May 18, 2015

Verification Plan and Report

Rupesh Basnet

Contents

1. Introduction 3
    1.1 Arithmetic Operations 3
    1.2 Logic Operations 3
    1.3 Shift Operations..4
    1.4 Scope 4
    1.5 Schedule 4
2. Verification Process.. 4
3. Verification Plan.. 5
    3.1 Stimulus Generation Plan 5
    3.2 Coverage.. 6
    3.3 Coverage Plan.. 6
4. Verification Environment 6
5. Verification Report 7
    5.1 Phase Completion Report 7
    5.2 Final Report 8
6. Conclusion 13
References

VERIFICATION PLAN and Report: PicoBlaze's ALU

Rupesh Basnet

1. Introduction

A simple PicoBlaze' ALU will be verified in this work using universal verification methodology (UVM) in systemverilog. PicoBlaze is an 8-bit RISC microcontroller. PicoBlaze's alu performs all microcontroller calculations including basic arithmetic operations such as addition and subtraction, bitwise logic operation such as AND, OR and XOR, shift and rotate operations, arithmetic compare and bitwise test operations etc. However in this work, the design is implemented with a simple ALU algorithm that performs arithmetic operations such as addition and subtraction, bitwise logic operation such as AND, OR and XOR and logical and arithmetic shift operations. So the work will mainly focus on verifying the functional correctness of the aforementioned functionality.

1.1 Arithmetic Operation

The PicoBlaze ALU provides basic byte-wide addition and subtraction instructions. The ALU provides two add instructions ADD and ADDCY that compute the sum of two 8-bit operands, either without or with carry. The first operand is a register location and the second operand can be either a register location or a literal constant. CARRY and ZERO flags indicate the status of the result. If the result is greater than 255, the CARRY flag is set and if the result is either 0 or 255, the ZERO flag is set. ADDCY instruction is an add operation with carry. If the carry flag is set, ADDCY adds an additional one to the resulting sum.

The PicoBlaze ALU provides two subtract instructions, SUB and SUBCY. SUB instruction computes the difference between two operands without carry and SUBCY instruction computes the difference between two numbers with carry (borrow). The CARRY flag indicates whether the subtract operation uses the borrow condition or not. The requirement with the operands is similar to ADD and ADDCY instructions. If the result is less than 0, then the CARRY flag is set. If the result is 0 or -256, then the ZERO flag is set. The SUBCY instruction is a subtract operation with borrow. If the CARRY flag is set, then SUBCY subtracts an additional one from the resulting difference.

1.2 Logic Operations

The logic instructions performs a bitwise logical AND, OR or XOR between two operands. The first operand is a register location and the second operand can be a register location or a literal constant. Figure 1 shows the corresponding bit locations in both operands are logically ANDed. If the resulting value is zero, the ZERO flag is set. The CARRY flag is always cleared by a logic instruction.

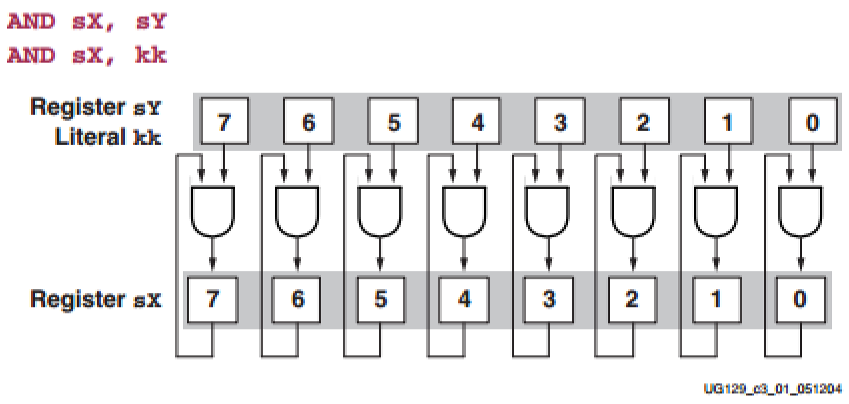


Figure 1: Bitwise AND instruction [1]

The OR and XOR instructions are similar to AND instruction illustrated in figure 1.

1.3 Shift Operations

The PicoBlaze ALU supports a rich set of shift instructions but here the focus will be on arithmetic shift right and left and logical shift right and left. All other shift instructions defined for PicoBlaze such as shift left and right with ‘0’ and ‘1’ fill, shift left and right extend bit 0, and shift left and right through all bits including carry is beyond the scope of this work.

1.4 Scope

The ALU will be implemented with basic arithmetic operations such as addition or subtraction, AND, OR and XOR logical operations and arithmetic and logical shift operations. All other operations such as arithmetic multiplication and division (most of the designs have dedicated hardware for this), logical operations such as NOT, NOR or XNOR, rotate operation will be beyond the scope of this work.

1.5 Schedule

Following are the steps will be followed to verify the PicoBlaze’s ALU.

- Understand PicoBlaze’s ALU Specification
- Develop the verification plan
- Build the verification environment using standard class library defined in UVM
- Develop an interface and connect it to Design Under Test (DUT) in top module
- Configuration classes will be developed so as to make the system easily reusable and quickly modifiable
- Develop the environment class and simple testcase and stimulate them. The environment will have sequencer, sequence, driver, before monitor, after monitor and a scoreboard. The UVM components will be connected to each other according to the requirement in the environment. Also the analysis communication will be used in order to establish communication between the agent and the scoreboard and the after monitor and the scoreboard. The before monitor will read the pin level transaction and translate them to the transaction level modelling (TLM) and send it to the scoreboard in order to calculate the expected result. The after monitor will read the DUT result and send it to the scoreboard for comparision.
- A transaction item class will be developed. This will be the transaction data that will be driven to DUT as input.
- The scoreboard will be extended from its base class uvm_scoreboard and the expected result will be compared against the DUT result sent by after monitor.

2. Verification Process

The complete verification of the DUT is achieved only after passing through several stages. At the very beginning an interface that the verification environment uses in order to drive test stimuli to the DUT should be created. System Verilog interface keyword will be used in order to create an interface that makes it possible for the verification environment to communicate with the DUT. The transaction item will be created extending uvm_sequence_item base class. The DUT has a 4 bit op code which has a possible combination of 16 values. So, the op code stimuli will be constrained to keep it in the range that has been defined while implementing the DUT in HDL.

A sequence class will be created extending uvm_sequence base class. This class will start the transaction item and randomize them such that the unexpected bugs could be located. A sequencer extended from uvm_sequencer will be used to make a co-ordination between the sequence and the driver. The driver will be extended from uvm_driver base class and is responsible for driving the test stimuli to the DUT through the interface. A monitor will be used in order to read the input stimuli to the DUT and to translate the pin level input into a TLM. The monitor will also forward the transaction read from the interface to the scoreboard for the comparison.

An active agent will encapsulate the sequencer, driver and the monitor. Another monitor called "after monitor" will be used to read the DUT output. This will also be sent to the scoreboard for the comparison. The scoreboard will take the input values from the before monitor and makes a calculation based on these input and produce an output. The DUT output sent by the after monitor will be compared against the calculated output. Every match or mismatch from the comparison will be reported uvm_info. The subscriber extended from uvm_subscriber will be implemented with covergroup in order to find the functional coverage.

The bins defined in the covergroup will be used to indicate to what extent the functional correctness is verified. Sample () method will procedurally sample the coverpoints defined in the covergroup. Moreover, get_coverage () method will be implemented with two reference arguments. The method will return the number of covered bins to the first argument and the total number of bins to the second argument. X% result means that out of 100 bins, only x number of bins are covered with the provided testcase.

3. Verification Plan

The design under test implements a simple ALU with basic arithmetic operation such as addition and subtraction, logical operations such as bitwise AND, OR and XOR, and logical and arithmetic shift operations. Following things need to be verified in order to verify the complete design's functional correctness.

1. Op code and Operation should match one another according to what is defined in the HDL implementation of the design. The HDL used in implementing the design is Verilog and the verification methodology is based on system Verilog.
2. ALU result will constantly be monitored by an uvm component called monitor. This monitor will translate the pin wiggling of DUT interface into TLM and sends it to the analysis component for the comparison against expected result. The scoreboard will be used as the analysis component that will compare the expected result and the actual result in order to verify the correctness of the result.
3. According to the HDL implementation of the design (ALU), all the op codes except the defined ones in HDL implementation produce a high impedance result. This should also be verified.
4. Verify if the reset pin is working or not.
5. Verify whether or not the carry flag is asserted correctly.
6. Verify the completeness of the transaction item to be driven to DUT i.e. verify the functional coverage.

3.1 Stimulus Generation Plan

The transaction item to be driven to DUT will randomly generated. The generated op code will be a 4 bit logic. Depending upon the bit combination, the DUT will decide which operation to carry out. Directed and constrained random testing can be exploited in order to address the corner cases. The transaction item class will be inherited from uvm_sequence_item base class. The transaction item class will have a 4 bit op code logic, two 8-bit operands, 3-bit shift_rotate logic such that at maximum, 8 bits will be shifted. 8-bit logic will be used in order to store the result of the ALU. The carry flag will be stored in carry logic as defined in the transaction item class.

3.2 Coverage

Coverage is a metric to measure the completeness of the verification. Functional Coverage will be used in order to verify the functional correctness of the system. Code coverage can also be used to get a measure of how many lines of code are executed, how many expressions and branches are executed. This is provided by the simulation tools. Besides these, assertion coverage can also be used to look for the desired behaviour in RTL. Assertion coverage is temporal in nature and has direct access to design variables. Assertion language constructs can be used for the assertion coverage.

3.2.1 Coverage Plan

Functional Coverage will be implemented in order to verify the functional correctness of the system. Covergroup will be defined in uvm_subscriber class. Coverpoint will be defined for the op code input that primarily drives the operation to be carried out in ALU. Coverage collection will be done procedurally with sample() method. The total number of cases and covered cases will be known using get_coverage() method. The code coverage is provided by the simulation tool.

4. Verification Environment

The verification language will be system Verilog. Universal Verification Methodology (UVM) will be used. The different uvm_components and uvm_objects will be exploited for the verification. Configuration object will be used wherever possible in order to make the system more reusable and easily modifiable. The verification tool will be Questa Sim and particularly for this work 10.0b version will be used. Figure 2 depicts the block diagram of the verification environment.

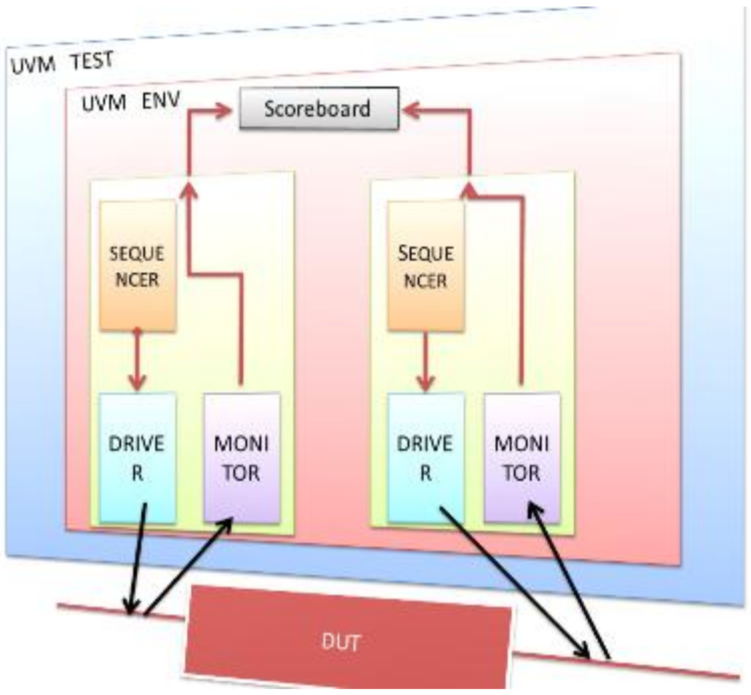


Figure 2: Verification environment block diagram [2]

5. Verification Reports

The verification reports are important metric to figure out the completion of different verification phases. The UVM components execute their behaviour in ordered and predefined phases. Figure 3 shows the hierarchy of UVM phases.

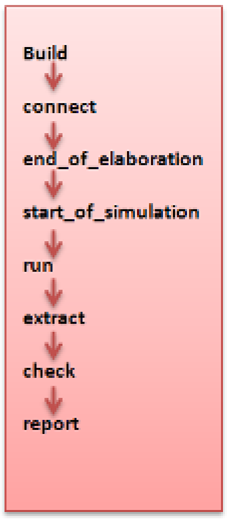
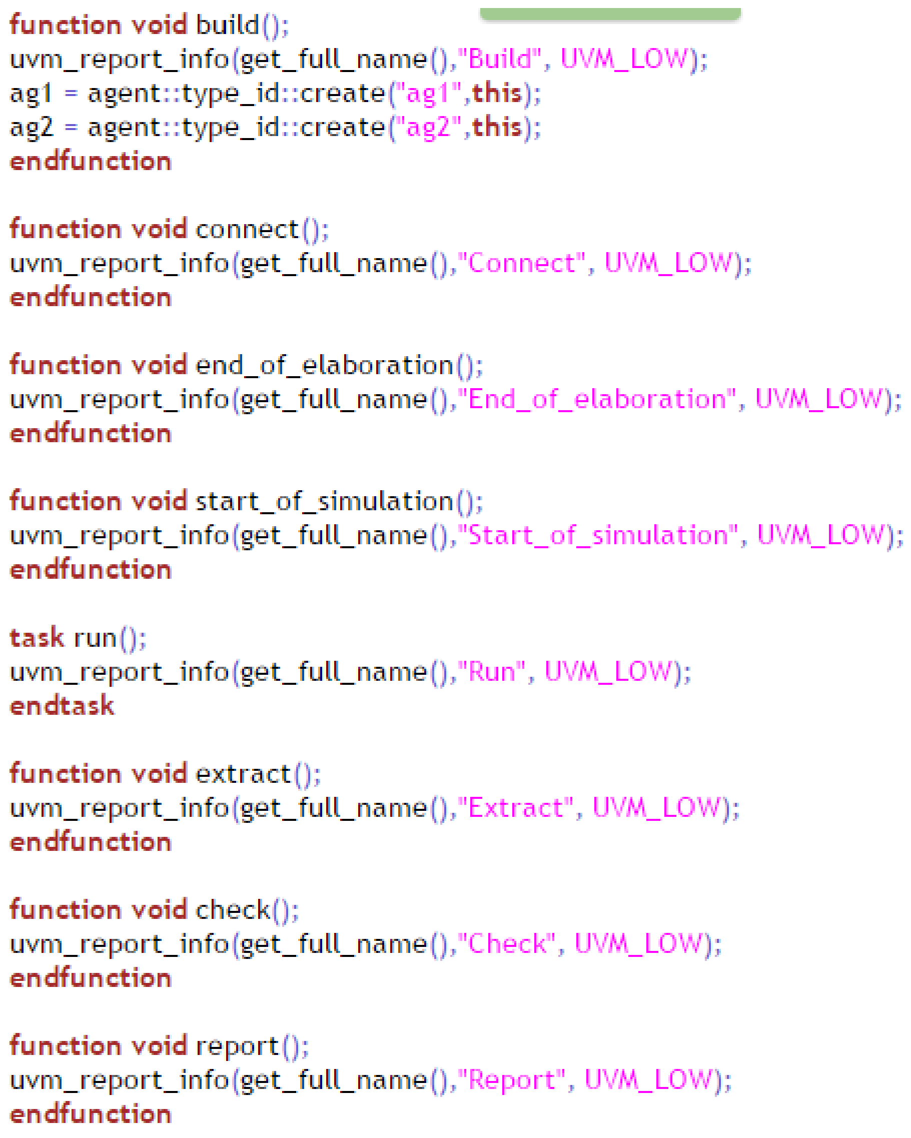


Figure 3: UVM phases [2]

The build() method is executed top to down. Parent build method is executed first and child objects build methods are executed. All other methods are executed bottom to up. The run method consumes much of the time and therefore the order in which this method is executed is undefined. [reference testbench.in]

5.1 Phase Completion Report

Each phase completion report can be printed to the transcript window of Questa simulator. This helps in debugging the execution flow in the verification environment and which phase is not executing. In this work, not any explicit message is printed in any phase completion. However, figure 4 is the screen capture from an online source that shows how the explicit messages can be printed in each phase to indicate the phase is executed.

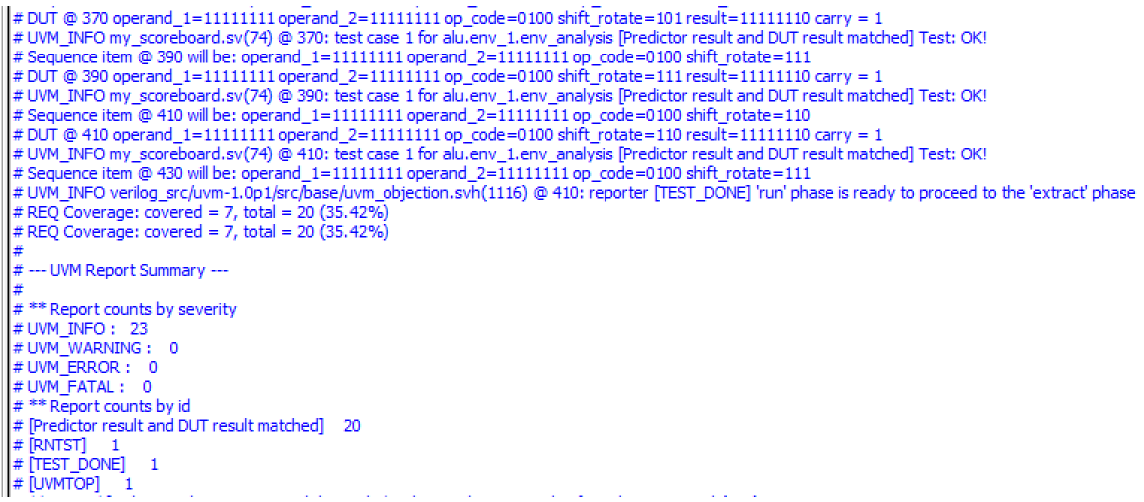


```
function void build();
uvm_report_info(get_full_name(),"Build", UVM_LOW);
ag1 = agent::type_id::create("ag1",this);
ag2 = agent::type_id::create("ag2",this);
endfunction

function void connect();
uvm_report_info(get_full_name(),"Connect", UVM_LOW);
endfunction

function void end_of_elaboration();
uvm_report_info(get_full_name(),"End_of_elaboration", UVM_LOW);
endfunction

function void start_of_simulation();
uvm_report_info(get_full_name(),"Start_of_simulation", UVM_LOW);
endfunction

task run();
uvm_report_info(get_full_name(),"Run", UVM_LOW);
endtask

function void extract();
uvm_report_info(get_full_name(),"Extract", UVM_LOW);
endfunction

function void check();
uvm_report_info(get_full_name(),"Check", UVM_LOW);
endfunction

function void report();
uvm_report_info(get_full_name(),"Report", UVM_LOW);
endfunction
```

Figure 4: ScreenCapture of UVM phases explicit message printing [2]

5.2 Final Report

The goals set in verification plan section 3 were verified step by step. The goal set in section 3.1 that the op codes should match the operation as per what is defined in the HDL implementation. It was verified that the op codes combination from 4’b0001 to 4’b1001 produced the bitwise AND, OR and XOR, addition, subtraction, and logical and arithmetic shift operations respectively. The DUT output result and the predictor calculation was compared in the scoreboard to verify the functional correctness of the DUT. Figure 5 shows the screen capture of the result obtained in the transcript window which verifies the goal set in 3.1 and 3.2.



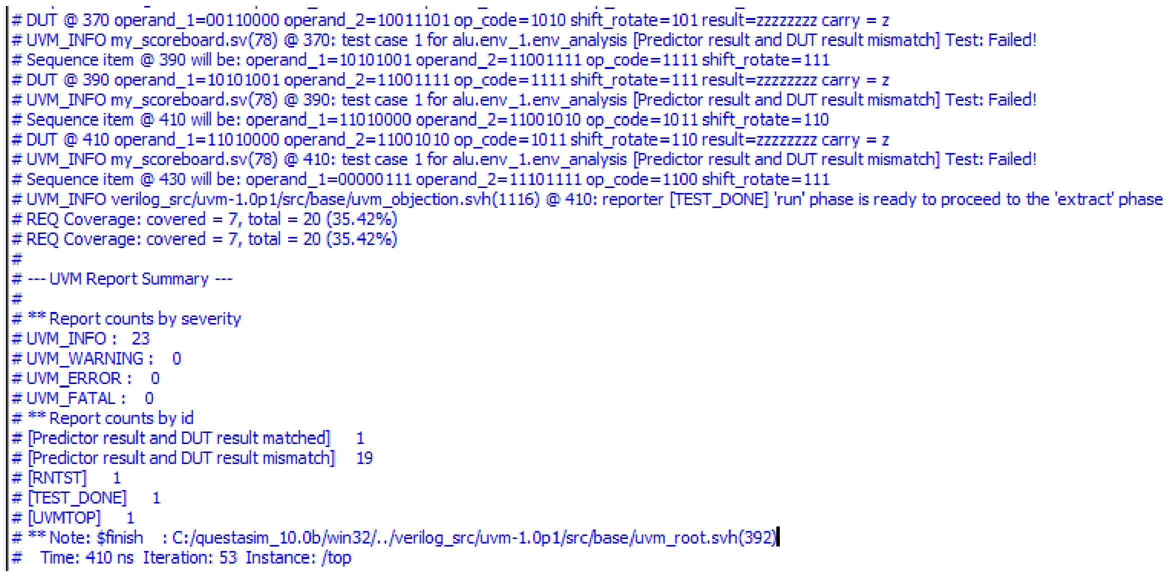
```
# DUT @ 370 operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=101 result=11111110 carry = 1
# UVM_INFO my_scoreboard.sv(74) @ 370: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 390 will be: operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=111
# DUT @ 390 operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=111 result=11111110 carry = 1
# UVM_INFO my_scoreboard.sv(74) @ 390: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 410 will be: operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=110
# DUT @ 410 operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=110 result=11111110 carry = 1
# UVM_INFO my_scoreboard.sv(74) @ 410: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 430 will be: operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=111
# UVM_INFO verilog_src/uvm-1.0p1/src/base/uvm_objection.svh(1116) @ 410: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# REQ Coverage: covered = 7, total = 20 (35.42%)
# REQ Coverage: covered = 7, total = 20 (35.42%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   23
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Predictor result and DUT result matched]    20
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
```

Figure 5: ScreenCapture of the obtained Output in Transcript window that verifies 3.1 and 3.2

The op code defined in the HDL implementation is a 4 bit logic vector. It has a possible combination of 16. Only values from 1 to 9 are defined and all undefined values greater than 9 should produce the high impedance result. This is verified constraining the op code input to generate values only greater than 9 in alu_transaction.sv file as below:

```
constraint opcode_range {
             op_code >= 10;
     }
```

The transcript window output capture is depicted in figure 6 that verifies 3.3.



```
# DUT @ 370 operand_1=00110000 operand_2=10011101 op_code=1010 shift_rotate=101 result=zzzzzzzz carry = z
# UVM_INFO my_scoreboard.sv(78) @ 370: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result mismatch] Test: Failed!
# Sequence item @ 390 will be: operand_1=10101001 operand_2=11001111 op_code=1111 shift_rotate=111
# DUT @ 390 operand_1=10101001 operand_2=11001111 op_code=1111 shift_rotate=111 result=zzzzzzzz carry = z
# UVM_INFO my_scoreboard.sv(78) @ 390: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result mismatch] Test: Failed!
# Sequence item @ 410 will be: operand_1=11010000 operand_2=11001010 op_code=1011 shift_rotate=110
# DUT @ 410 operand_1=11010000 operand_2=11001010 op_code=1011 shift_rotate=110 result=zzzzzzzz carry = z
# UVM_INFO my_scoreboard.sv(78) @ 410: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result mismatch] Test: Failed!
# Sequence item @ 430 will be: operand_1=00000111 operand_2=11101111 op_code=1100 shift_rotate=111
# UVM_INFO verilog_src/uvm-1.0p1/src/base/uvm_objection.svh(1116) @ 410: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# REQ Coverage: covered = 7, total = 20 (35.42%)
# REQ Coverage: covered = 7, total = 20 (35.42%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   23
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Predictor result and DUT result matched]    1
# [Predictor result and DUT result mismatch]    19
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# ** Note: $finish    : C:/questasim_10.0b/win32/../verilog_src/uvm-1.0p1/src/base/uvm_root.svh(392)
#    Time: 410 ns  Iteration: 53  Instance: /top
```

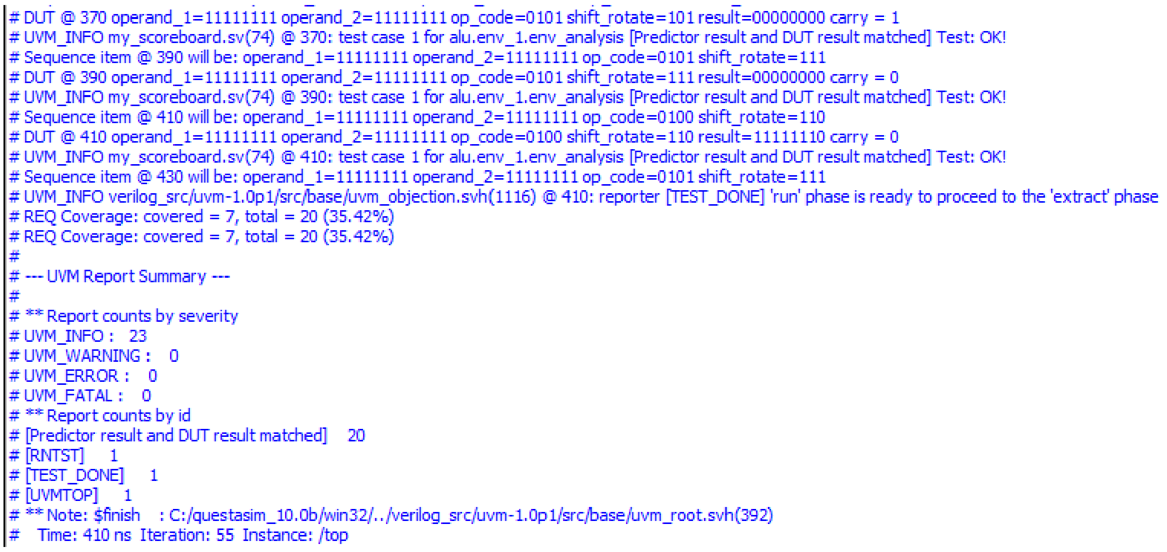
Figure 6: ScreenCapture showing high impedance result

The correct operation of reset pin is however not verified in this work. This was considered trivial for this work and hence left unverified. However in practical applications, reset is of significant importance and hence its functional correctness must be verified with high priority.

It was observed that with the regular randomized transaction, the verification environment was not able to verify the carry flag operation. The overflow condition when both the operands are 255 was not met and hence carry flag was never asserted high. For the first transaction the carry flag was in x state. This was because the input transaction items: op_code, operand_1, operand_2 and shift_rotate were assigned an initial value but the temp_result was not initialized. This register might have some garbage value and hence x was generated in carry flag. The corner case of operand_1 and operand_2 both being 255 and op_code being 4'b0100 should produce an overflow in the result. This case was achieved by constraining the operand_1, operand_2 and op_code inputs as below:

```
      constraint opcode_range {
            op_code >= 4'b0100;
            op_code <= 4'b0101;      }

      constraint opr1_range {
            operand_1 == 255; }

      constraint opr2_range {
            operand_2 == 255; }
```

The following figure 7 shows the transcript window output that verifies that the carry flag is asserted according to the specification.



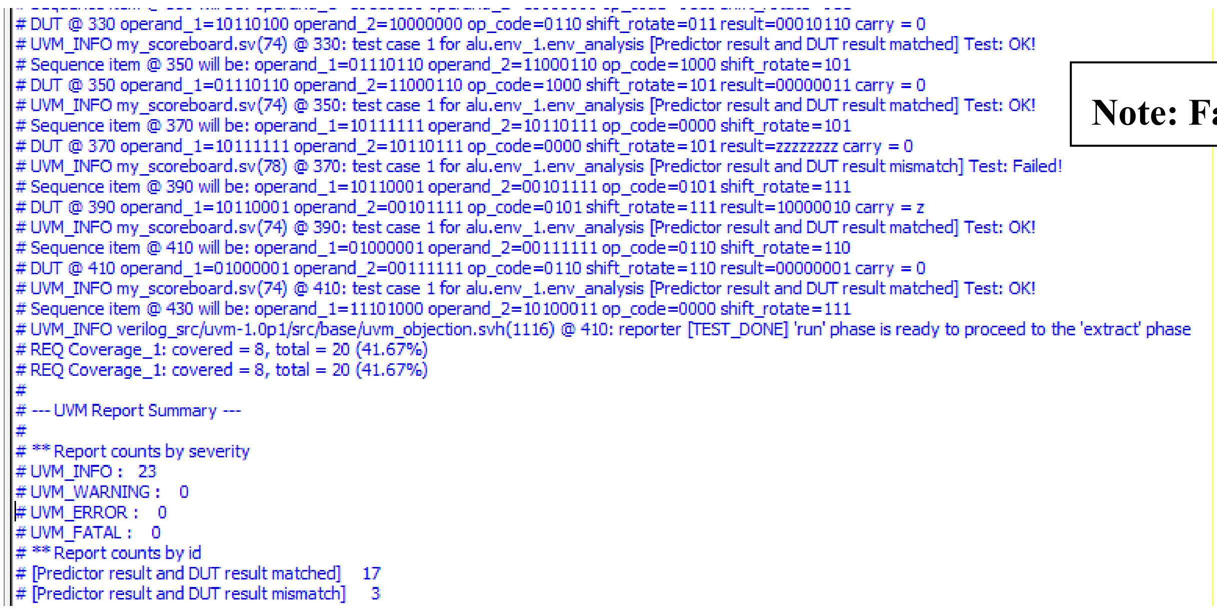
```
# DUT @ 370 operand_1=11111111 operand_2=11111111 op_code=0101 shift_rotate=101 result=00000000 carry = 1
# UVM_INFO my_scoreboard.sv(74) @ 370: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 390 will be: operand_1=11111111 operand_2=11111111 op_code=0101 shift_rotate=111
# DUT @ 390 operand_1=11111111 operand_2=11111111 op_code=0101 shift_rotate=111 result=00000000 carry = 0
# UVM_INFO my_scoreboard.sv(74) @ 390: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 410 will be: operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=110
# DUT @ 410 operand_1=11111111 operand_2=11111111 op_code=0100 shift_rotate=110 result=11111110 carry = 0
# UVM_INFO my_scoreboard.sv(74) @ 410: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 430 will be: operand_1=11111111 operand_2=11111111 op_code=0101 shift_rotate=111
# UVM_INFO verilog_src/uvm-1.0p1/src/base/uvm_objection.svh(1116) @ 410: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# REQ Coverage: covered = 7, total = 20 (35.42%)
# REQ Coverage: covered = 7, total = 20 (35.42%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   23
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Predictor result and DUT result matched]    20
# [RNTST]     1
# [TEST_DONE]     1
# [UVMTOP]     1
# ** Note: $finish    : C:/questasim_10.0b/win32/../verilog_src/uvm-1.0p1/src/base/uvm_root.svh(392)
#    Time: 410 ns  Iteration: 55  Instance: /top
```

Figure 7: ScreenCapture that verifies carry flag operation

The functional correctness is ensured only when there is 100% functional coverage and 100% code coverage. In this work, the number of transaction item that will be driven to the DUT is set to be 20 in the test class. The coverage collector was implemented with an explicit covergroup that had 4 coverpoints. The coverpoint op_code has 16 possible bins, operand_1 and operand_2 has 256 possible bins for each and shift_rotate has 8 possible bins. The cross coverage among op_code, operand_1 and operand_2 will generate 16*256*256 = 1048576 bins and the cross coverage among op_code, operand_1 and shift_rotate will generate 16*256*8 = 32768 bins. These all combinations were not however covered. Cross coverage is beyond the scope of this work. A simple coverage collector was implemented as:

```
covergroup int_coverage;
    op_code: coverpoint opc
      //{ bins op_c [9] = {[1:9]};}
      { bins low  = {[1:5]};
        bins high= {[6:9]};}
        //ignore_bins rest = {[10:15]};}
    operand_1: coverpoint op1

      //{ bins op_1[49] = {[1:49]};}
      { bins low  = {0};
        bins med  = {[1:254]};
        bins high = {255};}
    operand_2: coverpoint op2
      //{ bins op_2[49]= {[1:49]};}
        { bins low= {0};
          bins med  = {[1:254]};
          bins high= {255};}
    shift_rotate: coverpoint shift
      //{ bins shi[7]= {[0:7]};}
      { bins low = {[0:3]};
        bins high = {[4:7]};}

    //cross_1: cross opc,op1,op2;
    //cross_2: cross opc, op1,shift;

  endgroup: int_coverage

function void write(alu_transaction t);

    opc = t.op_code;
    op1 = t.operand_1;
    op2 = t.operand_2;
    shift = t.shift_rotate;

//coverage collection procedurally, this method can be alternatively used
//as sensitive to event
    int_coverage.sample();


  endfunction

  function void report_phase( uvm_phase phase );
      pct_1 = int_coverage.get_coverage(covered_1, total_1);
    //pct_2 = int_coverage.cross_2.get_coverage(covered_2, total_2);
      $display("REQ  Coverage_1:  covered  =  %0d,  total  =  %0d  (%5.2f%%)",
      covered_1, total_1, pct_1);
   // $display("REQ Coverage_2: covered = %0d, total = %0d (%5.2f%%)",
      covered_2, total_2, pct_2);

  endfunction
```

The following screen capture of the transcript window shows the obtained coverage report.



```
# DUT @ 330 operand_1=10110100 operand_2=10000000 op_code=0110 shift_rotate=011 result=00010110 carry = 0
# UVM_INFO my_scoreboard.sv(74) @ 330: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 350 will be: operand_1=01110110 operand_2=11000110 op_code=1000 shift_rotate=101
# DUT @ 350 operand_1=01110110 operand_2=11000110 op_code=1000 shift_rotate=101 result=00000011 carry = 0
# UVM_INFO my_scoreboard.sv(74) @ 350: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 370 will be: operand_1=10111111 operand_2=10110111 op_code=0000 shift_rotate=101
# DUT @ 370 operand_1=10111111 operand_2=10110111 op_code=0000 shift_rotate=101 result=zzzzzzzz carry = 0
# UVM_INFO my_scoreboard.sv(78) @ 370: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result mismatch] Test: Failed!
# Sequence item @ 390 will be: operand_1=10110001 operand_2=00101111 op_code=0101 shift_rotate=111
# DUT @ 390 operand_1=10110001 operand_2=00101111 op_code=0101 shift_rotate=111 result=10000010 carry = z
# UVM_INFO my_scoreboard.sv(74) @ 390: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 410 will be: operand_1=01000001 operand_2=00111111 op_code=0110 shift_rotate=110
# DUT @ 410 operand_1=01000001 operand_2=00111111 op_code=0110 shift_rotate=110 result=00000001 carry = 0
# UVM_INFO my_scoreboard.sv(74) @ 410: test case 1 for alu.env_1.env_analysis [Predictor result and DUT result matched] Test: OK!
# Sequence item @ 430 will be: operand_1=11101000 operand_2=10100011 op_code=0000 shift_rotate=111
# UVM_INFO verilog_src/uvm-1.0p1/src/base/uvm_objection.svh(1116) @ 410: reporter [TEST_DONE] 'run' phase is ready to proceed to the 'extract' phase
# REQ Coverage_1: covered = 8, total = 20 (41.67%)
# REQ Coverage_1: covered = 8, total = 20 (41.67%)
#
# --- UVM Report Summary ---
#
# ** Report counts by severity
# UVM_INFO :   23
# UVM_WARNING :    0
# UVM_ERROR :    0
# UVM_FATAL :    0
# ** Report counts by id
# [Predictor result and DUT result matched]    17
# [Predictor result and DUT result mismatch]    3
```

Note: Failed

Figure 8: ScreeCapture of Transcript window showing coverage result

The total number of bins was 20 and only 8 bins were covered yielding 41.6% of coverage. The test failed as shown in figure 8 shows that the compare function cannot evaluate the result being high impedance and hence fails the compare test. In this work, the DUT is not tested with all the possible values of input but few. On average 70% of the result comparison matched. About 30% of the result comparison mismatched because of the result being high impedance. In this way, it can be considered that 100% of functional correctness is verified within the range of the values in the bins. In order to check the values in the bin, transcript file can be used. However the following extract from the transcript file shows the test data used to verify the functional correctness of the DUT.

Table 1: Test data driven to DUT

Time_instant	Op_Code	Operand_1	Operand_2	Shift_rotate
@30	0001	00000000	00000000	000
@50	0010	10101100	01001010	110
@70	0111	00000011	10101000	101
@90	0100	11110100	00101110	110
@110	0111	10010111	10100010	011
@130	0100	00001100	11010010	100
@150	0101	11001001	11011100	110
@170	0000	10011100	11110000	101
@190	1001	11011100	01000000	011
@210	0000	01011001	10011110	100
@230	1001	01110101	00001011	001
@250	0010	01011000	10001000	010
@270	0010	01000111	10000011	001
@290	0001	10100010	00001100	101
@310	0010	01001001	10010011	011
@330	0110	10110100	10000000	011
@350	1000	01110110	11000110	101
@370	0000	10111111	10110111	101
@390	0101	10110001	00101111	111
@410	0110	01000001	00111111	110

6. Conclusion

The simple PicoBlaze's ALU was verified with the constraints defined. The goals set for this work as in section 3 were achieved except for the reset case which was not verified for the reason described above. The ALU was verified with 100% functional correctness for the range of values covered by the coverpoints.

References

1. http://www.xilinx.com/support/documentation/ip_documentation/ug129.pdf4. [Online]
2. http://www.testbench.in/UT_02_UVM_TESTBENCH.html [Online]