UVM Verification Environment of the Open Cores 8051 ALU

Omar Tarek Ahmed Amer	V23009946
Hussein Shedeed	V23010209
Nermeen Alaa	V23010319
Hisham Tarek	V23010326
Mennatullah Mohamed	V23010369
Lobna Ahmed	V23010502 [Section 18]

Contents

List of Figures

List of Tables

Introduction

This document is to show our final submission for CND212: A UVM testbench for the Open Cores [OC] 8051 ALU. The 8051 is an 8-bit microcontroller designed by Intel in 1981 for use in embedded systems. OC implemented this microcontroller in Verilog and the final project for CND212 is to build a UVM environment to test the ALU of this microcontroller.

This ALU can do 16 operations, which enable the microcontroller to perform all 74 instructions with different addressing modes. Table 1 shows all possible ALU opcodes.

Table 1 ALU Operations

Mnemonic	Description
NOP	**N**o **Op**eration
ADD	**ADD** *op1+op2* with carry.
SUB	**SUB** *op1-op2* with borrow.
MUL	**MUL**tiply *op1 * op2*
DIV	**Div**ided *op1/op2*
DA	**D**ecimal **A**djust *op1*
NOT	Bitwise **NOT** *op1*
AND	Bitwise **AND** *op1 & op2*
XOR	Bitwise **XOR** *op1 ^ op2*
OR	Bitwise **OR** *op1 | op2*
RL	**R**otate **L**eft *op1*
RLC	**R**otate **L**eft *op1* through **C**arry
RR	**R**otate **R**ight *op1*
RRC	**R**otate **R**ight *op1* through **C**arry
INC	**INC**rement (decrement) *op1*
XCH	e**XCH**ange *op1 <-> op2* (bytes or nibbles)

The ALU has 3 output flags: **C**arry, **A**uxiliary **C**arry, and **OV**erflow and is fully combinational except for the multiplication and division, as they are performed on separate modules.

All but two of the above operations are single cycle operations. The **MUL** and **DIV** require 4 clock cycles to obtain a valid result.

Some operations have different behaviors regarding the values *bit_in, srcCy,* or *srcAc*. One notable example is the **XCH** instruction, which either exchanges bytes or nibbles according to the *srcCy* input.

Verification Plan

Each operation is independent on the one before as well as the one after it. By applying constrained random transactions at the ALU's inputs and observing the output and by covering all possible input combinations we can verify that the ALU is functional. Follows is the plan we followed to

test each operation. The testing of each operation is as follows: present an input to the DUT, and compare the output based on the specifications of the 8051 instruction set (Brouwer, 2002). Along with the EDSim51 8051 emulator. One discrepancy was found and reported regarding the carry output of the decimal adjust (**DA**) instruction. No other discrepancies were found when comparing the OC8051 ALU with the instruction set specs or with the emulator.

Sequence items have been made with all possible input combinations and were arbitrated to the agent using a sequence. This method enabled us to get 100% functional coverage.

Testbench Architecture

The testbench architecture can be seen on the next page. We will discuss each part of it in the upcoming sections.

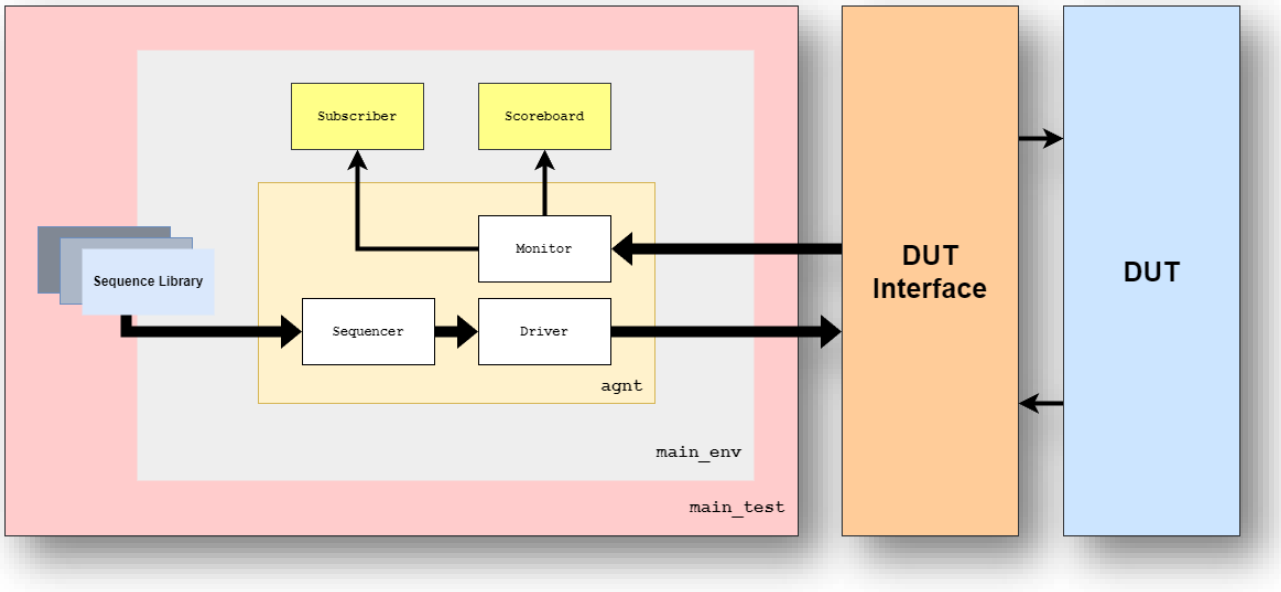


Figure 1 Testbench architecture.

DUT Interface

The interface has two clocking blocks: **drv** and **mon**. This is to make the code cleaner such that the clocking block that drives stimulus doesn't have access to the outputs of the ALU, and the monitoring clocking block has access to all signals (inputs and outputs) to monitor both stimulus and outputs. Sampling happens at *#1step* before the clock edge and outputting happens at the negative edge of the clock.

Furthermore, the interface has two tasks: *reset_alu* which resets the multiplier and divider, and *init_inputs* to avoid having x's at the start of the simulation.

All signals are of type logic except for the opcode. Making the opcode as an enumeration helped testing significantly. A package has been made with the typedef of the opcode. A module wrapper has been made to combine the given Verilog file of the ALU and the new SystemVerilog interface.

Table 2 Interface Specification

Signal Name	Width	Direction	Description
`clk`	1	Input	Positive edge clock.
`rst`	1		Active high async reset.
`op_code`	4		Enumerated type, Operation code.
`src1`	8		Operand 1
`src2`	8		Operand 2
`src3`	8		Operand 3
`srcCy`	1		Input carry
`srcAc`	1		Input auxiliary carry.
`bit_in`	1		Input bit. Required for some operations that operate on bits.
`des1`	8	Output	Destination 1
`des2`	8		Destination 2
`des_acc`	8		To accumulator.
`sub_res`	8		Subtraction result.
`desCy`	1		Output carry.
`desAc`	1		Output auxiliary carry.
`desOv`	1		Output overflow (or division divide by zero).

Sequences

The sequences in this project are simply operations with different opcodes. A class was derived from *uvm_sequence_item* for each type of transaction. Which would have been a lot of boilerplate code, but a macro has been used such that each class is declared in one line. All sequences have been grouped in a package that also includes the sequence library class, which was required to arbitrate between all sequence types and to simulate an actual real-life environment the ALU could be placed in.

```
`RAND_SEQ_CLASS(rst_seq, txn_to_send.rst == 1)
`RAND_SEQ_CLASS(nop_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == NOP))
`RAND_SEQ_CLASS(add_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == ADD))
`RAND_SEQ_CLASS(sub_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == SUB))
`RAND_SEQ_CLASS(mul_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == MUL))
`RAND_SEQ_CLASS(div_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == DIV))
`RAND_SEQ_CLASS(da_seq,  (txn_to_send.rst == 0) && (txn_to_send.op_code == DA))
`RAND_SEQ_CLASS(not_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == NOT))
`RAND_SEQ_CLASS(and_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == AND))
`RAND_SEQ_CLASS(xor_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == XOR))
`RAND_SEQ_CLASS(or_seq,  (txn_to_send.rst == 0) && (txn_to_send.op_code == OR))
`RAND_SEQ_CLASS(rl_seq,  (txn_to_send.rst == 0) && (txn_to_send.op_code == RL))
`RAND_SEQ_CLASS(rlc_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == RLC))
`RAND_SEQ_CLASS(rr_seq,  (txn_to_send.rst == 0) && (txn_to_send.op_code == RR))
`RAND_SEQ_CLASS(rrc_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == RRC))
`RAND_SEQ_CLASS(inc_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == INC))
`RAND_SEQ_CLASS(xch_seq, (txn_to_send.rst == 0) && (txn_to_send.op_code == XCH))
```

Figure 2 Macros for sequence items.

Sequencer

The standard *uvm_sequencer* was used. No extensions were made.

Driver

The driver converts the transactions into “pin wiggles” at the DUT interface.

Single Cycle Operations

For the single cycle operations, the DUT holds the data for a single clock cycle at the DUT inputs.

Multi Cycle Operations

For the multi cycle operations (MUL and DIV), the DUT holds the data for as long as it is required to acquire a valid result at the DUT outputs which is the same amount for both MUL and DIV (4 clock cycles).

Reset

The ALU reset is an asynchronous active high reset. When the driver encounters a reset sequence, it sets the operation to NOP and calls the *reset_alu task* from the interface.

Monitor

At each event of the monitoring clocking block, the monitor sends a transaction, which is a sequence item with data members the hold the value of every signal in the interface, to the scoreboard as well as to the subscriber which collects coverage.

Agent

The Agent holds the monitor, driver, and sequencer in one container.

Scoreboard

The scoreboard reads a transaction from the monitor via an analysis import. When it receives a transaction, it calls the necessary comparison functions to make sure that the result of the ALU is correct.

Table 3 Notable Comparison Functions

Function	Description
check_mul	Checks two different results, the HI and LO part of the multiplication.
check_div	Checks two different results, the remainder and quotient of the division.
check_da	Converts a number to its correct BCD representation.
check_xch	Exchanges bytes or nibbles between op1 and op2 based on the input carry.

All these functions, and all other checking functions are in a single package to facilitate coding and enhance code maintainability and readability.

Subscriber

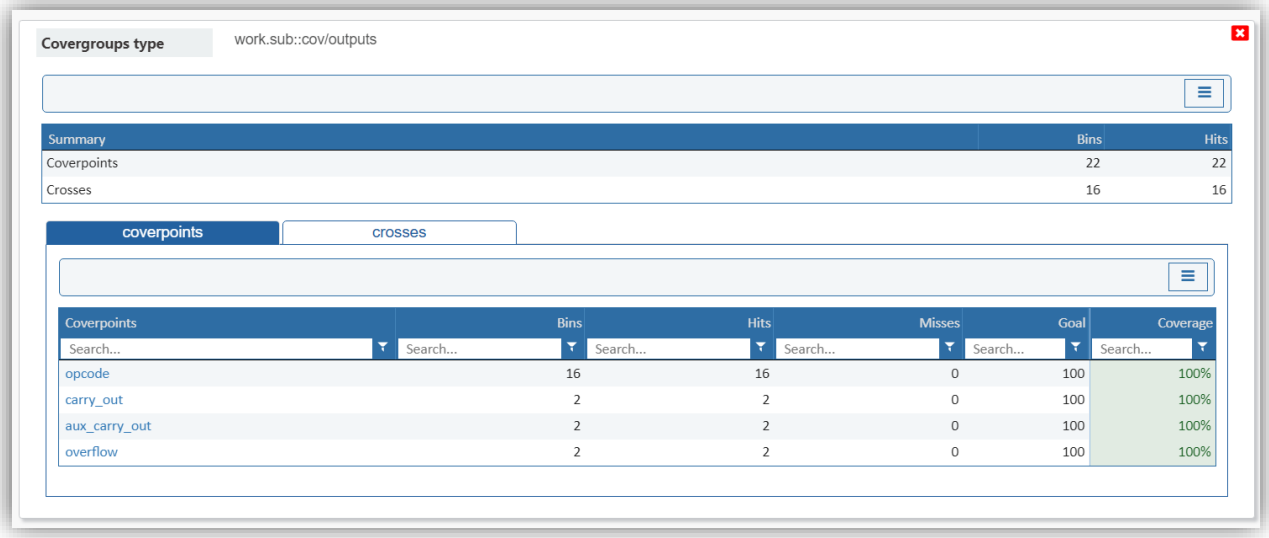
The subscriber works exactly as the scoreboard in the sense that it receives a transaction from the monitor. The only difference is that the subscriber we have is required to collect coverage. We had the option to collect coverage at the monitor, but we decided to abide by the single responsibility principle and keep the monitor strictly for monitoring and make another component that is strictly for coverage collection.

Functional Coverage Analysis

After applying completely random stimulus to our DUT and checking the validity of the results, we had to make sure that our testbench is generating sufficient stimulus to call our testbench a success. For this reason, we have made several cover points and cover groups:

Table 4 Covergroups and Coverpoints

Covergroup	Coverpoint	Description
operations	rst	Cover resets.
	carry_in	Cover carry-in.
	bit_in	Cover bit-in
	all_opcode_variants	Cross cover opcode with carry-in, and bit-in.
outputs	carry_out	Cover carry-out
	aux_carry_out	Cover auxiliary carry out
	overflow	Cover overflow
	all_output_results	Cross all opcodes with all valid flags.
multicycle_ops	consecutive_multicycle_ops	Cover two multicycle operations in a row.



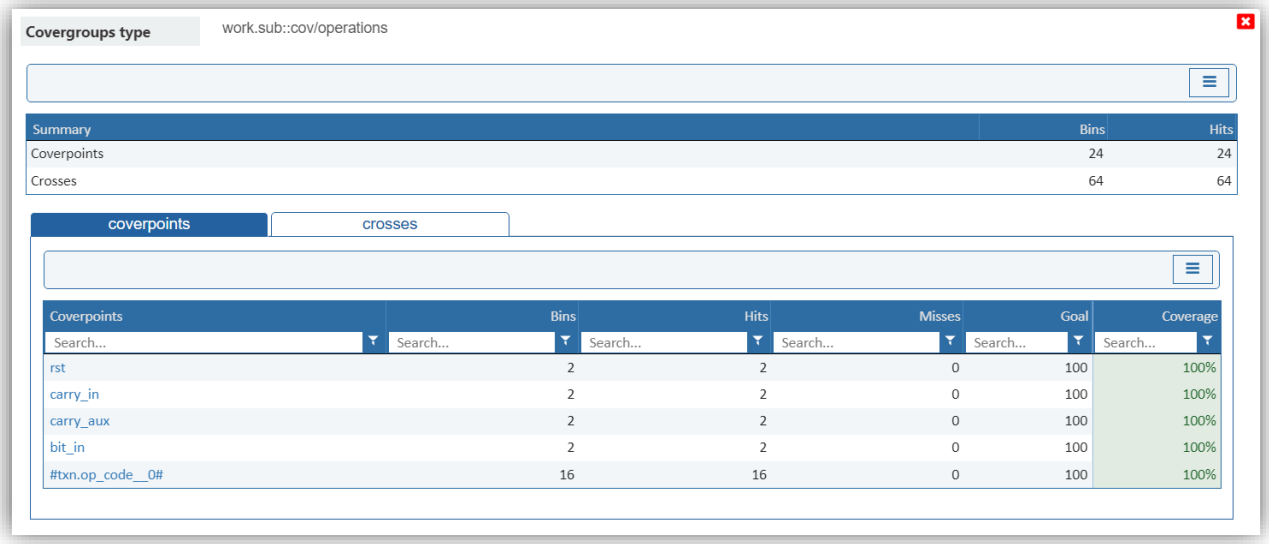
Covergroups type work.sub::cov/outputs

Summary	Bins	Hits
Coverpoints	22	22
Crosses	16	16

coverpoints | crosses

Coverpoints	Bins	Hits	Misses	Goal	Coverage
opcode	16	16	0	100	100%
carry_out	2	2	0	100	100%
aux_carry_out	2	2	0	100	100%
overflow	2	2	0	100	100%

Figure 3 Outputs Coverage



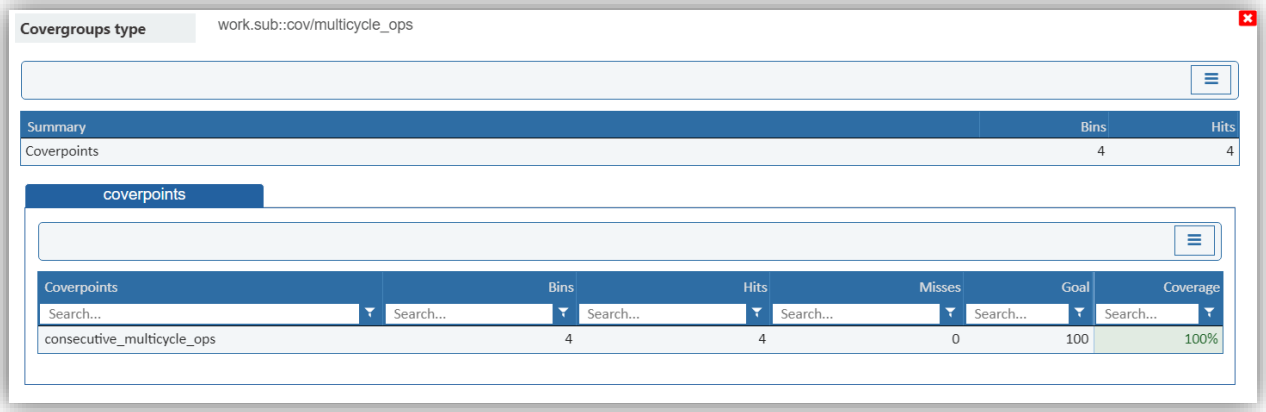
Covergroups type work.sub::cov/operations

Summary	Bins	Hits
Coverpoints	24	24
Crosses	64	64

coverpoints | crosses

Coverpoints	Bins	Hits	Misses	Goal	Coverage
rst	2	2	0	100	100%
carry_in	2	2	0	100	100%
carry_aux	2	2	0	100	100%
bit_in	2	2	0	100	100%
#txn.op_code__0#	16	16	0	100	100%

Figure 4 Operations Coverage



Covergroups type	work.sub::cov/multicycle_ops

Summary	Bins	Hits
Coverpoints	4	4

coverpoints

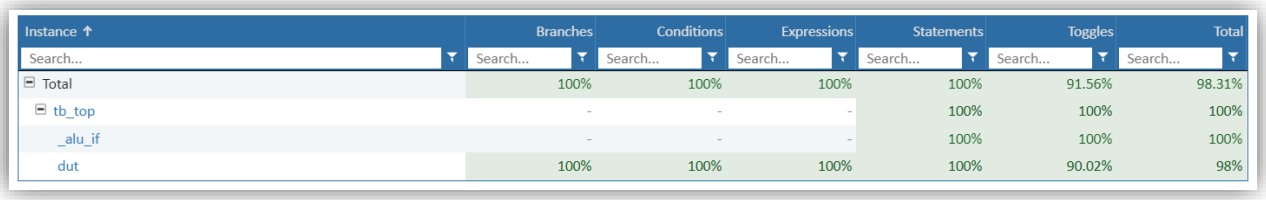
Coverpoints	Bins	Hits	Misses	Goal	Coverage
consecutive_multicycle_ops	4	4	0	100	100%

Figure 5 Multicycle Operations

As we can see, all functional coverage is at 100%. This means that our testbench works as intended and all our test cases are indeed comprehensive.

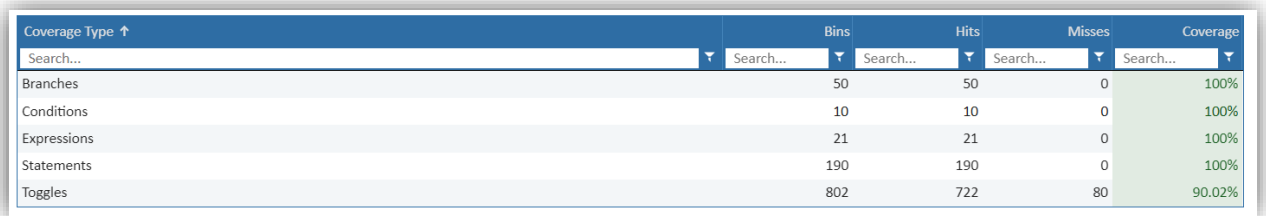
Code Coverage

The code coverage for the full environment is 98.31%. The full code coverage report can be found in the *cov_rep* directory. Open *index.html* with your browser.



Instance	Branches	Conditions	Expressions	Statements	Toggles	Total
Total	100%	100%	100%	100%	91.56%	98.31%
tb_top	-	-	-	100%	100%	100%
_alu_if	-	-	-	100%	100%	100%
dut	100%	100%	100%	100%	90.02%	98%

Figure 6 Code Coverage Summary

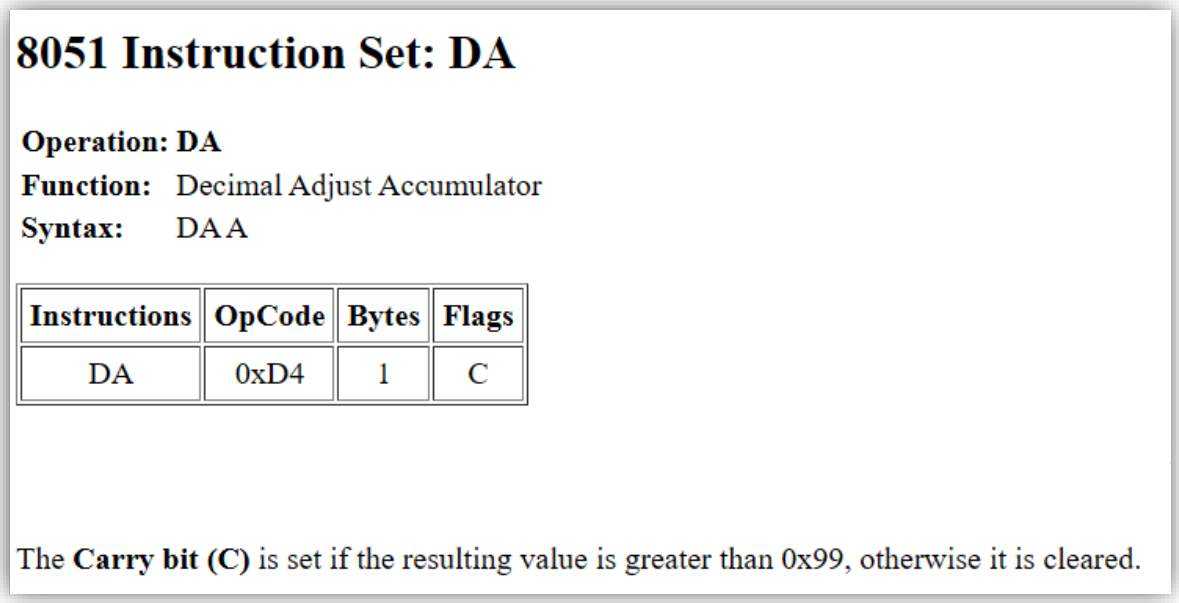


Coverage Type	Bins	Hits	Misses	Coverage
Branches	50	50	0	100%
Conditions	10	10	0	100%
Expressions	21	21	0	100%
Statements	190	190	0	100%
Toggles	802	722	80	90.02%

Figure 7 DUT Code Coverage

Simulation Results

One bug was found in the ALU. Consider Figure 8.



8051 Instruction Set: DA

Operation: DA
Function: Decimal Adjust Accumulator
Syntax: DA A

Instructions	OpCode	Bytes	Flags
DA	0xD4	1	C

The **Carry bit (C)** is set if the resulting value is greater than 0x99, otherwise it is cleared.

Figure 8 Correct DA behavior.

The DA instruction sets the carry bit if the result is greater than 0x99. This behavior was not observed in the design and thus declared an error. It is also declared *expected_2_fail*.

```
# ----------------------------------------
# Name          Type          Size  Value
# ----------------------------------------
# seq_item_mon  seq_item_mon  -     @104775
#   op_code     opcode_e      4     DA
#   src1        integral      8     'h2d
#   src2        integral      8     'h2d
#   src3        integral      8     'h24
#   srcAc       integral      1     'b0
#   srcCy       integral      1     'b0
#   bit_in      integral      1     'b1
#   rst         integral      1     'b0
#   des1        integral      8     'h2d
#   des2        integral      8     'h0
#   des_acc     integral      8     'h93
#   sub_result  integral      8     'h0
#   desCy       integral      1     'b1
#   desAc       integral      1     'b0
#   desOv       integral      1     'b0
# ----------------------------------------
# UVM_ERROR ./ver/checker_functions_pkg.sv(128) @ 813450: reporter [expected_2_fail FAIL DA CARRY] BAD CARRY
```

Figure 9 Incorrect behavior from the OC8051 ALU. [Simulation Result]

Here we see *desCy* is smaller than 0x99, but the carry flag is set. This was extracted from the simulation log. We mark this as *expected_2_fail* and continue testing. Any other case that fails is not expected and thus stops the simulation with a UVM_FATAL macro.

Conclusion

In conclusion, the UVM testbench successfully verified the functionality of the Open Cores 8051 ALU. The testbench achieved 100% functional coverage, ensuring that all possible ALU operations were tested under various conditions. Additionally, code coverage reached 98.31%, indicating that most of the design code was exercised during the simulation.

One bug related to the carry bit behavior of the DA instruction was identified and reported. The testbench effectively distinguished between expected and unexpected failures, allowing for efficient debugging.

Overall, the UVM testbench served as a comprehensive verification environment for the Open Cores 8051 ALU, guaranteeing its correctness and reliability.