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 a reusable 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. It 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. It is used to verify the design output against the modeled output.
  - **Coverage**: Helps measure quality of stimulus. It 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
How was it done in the past?

```verilog
initial begin
@negeedge clk;
opcode = 3'b000; // ADD
A = 8'h05;
B = 8'h50;
@negeedge clk;
assert (OUT == 8'h55);
@negeedge clk;
A = 8'hFF;
B = 8'h01;
@negeedge clk;
assert (OUT == 16'h100);
@negeedge clk;
@negeedge clk;
$finish;
end
```

What are some of the issues with this approach?

A bit of planning goes a long way

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

- 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 instantiates the DUT
Main Test

- 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

```verilog
program test;
int num_trxn = 30;
alu_bfm bfm = new (alu_if);
alu_tb sb = new (alu_if);

// Generate a basic clock
initial begin
    clk = '1'b;
    forever begin
        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; i++) begin
        $regwrite clk;
    end
    $finish;
end
```

Transaction class

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

```verilog
class Alu_trxn;
    typedef enum
        ADD=0,
        SUB=1,
        ML=2,
        AND=4,
        OR=5,
        HD=6,
        XOR=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

endclass // alu_trxn
```
The BFM/driver class:
- Has a handle to a virtual interface
- Declares an 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

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 design
  - 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?

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

Let’s solve this …

- If time permits, I’ll throw in a System Verilog problem that we can solve together!
Useful pointers

- [https://verificationacademy.com/](https://verificationacademy.com/)
- [SV Unit YouTube Video](https://www.youtube.com/watch?v=dQw4w9WgXcQ)
- [EDA Playground](http://eda-playground.com/)
- [http://testbench.in/](http://testbench.in/)
- Search for SystemVerilog on YouTube

---

**ARM Cortex A72 CPU**
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?