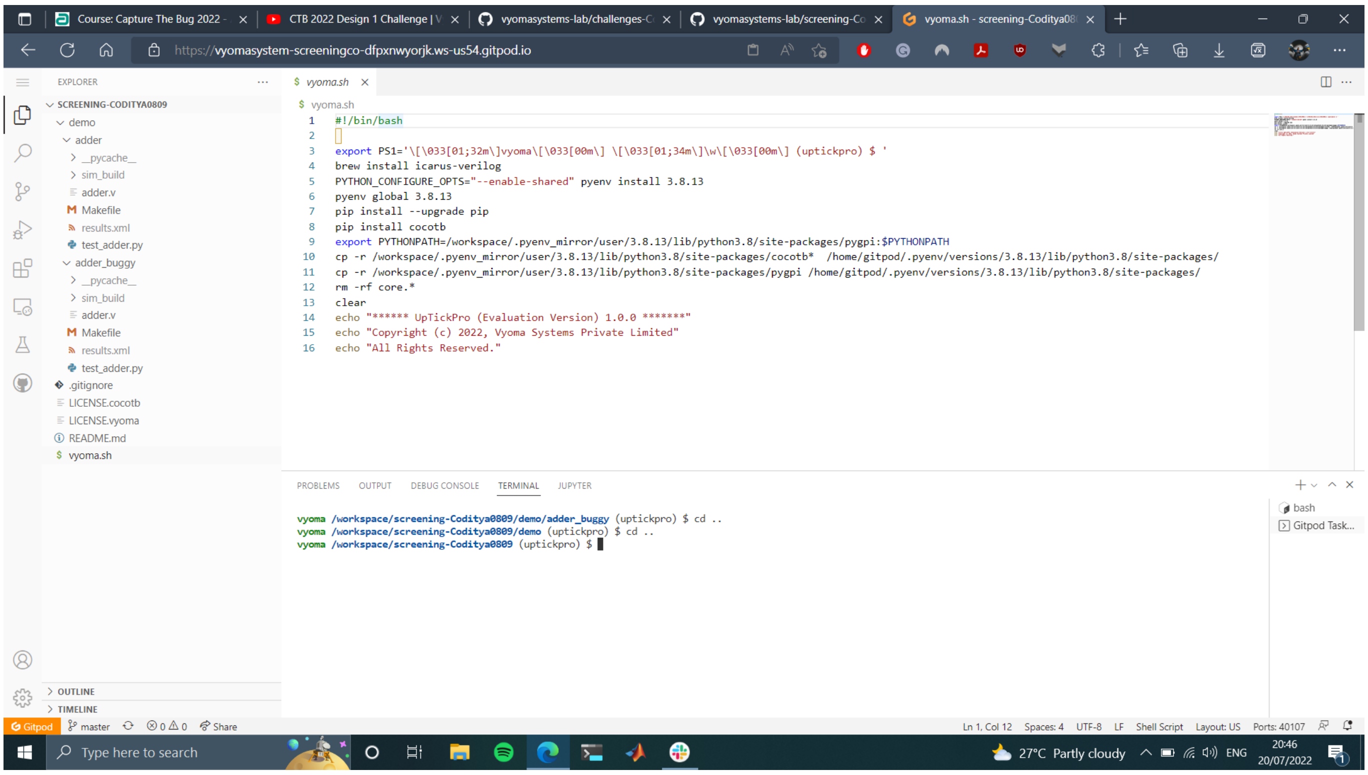
Level-2 Design Verification

The verification environment was setup using Vyoma's UpTickPro provided for the hackathon.



```
#!/bin/bash

export PS1='\[\033[01;32m\]vyoma\[\033[00m\] \[\033[01;34m\]\w\[\033[00m\] (uptickpro) $ '
brew install icarus-verilog
PYTHON_CONFIGURE_OPTS="--enable-shared" pyenv install 3.8.13
pyenv global 3.8.13
pip install --upgrade pip
pip install cocotb
export PYTHONPATH=/workspace/.pyenv_mirror/user/3.8.13/lib/python3.8/site-packages/pygpi:$PYTHONPATH
cp -r /workspace/.pyenv_mirror/user/3.8.13/lib/python3.8/site-packages/cocotb*  /home/gitpod/.pyenv/versions/3.8.13/lib/python3.8/site-packages/
cp -r /workspace/.pyenv_mirror/user/3.8.13/lib/python3.8/site-packages/pygpi /home/gitpod/.pyenv/versions/3.8.13/lib/python3.8/site-packages/
rm -rf core.*
clear
echo "****** UpTickPro (Evaluation Version) 1.0.0 *******"
echo "Copyright (c) 2022, Vyoma Systems Private Limited"
echo "All Rights Reserved."
```

```
vyoma /workspace/screening-Coditya0809/demo/adder_buggy (uptickpro) $ cd ..
vyoma /workspace/screening-Coditya0809/demo (uptickpro) $ cd ..
vyoma /workspace/screening-Coditya0809 (uptickpro) $
```

Verification Environment

The CoCoTb based Python test was developed as explained. The test drives inputs to the mips_16_processor (dut). The only inputs are clock (dut.clk) and reset(dut.rst).

Test : Generating constrained random inputs based on the study of model_mkbitmanip.py file

The values were assigned to the input ports using

```
err_count = 0
    for i in range (1000):
        # input transaction
        mav_putvalue_src1 = random.getrandbits(32)
        mav_putvalue_src2 = random.getrandbits(32)
        mav_putvalue_src3 = random.getrandbits(32)

        # intelligently choosing a random instruction
        mav_putvalue_instr = random.getrandbits(32)
        le = bin(mav_putvalue_instr)[2:].zfill(32)

        # choosing the random sequences
        opcode = random.choices(opcode_seq,cum_weights = [35, 58],k=1)[0]
        func7 = random.choice(func7_seq)
        func3_r = random.choice(func3_r_seq)
        func3_imm = random.choice(func3_imm_seq)
        func7_imm = random.choice(func7_imm_seq)
        func7_2bit = random.choice(func7_2bit_seq)
        func7_fsri_1bit = random.choice(["0","1"])

        if (opcode == "0110011"): # R-type Instruction
            if(func7_2bit == "00"): # Normal R-type Instruction
                le = func7 + le[7:17] + func3_r + le[20:25] + opcode
            else: #R4-type Instruction
                le = le[0:5] + func7_2bit + le[7:17] + func3_imm + le[20:25] +
opcode

        if (opcode == "0010011"): # I-type Instruction
            if(func7_fsri_1bit == "1"): # FSRI Instruction
                le = le[0:5] + func7_fsri_1bit + le[6:17] + "101" + le[20:25] +
opcode
            else: # Normal I-type Instruction
                le = func7_imm + "0" + le[6:17] + func3_imm + le[20:25] + opcode

        mav_putvalue_instr = int(le,2)

        # expected output from the model
        expected_mav_putvalue = bitmanip(mav_putvalue_instr, mav_putvalue_src1,
mav_putvalue_src2, mav_putvalue_src3)

        # driving the input transaction
        dut.mav_putvalue_src1.value = mav_putvalue_src1
        dut.mav_putvalue_src2.value = mav_putvalue_src2
        dut.mav_putvalue_src3.value = mav_putvalue_src3
        dut.EN_mav_putvalue.value = 1
        dut.mav_putvalue_instr.value = mav_putvalue_instr

        yield Timer(1)

        # obtaining the output
        dut_output = dut.mav_putvalue.value

        cocotb.log.info(f'DUT OUTPUT={hex(dut_output)}')
        cocotb.log.info(f'EXPECTED OUTPUT={hex(expected_mav_putvalue)}')

        # comparison
        error_message = f'Value mismatch DUT = {hex(dut_output)} does not match
MODEL = {hex(expected_mav_putvalue)}'
        if (dut_output != expected_mav_putvalue):
            if (hex(expected_mav_putvalue) != "0x0"):
                err_count = err_count + 1
                dut._log.info(error_message)
```

The assert statement was used for comparing the mkbitmanip's output to the expected model value.

The following assert statement was used:

```
final_error_message = f'The behaviour of the DUT differs from that of the model.
The DUT fails for {err_count}/{i+1} instructions.'
    assert err_count == 0, final_error_message
```

I kept a count of all the instructions (repititions also counted as errors) that were not functioning correctly with the help of the variable `err_count`. The `i` value is the total number of instructions (1000) that I ran the design for.

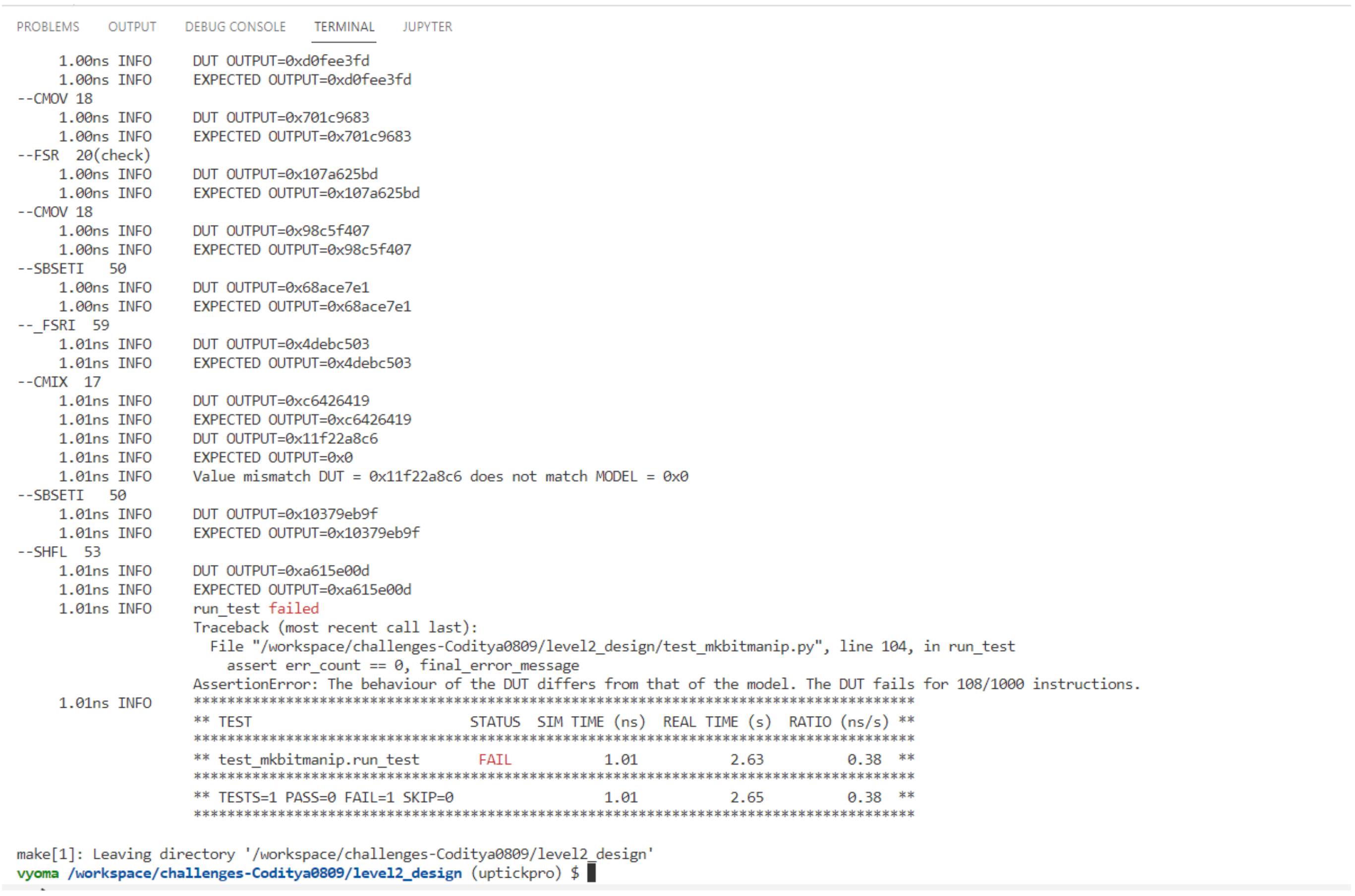
Test Scenario **(Important)**

- Test Inputs: Random valid instructions were given to the dut, and each instruction was checked for correct operation.
- Expected Output: The expected output for each instruction can be seen on the terminal window. The expected output was calculated by a model_mkbitmanip which does the correct operations according to the instructions given.
- Observed Output: All of the instructions were seen to satisfy the expected outputs but for the ANDN instruction.

Failed Test Cases

```
     0.86ns INFO     EXPECTED OUTPUT=0x1
--CMIX  17
     0.86ns INFO     DUT OUTPUT=0x2a2c11e9
     0.86ns INFO     EXPECTED OUTPUT=0x2a2c11e9
--_FSRI  59
     0.86ns INFO     DUT OUTPUT=0x77dde421
     0.86ns INFO     EXPECTED OUTPUT=0x77dde421
--FSL 19
     0.86ns INFO     DUT OUTPUT=0x1ec902b1b
     0.86ns INFO     EXPECTED OUTPUT=0x1ec902b1b
--GORCI 57
     0.86ns INFO     DUT OUTPUT=0x1ffffffff
     0.86ns INFO     EXPECTED OUTPUT=0x1ffffffff
--_FSRI  59
     0.86ns INFO     DUT OUTPUT=0x1cdbacc19
     0.86ns INFO     EXPECTED OUTPUT=0x1cdbacc19
--ANDN 1
     0.87ns INFO     DUT OUTPUT=0x1860c319d
     0.87ns INFO     EXPECTED OUTPUT=0x59004043
     0.87ns INFO     Value mismatch DUT = 0x1860c319d does not match MODEL = 0x59004043
--CMIX  17
     0.87ns INFO     DUT OUTPUT=0xe1d8cfff
     0.87ns INFO     EXPECTED OUTPUT=0xe1d8cfff
--SBCLRI   49
     0.87ns INFO     DUT OUTPUT=0x163c02495
     0.87ns INFO     EXPECTED OUTPUT=0x163c02495
--CMOV 18
     0.87ns INFO     DUT OUTPUT=0x385b686b
     0.87ns INFO     EXPECTED OUTPUT=0x385b686b
--FSR  20(check)
     0.87ns INFO     DUT OUTPUT=0x960de189
     0.87ns INFO     EXPECTED OUTPUT=0x960de189
--CMIX  17
     0.87ns INFO     DUT OUTPUT=0x38dd68cb
     0.87ns INFO     EXPECTED OUTPUT=0x38dd68cb
--_FSRI  59
     0.87ns INFO     DUT OUTPUT=0x1b2ccc18d
     0.87ns INFO     EXPECTED OUTPUT=0x1b2ccc18d
--FSR  20(check)
     0.87ns INFO     DUT OUTPUT=0x1590262eb
     0.87ns INFO     EXPECTED OUTPUT=0x1590262eb
--CMIX  17
     0.87ns INFO     DUT OUTPUT=0x17fe36ddf
     0.87ns INFO     EXPECTED OUTPUT=0x17fe36ddf
```

Output mismatches for the instruction ANDN proving that there is a design bug.



```
     1.00ns INFO     DUT OUTPUT=0xd0fee3fd
     1.00ns INFO     EXPECTED OUTPUT=0xd0fee3fd
--CMOV 18
     1.00ns INFO     DUT OUTPUT=0x701c9683
     1.00ns INFO     EXPECTED OUTPUT=0x701c9683
--FSR  20(check)
     1.00ns INFO     DUT OUTPUT=0x107a625bd
     1.00ns INFO     EXPECTED OUTPUT=0x107a625bd
--CMOV 18
     1.00ns INFO     DUT OUTPUT=0x98c5f407
     1.00ns INFO     EXPECTED OUTPUT=0x98c5f407
--SBSETI   50
     1.00ns INFO     DUT OUTPUT=0x68ace7e1
     1.00ns INFO     EXPECTED OUTPUT=0x68ace7e1
--_FSRI  59
     1.01ns INFO     DUT OUTPUT=0x4debc503
     1.01ns INFO     EXPECTED OUTPUT=0x4debc503
--CMIX  17
     1.01ns INFO     DUT OUTPUT=0xc6426419
     1.01ns INFO     EXPECTED OUTPUT=0xc6426419
     1.01ns INFO     DUT OUTPUT=0x11f22a8c6
     1.01ns INFO     EXPECTED OUTPUT=0x0
     1.01ns INFO     Value mismatch DUT = 0x11f22a8c6 does not match MODEL = 0x0
--SBSETI   50
     1.01ns INFO     DUT OUTPUT=0x10379eb9f
     1.01ns INFO     EXPECTED OUTPUT=0x10379eb9f
--SHFL  53
     1.01ns INFO     DUT OUTPUT=0xa615e00d
     1.01ns INFO     EXPECTED OUTPUT=0xa615e00d
     1.01ns INFO     run_test failed
                     Traceback (most recent call last):
                       File "/workspace/challenges-Coditya0809/level2_design/test_mkbitmanip.py", line 104, in run_test
                         assert err_count == 0, final_error_message
                     AssertionError: The behaviour of the DUT differs from that of the model. The DUT fails for 108/1000 instructions.
     1.01ns INFO     ******************************************************************************************
                     ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                     ******************************************************************************************
                     ** test_mkbitmanip.run_test       FAIL           1.01           2.63           0.38  **
                     ******************************************************************************************
                     ** TESTS=1 PASS=0 FAIL=1 SKIP=0                 1.01           2.65           0.38  **
                     ******************************************************************************************

make[1]: Leaving directory '/workspace/challenges-Coditya0809/level2_design'
vyoma /workspace/challenges-Coditya0809/level2_design (uptickpro) $
```

Design Bug

Based on the above test inputs and analysing the design, I could see the following:

```
assign mav_putvalue =
         { x__h33,
           (mav_putvalue_instr[6:0] == 7'b0110011 ||
        mav_putvalue_instr[6:0] == 7'b0010011) &&
           mav_putvalue_instr_BITS_31_TO_25_EQ_0b100000_A_ETC___d2336 } ;
...

assign x__h33 =
         NOT_mav_putvalue_instr_BITS_31_TO_25_EQ_0b1000_ETC___d196 ?
           field1___1__h3327 :
           (NOT_mav_putvalue_instr_BITS_31_TO_25_EQ_0b1000_ETC___d299 ?
          field1___1__h4164 :
          IF_NOT_mav_putvalue_instr_BITS_14_TO_12_CONCAT_ETC___d2279) ;

...

assign IF_NOT_mav_putvalue_instr_BITS_14_TO_12_CONCAT_ETC___d2279 =
         NOT_mav_putvalue_instr_BITS_14_TO_12_CONCAT_ma_ETC___d339 ?
           field1___1__h4621 :
           (NOT_mav_putvalue_instr_BITS_31_TO_25_EQ_0b1000_ETC___d365 ?
          IF_mav_putvalue_instr_BITS_21_TO_20_66_EQ_0b0__ETC___d2277 :
          field1__h109) ;

...
```

```
assign field1__h109 =
         (mav_putvalue_instr[31:25] == 7'b0100000 &&
          x__h254 == 10'b1110110011) ?
           x__h39889 :                                        <==== BUG (Gives
the value of src1 & src2, but we want src1 & ~src2)
           IF_mav_putvalue_instr_BITS_31_TO_25_EQ_0b10000_ETC___d2273 ;

...

assign x__h39889 = mav_putvalue_src1 & mav_putvalue_src2 ;
```

Upon backtracking the output value `mav_putvalue`, I started going deep into the chain until I found the place where `mav_putvalue_src1 & mav_putvalue_src2` was being assigned. Then, to not disturb the functioning of other instructions, I added a new wire, which was named `x__h39890`, for no reason at all. (:p Because the code was not readable as is, I chose a random name for this wire)

Then, I assigned `mav_putvalue_src1 & ~mav_putvalue_src2` to this new wire `x__h39890` and used this wire to feed the ANDN instruction correctly. Luckily, after 10-12 hours of struggling to understand the code, I was successfull in passing the ANDN instruction as well.

The following section shows the passed test case for ANDN instruction without disturbing the functioning of other instructions.

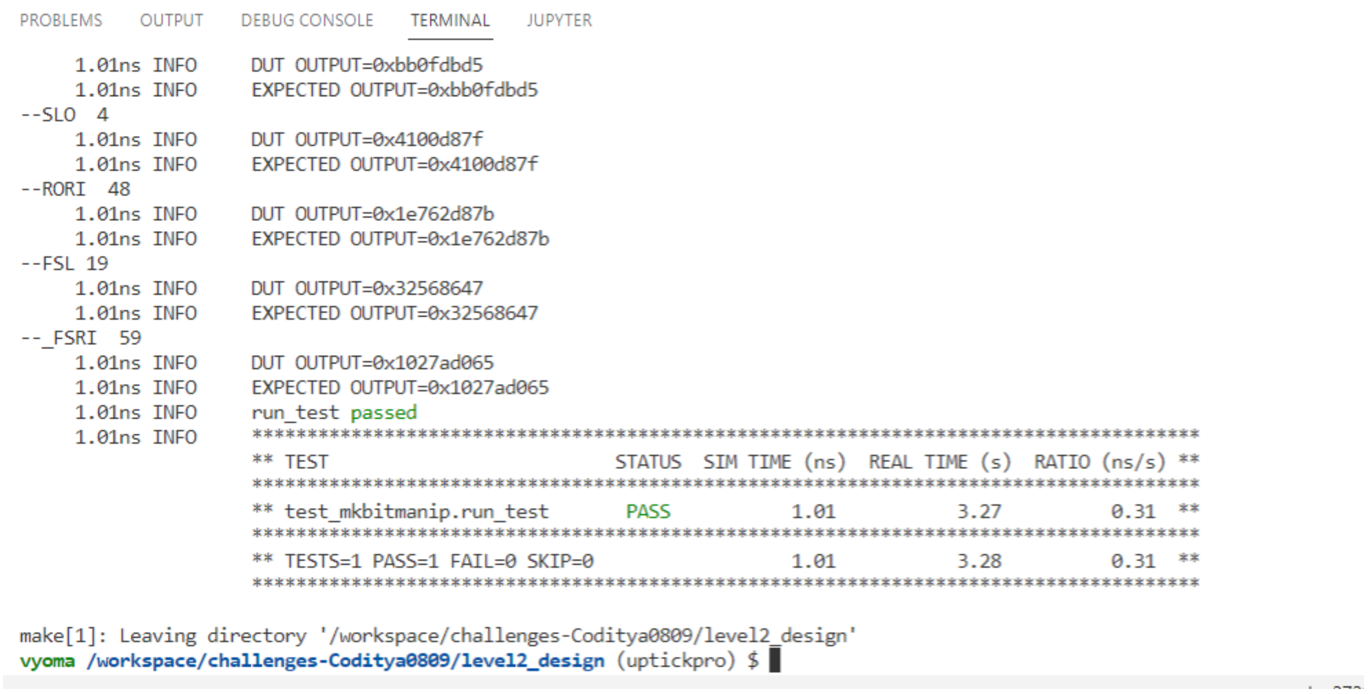
Design Fix

Updating the design and re-running the test made the test pass.

1. Screenshot of ANDN Instruciton giving correct output

```
     0.94ns INFO     EXPECTED OUTPUT=0x0
--RORI  48
     0.94ns INFO     DUT OUTPUT=0xe77493a3
     0.94ns INFO     EXPECTED OUTPUT=0xe77493a3
--ANDN 1
     0.94ns INFO     DUT OUTPUT=0x808d1025
     0.94ns INFO     EXPECTED OUTPUT=0x808d1025
--ROL  6
     0.94ns INFO     DUT OUTPUT=0x45acfcb3
     0.94ns INFO     EXPECTED OUTPUT=0x45acfcb3
     0.94ns INFO     DUT OUTPUT=0x1af0bf860
     0.94ns INFO     EXPECTED OUTPUT=0x0
--GORCI 57
     0.94ns INFO     DUT OUTPUT=0x13f893f89
     0.94ns INFO     EXPECTED OUTPUT=0x13f893f89
--_FSRI  59
     0.94ns INFO     DUT OUTPUT=0x126bc435f
     0.94ns INFO     EXPECTED OUTPUT=0x126bc435f
--SBINVI  51
     0.94ns INFO     DUT OUTPUT=0x11c9da62b
     0.94ns INFO     EXPECTED OUTPUT=0x11c9da62b
--FSL 19
     0.94ns INFO     DUT OUTPUT=0x1f2db8251
     0.94ns INFO     EXPECTED OUTPUT=0x1f2db8251
--CMOV 18
```

2. Screenshot of PASS status of the test



```
     1.01ns INFO     DUT OUTPUT=0xbb0fdbd5
     1.01ns INFO     EXPECTED OUTPUT=0xbb0fdbd5
--SLO  4
     1.01ns INFO     DUT OUTPUT=0x4100d87f
     1.01ns INFO     EXPECTED OUTPUT=0x4100d87f
--RORI  48
     1.01ns INFO     DUT OUTPUT=0x1e762d87b
     1.01ns INFO     EXPECTED OUTPUT=0x1e762d87b
--FSL 19
     1.01ns INFO     DUT OUTPUT=0x32568647
     1.01ns INFO     EXPECTED OUTPUT=0x32568647
--_FSRI  59
     1.01ns INFO     DUT OUTPUT=0x1027ad065
     1.01ns INFO     EXPECTED OUTPUT=0x1027ad065
     1.01ns INFO     run_test passed
     1.01ns INFO     **************************************************************************************
                     ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                     **************************************************************************************
                     ** test_mkbitmanip.run_test       PASS           1.01           3.27          0.31  **
                     **************************************************************************************
                     ** TESTS=1 PASS=1 FAIL=0 SKIP=0                  1.01           3.28          0.31  **
                     **************************************************************************************

make[1]: Leaving directory '/workspace/challenges-Coditya0809/level2_design'
vyoma /workspace/challenges-Coditya0809/level2_design (uptickpro) $
```

The updated design is checked in as ../level2_design_fix/mkbitmanip_fix.v

Verification Strategy

1. I intelligently tried to generate random valid instructions for the bit-manipulator design. Although, I can say that I was about 80% successfull in generating random valid instructions because some instructions had an expected value of `0x0` which meant they were invalid instructions.

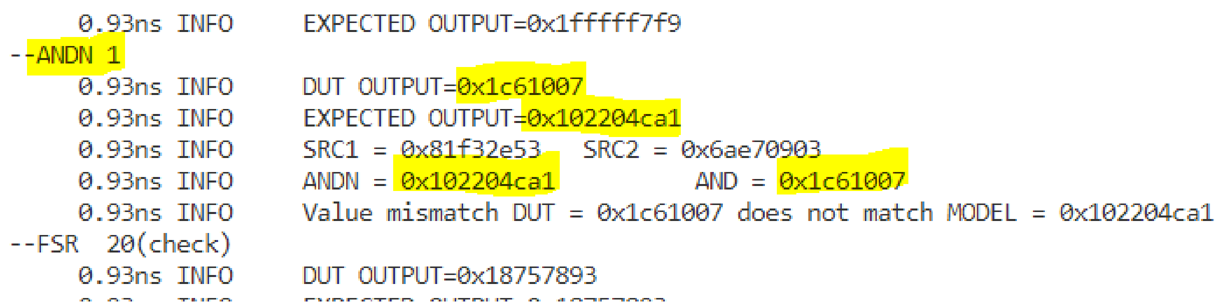
2. I run the test for 1000 constrained random instructions, and observed that all the instructions except `ANDN` were giving the expected value at the output. Then, I suspected that there was an arithmetic mistake done in the calculation of ANDN instruction. (There I go! To do the most tiresome job in verification: Verification! :p)
3. I started looking at the code in `mkbitmanip.v`. It was just 4500+ lines of code (LOC). Then, I tried my best understanding what was happening on a broad scale. Then, I thought that I will search the code for a line that assigns `mav_putvalue_src1 & mav_putvalue_src2` and then change it to `mav_putvalue_src1 & ~mav_putvalue_src2`. But, so foolish of me to think that it would solve the problem! Doing this did not help in correcting `ANDN` instruction, but instead it also spoiled the fi=unctioning of some other instructions like CMOV, CMIX, SBINV, etc..

I confirmed this by the following:

```
result_andn = int(dut.mav_putvalue_src1.value) & ~int(dut.mav_putvalue_src2.value)
result_andn = result_andn & 0xffffffff
result_andn = (result_andn << 1) | 1

result_and = int(dut.mav_putvalue_src1.value) & int(dut.mav_putvalue_src2.value)
result_and = result_and & 0xffffffff
result_and = (result_and << 1) | 1
cocotb.log.info(f'ANDN = {hex(result_andn)} \t AND = {hex(result_and)}')
```

The result:



```
     0.93ns INFO     EXPECTED OUTPUT=0x1fffff7f9
--ANDN 1
     0.93ns INFO     DUT OUTPUT=0x1c61007
     0.93ns INFO     EXPECTED OUTPUT=0x102204ca1
     0.93ns INFO     SRC1 = 0x81f32e53    SRC2 = 0x6ae70903
     0.93ns INFO     ANDN = 0x102204ca1          AND = 0x1c61007
     0.93ns INFO     Value mismatch DUT = 0x1c61007 does not match MODEL = 0x102204ca1
--FSR  20(check)
     0.93ns INFO     DUT OUTPUT=0x18757893
```

4. Then, there was a lot of random searching of signals thorugh the code. At one point I started at some assignment to a wire and ended up in a completely random assignment of another wire. I cried, I struggled, I was helpless at this point in time. I truly felt worthless.
5. Next day, fresh start, continuing my random *back-tracking* of signals. I thought after a while of random searching that why not start from the absolute *back* if I was going for *back-tracking* after all. There I started my journey from `mav_putvalue --> x__h33 --> IF_NOT_mav_putvalue_instr_BITS_14_TO_12_CONCAT_ETC___d2279 --> field1__h109 --> x__h39889 --> assign x__h39889 = mav_putvalue_src1 & mav_putvalue_src2 ;`. Upon verifying that these were the correct signals which affected the ANDN instruction (checked opcode and func7 bits for ANDN), I then decided that this x__h39889 had to be given the ANDN value `mav_putvalue_src1 & ~mav_putvalue_src2`. I decided to use another wire and assigned it to `field1__h109`.
6. I tried to run the test, not with much enthusiasm though. But, to my surprise, the test passed!

7. I shut my laptop's screen, jumped (figuratively) for a good 10 minutes and started to make this report. (sed)
8. I decided I was never doing something like this again.
9. I was wrong, I was experienced, I was motivated!

Is the verification complete ?

xD xD xD Nope! First of all, my randomization of generating instructions might not have been perfect, which meant all the possible 63 instructions might not have been generated. Although, I am confident that atleast 80% of all instructions must have been generated. I could not find any other bug other than that of ANDN instruction unless I missed a super simple bug that did not need hours to capture (facepalm).