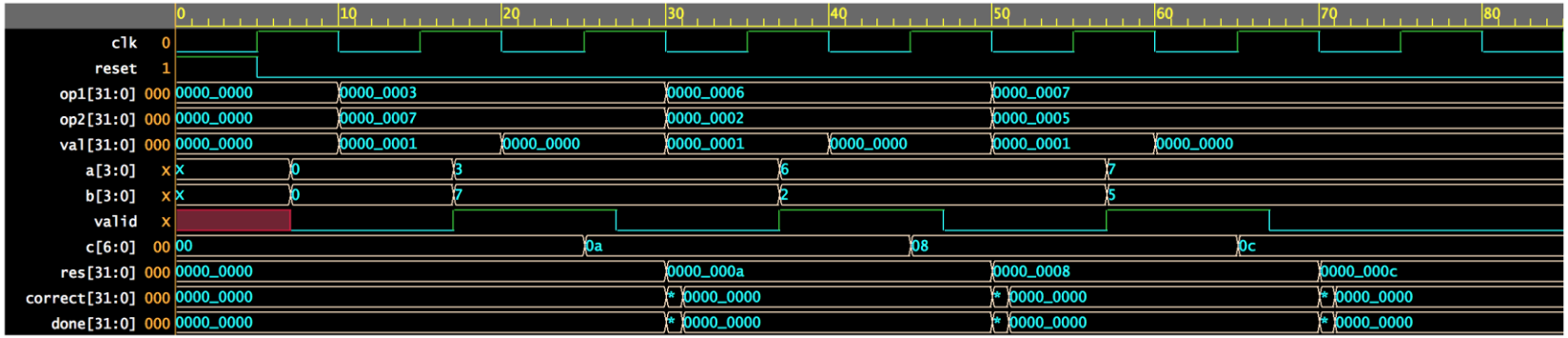
Checker




```
module checker (input int op1, op2, val, res, output int done, correct);

  int op1_d, op2_d, val_d, op1_dd, op2_dd, val_dd, op1_ddd, op2_ddd, val_ddd;
  assign #5 op1_d = op1;
  assign #5ns op2_d = op2;
  assign #5ns val_d = val;
  assign #5ns op1_dd = op1_d;
  assign #5ns op2_dd = op2_d;
  assign #5ns val_dd = val_d;
  assign #5ns op1_ddd = op1_dd;
  assign #5ns op2_ddd = op2_dd;
  assign #5ns val_ddd = val_dd;

  task main;
    forever begin
      @(res iff (val_ddd==1))
      if(res == op1_ddd+op2_ddd) begin
          done = 1;
          correct = 1;
          $display("Result is as Expected"); end
        else begin
          done = 1;
          correct = 0;
          $error("Wrong Result.\n\tExpeced: %0d Actual: %0d",(op1_ddd+op2_ddd),res); end
        # 1
        done = 0;
        correct = 0;
    end
  endtask

endmodule
```

Note our need for many delayed signals here ➔ solve with mailboxes!

Running simulation



- Do you see the impact of the clocking block?
  - tb.op1➔ dut.a
  - dut.c ➔ tb.res
- Play with it yourself at: https://www.edaplayground.com/x/fQt

```
[ DRIVER ] ----- Reset Started -----
[ DRIVER ] ----- Reset Ended   -----
-------------------------
- op1 = 3, op2 = 7
-------------------------
-------------------------
- op1 = 6, op2 = 2
-------------------------
Result is as Expected
Total nb of tests = 1 ; correct nb of tests = 1
-------------------------
- op1 = 7, op2 = 5
-------------------------
Result is as Expected
Total nb of tests = 2 ; correct nb of tests = 2
Result is as Expected
Total nb of tests = 3 ; correct nb of tests = 3
$finish called from file "environment.sv", line 46.
$finish at simulation time                90
```


KU LEUVEN

micas KU LEUVEN

Outline

- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks ; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - **Classes & Mailboxes**
    - Example testbench
  - Coverage and randomization
  - Outlook

KU LEUVEN

KU LEUVEN micas

Classes

- SystemVerilog provides Object Oriented Class Programming
  - To simplify reuse of components!
- Classes define data and tasks/functions that can operate on data
- Class objects can be dynamically created/deleted/assigned/accessed in simulation
- Several class objects are predefined = useful!!

```
class user_class;
    int data_var;

  function new(input int a);
     data_var = a;
   endfunction
  function int get;
     return data_var ;
   endfunction
   task set (int temp);
     data_var = temp ;
    endtask
  endclass
```

```
user_class c1;
c1 = new(2);
… …
c1.data_var = 22;
c1.set (44);
$display (c1.get());
… …
```

declaring a class object (null pointer)

creating a class object

Assigning a class property directly

Assigning a class property through the class method(task)

Retrieving a class property through the class method (function)

Mailbox

- Predefined class object, simplifying communication between class instances
- Can be seen as a FIFO (first in – first out)
- Predefined methods to push/pull things in mailbox

➔ *See example in testbench!*

new() : **function** new (**int** bound = 0); *// default bound 0*
mailbox constructor, optionally specifies the max size

num() : **function** int num();
returns the number of messages currently in mailbox

put() : **task** put (singular message);
places a message in mailbox; blocks if mailbox is full

get() : **task** get (**ref** singular message);
retrieves message from mailbox; blocks if mailbox empty; error if type mismatch

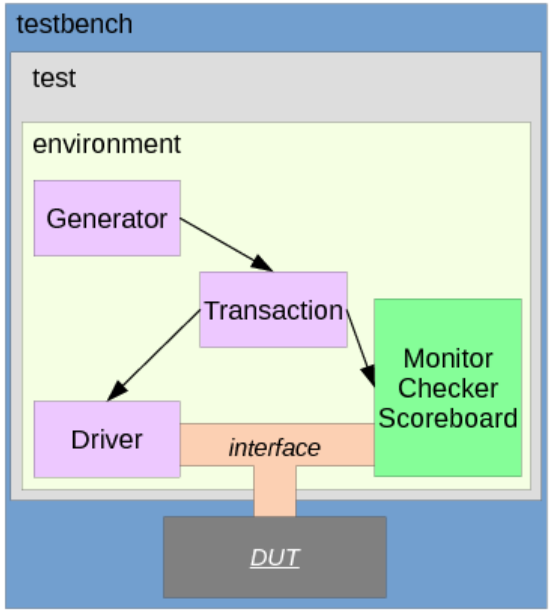
peek() : **task** peek (**ref** singular message);
copies message from mailbox; blocks if mailbox empty; error if type mismatch

Outline

- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks ; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - **Example testbench**
  - Coverage and randomization
  - Outlook

Example test environment

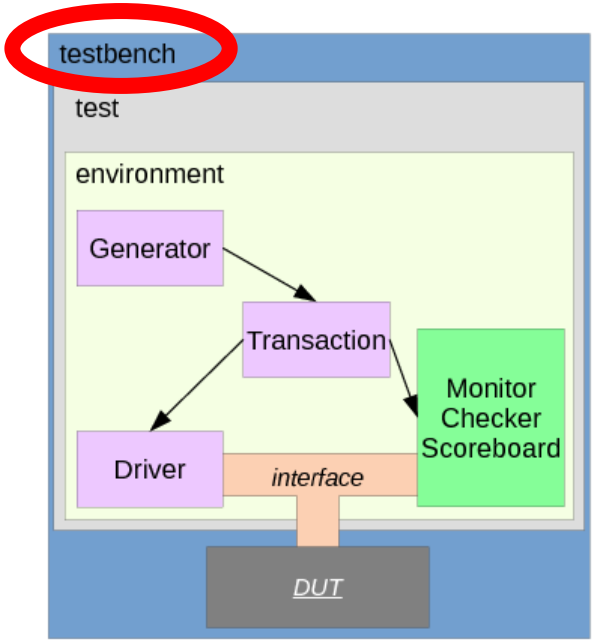
- Same DUT (adder)
- Adding:
  - Classes
- Interface and DUT untouched…



Testbench

- Testbench is more-or-less unaltered
- Reset signal is no longer generated
  - It is now driven by the Driver instead of by the testbench

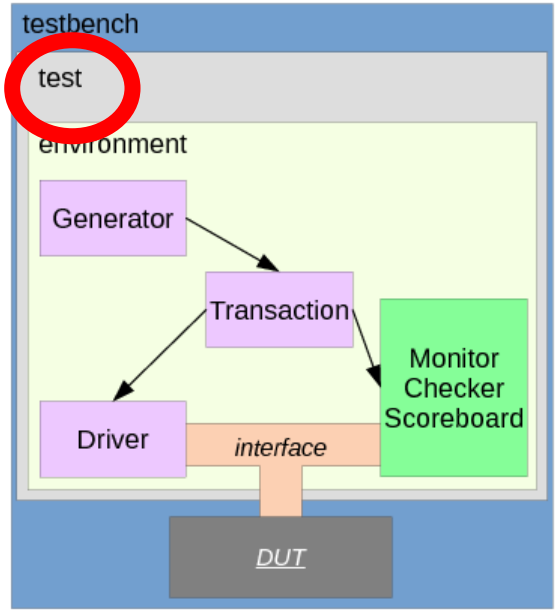
```
`include "DUT.v"
`include "interface.sv"
`include "test.sv"

module tbench_top;
  bit clk;
    intf i_intf(clk); // interface

  always #5 clk = ~clk;       // clock generation

  test t1(i_intf.tb);
  adder DUT (
    .clk(i_intf.clk),     .reset(i_intf.reset),
    .a(i_intf.a),     .b(i_intf.b),
    .valid(i_intf.valid),     .c(i_intf.c));
endmodule : tbench_top
```



Test

- Test is now a program
- Includes environment class
- Creates “new” environment class instance

```
`include “interface.sv”
`include "Environment.sv"

program test(intf i_intf);

  Environment env1 = new(i_intf);

  initial
  begin
    env1.run();
  end

endprogram : test
```

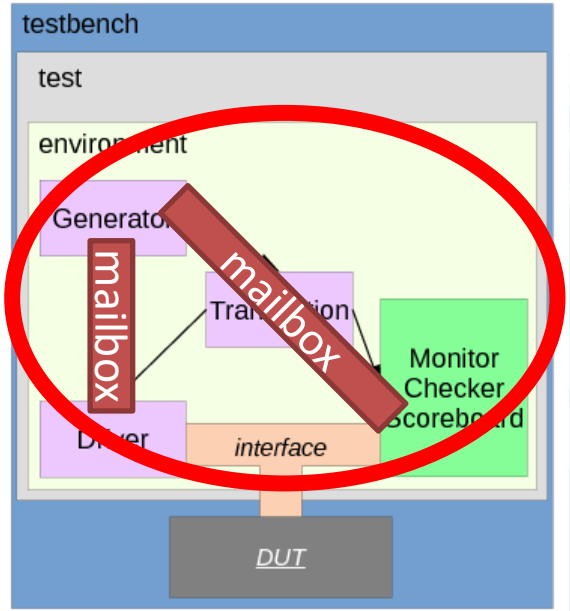


Environment

```
… // includes
class Environment;
   virtual intf ifc;
   Generator gen;
   Driver drv;
   MonCheckScore mcs;
   mailbox #(Transaction) gen2drv;
   mailbox #(Transaction)gen2mcs;

 function new(virtual intf i);
     ifc = i;
     gen2drv = new(5);
     gen2mcs = new(5);
     gen = new(gen2drv, gen2mcs);
     drv = new(i, gen2drv);
     mcs = new(i, gen2mcs);
 endfunction : new

 task run;
   $display("[ENV] start-of-life");
   pre_test();
   test();
   post_test();
   $display("[ENV] end-of-life");
   repeat (100) @(posedge ifc.clk);
   $finish;
 endtask
```

```
task pre_test();
 drv.reset();
endtask
```

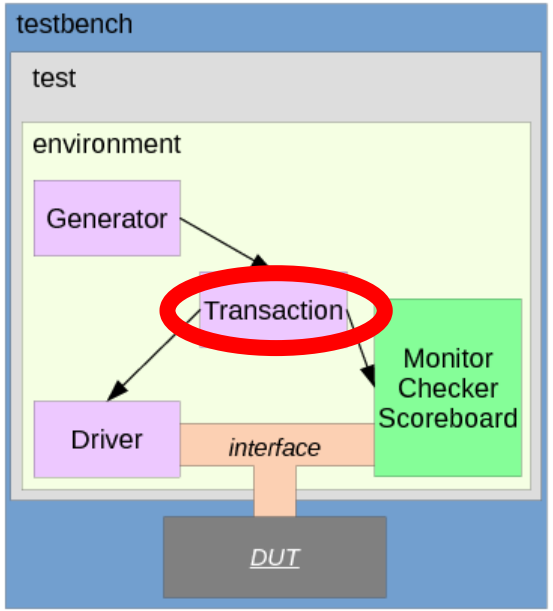
```
task test();
  fork
    gen.run();
    drv.run();
    mcs.run();
  join_none
endtask
```



Transaction (and randomization)

- The Transaction class groups variables that represent a single transaction
- op1 and op2 are **randomizable (see later)**!!

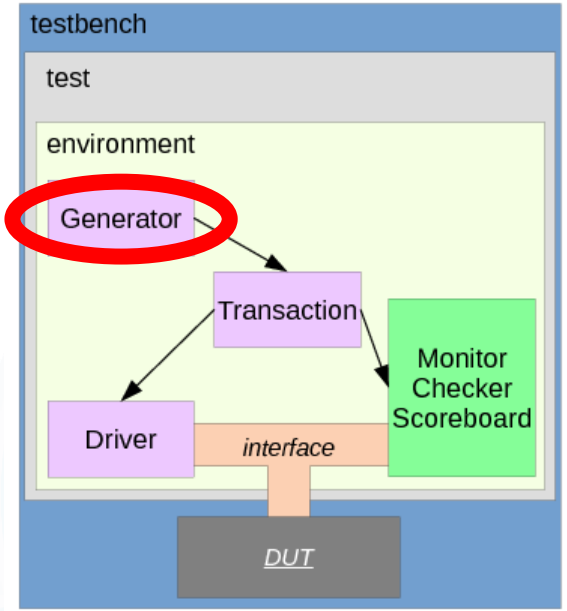
```
`ifndef SV_TRANSACTION
`define SV_TRANSACTION

class Transaction;
  rand int op1, op2;

  function new();
    op1=0;
    op2=0;
  endfunction : new

endclass : Transaction
`endif
```



Generator

```
`include "Transaction.sv"

class Generator;
  mailbox #(Transaction) gen2drv;
  mailbox #(Transaction) gen2mcs;
  Transaction tract;

  function new(#(Transaction) g2d, mailbox #(Transaction) g2m);
        gen2drv = g2d;
        gen2mcs = g2m;
  endfunction : new

  task run();
    #10;              // wait some time so the Driver is listening
    for(int i=0;i<NO_tests;i++)
    begin
      void'(tract.randomize());
      $display("[GEN] generated new operands (%0d, %0d)", tract.op1, tract.op2);

      gen2drv.put(tract);
      gen2mcs.put(tract);

  endtask : run
endclass : Generator
```

Generates random transactions
Pushes them in mailbox of driver and mcs



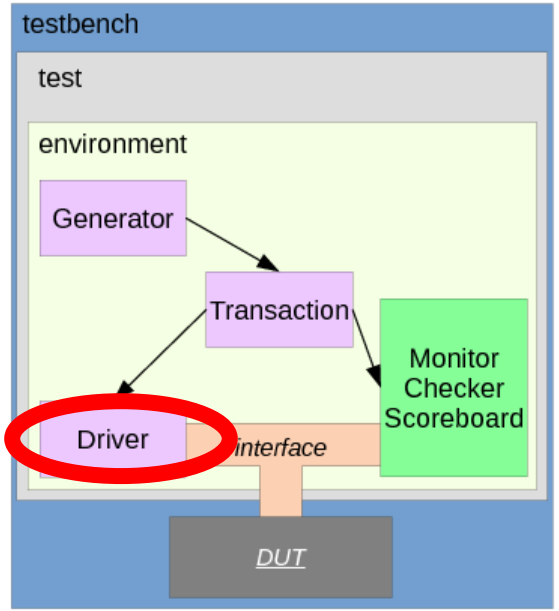
randomize = predefined task on every class
➔ gives rand values to the class rand variables

KU LEUVEN

Driver

```
… // includes
class Driver;
   virtual intf ifc;   // virtual, as interface is described
and linked elsewhere
   mailbox #(Transaction) gen2drv;

 function new(virtual intf i, mailbox #(Transaction)g2d);
      this.ifc = i;
      gen2drv = g2d;
 endfunction : new

 task reset;
   $display("[DRV] -- Reset Started --");
   ifc.reset <= 1;
   ifc.cb.valid <= 0;
   ifc.cb.a <= 0;
   ifc.cb.b <= 0;
   $display("[DRV] -- Reset Ended --");
   repeat (10) @(posedge ifc.clk);
   ifc.reset <= 1'h0;
 endtask
```

```
 task run();
   Transaction tract;
   int rv;

   $display("[DRV] run started");
   forever
   begin
     gen2drv.get(tract);
     $display("[DRV] a set to %0d, b set to %0d,
op set to %0d", tract.a, tract.b, tract.op);
       ifc.cb.a <= tract.op1;
       ifc.cb.b <= tract.op2;
       ifc.cb.valid <= 1;
       @(posedge ifc.clk)
       ifc.cb.valid <= 0;
      ....
     end
   end
 endtask : run
endclass : Driver
```

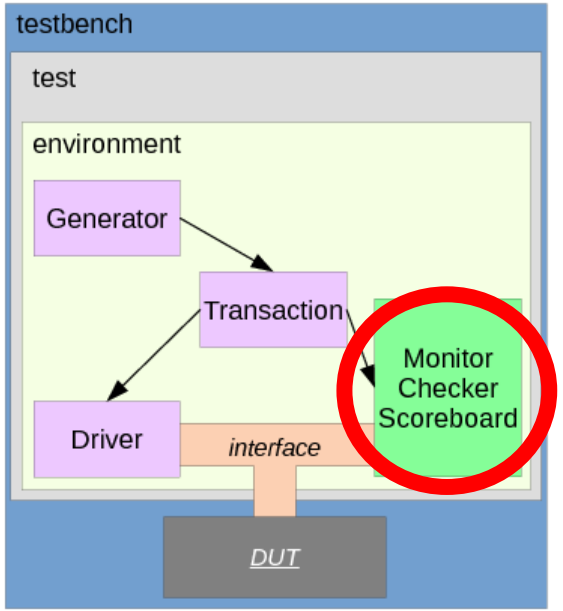


KU LEUVEN

micas KU LEUVEN

Monitor - Checker - Scoreboard

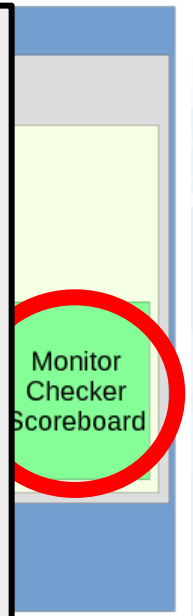
```
... // includes
class MonCheScb;
   virtual intf ifc;
   mailbox #(int signed) gen2mcs;

  function new(virtual intf i, mailbox #(int signed) g2m);
    this.ifc = i;
    gen2mcs = g2m;
  endfunction : new

  task run;
    $display("[MCS] run started");
    fork
      runMonitor();
      runCheckerScoreboard();
    join
    $display("[MCS] run stopped");
  endtask : run
```



Monitor - Checker – Scoreboard (2)

```
task runMonitor;
  $timeformat(-9, 2, " ns", 15);
  forever
  begin
    @(posedge ifc.clk iff ifc.cb.valid);
    begin
      result = ifc.c;
      $display("[MON] %t result captured %0d",
$time, result);
    end
  end
endtask : runMonitor
```

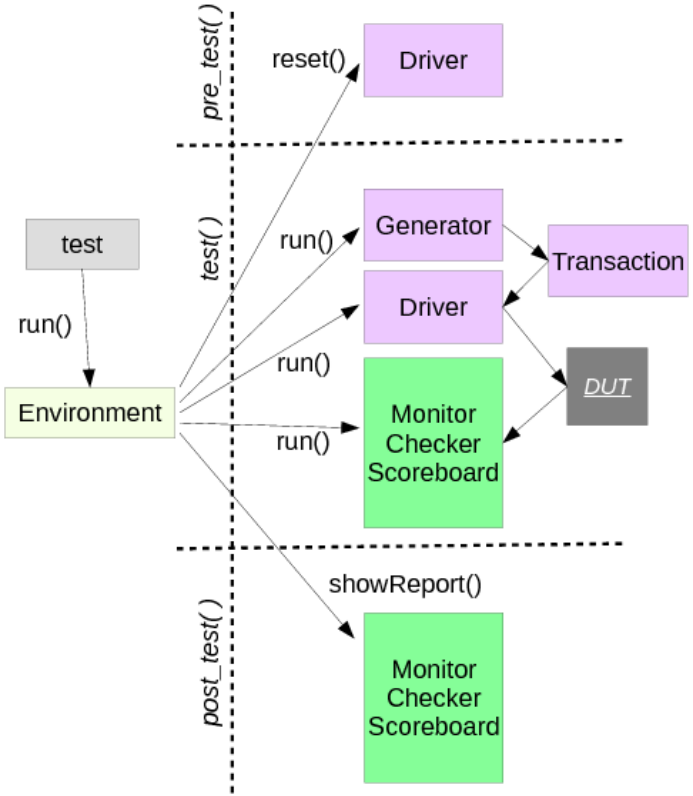
```
task runCheckerScoreboard;
  int op1, op2;
  forever
  begin
    gen2mcs.get(tract);
    op1 = tract.op1;
    op2 = tract.op2;
    tests++;
    if (result == (op1+op2) )
    begin
      tests_ok++;
      $display("[CHKSCB] %t result captured %0d (%0d, %0d)", $time,
result, op1, op2);
    end else begin
      tests_nok++;
      $display("[CHKSCB] %t result captured %0d (%0d, %0d)", $time,
result, op1, op2);
    end
   end
  end
endtask : runCheckerScoreboard
```



KU LEUVEN

micas KU LEUVEN

Example test environment - results (Transcript)




```
# [ENV] start-of-life
# [DRV] ----- Reset Started -----
# [DRV] ----- Reset Ended   -----
# [DRV] run started
# [MCS] run started
# [GEN] generated new operands (3, 1)
# [DRV] op1 set to 3
# [DRV] op2 set to 1
# [MON]       115.00 ns result captured 4
# [CHKSCB]       115.00 ns result captured 4 (3, 1)
# [GEN] generated new operands (1, 10)
# [DRV] op1 set to 1
# [DRV] op2 set to 10
# [MON]       125.00 ns result captured 11
# [CHKSCB]       125.00 ns result captured 11 (1, 10)
# [GEN] generated new operands (2, 14)
# [DRV] op1 set to 2
# [DRV] op2 set to 14
# [MON]       135.00 ns result captured 16
# [CHKSCB]       135.00 ns result captured 16 (2, 14)
# [GEN] generated new operands (6, 6)
# [DRV] op1 set to 6
# [DRV] op2 set to 6
# [MON]       145.00 ns result captured 12
# [CHKSCB]       145.00 ns result captured 12 (6, 6)
# [GEN] generated new operands (6, 13)
# [DRV] op1 set to 6
# [DRV] op2 set to 13
# [MON]       155.00 ns result captured 19
# [CHKSCB]       155.00 ns result captured 19 (6, 13)
# [SCB] # TESTS :     5
# [SCB] # TESTS OK :  5
# [SCB] # TESTS NOK : 0
# [ENV] end-of-life
```

Outline

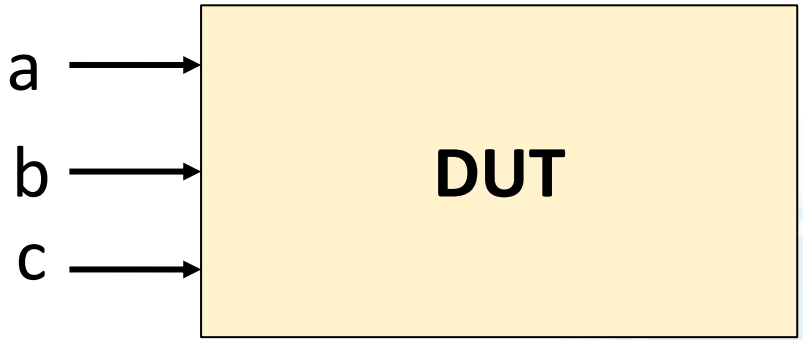
- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks ; process control
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - **Coverage and randomization**
  - Outlook

KU LEUVEN

KU LEUVEN micas

How do we know the DUT is fully verified?

Random stimulus



Quantify how much of DUT is covered in verification

Different types of coverage:

1. Code coverage: Are all code lines/states reached?
   - Extracted automatically by the tools
2. Functional coverage: Additional coverage defined by you

Code Coverage

- Types of code coverage: *line, cond, fsm, tgl, branch*
- Tool automatically generates html report

NAME	SCORE	LINE	COND	TOGGLE	FSM	BRANCH
DUT	86.86	100.00	88.57	92.86	60.00	92.86

KU LEUVEN

micas KU LEUVEN

Code Coverage

- Types of code coverage: *line*, *cond*, *fsm*, *tgl*, *branch*

Line Coverage for Module : dominated_average

	Line No.	Total	Covered	Percent
TOTAL		87	87	100.00
ALWAYS	36	3	3	100.00
ALWAYS	49	15	15	100.00
ALWAYS	71	69	69	100.00

```
35                      always_ff @(posedge clk or negedge arst_n_in) begin
36          1/1           if(~arst_n_in) begin
37          1/1             state <= READY;
38                        end else begin
39          1/1             state <= next_state;
40                        end
41                      end
42
43
44
45                      int signed stored_a, stored_b, to_stored_a, to_stored_b;
46                      int signed stored_msc_1, stored_msc_2, stored_msc_3, to_stored_msc_1, to_stored_msc_2, to_stored_msc_3;
47                      logic to_a_dominating, to_b_dominating, a_dominating, b_dominating;
48                      always_ff @(posedge clk or negedge arst_n_in) begin
49          1/1                   if(~arst_n_in) begin
```


KU LEUVEN

KU LEUVEN micas

Code Coverage

- Types of code coverage: *line*, *cond*, *fsm*, *tgl*, *branch*

```
LINE       162
SUB-EXPRESSION ((b == 0) ? stored_msc_1 : (a_dominating ? stored_a : (b_dominating ? stored_b : ((from_adder >>> 1)))))
                ----1---
```

-1-	Status
0	Covered
1	Not Covered

```
LINE       162
SUB-EXPRESSION (a_dominating ? stored_a : (b_dominating ? stored_b : ((from_adder >>> 1))))
                ------1-----
```

-1-	Status
0	Covered
1	Covered

```
LINE       162
SUB-EXPRESSION (b_dominating ? stored_b : ((from_adder >>> 1)))
                ------1-----
```

-1-	Status

KU LEUVEN

Code Coverage

- Types of code coverage: *line*, *cond*, *fsm*, *tgl*, *branch*

FSM Coverage for Module : dominated_average

Summary for FSM :: state

	Total	Covered	Percent	
States	6	6	100.00	(Not included in score)
Transitions	10	6	60.00	
Sequences	0	0		

State, Transition and Sequence Details for FSM :: state

states	Line No.	Covered
READY	37	Covered
ABS_B	103	Covered
DIFF	117	Covered
ABS_DIFF	131	Covered
DOMINATION_DETERMINATION	144	Covered
FINAL	156	Covered

transitions	Line No.	Covered
READY->ABS_B	103	Covered
ABS_B->READY	37	Not Covered
ABS_B->DIFF	117	Covered
DIFF->READY	37	Not Covered
DIFF->ABS_DIFF	131	Covered
ABS_DIFF->READY	37	Not Covered
ABS_DIFF->DOMINATION_DETERMINATION	144	Covered
DOMINATION_DETERMINATION->READY	37	Not Covered
DOMINATION_DETERMINATION->FINAL	156	Covered
FINAL->READY	37	Covered

KU LEUVEN

KU LEUVEN micas

Code Coverage

- Types of code coverage: *line, cond, fsm, tgl, branch*

Toggle Coverage for Module : dominated_average

	Total	Covered	Percent
Totals	9	8	88.89
Total Bits	18	17	94.44
Total Bits 0->1	9	9	100.00
Total Bits 1->0	9	8	88.89
Ports	5	4	80.00
Port Bits	10	9	90.00
Port Bits 0->1	5	5	100.00
Port Bits 1->0	5	4	80.00
Signals	4	4	100.00
Signal Bits	8	8	100.00
Signal Bits 0->1	4	4	100.00
Signal Bits 1->0	4	4	100.00

Port Details

Name	Toggle	Toggle 1->0	Toggle 0->1	Direction
clk	Yes	Yes	Yes	INPUT
arst_n_in	No	No	Yes	INPUT
start	Yes	Yes	Yes	INPUT
to_adder_add_sub	Yes	Yes	Yes	OUTPUT
done	Yes	Yes	Yes	OUTPUT

KU LEUVEN

micas KU LEUVEN

Code Coverage

- Types of code coverage: *line*, *cond*, *fsm*, *tgl*, *branch*

```
153             to_stored_msc_1 <= a <<< 1;//only needed when b==0
154             to_stored_msc_2 <= a <<< 1;//only needed when a==0. BUG: should be b<<<1 (line always hit, probability of effect on output=1/2^32
155
156             next_state <= FINAL;
157           end else if (state == FINAL) begin
                      -10-
158             //(a+b)/2
159             to_adder_a <= stored_a;
160             to_adder_b <= stored_b;
161             to_adder_add_sub <= 1; //add
162             out <= (a==0) ? stored_msc_2 : ((b==0) ? stored_msc_1 : (a_dominating ? stored_a : (b_dominating ? stored_b : (from_adder>>>1))))
                            -11-                  -12-                    -13-                      -14-
                            ==>                   ==>                     ==>                       ==>
                                                                                                    ==>
163             next_state <= READY;
164           end else begin
165             assert(0);
                ==>
```

Branches:

-1-	-2-	-3-	-4-	-5-	-6-	-7-	-8-	-9-	-10-	-11-	-12-	-13-	-14-	Status
1	1	-	-	-	-	-	-	-	-	-	-	-	-	Covered
1	0	1	-	-	-	-	-	-	-	-	-	-	-	Covered
1	0	0	-	-	-	-	-	-	-	-	-	-	-	Covered
0	-	-	1	1	-	-	-	-	-	-	-	-	-	Covered
0	-	-	1	0	-	-	-	-	-	-	-	-	-	Covered
0	-	-	0	-	1	-	-	-	-	-	-	-	-	Covered
0	-	-	0	-	0	1	1	-	-	-	-	-	-	Covered
0	-	-	0	-	0	1	0	-	-	-	-	-	-	Covered
0	-	-	0	-	0	0	-	1	-	-	-	-	-	Covered
0	-	-	0	-	0	0	-	0	1	1	-	-	-	Not Covered
0	-	-	0	-	0	0	-	0	1	0	1	-	-	Not Covered
0	-	-	0	-	0	0	-	0	1	0	0	1	-	Covered
0	-	-	0	-	0	0	-	0	1	0	0	0	1	Covered
0	-	-	0	-	0	0	-	0	1	0	0	0	0	Covered
0	-	-	0	-	0	0	-	0	0	-	-	-	-	Not Covered

KU LEUVEN

nicas KU LEUVEN

Is DUT fully verified after 100% Code coverage?

- Limitation Of Code Coverage
  - Coverage does not know anything about functionality of design
  - There is no way to find what is missing in the code

Verification is not completed after 100% code coverage

Functional coverage

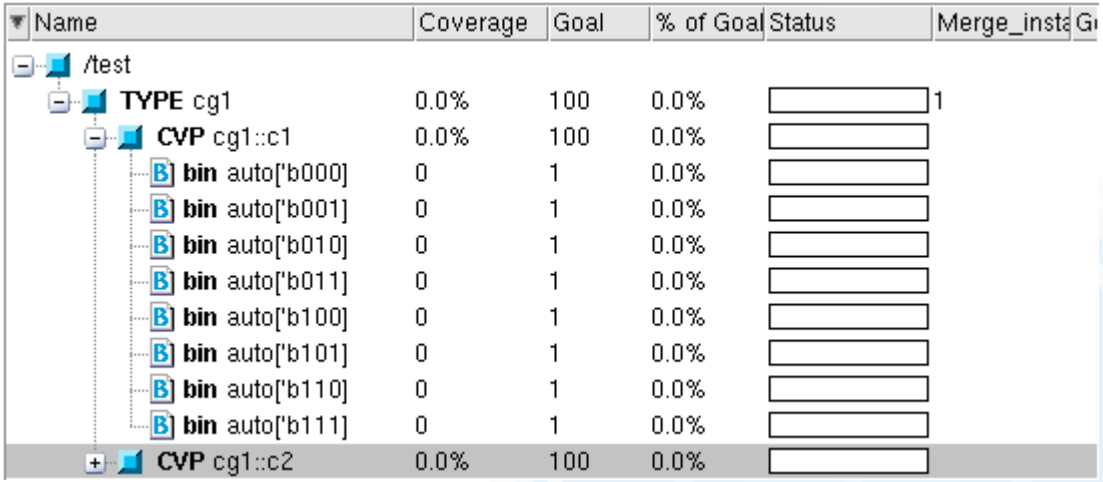
- YOU determine what additional coverage gets checked

- Examples:
  - Did I see a case where 0<A<5?
  - Did the reset happen while input “A == 0”?
  - Did I simulate a sequence of inputs A = 3 -> 4 -> 8

Possible through user defined cover points and covergroups!

Functional coverage: covergroup and coverpoint

```
module test;
    bit clk;
    logic [2:0] addr, data;
     … …
    covergroup cg1@(posedge clk);
        c1: coverpoint addr;
        c2: coverpoint data;
    endgroup
    … …
    cg1 cg_inst = new();
endmodule
```



Name	Coverage	Goal	% of Goal	Status	Merge_inst	G
/test						
TYPE cg1	0.0%	100	0.0%		1	
CVP cg1::c1	0.0%	100	0.0%			
bin auto['b000]	0	1	0.0%			
bin auto['b001]	0	1	0.0%			
bin auto['b010]	0	1	0.0%			
bin auto['b011]	0	1	0.0%			
bin auto['b100]	0	1	0.0%			
bin auto['b101]	0	1	0.0%			
bin auto['b110]	0	1	0.0%			
bin auto['b111]	0	1	0.0%			
CVP cg1::c2	0.0%	100	0.0%			

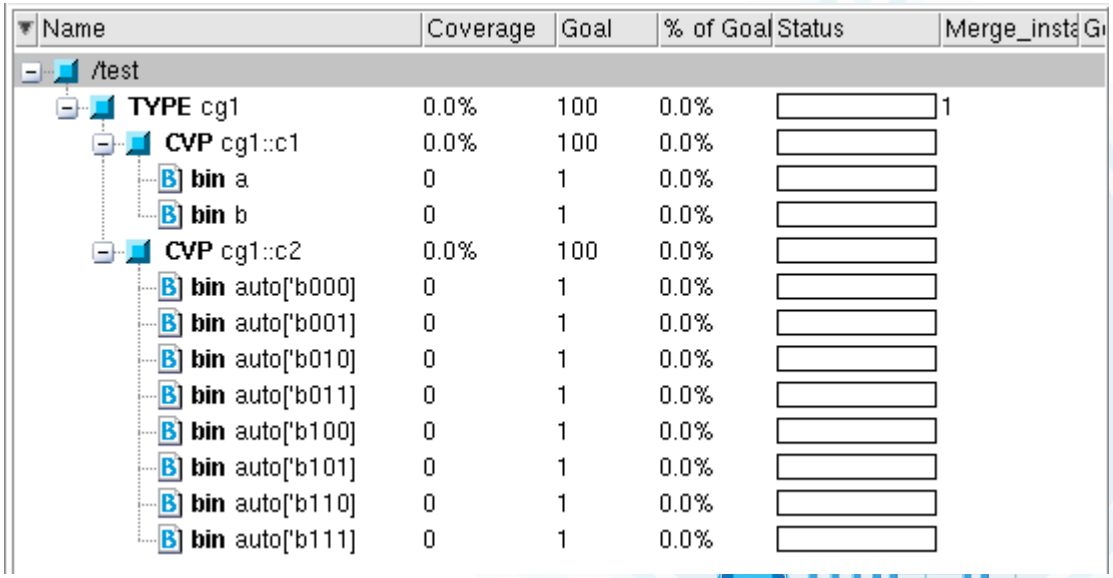
KU LEUVEN

micas KU LEUVEN

Functional coverage: bins

Bins let user define regions of interest

```
module test;
   bit clk;
   logic [2:0] addr, data;

   covergroup cg1@(posedge clk);
      c1: coverpoint addr{
         bins a = {1,2};
         bins b = {3,4,5,6,7};}
      c2: coverpoint data;
   endgroup

   cg1 cg_inst = new ;
endmodule
```

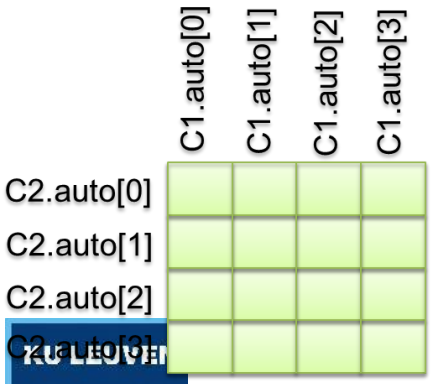


Name	Coverage	Goal	% of Goal	Status	Merge_inst	G
/test						
TYPE cg1	0.0%	100	0.0%		1	
CVP cg1::c1	0.0%	100	0.0%			
bin a	0	1	0.0%			
bin b	0	1	0.0%			
CVP cg1::c2	0.0%	100	0.0%			
bin auto['b000]	0	1	0.0%			
bin auto['b001]	0	1	0.0%			
bin auto['b010]	0	1	0.0%			
bin auto['b011]	0	1	0.0%			
bin auto['b100]	0	1	0.0%			
bin auto['b101]	0	1	0.0%			
bin auto['b110]	0	1	0.0%			
bin auto['b111]	0	1	0.0%			

Functional coverage: cross

Cross: for co-occurrence of 2 variables

```
bit clk; logic [1:0] addr, data; int i =1;

covergroup cg1@(posedge clk);
    c1xc2 : cross addr, data;
endgroup
cg1 cg_inst = new;
```



/test1				
TYPE cg1	81.2%	100	81.2%	1
CVP cg1::addr	100.0%	100	100.0%	
CVP cg1::data	100.0%	100	100.0%	
CROSS cg1::c1xc2	43.7%	100	43.7%	
bin <auto['b00],auto['b00]>	0	1	0.0%	
bin <auto['b01],auto['b00]>	1	1	100.0%	
bin <auto['b10],auto['b00]>	0	1	0.0%	
bin <auto['b11],auto['b00]>	1	1	100.0%	
bin <auto['b00],auto['b01]>	0	1	0.0%	
bin <auto['b01],auto['b01]>	1	1	100.0%	
bin <auto['b10],auto['b01]>	0	1	0.0%	
bin <auto['b11],auto['b01]>	8	1	800.0%	
bin <auto['b00],auto['b10]>	2	1	200.0%	
bin <auto['b01],auto['b10]>	1	1	100.0%	
bin <auto['b10],auto['b10]>	0	1	0.0%	
bin <auto['b11],auto['b10]>	0	1	0.0%	
bin <auto['b00],auto['b11]>	0	1	0.0%	
bin <auto['b01],auto['b11]>	0	1	0.0%	
bin <auto['b10],auto['b11]>	2	1	200.0%	
bin <auto['b11],auto['b11]>	0	1	0.0%	

What is my coverage is not good?

- Simulate longer
- Better constraints for randomization

KU LEUVEN

KU LEUVEN micas

(Constrained) randomization

- You can control this "randomness"

```
`ifndef SV_TRANSACTION
`define SV_TRANSACTION

class Transaction;
 rand int op1, op2;
  constraint c1 {op1 < op2;}
  constraint c2 {op1 > 0;}
  constraint c3 {op2 dist {0:=5 , 1:=3 , 3:=2} ;}
                    % 50% chance 0; 30% 2 and 20%3
 function new();
   op1=0;
   op2=0;
 endfunction : new

endclass : Transaction
`endif
```

```
`include "Transaction.sv"

class Generator;
 mailbox #(Transaction) gen2drv;
 mailbox #(Transaction) gen2mcs;
 Transaction tract;

 function new(#(Transaction) g2d, mailbox #(Transaction) g2m);
      gen2drv = g2d;
      gen2mcs = g2m;
 endfunction : new

 task run();
  #10;            // wait some time so the Driver is listening
  for(int i=0;i<NO_tests;i++)
  begin
    void'(tract.randomize());
    $display("[GEN] generated new operands (%0d, %0d)", tract.op1, tract.op2);

    gen2drv.put(tract);
    gen2mcs.put(tract);

 endtask : run
endclass : Generator
```


KU LEUVEN

Outline

- Layered testbench structure
- System Verilog to the rescue
  - System Verilog constructs
    - Example testbench
  - Programs, functions and tasks ; process co
  - Interfaces, modports and clocking blocks
    - Example testbench
  - Classes & mailboxes
    - Example testbench
  - Coverage and randomization
  - **Outlook**

Needs practice…
➔ Exercise sessions

KU LEUVEN

micas KU LEUVEN

Real industrial verification…

- Exploits assertions
  - Automated checks embedded inside the Verilog code of the DUT
- Uses UVM: Universal Verification Methodology
  - Much more strict / streamlined way of doing verification
  - Large library of pre-made verification blocks, flows, methodology
  - Much more complex…

System Verilog: Design & verification!

[Stuart Sutherland 2013]

Synthesizable!

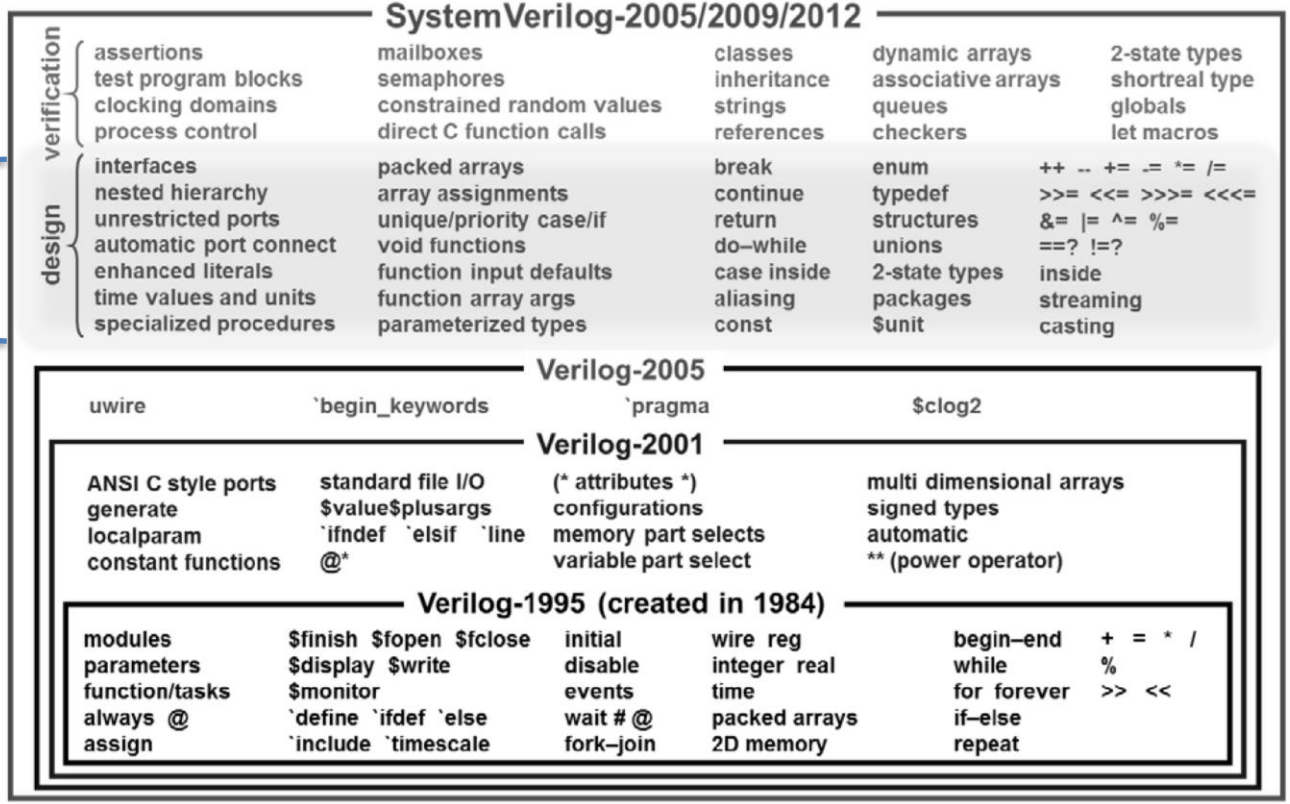


Figure 1. Verilog to SystemVerilog growth chart

KU LEUVEN

KU LEUVEN

KU LEUVEN micas