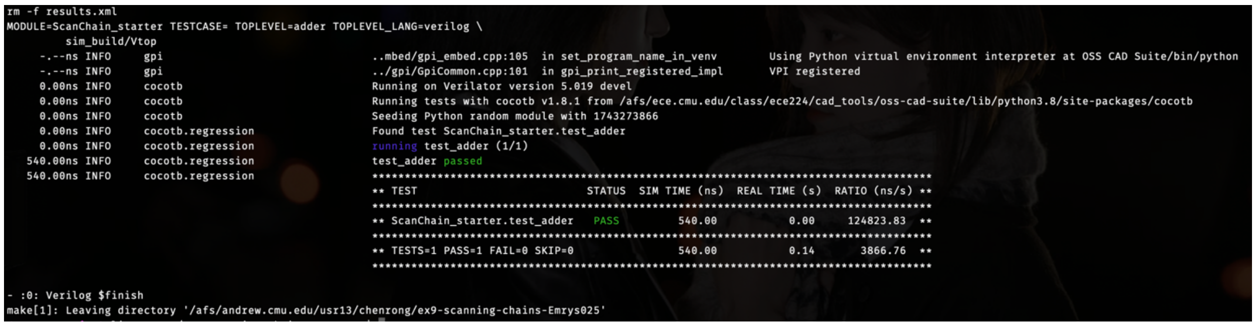
Task #1

The code is included in the ScanChain_starter.py file.

Task #2

The test code for adder design is also included in the ScanChain_starter.py file. This is one of the test results I got:



```
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 interpreter 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/python3.8/site-packages/cocotb
     0.00ns INFO     cocotb                             Seeding Python random module with 1743273866
     0.00ns INFO     cocotb.regression                  Found test ScanChain_starter.test_adder
     0.00ns INFO     cocotb.regression                  running test_adder (1/1)
   540.00ns INFO     cocotb.regression                  test_adder passed
   540.00ns INFO     cocotb.regression                  **************************************************************************************
                                                        ** TEST                          STATUS  SIM TIME (ns)  REAL TIME (s)  RATIO (ns/s) **
                                                        **************************************************************************************
                                                        ** ScanChain_starter.test_adder   PASS         540.00           0.00     124823.83  **
                                                        **************************************************************************************
                                                        ** TESTS=1 PASS=1 FAIL=0 SKIP=0                540.00           0.14       3866.76  **
                                                        **************************************************************************************

- :0: Verilog $finish
make[1]: Leaving directory '/afs/andrew.cmu.edu/usr13/chenrong/ex9-scanning-chains-Emrys025'
```

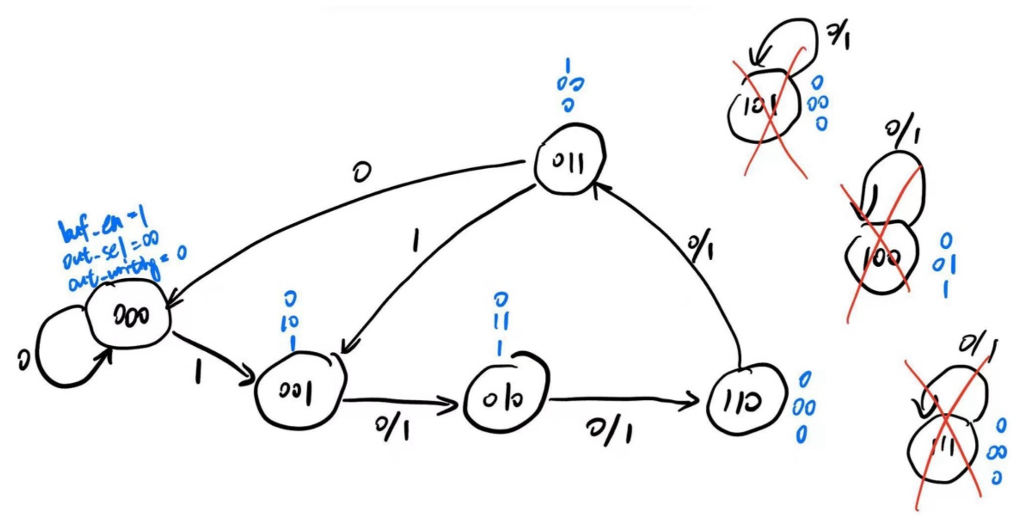
In this part I did not use the provided setup_chain function and the returned ScanChain object, because the dut is fixed and very simple. I took a look at the .log file of the adder, and created a list holding the certain test vector. Then, I used the methods I just wrote to fill in the scan chain, wait a cycle, and get the output from the scan chain.

Originally I misunderstood the index of the x_out register, a_reg register, and b_reg register in the .log file. I thought the indexes of these registers start from left to right. While debugging, I printed out the values before and after calculation. Then I realized the error and reversed the input and output values.

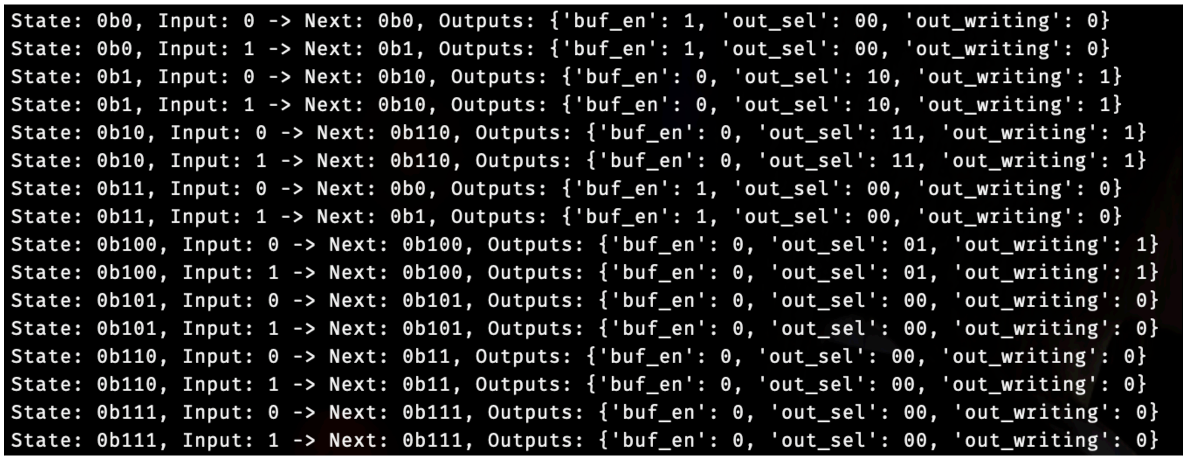
Task #3

The code is included in the ScanChain_starter.py file in a function called test_hidden_fsm.

This is the state diagram I got:



And this is the output I got by running the test code:



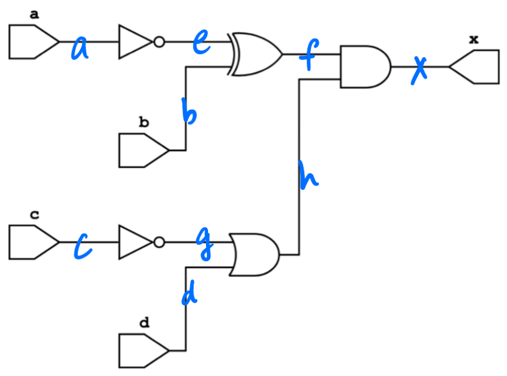
```
State: 0b0, Input: 0 -> Next: 0b0, Outputs: {'buf_en': 1, 'out_sel': 00, 'out_writing': 0}
State: 0b0, Input: 1 -> Next: 0b1, Outputs: {'buf_en': 1, 'out_sel': 00, 'out_writing': 0}
State: 0b1, Input: 0 -> Next: 0b10, Outputs: {'buf_en': 0, 'out_sel': 10, 'out_writing': 1}
State: 0b1, Input: 1 -> Next: 0b10, Outputs: {'buf_en': 0, 'out_sel': 10, 'out_writing': 1}
State: 0b10, Input: 0 -> Next: 0b110, Outputs: {'buf_en': 0, 'out_sel': 11, 'out_writing': 1}
State: 0b10, Input: 1 -> Next: 0b110, Outputs: {'buf_en': 0, 'out_sel': 11, 'out_writing': 1}
State: 0b11, Input: 0 -> Next: 0b0, Outputs: {'buf_en': 1, 'out_sel': 00, 'out_writing': 0}
State: 0b11, Input: 1 -> Next: 0b1, Outputs: {'buf_en': 1, 'out_sel': 00, 'out_writing': 0}
State: 0b100, Input: 0 -> Next: 0b100, Outputs: {'buf_en': 0, 'out_sel': 01, 'out_writing': 1}
State: 0b100, Input: 1 -> Next: 0b100, Outputs: {'buf_en': 0, 'out_sel': 01, 'out_writing': 1}
State: 0b101, Input: 0 -> Next: 0b101, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
State: 0b101, Input: 1 -> Next: 0b101, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
State: 0b110, Input: 0 -> Next: 0b11, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
State: 0b110, Input: 1 -> Next: 0b11, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
State: 0b111, Input: 0 -> Next: 0b111, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
State: 0b111, Input: 1 -> Next: 0b111, Outputs: {'buf_en': 0, 'out_sel': 00, 'out_writing': 0}
```

For this task, unlike the previous task, I used the provided parsing function to parse the log file. As a result, my code can be applied to other state machines.

To reveal the underlying state transitions, I iterated through all possible current states and inputs, captured the corresponding outputs, and stepped one cycle to see that are their next states. Some states are isolated with no transitions in the state diagram. I believe those states are not used in the FSM.

By doing this task, I feel the power of scan chain for testing control flow logic. It can help us uncover the full behavior of the underlying circuit. Therefore, it will help us find the errors.

Task #4



Test vectors (abcd) and the detectable faults:

0000: a SA1, b SA1, c SA1, e SA0, f SA0, g SA0, h SA0, x SA0 (fault if x = 0)

0001: a SA1, b SA1, e SA0, f SA0, h SA0, x SA0 (fault if x = 0)

0010: c SA0, d SA1, g SA1, h SA1, x SA1 (fault if x = 1)

0011: a SA1, b SA1, d SA0, e SA0, f SA0, h SA0, x SA0 (fault if x = 0)

0100: a SA1, b SA0, e SA0, f SA1, x SA1 (fault if x = 1)

0101: a SA1, b SA0, e SA0, f SA1, x SA1 (fault if x = 1)

0110: x SA1 (fault if x = 1)

0111: a SA1, b SA0, e SA0, f SA1, x SA1 (fault if x = 1)

1000: a SA0, b SA1, e SA1, f SA1, x SA1 (fault if x = 1)

1001: a SA0, b SA1, e SA1, f SA1, x SA1 (fault if x = 1)

1010: x SA1 (fault if x = 1)

1011: a SA0, b SA1, e SA1, f SA1, x SA1 (fault if x = 1)

1100: a SA0, b SA0, c SA1, e SA1, f SA0, g SA0, h SA0, x SA0 (fault if x = 0)

1101: a SA0, b SA0, e SA1, f SA0, h SA0, x SA0 (fault if x = 0)

1110: c SA0, d SA1, g SA1, h SA1, x SA1 (fault if x = 1)

1111: a SA0, b SA0, d SA0, e SA1, f SA0, h SA0, x SA0 (fault if x = 0)

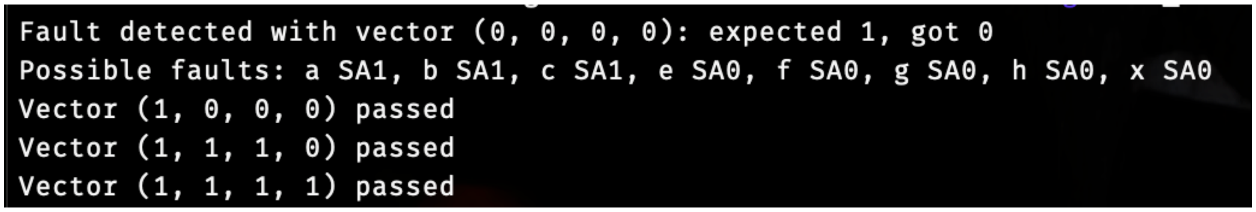
Then, I draw a chart including the above data:

	✓ a SA1	✓ a SA0	✓ b SA1	✓ b SA0	✓ c SA1	✓ c SA0	✓ d SA1	✓ d SA0	✓ e SA1	✓ e SA0	✓ f SA1	✓ f SA0	✓ g SA1	✓ g SA0	✓ h SA1	✓ h SA0	✓ x SA1	✓ x SA0
✓ 0000	✓		✓		✓					✓		✓		✓		✓		✓
0001	✓		✓							✓		✓				✓		✓
0010						✓	✓						✓		✓		✓	
0011	✓		✓					✓		✓		✓				✓		✓
0100	✓			✓						✓	✓						✓	
0101	✓			✓						✓	✓						✓	
0110																	✓	
0111	✓			✓						✓	✓						✓	
✓ 1000		✓	✓						✓		✓						✓	
1001		✓	✓						✓		✓						✓	
1010																	✓	
1011		✓	✓						✓		✓						✓	
1100		✓		✓	✓				✓			✓		✓		✓		✓
1101		✓		✓					✓			✓				✓		✓
✓ 1110						✓	✓						✓		✓		✓	
✓ 1111		✓		✓				✓	✓			✓				✓		✓

As we can see, I think the minimum set of test vectors could be 0000, 1000, 1110, and 1111.

Again, my testbench code is in the file ScanChain_starter.py in the function test_fault_detection.

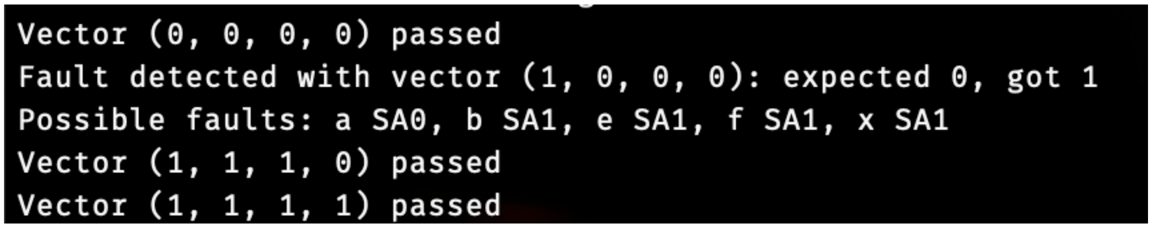
This is my result for fault1.sv:



```
Fault detected with vector (0, 0, 0, 0): expected 1, got 0
Possible faults: a SA1, b SA1, c SA1, e SA0, f SA0, g SA0, h SA0, x SA0
Vector (1, 0, 0, 0) passed
Vector (1, 1, 1, 0) passed
Vector (1, 1, 1, 1) passed
```

The fault is in the first vector while not in the other three vectors. Therefore, it could be "a SA1", "c SA1", "e SA0", or "g SA0".

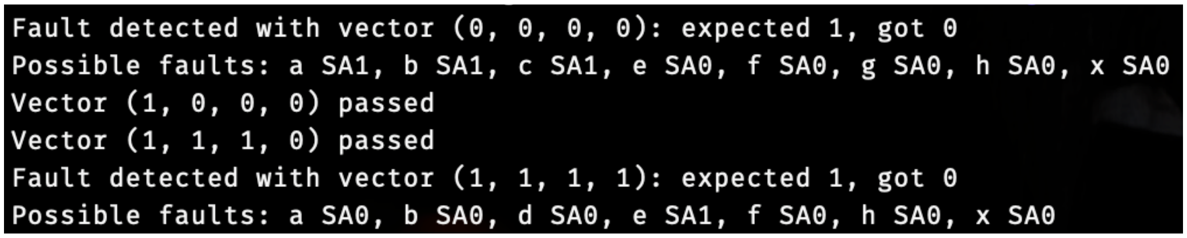
This is my result for fault2.sv:



```
Vector (0, 0, 0, 0) passed
Fault detected with vector (1, 0, 0, 0): expected 0, got 1
Possible faults: a SA0, b SA1, e SA1, f SA1, x SA1
Vector (1, 1, 1, 0) passed
Vector (1, 1, 1, 1) passed
```

The fault is in the second vector while not in the other three vectors. Therefore, it could only be "f SA1".

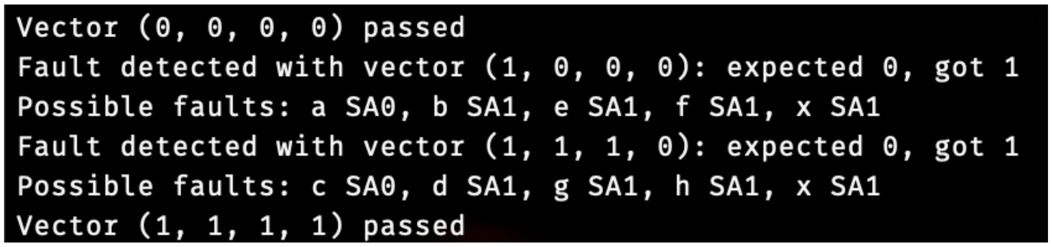
This is my result for fault3.sv:



```
Fault detected with vector (0, 0, 0, 0): expected 1, got 0
Possible faults: a SA1, b SA1, c SA1, e SA0, f SA0, g SA0, h SA0, x SA0
Vector (1, 0, 0, 0) passed
Vector (1, 1, 1, 0) passed
Fault detected with vector (1, 1, 1, 1): expected 1, got 0
Possible faults: a SA0, b SA0, d SA0, e SA1, f SA0, h SA0, x SA0
```

The fault is in the first and the fourth vectors while not in the other two vectors. Therefore, it could be "f SA0", "h SA0", or "x SA0".

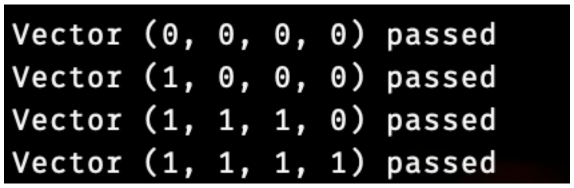
This is my result for fault4.sv:



```
Vector (0, 0, 0, 0) passed
Fault detected with vector (1, 0, 0, 0): expected 0, got 1
Possible faults: a SA0, b SA1, e SA1, f SA1, x SA1
Fault detected with vector (1, 1, 1, 0): expected 0, got 1
Possible faults: c SA0, d SA1, g SA1, h SA1, x SA1
Vector (1, 1, 1, 1) passed
```

The fault is in the second and the third vectors while not in the other two vectors. Therefore, it could only be "x SA1".

This is my result for fault5.sv:



```
Vector (0, 0, 0, 0) passed
Vector (1, 0, 0, 0) passed
Vector (1, 1, 1, 0) passed
Vector (1, 1, 1, 1) passed
```

Seems no fault in this design.

For this part, I actually tried all possible input combinations of abcd. Then, I see if modifying one of the wires will change the result x. Therefore, I recorded all faults detectable with all possible inputs.

Then, in a table, I listed all inputs with their detectable faults. I tried to find the minimal number of inputs that can cover all possible faults in the circuits. I got the result the using 4 test vectors I can cover all possible faults.

Task #5

1. One possibility is that two wires are accidentally connected. Therefore, they are forced to have the same value. This may cause unintended behaviors. To detect them, maybe we can use test vectors that have conflict values on pairs of wires.
   Another possibility is to have disconnected open wires. This may cause metastability and leakage current. We may be possible to detect the current leakage in order to detech the error.
2. The benefits of having a scan chain include simplifying the deterministic testing process, enabling observability and controllability, and reducing test generation complexity.
   The downsides include the overhead of power and area, the need to have a global buffered enable signal, timing overhead, and maybe security risks.
3. One possible thing we can do is to use Built-in Self Test. We can embed the self test circuitry. We can also increase the fault tolerance of the chip, like increasing redundancy and introducing error checking. We can also run a lot of functional tests to verify the overall function of the entire chip. We can also use some scope to check the detailed circuit, but this introduces a huge amount of effort.
4. After running major optimizations, the FFs in the scan chain may be reordered and sometimes can break the test. It may also violate the timing. We can run scan integrity check to check the functionality of the scan chain after PnR optimizations.

Task #6

I have started working on the project and I'm actively approaching the point of having working RTL. I have developed the IO system of my design and currently working on the core computation blocks. It's possible I can finish the RTL next week.