Design and UVM Verification of ALU

Saad Kouzmane

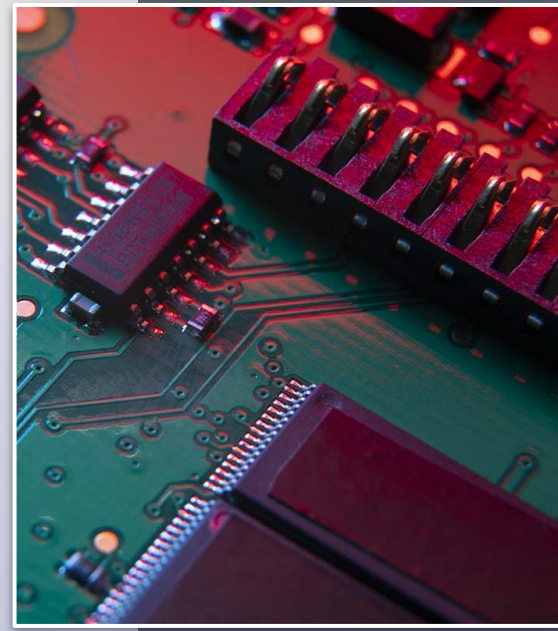
Table of contents

01 Introduction

02 Objective

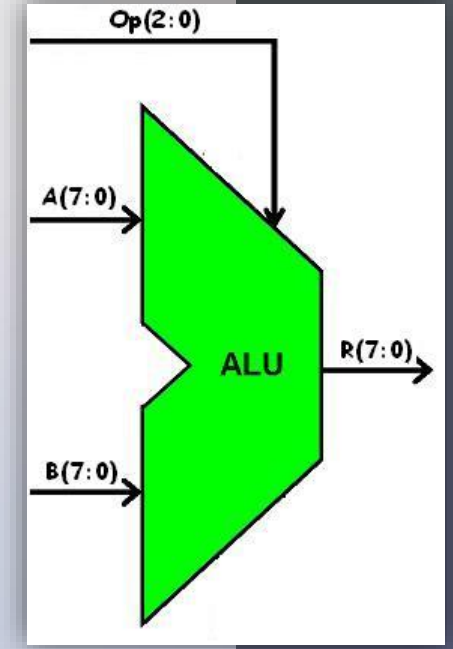
03 UVM Overview

04 UVM ALU Testing



Introduction

In this project, we demonstrate the design and verification of an Arithmetic Logic Unit (ALU) using Universal Verification Methodology (UVM).
UVM is a powerful framework widely adopted in the semiconductor industry for verifying complex digital designs.
This project provides valuable insights into digital logic design, verification techniques, and the practical application of UVM in the context of ALU design.



Objectives

Design ALU:

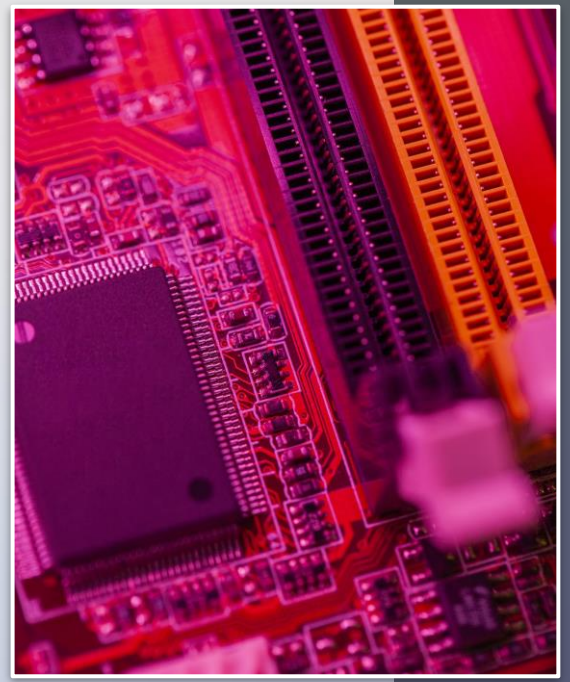
Create a fully functional Arithmetic Logic Unit (ALU) with various operations

Understand UVM

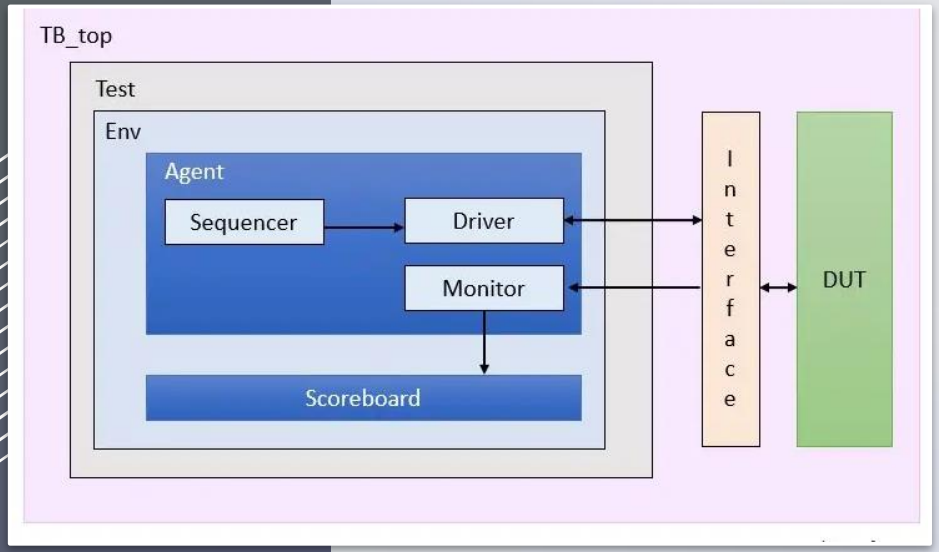
Gain an understanding of the Universal Verification Methodology and its key concepts,

Verify ALU Functionality

Develop a comprehensive verification environment using UVM to thoroughly test the ALU's functionality, ensuring it performs operations accurately.



UVM Overview



The Universal Verification Methodology (UVM) architecture is a standardized framework for verifying digital designs. It consists of key components like sequences, drivers, monitors, agents, and a scoreboard. These components work together to create a structured and efficient environment for testing digital designs. UVM promotes reusability, scalability, and ease of verification, making it a popular choice for complex design verification tasks.

UVM ALU Testing

Design under test(DUT):

In the UVM architecture, the DUT (Design Under Test) is like the main character. It's an 8-bit ALU (Arithmetic Logic Unit) module that we want to make sure works correctly. The DUT follows a script (test) and interacts with other characters (testbench components) to prove it's doing the right math operations and giving the right answers.

```
          input clk,
          input [7:0] A,B,  // ALU 8bit Inputs
input [2:0] op,// ALU operation Selection
          output [7:0] ALU_Out, // ALU 8bit Output

  );
  reg [7:0] ALU_Result;
  assign ALU_Out = ALU_Result; // ALU out

always @(posedge clk)
  begin
    case(op)
      4'b0000: // Addition
         ALU_Result = A + B ;
      4'b0001: // Subtraction
         ALU_Result = A - B ;
      4'b0010: // Multiplication
         ALU_Result = A * B;
      4'b0011: // Division
         ALU_Result = A/B;
      4'b0100: // Logical shift left
         ALU_Result = A<<1;
      4'b0101: // Logical shift right
         ALU_Result = A>>1;
      4'b0110: // Rotate left
         ALU_Result = {A[6:0],A[7]};
      4'b0111: // Rotate right
         ALU_Result = {A[0],A[7:1]};

        default: ALU_Result = A + B ;
      endcase
  end

endmodule
```

UVM ALU Testing

Interface

The UVM interface acts like a bridge, connecting the main action (the DUT) to the backstage crew (drivers, monitors, etc.). It makes sure the main character gets the right information and shares results with everyone involved. It's like a communication wizard, making things happen in our digital play.

```
interface alu_if();
  logic [7:0] A;
  logic [7:0] B;
  logic [2:0] op;
  logic [7:0] ALU_Out;
  logic clk;


endinterface
```

UVM ALU Testing

Sequence

UVM sequences are made up of several data items which can be put together in different ways to create interesting scenarios. They are executed by an assigned sequencer which then sends data items to the driver. Hence, sequences make up the core stimuli of any verification plan.

```
class generator extends uvm_sequence#(transaction); //t SV
object
`uvm_object_utils(generator)

transaction t;
integer i;

function new(input string inst = "GEN");
    super.new(inst);
endfunction

//no phases in generator
virtual task body();
    t = transaction::type_id::create("TRANS");
    for(i =0; i< 15; i++) begin
        start_item(t);
        t.randomize();
        `uvm_info("GEN", "Data send to Driver", UVM_NONE);
        t.print(uvm_default_line_printer);
        #10;
        finish_item(t);
    end
endtask

endclass
```

UVM ALU Testing

Transaction

transaction is like a messenger. Its purpose is to carry important data and instructions. In our ALU project, the transaction class holds the data to test our ALU and checks the expected results. Transactions make testing organized and easy. They help ensure our ALU works correctly and can be used in other projects too.

```
class transaction extends uvm_sequence_item;

rand bit [7:0] A;
rand bit [7:0] B;
rand bit [2:0] op;
bit [7:0] ALU_Out;

  //constraint addr_C {op > 3; op < 5;};

function new(input string inst = "TRANS");
    super.new(inst);
endfunction
// register to factory
`uvm_object_utils_begin(transaction)
`uvm_field_int(A,UVM_DEFAULT)
`uvm_field_int(B,UVM_DEFAULT)
`uvm_field_int(op,UVM_DEFAULT)
`uvm_field_int(ALU_Out,UVM_DEFAULT)
`uvm_object_utils_end

endclass
```

UVM ALU Testing

Driver

The driver signals to the ALU interface by delivering the transaction data and control signals to the ALU. It acts as the intermediary between the test environment and the ALU. The driver takes care of sending the appropriate signals like data, clock, and control to the ALU, ensuring that the ALU processes the data correctly and produces the expected results.

```
class driver extends uvm_driver#(transaction);
`uvm_component_utils(driver)


function new(input string inst = "DRV", uvm_component c);
    super.new(inst,c);
endfunction

transaction data;
virtual alu_if aif;
virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    data = transaction::type_id::create("TRANS");
    if(!uvm_config_db#(virtual alu_if)::get(this,"","aif",aif))
        `uvm_info("DRV", "Unable to access Interface", UVM_NONE);
endfunction

virtual task run_phase(uvm_phase phase);
    forever begin
    //specify data coming from tlm port
        seq_item_port.get_next_item(data);
        aif.A = data.A;
        aif.B = data.B;
        aif.op=data.op;
        `uvm_info("DRV","Send data to DUT", UVM_NONE);
        data.print(uvm_default_line_printer);
        seq_item_port.item_done();
    @(posedge aif.clk);
end
endtask

endclass
```

UVM ALU Testing

Monitor

The monitor is responsible for capturing signal activity from the design interface and translating it into transaction-level data objects. These data objects can be sent to other UVM components for further analysis and verification,

```
class monitor extends uvm_monitor;
`uvm_component_utils(monitor)
uvm_analysis_port #(transaction) send;
virtual alu_if aif;
transaction t;

function new(input string inst = "MON", uvm_component c);
    super.new(inst,c);
    send = new("WRITE",this);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t = transaction::type_id::create("TRANS");
    if(!uvm_config_db#(virtual alu_if)::get(this,"","aif",aif))
        `uvm_info("MON", "Unable to access Interface", UVM_NONE);
endfunction

virtual task run_phase(uvm_phase phase);
forever begin
  @(posedge aif.clk);
  t.A = aif.A  ;
  t.B =aif.B ;
  t.op = aif.op;

  // wait clock cycle to send it to scoreboard to not get garbage
data
 @(posedge aif.clk)
t.ALU_Out = aif.ALU_Out ;
`uvm_info("MON","Send data to Scoreboard", UVM_NONE);
t.print(uvm_default_line_printer);
send.write(t);

end
endtask

endclass
```

UVM ALU Testing

Agent

Agent is like a team leader in a factory. It manages a group of specialized workers (sequencer, driver, and monitor) that work together to verify the ALU. The agent sets up how these workers communicate and work, and it can be configured to adjust how the verification process happens. It's like a supervisor making sure the right tasks get done

```
class agent extends uvm_agent;
`uvm_component_utils(agent)

function new(input string inst = "AGENT", uvm_component c);
super.new(inst,c);
endfunction

monitor m;
driver d;
//type of data object
uvm_sequencer #(transaction) seq;


virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    m = monitor::type_id::create("MON",this);
    d = driver::type_id::create("DRV",this);
    seq = uvm_sequencer #(transaction)::type_id::create("SEQ",this);
endfunction

virtual function void connect_phase(uvm_phase phase);
    super.connect_phase(phase);
    d.seq_item_port.connect(seq.seq_item_export);
endfunction
endclass
```

UVM ALU Testing

Environment

Environment is like the entire factory where all the work happens. It ensures that the agent and other UVM components work together seamlessly to verify the ALU design. It's like a supervisor overseeing the work of a single team to make sure everything is in order and the verification process is successful

```
class transaction extends uvm_sequence_item;

rand bit [7:0] A;
rand bit [7:0] B;
rand bit [2:0] op;
bit [7:0] ALU_Out;

  //constraint addr_C {op > 3; op < 5;};

function new(input string inst = "TRANS");
    super.new(inst);
endfunction
// register to factory
`uvm_object_utils_begin(transaction)
`uvm_field_int(A,UVM_DEFAULT)
`uvm_field_int(B,UVM_DEFAULT)
`uvm_field_int(op,UVM_DEFAULT)
`uvm_field_int(ALU_Out,UVM_DEFAULT)
`uvm_object_utils_end

endclass
```

UVM ALU Testing

Test

The UVM test acts as the conductor of the entire verification process. It orchestrates the generator, environment, and the ALU under test to run a series of test scenarios. Think of it as the director of a play, ensuring that all actors (components) are in their roles and the show (verification) runs smoothly

```
class test extends uvm_test;
`uvm_component_utils(test)

function new(input string inst = "TEST", uvm_component c);
super.new(inst,c);
endfunction

generator gen;
env e;

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    e = env::type_id::create("ENV",this);
    gen = generator::type_id::create("GEN",this);
endfunction

virtual task run_phase(uvm_phase phase);
    phase.raise_objection(this);
    gen.start(e.a.seq);
    #10;
    phase.drop_objection(this);
endtask
endclass
```

UVM ALU Testing

Scoreboard

The UVM scoreboard is the referee of the verification process, responsible for comparing the test data generated by the ALU with the golden or expected data. It ensures that the ALU's output matches the predefined standards, acting as a judge to determine whether the ALU has performed its task correctly. It's like a match between the test data (the player) and the golden data (the rulebook) to decide if the ALU is in compliance. If there's a mismatch, the scoreboard raises a flag

```
class scoreboard extends uvm_scoreboard;
`uvm_component_utils(scoreboard)
// first type of data and class where we have impl
uvm_analysis_imp #(transaction,scoreboard) recv;
transaction t;


function new(input string inst = "SCO", uvm_component c);
    super.new(inst,c);
    recv = new("READ",this);
endfunction

virtual function void build_phase(uvm_phase phase);
    super.build_phase(phase);
    t = transaction::type_id::create("TRANS");
endfunction
// write methode and perfom the check
virtual function void write(transaction data);
  t =data;

`uvm_info("SCO","Data rcvd from Monitor", UVM_NONE);
t.print(uvm_default_line_printer);

  case (data.op)
    4'b0000:
      if (data.ALU_Out == data.A + data.B)
        `uvm_info("SCO", "Test  of Addition Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Addition Failed", UVM_NONE)
    4'b0001:
      if (data.ALU_Out == data.A - data.B)
        `uvm_info("SCO", "Test of Subtraction Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Subtraction Failed", UVM_NONE)
     4'b0010:
       if (data.ALU_Out == data.A * data.B)
        `uvm_info("SCO", "Test of Multiplication Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Multiplication Failed", UVM_NONE)
     4'b0011:
       if (data.ALU_Out == data.A / data.B)
        `uvm_info("SCO", "Test of Division Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Division Failed", UVM_NONE)
     4'b0100:
      if (data.ALU_Out == data.A<<1)
        `uvm_info("SCO", "Test of shift left Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of shift left Failed", UVM_NONE)
     4'b0101:
      if (data.ALU_Out == data.A >>1)
        `uvm_info("SCO", "Test of shift right Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of shift right Failed", UVM_NONE)
     4'b0110:
        if (data.ALU_Out =={data.A[6:0],data.A[7]})
        `uvm_info("SCO", "Test of Rotate left Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Rotate left Failed", UVM_NONE)
     4'b0111:
        if (data.ALU_Out =={data.A[0],data.A[7:1]})
          `uvm_info("SCO", "Test of Rotate right Passed", UVM_NONE)
      else
        `uvm_info("SCO", "Test of Rotate rigt Failed", UVM_NONE)

    default:
      `uvm_info("SCO", "Unsupported ALU_Sel value", UVM_NONE)
  endcase


endfunction
```

Result of the test

Now, let's take a look at the results of the verification:
The test begins, data is generated, and sent to the ALU via a driver. The monitor captures the ALU's output, which is then compared by the scoreboard to ensure it matches the expected results. In this case, the test passes for the addition operation, confirming that the ALU functions correctly. These logs provide a step-by-step record of the verification process.

```
# KERNEL: UVM_INFO @ 0: reporter [RNTST] Running test ...
# KERNEL: UVM_INFO /home/runner/sequence.sv(17) @ 0: TEST.ENV.AGENT.SEQ@@GEN [GEN] Data send to Driver
# KERNEL: TRANS: (transaction@605) { A: 'he3  B: 'hd5  op: 'h1  ALU_Out: 'h0  begin_time: 0  depth: 'd2  parent sequ
# KERNEL: UVM_INFO /home/runner/driver.sv(25) @ 10: TEST.ENV.AGENT.DRV [DRV] Send data to DUT
# KERNEL: TRANS: (transaction@605) { A: 'he3  B: 'hd5  op: 'h1  ALU_Out: 'h0  begin_time: 0  depth: 'd2  parent sequ
# KERNEL: UVM_INFO /home/runner/monitor.sv(29) @ 30: TEST.ENV.AGENT.MON [MON] Send data to Scoreboard
# KERNEL: TRANS: (transaction@583) { A: 'h0  B: 'h0  op: 'h0  ALU_Out: 'h0  }
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(21) @ 30: TEST.ENV.SCO [SCO] Data rcvd from Monitor
# KERNEL: TRANS: (transaction@583) { A: 'h0  B: 'h0  op: 'h0  ALU_Out: 'h0  }
# KERNEL: UVM_INFO /home/runner/scoreboard.sv(27) @ 30: TEST.ENV.SCO [SCO] Test  of Addition Passed
```

linkedin.com/in/saad-kouzmane/

Thanks for reading!