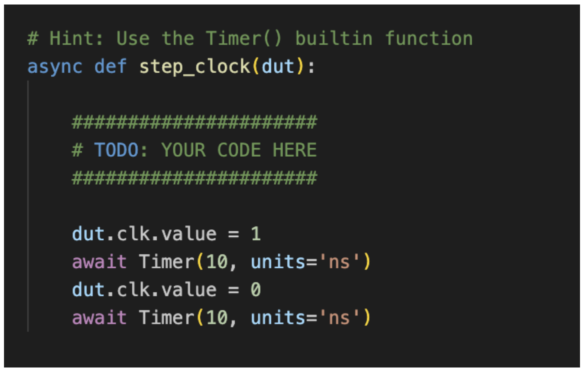
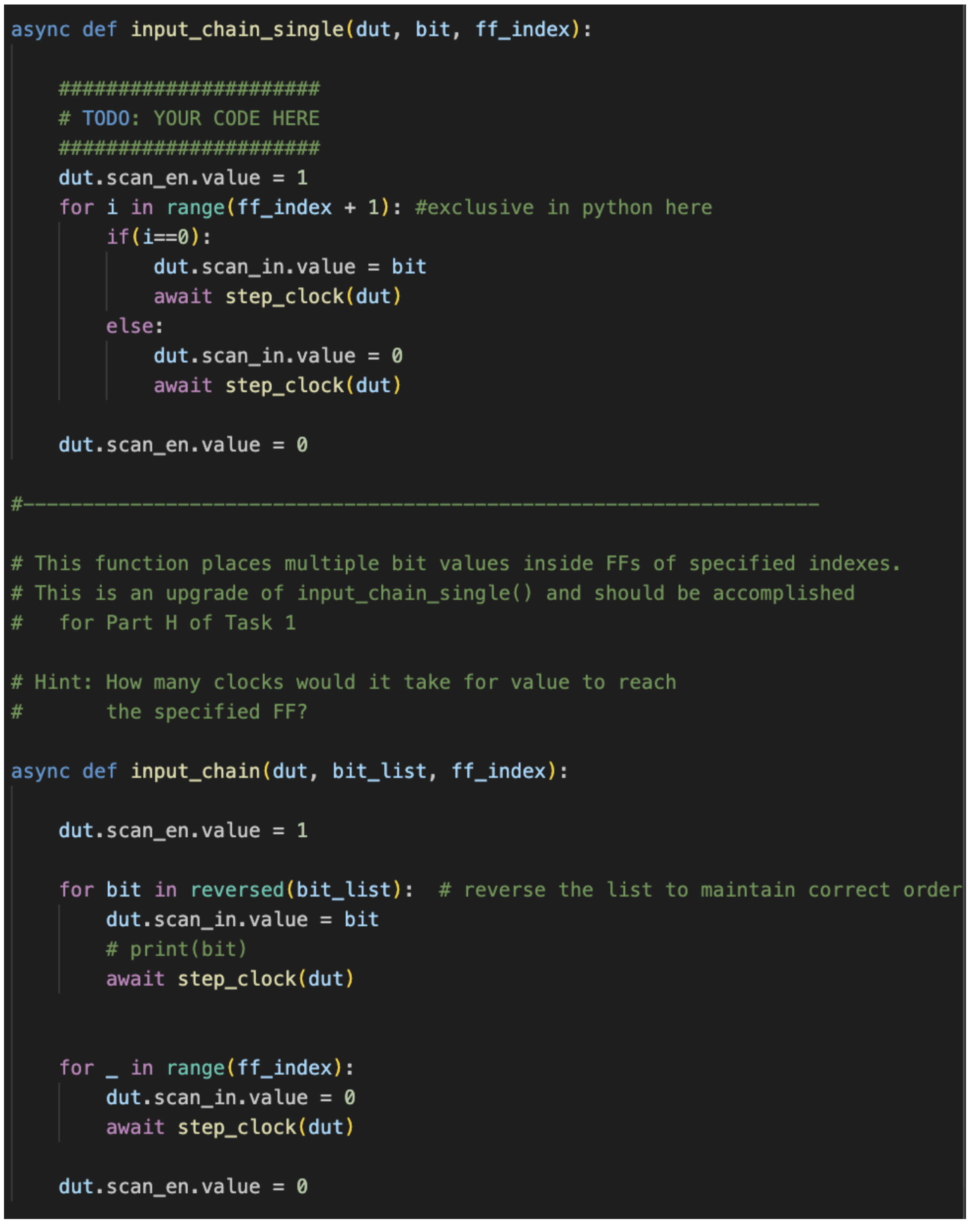
Task1: (code checked with TA during OH, also pushed)
clock:



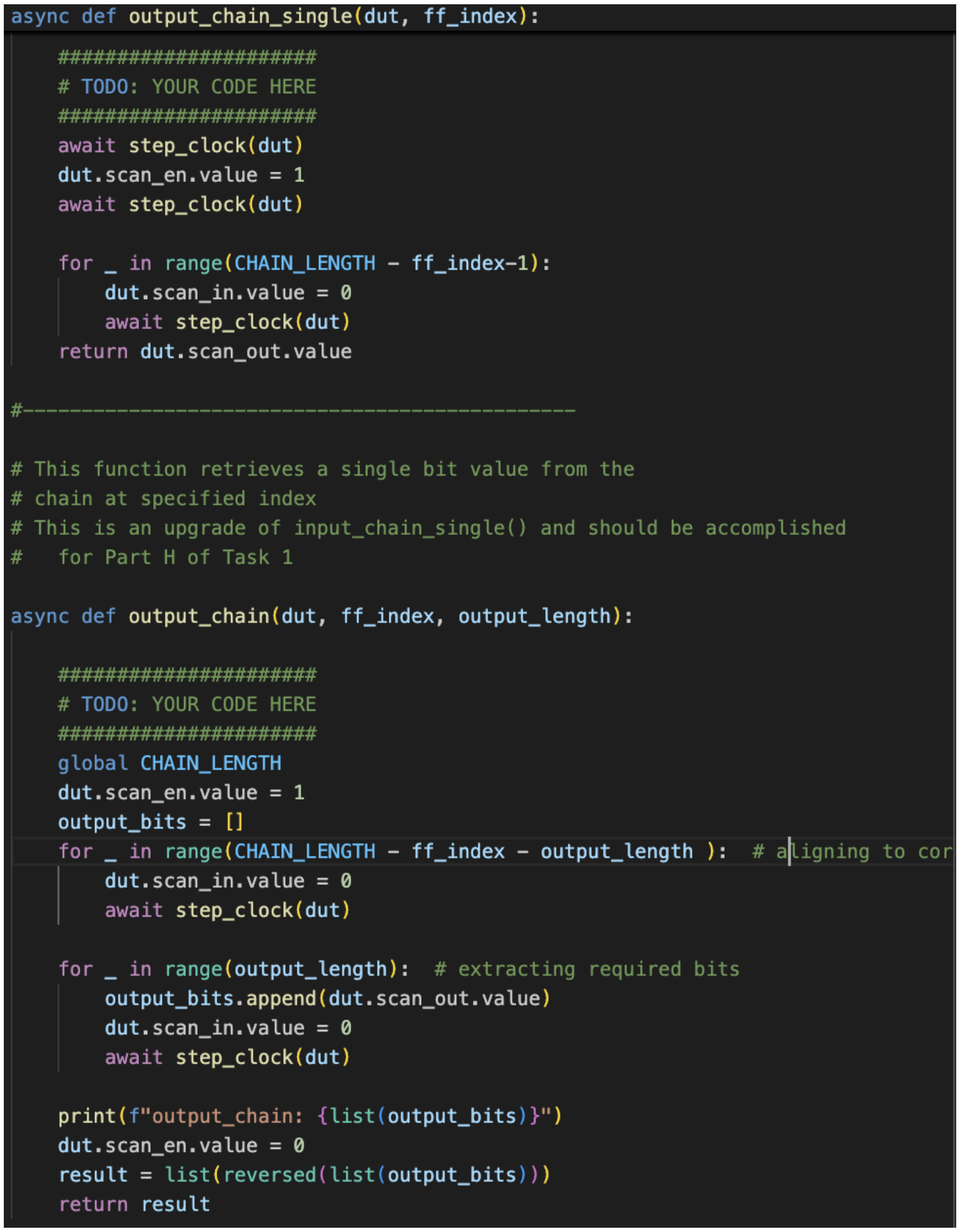
```python
# Hint: Use the Timer() builtin function
async def step_clock(dut):

    #####################
    # TODO: YOUR CODE HERE
    #####################

    dut.clk.value = 1
    await Timer(10, units='ns')
    dut.clk.value = 0
    await Timer(10, units='ns')
```

input:



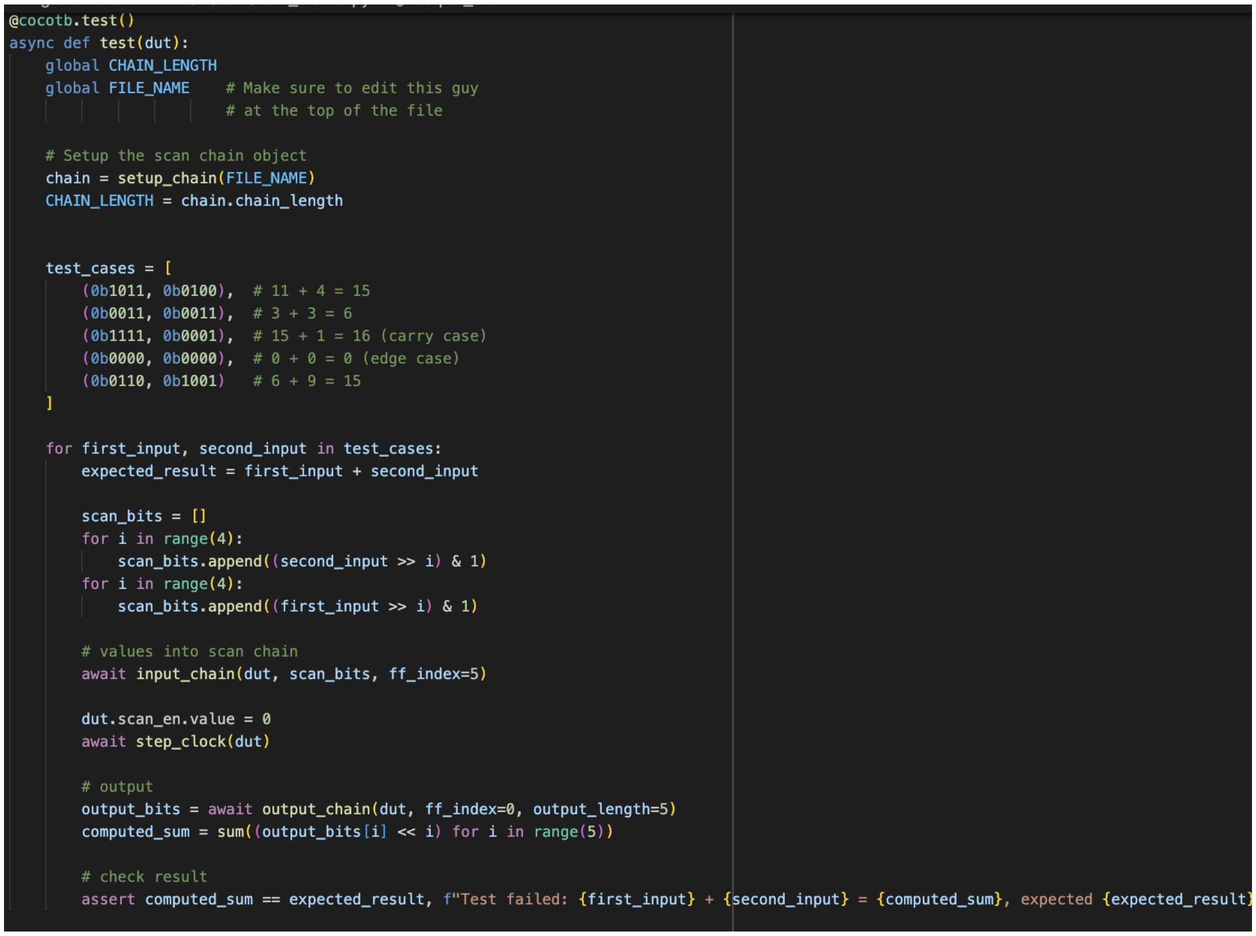
```python
async def input_chain_single(dut, bit, ff_index):

    #####################
    # TODO: YOUR CODE HERE
    #####################
    dut.scan_en.value = 1
    for i in range(ff_index + 1): #exclusive in python here
        if(i==0):
            dut.scan_in.value = bit
            await step_clock(dut)
        else:
            dut.scan_in.value = 0
            await step_clock(dut)

    dut.scan_en.value = 0

#-------------------------------------------------------------------

# This function places multiple bit values inside FFs of specified indexes.
# This is an upgrade of input_chain_single() and should be accomplished
#   for Part H of Task 1

# Hint: How many clocks would it take for value to reach
#       the specified FF?

async def input_chain(dut, bit_list, ff_index):

    dut.scan_en.value = 1

    for bit in reversed(bit_list):  # reverse the list to maintain correct order
        dut.scan_in.value = bit
        # print(bit)
        await step_clock(dut)


    for _ in range(ff_index):
        dut.scan_in.value = 0
        await step_clock(dut)

    dut.scan_en.value = 0
```

output:



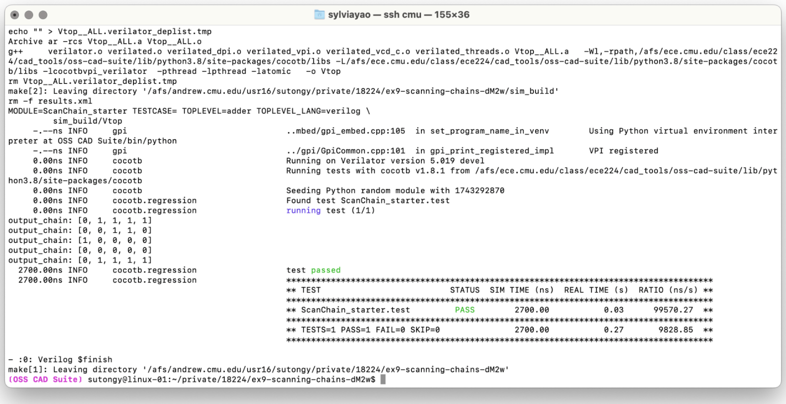
```
async def output_chain_single(dut, ff_index):

    #####################
    # TODO: YOUR CODE HERE
    #####################
    await step_clock(dut)
    dut.scan_en.value = 1
    await step_clock(dut)

    for _ in range(CHAIN_LENGTH - ff_index-1):
        dut.scan_in.value = 0
        await step_clock(dut)
    return dut.scan_out.value

#-----------------------------------------------

# This function retrieves a single bit value from the
# chain at specified index
# This is an upgrade of input_chain_single() and should be accomplished
#   for Part H of Task 1

async def output_chain(dut, ff_index, output_length):

    #####################
    # TODO: YOUR CODE HERE
    #####################
    global CHAIN_LENGTH
    dut.scan_en.value = 1
    output_bits = []
    for _ in range(CHAIN_LENGTH - ff_index - output_length ):  # aligning to cor
        dut.scan_in.value = 0
        await step_clock(dut)

    for _ in range(output_length):  # extracting required bits
        output_bits.append(dut.scan_out.value)
        dut.scan_in.value = 0
        await step_clock(dut)

    print(f"output_chain: {list(output_bits)}")
    dut.scan_en.value = 0
    result = list(reversed(list(output_bits)))
    return result
```

Task2:
code:



```python
@cocotb.test()
async def test(dut):
    global CHAIN_LENGTH
    global FILE_NAME    # Make sure to edit this guy
                        # at the top of the file

    # Setup the scan chain object
    chain = setup_chain(FILE_NAME)
    CHAIN_LENGTH = chain.chain_length


    test_cases = [
        (0b1011, 0b0100),  # 11 + 4 = 15
        (0b0011, 0b0011),  # 3 + 3 = 6
        (0b1111, 0b0001),  # 15 + 1 = 16 (carry case)
        (0b0000, 0b0000),  # 0 + 0 = 0 (edge case)
        (0b0110, 0b1001)   # 6 + 9 = 15
    ]

    for first_input, second_input in test_cases:
        expected_result = first_input + second_input

        scan_bits = []
        for i in range(4):
            scan_bits.append((second_input >> i) & 1)
        for i in range(4):
            scan_bits.append((first_input >> i) & 1)

        # values into scan chain
        await input_chain(dut, scan_bits, ff_index=5)

        dut.scan_en.value = 0
        await step_clock(dut)

        # output
        output_bits = await output_chain(dut, ff_index=0, output_length=5)
        computed_sum = sum((output_bits[i] << i) for i in range(5))

        # check result
        assert computed_sum == expected_result, f"Test failed: {first_input} + {second_input} = {computed_sum}, expected {expected_result
```

Output:



sylviayao — ssh cmu — 155×36

```
echo "" > Vtop__ALL.verilator_deplist.tmp
Archive ar -rcs Vtop__ALL.a Vtop__ALL.o
g++    verilator.o verilated.o verilated_dpi.o verilated_vpi.o verilated_vcd_c.o verilated_threads.o Vtop__ALL.a   -Wl,-rpath,/afs/ece.cmu.edu/class/ece22
4/cad_tools/oss-cad-suite/lib/python3.8/site-packages/cocotb/libs -L/afs/ece.cmu.edu/class/ece224/cad_tools/oss-cad-suite/lib/python3.8/site-packages/cocot
b/libs -lcocotbvpi_verilator  -pthread -lpthread -latomic   -o Vtop
rm Vtop__ALL.verilator_deplist.tmp
make[2]: Leaving directory '/afs/andrew.cmu.edu/usr16/sutongy/private/18224/ex9-scanning-chains-dM2w/sim_build'
rm -f results.xml
MODULE=ScanChain_starter TESTCASE= TOPLEVEL=adder TOPLEVEL_LANG=verilog \
         sim_build/Vtop
     -.--ns INFO     gpi                                ..mbed/gpi_embed.cpp:105  in set_program_name_in_venv        Using Python virtual environment inter
preter at OSS CAD Suite/bin/python
     -.--ns INFO     gpi                                ../gpi/GpiCommon.cpp:101  in gpi_print_registered_impl       VPI registered
     0.00ns INFO     cocotb                             Running on Verilator version 5.019 devel
     0.00ns INFO     cocotb                             Running tests with cocotb v1.8.1 from /afs/ece.cmu.edu/class/ece224/cad_tools/oss-cad-suite/lib/pyt
hon3.8/site-packages/cocotb
     0.00ns INFO     cocotb                             Seeding Python random module with 1743292870
     0.00ns INFO     cocotb.regression                  Found test ScanChain_starter.test
     0.00ns INFO     cocotb.regression                  running test (1/1)
output_chain: [0, 1, 1, 1, 1]
output_chain: [0, 0, 1, 1, 0]
output_chain: [1, 0, 0, 0, 0]
output_chain: [0, 0, 0, 0, 0]
output_chain: [0, 1, 1, 1, 1]
  2700.00ns INFO     cocotb.regression                  test passed
  2700.00ns INFO     cocotb.regression                  **************************************************************************************
                                                        ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        **************************************************************************************
                                                        ** ScanChain_starter.test         PASS        2700.00           0.03      99570.27  **
                                                        **************************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0               2700.00           0.27       9828.85  **
                                                        **************************************************************************************

- :0: Verilog $finish
make[1]: Leaving directory '/afs/andrew.cmu.edu/usr16/sutongy/private/18224/ex9-scanning-chains-dM2w'
(OSS CAD Suite) sutongy@linux-01:~/private/18224/ex9-scanning-chains-dM2w$ 
```

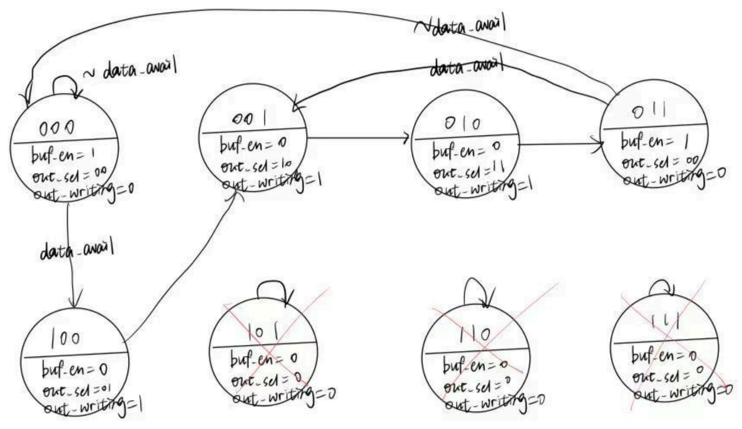
Design Artifacts: The scan chain-based adder design follows a serial input and output mechanism for loading operands and retrieving results. The key components of the design include:

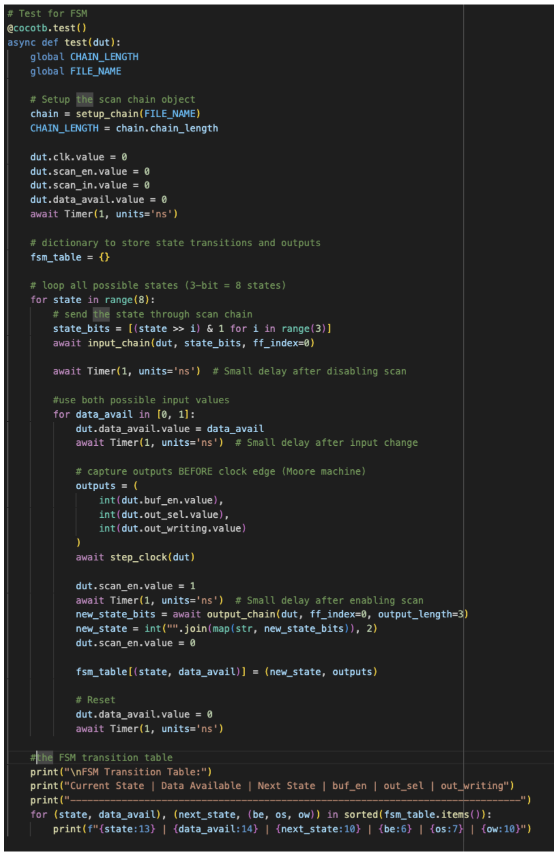
1. Clocking Mechanism:
    - The design relies on a sequential clocking system to shift values into the scan chain and retrieve results.
    - The step_clock() function ensures controlled timing for serial data movement.
2. Functional Units:
    - The adder logic operates once the scan enable signal is disabled, processing the loaded values.
    - The sum is then extracted using a predefined sequence of scan-out operations.
3. Testing Approach:
    - The testbench loads operands serially into the scan chain.
    - Expected results are computed for validation.
    - The computed sum is extracted bit-by-bit and compared against expectations.

Reflection: The primary challenge in testing the adder design was ensuring the correctness of serial bit-shifting operations within the scan chain.

1. Bit Ordering:
    - Initial tests revealed incorrect sum values due to bit misalignment in the input shifting process.
    - This was corrected by reversing the input sequence before shifting.
2. Test Coverage Improvements:
    - Beyond simple addition, test cases were expanded to include edge cases. zero values, carry-over scenarios.
    - A structured test framework was implemented to allow automated execution of multiple test cases.
3. Learn:
    - Understanding how serial scan chains function in real hardware is critical for debugging and validation.
    - Properly structuring test cases in a modular fashion ensures reusability and scalability for future enhancements.

Task 3:





```python
# Test for FSM
@cocotb.test()
async def test(dut):
    global CHAIN_LENGTH
    global FILE_NAME

    # Setup the scan chain object
    chain = setup_chain(FILE_NAME)
    CHAIN_LENGTH = chain.chain_length

    dut.clk.value = 0
    dut.scan_en.value = 0
    dut.scan_in.value = 0
    dut.data_avail.value = 0
    await Timer(1, units='ns')

    # dictionary to store state transitions and outputs
    fsm_table = {}

    # loop all possible states (3-bit = 8 states)
    for state in range(8):
        # send the state through scan chain
        state_bits = [(state >> i) & 1 for i in range(3)]
        await input_chain(dut, state_bits, ff_index=0)

        await Timer(1, units='ns')  # Small delay after disabling scan

        #use both possible input values
        for data_avail in [0, 1]:
            dut.data_avail.value = data_avail
            await Timer(1, units='ns')  # Small delay after input change

            # capture outputs BEFORE clock edge (Moore machine)
            outputs = (
                int(dut.buf_en.value),
                int(dut.out_sel.value),
                int(dut.out_writing.value)
            )
            await step_clock(dut)

            dut.scan_en.value = 1
            await Timer(1, units='ns')  # Small delay after enabling scan
            new_state_bits = await output_chain(dut, ff_index=0, output_length=3)
            new_state = int("".join(map(str, new_state_bits)), 2)
            dut.scan_en.value = 0

            fsm_table[(state, data_avail)] = (new_state, outputs)

            # Reset
            dut.data_avail.value = 0
            await Timer(1, units='ns')

    #the FSM transition table
    print("\nFSM Transition Table:")
    print("Current State | Data Available | Next State | buf_en | out_sel | out_writing")
    print("------------------------------------------------------------------------------")
    for (state, data_avail), (next_state, (be, os, ow)) in sorted(fsm_table.items()):
        print(f"{state:13} | {data_avail:14} | {next_state:10} | {be:6} | {os:7} | {ow:10}")
```

Design Artifacts:
State Setup via Scan Chain: The FSM’s state is set through the scan chain (which simulates the loading of state values into flip-flops).

Test of Input Combinations: For each state, the data_avail input is tested with both possible values (0 and 1) to observe how the FSM behaves in different conditions.

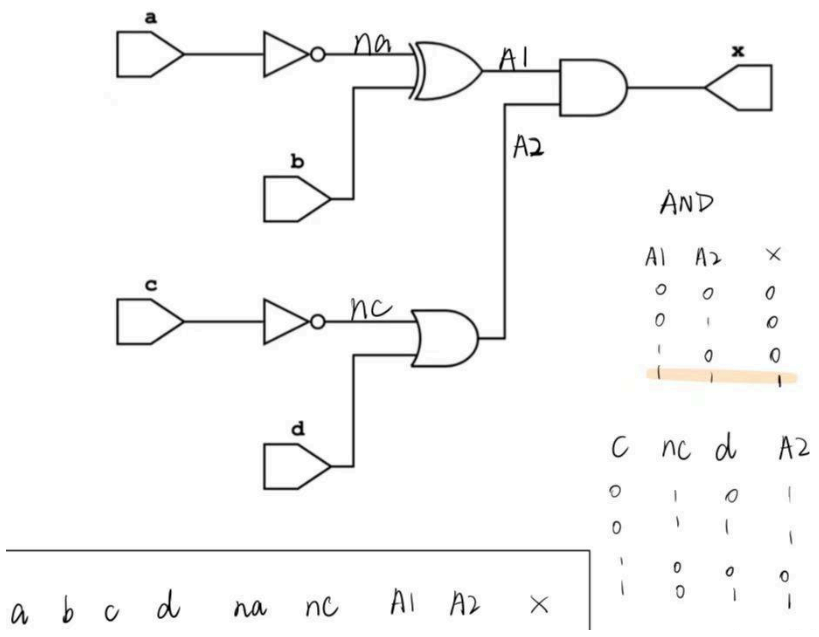
Output Monitoring: The outputs (buf_en, out_sel, out_writing) are monitored before and after the clock edge to capture the FSM's response.

State Transition Verification: The new state is read through the scan chain after the clock edge to verify the correct transition.

FSM Transition Table: The state transitions and output values are logged in a dictionary and then printed for analysis.

What i learned:
State Encoding and Transition Verification: Ensuring that the FSM handles all state encodings and transitions correctly is critical for the correct operation of the system. Testing all possible states and inputs ensures no state is overlooked, and we can catch edge cases.
Handling Outputs in FSMs: Understanding how the outputs are derived from the state and inputs is just as important as verifying state transitions. It’s crucial to ensure that outputs are generated at the correct time and match expectations.
Impact of Timing Delays: the importance of timing in simulations. The small delays between signal changes and the clock edges are necessary to allow the FSM to propagate and stabilize its outputs before they are sampled

Task 4:
Label wires like this:



AND

A1	A2	x
0	0	0
0	1	0
1	0	0
1	1	1

c	nc	d	A2
0	1	0	1
0	1	1	1
1	0	0	0
1	0	1	1

a	b	c	d	na	nc	A1	A2	x

Faults found and test vector fails:
1: a stuck at 1, (0000 fails)
2: A1 stuck at 1, (1000 fails)
3: x stuck at 0 or A2 stuck at 0, (0000,1100,1111 fail)
4: x stuck at 1, (1000, 1110 fail)
5: no fault

Test vector uses:

```
((0, 0, 0, 0), 1),  # Tests a,b,c stuck-at-1, A1/A2 stuck-at-0, x stuck-at-0
((1, 0, 0, 0), 0),  # Tests a stuck-at-0, b stuck-at-1, A1 stuck-at-1, x stuck-at-1
((1, 1, 0, 0), 1),  # Tests a/b stuck-at-0, c stuck-at-1,  A1/A2/x stuck-at-0
((1, 1, 1, 0), 0),  # Tests d stuck-at-1, c stuck-at-0, A1/A2 stuck-at-1, x stuck-at-1
((1, 1, 1, 1), 1)   # Tests A1/A2/x stuck-at-0, a/b/d stuck-at-0
```

Fault Isolation Strategy:

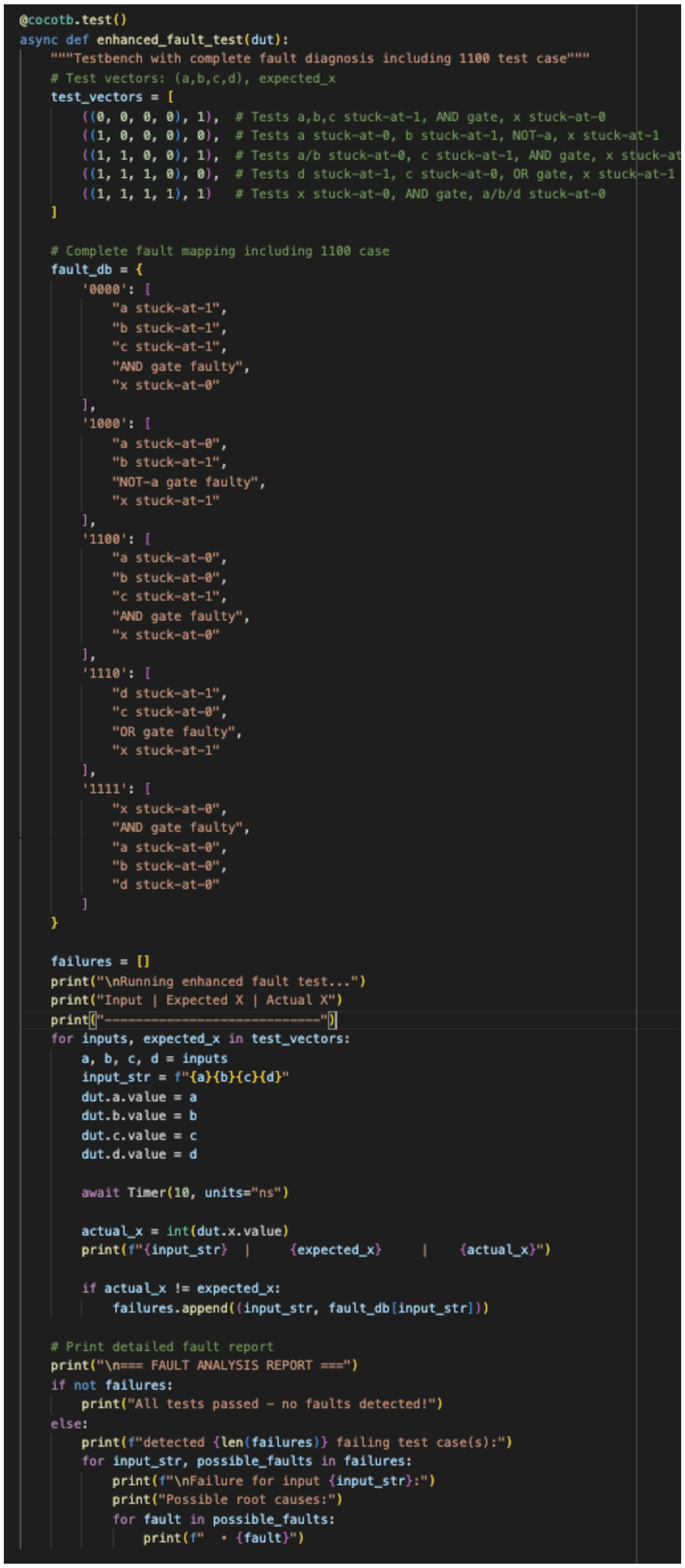
1. Intersection Analysis:
    - Each test vector produces a specific "fault signature" when failures occur
    - By checking which faults appear across multiple failing vectors, we eliminate impossibilities
    - The real fault must explain all observed failures simultaneously
2. Example Diagnosis:

- If only the first test vector (0000) fails → only can be a stuck-at-1 (only fault that doesn't affect other tests)
- If V1 and V5 (1111) fail → only can be x stuck-at-1, since there are only 2 overlapped faults between these 2 tests, A1 stuck-at-0 or x-stuck-at-0, but 1100 will also fails if it’s A1 stuck-at-0, so can only be x-stuck-at-0 if only 0000 and 1111fail

Way to choose the test vectors:

	aSA1	aSA0	bSA1	bSA0	cSA1	cSA0	dSA1	dSA0	A1SA1	A1SA0	A2SA1	A2SA0	X1	X0
0000	✓		✓		✓					✓		✓		✓
0001	✓		✓							✓		✓		✓
0010						✓	✓				✓		✓	
0011	✓		✓					✓		✓		✓		✓
0100	✓			✓					✓				✓	
0101	✓			✓					✓				✓	
0110													✓	
0111	✓			✓					✓				✓	
1000		✓	✓						✓				✓	
1~~0~~001		✓	✓						✓				✓	
1010													✓	
1011		✓	✓						✓				✓	
1100		✓		✓	✓					✓		✓		✓
1101		✓		✓								✓		✓
1110						✓	✓			✓	✓		✓	
1111		✓		✓				✓		✓		✓		✓

TB code:



```python
@cocotb.test()
async def enhanced_fault_test(dut):
    """Testbench with complete fault diagnosis including 1100 test case"""
    # Test vectors: (a,b,c,d), expected_x
    test_vectors = [
        ((0, 0, 0, 0), 1),  # Tests a,b,c stuck-at-1, AND gate, x stuck-at-0
        ((1, 0, 0, 0), 0),  # Tests a stuck-at-0, b stuck-at-1, NOT-a, x stuck-at-1
        ((1, 1, 0, 0), 1),  # Tests a/b stuck-at-0, c stuck-at-1, AND gate, x stuck-at
        ((1, 1, 1, 0), 0),  # Tests d stuck-at-1, c stuck-at-0, OR gate, x stuck-at-1
        ((1, 1, 1, 1), 1)   # Tests x stuck-at-0, AND gate, a/b/d stuck-at-0
    ]

    # Complete fault mapping including 1100 case
    fault_db = {
        '0000': [
            "a stuck-at-1",
            "b stuck-at-1",
            "c stuck-at-1",
            "AND gate faulty",
            "x stuck-at-0"
        ],
        '1000': [
            "a stuck-at-0",
            "b stuck-at-1",
            "NOT-a gate faulty",
            "x stuck-at-1"
        ],
        '1100': [
            "a stuck-at-0",
            "b stuck-at-0",
            "c stuck-at-1",
            "AND gate faulty",
            "x stuck-at-0"
        ],
        '1110': [
            "d stuck-at-1",
            "c stuck-at-0",
            "OR gate faulty",
            "x stuck-at-1"
        ],
        '1111': [
            "x stuck-at-0",
            "AND gate faulty",
            "a stuck-at-0",
            "b stuck-at-0",
            "d stuck-at-0"
        ]
    }

    failures = []
    print("\nRunning enhanced fault test...")
    print("Input | Expected X | Actual X")
    print("-----------------------------")
    for inputs, expected_x in test_vectors:
        a, b, c, d = inputs
        input_str = f"{a}{b}{c}{d}"
        dut.a.value = a
        dut.b.value = b
        dut.c.value = c
        dut.d.value = d

        await Timer(10, units="ns")

        actual_x = int(dut.x.value)
        print(f"{input_str}  |     {expected_x}     |    {actual_x}")

        if actual_x != expected_x:
            failures.append((input_str, fault_db[input_str]))

    # Print detailed fault report
    print("\n=== FAULT ANALYSIS REPORT ===")
    if not failures:
        print("All tests passed - no faults detected!")
    else:
        print(f"detected {len(failures)} failing test case(s):")
        for input_str, possible_faults in failures:
            print(f"\nFailure for input {input_str}:")
            print("Possible root causes:")
            for fault in possible_faults:
                print(f"  • {fault}")
```

Task5:

1. Other Faults & Detection
   - Bridging Faults: Shorts between adjacent wires. detect with:
     - measures quiescent current spikes
     - Voltage contrast imaging (physical inspection)
   - Open Faults: Broken connections. detect with:
     - At-speed testing (timing-sensitive patterns)
     - Boundary scan

2. Scan Chain Trade-offs

Benefits	Downsides
high fault coverage	area overhead
debuggability	power dissipation during test
Automation (ATPG)	performance impact, critical paths
low test time	security risks (data leakage)

3. Testing Secure Circuits Without Scan
   - Built-In Self-Test(mentioned in lec): On-chip test pattern generation/verification.
   - Functional Tests: Cryptographic checksums of known outputs
   - Parametric Monitoring: Ring oscillators to detect timing/power anomalies.
   - Hardware Trojans: Use statistical methods to detect deviations
   -

4. Pre-Optimization Scan Insertion Issues
   - Problem: Scan flops may block logic optimizations like buffering or gate merging
   - Solution
     - Re-timing: Move scan flops post-optimization
     - Partial Scan: Only scan critical flip-flops
     - Synthesis-aware insertion: Use tools that co-optimize scan and logic

Task6:

I am currently working on the design and implementation of the Floating-Point Arithmetic Unit core. The RTL for basic operations like addition, subtraction, multiplication, and division is progressing well, and I have kind of completed the basic architecture for handling IEEE 754 floating-point arithmetic. The control unit and datapath have been structured, and I am working on integrating the SPI interface to receive inputs and return the results.

At this point, I'm focusing on ensuring that the FPU handles edge cases, such as special values (NaN, Infinity) and rounding errors, correctly. I've also started drafting the testbenches for unit testing the FPU modules (but only a little bit). The goal is to complete initial testing and debugging this week, followed by integrating everything into a cohesive system for final validation.

Next steps involve refining the SPI communication with error checking and finalizing the full-system testbenches. I anticipate synthesizing the design soon to ensure it meets timing and performance targets.