UVM Framework
ALU TUTORIAL
A Step By Step Guide

Using UVMF_2019.4

Sept 2016

Mentor®
A Siemens Business

Agenda

- Introduction
- ALU Overview
- Config Files Explained
- Compile and Simulate Generated Code
- Adding DUT Specific Functionality

overview

Tutorial Aims

- **UVM Framework Steps**
  - The main aim is to take the user through the complete flow (from start to finish) to show how to generate a working UVM testbench for a simple IP using the UVM Framework (UVMF) code generators
  - Each step is explained to help the user understand what information has to be provided to the UVMF code generators in order to produce the initial testbench infrastructure
  - The tutorial also highlights where the user is expected to modify the generated code to add DUT specific functionality in order to create a fully operational UVM testbench

Tutorial Logistics

- **YAML Configuration Files**
  - YAML configuration files are used as the inputs to the UVM Framework code generators and they determine the content of the generated code
  - A set of completed YAML configuration files for the ALU project are located within the **yaml_config_files** folder
    - The user can optionally use these as a starting point to get the initial code generated
  - The UVMF Code Generator will NOT overwrite existing code. This ensures that any edits you have subsequently made to the generated code cannot be inadvertently overwritten
    - If code already exists in the specified output directory then the UVMF Code generators will issue "skipping" messages like the following:

```
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\src\ALU_out_monitor.svh, already exists
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\.project, already exists
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\ALU_out_filelist_hdl.f, already exists
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\src\ALU_out_sequence_base.svh, already exists
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\src\ALU_out_if.sv, already exists
Skipping C:\Temp\alu_3_6h\uvmf_template_output\verification_ip\interface_packages\ALU_out_pkg\ALU_out_pkg_sve.F, already exists
```

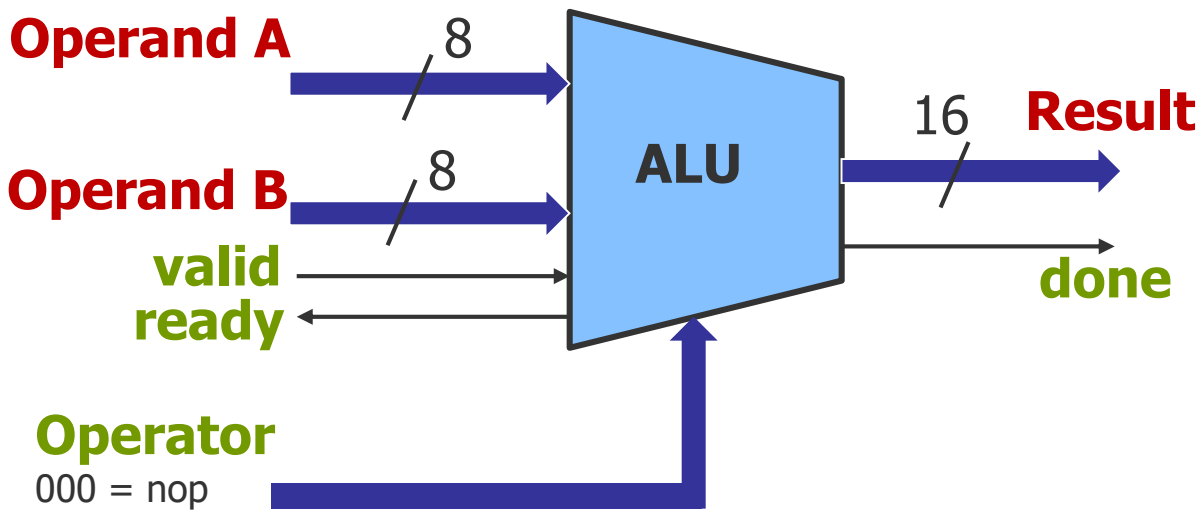
- **Completed Solution**
  - A fully completed version of the ALU UVMF testbench is provided within the ***completed_solution*** folder as a reference to the user
  - This contains all of the modification made after the initial code generation to add the DUT specific details

Agenda

- Introduction
- ALU Overview
- Config Files Explained
- Compile and Simulate Generated Code
- Adding DUT Specific Functionality

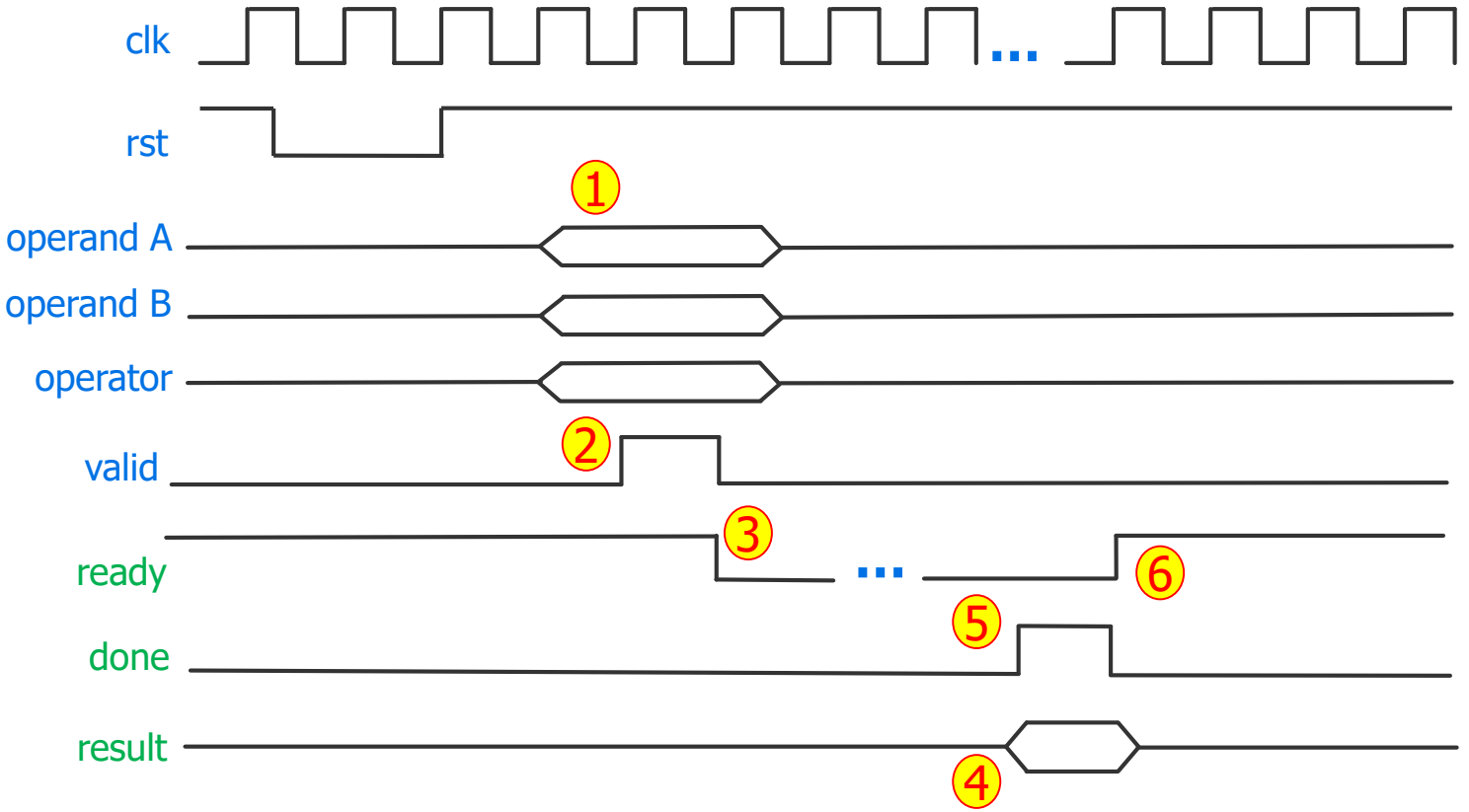
overview

ALU : Block Diagram



overview

ALU : Timing Diagram



1. Apply operands & operation on input pins
2. Raise valid for 1 cycle (start)
3. ALU drops ready signals
4. After X cycles, ALU presents result
5. ALU raises done for 1 cycle
6. ALU raises ready signal

Agenda

- ALU Overview
- Config Files Explained
- Compile and Simulate Generated Code
- Adding DUT Specific Functionality

YAML

YAML Config Files

- **YAML** (*YAML Ain't Markup Language*)
  - A human readable language used for capturing data
  - Has minimal syntax and uses Python style indentation to indicate nesting
  - Used in UVMF to capture information at a high level that can subsequently be used by code generators to construct elements of a UVM testbench code

- **YAML File Format**
  - All UVMF YAML files must be presented as part of a specific top level format shown opposite
  - Whitespace indentation is used to denote structure; however, tab characters are never allowed as indentation
  - Comments start with the number sign (#), can start anywhere on a line and continue until the end of the line
  - The information can be spread across multiple files or can be contained in a single file

```
# comments in YAML look like this

uvmf:
        interfaces:
                "<interface_nameA>"
                        <properties>
                "<interface_nameB>"
                        <properties>
        util_components:
                "<util_component>"
                        <properties>
        environments:
                "<env_nameA>"
                        <properties>
        benches:
                "<bench_nameA>"
                        <properties>
```

YAML Config Files

- **Number Of Agents**
  - First need to determine how many interfaces there are for the ALU
  - Group signals into interfaces
  - Create a YAML configuration file for each interface
    - The YAML config file captures the pin information and the transaction information for the interface (agent)
  - For the ALU, you have 2 separate interface files, 1 environment file and 1 bench file

- **Required ALU Interface Config files**

  - **ALU_IN Interface**
    - All signals that are associated with specifying the ALU operation to perform
  - **ALU_OUT Interface**
    - All signals that are associated with specifying the ALU result

YAML

Interface Config File
ALU_in_interface.yaml

uvmf:

interfaces:

ALU_in:

- Tabbed indents are required
- Identifies that subsequent data information is for a UVMF interface to be named 'ALU_in'

clock: clk

- Identifies the primary clock to be used in the interface agent as 'clk' [1]

reset: rst

reset_assertion_level: 'False'

- Identifies the primary reset to be used in the interface agent as 'rst' with active low polarity

alu_in_interface.yaml

```
1 uvmf:
2   interfaces:
3     ALU_in:
4       clock: clk
5       reset: rst
6       reset_assertion_level: 'False'
```

NOTES:

1. Clock signal is required. Additional clocks will have to be manually added after code generation

in_ifc

Interface Config File
ALU_in_interface.yaml

config_constraints: []

config_vars: []

- This is where you can specify configuration variables and any corresponding constraints for the agents
- Not being use for this ALU example

hdl_typedefs:

- name: alu_in_op_t

type: enum bit[2:0] {no_op = 3'b000, add_op ……., rst_op = 3'b111}

- Define any types used by the HDL side of the testbench
- Here you define the valid ALU operations and specify the bit values for each operation

hvl_typedefs:

- Define any types used by the HVL side of the testbench

```
8       config_constraints: []
9       config_vars: []
10
11      hdl_typedefs:
12      - name: alu_in_op_t
13        type: enum bit[2:0] {no_op = 3'b000, add_op = 3'b001, and_op = 3'b010, xor_op
14          = 3'b011, mul_op = 3'b100, rst_op = 3'b111}
15      hvl_typedefs: []
16
```

in_ifc

Interface Config File
ALU_in_interface.yaml

parameters:

- **name: ALU_IN_OP_WIDTH**
  type: int
  value: '8'
  - Defines an integer parameter named 'ALU_IN_OP_WIDTH' which has a default value of 8
  - Any parameters defined here will impact be passed to multiple classes within the agent

```
17      parameters:
18      - name: ALU_IN_OP_WIDTH
19        type: int
20        value: '8'
21
```

in_ifc

Interface Config File
ALU_in_interface.yaml

ports:

- **dir: output**
- **name: alu_rst**
- **width: '1'**

— Here we define all of the signal names, directions and widths for the agent/interface

NOTES:

— Direction specified here is in relation to the testbench
  - i.e. 'alu_rst' is an output from the testbench and an input pin on the DUT

— The agent has to be able to execute a 'rst_op' operation and will need to drive the ALU reset pin in response to such a request

— The 'a' & 'b' use the ALU_IN_OP_WIDTH parameter which was defined under the **parameters** section of the config file

```
21
22      ports:
23      - name: alu_rst
24        dir: output
25        width: '1'
26      - name: ready
27        dir: input
28        width: '1'
29      - name: valid
30        dir: output
31        width: '1'
32      - name: op
33        dir: output
34        width: '3'
35      - name: a
36        dir: output
37        width: ALU_IN_OP_WIDTH
38      - name: b
39        dir: output
40        width: ALU_IN_OP_WIDTH
41
```

in_ifc

Interface Config File
ALU_in_interface.yaml

response_info:

data: []

operation: 1'b0

- The ***data*** directive allows the user to specify what response data should be passed back from the driver to the originating sequence. We have no response data for this ALU interface
- The ***operation*** directive allows the user to define if the driver should pass any response data back to the sequence. Here we set the value to 1'b0 which tells the driver not to send back any response

```
41
42      response_info:
43        data: []
44        operation: 1'b0
45
```

NOTES:

- Failing to disable responses when none are returned, will result in run times errors reporting 'response queue overflow'

in_ifc

Interface Config File
ALU_in_interface.yaml

transaction_contraints:

- **name : valid_op_c**
- **value: `{op inside {no_op,add_op, and_op, xor_op, mul_op};}'**

— Defines any constraints to be used on the transaction variables
— Here we define a constraint named 'valid_op_c' to contain all valid ALU operands except the rst_op operand (as we only want to issue rst_op operands in a directed test type mode)
— The syntax for the constraint is pure SystemVerilog

```
46      transaction_constraints:
47      - name: valid_op_c
48        value: '{ op inside {no_op, add_op, and_op, xor_op, mul_op}; }'
49
50      transaction_vars:
51      - name: op
52        type: alu_in_op_t
53        iscompare: 'True'
54        isrand: 'True'
55      - name: a
56        type: bit [ALU_IN_OP_WIDTH-1:0]
57        iscompare: 'True'
58        isrand: 'True'
59      - name: b
60        type: bit [ALU_IN_OP_WIDTH-1:0]
61        iscompare: 'True'
62        isrand: 'True'
63
```


in_ifc

Interface Config File
ALU_in_interface.yaml

transaction_vars:

- **iscompare : 'True'**
- **israND: 'True'**
- **name: 'op'**

  - Defines any variable to be used by the transaction class.
  - Variables in the transaction class reflect the untimed information used during a transfer on the bus
  - For the ALU, the transaction will specify the operation and the a & b operands
  - Each of these transaction variables can be randomized since we specified israND: 'True'
  - Within the generated transaction class, each of the transaction variables will be included in a do_compare method since we specified iscompare: 'True'

NOTES:

- Unconstrained arrays cannot be specified
- If you need an unconstrained array, declare a fixed array and modify the generated code

```
46      transaction_constraints:
47      - name: valid_op_c
48        value: '{ op inside {no_op, add_op, and_op, xor_op, mul_op}; }'
49
50      transaction_vars:
51      - name: op
52        type: alu_in_op_t
53        iscompare: 'True'
54        israND: 'True'
55      - name: a
56        type: bit [ALU_IN_OP_WIDTH-1:0]
57        iscompare: 'True'
58        israND: 'True'
59      - name: b
60        type: bit [ALU_IN_OP_WIDTH-1:0]
61        iscompare: 'True'
62        israND: 'True'
63
```

in_ifc

Generated UVMF Code
ALU_in interface package

- **Generating the Interface Code**
  - Execute the following command to generate the ***ALU_in*** agent code

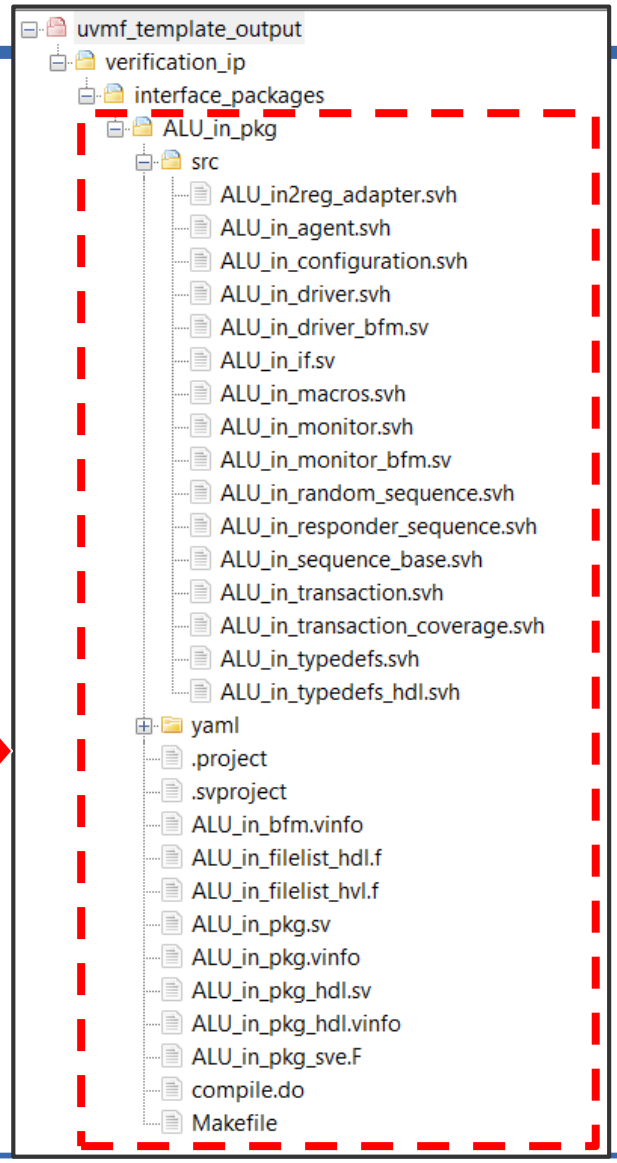
    python $UVMF_HOME/scripts/yaml2uvmf.py ALU_in_interface.yaml

  - You can create a simple .bat file on Windows to set the $UVMF_HOME environment variable and then call python on your config file

```
1  @set QUESTA_ROOT=C:/MentorTools/questasim_2019.2
2  @set UVMF_HOME=C:/graemej/UVM_FRAMEWORK/UVMF_Repo_2019.4
3
4  python %UVMF_HOME%/scripts/yaml2uvmf.py ALU_in_interface.yaml
5
6  pause
```

  - All UVMF agent code is placed under **uvmf_template_output / verification_ip / interface_packages**

  - All generated code for the **ALU_in** agent will be saved under the **ALU_in_pkg** folder [as shown opposite]



in_ifc

Interface Config File
parameterized ALU agent

```
17      parameters:
18      - name: ALU_IN_OP_WIDTH
19        type: int
20        value: '8'
21
```

- **Impact of Using Parameters**
  - All classes and interfaces within the generated UVMF code for the ALU agent will be parameterized with the parameters we specified in the YAML configuration file
  - ***Tip***: Only use parameters if you really need them as it does complicate the generated code

```
interface  ALU_in_if  #(
  int ALU_IN_OP_WIDTH = 8)
(
```

```
class ALU_in_monitor  #(
      int ALU_IN_OP_WIDTH = 8
      ) extends uvmf_monitor_base
```

```
class ALU_in_transaction      #(
    int ALU_IN_OP_WIDTH = 8
    ) extends uvmf_transaction_base;
```

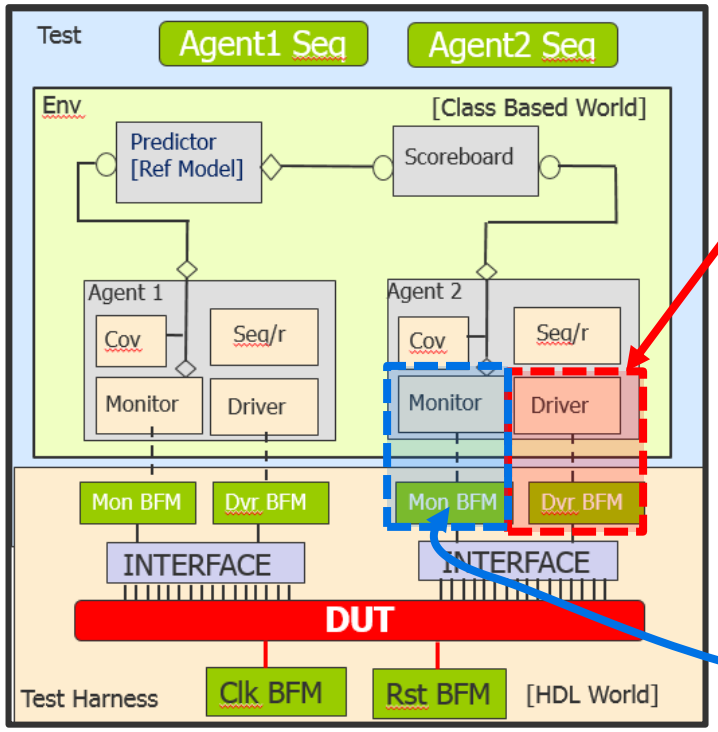
```
class ALU_in_configuration #(int ALU_IN_OP_WIDTH = 8) extends uvmf_parameterized_agent_configuration_base #(
                           .DRIVER_BFM_BIND_T(virtual ALU_in_driver_bfm #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH))),
                           .MONITOR_BFM_BIND_T(virtual ALU_in_monitor_bfm #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH)))
                          );
```

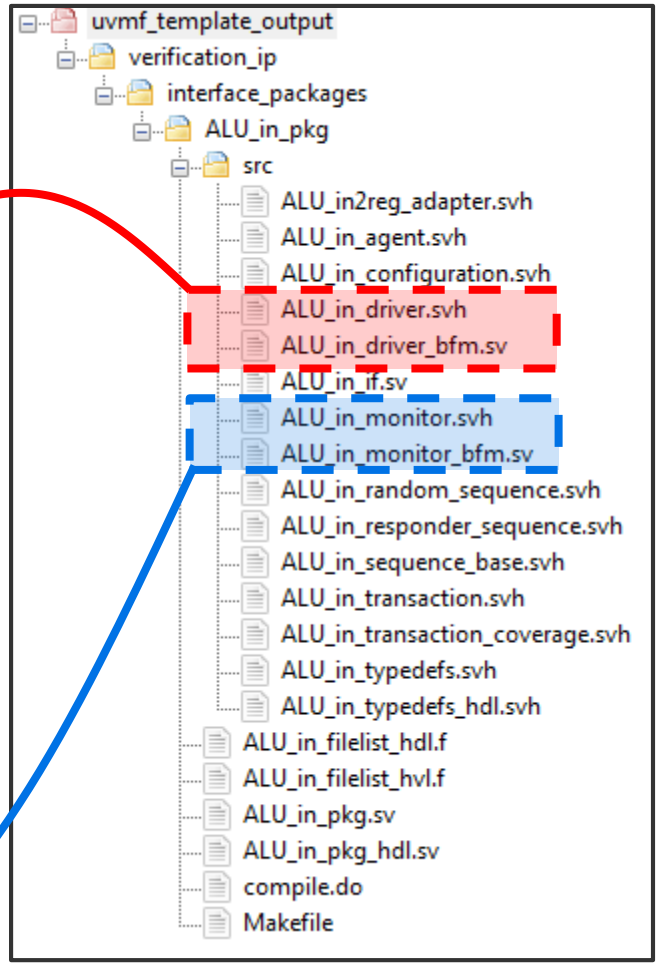
```
class ALU_in_agent #(
     int ALU_IN_OP_WIDTH = 8
     ) extends uvmf_parameterized_agent #(
                    .CONFIG_T(ALU_in_configuration #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH) )),
                    .DRIVER_T(ALU_in_driver #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH) )),
                    .MONITOR_T(ALU_in_monitor #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH) )),
                    .COVERAGE_T(ALU_in_transaction_coverage #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH) )),
                    .TRANS_T(ALU_in_transaction #(.ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH) ))
                   );
```

in_ifc

Generated UVMF Code
ALU_in

- **UVMF Transactors**
  - ALU_in_driver & ALU_in_monitor are class based and will be instantiated inside the agent class
  - ALU_in_driver_bfm & ALU_in_monitor_bfm are interfaces & will be instantiated in the top level testbench module (hdl_top)





Generated UVMF Code
ALU_in

- **Looking at the *ALU_in_pkg* directory**
- **Contains the following key files**

— **ALU_in_filelist_hdl.f**
Compilation list of hdl files (the interface and the 2 BFMs)

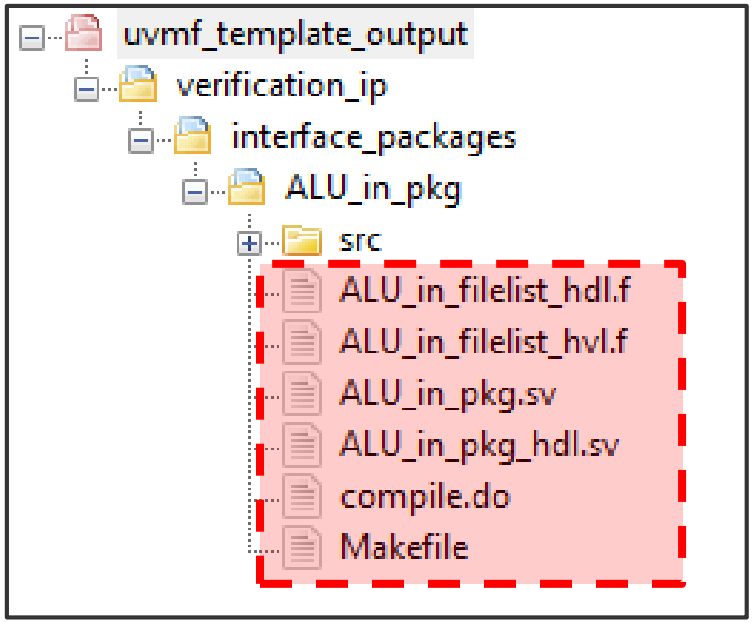
— **ALU_in_filelist_hvl.f**
Compilation list of hvl files (all other files)

— **ALU_in_pkg.sv**
This is the verification package (HVL) that `includes all the generated classes for our VIP agent (all from directory src)

— **ALU_in_pkg_hdl.sv**
This package will be used for the HDL part of the VIP. The HDL part is synthesized by the emulator.

— **compile.do**
Contains the compile command for the generated agent. Use on Windows or Linux

— **Makefile**
Contains the compile commands for the generated agent. Use on Linux

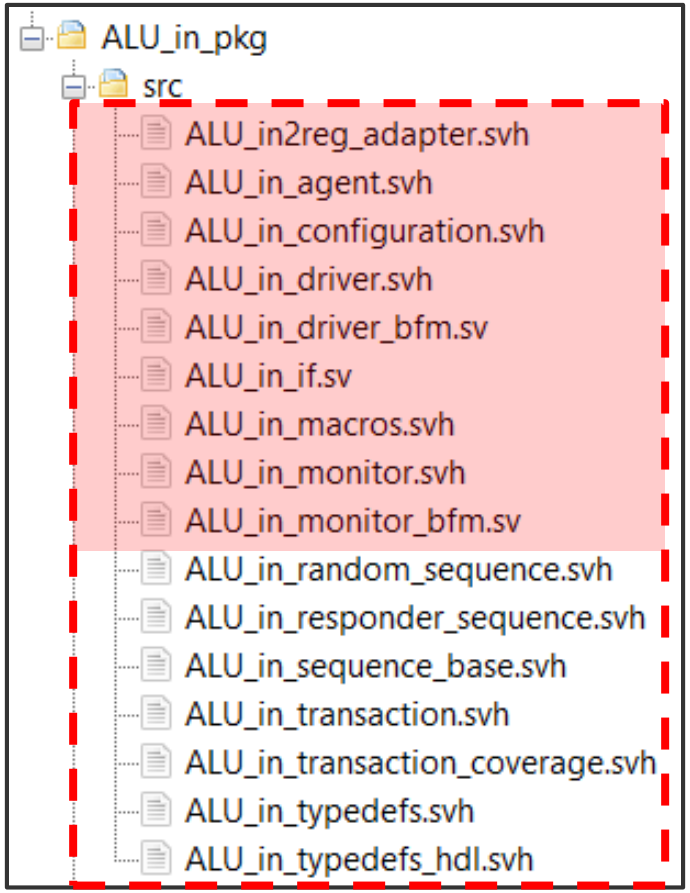


NOTES:
There may be some additional meta-data files generated in the ALU_in_pkg directory. These can be ignored.

in_ifc

Generated UVMF Code
ALU_in

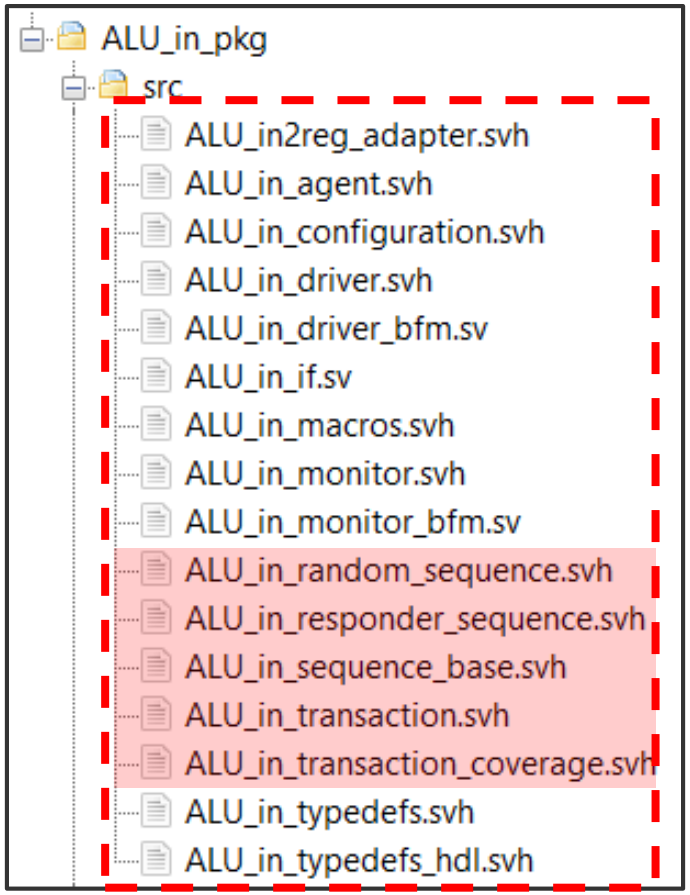
- **Looking at the *ALU_in_pkg/src* directory**
  - **ALU_inreg_adaptor.svh**
    Template adaptor for UVM register layer. Requires user to fill in functionality.
  - **ALU_in_agent.svh**
    Agents class (parameterized)
  - **ALU_in_configuration.svh**
    Configuration class for the agent
  - **ALU_in_driver.svh**
    Driver class to be instantiated in the agent
  - **ALU_in_driver_bfm.sv**
    Bus functional model to convert transactions to protocol pin wiggles. Requires user to fill in functionality
  - **ALU_in_if.sv**
    Signal interface for the agent. User can optionally add protocol assertions in here.
  - **ALU_in_macros.sv**
    Defines structs used to pass data between classes, hvl, BFMs and hdl
  - **ALU_in_monitor.svh**
    Monitor class to be instantiated in the agent
  - **ALU_in_monitor_bfm.sv**
    Bus functional model to convert the protocol pin wiggles to transactions. Requires user to fill in functionality



in_ifc

Generated UVMF Code
ALU_in

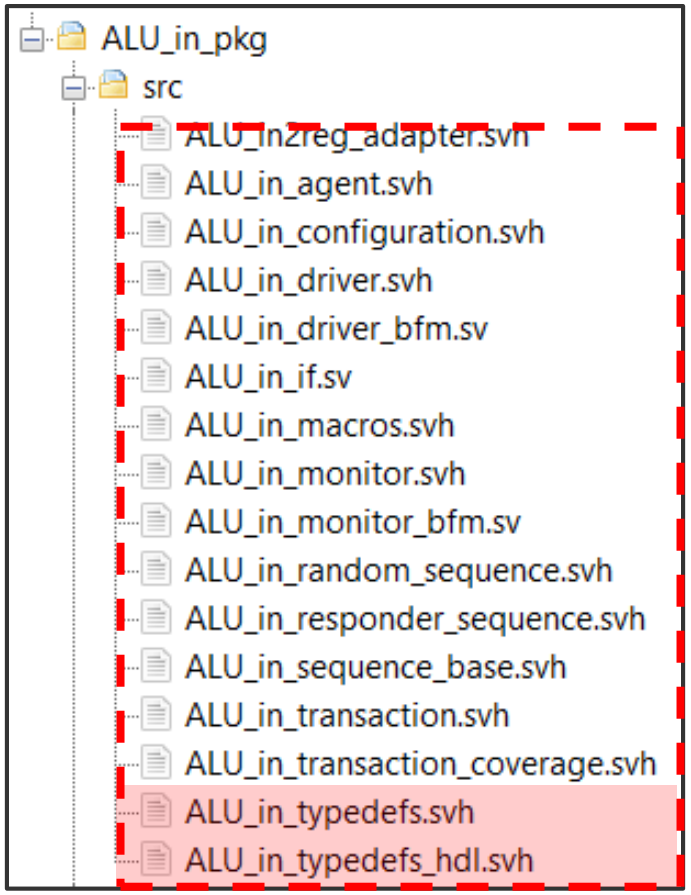
- **Looking at the *ALU_in_pkg/src* directory**
  - **ALU_in_random_sequence.svh**
    Starter sequence. Randomizes 1 instance of the ALU_in transaction class and sends to sequencer.
    Extended from ALU_in_sequence_base
  - **ALU_in_responder_sequence.svh**
    This sequence class can be used to provide stimulus when an interface has been configured to run in a responder mode. Requires user to fill in functionality
  - **ALU_in_sequence_base.svh**
    Base class with useful methods that all inherited sequences can utilize
  - **ALU_in_transaction.svh**
    This is the sequence item that we will use in our sequences. Extends from 'uvmf_transaction_base.svh' which contains global "id" which holds a unique number for every transaction. Also contains several methods for printing, comparing, etc
  - **ALU_in_transaction_coverage.svh**
    This class records ALU_in transaction information using a covergroup named ALU_in_transaction_cg.
    An instance of this coverage component is instantiated in the uvmf_parameterized_agent if the has_coverage flag is set



in_ifc

Generated UVMF Code
ALU_in

- **Looking at the *ALU_in_pkg/src* directory**
  - **ALU_in_typedefs.svh**
    This file contains defines and typedefs used only in the testbench (HVL) side of the testbench. Package may not contain any defines or typedefs after but will still be generated
  - **ALU_in_typedefs_hdl.svh**
    This file contains defines and typedefs used by the interface package performing transaction level simulation activities. This package is used by the driver/monitor BFMs



in_ifc

Interface Config File
ALU_out_interface.yaml

```
uvmf:
    interfaces:
        ALU_out:
            clock: clk
            reset: rst
            reset_assertion_level: 'False'
```

ALU_out_interface.yaml

```
1 uvmf:
2   interfaces:
3     ALU_out:
4       clock: clk
5       reset: rst
6       reset_assertion_level: 'False'
```

- Separate YAML configuration file for the ALU output signal interface
- Agent name = ALU_out
- Primary clock = clk
- Primary reset = rst, with active low polarity

out_ifc

Interface Config File
ALU_out_interface.yaml

config_constraints: []

config_vars: []

- No configuration variables or constrains specified for the ALU_out agent

hdl_typedefs: []

hvl_typedefs: []

- No types declared for the ALU_out agents

parameters:

- **name: ALU_OUT_RESULT_WIDTH**
- **type: int**
- **value: '16'**
  - Parameter to define the width of the ALU result defined

```
 8       config_constraints: []
 9       config_vars: []
10
11       hdl_typedefs: []
12       hvl_typedefs: []
13
14       parameters:
15       - name: ALU_OUT_RESULT_WIDTH
16         type: int
17         value: '16'
18
```

out_ifc

Interface Config File
ALU_out_interface.yaml

ports:

- **dir: input**

  name: done

  width: '1'

  — Define names, directions and widths of all signals on the ALU output interface

NOTES:

— Direction specified here is in relation to the testbench
  - i.e. 'done' is an output from the ALU (DUT) and an input to the testbench

— The 'result' signal use the ALU_OUT_RESULT_WIDTH parameter which was defined in the parameter section of the YAML configuration file

```
19      ports:
20      - dir: input
21        name: done
22        width: '1'
23      - dir: input
24        name: result
25        width: ALU_OUT_RESULT_WIDTH
26
```

out_ifc

Interface Config File
ALU_out_interface.yaml

response_info:

data: []

operation: 1’b0

```
27      response_info:
28        data: []
29        operation: 1'b0
30
```

- For the ALU_out agent, we have no response data to be passed back to the sequence
- Leave with empty/default values.

out_ifc

Interface Config File
ALU_out_interface.yaml

transaction_contraints: []

transaction_vars:

- **iscompare:** 'True'
  israndd: 'False'
  name: result
  type: bit [ALU_OUT_RESULT_WIDTH-1:0]

```
30
31      transaction_constraints: []
32      transaction_vars:
33      - iscompare: 'True'
34        israndd: 'False'
35        name: result
36        type: bit [ALU_OUT_RESULT_WIDTH-1:0]
37
```

- The **ALU_out** agent has no transaction constraints

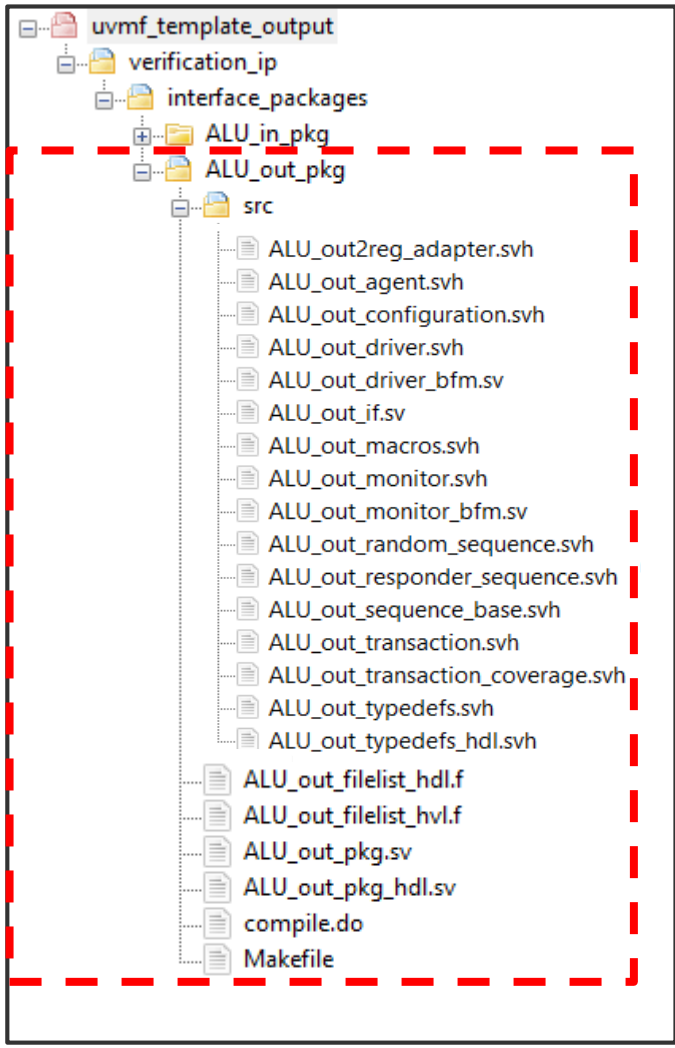
- The **ALU_out** agent has a transaction class which contains a single variable called 'result'. The width of the variable is defined by the agent parameter ALU_OUT_RESULT_WIDTH), which is set to 16 by default.
- Since we only monitor this interface, there is no need to randomize the transaction so we specify israndd: 'False'
- We will want to compare the result variable in the transaction class do_compare method so we specify iscompare: 'True'

out_ifc

Generated UVMF Code
ALU_out_if

- **Generating the Interface Code**
  - Execute the following command to generate the ***ALU_in*** agent code

python $UVMF_HOME/scripts/yaml2uvmf.py ALU_out_interface.yaml



  - All UVMF agent code is placed under **uvmf_template_output / verification_ip / interface_packages**
  - All generated code for the ALU_out agent will be saved under the **ALU_out_pkg** folder [as shown opposite]
  - There may be some additional meta-data files generated in the ALU_out_pkg directory. These can be ignored

out_ifc

ALU Environment Config File
ALU_environment.yaml

- UVMF Environment Packages
  - UVMF uses a separate environment level YAML configuration file to generate the environment level classes
  - The classes for the environment, its configuration, and sequence are included in the generated environment package
  - It can optionally include predictor and scoreboard classes
  - The environment package can be reused when a block level UVMF testbench is being used as part of a subsystem/chip level testbench.
- The environment config file ***(ALU_environment.yaml)*** is covered in more detail in the following slides

env

Environment Config File
ALU_environment.yaml

```
uvmf:
  environments:
    ALU:

      agents:
      - name: ALU_in_agent
        type:  ALU_in
      - name: ALU_out_agent
        type:  ALU_out
```

Tells UVMF code generator to create an environment with name 'ALU_env_pkg'

ALU_environment.yaml

```
1  uvmf:
2    environments:
3      ALU:
4
5        agents:
6        - name: ALU_in_agent
7          type: ALU_in
8        - name: ALU_out_agent
9          type: ALU_out
10
```

Tells UVMF code generator to include 1 x ALU_in agent and 1 x ALU_out agent.
-name defines the instance name
-type is the name of the agent given by the user in the interface YAML configuration file

env

Environment Config File
ALU_environment.yaml

```
analysis_components:
- name: ALU_pred
  type: ALU_predictor
```

```
10
11      analysis_components:
12      - name: ALU_pred
13        type: ALU_predictor
14      analysis_exports: []
15      analysis_ports: []
16
```

Tells UVMF code generator to include a component of type ALU_predictor with instance name ALU_pred.
This ALU_predictor component has not been defined yet and will be described in a separate YAML configuration file.

```
analysis_exports: []
analysis_ports: []
```

These allow the user to specify analysis exports & ports to add to the environment class, typically implemented when the block level environment is to be utilized within a larger system level UVM testbench.

We don't need to specify anything here for the ALU testbench.

env

Environment Config File
ALU_environment.yaml

config_constraints: []

config_vars: []

parameters: []

```
17      config_constraints: []
18      config_vars: []
19
20      parameters: []
21
```

These allow the user to specify environment level configuration variables, configuration constraints and parameters for the environment class.

Parameters specified here can be passed down into any of the instantiated agents or other analysis components.

We don’t need to specify anything here for the ALU testbench.

env

Environment Config File
ALU_environment.yaml

```
scoreboards:
- name: ALU_sb
  sb_type: uvmf_in_order_scoreboard
  trans_type: ALU_out_transaction

subenvs: []
```

```
22      scoreboards:
23      - name: ALU_sb
24        sb_type: uvmf_in_order_scoreboard
25        trans_type: ALU_out_transaction
26
27      subenvs: []
```

The *scoreboards* entry allow the user to specify any scoreboard components to be added the environment class.

Here we specify the following for ALU testbench

- Add a scoreboard component with instance name = ALU_sb
- The class type for the scoreboard = uvmf_in_order_scoreboard. This is a UVMF base library component
- We define the transaction class that the scoreboard will operate on to be ALU_out_transaction

The subenvs entry allows the user to import other pre-generated UVMF environments, thus creating a hierarchical environment.

Typically this is used when importing QVIP UVMF environments or creating a system level UVMF testbench that is reusing block level UVMF environments

We don't need to specify anything here for the ALU testbench

env

Environment Config File
ALU_environment.yaml

tlm_connections

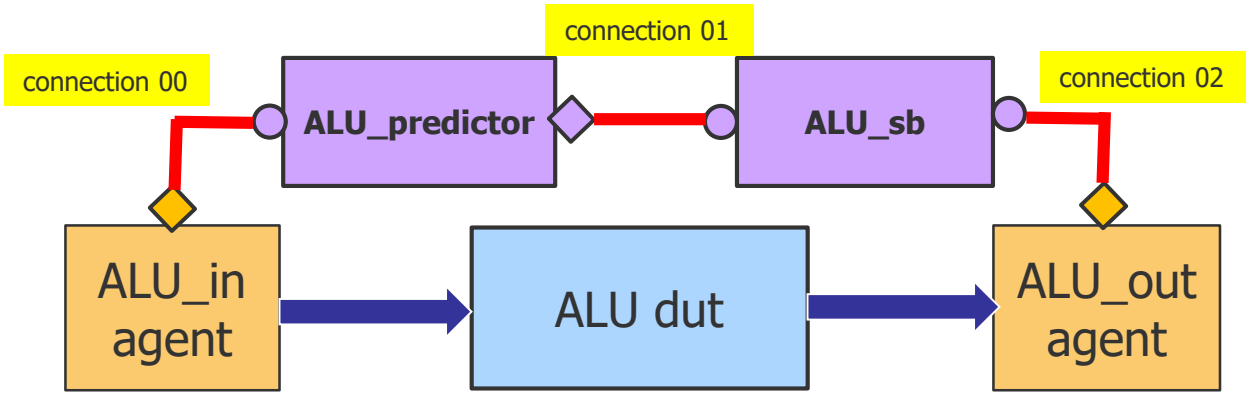
- driver: ALU_in_agent.monitored_ap
  receiver: ALU_pred.ALU_in_agent_ae

```
29      tlm_connections:
30      - driver: ALU_in_agent.monitored_ap          # connection 00
31        receiver: ALU_pred.ALU_in_agent_ae
32      - driver: ALU_pred.ALU_sb_ap                 # connection 01
33        receiver: ALU_sb.expected_analysis_export
34      - driver: ALU_out_agent.monitored_ap         # connection 02
35        receiver: ALU_sb.actual_analysis_export
```

The *tlm_connections* entry allows the user to specify a point to point connection between 2 ports/exports

The *driver* entry is the start point
The *receiver* entry is the end point

For the ALU testbench we specify 3 connections as shown in the diagram below



Environment Config File
ALU_environment.yaml

- TLM Connections : Details

```
- driver: ALU_in_agent.monitored_ap              # connection 00
  receiver: ALU_pred.ALU_in_agent_ae
```

ALU_in_agent : instance name of agent
monitored_ap : fixed name for analysis port on all UVMF agents
ALU_pred : instance name of ALU predictor
ALU_in_agent_ae : fixed name for predictor analysis export

```
- driver: ALU_pred.ALU_sb_ap                     # connection 01
  receiver: ALU_sb.expected_analysis_export
```

ALU_pred : instance name of predictor
ALU_sb_ap : fixed name for analysis port on scoreboard [‘_ap’ added to inst name]
ALU_sb : instance name of scoreboard
expected_analysis_export : fixed name for scoreboard ‘expected’ analysis export

env

Environment Config File
ALU_environment.yaml

- TLM Connections : Details (cont)

```
- driver: ALU_out_agent.monitored_ap          # connection 02
  receiver: ALU_sb.actual_analysis_export
```

ALU_out_agent : instance name of agent
monitored_ap : fixed name for analysis port on all UVMF agents
ALU_sb : instance name of ALU scoreboard
actual_analysis_export : fixed name for scoreboard 'actual' analysis export

env

Environment Config File
ALU_util_comp_alu_predictor.yaml

- The predictor to be used in our environment can optionally either be defined in the same YAML file as the environment or in a separate file. For the ALU testbench we shall use a separate file

```
uvmf:
    util_components:
      ALU_predictor:
        analysis_exports:
        - name: ALU_in_agent_ae
          type:   'ALU_in_transaction #()'
        analysis_ports:
        - name: ALU_sb_ap
          type:   'ALU_out_transaction #()'
      type: predictor
```

Tells UVMF code generator to create an analysis component with name 'ALU_predictor'

Tells UVMF code generator to create exports/ports with the specified names and specified transaction types

Currently the only supported util_component type is predictor. In future releases of UVMF this will be expanded to include coverage components

ALU_util_comp_alu_predictor.yaml

```
1  uvmf:
2    util_components:
3      ALU_predictor:
4        analysis_exports:
5        - name: ALU_in_agent_ae
6          type: 'ALU_in_transaction #()'
7        analysis_ports:
8        - name: ALU_sb_ap
9          type: 'ALU_out_transaction #()'
10       type: predictor
```

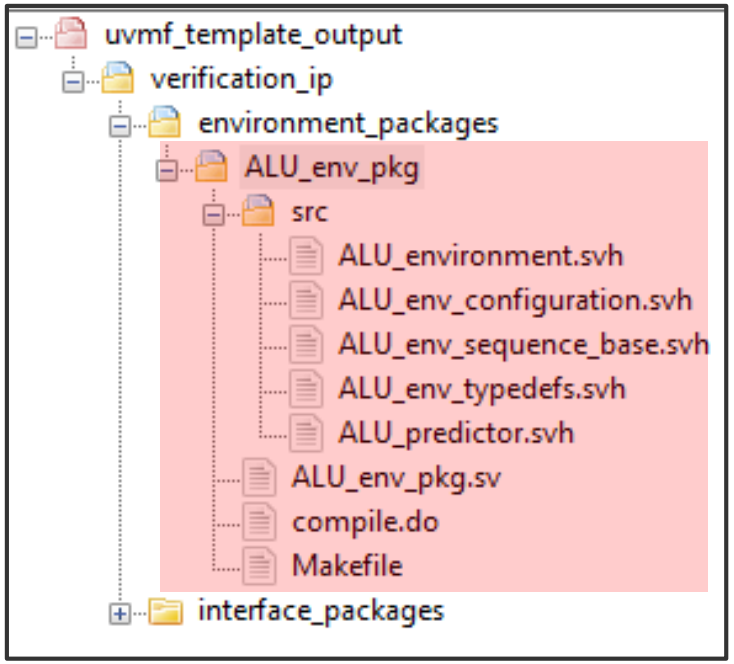

env

Generated UVMF Code
ALU_env_pkg

- **Generating the Environment Code**
  - In order to generate the environment level code, we need to specify the YAML configuration files for the environment, the predictor component and for each of the interfaces instantiated in that environment
  - Execute the following command to generate the ***ALU_in*** environment code

```
python $UVMF_HOME/scripts/yaml2uvmf.py \
        ALU_in_interface.yaml \
        ALU_out_interface.yaml \
        ALU_util_comp_alu_predictor.yaml \
        ALU_environment.yaml
```

  - You can create a simple .bat file on Windows to set the $UVMF_HOME environment variable and then call python on your YAML config files as shown below



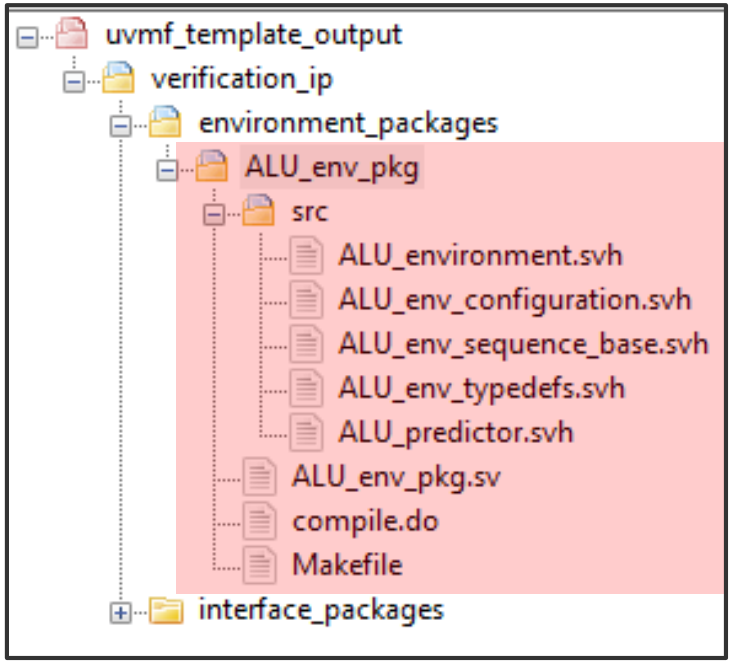

```
1  @set QUESTA_ROOT=C:/MentorTools/questasim_10.7c
2  @set UVMF_HOME=%QUESTA_ROOT%/examples/UVM_Framework/UVMF_3.6h
3
4  python %UVMF_HOME%/scripts/yaml2uvmf.py ALU_in_interface.yaml ALU_out_interface.yaml ALU_util_comp_alu_predictor.yaml
   ALU_environment.yaml
5
6  pause
```

env

Generated UVMF Code
ALU_env_pkg

- Files get generated under uvmf_template_output/verification_ip/environment_packages/ALU_env_pkg
- The key files are as follows:

- ALU_environment.svh
  - ALU environment class that instantiates the agents, scoreboards, predictors & connects them
- ALU_env_configuration.svh
  - Configuration class for 'ALU' environment
- ALU_env_sequence_base.svh
  - Base sequence class for any environment level sequences
- ALU_env_typedefs.svh
  - Contains any defines and typedefs to be compiled for use with the environment package
- ALU_predictor.svh
  - Generated predictor class for ALU environment.
  - User will need to add code to predictor model to implement the prediction function
- ALU_env_pkg.sv
  - ALU Environment package
- compile.do
  - Contains the compile commands for the generated environment package. Use on Windows or Linux
- Makefile
  - Contains the compile commands for the generated environment package. Use on Linux



env

ALU Bench Config File
ALU_bench.yaml

- UVMF Top Level Testbench
  - The bench config file will be used to generate the UVMF top level testbench
  - The top level testbench will instantiate the ALU env (which in turn instantiates the ALU interface agents as well as the environment configuration class
  - It facilitates the top-down configuration of the environment, which in turn configures the agents.
  - It provides a default sequences and a default test to run
  - It provides a simulation directory and makefile/run.do file for compiling and simulating the generated code
  - The code generated from the bench level config file is specific to the DUT it is testing and in general will be non-reusable code.

- Lets look at the bench config file ***(ALU_bench.yaml)*** in more detail

bench

ALU Bench Config File

ALU_bench.yaml

```
uvmf:
  benches:
    ALU:
```

Tells UVMF code generator to create a bench with name 'ALU'

```
      active_passive:
      - bfm_name: ALU_in_agent
        value: ACTIVE
      - bfm_name: ALU_out_agent
        value: PASSIVE
```

Tells UVMF code generator to include the specified agents & defines if the agents are ACTIVE or PASSIVE

For the ALU testbench, the ALU_in_agent will be generating stimulus and monitoring the signal interface, so set it ACTIVE.

The ALU_out_agent will not drive stimulus as it only monitors DUT output signals, so set it PASSIVE.

ALU_bench.yaml

```
1  uvmf:
2    benches:
3      ALU:
4
5        active_passive:
6        - bfm_name: ALU_in_agent
7          value: ACTIVE
8        - bfm_name: ALU_out_agent
9          value: PASSIVE
10
11       clock_half_period: 5ns
12       clock_phase_offset: 9ns
13
14       interface_params: []
15
16       reset_assertion_level: 'False'
17       reset_duration: 200ns
18
19       top_env: ALU
```

bench

ALU Bench Config File

ALU_bench.yaml

clock_half_period: 5ns
clock_phase_offset: 9ns

Defines the period and phase offset of the top level testbench clock

interface_params: []

Enables user to specify how any underlying BFMs should be parameterized. Not used for ALU testbench

reset_assertion_level: 'False'
reset_duration: 200ns

Defines the assertion level and duration of the top level testbench reset

top_env: ALU

The name of the top level environment to instantiate in this bench. For the ALU testbench this is the ALU environment

ALU_bench.yaml

```
1  uvmf:
2    benches:
3      ALU:
4
5        active_passive:
6        - bfm_name: ALU_in_agent
7          value: ACTIVE
8        - bfm_name: ALU_out_agent
9          value: PASSIVE
10
11       clock_half_period: 5ns
12       clock_phase_offset: 9ns
13
14       interface_params: []
15
16       reset_assertion_level: 'False'
17       reset_duration: 200ns
18
19       top_env: ALU
```

bench

Generated UVMF Code
ALU testbench

- **Generating the Testbench Code**
  - In order to generate the top level testbench code, we need to specify the YAML configuration file for the bench and for each of the environments, predictors & interfaces instantiated in that bench
  - Execute the following command to generate the ***ALU*** bench code

```
python $UVMF_HOME/scripts/yaml2uvmf.py \
        ALU_in_interface.yaml \
        ALU_out_interface.yaml \
        ALU_util_comp_alu_predictor.yaml \
        ALU_environment.yaml \
        ALU_bench.yaml
```

  - You can create a simple .bat file on Windows to set the $UVMF_HOME environment variable and then call python on your YAML config files as shown below

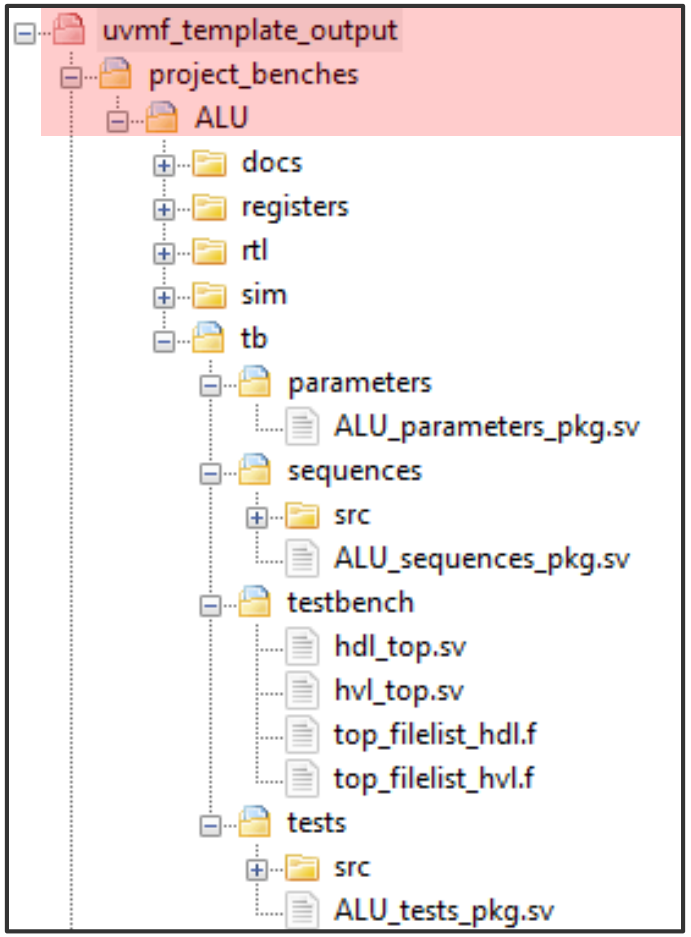
```
1  @set QUESTA_ROOT=C:/MentorTools/questasim_10.7c
2  @set UVMF_HOME=%QUESTA_ROOT%/examples/UVM_Framework/UVMF_3.6h
3
4  python %UVMF_HOME%/scripts/yaml2uvmf.py ALU_in_interface.yaml ALU_out_interface.yaml ALU_util_comp_alu_predictor.yaml ALU_environment.yaml ALU_bench.yaml
5
6  pause
```

bench

Generated UVMF Code
project_benches/ALU

- Generates the top level UVMF testbench plus scripts for compiling and running the simulation
- Files generated under

  uvmf_template_output/project_benches/ALU



bench

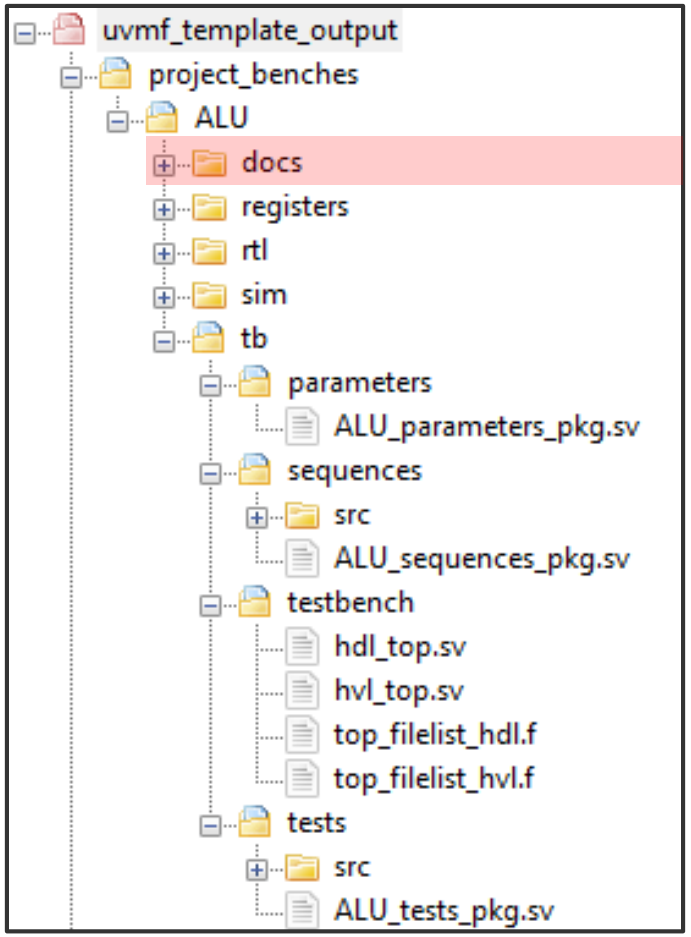
Generated UVMF Code
project_benches/ALU

- Files generated under

  uvmf_template_output/project_benches/ALU

  - **docs**

    — Placeholder folder for user to place documentation



bench

Mentor A Siemens Business

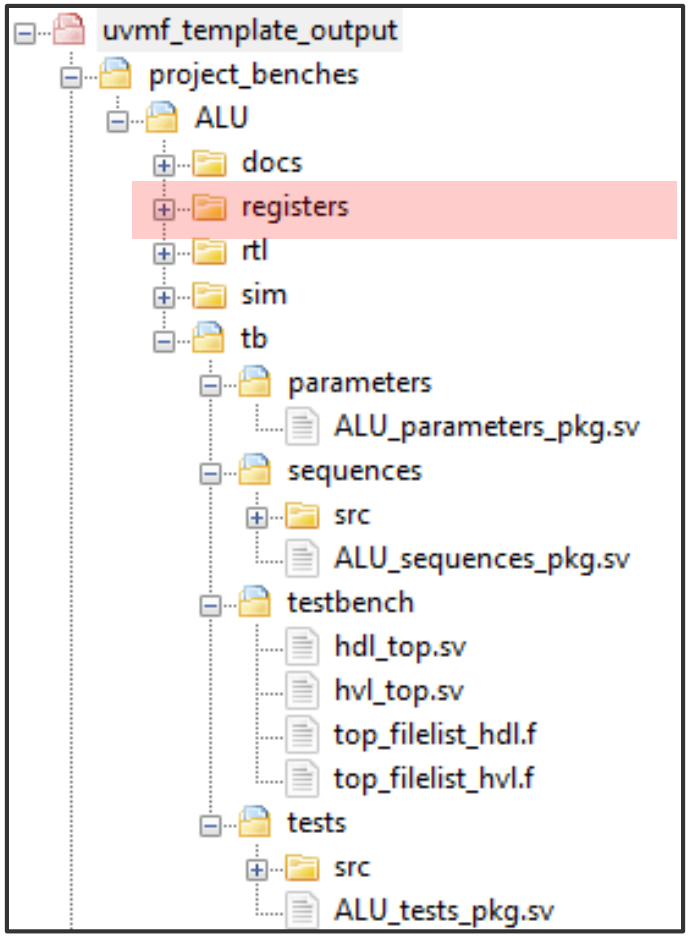
Generated UVMF Code
project_benches/ALU

- Files generated under

  uvmf_template_output/project_benches/ALU

  - **registers**

    — Placeholder folder for user to place register layer package



bench

Generated UVMF Code
project_benches/ALU

- Files generated under

uvmf_template_output/project_benches/ALU

- **rtl**

— Placeholder folder for user to place RTL DUT code

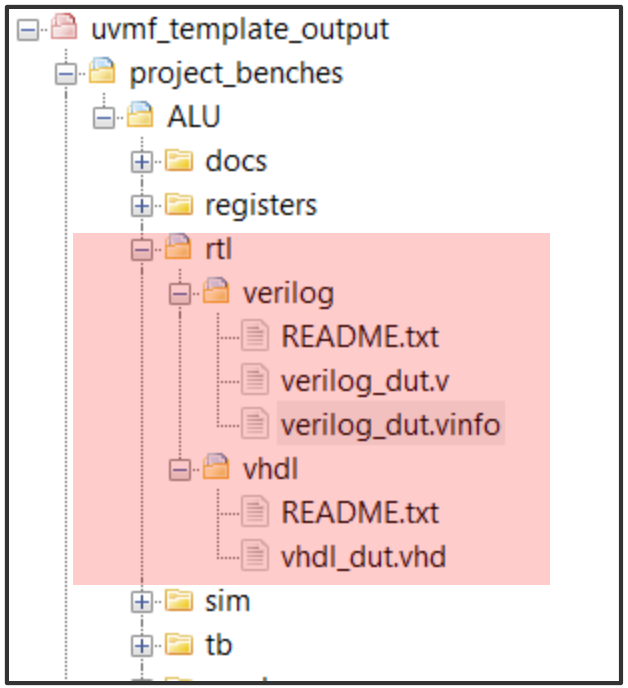
NOTE:
This is an optional location to place the DUT code.
The user can place their DUT code anywhere they want.

- **rtl/verilog/verilog_dut.v**

— Dummy Verilog DUT code created by the code generator and instantiated in hdl_top

- **rtl/vhdl/vhdl_dut.v**

— Dummy VHDL DUT code created by the code generator and instantiated in hdl_top



bench

Generated UVMF Code
project_benches/ALU

- Files generated under

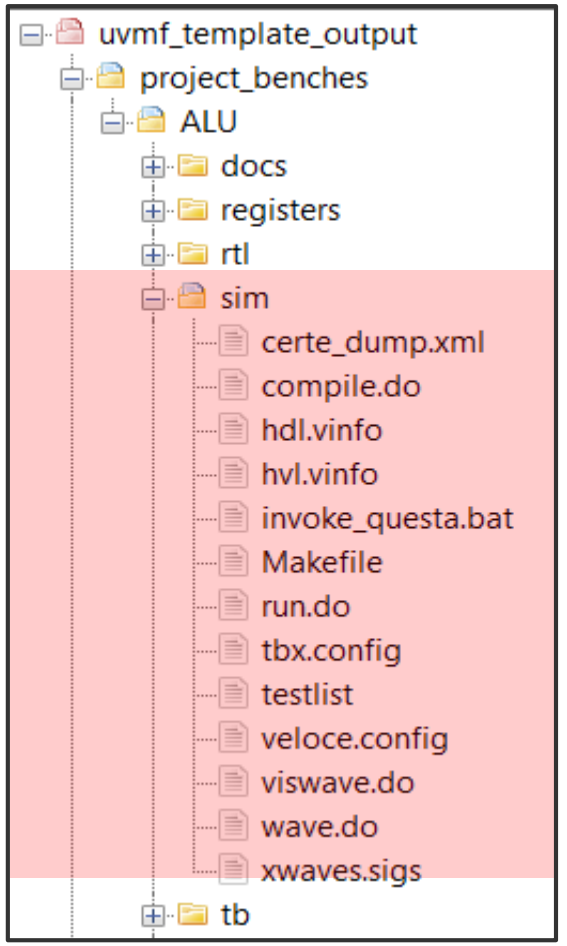
  uvmf_template_output/project_benches/ALU

  - **sim**

    — Directory where user should run simulations

    — Contains *invoke_questa.bat* script for windows users to compile & load the simulation
    — Contains *Makefile* for Linux users to compile & run testbench
    — Contains *compile.do* for Windows users to compile the testbench
    — Contains *run.do* for Windows users to run the testbench
    — Contains *wave.do* which is populated with agent transactions & interface signals

    — Also contain some other support files for emulation users plus a testlist for Questa VRM users.



bench

Mentor A Siemens Business

Generated UVMF Code
project_benches/ALU

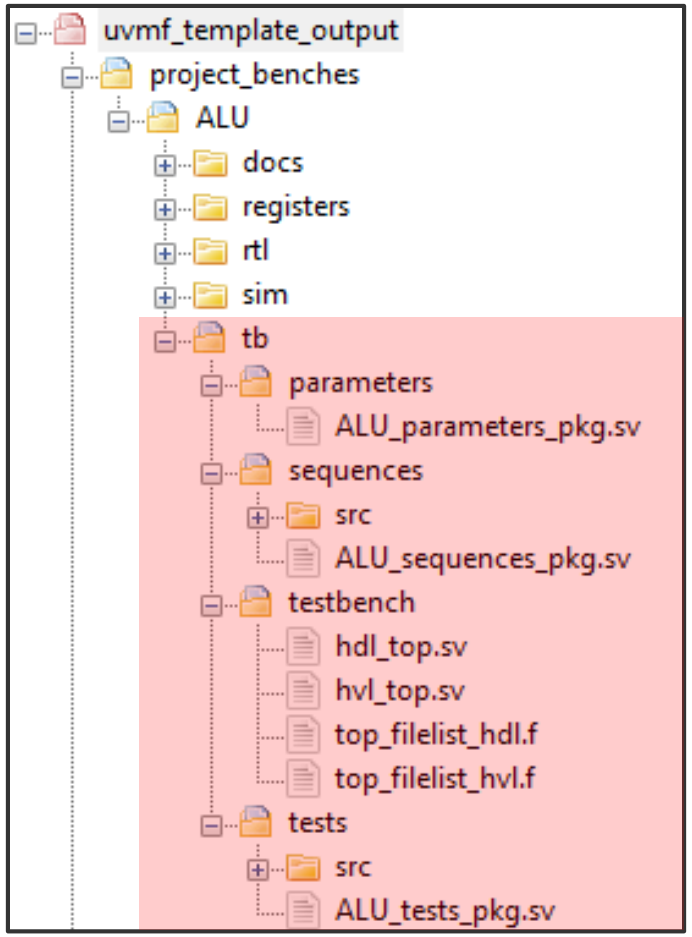
- Files generated under

uvmf_template_output/project_benches/ALU

- **tb**

— Multiple sub-folders for testbench

- **Parameters**
  - — Top level testbench params package (interface names, etc)
- **Sequences**
  - — top level sequence and a sequence base class
  - — register sequence template class
  - — Sequence package
- **testbench**
  - — hdl_top.sv : top level module based TB
  - — hvl_top.sv : non-synthesizable parts of top level TB
- **tests**
  - — default test (test_top), extended from test base class
  - — Derived test (extended from test_top) template class
  - — register test template class
  - — test package



bench

Mentor A Siemens Business

Agenda

- Introduction
- ALU Overview
- Config Files Explained
- Compile and Simulate Generated Code
- Adding DUT Specific Functionality

sim

Out Of The Box Simulation
project_benches/ALU/sim

- OS Considerations

— Linux
  - make debug : compiles and loads the testbench in the Questa GUI (interactive simulation)
  - make build : only compiles the generated code
  - make run_gui : invokes interactive simulation without recompiling the code
  - make run_cli : invokes command line simulation without recompiling the code

— Windows
  - do compile.do : compiles the generated testbench code
  - do run.do : loads the testbench in Questa
  - invoke_questa.bat : invokes Questa and executes the compile.do and run.do TCL files to compiled and load the simulation

— Makefile sets default OS architecture to 32 bit
  - If running on the 64 bit version of Questa on Linux, you can change the OS setting to 64 bit as follows;

```
make build MACHINE_ARCH='-64'
make debug MACHINE_ARCH='-64'
```


sim

Out Of The Box Simulation
project_benches/ALU/sim

- Bench level config file
  - Generated Makefile / run.do invokes vsim with +UVM_TESTNAME=test_top
  - Also applies several other switches to vsim that are required to run the UVMF simulation
    - ***Example vsim command from the run.do script is shown below***
    - ***We recommend that you do not remove any of the switches being applied***

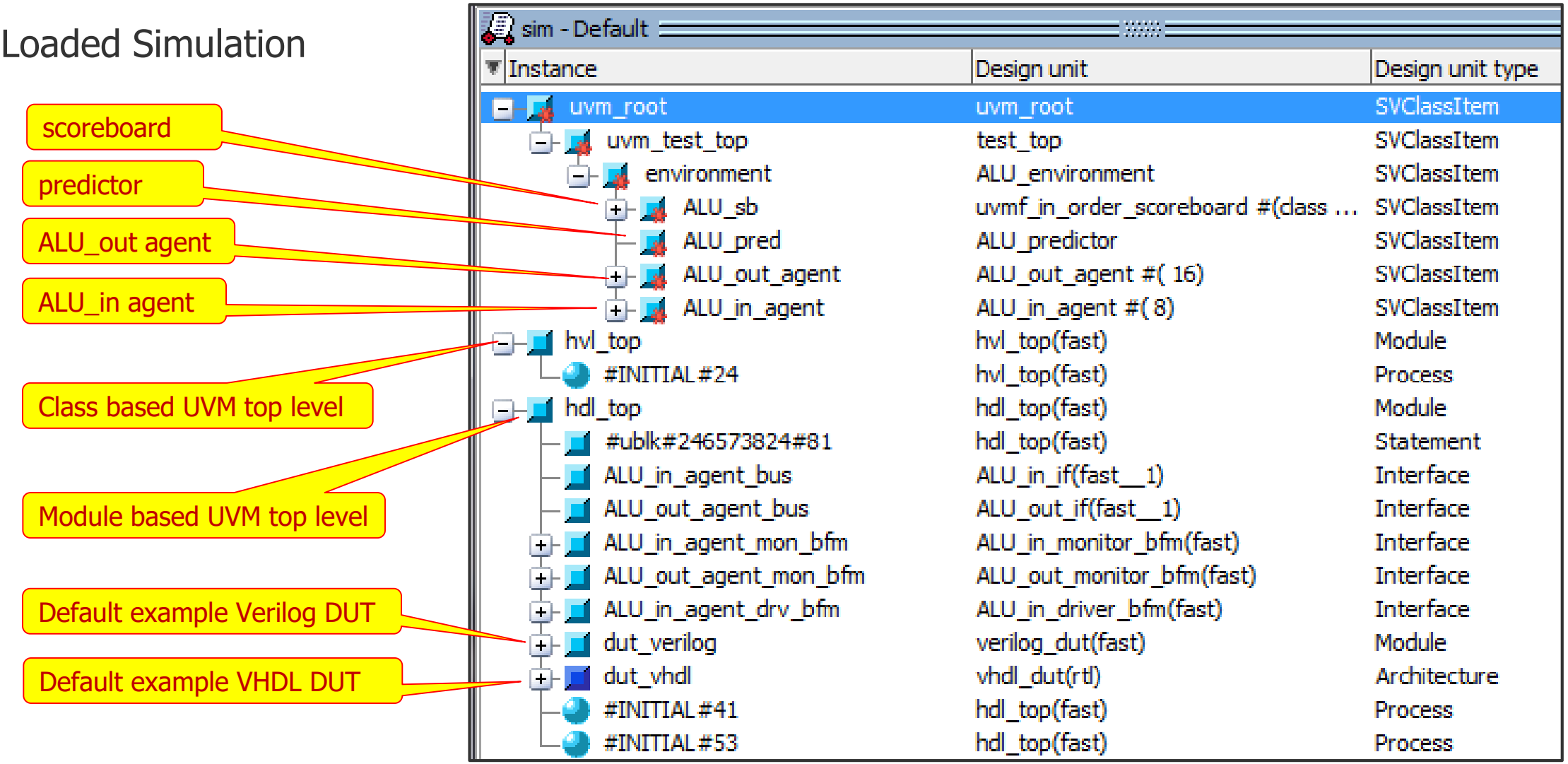
```
vsim -i  -32  -sv_seed random +UVM_TESTNAME=test_top \
              +UVM_VERBOSITY=UVM_HIGH  \
              -permit_unmatched_virtual_intf +notimingchecks \
              -suppress 8887 -uvmcontrol=all -msgmode both \
              -classdebug –assertdebug \
              +uvm_set_config_int=*,enable_transaction_viewing,1 \
              -do " set NoQuitOnFinish 1; onbreak {resume}; run 0; \
                    do wave.do; set PrefSource(OpenOnBreak) 0; \
                    radix hex showbase; " optimized_debug_top_tb
```

sim

Out Of The Box Simulation

project_benches/ALU/sim

- Loaded Simulation



sim

Out Of The Box Simulation
project_benches/ALU/sim

- Wave Window : Auto populated with UVMF agent interface signals and transactions
  - Screen shot shows wave window after running simulation for 500 ns

Transactions generated from the UVMF agent monitors
- Just shows default values for transaction data members. DUT not driving any values yet.

sim

Out Of The Box Simulation

project_benches/ALU/tb/tests/src

- Default test : top_test.svh
  - Extends from uvmf_test_base
  - Is parameterized with the configuration, environment and sequence to use
  - Build phase kicks off top down configuration

```
24 typedef ALU_env_configuration ALU_env_configuration_t;
25 typedef ALU_environment ALU_environment_t;
26
27 class test_top extends uvmf_test_base #(.CONFIG_T(ALU_env_configuration_t),
28                                          .ENV_T(ALU_environment_t),
29                                          .TOP_LEVEL_SEQ_T(ALU_bench_sequence_base));
30
31   `uvm_component_utils( test_top );
32
33 string interface_names[] = {
34     ALU_in_agent_BFM /* ALU_in_agent       [0] */ ,
35     ALU_out_agent_BFM /* ALU_out_agent     [1] */
36 };
37
38 uvmf_active_passive_t interface_activities[] = {
39     ACTIVE /* ALU_in_agent      [0] */ ,
40     PASSIVE /* ALU_out_agent     [1] */
41 };
42
43 // ****************************************************************************
44 // FUNCTION: new()
45   function new( string name = "", uvm_component parent = null );
46     super.new( name ,parent );
47   endfunction
48
49 // ****************************************************************************
50 // FUNCTION: build_phase()
51   virtual function void build_phase(uvm_phase phase);
52
53     super.build_phase(phase);
54     configuration.initialize(BLOCK, "uvm_test_top.environment", interface_names, null, interface_activities);
55   endfunction
56
57 endclass
```

This is the default top level virtual sequence that gets executed

sim

Out Of The Box Simulation

project_benches/ALU/tb/sequences/src

- Default virtual sequence : ALU_bench_sequence_base.svh

1. Construct ALU_in agent sequence
2. Both agents held until reset is de-asserted
3. Placeholder to start a responder sequence
4. Starts agent random sequence and repeats 25 times. (Only the ALU_in agent was set ACTIVE, so only one sequence executed here)
5. Wait for 400 clock cycles after sequences have finished to flush and data from DUT

```
82  virtual task body();
83
84  // Construct sequences here                                                    1
85   ALU_in_agent_random_seq     = ALU_in_agent_random_seq_t::type_id::create("ALU_in_agent_random_seq");
86
87   fork
88    ALU_in_agent_config.wait_for_reset();                                       2
89    ALU_out_agent_config.wait_for_reset();
90   join
91
92  // Start RESPONDER sequences here
93   fork                                                                        3
94   join_none
95
96  // Start INITIATOR sequences here
97   fork
98       repeat (25) ALU_in_agent_random_seq.start(ALU_in_agent_sequencer);      4
99   join
100
101  // UVMF_CHANGE_ME : Extend the simulation XXX number of clocks after
102  // the last sequence to allow for the last sequence item to flow
103  // through the design.
104
105 fork
106   ALU_in_agent_config.wait_for_num_clocks(400);                               5
107   ALU_out_agent_config.wait_for_num_clocks(400);
108 join
109
110 endtask
```

sim

Out Of The Box Simulation

project_benches/ALU/tb/sequences/src

- Default sequence : ALU_bench_sequence_base.svh
  - Sequence body (shown on previous slide)
    - creates agent sequences
      - The ALU_out agent is passive so it has no default sequence to run
    - Repeats each agent sequence to run 25 times
      - The default random sequence randomizes and generates 1 transaction.
    - Uses utility methods in agent configs to wait for specified number of clocks
  - Sequence code (not shown)
    - Extends from uvmf_sequence_base
    - Defines sequence handles for to run on each active agent.
    - Gets the config handles for each agent from the UVM config DB
    - Get the sequencer handles for each agent from the UVM config DB

sim

Agenda

- Introduction
- ALU Overview
- Config Files Explained
- Compile and Simulate Generated Code
- Adding DUT Specific Functionality

DUT-code

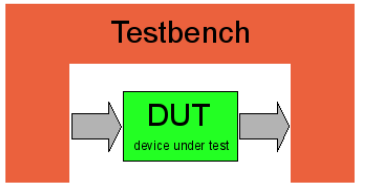
Completing the UVMF Testbench

- User modifications to the UVMF Generated Code
  - You generated the UVMF testbench from bench level YAML configuration file. Now you need to add DUT specific code into some of the generated files.
  - These modification steps include
    1. Adding the DUT & wiring it up to the BFMs and the clock/reset
    2. Adding protocol specific information to the driver BFMs
    3. Adding protocol specific information to the monitor BFMs
    4. Adding DUT specific behavior to the predictor
  - You will then need to create additional tests & sequences to exercise the DUT functionality, which requires the following steps
    1. Extending the default test to create a new test which overrides the default sequence
    2. Extending the default sequence to create a new sequence that generates the desired stimulus for the test.
- The following slides will look at the code changes required to implement each of the above steps

DUT-code

Instantiate & Wire Up the DUT

project_benches/ALU/tb/testbench/hdl_top.sv



- Clocks & Resets
  - The hdl_top module contains simple clock and reset generation code that the user can modify to change frequencies, add more clocks, etc depending on the need for their specific DUT
  - The clock frequency, clock offset, reset polarity & reset duration were specified from the bench level YAML configuration file

```
11      clock_half_period: 5ns
12      clock_phase_offset: 9ns
13
14      interface_params: []
15
16      reset_assertion_level: 'False'
17      reset_duration: 200ns
```

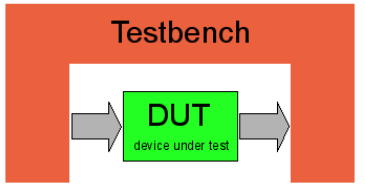
YAML ALU BENCH CONFIG

```
34 module hdl_top;
35 // pragma attribute hdl_top partition_module_xrtl
36
37
38 bit clk;
39    // Instantiate a clk driver
40    // tbx clkgen
41    initial begin
42       clk = 0;
43       #9ns;
44       forever begin
45          clk = ~clk;
46          #5ns;
47       end
48    end
49
50 bit rst;
51    // Instantiate a rst driver
52    // tbx clkgen
53    initial begin
54       rst = 0;
55       #200ns;
56       rst =  1;
57    end
```

  - In the case of the ALU IP we can leave this code unmodified
  - For other DUTs you may need to modify this code to generate different frequency clocks or opposite polarity resets

DUT-code

Instantiate & Wire Up the DUT
project_benches/ALU/tb/testbench/hdl_top.sv



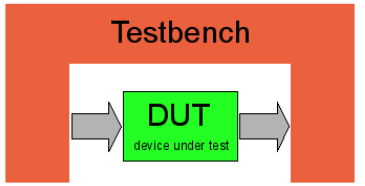
- The DUT
  - The DUT RTL model is ***alu.v***
  - This file is located at the top level of the tutorial directory.
  - You can copy this file into the corresponding rtl folder under your ***uvmf_template_output/project_benches/ALU/rtl/verilog*** which was created when you ran your python config files

NOTE: You do not have to place the DUT RTL code in this this directory. This is merely a placeholder location for DUT source code but it can reside anywhere on disk.

DUT-code

Instantiate & Wire Up the DUT

project_benches/ALU/tb/testbench/hdl_top.sv



- Instantiate the DUT
  - Edit the file ***project_benches/ALU/tb/testbench/hdl_top.sv***
  - Find the comment that says "Instantiate DUT here"
  - Remove the instantiations of 'dut_verilog' & 'dut_vhdl'
  - Add instance of the ALU and wire up ports to the corresponding agent interface

Original Code

```
75 // UVMF_CHANGE_ME : Add DUT and connect to signals in _bus interfaces listed above
76 // Instantiate your DUT here
77 // These DUT's instantiated to show verilog and vhdl instantiation
78 verilog_dut         dut_verilog(.clk(clk), .rst(rst), .in_signal(vhdl_to_verilog_signal), .out_signal(verilog_to_vhdl_signal));
79 \work.vhdl_dut(rtl) dut_vhdl(    .clk(clk), .rst(rst), .in_signal(verilog_to_vhdl_signal), .out_signal(vhdl_to_verilog_signal));
80
```

Modified Code

```
74
75 // UVMF_CHANGE_ME : Add DUT and connect to signals
76 // Instantiate your DUT here
77   alu   #(.OP_WIDTH(8), .RESULT_WIDTH(16)) DUT  (
78       // ALU connections
79       .clk    (ALU_in_agent_bus.clk ) ,
80       .rst    (ALU_in_agent_bus.alu_rst ) ,
81       .ready  (ALU_in_agent_bus.ready ) ,
82       .valid  (ALU_in_agent_bus.valid ) ,
83       .op     (ALU_in_agent_bus.op ) ,
84       .a      (ALU_in_agent_bus.a ) ,
85       .b      (ALU_in_agent_bus.b ) ,
86       .done   (ALU_out_agent_bus.done ) ,
87       .result (ALU_out_agent_bus.result ) );
```

NOTES:

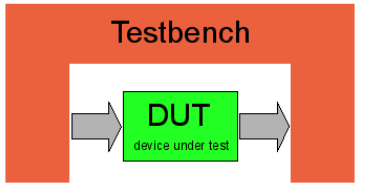
The following files have define the Systemverilog interface signals:

- uvmf_template_output/verification_ip/interfaces/ALU_in_pkg/src/ALU_in_if.sv
- uvmf_template_output/verification_ip/interfaces/ALU_out_pkg/src/ALU_out_if.sv

The testbench power up reset signal 'rst' is passed in to the agent BFM interfaces. However, the ALU 'rst' pin needs to be driven active either when the power up reset is active or when a RST_OP transaction is generated by a UVM test/sequence. The ALU 'rst' is therefore be driven by the ALU_in agent. You will modify the ALU_in driver BFM file to implement this in a later step.

DUT-code

Instantiate & Wire Up the DUT
project_benches/ALU/tb/testbench/hdl_top.sv



- Compiling The DUT
  - Go to the folder ***project_benches/ALU/sim***
  - There is a compile.do for Windows customers and a Makefile for Linux users.

  - **compile.do** :
    - Remove the compilation lines for the default verilog_dut.v & vhdl_dut.vhd

      Original

```
34 vlog -sv -timescale 1ps/1ps -suppress 2223,2286 $env(UVMF_PROJECT_DIR)/rtl/verilog/verilog_dut.v
35 vcom $env(UVMF_PROJECT_DIR)/rtl/vhdl/vhdl_dut.vhd
```

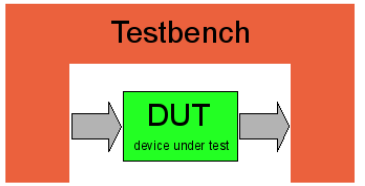
    - Replace with the vlog command to compile the ALU.v source file

      Modified

```
34 vlog -sv -timescale 1ps/1ps -suppress 2223,2286 $env(UVMF_PROJECT_DIR)/rtl/verilog/alu.v
35
```

DUT-code

Instantiate & Wire Up the DUT
project_benches/ALU/tb/testbench/hdl_top.sv



- Compiling The DUT
  - Go to the folder ***project_benches/ALU/sim***
  - There is a compile.do for Windows customers and a Makefile for Linux users.
  - **Makefile**
    - Modify the source file list from the default verilog_dut.sv to use the alu.v source file

Original

```
113 # UVMF_CHANGE_ME : Reference Verilog DUT source.
114 ALU_VERILOG_DUT =\
115 $(UVMF_PROJECT_DIR)/rtl/verilog/verilog_dut.v
```

Modified

```
113 # UVMF_CHANGE_ME : Reference Verilog DUT source.
114 ALU_VERILOG_DUT =\
115 $(UVMF_PROJECT_DIR)/rtl/verilog/alu.v
```

    - Modify the comp_ALU_dut target to only now compile up a Verilog DUT

Original

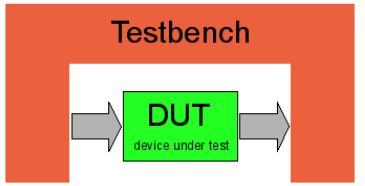
```
152 # UVMF_CHANGE_ME : Add make target to compile your dut here
153 comp_ALU_dut: comp_ALU_verilog_dut comp_ALU_vhdl_dut
154
```

Modified

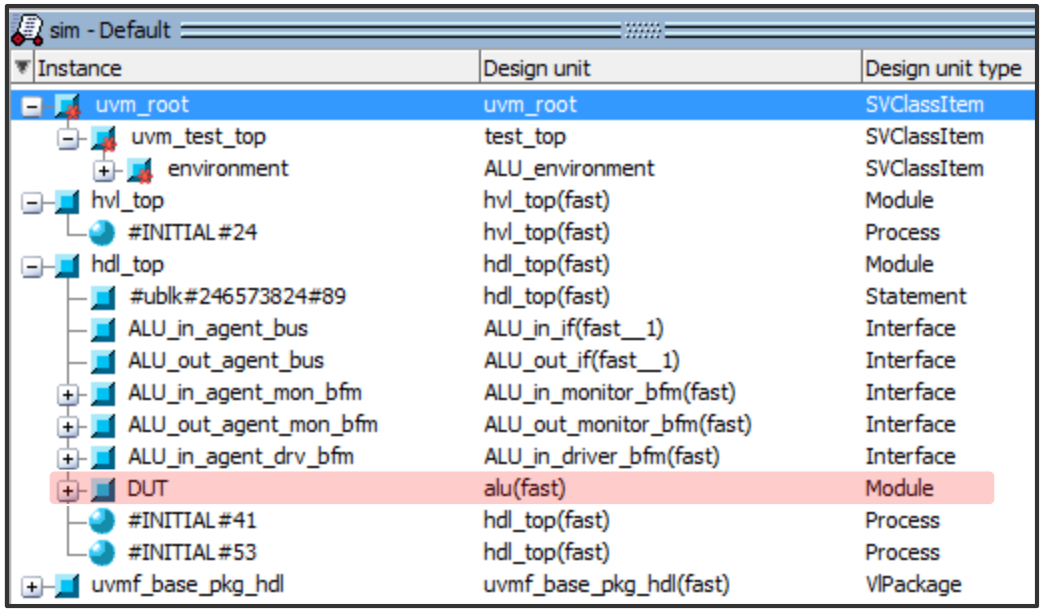
```
152 # UVMF_CHANGE_ME : Add make target to compile your dut here
153 comp_ALU_dut: comp_ALU_verilog_dut
154
```

DUT-code

Instantiate & Wire Up the DUT
project_benches/ALU/tb/testbench/hdl_top.sv



- Simulating with the ALU DUT
  - Go to the folder ***project_benches/ALU/sim***
  - LINUX
    - Use the Makefile (make build) to compile the testbench
    - Check that there are no compile errors
    - Use the Makefile (make debug) to load the simulation
    - Check that the ALU (instance name DUT) appears in the hierarchy of hdl_top
  - Windows
    - Use the invoke_questa.bat script to compile & load the testbench
    - Check that there are no compile errors
    - Check that the ALU (instance name DUT) appears in the hierarchy of hdl_top



DUT-code

Adding Protocol Information To The Driver BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_driver_bfm.sv

- Modifying the ALU_in driver BFM
  - Go to the folder ***verification_ip/interface_packages/ALU_in_pkg/src***
  - Edit the file ***ALU_in_driver_bfm.sv*** and locate the '***initiate_and_get_response***' task
  - By default the UVMF generator just has 4 consecutive clock delays inserted in to the driver. No data is actually driven onto the ALU_in bus interface
  - This code needs to be modified to implement the interface protocol

Original Code

```
task initiate_and_get_response(
   // This argument passes transaction variables used by an initiator
   // to perform the initial part of a protocol transfer.  The values
   // come from a sequence item created in a sequence.
   input ALU_in_initiator_s ALU_in_initiator_struct,
   // This argument is used to send data received from the responder
   // back to the sequence item.  The sequence item is returned to the sequence.
   output ALU_in_responder_s ALU_in_responder_struct
   );// pragma tbx xtf
```

```
   // Initiate a transfer using the data received.
   @(posedge clk_i);
   @(posedge clk_i);
   // Wait for the responder to complete the transfer then place the responder data into
   // ALU_in_responder_struct.
   @(posedge clk_i);
   @(posedge clk_i);
  endtask
// pragma uvmf custom initiate_and_get_response end
```

DUT-code

Adding Protocol Information To The Driver BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_driver_bfm.sv

- Modifying the ALU_in driver BFM
  - Replace the 4 consecutive clock cycle delays with the following code

```
234     @(posedge clk_i);
235     $display("alu_in_driver_bfm : Inside knitiate_and_get_response");
236     case (ALU_in_initiator_struct.op)
237       rst_op  : do_assert_rst(ALU_in_initiator_struct.op);
238       default : alu_in_op(ALU_in_initiator_struct.op, ALU_in_initiator_struct.a, ALU_in_initiator_struct.b);
239     endcase
240   endtask
```

Modified Code

NOTES:
The reset op code only drives the ALU reset pin and is therefore handled separately in it's own task.

  - Add the following 2 tasks to the module

```
242   // ***************************************************
243   task do_assert_rst(input alu_in_op_t op);
244   $display("%d ***************   Starting Reset", $time);
245      op_o <= op;
246      alu_rst_o <= 1'b0;
247      repeat (10) @(posedge clk_i);
248      alu_rst_o <= 1'b1;
249      repeat (5) @(posedge clk_i);
250   $display("%d ***************   Ending Reset", $time);
251   endtask
```

New Code

NOTES:
Any control signals like the ALU reset & valid must be driven at all times
Any data signals like the ALU operator and operands are only driven for specific cycles and then set back to 'Z' values

```
253 // ***************************************************
254   task alu_in_op(input alu_in_op_t op,
255                  input bit [ALU_IN_OP_WIDTH-1:0] a,
256                  input bit [ALU_IN_OP_WIDTH-1:0] b);
257
258     alu_rst_o <= 1'b1;
259     while ( ready_i == 1'b0 ) @(posedge clk_i) ;
260     valid_o <= 1'b1;
261     op_o <= op;
262     a_o <= a;
263     b_o <= b;
264
265     @(posedge clk_i);
266     valid_o <= 1'b0;
267     op_o <= {3{1'bz}};
268     a_o <= {ALU_IN_OP_WIDTH{1'bz}};
269     b_o <= {ALU_IN_OP_WIDTH{1'bz}};
270
271   endtask
```

New Code

DUT-code

Adding Protocol Information To The Driver BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_driver_bfm.sv

- Modifying the ALU_in driver BFM
  - Modify the code that generates the ALU_rst_o signal as shown below

```
102   // These are signals marked as 'output' by the config file, but the outputs will
103   // not be driven by this BFM unless placed in INITIATOR mode.
104   assign bus.alu_rst = (initiator_responder == INITIATOR) ? alu_rst_o : 'bz;
105   assign alu_rst_i = bus.alu_rst;
```

Original Code

```
102   // These are signals marked as 'output' by the config file, but the outputs will
103   // not be driven by this BFM unless placed in INITIATOR mode.
104   assign bus.alu_rst = (initiator_responder == INITIATOR) ? (alu_rst_o && rst_i) : 'bz;
105   assign alu_rst_i = bus.alu_rst;
```

Modified Code

NOTES:

- We want to reset the ALU if either the top level reset (rst_i) is active or when a RST_OP operation is received (which drives alu_rst_o low). So we logically AND the 2 reset driving signals together.

DUT-code

Adding Protocol Information To The Driver BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_driver_bfm.sv

- Modifying the ALU_in driver BFM
  - The driver outputs are declared as type 'reg' & given an initial value of 'z'

```
82   // INITIATOR mode output signals
83   tri  alu_rst_i;
84   reg  alu_rst_o = 'bz;
85   tri  valid_i;
86   reg  valid_o = 'bz;
87   tri [2:0] op_i;
88   reg [2:0] op_o = 'bz;
89   tri [ALU_IN_OP_WIDTH-1:0] a_i;
90   reg [ALU_IN_OP_WIDTH-1:0] a_o = 'bz;
91   tri [ALU_IN_OP_WIDTH-1:0] b_i;
92   reg [ALU_IN_OP_WIDTH-1:0] b_o = 'bz;
```

  - For any DUT controls that need to be at a defined '0' or '1' value, then we need to add code to reset them

```
160  // pragma uvmf custom interface_item_additional begin
161    always@(negedge rst_i)
162    begin
163          alu_rst_o <= 1'b0;
164          valid_o   <= 1'b0;
165    end
166
167    always@(posedge rst_i)
168    begin
169          alu_rst_o <= 1'b1;
170    end
171  // pragma uvmf custom interface_item_additional end
```

Additional Code

NOTES:

- We add an always block inside the driver bfm that is sensitive to the reset to procedurally assign the storage signals that are continuously driven onto the pins based on INITIATOR/RESPONDER setting.
- For the ALU we set the valid_o and alu_rst_o signals to 0 when reset is activated
- For the ALU we have to also add an always block to set alu_rst_0 when the external reset is removed.

DUT-code

Adding Protocol Information To The Driver BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_driver_bfm.sv

- Checking the driver BFM code changes
  - Use the compile.do/run.do or the Makefile (make debug) to check that there are no compilation errors in the code you have modified/added.

1. If you look at the ALU signals (ALU_in_agent_bus & ALU_out_agent_bus) you will see that the testbench is sending operations to the ALU and that results are being generated
2. The transactions are still showing incorrect value since the monitor code has not been modified yet. You will fix this next.

DUT-code

Adding Protocol Information To The Monitor BFM

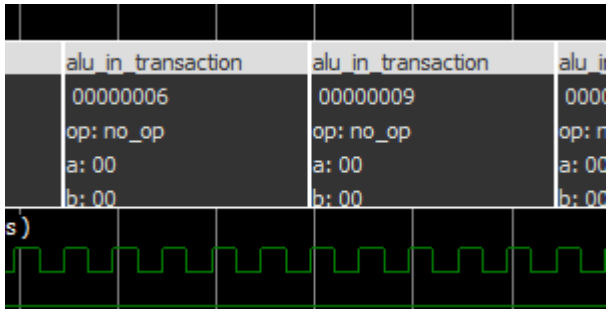
verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_monitor_bfm.sv

- Modifying the ALU_in monitor BFM
  - Go to the folder ***verification_ip/interface_packages/ALU_in_pkg/src***
  - Edit the file ***ALU_in_monitor_bfm.sv*** and locate the '***do_monitor***' task
  - By default the UVMF generator just has 4 consecutive clock delays inserted in to the monitor. No data is actually read from the ALU_in bus interface
  - This code needs to be modified to implement the interface protocol

Original Code

```
138  task do_monitor(output ALU_in_monitor_s ALU_in_monitor_struct);
139    // UVMF_CHANGE_ME : Implement protocol monitoring.  The commented reference code
       ✂
165    @(posedge clk_i);
166    @(posedge clk_i);
167    @(posedge clk_i);
168    @(posedge clk_i);
169    // pragma uvmf custom do_monitor end
170  endtask
```

NOTES
This is why we see the ALU_in_transactions are all 4 cycles long and the displayed data values are just the language type defaults



DUT-code

Adding Protocol Information To The Monitor BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_monitor_bfm.sv

- Modifying the ALU_in monitor BFM
  - Replace the 4 consecutive clock cycle delays with the following code

Modified Code

```
138  task do_monitor(
139          output ALU_in_monitor_s ALU_in_monitor_struct);
140    // UVMF_CHANGE_ME : Implement protocol monitoring.

                    ✂

141    // Hold here until signal event happens to capture bus values
166      while (valid_i == 1'b0 && alu_rst_i == 1'b1) begin
167        @(posedge clk_i);
168      end
169      ALU_in_monitor_struct.op = alu_in_op_t'(op_i);
170      ALU_in_monitor_struct.a  = a_i;
171      ALU_in_monitor_struct.b  = b_i;
172
173      if (alu_rst_i == 1'b0) begin
174        while (alu_rst_i == 1'b0) begin
175          @(posedge clk_i);
176          ALU_in_monitor_struct.op  = rst_op;
177        end
178      end
179
180  endtask
```

NOTES
- Wait until either valid goes high or reset goes active
- Read bus values
- If reset active then wait until reset becomes inactive, then assign RST_OP code

DUT-code

Adding Protocol Information To The Monitor BFM

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_monitor_bfm.sv

- Checking the monitor BFM code changes
  - Use invoke_questa.bat or the Makefile (make debug) to check that there are no compilation errors in the code you have modified/added.

1. The ALU_in transactions are now showing the actual stimulus data values and the transaction lengths match the corresponding pin signal activity. Due to simulator randomization variation, you my see different data values being generated.

2. The ALU_out_transactions still have default values since you have not modified the ALU_out_monitor BFM code yet.

DUT-code

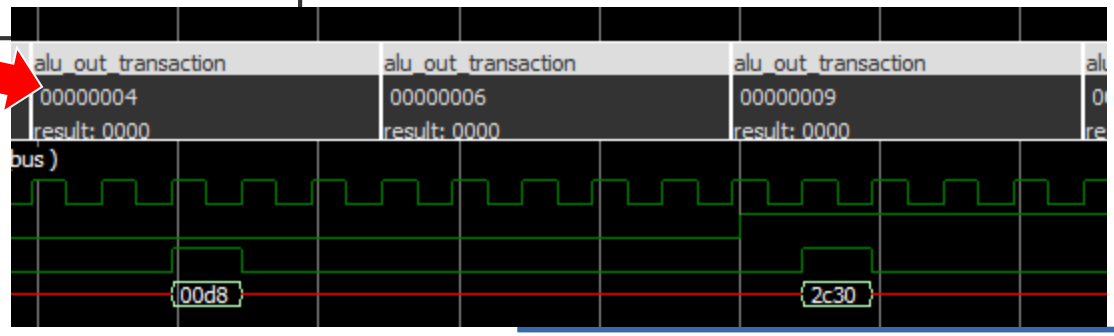
Adding Protocol Information To The Monitor BFM

verification_ip/interface_packages/ALU_out_pkg/src/ALU_out_monitor_bfm.sv

- Modifying the ALU_out monitor BFM
  - Go to the folder ***verification_ip/interface_packages/ALU_out_pkg/src***
  - Edit the file ***ALU_out_monitor_bfm.sv*** and locate the '***do_monitor***' task
  - By default the UVMF generator just has 4 consecutive clock delays inserted in to the monitor. No data is actually read from the ALU_out bus interface
  - This code needs to be modified to implement the interface protocol

Original Code

```
128   // ****************************************************************************
129
130   task do_monitor(output ALU_out_monitor_s ALU_out_monitor_struct);
131     // UVMF_CHANGE_ME : Implement protocol monitoring.  The commented reference code
151     @(posedge clk_i);
152     @(posedge clk_i);
153     @(posedge clk_i);
154     @(posedge clk_i);
155     // pragma uvmf custom do_
156   endtask
```



DUT-code

NOTES
This is why we see the ALU_out_transactions are all 4 cycles long and the displayed 'result' data values are just the language type defaults

Adding Protocol Information To The Monitor BFM

verification_ip/interface_packages/ALU_out_pkg/src/ALU_out_monitor_bfm.sv

- Modifying the ALU_out monitor BFM
  - Replace the 4 consecutive clock cycle delays with the following code

```
128   // ***************************************************************
129
130   task do_monitor(output ALU_out_monitor_s ALU_out_monitor_struct);
131     // UVMF CHANGE ME : Implement protocol monitoring.  The commente
151
152       while ( done_i == 1'b0) @(posedge clk_i)
153       ALU_out_monitor_struct.result = result_i;
154
155     // pragma uvmf custom do_monitor end
156   endtask
```

Modified Code

NOTES
- Wait until done_i goes high
- Read result value

DUT-code

Adding Protocol Information To The Monitor BFM

verification_ip/interface_packages/ALU_out_pkg/src/ALU_out_monitor_bfm.sv

- Checking the monitor BFM code changes
  - Use the compile.do/run.do or the Makefile (make debug) to check that there are no compilation errors in the code you have modified/added.

1. The ALU_out transactions are now showing the actual result data values and the transaction lengths match the corresponding pin signal activity.

DUT-code

Adding DUT Behaviour To The Predictor

verification_ip/environment_packages/ALU_env_pkg/src/ALU_predictor.svh

- Modifying the ALU predictor
  - Go to the folder ***verification_ip/environment_packages/ALU_env_pkg/src***
  - Edit the file ***ALU_predictor.svh*** and locate the `***write_ALU_in_agent_ae***` task
  - Transactions received through ALU_in_agent_ae initiate the execution of this function
  - This function performs prediction of DUT output values based on DUT input, configuration and state
  - This code needs to be modified to implement the DUT functionality

Original Code

```
72   // FUNCTION: write_ALU_in_agent_ae
73   // Transactions received through ALU_in_agent_ae initiate the execution of this function.
74   // This function performs prediction of DUT output values based on DUT input, configuration and state
75   virtual function void write_ALU_in_agent_ae(ALU_in_transaction #() t);
76     // pragma uvmf custom ALU_in_agent_ae_predictor begin
77     `uvm_info("PRED", "Transaction Received through ALU_in_agent_ae", UVM_MEDIUM)
78     `uvm_info("PRED", {"            Data: ",t.convert2string()}, UVM_FULL)
79
80     // Construct one of each output transaction type.
81     ALU_sb_ap_output_transaction = ALU_sb_ap_output_transaction_t::type_id::create("ALU_sb_ap_output_transaction");
82
83     //  UVMF_CHANGE_ME: Implement predictor model here.
84     `uvm_info("UNIMPLEMENTED_PREDICTOR_MODEL", "**********************************************************************************
85     `uvm_info("UNIMPLEMENTED_PREDICTOR_MODEL", "UVMF_CHANGE_ME: The ALU_predictor::write_ALU_in_agent_ae function needs to be completed
86     `uvm_info("UNIMPLEMENTED_PREDICTOR_MODEL", "**********************************************************************************
87
88     // Code for sending output transaction out through ALU_sb_ap
89     ALU_sb_ap.write(ALU_sb_ap_output_transaction);
90     // pragma uvmf custom ALU_in_agent_ae_predictor end
91   endfunction
```

Prediction

Adding DUT Behaviour To The Predictor

verification_ip/environment_packages/ALU_env_pkg/src/ALU_predictor.svh

- Modifying the ALU predictor
  - Insert the following case statement into the task to implement the ALU operations
  - Note that we deliberately ignore the RST_OP op code, taking care not to write a transaction out to the analysis export (which is connected to the scoreboard).

```
75   // Construct one of each output transaction type.
76   ALU_sb_ap_output_transaction = ALU_sb_ap_output_transaction_t::type_id::create("ALU_sb_ap_output_transaction");
77
78     //  UVMF_CHANGE_ME: Implement predictor model here.
79       case (t.op)
80         add_op: begin
81                   ALU_sb_ap_output_transaction.result = t.a + t.b;
82                   `uvm_info("PREDICT",{"ALU_OUT: ",ALU_sb_ap_output_transaction.convert2string()},UVM_MEDIUM);
83                   // Code for sending output transaction out through alu_sb_ap
84                   ALU_sb_ap.write(ALU_sb_ap_output_transaction);
85                 end
86         and_op: begin
87                   ALU_sb_ap_output_transaction.result = t.a & t.b;
88                   `uvm_info("PREDICT",{"ALU_OUT: ",ALU_sb_ap_output_transaction.convert2string()},UVM_MEDIUM);
89                   // Code for sending output transaction out through alu_sb_ap
90                   ALU_sb_ap.write(ALU_sb_ap_output_transaction);
91                 end
92         xor_op: begin
93                   ALU_sb_ap_output_transaction.result = t.a ^ t.b;
94                   `uvm_info("PREDICT",{"ALU_OUT: ",ALU_sb_ap_output_transaction.convert2string()},UVM_MEDIUM);
95                   // Code for sending output transaction out through alu_sb_ap
96                   ALU_sb_ap.write(ALU_sb_ap_output_transaction);
97                 end
98         mul_op: begin
99                   ALU_sb_ap_output_transaction.result = t.a * t.b;
100                  `uvm_info("PREDICT",{"ALU_OUT: ",ALU_sb_ap_output_transaction.convert2string()},UVM_MEDIUM);
101                  // Code for sending output transaction out through alu_sb_ap
102                  ALU_sb_ap.write(ALU_sb_ap_output_transaction);
103                end
104      endcase // case (op_set)
105
106  endfunction
```

Modified Code

Prediction

Adding DUT Behaviour To The Predictor

verification_ip/environment_packages/ALU_env_pkg/src/ALU_predictor.svh

- Checking the predictor code changes
  - Use the invoke_questa.bat or the Makefile (make debug) to check that there are no compilation errors in the code you have modified/added.
  - Run the simulation to the end and examine the transcript
  - During the simulation, the scoreboard will generate a message each time it does a compare of the expected data from the predictor and the actual data from the DUT
  - You should see several messages (one for each transaction) similar to the following:

```
# UVM_INFO
C:/MentorTools/questasim_10.7b/examples/UVM_Framework/UVMF_3.6h/uvmf_base_pkg/src/uvmf_in_order_scoreboard.svh(106) @
2269000: uvm_test_top.environment.ALU_sb [SCBD] MATCH! - EXPECTED: result:0x00fc ACTUAL: result:0x00fc
```

  - The scoreboard will also report a summary of the total number of compares and the number of MATCHES/MISMATCHES

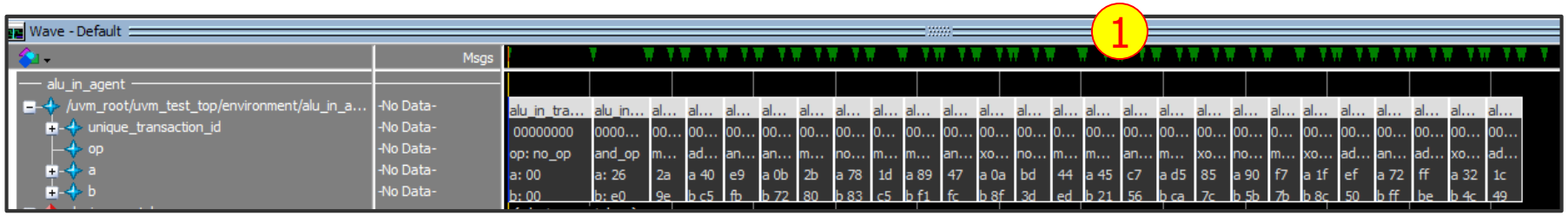
```
# UVM_INFO C:/MentorTools/questasim_10.7b/examples/UVM_Framework/UVMF_3.6h/uvmf_base_pkg/src/uvmf_scoreboard_base.svh(234)
@ 6379000: uvm_test_top.environment.ALU_sb [SCBD] SCOREBOARD_RESULTS:  PREDICTED_TRANSACTIONS=22 MATCHES=22
MISMATCHES=0
```

  - If you see the above messages, then the predictor and scoreboard are working correctly

Prediction

Status So Far

- Basic ALU operation appears to be working
  1. There should be no errors at this stage. In the wave window there should be no red triangles, which would indicate UVM Errors from the scoreboard

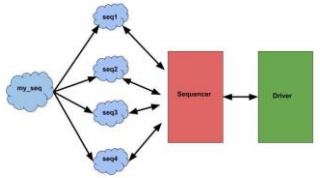


- Still To Do
  - Create a new test & sequence to exercise the RST_OP which is currently not being tested.
  - This is due to the constraint we specified back in the YAML config file for the ALU_in interface which only selects from the following ALU operations.

```
46      transaction_constraints:
47      - name: valid_op_c
48        value: '{ op inside {no_op, add_op, and_op, xor_op, mul_op}; }'
49
```

Test & Sequence

Creating an interface reset sequence
verification_ip/interface_packages/ALU_in_pkg/src
ALU_in_reset_sequence.svh



- UVMF Generated Sequence

1. ALU_in agent was generated with the following sequence

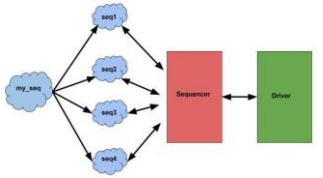
   interface_packages/ALU_in_pkg/src/ALU_in_random_sequence.svh

2. It randomizes ALU operations, selecting from no_op, add_op, and_op, xor_op & mul_op
3. Copy the ALU_in_random_sequence.svh file to ALU_in_reset_sequence.svh
4. Edit the sequence and change all references to ALU_in_random_sequence to ALU_in_reset_sequence.
5. After the randomization of the ALU_in_transaction, set the ALU op = RST_OP

Test & Sequence

Creating an interface reset sequence

verification_ip/interface_packages/ALU_in_pkg/src
ALU_in_reset_sequence.svh



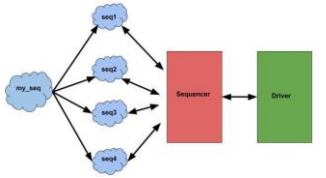

```
18 class ALU_in_reset_sequence #(
19       int ALU_IN_OP_WIDTH = 8
20       )
21   extends ALU_in_sequence_base #(
22       .ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH)
23       );
24
25   `uvm_object_param_utils( ALU_in_reset_sequence #(
26                            ALU_IN_OP_WIDTH
27                            ))
28
29   // pragma uvmf custom class_item_additional begin
30   // pragma uvmf custom class_item_additional end
31
32   //***************************************************************
33   function new(string name = "");
34     super.new(name);
35   endfunction: new
36
37   // ****************************************************************************
38   // TASK : body()
39   // This task is automatically executed when this sequence is started using the
40   // start(sequencerHandle) task.
41   //
42   task body();
43     begin
44       // Construct the transaction
45       req=ALU_in_transaction#(
46                 .ALU_IN_OP_WIDTH(ALU_IN_OP_WIDTH)
47                 )::type_id::create("req");
48       start_item(req);
49       // Randomize the transaction
50       if(!req.randomize()) `uvm_fatal("SEQ", "ALU_in_reset_sequence::body()-ALU_in_transaction randomization failed")
51       // set the operation to be a reset
52       req.op = rst_op;
53       // Send the transaction to the ALU_in_driver_bfm via the sequencer and ALU_in_driver.
54       finish_item(req);
55       `uvm_info("SEQ", {"Response:",req.convert2string()},UVM_MEDIUM)
56     end
57
58   endtask
59
60 endclass: ALU_in_reset_sequence
```

Modified Code

Test & Sequence

Creating an interface reset sequence

verification_ip/interface_packages/ALU_in_pkg/ALU_in_pkg.sv



STEPS

- Update the ALU_in_pkg to include the newly created ALU_reset_sequence.svh file
- The compilation script will compile this package and therefore all files that it includes

```
31 package ALU_in_pkg;
32
33     import uvm_pkg::*;
34     import uvmf_base_pkg_hdl::*;
35     import uvmf_base_pkg::*;
36     import ALU_in_pkg_hdl::*;
37
38     `include "uvm_macros.svh"
39     `include "src/ALU_in_macros.svh"
40
41     export ALU_in_pkg_hdl::*;
42
43
44
45     // Parameters defined as HVL parameters
46
47     `include "src/ALU_in_typedefs.svh"
48     `include "src/ALU_in_transaction.svh"
49
50     `include "src/ALU_in_configuration.svh"
51     `include "src/ALU_in_driver.svh"
52     `include "src/ALU_in_monitor.svh"
53
54     `include "src/ALU_in_transaction_coverage.svh"
55     `include "src/ALU_in_sequence_base.svh"
56     `include "src/ALU_in_random_sequence.svh"
57     `include "src/ALU_in_reset_sequence.svh"
58
59     `include "src/ALU_in_responder_sequence.svh"
60     `include "src/ALU_in2reg_adapter.svh"
61
62     `include "src/ALU_in_agent.svh"
63
64     // pragma uvmf custom package_item_additional begin
65     // UVMF_CHANGE_ME : When adding new interface sequences to the src directory
66     //    be sure to add the sequence file here so that it will be
67     //    compiled as part of the interface package.  Be sure to place
68     //    the new sequence after any base sequences of the new sequence.
69     // pragma uvmf custom package_item_additional end
70
71 endpackage
```

Modified Code

Test & Sequence

Creating a new bench virtual sequence

project_benches/ALU/tb/sequences/src/ALU_random_sequence.svh

- UVMF Generated Sequence

1. ALU_in bench was generated with the following virtual sequence

   project_benches/ALU/tb/sequences/src/ALU_bench_sequence_base.svh

2. This is the default sequence that gets ran by the default test.
3. Extend this sequence to create a new sequence called ALU_random_sequence.
4. We have the handle for the ALU_in_random sequence from the base class, but we need to define a handle for the new ALU_in_reset_sequence
5. Create the sequences
6. Utilize the agent configuration methods to wait until the reset is released and then wait a few clock cycles more
7. In the body of the sequence we will generate some random ALU operations, followed by a reset operation and then we will generate some more random ALU operations.
8. Utilize the agent configuration method to wait 50 clock cycles before ending the sequence

Test & Sequence

Creating a new bench virtual sequence

project_benches/ALU/tb/sequences/src/ALU_random_sequence.svh

```
24 class ALU_random_sequence #(int ALU_IN_OP_WIDTH = 8) extends ALU_bench_sequence_base;
25
26   `uvm_object_utils(ALU_random_sequence)
27
28   // Define type and handle for reset sequence
29   typedef ALU_in_reset_sequence #(ALU_IN_OP_WIDTH) ALU_in_reset_sequence_t;
30   ALU_in_reset_sequence_t ALU_in_reset_s;
31
32   // Random sequence typedef and handle defined in base class
33
34   // constructor
35   function new(string name = "");
36     super.new(name);
37   endfunction : new
38
39   virtual task body();
40
41     ALU_in_agent_random_seq = ALU_in_random_sequence#()::type_id::create("ALU_in_agent_random_seq");
42     ALU_in_reset_s  = ALU_in_reset_sequence#()::type_id::create("ALU_in_reset_s");
43
44     ALU_in_agent_config.wait_for_reset();
45     ALU_in_agent_config.wait_for_num_clocks(10);
46
47     repeat (10) ALU_in_agent_random_seq.start(ALU_in_agent_sequencer);
48     ALU_in_reset_s.start(ALU_in_agent_sequencer);
49     repeat (5) ALU_in_agent_random_seq.start(ALU_in_agent_sequencer);
50
51     ALU_in_agent_config.wait_for_num_clocks(50);  // 50 = 1000ns/20ns
52
53   endtask
54
55 endclass : ALU_random_sequence
```

3 4 5 6 7 8

New Sequence Code

Creating a new bench level sequence

project_benches/ALU/tb/sequences/ALU_sequence_pkg.sv

STEPS

- Update the ALU_sequences_pkg to include the newly created ALU_random_sequence.svh file
- The compilation script will compile this package and therefore all files that it includes

```
22 package ALU_sequences_pkg;
23   import uvm_pkg::*;
24   import uvmf_base_pkg::*;
25   import ALU_in_pkg::*;
26   import ALU_in_pkg_hdl::*;
27   import ALU_out_pkg::*;
28   import ALU_out_pkg_hdl::*;
29   import ALU_parameters_pkg::*;
30   import ALU_env_pkg::*;
31   `include "uvm_macros.svh"
32   `include "src/ALU_bench_sequence_base.svh"
33   `include "src/register_test_sequence.svh"
34   `include "src/example_derived_test_sequence.svh"
35   `include "src/ALU_random_sequence.svh"
36
37   // pragma uvmf custom package_item_additional begin
38   // UVMF_CHANGE_ME : When adding new sequences to the src directory
39   //    be sure to add the sequence file here so that it will be
40   //    compiled as part of the sequence package.  Be sure to place
41   //    the new sequence after any base sequences of the new sequence.
42   // pragma uvmf custom package_item_additional end
43
44 endpackage
```

Modified Code

Adding a New UVM Test

project_benches/ALU/tb/tests/src/ALU_random_test.svh

- Example derived test provided in

  tests/src/example_derived_test.svh

1. Create a new test, ***ALU_random_test*** & extend it from ***test_top***

2. In the build phase, specify a factory override of the default sequence (which is ***ALU_bench_sequence_base***) to replace it with the new sequence ***ALU_random_sequence***.

```
24  class ALU_random_test extends test_top;
25
26    `uvm_component_utils(ALU_random_test)
27
28    // constructor
29    function new(string name = "", uvm_component parent = null );
30      super.new(name, parent);
31    endfunction : new
32
33
34    virtual function void build_phase(uvm_phase phase );
35      // UVM FActory override.  Override ALU_bench_sequence_base with ALU_random_sequence
36      ALU_bench_sequence_base::type_id::set_type_override(ALU_random_sequence #(8)::get_type());
37      super.build_phase(phase);
38    endfunction : build_phase
39
40
41  endclass : ALU_random_test
```

New Code

Test & Sequence

Adding a New UVM Test

project_benches/ALU/tb/tests/ALU_tests_pkg.sv

- Add the new test to the ALU_tests_pkg

```
21 package ALU_tests_pkg;
22
23   import uvm_pkg::*;
24   import uvmf_base_pkg::*;
25   import ALU_parameters_pkg::*;
26   import ALU_env_pkg::*;
27   import ALU_sequences_pkg::*;
28   import ALU_in_pkg::*;
29   import ALU_in_pkg_hdl::*;
30   import ALU_out_pkg::*;
31   import ALU_out_pkg_hdl::*;
32
33    `include "uvm_macros.svh"
34
35    `include "src/test_top.svh"
36    `include "src/register_test.svh"
37    `include "src/example_derived_test.svh"
38    `include "src/ALU_random_test.svh"
39
40 endpackage
```

Modified Code

Test & Sequence

Simulating The New Test

- Go to the ***project_benches/ALU/sim*** folder
- Windows Users
  - Edit run.do and modify the modify the last line where +UVM_TESTNAME specifies the test to run

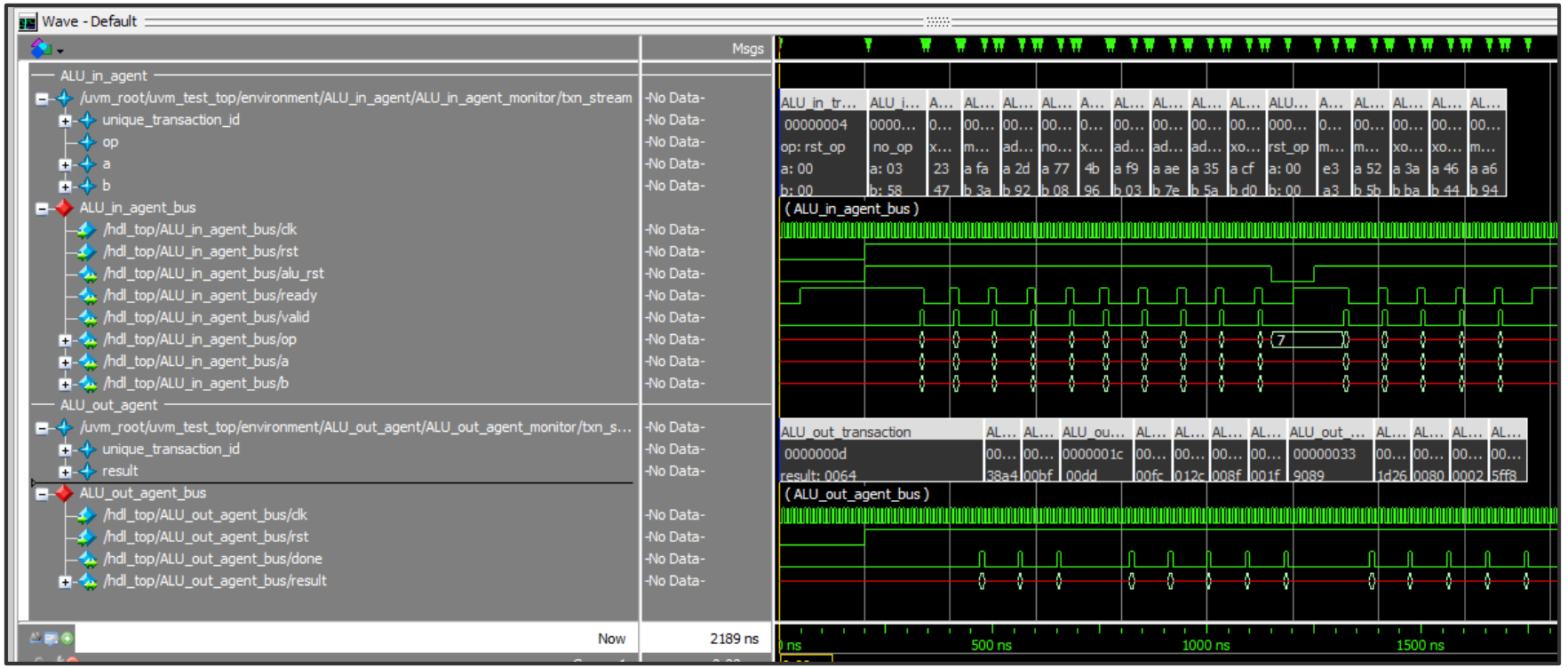
```
 9  quietly set cmd [format "vsim -i -sv_seed random +UVM_TESTNAME=ALU_random_test +UVM_VERBOSITY=UVM_HIGH
10  eval $cmd
```

- Linux Users
  - Execute 'make debug TEST_NAME=ALU_random_test'

Sim 2

Simulating The New Test

- Observe from the wave window
  1. No operations occur during the reset
  2. 10 random operations are then applied to the ALU
  3. A reset is then applied to the ALU
  4. 5 further random operations are then applied to the ALU



Sim 2

Adding Colour To The Transactions

verification_ip/interface_packages/ALU_in_pkg/src/ALU_in_transaction.svh

- Edit the file ALU_in_transaction.svh
- Find the function called 'add_to_wave'
  - It contains some comments about adding colour to the transactions
  - Add the code highlighted below to the function which will assign a different colour to the transactions depending on the opcode value

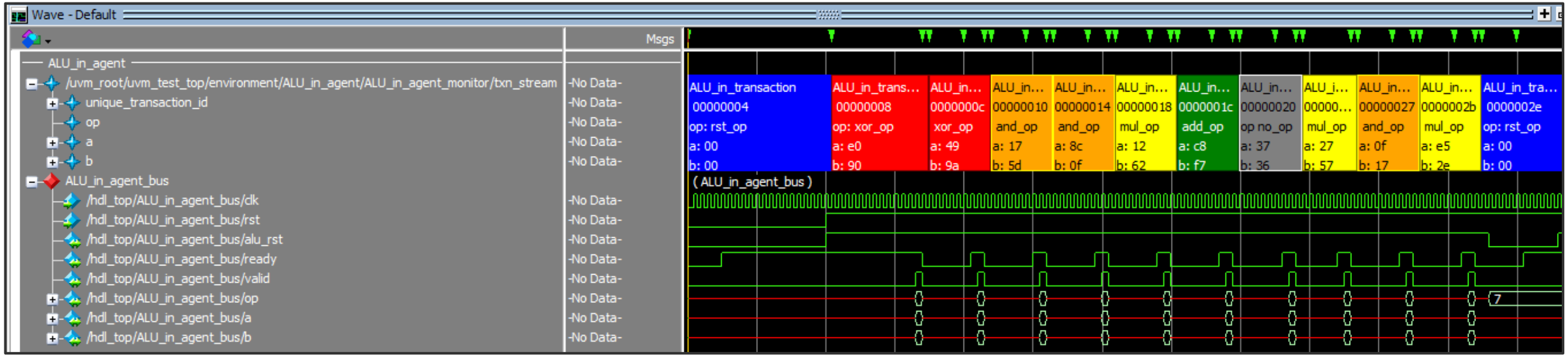
```
96    virtual function void add_to_wave(int transaction_viewing_stream_h);
97      if (transaction_view_h == 0)
98        transaction_view_h = $begin_transaction(transaction_viewing_stream_h,"ALU_in_transaction",start_time);
99      case (op)
100       no_op :   $add_color(transaction_view_h,"grey");
101       add_op :  $add_color(transaction_view_h,"green");
102       and_op :  $add_color(transaction_view_h,"orange");
103       xor_op :  $add_color(transaction_view_h,"red");
104       mul_op :  $add_color(transaction_view_h,"yellow");
105       rst_op :  $add_color(transaction_view_h,"blue");
106       default : $add_color(transaction_view_h,"cyan");
107     endcase
108     super.add_to_wave(transaction_view_h);
109 // UVMF_CHANGE_ME : Eliminate transaction variables not wanted in transaction viewing in the waveform viewer
110     $add_attribute(transaction_view_h,op,"op");
111     $add_attribute(transaction_view_h,a,"a");
112     $add_attribute(transaction_view_h,b,"b");
113     $end_transaction(transaction_view_h,end_time);
114     $free_transaction(transaction_view_h);
115   endfunction
```

Sim 2

Re-Simulate The New Test

The End

- Observe from the wave window
  1. The ALU_in transactions are now colour coded depending on the op code
  2. The colours match those specified in the add_to_wave function of the ALU_in_transaction class



- That completes the steps to get the UVMF environment running

A Siemens Business

www.mentor.com