Verification Testbench

Nagesh Loke
ARM CPU Verification Lead/Manager

What to expect?

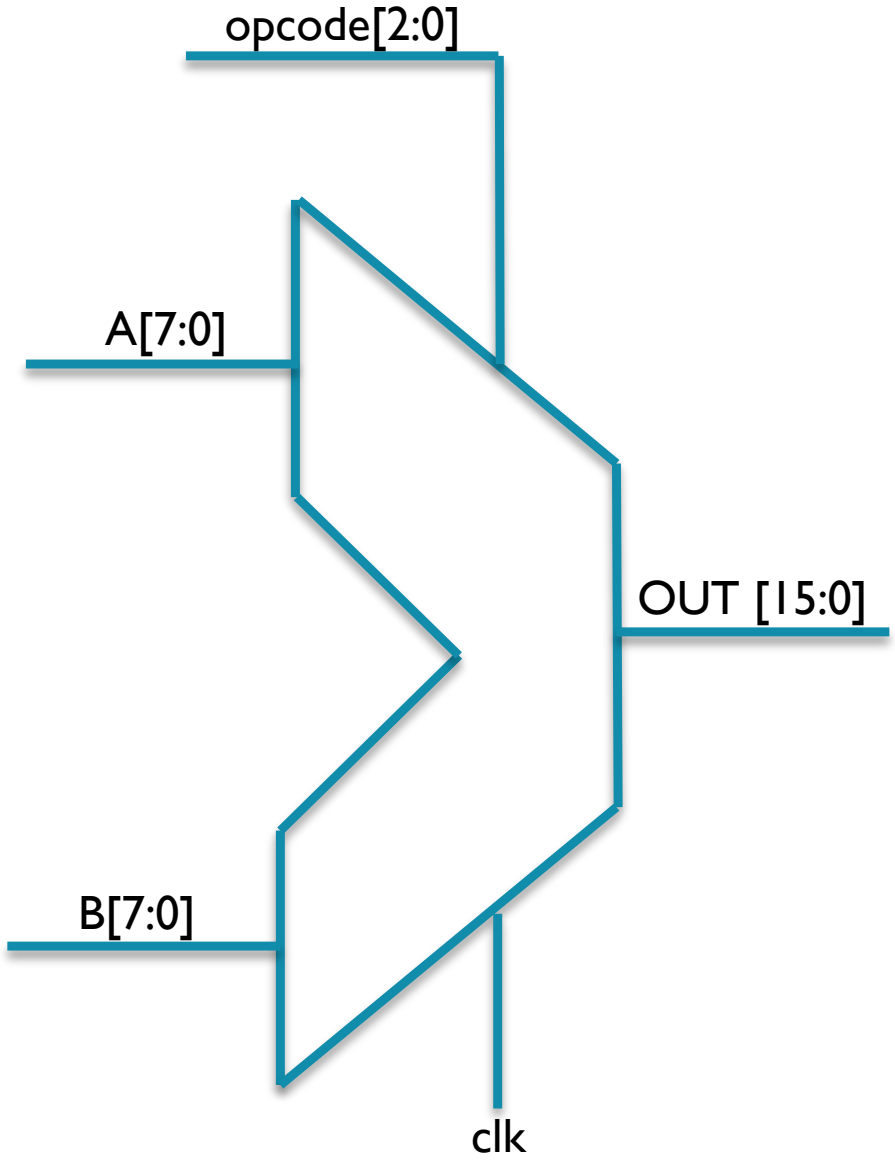
- This lecture aims to:
  - provide an idea of what a testbench is
  - help develop an understanding of the various components of a testbench
  - build an appreciation of the complexity in a testbench
  - highlight why it is as much a software problem as is a hardware problem

What is a testbench?

- A testbench helps build an environment to test and verify a design
- The key components of a testbench are:
- Stimulus
  - is used to drive inputs of the design to generate a high-level of confidence
  - should be able to exercise all normal input scenarios and a good portion of critical combinations with ease
- Checker
  - is a parallel & independent implementation of the specification
  - is used verify the design output against the modeled output
- Coverage
  - Helps measure quality of stimulus
  - Provides a measure of confidence to help determine closure of verification effort

What are we verifying?

- An ALU has
  - an input clock
  - two 8-bit inputs as operands
  - a 3-bit opcode as an operator
  - a 16-bit output
- Performs the following operations:
  - ADD, SUB, AND, OR, XOR, MUL, XNOR



ARM

How was it done in the past?

```
initial begin

    @(negedge clk);

    opcode = 3'b000;  // ADD

    A = 8'h05;
    B = 8'h50;
    @(negedge clk);
    assert (OUT == 8'h55);
    @(negedge clk);

    A = 8'hFF;
    B = 8'h01;
    @(negedge clk);
    assert (OUT == 16'h100);
    @(negedge clk);

    @(negedge clk);
    $finish;

end
```

What are some of the issues with this approach?

What should the approach be?

- Start with a Verification plan
- A Verification plan talks about:
  - various design features and scenarios that need to be tested
  - architecture of the testbench
  - reuse in higher level testbenches

- Testbench should have the ability to:
  - test as many input data and opcode combinations as possible
  - test different orders of opcodes
  - stress key features/combinations
  - use more machine time and less human time

SystemVerilog

- SystemVerilog as a hardware verification language provides a rich set of features
- Data Types & Aggregate data types
  - Class, Event, Enum, Cast, Parameterization, Arrays, Associative arrays, Queues and manipulating methods
- OOP functionality
  - Classes, Inheritance, Encapsulation, Polymorphism, memory management
- Processes
  - fork-join control, wait statements
- Clocking blocks
- Interprocess synchronization & communication
  - Semaphores, Mailboxes, named events
- Assertions
- Functional Coverage
- Virtual Interfaces
- Constraints

Components of a testbench

```
`include "alu_intf.svh"
`include "alu_trxn.svh"
`include "alu_mon.svh"
`include "alu_bfm.svh"
`include "alu_sb.svh"

module alu_tb ();

   logic          clk;
   logic [7:0]    A,B;
   logic [2:0]    opcode;
   logic [15:0]   OUT;
   logic          done;

   // Instantiate the interface
   alu_intf alu_if
     (
      .clk(clk),
      .A(A),
      .B(B),
      .opcode (opcode),
      .OUT(OUT)
      );

   // Instantiate the design
   ALU ALU (.*);
```

- The ALU testbench module now looks different
- It includes headers for various components
  - ALU Interface
  - ALU Transaction
  - ALU Monitor
  - ALU BFM (driver)
  - ALU Scoreboard
- It creates the interfaces
- It instantiates the DUT

ARM

Main Test

```
program test;
   int            num_trxn = 10;
   alu_bfm bfm   = new (alu_if);
   alu_sb  sb    = new (alu_if);

   // Generate a basic clock
   initial begin
      clk = 1'b0;
      forever begin
         #5 clk = !clk;
      end
   end

   // Call the testbench components
   initial begin
      fork
         bfm.drive();
         sb.check();
      join_none
   end

   // Give the test some time before $finish is called
   initial begin
      int i;
      for (i=0; i<num_trxn*10; i++) begin
         @(negedge clk);
      end

      $finish;
   end
```

- A program block is the main entry point
- A bfm object and a scoreboard object are created
- All the components are started
- A fork/join process ensures that they all start in parallel
- We exit the fork statement at 0 time
- Simulation is stopped when $finish is called
- Multiple initial blocks execute in parallel

Transaction class

```
class alu_trxn ;
  typedef enum
                {
                 ADD=0,
                 SUB=1,
                 MUL=2,
                 AND=4,
                 OR=5,
                 XOR=6,
                 XNOR=7
                 }
                OPTYPE;

   rand logic [7:0] a, b;
   rand logic [15:0] out;

   rand OPTYPE op;

   function print_trxn (string name);
      $info ("%s: op=%s A=%h, B=%h", name, op.name, a, b);
   endfunction // print_trxn

endclass // alu_trxn
```

- The ALU transaction class:
  - Uses an enum type for optype
  - Uses “rand” to declare inputs that need to be driven with random values
  - Has a print utility that can be used with a transaction handle/object

BFM/driver

```
class alu_bfm;
   virtual interface alu_intf alu_if;
   alu_trxn trxn;

   extern task drive ();
   extern task drive_trxn (alu_trxn trxn);

   function new (virtual interface alu_intf alu_if_in);
      alu_if = alu_if_in;
   endfunction // new

endclass // alu_bfm

task alu_bfm::drive ();

   while (1) begin
      trxn = new ();
      trxn.randomize();
      trxn.print_trxn("bfm");
      drive_trxn (trxn);
   end

endtask // drive

task alu_bfm::drive_trxn (alu_trxn trxn);
   @ alu_if.cb;
   alu_if.cb.opcode <= trxn.op;
   alu_if.cb.A <= trxn.a;
   alu_if.cb.B <= trxn.b;
   trxn.out <= alu_if.cb.OUT;
endtask // drive_trxn
```

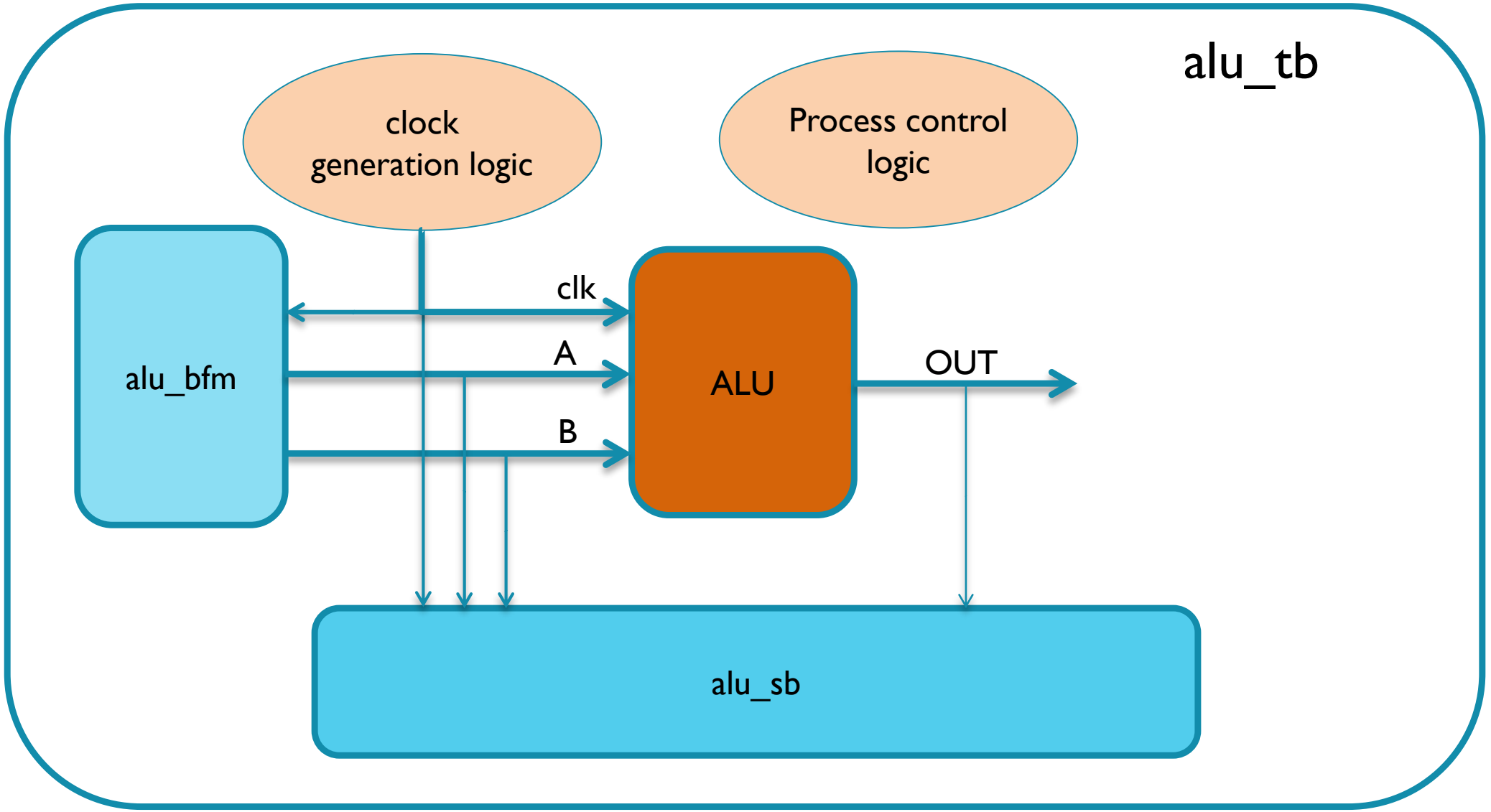
- The BFM/driver class:
  - Has a handle to a virtual interface
  - Declares a alu_trxn data type
  - Has a constructor
  - drive() task:
    - Does not end
    - Creates a new transaction
    - Randomizes the transaction
    - Passes the handle to drive_trxn() task
  - drive_trxn () task
    - consumes time
    - drives the input signals based on the values in the trxn class
    - Uses clocking block and non-blocking assignments
    - Adheres to pin level timing of signals

Scoreboard

```
function alu_sb::compute_expected_value (alu_trxn trxn_in);

   logic [7:0]    A, B;
   logic [15:0]   expected_out;
   logic [2:0]    opcode;

   A = trxn_in.a;
   B = trxn_in.b;
   opcode = trxn_in.op;

   case (opcode)
     3'b000: expected_out = A+B;  // ADD
     3'b001: expected_out = A-B;  // SUV
     3'b010: expected_out = A*B;  // MUL
     3'b100: expected_out = A&B;  // AND
     3'b101: expected_out = A|B;  // OR
     3'b110: expected_out = A^B;  // XOR
     3'b111: expected_out = ~A^B; // XNOR
   endcase // case (trxn_in.op)

   assert (expected_out == trxn_in.out)
     else
       $fatal (
               1,
               "op=%s, A=%h, B=%h, OUT=%h, expected_out=%h",
               trxn_in.op.name, A, B, trxn.out, expected_out
               );

endfunction // get_expected_value


task alu_sb::check ();
   int i;

   trxn = new ();

   @ alu_if.cb;

   while (1) begin
      @ (posedge alu_if.cb);
      $cast (trxn.op, alu_if.cb.opcode);
      trxn.a    <= alu_if.cb.A;
      trxn.b    <= alu_if.cb.B;
      trxn.out  <= alu_if.OUT;
      @ (alu_if.cb);
      compute_expected_value (trxn);
   end

endtask // check
```

- The Scoreboard:
  - Functionality is to continuously check the output independent of the input stimulus
  - check() task:
    - Collects information from the interface and populates the trxn class
    - Calls a compute_expected_out function
  - compute_expected_out() task
    - Implements the model of the deisign
    - Takes in the inputs and gets an expected output
    - Compares the actual output against the expected output
    - Issues an error message if the comparison fails

How does the testbench look like?



ARM

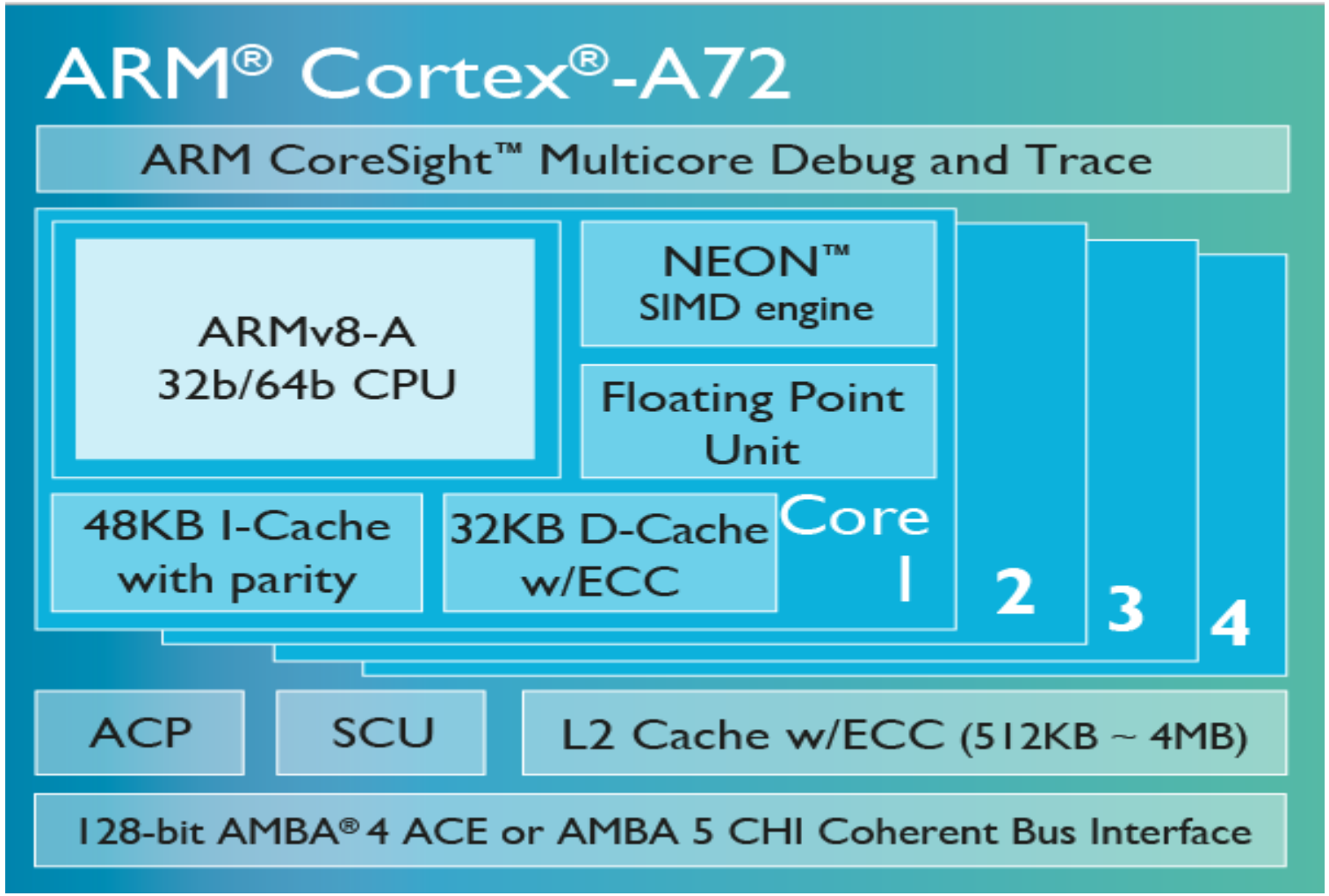
How do we know we are done?

- With a random testbench it is difficult to know what scenarios have been exercised
- Two techniques are typically used to get a measure of what's done
- Code Coverage
  - No additional instrumentation is needed
  - Toggle, Statement, Expression, Branch coverage
- Functional Coverage
  - Requires planning
  - Requires instrumenting code
  - SystemVerilog provide constructs to support functional coverage
  - Provides detailed reports on how frequently coverage was hit with the test sample
- Coverage closure is an important aspect of verification quality

What did we go over …

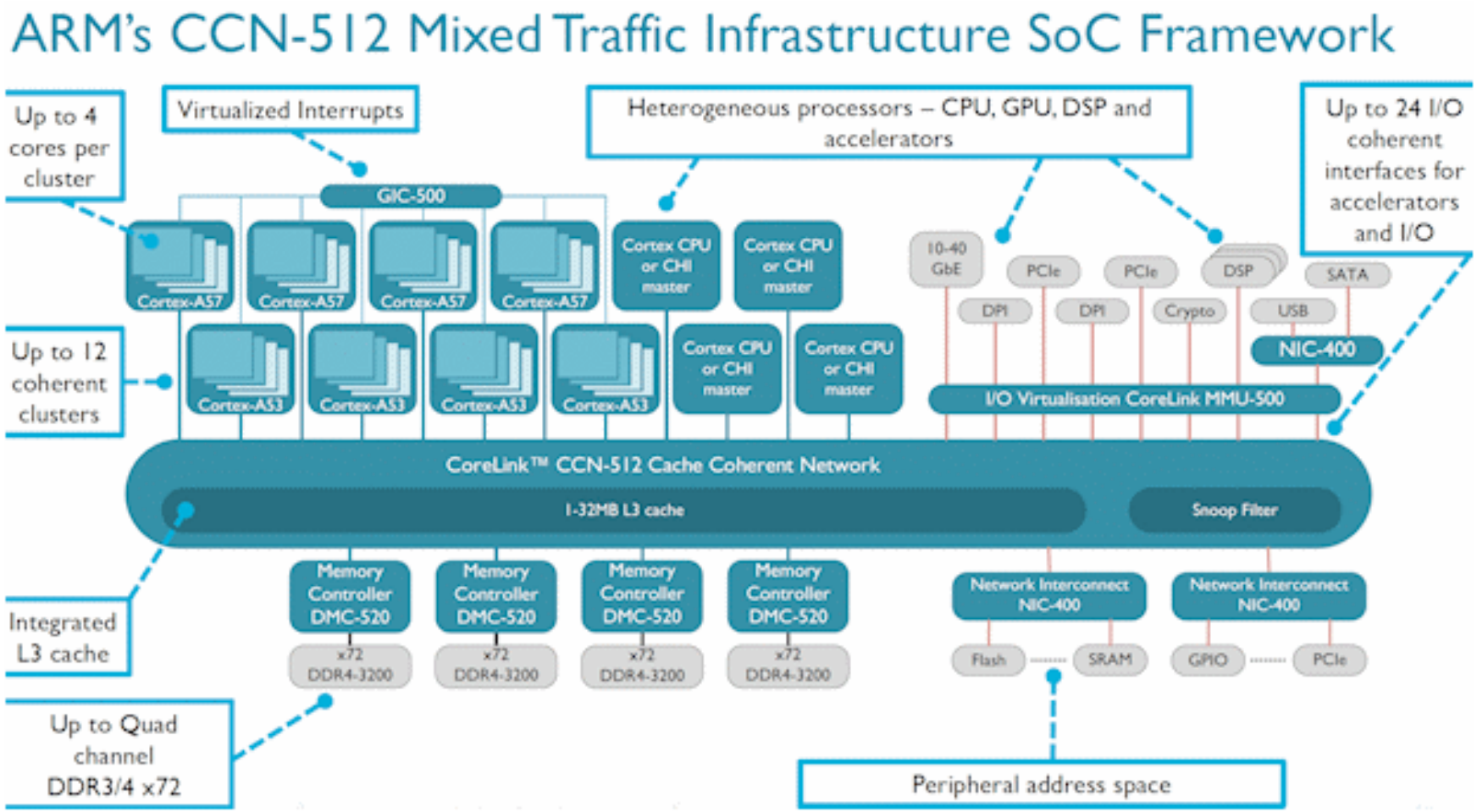
- Built a directed and random testbench
- Discussed various components of a testbench
- Modularized and built in complexity into a testbench … for a reason
- Demonstrated that verification and testbench development requires good planning and software skills

ARM Cortex A72 CPU



ARM

ARM CCN-512 SoC Framework



ARM

What are the challenges of verifying complex systems?

- Typical processor development from scratch could be about 100s of staff years effort
- Multiple parallel developments, multiple sites and it takes a crowd to verify a processor
- The challenges are numerous – especially when units are put together as a larger unit or a whole processor and verified
- Reuse of code becomes an absolute key to avoid duplication of work
- Multiple times it is essential to integrate an external IP into your system
- The IP can come with it's own verification implementation
- This requires rigorous planning, code structure, & lockstep development
- Standardization becomes a key consideration

- So how do we solve this?

Useful pointers

- https://verificationacademy.com/
- SV Unit YouTube Video
- EDA Playground
- http://testbench.in/
- Search for SystemVerilog on YouTube

Let's solve this …

```
class base;
  int a;
  static bit b;
  function new(int val);
    a = val;
  endfunction
  virtual function void say_hi();
    $display($psprintf("hello from base (a == %0d)", a));
  endfunction
class sub extends base;
  int c;
  function new(int val);
    super.new(val);
    c = val;
  endfunction
  virtual function void say_hi();
    super.say_hi();
    $display($psprintf("hello from sub (c == %0d)", c));
  endfunction
endclass
```

```
program a;
  initial begin
    base b1, b2;
    sub s1, s2;
    othersub os1;

    b1 = new(1);
    s1 = new(2);
    os1 = new(3);

    s1.say_hi();
    os1.say_hi();

    b2 = os1;
    b2.say_hi();

    $display($psprintf("b == %0d", base::b));
    $display($psprintf("b == %0d", othersub::b));
  end
endprogram
```