UVM

UVM Generator

uvmgen

```
UVM Version : 2
Complete Env.(1) OR Individual Template(2)? : 1
Want to create your own methods[Instead of uvm shorthand macros]? : n
RAL env? : 0
Env. Name : environment
Agents? : y
Name of master agent:  : ethernet
Name of sequencer in ethernet master agent:  : eth_sequencer
Name of driver in ethernet master agent:  : eth_driver
Name of monitor in ethernet master agent:  : eth_monitor
Name of interface related to ethernet master agent:  : eth_intf
Name of transaction in ethernet master agent:  : eth_data
BU class for this transaction? : n
Name of slave agent:  : slave
Name of sequencer in slave slave agent:  : slave_sequencer
Name of driver in slave slave agent:  : slave_driver
Name of monitor in slave slave agent:  : slave_monitor
Name of physical interface related to slave slave agent:  : slave_intf
Name of transaction related to slave slave agent:  : eth_data
Driver information for the slave agent slave :  :
Driver Type : Driver, PULL DRIVER (uvm_driver)
Driver information for the master agent ethernet :  :
Driver Type : Driver, PULL DRIVER (uvm_driver)
Scoreboard? : y
Name of Scoreboard Class : scoreboard
```

8x8 Router UVM testbench

- /home/bgreene/ELEN613/RT_uvm
- Source code
  - proj/src
- Compile + Run directory
  - Proj/run

Ethernet Agent

- Verilog files
  - ethernet_eth_data.sv
  - ethernet_eth_driver.sv
  - ethernet_eth_intf.sv
  - ethernet_eth_monitor.sv
  - ethernet_eth_sequencer.sv
  - ethernet_sequence_library.sv

Ethernet_eth_data

```
class eth_data extends uvm_sequence_item;

  typedef enum {READ, WRITE } kinds_e;
  rand kinds_e kind;
  typedef enum {IS_OK, ERROR} status_e;
  rand status_e status;
  rand byte sa;

  // ToDo: Add constraint blocks to prevent error injection
  // ToDo: Add relevant class properties to define all transactions
  // ToDo: Modify/add symbolic transaction identifiers to match
   rand bit idle;
   rand bit [3:0] delay;
   rand bit [2:0] src,dst;
   rand bit [31:0] payload;

  constraint eth_data_valid {
     // ToDo: Define constraint to make descriptor valid
     delay == 10;
      idle == 0;
     status == IS_OK;
  }
  `uvm_object_utils_begin(eth_data)

     // ToDo: add properties using macros here

     `uvm_field_enum(kinds_e,kind,UVM_ALL_ON)
     `uvm_field_enum(status_e,status, UVM_ALL_ON)
     `uvm_field_int(idle, UVM_ALL_ON)
     `uvm_field_int(delay, UVM_ALL_ON)
     `uvm_field_int(src, UVM_ALL_ON)
     `uvm_field_int(dst, UVM_ALL_ON)
     `uvm_field_int(payload, UVM_ALL_ON)
  `uvm_object_utils_end

  extern function new(string name = "Trans");
endclass: eth_data


function eth_data::new(string name = "Trans");
  super.new(name);
endfunction: new
```

Ethernet_eth_driver

```
task eth_driver::reset_phase(uvm_phase phase);
  super.reset_phase(phase);
  // ToDo: Reset output signals
    drv_if.mck.frame_n <= '1;
    drv_if.mck.valid_n <= '1;
    drv_if.mck.di <= 'x;
    repeat (1) @(drv_if.mck);
endtask: reset_phase

task eth_driver::run_phase(uvm_phase phase);
  super.run_phase(phase);
  repeat (50) @(drv_if.mck);
fork
     tx_driver(0);
  join
endtask: run_phase

task eth_driver::tx_driver(int i);
 forever begin
     eth_data tr;
     // ToDo: Set output signals to their idle state
     this.drv_if.master.async_en      <= 0;
     `uvm_info("environment_DRIVER", "Starting transaction...",UVM_LOW)
     seq_item_port.get_next_item(tr);
      if (tr.dst==7 && tr.src==7) send1pkt(tr.src,tr);
     seq_item_port.item_done();
     `uvm_info("environment_DRIVER", "Completed transaction...",UVM_LOW)
     `uvm_info("environment_DRIVER", tr.sprint(),UVM_HIGH)
     `uvm_do_callbacks(eth_driver,eth_driver_callbacks,
                post_tx(this, tr))
  end
endtask : tx_driver
```

```
task eth_driver::send1pkt(int i,eth_data tr);
  if (i!== tr.src) $display("ERROR %d != %d",i,tr.src);
  if (tr.idle)
    begin
      repeat(tr.delay)  @(drv_if.mck);
      return;
    end
  $display($time, ": Sending packet src=%1d:dst=%1d",tr.src,tr.dst);
  repeat(5) @(drv_if.mck);
  drv_if.mck.frame_n[tr.src] <= 1'b0;
  drv_if.mck.di[tr.src] <= tr.dst[0];
  @(drv_if.mck)  drv_if.mck.di[tr.src] <= tr.dst[1];
  @(drv_if.mck)  drv_if.mck.di[tr.src] <= tr.dst[2];
  @(drv_if.mck)  drv_if.mck.di[tr.src] <= 1'b0;
  // Padding
  repeat(1) @(drv_if.mck) ;
  for (int i=0;i<32;i=i+1) begin
    drv_if.mck.valid_n[tr.src] <= 1'b0;
    drv_if.mck.di[tr.src] <= tr.payload[i];
    drv_if.mck.frame_n[tr.src] <= i==31;
    @(drv_if.mck);
  end
  drv_if.mck.valid_n[tr.src] <= 1'b1;
  drv_if.mck.di[tr.src] <= 1'bx;
  repeat (5)  @(drv_if.mck);

endtask:send1pkt
```

Ethernet_eth_monitor

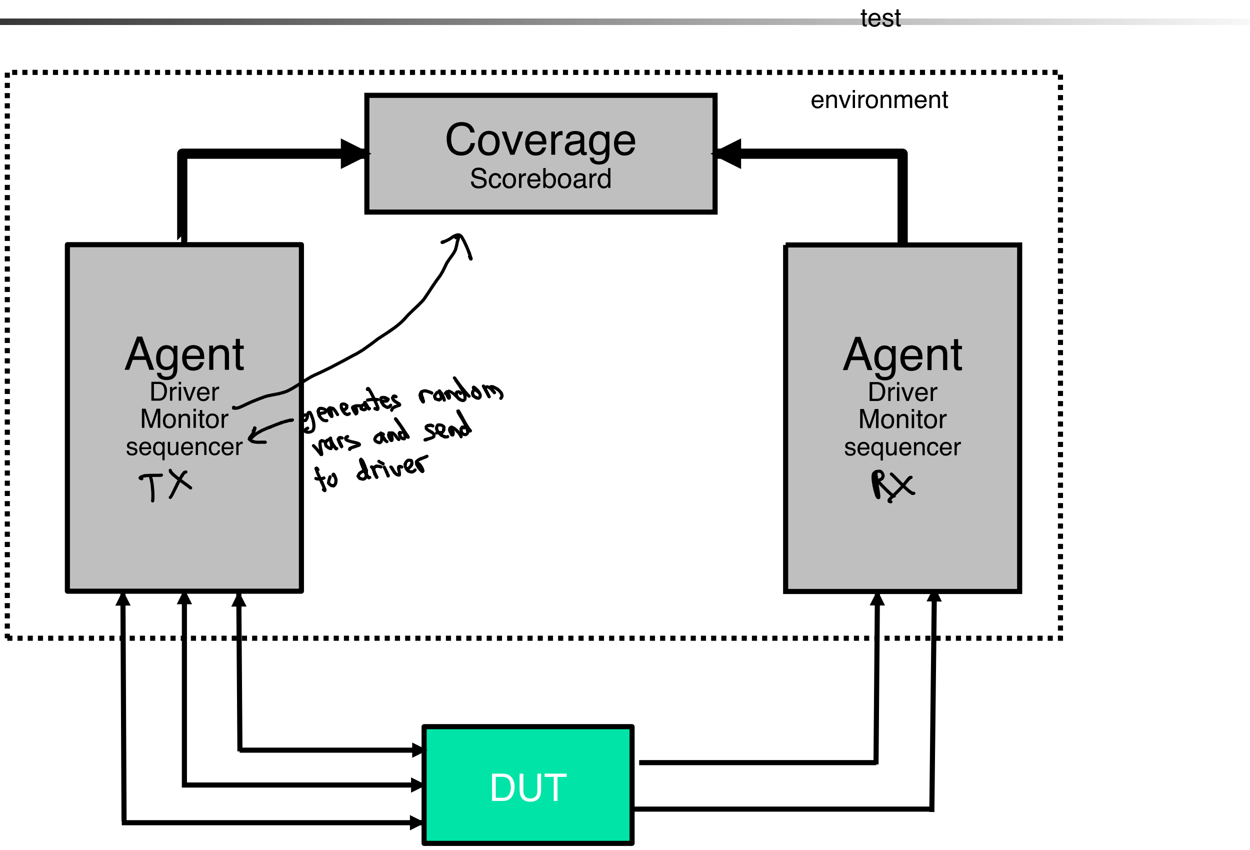
```
task eth_monitor::tx_monitor();
  forever begin
    eth_data tr;
    // ToDo: Wait for start of transaction

    `uvm_do_callbacks(eth_monitor,eth_monitor_callbacks,
              pre_trans(this, tr))
    `uvm_info("environment_MONITOR", "Starting transaction...",UVM_LOW)
    // ToDo: Observe first half of transaction
    rcv1pkt(7,tr);
    `uvm_do_callbacks(eth_monitor,eth_monitor_callbacks,
              pre_ack(this, tr))
    // ToDo: React to observed transaction with ACK/NAK
    `uvm_info("environment_MONITOR", "Completed transaction...",UVM_LOW)
    `uvm_info("environment_MONITOR", tr.sprint(),UVM_LOW)
    `uvm_do_callbacks(eth_monitor,eth_monitor_callbacks,
              post_trans(this, tr))
    mon_analysis_port.write(tr);
  end
endtask: tx_monitor
```

```
task eth_monitor::run_phase(uvm_phase phase);
  super.run_phase(phase);
 // phase.raise_objection(this,""); //Raise/drop objections in sequence file
  fork
    tx_monitor();
  join
 // phase.drop_objection(this);

task automatic eth_monitor::rcv1pkt(int i, ref eth_data tr);
   eth_data tmp;
   tmp = new();
   tmp.dst=i;
   while (mon_if.pck.frame_n[i]!=='0) @(mon_if.pck) ;
   while (mon_if.pck.valid_n[i]!=='0) @(mon_if.pck) ;
   for (int j=0;j<32;j=j+1) begin
     tmp.payload[j] <= mon_if.pck.di[i];
     @(mon_if.pck);
   end
   tr = tmp;
endtask:rcv1pkt
```

Overview of Components



UVM Phases

Build
Connect
End of elaboration
Start of simulation
Run (multiple sub-phases here
    Pre_reset,reset,post_reset,pre-configure,configurage,post-configure,pre_main,main,post-main,pre-shutdown,shutdown ,post-shutdown
Extract
Check
Report
final

packages → reusable scopes(block of code)
typedefs

UVM hello-world example

To compile

$\overbrace{\text{-ntb\_opts}}^{\text{native testbench options}}$

```
Vcs test.v –ntb_opts uvm
./simv +UVM_TESTNAME=my_test
```

running "my" → Task

package pl;
typedef logic my lo

```
module test;
import uvm_pkg::*;
class my_test extends uvm_test;
  `uvm_component_utils(my_test)
  function new(string name, uvm_component parent);
    super.new(name,parent);
  endfunction
  virtual task run_phase(uvm_phase phase);
    `uvm_info("TEST", "MY_TEST", UVM_MEDIUM);
  endtask
endclass
initial run_test();
endmodule


UVM_INFO @ 0: reporter [RNTST] Running test my_test...
UVM_INFO uvm.v(9) @ 0: uvm_test_top [TEST] MY_TEST
```

all classes must have base class of

U :

Macro Automation

UVM provides build-in macros to automate writing lots of repetitive code

```
`uvm_component_utils(my_test)
```

Registers the class in the factory

What is the “factory”

As the name implies, the uvm_factory is used to manufacture (create) UVM objects and components. Only one instance of the factory is present in a given simulation.

One should not use the SV method “new”, instead use the factory method uvm_ factory::create_object_by_type), this actually creates an derived class of the user-type, more efficient than creating a full-blown new class

Messaging Macros

what [illegible]

`uvm_info(ID,MSG,VERBOSITY)

- UVM_LOW
- UVM_MEDIUM
- UVM_HIGH
- UVM_FULL
- Controlled via +UVM_VERBOSITY at runtime

`uvm_warning(ID,MSG)

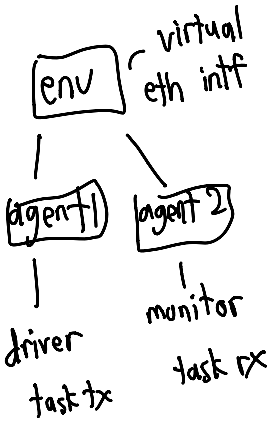
`uvm_error(ID,MSG)

`uvm_fatal(ID,MSG)

All uvm messages can be filtered, or controlled by verbosity

Agent

Agents encapsulate driver, monitor and sequencer



intf1.cb.xy2<=

```
class ethernet extends uvm_agent;
    // ToDo: add uvm agent properties here
    protected uvm_active_passive_enum is_active = UVM_ACTIVE;
    eth_sequencer mast_sqr;
    eth_driver mast_drv;
    eth_monitor mast_mon;
    typedef virtual eth_intf vif;
    vif mast_agt_if;

    `uvm_component_utils_begin(ethernet)
    //ToDo: add field utils macros here if required
    `uvm_component_utils_end
```

Build phase of Agent

All components must be created in the build-phase

```
virtual function void build_phase(uvm_phase phase);
     super.build_phase(phase);
     mast_mon = eth_monitor::type_id::create("mast_mon", this);
     if (is_active == UVM_ACTIVE) begin
        mast_sqr = eth_sequencer::type_id::create("mast_sqr", this);
        mast_drv = eth_driver::type_id::create("mast_drv", this);
     end
     if (!uvm_config_db#(vif)::get(this, "", "mst_if", mast_agt_if)) begin
        `uvm_fatal("AGT/NOVIF", "No virtual interface specified for this agent i
     end
     uvm_config_db# (vif)::set(this,"mast_drv","mst_if",mast_drv.drv_if);
     uvm_config_db# (vif)::set(this,"mast_mon","mst_if",mast_mon.mon_if);
  endfunction: build_phase
```

Uvm_config_db

Centralized database where type specific information can be stored and retrieved

uvm_resource_db is the low-level resource database that users can write to or read from.
The uvm_config_db is layered on top of the resource database and provides a typed interface for a configuration setting that is consistent with the
Information can be read from or written to the database at any time during simulation. A resource may be

```
logic x,y;
initial begin
x = 1;
$display(x,y);
uvm_config_db# (logic)::set(null,"mast_drv","x",x);
$display(x,y);
uvm_config_db# (logic)::get(null,"mast_drv","x",y);
$display(x,y);
```

Connect phase of Agent

This phase connects the TLM ports between the driver and the sequencer

TLM = transaction level modeling

Methods used in TLM

- Get,put

connect sequencer & driver

```
virtual function void connect_phase(uvm_phase phase);
     super.connect_phase(phase);
     if (is_active == UVM_ACTIVE) begin
                 mast_drv.seq_item_port.connect(mast_sqr.seq_item_export);
     end
  endfunction
```

Driver Class <-> sequencer

Driver class received transaction data from sequencer via the TLM port with get_next_item call and item_done

```
task eth_driver::tx_driver(int i);
 forever begin
      eth_data tr;
      // ToDo: Set output signals to their idle state
      this.drv_if.master.async_en      <= 0;
      `uvm_info("environment_DRIVER", "Starting transaction...",UVM_LOW)
      seq_item_port.get_next_item(tr);
       if (tr.dst==7 && tr.src==7) send1pkt(tr.src,tr); — would be tx method
      seq_item_port.item_done();
      `uvm_info("environment_DRIVER", "Completed transaction...",UVM_LOW)
      `uvm_info("environment_DRIVER", tr.sprint(),UVM_HIGH)
      `uvm_do_callbacks(eth_driver,eth_driver_callbacks,
                   post_tx(this, tr))

   end
endtask : tx_driver
```

Sequence Class

Stimulus generation is a sequence
2 handles
- Req = request to driver
- Rsp = response from driver

`uvm_do

`uvm_do_with

packet_length=25;

or something like that

```
class sequence_0 extends base_sequence;
  `uvm_object_utils(sequence_0)
  `uvm_add_to_seq_lib(sequence_0,eth_sequencer_sequence_library

  function new(string name = "seq_0");
    super.new(name);
endfunction:new

  virtual task body();
    $display("inside sequence_0" );
    repeat(10) begin
      `uvm_do(req);  ← sends request to driver
        #500; //Dummy delay added here for test run
    end
  endtask
endclass
```

Driver – sequence interaction

```
class sequence_0 extends base_sequence;

  virtual task body();
    $display("inside sequence_0" );
    repeat(1) begin
      `uvm_do(req);
          //`uvm_create(req);
          //`start_item(req);
          //`req.randomize();
          //finish_item(req);
    end
  endtask
endclass
```

```
Task start_item(req)
  sqr.wait_for_grant()
  sqr.begin_tr(req)
endtask
```

```
Eth_driver::tx_driver();

Seq_item_port.get_next_item(tr);
.. Process ..
Seq_item_port.item_done();
```

```
Task finish_item(req)
  sqr_send_request(req)
  sqr.wait_for_item_done();
  sqr.end_tr(req)
endtask
```

(within tests directory)

Environment_test

Purpose is to test out environment , first test to run

Let's just run the sequence "sequence_0"

```
class environment_test extends uvm_test;
  `uvm_component_utils(environment_test)
  environment_env env;
  function new(string name, uvm_component parent);
    super.new(name, parent);
  endfunction
  virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    env = environment_env::type_id::create("env", this);
    uvm_config_db #(uvm_object_wrapper)::set(this, "env.*.configure_phase", "default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this, "env.*.main_phase", "default_sequence", null);
    uvm_config_db #(uvm_object_wrapper)::set(this, "env.master_agent.mast_sqr.main_phase", "default_sequ
  endfunction
endclass : environment_test
```

telling not to do reg sequence →

$$\begin{tikzcd}
\text{RT\_uvm} & \text{automation macros} \\
\text{UVM} \arrow[u] \arrow[r] \arrow[d] & \text{UVM generator} \arrow[u] \\
\text{RT\_Layered\_TB} &
\end{tikzcd}$$

data class

covergroup

$$\begin{tikzcd}
\text{wrapper} & \\
\text{coverage} \arrow[u] \arrow[r] \arrow[d] & \text{feature in the IP} \\
\text{DUT} &
\end{tikzcd}$$

New Generator

```
tr.randomize() with {src==0; dst==7;};
```

<u>Life</u>

$$\begin{tikzcd}
\boxed{\text{gen}} \arrow[d] & \boxed{\text{SB}} & \\
\boxed{\text{Driver}} \arrow[r] \arrow[ur, bend left] & \boxed{\text{DUT}} \arrow[r] & \boxed{\text{monitor}} \arrow[ul]
\end{tikzcd}$$

```
make → simv + UVM_TESTNAME = environment_test
```

make UVM_TEST = test01

accumulating coverage

simv + UVM_TESTNAME =test01 +

simv + UVM_TESTNAME=test02 +