Chapter 6
Architecting Testbench

教育部顧問室
「超大型積體電路與系統設計」教育改進計畫
EDA聯盟編製

Purpose

- Focuses on the structure of the testbench
- Show good stimulus generators and response monitors to minimize maintenance, facilitate implementing a large number of testbenches, and promote the reusability of verification component.

Outline

- *Reusable verification components*
- Verilog Implementation
- Autonomous Generation and Monitoring
- Input and Output Paths
- Verifying Configurable Designs
- Summary

Reusable verification components

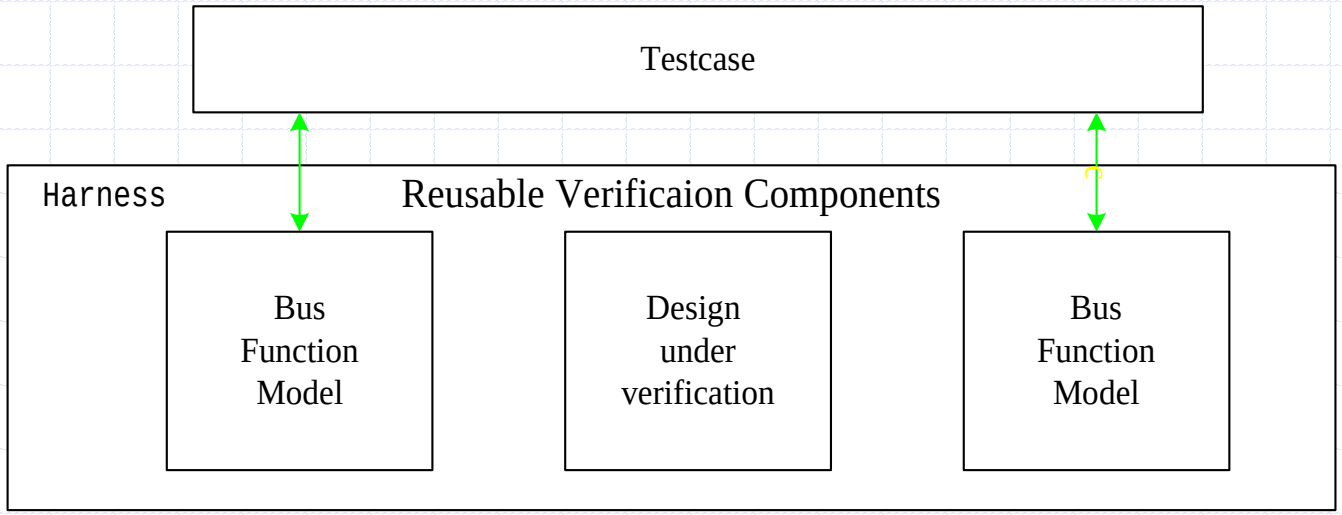
- Goal:
  - Maximize the amount of verification code reused across testbenches.
  - Minimize the development efforts.

Structure of Testbech

- Two major components of a testbench:
  - Resuable test harness
  - Testcase-specific code

What is Test harness

- Low-level layer common to all testbenches for the design under verification.

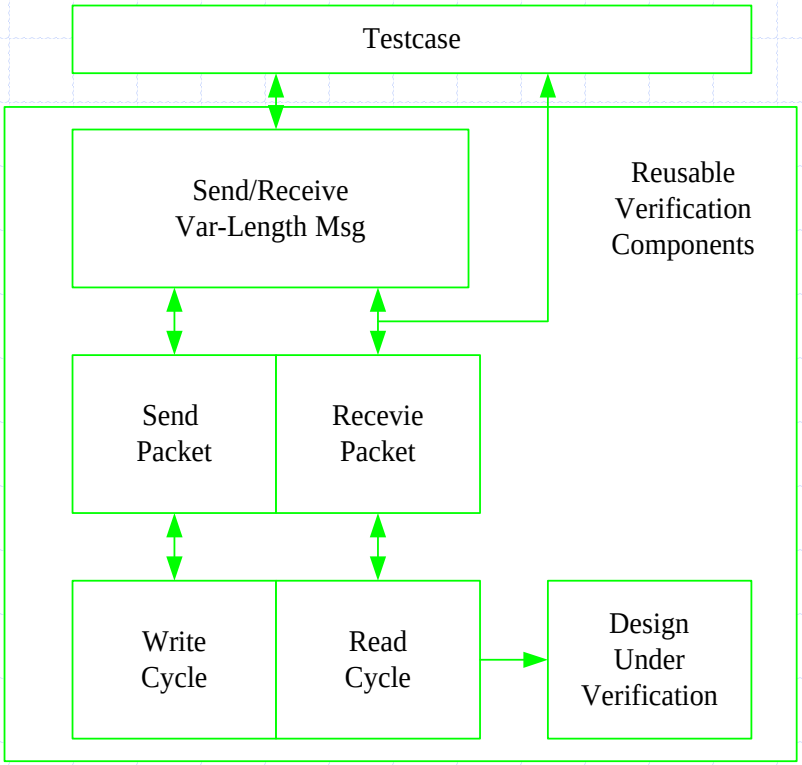


Reusable utility routine

- Many testbenches share some common functionality.
- Once the low-level features are verified, the repetitive nature of communicating with the device under verifiction can be abstracted into high level utility routine

Example

- Structure of a testbench with reusable utility routines



Procedural Interface

- To reusable by many testcases, we must define a procedural interface independent of their detail implementaion.
- All components is accessed through procedures or tasks.
- Never through global varibles or singals.

Flexibility through layers

- Verification components must be flexible to provide functionality for all testbenches.
  - Layering utility routine on top of general purpose lower-level routines.
  - The low-level layer provides detail control
  - The high-level layer provides greater abstraction

Flexibility through layers

- Don’t implement all functionality in single level.
  - Complicate the implementation of the bus-functional models.
  - Increasing the risk of introducing a functional failure

Procedural Interface

- Procedural interfaces remove the testcase from knowing the low-level details of the physical interfaces on the design.
- Well-designed procedural interface:
  - The physical interface of design can be modified without having to modify any testbench

Example

- A processor interface is changed from a VME bus to a X86 bus.
  - All that needs to be modified is the implementation of CPU bus-functional model
- A data transmission protocol from parallel to serial

Development Process

- Don’t write the ultimate verification component that includes every configuration option.
- Use the verification plan to determine the required functionality.

Development Process

- Start with the basic functions required by basic testbenches.
- Add configurability to the bus-functional models or creating uility routines.
- The procedual interface are maintained to avoid breaking testbeches.

Development Process

- The incremental approach minimizes development effort:
  - Won't develop nouse functionality
  - Minimize your debugging effort
  - Allows the development of the verification infrastructure to parallel the development of the testbenches.

Outline

- Reusable verification components
- *Verilog Implementation*
- Autonomous Generation and Monitoring
- Input and Output Paths
- Verifying Configurable Designs
- Summary

Verilog Implementation

- Starting with a monolithic testbench
- Goal:
  - Refine it into layers of bus-functional models, utility packages ,and testcases, with well-defined procedural interface.
  - Obtain a flexible implemention strategy.

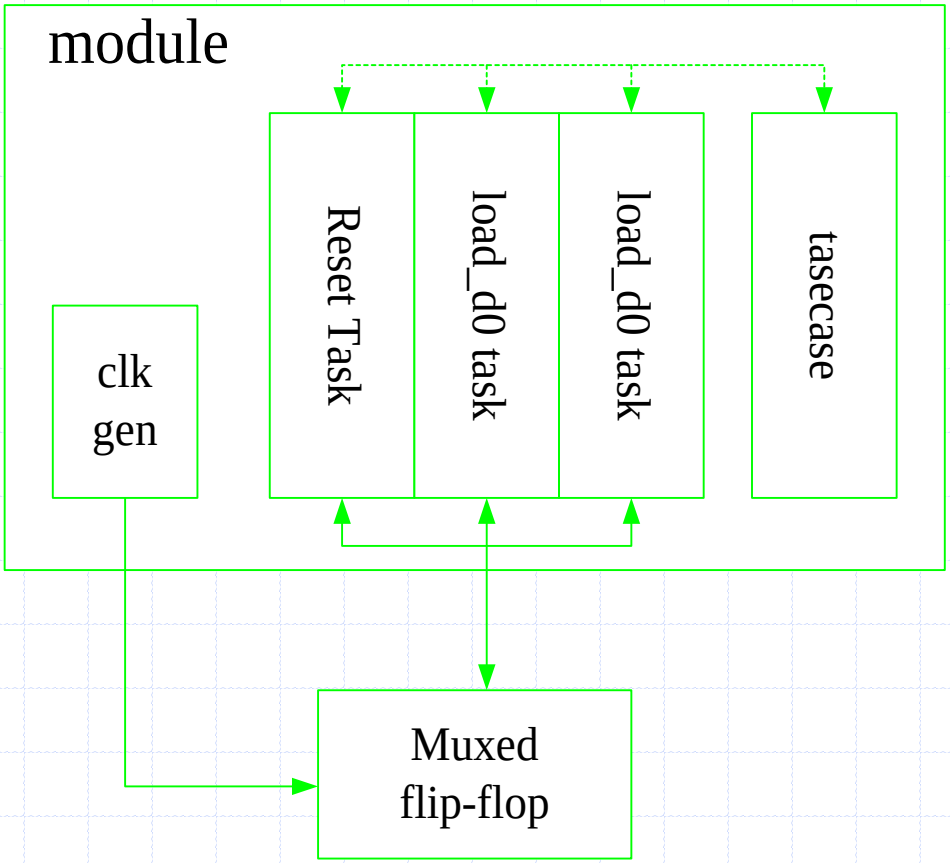
Verilog Implementation

- Leave the same portions of all testbenches in the level of hierarchy immediately surrounding the design under verification.
- Move the control structure unique to each testcase into a higher level of hierarchy

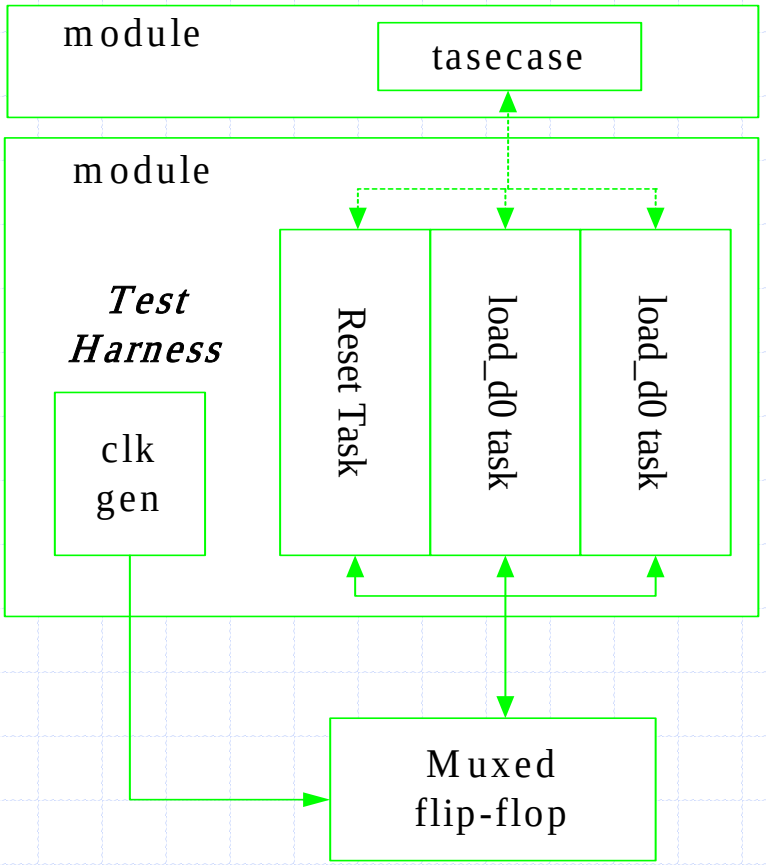
Verilog Implementation

- Test harness:
  - The level of hierarchy containing the reusable verification components.
  - Self-contained and provide all signals to operate the design under verificaion.
  - Also contain the clock and reset generator.

Non-Reusable Structure



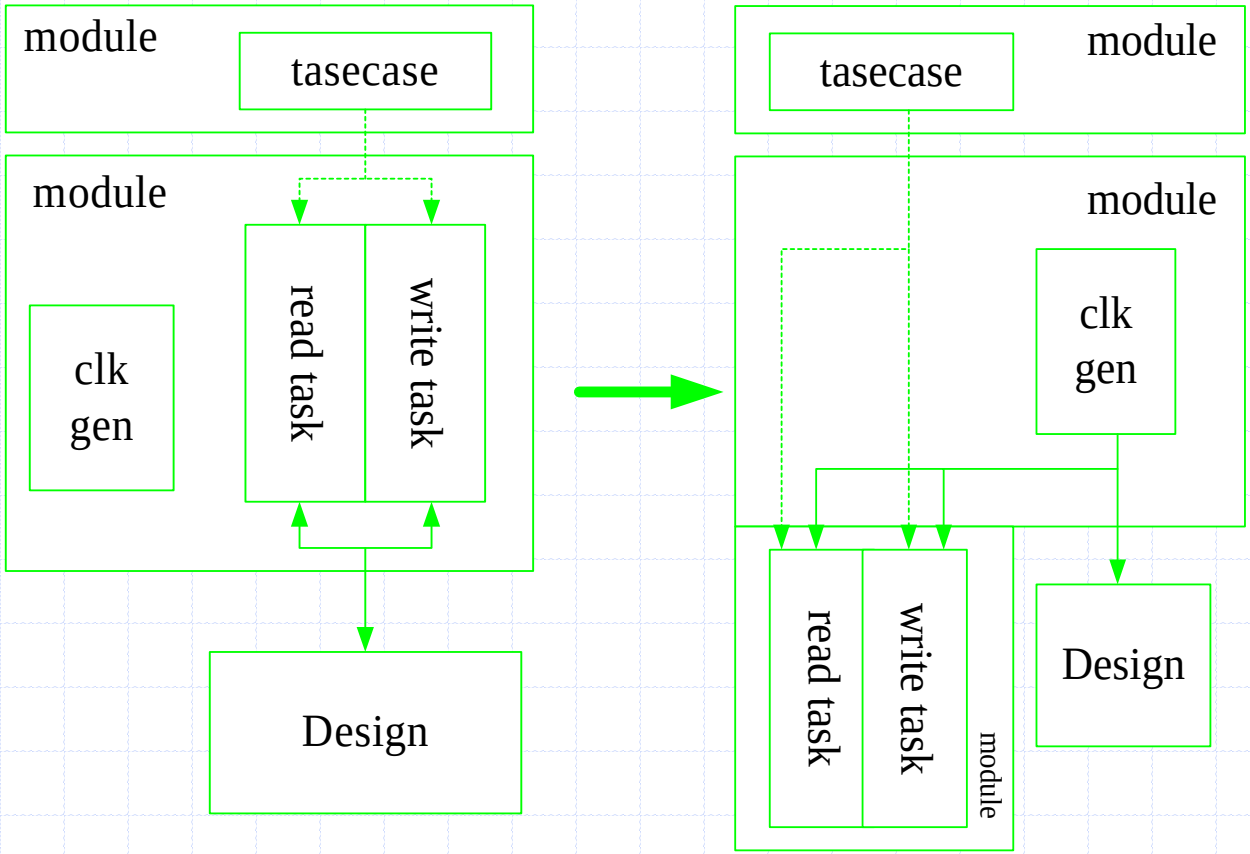
Reuseable Structure



Packaging Bus-Functional models

- Help the reusability of bus-functional models between test harness for different designs
- Tasks providing a complete bus-functional model should be packaged .
  - Make it easy to reuse between harness

Packaging Bus-Functional models



Utility Packages

- Utility routines
  - Provide addition levels of abstraction to testcase also composed of a series of tasks amd functions
  - Can be encapsulated in separate modules
  - Using hierarchical names to acess low-level procedural interfaces.

Utility Packages

- Utility packages are never instantiated
  - It runs in parallel with testbench and design
  - Access the tasks and functions in the test harness using absolute hierarchical names

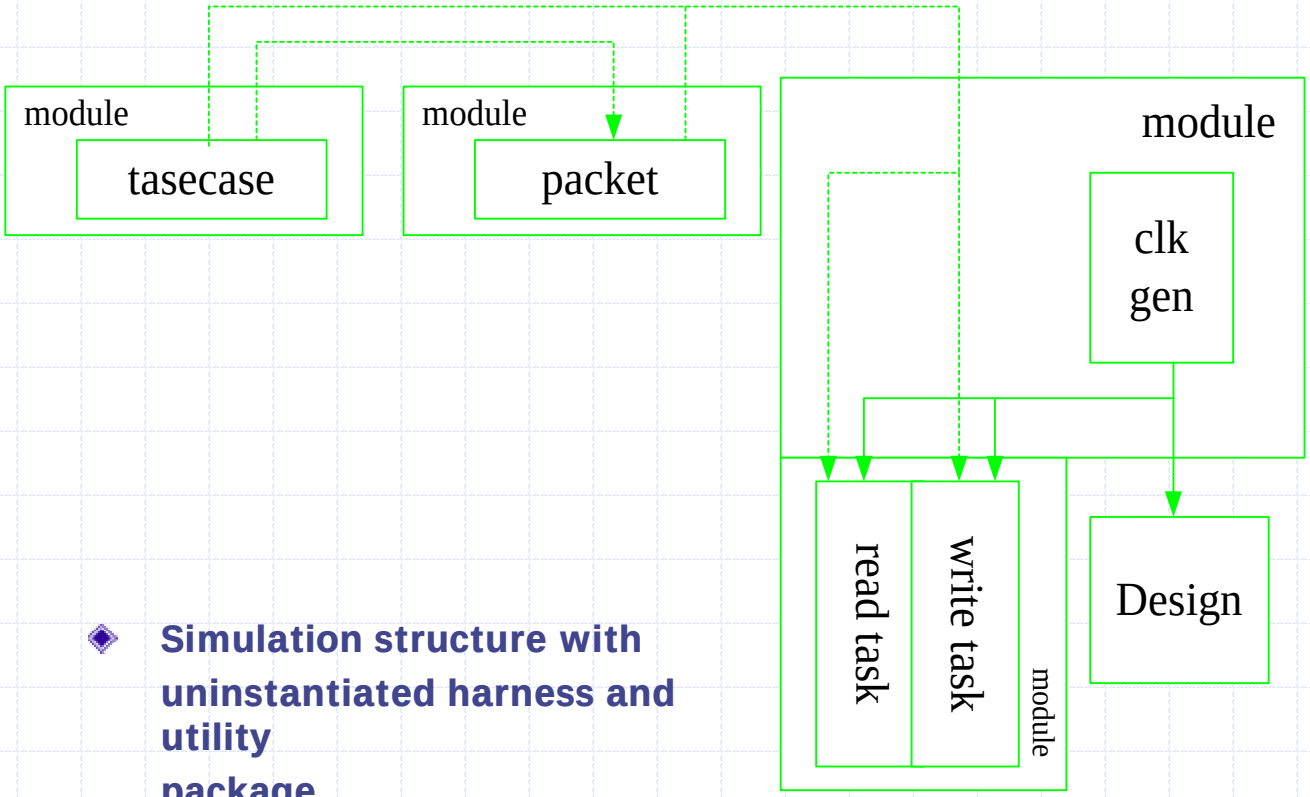
Example

```
module packet;
task send;
    input  [64*8:1] pkt;
    reg    [   15:0] word;
    interger   i;
begin
    for (i=0;i < 32; i = i + 1) begin
            word = pkt[16:1];
            testcase.th.cpu.write(24'h10_0000+i, word);
            pkt = pkt >> 16;
    end
end
endmodule
```


Utility Packages

- Leave the test harness uninstantiated, forming its own simulation top-level module.
- Use absolute hierarchical name to access tasks and functions in the test harness.

Utility Packages

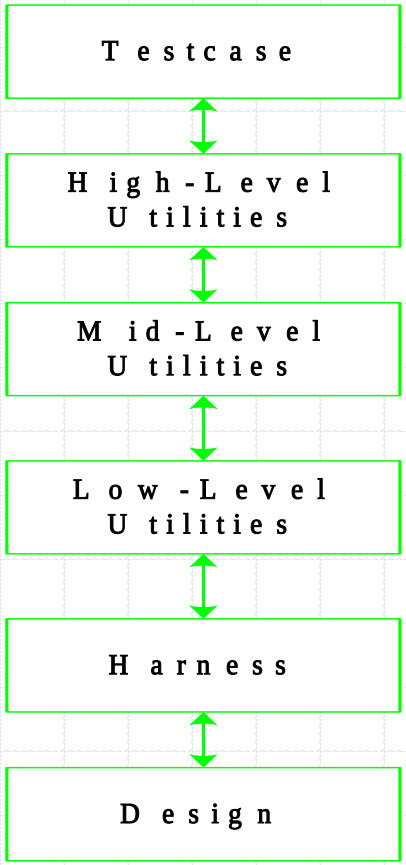


- **Simulation structure with uninstantiated harness and utility package**

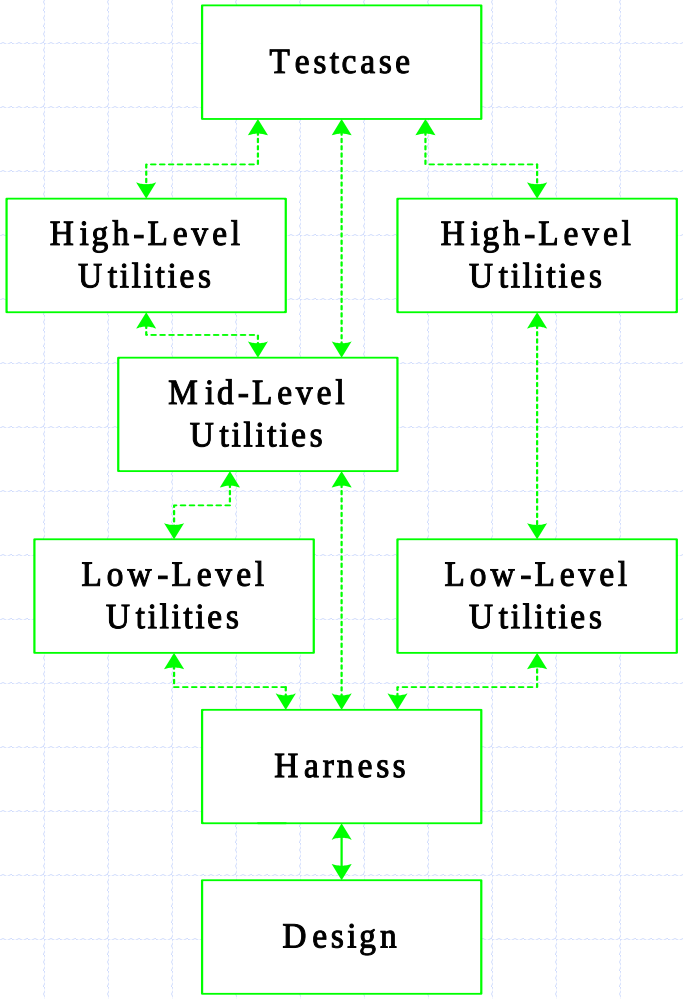
Verilog Implementation

- Verilog simulation with multiple top-level module use Verilog-XL or VCS:
  - %verilog testcase.v packet.v harness.v \
    i386sx.v design.v

Single top-level



Uninstantiated modules



Outline

- Reusable verification components
- Verilog Implementation
- ***Autonomous Generation and Monitoring***
- Input and Output Paths
- Verifying Configurable Designs
- Summary

Autonomous Stimulus

- The packaged bus-functional models can contain processes and always or initial blocks.
- Can perform a vareity of tasks such as safety checks, data gerenation, or collecting response for later retrieval.

Autonomous Stimulus

- Protocols may require more data than is relevant for the testcase.
- The access procedures would interface with a transmission process.

Random Stimulus

- The content of generated data can be random
- The packaged bus model can contain an algorithms to generate random data, in a random sequence, at random intervals.

Example

- **Bus-Functional model for a i386SX gerenating random cycles**

```
Module i386sx(…);
Task read;
      …
End task
Task write
      …
End task
Always
Begin: random_generator
      reg [23:0] addr;
      reg [15:0] data;
      // random interval (0-255)
      #($random >> 24);
      //random even address
      addr [23:21] = 3'b000;
      addr [20:  1] = $random;
      addr [       0] = 1'b0;
      //random read or write
      if( $random % 2) begin
          write (addr, $random);
      end else begin
          read (addr, data);
      end
End
Endmodule
```


Random Stimulus

- Autonomous generators can help compute the expected response.
- The CRC is computed based on the randomly generated payload and destination description.

Example

- Random generator helping to verify output

```
Always
Begin: monitor
      reg`packet_typ pkt;
      // Generate the header
      pkt`src_addr = my_id;
      pkt`dst_addr = $random;
      // Which port does this packet goes to?
      pkt`out_port_id = re_table[ pkt `strm_id];
      //Next in a random stream
      pkt`strm_id = {$random, my_id};
      pkt`seq_num = seq_num[pkt`strm_id];
      //Fill the payload
      pkt`filler = $random;
      pkt`crc = computer_crc(pkt);
      //send the packet
      send_pkt(pkt);
      seq_num[pkt`strm_id] = seq_num[pkt`strm_id] + 1;
End
```

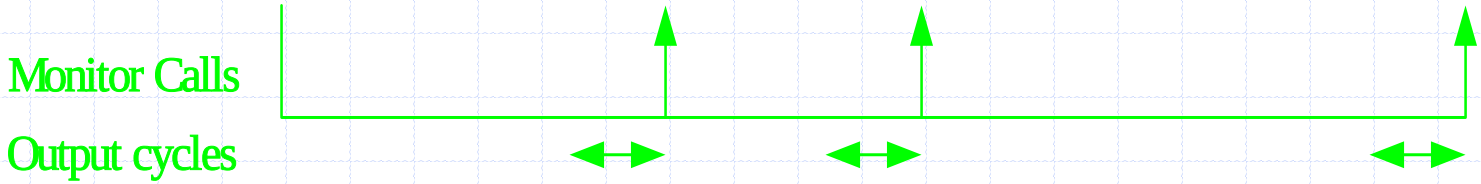

Injecting Errors

- Generators can be configured to inject errors
- An example:

```
Always
Begin: monitor
    pkt`crc = computer_crc(pkt);
    …
    //randomly corrupt the CRC for 1 % of cell
    if($random %100 == 0) begin
      pkt`seq_num = $random;
      pkt`crc = pkt`crc ^ (1 << ($random %8));
    end else begin
      seq_num[pkt`strm_id] = seq_num[pkt`strm_id] + 1;
    end
    //send the packet
    send_pkt(pkt);
end
```


Autonomous Monitoring

- The monitor is used when the design controls the timing.
- Monitor must always be listening.
- Proper call timing for monitoring procedure



Outline

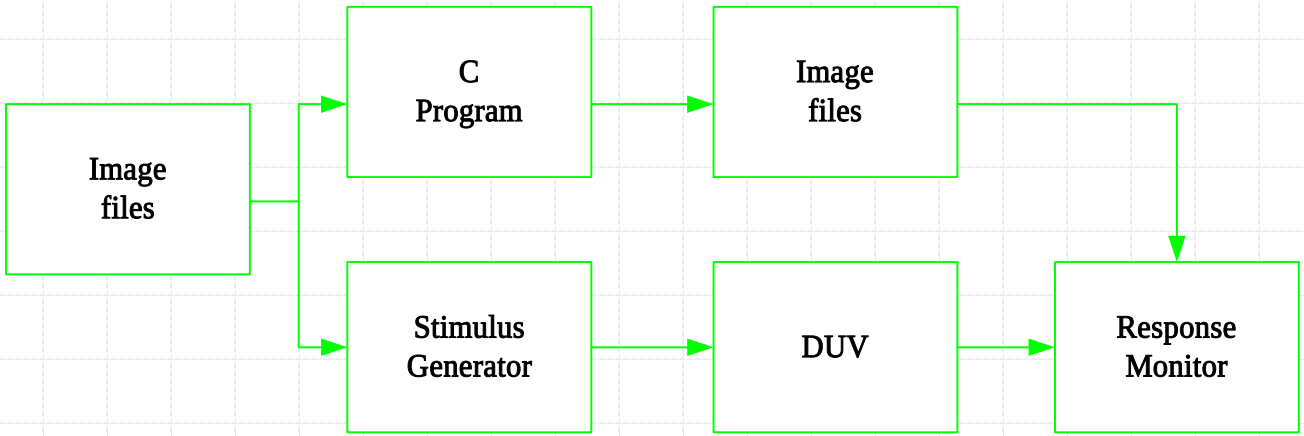
- Reusable verification components
- Verilog Implementation
- Autonomous Generation and Monitoring
- *Input and Output Paths*
- Verifying Configurable Designs
- Summary

Purpose

- Describe how test data can be obtained from external files.
- Configure reusable verification components properly.
- Ensure the simulation results are not clobbered by using unique output file names

Programmable Testbenches

- Testbenches can be programmed through external file.
- Using external input files for stimulus and response



Configuration Files

- Configuration should be controlled by the testcase.
- Avoid using external configuration files.
  - Improper configuation using an exteranl file

```
Initial
Begin: init_seed
    Integer seed[0:0];
    $readmemh("seed.in", seed);
    $random(seed[0]);
end
```


Configuration Files

- Make filenames configurable
  - Don’t hardcode the name of file providing the information in the bus-funcitonal model
  - Use-specified filename in verilog:

```
Task data_from_file;
  input [8*32:1] name;
  $readmemh(name, mem);
endtask
```


Concurrent Simulations

- Make sure filenames are unique
  - A hardcoded filename creates collisions between two simulations which run concurrently
  - Generateing unique filenames in verilog

```
parameter testcase = "…";
initial
begin
   $dumpfile({testcase, ".dump"});
   $dumpvars;
end
```


Compile-Time Configuration

- Most verilog simulators always recompiling the source code before each simulation.
- Avoid using compile-time configuration of bus-functional models.
  - Minimize the compilation requirement
  - Avoid making managing a testcase more compilcated as an additional file.
  - Can run concurrent compiled simulations.

Outline:

- Reusable verification components
- Verilog Implementation
- Autonomous Generation and Monitoring
- Input and Output Paths
- *Verifying Configurable Designs*
- Summary

Two kinds of design configurability

- Soft configurability:
  - Be performed through a programable interface and can be changed during the operation of the design
- Example:
  - The offsets for almost-full or almost-empty flags on a FIFO.
  - The baud rate of a UART

Two kinds of design configurability

- Hard configuration:
  - It's fundamental to the functional nature of the design.
  - it can't be modified during normal operations.
- Example:
  - A PCI interface operates at 33 or 66 MHz
  - The width and depth of a FIFO

Configurable Testbenches

- Configure the testbench to match the design
  - The configuration of the testbench must be consist with the configuration of the design
  - Using a configuration techique similar to the one used by the design
  - Using generics or parameters to configure testbench.

Configurable Testbenches

- Configurable memory model:
  - module memory(data, addr, rw, cs);
    parameter DWIDTH = 1,AWIDTH = 1;
    inout [DWIDTH-1:0] data;
    input [AWIDTH-1:0] addr;
    …
    endmodule
- Using a configurable model in verilog:
  - module system(…)
    parameter ASIZE = 10, _DSIZE = 16;
    wire [DSIZE-1:0] data;
    reg [ASIZE-1:0] addr;
    reg rw,cs0,cs1;
    memory m0(data, addr, rw,cs0);
    defparam m0.AWIDTH = ASIZE, m0.DWIDTH = DSIZE;
    endmodule

Top Level Generics and Parameters

- Top-level modules and entities can have generics or parameter.
- Top-level generics or parameters need to be defined
  - Some simulation tools allow seting the generics and parameter via the commandline.
  - Use a defparam module in verilog to set parameter.

Outline

- Reusable verification components
- Verilog Implementation
- Autonomous Generation and Monitoring
- Input and Output Paths
- Verifying Configurable Designs
- *Summary*

Summary

- This chapter focused on the implementation of testbench for a device under verification
- The portion of the testbeches that is common between all testcase is structured into a test harness.
- The configuration of bus-functional models by each testcases should be limited to using the available procedural interfaces.