Training on
IP & SoC Functional Verification Methodology
Using UVM

LAB
Test Sequences

Objectives

This lab goes through the concept of sequences.

- Goals:
  - Understand the concept of UVM sequences
  - Implement directed test sequences
  - Implement random test sequences
- Design Example:
  - UART 16550 (opencores.org)

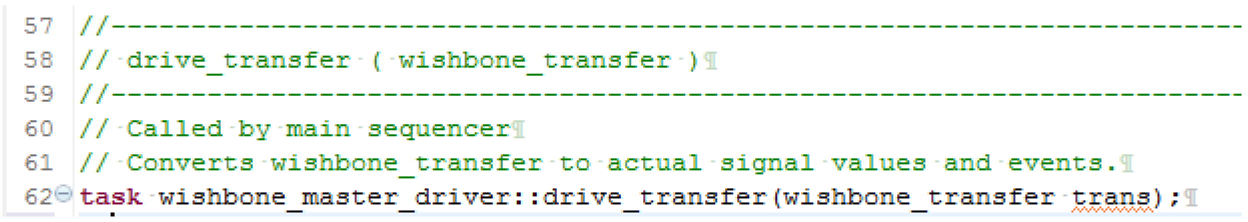
Introduction

The design under test (DUT) is a UART 16550 from opencores.org.
The goal of this design is to perform serial transfers on the Tx line and receive Rx transfers from the Rx line, given registers programmable thru a APB interface.

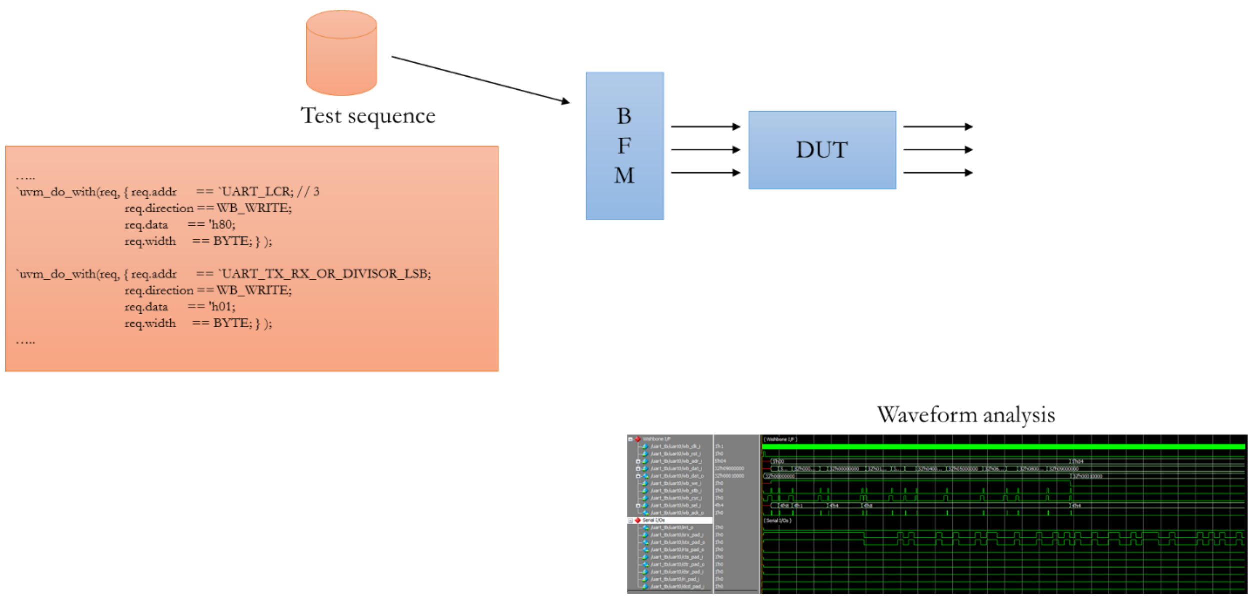
The testbench includes a APB Verification IP which main driver class is defined and implemented in the following two files:

- ambersoc\aedvices\vip\APB\src\sv\APB_master_driver.svh (declaration)
- ambersoc\aedvices\vip\APB\src\sv\APB_master_driver.sv (implementation)

The SystemVerilog task **drive_trans()** is responsible of driving the APB bus interface from generated transactions of the test sequences.



```
//----------------------------------------------------------------------
// drive_transfer ( wishbone_transfer )
//----------------------------------------------------------------------
// Called by main sequencer
// Converts wishbone_transfer to actual signal values and events.
task wishbone_master_driver::drive_transfer(wishbone_transfer trans);
```



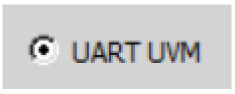
```
…..
`uvm_do_with(req, { req.addr      == `UART_LCR; // 3
                    req.direction == WB_WRITE;
                    req.data      == 'h80;
                    req.width     == BYTE; } );

`uvm_do_with(req, { req.addr      == `UART_TX_RX_OR_DIVISOR_LSB;
                    req.direction == WB_WRITE;
                    req.data      == 'h01;
                    req.width     == BYTE; } );
…..
```

The goal of this lab is to implement test sequences to exercise the different functionalities of the UART.

Instructions

Follow instructions given in “aedv_training_labs_intructions_for_questa.pdf”.
Open the file “<SANDBOX>/labs-Xdays/labNN-uvm_sequences/lab.sv”
Select


UART UVM

Implement a random init sequence

The provided test is a “directed” test and only simulate one specific configuration.

In order to automate different test for different configurations, we want to implement an init sequence which randomly choose the following configuration fields:

- Interrupt enable or disable (register IER bit 0)
- Clock divider set to random values between 1 to 0x10 (register DIVISOR_LSB)
- Parity even or odd
- Char length set to 5 to 8 randomly

For this, we will create a new sequence named “lab4_uart_init_seq” based on the following template

```
class MY_SEQ_NAME extends wishbone_base_sequence;
    // Register the sequence into the UVM framework
    `uvm_object_utils(MY_SEQ_NAME)

    // Some random parameters
    rand bit enable;
    rand int freq_div;

    // Some constraints
    constraint freq_div_c { freq_div inside {[1:10]};

    // Sub Sequence
    MY_SUB_SEQ subseq;

    // Main sequence body
    task body();
        if ( enable )
            `uvm_do(req)
        `uvm_do(subseq)
    endtask
endclass
```

Instructions:

- Search for `LAB-TODO-STEP-1-a`
- Create a new class "lab4_uart_init_seq" based on the above template
    - Add random parameters for each of the following
        - Interrupt enable or disable
        - Clock divider value
        - Parity even or odd
        - Char length
    - Search for `LAB-TODO-STEP-1-b`
      From the existing lab4_test_sequence, cut and paste the init part doing the register accesses (between the comments "UART Init start" and "Transmit Data") to the newly created lab4_uart_init_seq class.
    - Modify the copied register values accordingly using the random parameters.
    - Add a constraint to force the parity to be set to even when the width is set to 8

- In "lab4_test_sequence"
    - Search for `LAB-TODO-STEP-1-c`
    - Instantiate the new init sequence
    - Search for `LAB-TODO-STEP-1-d`
    - Call the init sequence
      `uvm_do(init_seq)

More Random Testing

Exercise:

- Search for `LAB-TODO-STEP-2-a`
- Create a sequence A which
    - Performs a random number (between 10 and 100) of write to the transmit buffer of random data. Cut and paste the template found on `LAB-TODO-STEP-2-b`

Note: After writing to the Tx buffer, poll check the transmit FIFO status register:

```
// Poll register: UART_LSR 020
fifo_emtpy = 0;
while ( fifo_empty == 0 ) begin
      `uvm_do_with(req , { req.addr == `UART_LSR;
                           req.direction == WB_READ;
                           req.width     == BYTE;} )

    if ( req.data[5] == 1 )
          fifo_empty = 1;
end
```

- Search for `LAB-TODO-STEP-2-c`
- Instantiate the test sequence
- `LAB-TODO-STEP-2-d`
- Use uvm_do to call the newly created test sequence

- Search for `LAB-TODO-STEP-3-a`
- Create a sequence B which
    - Performs a random number (between 10 and 100) of write to the transmit buffer of incremental data (start with a random number, then increment)

- Search for `LAB-TODO-STEP-4-a`
- Create a sequence C which performs
    - The init sequence with only 8 bit char width
    - Select randomly one of the above two random sequences A or B (hint: use randcase)
- Create a test sequence which randomly selects one of the above sequences (A, B , C , init) in a loop of 10 to 20 subsequences (see slides).
- Search for `LAB-TODO-STEP-4-b`
- Make the test to call your new test sequences

Constraining Delays using uvm_do

Exercise:

- In one of the previous random sequences, add a "delay" random variables that take a random value between 1 and 20.
- Modify the uvm_do_with to pass this delay to all the generated transaction (at least in the polling loop)

  `uvm_do_with(req , { req. delay == local:: delay; })

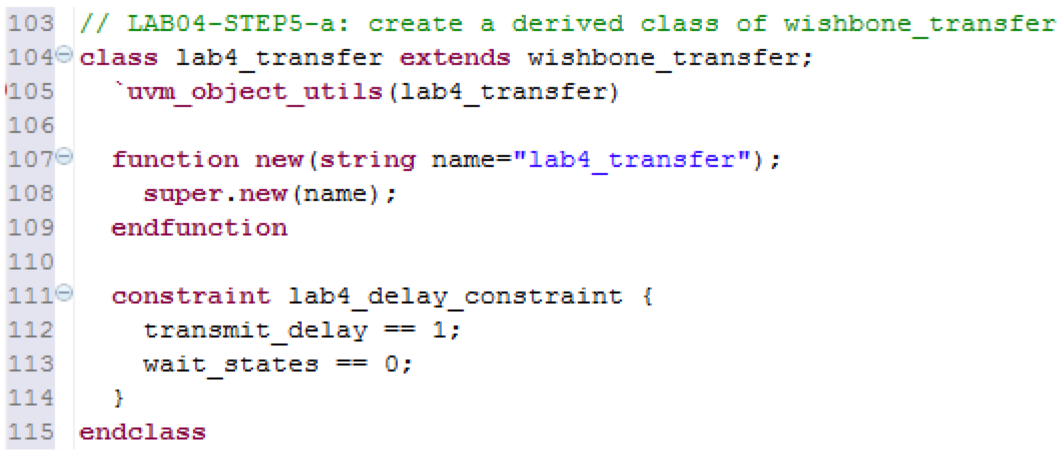
Rerun a test which runs the sequence. What do you observe?

Constraining Delays using derived class and the UVM factory

We now want to over constrain all transactions. However we don't want to modify the base APB_transfer class nor each of the `uvm_do call.
The concept of the factory is an Object Oriented Programming pattern which allow to register a new class as the one to use for a specific object. It allows to dynamically choose the type of an object at its creation time.
The UVM class libraries implement the factory thru the registering macros `uvm_object_utils and `uvm_component_utils which register each class in the factory, and the factory object which allows to replace any instance.
We will use this mechanism to replace the base APB_transfer class by our own implementation.

Exercise:

- Search for `LAB-TODO-STEP5-a`
- Implement a new class "lab_transfer" which forces the delay to be set to 1
-



```
// LAB04-STEP5-a: create a derived class of wishbone_transfer
class lab4_transfer extends wishbone_transfer;
  `uvm_object_utils(lab4_transfer)

  function new(string name="lab4_transfer");
    super.new(name);
  endfunction

  constraint lab4_delay_constraint {
    transmit_delay == 1;
    wait_states == 0;
  }
endclass
```

-

- Rerun the test. Does it constrain the transaction delays?
- Search for
- Add the following line in the build_phase of the test.

```
factory.set_type_override_by_type(apb_transfer::get_type(),la
b_apb_transfer::get_type());
```

- Rerun the test. Does it constraint the transaction delays?

Appendix A: UART registers

Note: Full detailed registers are described in the UART specification under opencores/docs

4.1 Registers list

Name	Address	Width	Access	Description
Receiver Buffer	0	8	R	Receiver FIFO output
Transmitter Holding Register (THR)	0	8	W	Transmit FIFO input
Interrupt Enable	1	8	RW	Enable/Mask interrupts generated by the UART
Interrupt Identification	2	8	R	Get interrupt information
FIFO Control	2	8	W	Control FIFO options
Line Control Register	3	8	RW	Control connection
Modem Control	4	8	W	Controls modem
Line Status	5	8	R	Status information
Modem Status	6	8	R	Modem Status

In addition, there are 2 Clock Divisor registers that together form one 16-bit.
The registers can be accessed when the 7th (DLAB) bit of the Line Control Register is set to '1'. At this time the above registers at addresses 0-1 can't be accessed.

Name	Address	Width	Access	Description
Divisor Latch Byte 1 (LSB)	0	8	RW	The LSB of the divisor latch
Divisor Latch Byte 2	1	8	RW	The MSB of the divisor latch