ECE 745 PROJECT REPORT

VERIFICATION OF PIPELINED LC3 MICROCONTROLLER

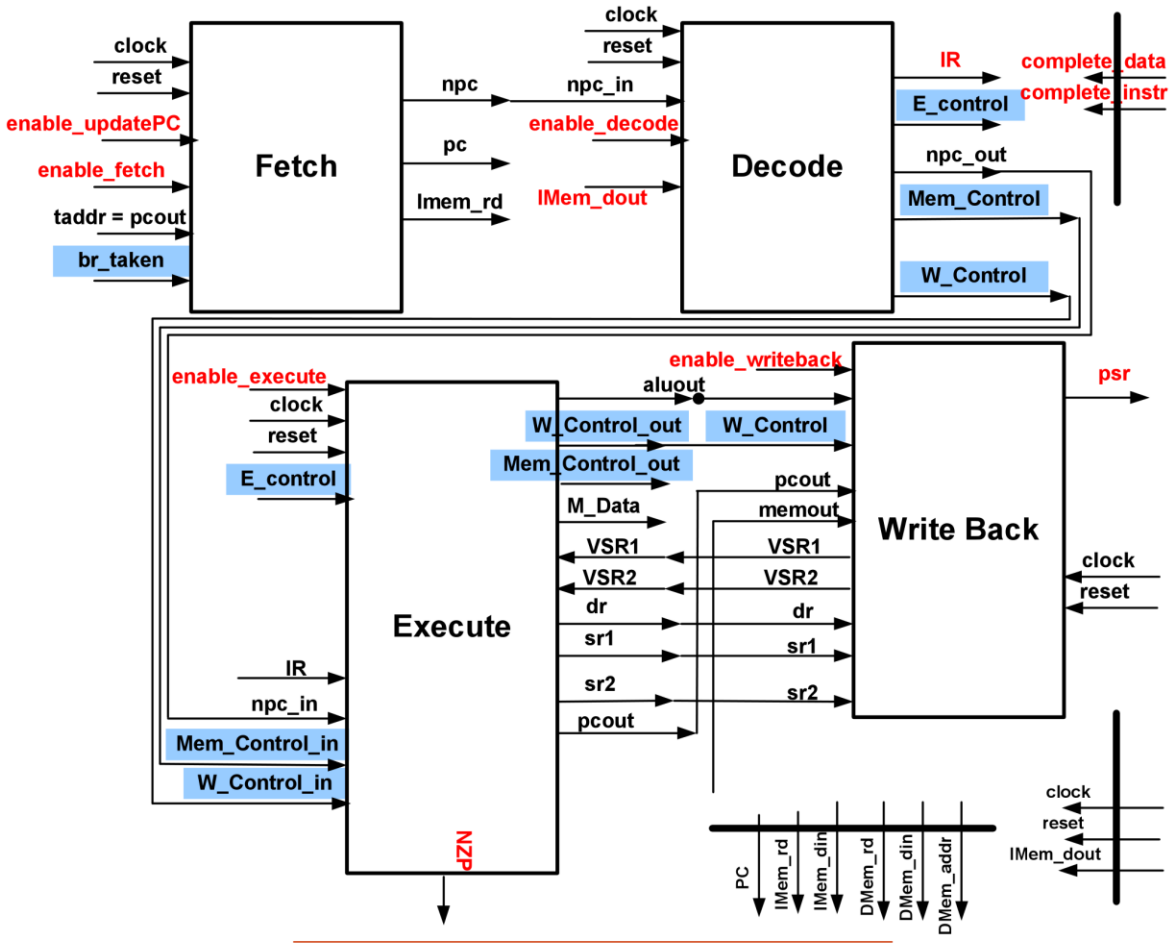
SUBMITTED BY: GROUP 5

1) **AMARESH SUBBURAJ**
2) **KAUSHIK MANIKANDAN PARASURAMAN**
3) **ADITYA KUMARAN KRISHNAPRAKASH**
4) **ABHISHEK KULHARI**

INTRODUCTION:

This project deals with the data and control path verification of the pipelined LC3 Microcontroller with a comprehensive instruction set. Each instruction in the LC3 microcontroller goes through 5 cycles:

Fetch->Decode->Execute->Writeback->UpdatePC.



The LC3 uses a 16-bit addressable memory with a 2^{16} location address space. The register file contains eight general purpose registers. Each instruction is 16 bits wide and has a 4-bit opcode. The instruction set defines 15 instructions which involve ALU, Control and Memory operations.

TESTING METHODOLOGY:

The testing of the microcontroller is based on a layered testbench which verifies the various internal stages at the lowest level, and the interactions of these internal stages at a higher level. We start with Constrained Random Testing approach to achieve a broad initial verification base and then narrow down on the specific corner cases with Directed Testing.

- For each internal stage (FETCH, DECODE, EXECUTE, MEM ACCESS, WRITEBACK), a Golden Reference model is created and checked with the actual output from the DUT. This helps in verifying the functionality of the stage. This model is based on the specifications of the LC3's functioning.
- Class Generator along with class Transaction is used to form the instructions for the DUT. The class Transaction contains the constraints for each instruction generated by class Generator. These constrained random instructions try to simulate most of the valid operations of the LC3.
- The class Driver is used to drive the other modules with the data. It applies the reset and runs the testbench along with the task of synchronizing the Generator and Transaction module using mailboxes.
- The class Controller is responsible for the control operations of the LC3. The state of the LC3 based on previous operations and the sequence of operations that should be executed by the DUT based on the opcode of each instruction is controlled and checked with a Golden Reference using this class.
- The coverage of the verification is handled by the class cover_group. To test whether each instruction and valid state of the DUT, various cover points are defined here.
- The interfaces for different modules as well as the coverage properties for each internal stage of the LC3 is defined in the class Project_interfaces.
- Finally, the module Testbench_top is used to execute the above classes.

These are the following classes that we have used in the project:

1. **Transaction class**: In this class we defined the constraints that are required for the constrained random testing of the module. There is no such constraint between a memory and branch instruction. We wrote a separate task to generate a complete instruction by combining opcodes, registers used for the operation, immediate mode in case of ALU operations etc.; according to the ISA of LC3 microcontroller.

2. **Generator Class**: We instantiate a new object of transaction class in the generator and randomize it. The randomized transaction object is then placed in a mailbox. After placing the transaction in the mailbox, the generator class waits for a trigger before generating a new transaction.

3. **Driver Class**: The driver class gets a transaction from the mailbox, after which it triggers the event for which the generator class waits. The driver then drives the DUT signals such as instr_dout and Data_dout only when instrmem_rd is high.

4. **Golden reference** classes for each stage of the LC3 Microcontroller: We used System Verilog classes to model the behavior of the DUT according to the spec sheet and the clean DUT that was provided. Each class has a run task which emulates the function of the DUT and a checker task which checks at every clock edge for any bugs. We placed asynchronous and synchronous

Running on ModelSim:

1) Open the terminal in the folder containing the files.
2) Add modelsim using “add modelsim10.3b” command followed by “setenv MODELSIM modelsim.ini” and “vlib mti_lib” commands.
3) Open the GUI using “vsim -novopt &”. On modelsim compile using “do compile.do” command.
4) The “run -all” command is used to run the simulation. We can add the required signals from the simulation pane of modelsim.

COVERAGE STRATEGY:

Various cover groups and cover properties are used to verify and report the functional coverage of the testbench. Cover groups for different instructions are defined as follows:

ALU operations:

Covergroup ALU_OPR_cg

COVERPOINT	***DEFINITION***
Cov_alu_opcode	Covers all the ALU instructions; (ADD, AND, NOT)
Cov_imm_en	Covers if_LC3_Top.Instr_dout[5] when instruction is ALU.
Cov_SR1	Covers if_LC3_Top.Instr_dout[8:6] when instruction is ALU.
Cov_SR2	Covers if_LC3_Top.Instr_dout[2:0] when instruction is ALU.
Cov_DR	Covers if_LC3_Top.Instr_dout[11:9] when instruction is ALU.
Cov_imm5	Covers if_LC3_Top.Instr_dout[4:0] when instruction is immediate type ALU instruction.
Xc_opcode_imm_en	Cross coverage for Cov_imm_en and Cov_alu_opcode.
Xc_opcode_dr_sr1_imm5	Cross coverage for Cov_alu_opcode, Cov_SR1, Cov_DR, Cov_imm5 and Cov_imm_en for only immediate mode ALU instructions.
Xc_opcode_dr_sr1_sr2	Cross coverage for Cov_alu_opcode, Cov_SR1, Cov_SR2 and Cov_DR for only register mode ALU instructions.
Cov_aluin1	Covers ALU operand values (aluin1) binned into 8 bins.
Cov_aluin1_corner	Covers the corner case values of aluin1(all zeroes, all ones, alternate 01, alternate 10, positive and negative values)
Cov_aluin2	Covers ALU operand values aluin2 binned into 8 bins.
Cov_aluin2_corner	Covers the corner cases of aluin2(all zeroes, all ones, alternate 01, alternate 10, positive and negative values)
Cov_aluin1_VSR1	Covers exe_int.VSR1 for all possible values only for ALU instructions.
Cov_aluin2_VSR2	Covers exe_int.VSR2 for all possible values only for ALU instructions.

Cov_opr_zero_zero	Cross covers the case when aluin1 and aluin2 are zeros.
Cov_opr_zero_all1	Cross covers the case when aluin1 is zero and aluin2 is all ones.
Cov_opr_all1_zero	Cross covers the case when aluin1 is all ones and aluin2 is zero.
Cov_opr_all1_all1	Cross covers the case when aluin1 and aluin2 are all ones.
Cov_opr_alt01_alt01	Cross covers the case where value of aluin1 and aluin2 = alternating 01.
Cov_opr_alt01_alt10	Cross covers the case where value of aluin1=alternating 01 and value of aluin2=alternating 10.
Cov_opr_alt10_alt01	Cross covers the case where value of aluin1=alternating 10 and aluin2=alternating 01.
Cov_opr_alt10_alt10	Cross covers the case where value of aluin1 and aluin2 is alternating 10.
Cov_opr_pos_pos	Cross covers the case where value of aluin1 and aluin2 is positive.
Cov_opr_pos_neg	Cross covers the case where value of aluin1 is positive and value of aluin2 is negative.
Cov_opr_neg_pos	Cross covers the case where value of aluin1 is negative and that of aluin2 is positive.
Cov_opr_neg_neg	Cross covers the case where value of aluin1 and aluin2 is negative.

Memory operations:

Covergroup MEM_OPR_cg

COVERPOINT	***DEFINITION***
Cov_mem_opcode	Covers all the MEM instructions(LD,LDR,LDI,LEA,ST,STR,STI).
Cov_BaseR	Covers if_LC3_Top.Instr_dout[8:6] for all possible values if the instruction is LDR/STR.
Cov_SR	Covers if_LC3_Top.Instr_dout[11:9] for all possible values if the instruction is ST/STR/STI.
Cov_DR	Covers if_LC3_Top.Instr_dout[11:9] for all possible values if the instruction is LD/LDR/LDI/LEA.
Cov_PCoffset6	Covers if_LC3_Top.Instr_dout[5:0] if the instruction is LDR/STR.
Cov_PCoffset9	Covers if_LC3_Top.Instr_dout[8:0] if the instruction is LD/LDI/LEA/ST/STI.
Cov_PCoffset9_c	Covers if_LC3_Top.Instr_dout[8:0] for all corner case values(all zeroes, all ones,

	alternate 01, alternate 10, positive and negative values).
Cov_PCoffset6_c	Covers if_LC3_Top.Instr_dout[5:0] for all corner case values(all zeroes, all ones, alternate 01, alternate 10, positive and negative values).
Xc_BaseR_DR_offset6	Cross coverage for Cov_BaseR, Cov_DR and Cov_PCoffset6 values only if instruction is LDR.
Xc_BaseR_DR_offset6_corner	Cross coverage for Cov_BaseR, Cov_DR and Cov_PCoffset6_c only if the instruction is LDR.
Xc_BaseR_SR_offset6	Cross coverage for Cov_BaseR, Cov_SR and Cov_PCoffset6 values only if the instruction is STR.
Xc_BaseR_SR_offset6_c	Cross coverage for Cov_BaseR, Cov_SR and Cov_PCoffset6_c values only if the instruction is STR.
Xc_DR_offset9	Cross coverage for Cov_DR and Cov_PCoffset9 values only if instruction is LD/LDI/LEA.
Xc_DR_offset9_corner	Cross coverage for Cov_DR and Cov_PCoffset9_c values only if instruction is LD/LDI/LEA.

Control operations:

Covergroup CTRL_OPR_cg

COVERPOINT	***DEFINITION***
Cov_ctrl_opcode	Covers all the Control instructions.(BR,JMP)
Cov_BaseR	Covers if_LC3_Top.Instr_dout[8:6] if the instruction is a JMP.
Cov_PSR	Covers wb_int.psr for negative, positive and zero values.
Cov_NZP	Covers exe_int.NZP for all possible values only if branch or jump instruction.
Cov_PCoffset9	Covers if_LC3_Top.Instr_dout[8:0] binned to 8 bins.
Cov_PCoffset9_c	Covers if_LC3_Top.Instr_dout[8:0] for all the corner case values for PCoffset9(all zeroes, all ones, alternate 01, alternate 10, positive and negative values).
Xc_NZP_PSR	Cross covers Cov_NZP and Cov_PSR.

Operation Sequence:

Covergroup OPR_SEQ_cg

COVERPOINT	***DEFINITION***
Cov_alu_first	Covers the cases where ALU operations are executed first followed by ALU, Memory and Control instructions.
Cov_mem_first	Covers the cases where Memory operations are executed first followed by ALU and Branch instructions.
Cov_control_first	Covers the cases where Control operations are executed first followed by Memory and ALU instructions.

COVER PROPERTIES:

The following concurrent assertions has been added to check if the test bench exercises these cases.

1. **Reset:**

Cover properties were used to verify the reset behavior of all the enable signals in the controller block have been included in the interface of the blocks.

2. **br_taken:**

We have included assertion, cover property in the controller block to check and cover the br_taken condition if it is satisfied.

3. **Bypass signals:**

Assertions have been used to check the conditions where the bypass signals(bypass_alu1, bypass_alu2, bypassmem1 and bypassmem2) should be enabled.

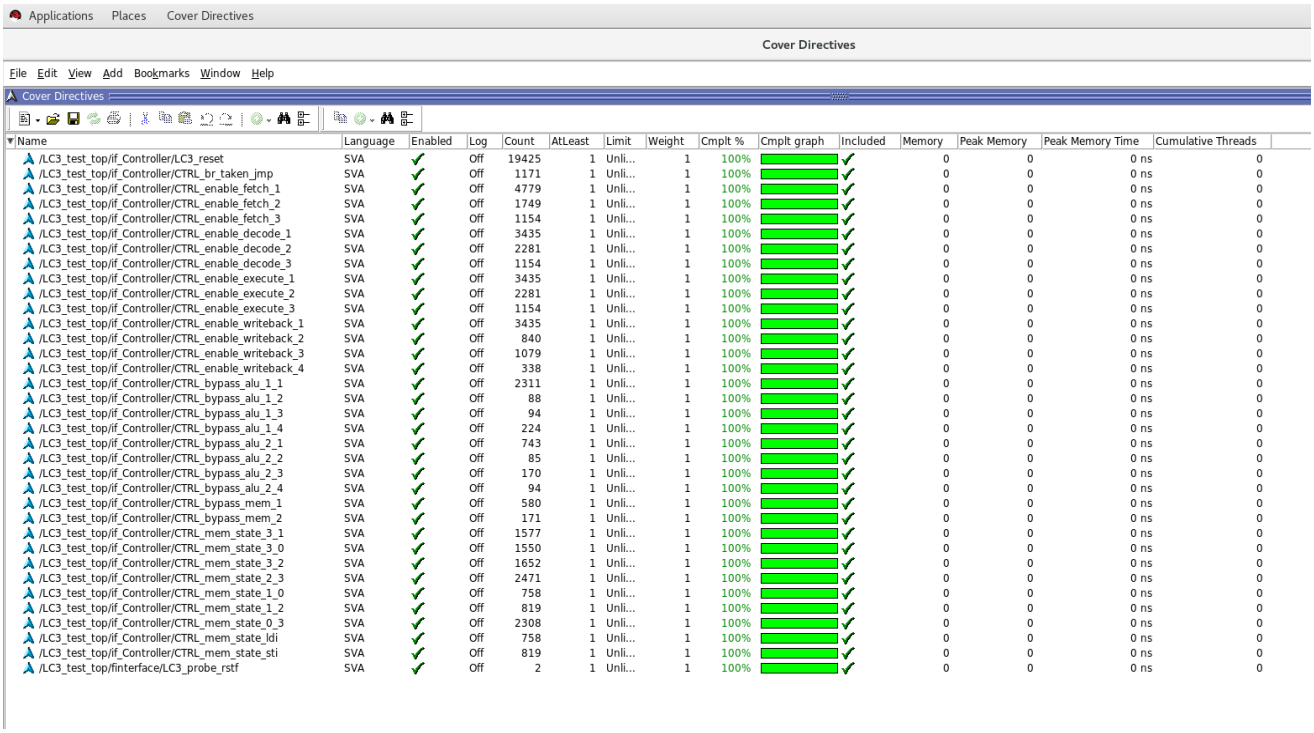
4. **Enable signals:**

We used assertions to check if the enable signals (enable fetch, enable decode, enable execute and enable writeback) are going low at the correct clock cycles.

5**. Mem_state signals:**

Assertions are used to check for the flow of states (from mem state 3-1, 3-0, 3-2, 2-3, 1-0, 1-2, 0-3) and the memory state transitions for LDI and STI.

Cover Properties: (100% Coverage Achieved)



Name	Language	Enabled	Log	Count	AtLeast	Limit	Weight	Cmplt %	Cmplt graph	Included	Memory	Peak Memory	Peak Memory Time	Cumulative Threads
/LC3_test_top/if_Controller/LC3_reset	SVA	✓	Off	19425	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_br_taken_jmp	SVA	✓	Off	1171	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_fetch_1	SVA	✓	Off	4779	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_fetch_2	SVA	✓	Off	1749	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_fetch_3	SVA	✓	Off	1154	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_decode_1	SVA	✓	Off	3435	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_decode_2	SVA	✓	Off	2281	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_decode_3	SVA	✓	Off	1154	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_execute_1	SVA	✓	Off	3435	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_execute_2	SVA	✓	Off	2281	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_execute_3	SVA	✓	Off	1154	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_writeback_1	SVA	✓	Off	3435	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_writeback_2	SVA	✓	Off	840	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_writeback_3	SVA	✓	Off	1079	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_enable_writeback_4	SVA	✓	Off	338	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_1_1	SVA	✓	Off	2311	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_1_2	SVA	✓	Off	88	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_1_3	SVA	✓	Off	94	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_1_4	SVA	✓	Off	224	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_2_1	SVA	✓	Off	743	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_2_2	SVA	✓	Off	85	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_2_3	SVA	✓	Off	170	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_alu_2_4	SVA	✓	Off	94	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_mem_1	SVA	✓	Off	580	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_bypass_mem_2	SVA	✓	Off	171	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_3_1	SVA	✓	Off	1577	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_3_0	SVA	✓	Off	1550	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_3_2	SVA	✓	Off	1652	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_2_3	SVA	✓	Off	2471	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_1_0	SVA	✓	Off	758	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_1_2	SVA	✓	Off	819	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_0_3	SVA	✓	Off	2308	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_ldi	SVA	✓	Off	758	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/if_Controller/CTRL_mem_state_sti	SVA	✓	Off	819	1	Unli...	1	100%	✓	✓	0	0	0 ns	0
/LC3_test_top/finterface/LC3_probe_rstf	SVA	✓	Off	2	1	Unli...	1	100%	✓	✓	0	0	0 ns	0

BUG REPORT:

The following bugs were found in the DUT:

BUG 1:

STAGE/BLOCK: Controller Stage

SIGNAL: bypass_alu_2

We get an error in the Controller block, the bypass_alu_2 signal when an arithmetic instruction(AND , NOT, ADD) is followed by a store instruction(STR, STI, ST) , i.e. when the Store operation is in decode stage [IR] and the Arithmetic operation is in the execute stage [IRexec].

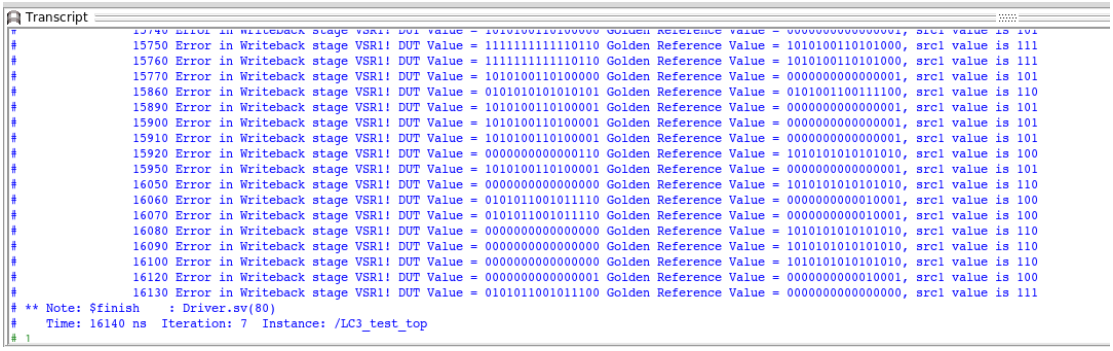
```
VSIM 16> run -all
# reset applied
# reset finished
#                 5360bypass_alu_2 Error is 1, correct value should be 0
#                 5980bypass_alu_2 Error is 1, correct value should be 0
#                 7010bypass_alu_2 Error is 0, correct value should be 1
#                 7190bypass_alu_2 Error is 1, correct value should be 0
#                 8320bypass_alu_2 Error is 0, correct value should be 1
#                13690bypass_alu_2 Error is 1, correct value should be 0
#                14410bypass_alu_2 Error is 0, correct value should be 1
#                15890bypass_alu_2 Error is 0, correct value should be 1
# ** Note: $finish    : Driver.sv(80)
#    Time: 16140 ns  Iteration: 7  Instance: /LC3_test_top
# 1
# Break in Task Testbench_top_sv_unit/Driver::Drive_run at Driver.sv line 80
```

BUG 2:

STAGE/BLOCK: Writeback Stage

SIGNAL: VSR1

We get an error in the Writeback block, the VSR1 value read by the DUT is wrong because of a stuck at zero fault in the MSB bit of the sr1 value(index of Register File). Therefore, for sr1 value 7 it reads from 3, for sr1 value 6 it reads from 2, for sr1 value 5 it reads from 1 and for sr1 value 4 it reads from 0.

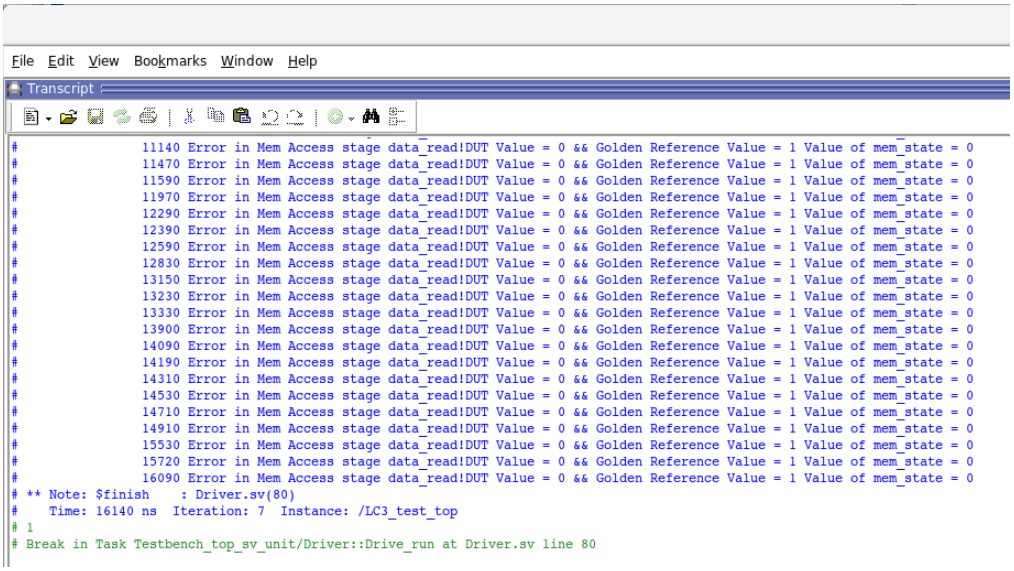


BUG 3:

STAGE/BLOCK: MemAccess Stage

SIGNAL: DMem_rd

We get an error in the MemAccess block, the value of DMem_rd is wrong because for memory state=0(Read state), the value of DMem_rd should be 1 but is 0 (Stuck at zero fault).

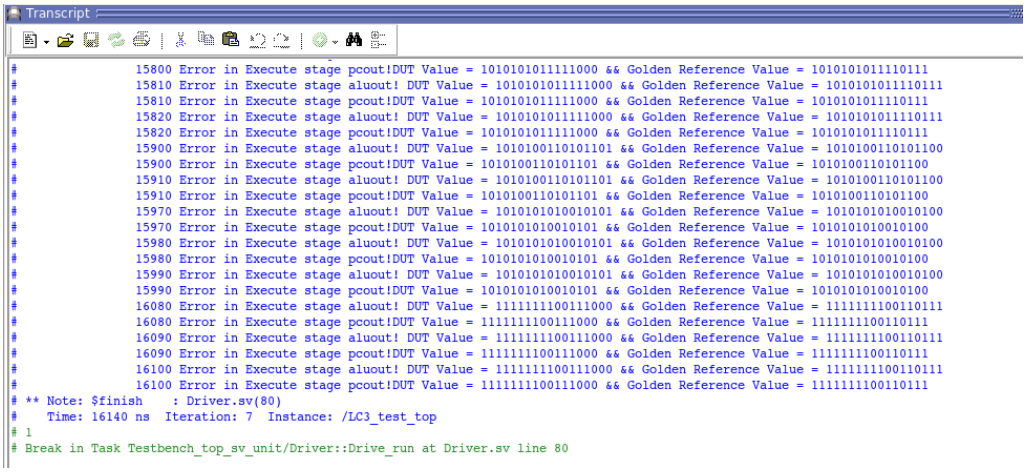


BUG 4:

STAGE/BLOCK: EXECUTE Stage

SIGNAL: pc_out and alu_out

We get an error in the Execute block, the value of pc_out and alu_out is wrong for the following instructions that have pcselect2 as 1 and pcselect1 as 1 which are BR, LD, LDI, LEA, ST and STI. The value of alu_out and pc_out is 1 more than the value it should be.



BUG 5:

There is an error in many stages of the block because the buggy DUT fetches and decodes at the same clock cycle. It is not decoding in the cycle after fetch. [IR] AND[IMem_dout] signal value is the same at a particular clock cycle.