18-224 Exercise 9

Rudy Sorensen

March 30, 2025

1 Task 1

1.1 Parts E - H

Code is uploaded in repo in file ScanChain_starter.py

2 Task 2

Code for testing adder is in ScanChain_starter.py. The functions used are adder_test() and gen_test_case(). I did CRT with 20 sets of inputs to verify my work. I don't have any design artifacts besides the output of my tests to prove their correctness. It is after this paragraph. As for reflecting on my work for this task, I don't have much to say. I basically pulled my gen_test_case() function from exercise 5 and tweaked it a bit so that the inputs were in the form of a list. This allowed me to easily input my test cases into my input_chain() function. I then waited a clock cycle and simply had an assertion that compared the value in the x_out register to the actual sum.

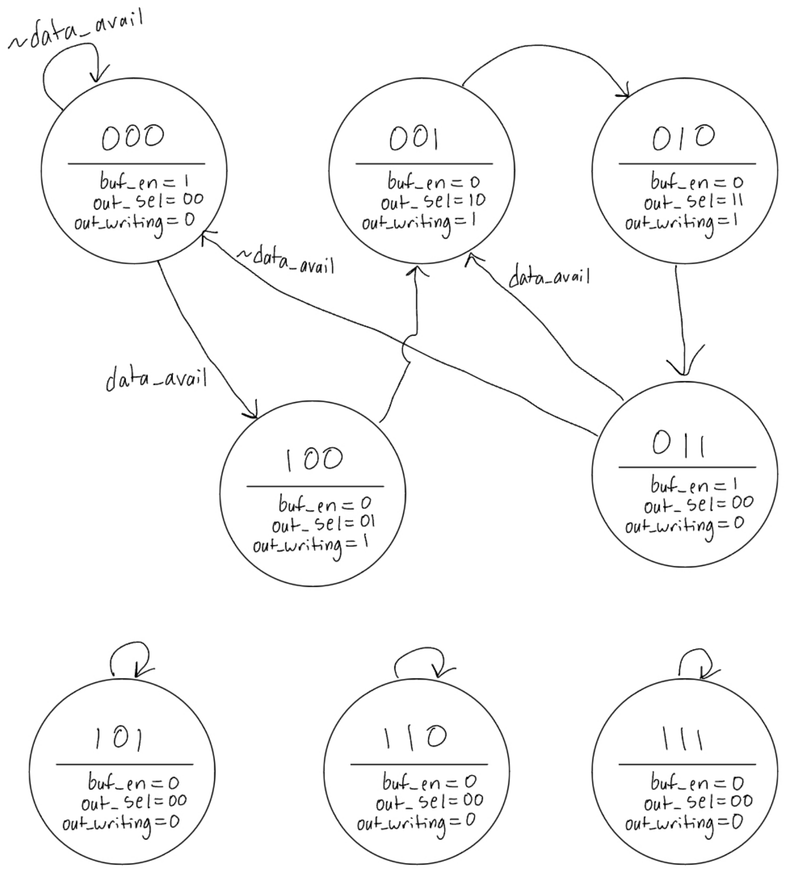
```
TEST 0:
A: [0, 0, 1, 0]
B: [0, 1, 0, 1]
X: 01110
CORRECT SUM:0b1110

TEST 1:
A: [1, 1, 0, 0]
B: [0, 0, 1, 0]
X: 00111
CORRECT SUM:0b111

TEST 2:
A: [1, 1, 1, 0]
B: [0, 0, 0, 0]
X: 00111
CORRECT SUM:0b111

TEST 3:
A: [1, 1, 1, 1]
B: [1, 0, 0, 1]
X: 11000
CORRECT SUM:0b11000

TEST 4:
A: [1, 0, 0, 0]
B: [1, 1, 1, 1]
X: 10000
CORRECT SUM:0b10000

TEST 5:
A: [0, 0, 1, 0]
B: [0, 1, 1, 0]
X: 01010
CORRECT SUM:0b1010

TEST 6:
A: [1, 0, 1, 1]
B: [0, 1, 0, 1]
```

```
X: 10111
CORRECT SUM:0b10111

TEST 7:
A: [1, 1, 1, 0]
B: [1, 1, 0, 0]
X: 01010
CORRECT SUM:0b1010

TEST 8:
A: [0, 0, 0, 1]
B: [0, 0, 0, 0]
X: 01000
CORRECT SUM:0b1000

TEST 9:
A: [1, 0, 0, 1]
B: [1, 0, 0, 0]
X: 01010
CORRECT SUM:0b1010

TEST 10:
A: [0, 1, 0, 1]
B: [0, 1, 0, 1]
X: 10100
CORRECT SUM:0b10100

TEST 11:
A: [1, 1, 0, 0]
B: [1, 0, 1, 1]
X: 10000
CORRECT SUM:0b10000

TEST 12:
A: [1, 1, 0, 1]
B: [0, 0, 0, 0]
X: 01011
CORRECT SUM:0b1011

TEST 13:
A: [0, 0, 0, 1]
B: [1, 1, 0, 1]
X: 10011
CORRECT SUM:0b10011

TEST 14:
A: [0, 0, 0, 0]
B: [0, 0, 1, 0]
X: 00100
CORRECT SUM:0b100

TEST 15:
```

```
A: [0, 1, 1, 1]
B: [1, 0, 0, 0]
X: 01111
CORRECT SUM:0b1111

TEST 16:
A: [0, 0, 1, 0]
B: [1, 1, 1, 1]
X: 10011
CORRECT SUM:0b10011

TEST 17:
A: [1, 0, 0, 1]
B: [0, 0, 1, 0]
X: 01101
CORRECT SUM:0b1101

TEST 18:
A: [0, 0, 1, 0]
B: [0, 1, 1, 0]
X: 01010
CORRECT SUM:0b1010

TEST 19:
A: [0, 0, 0, 0]
B: [1, 1, 0, 0]
X: 00011
CORRECT SUM:0b11
```

3 Task 3

My code for this task is in ScanChain_starter.py under the function hidden_test(). The output of that function is in this document after the reconstructed state diagram. For reflection, I'll start by describing the process. Since I saw that cur_state was only a 3-bit vector in the .log file and that there was only one 1-bit input, I used the scan chain to feed each possible state in twice, once with data_avail set to 0 and then again with data_avail set to 1. Then I simply waited a cycle, used my output_chain() function to grab the next state out of the cur_state registers, and printed that alongside the output variables. I think that it was a pretty cool exercise showing how useful scan chains can actually be even with limited knowledge of the design. It was definitely made easier by it being a Moore machine and only having one 1-bit input though.



```
data_avail: 0
CURR_STATE: [0, 0, 0]
NEXT_STATE: [0, 0, 0]
BUF_EN: 1
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 1
CURR_STATE: [0, 0, 0]
```

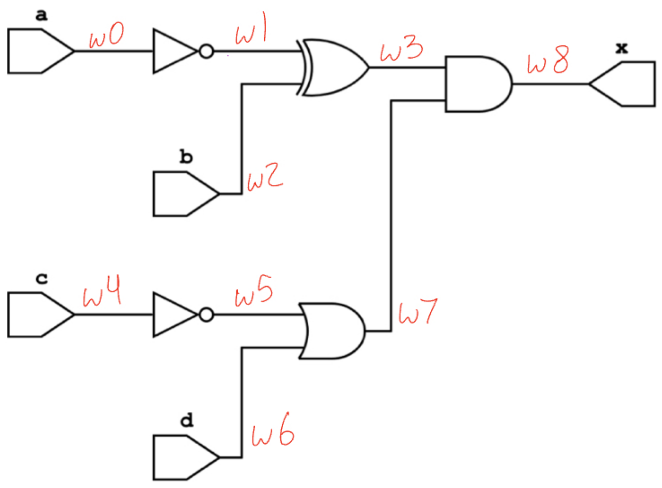
```
NEXT_STATE: [1, 0, 0]
BUF_EN: 1
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 0
CURR_STATE: [0, 0, 1]
NEXT_STATE: [0, 1, 0]
BUF_EN: 0
OUT_SEL: 10
OUT_WRITING: 1

data_avail: 1
CURR_STATE: [0, 0, 1]
NEXT_STATE: [0, 1, 0]
BUF_EN: 0
OUT_SEL: 10
OUT_WRITING: 1

data_avail: 0
CURR_STATE: [0, 1, 0]
NEXT_STATE: [0, 1, 1]
BUF_EN: 0
OUT_SEL: 11
OUT_WRITING: 1

data_avail: 1
CURR_STATE: [0, 1, 0]
NEXT_STATE: [0, 1, 1]
BUF_EN: 0
OUT_SEL: 11
OUT_WRITING: 1

data_avail: 0
CURR_STATE: [0, 1, 1]
NEXT_STATE: [0, 0, 0]
BUF_EN: 1
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 1
CURR_STATE: [0, 1, 1]
NEXT_STATE: [1, 0, 0]
BUF_EN: 1
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 0
CURR_STATE: [1, 0, 0]
NEXT_STATE: [0, 0, 1]
BUF_EN: 0
OUT_SEL: 01
```

```
OUT_WRITING: 1

data_avail: 1
CURR_STATE: [1, 0, 0]
NEXT_STATE: [0, 0, 1]
BUF_EN: 0
OUT_SEL: 01
OUT_WRITING: 1

data_avail: 0
CURR_STATE: [1, 0, 1]
NEXT_STATE: [1, 0, 1]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 1
CURR_STATE: [1, 0, 1]
NEXT_STATE: [1, 0, 1]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 0
CURR_STATE: [1, 1, 0]
NEXT_STATE: [1, 1, 0]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 1
CURR_STATE: [1, 1, 0]
NEXT_STATE: [1, 1, 0]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 0
CURR_STATE: [1, 1, 1]
NEXT_STATE: [1, 1, 1]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0

data_avail: 1
CURR_STATE: [1, 1, 1]
NEXT_STATE: [1, 1, 1]
BUF_EN: 0
OUT_SEL: 00
OUT_WRITING: 0
```

4 Task 4

4.1 Part A

Vectors highlighted the same color will catch more than one fault.



	{d, c, b, a} SA0	SA1	
w0	{X111}, {0X11}	{X100}, {0X00}	w8==0, w0 SA∅ w8==0, w0 SA1
w1	{X100}, {0X00}	{X111}, {0X11}	w8==0, w1 SA∅ w8==0, w1 SA1
w2	{X111}, {0X11}	{X100}, {0X00}	w8==0, w2 SA∅ w8==0, w2 SA1
w3	{X111}, {0X11} $\overset{+}{ab=00}$	{X110}, {X101}, {0X10}, {0X01}	w8==0, w3 SA∅ w8==0, w3 SA1
w4	{0111}, {0100}	{0011}, {0000}	w8==1, w4 SA∅ w8==0, w4 SA1
w5	{0011}, {0000}	{0111}, {0000}	w8==0, w5 SA∅ w8==1, w5 SA1
w6	{1111}, {1100}	{0111}, {0100}	w8==0, w6 SA∅ w8==1, w6 SA1
w7	{1111}, {1100}, ...	{0111}, {0100} ...	w8==0, w7 SA∅ w8==1, w7 SA1
w8	{1111}, {1100}, ...	{0101}, ...	w8==0, w8 SA∅ w8==1, w8 SA1

4.2 Part B

I chose these vectors by starting at the fault and selecting the values of other variables such that they would result in a contradictory output that I could actually view at x. This process resulted in the vectors described in the previous table. I then fed all of the vectors into the design and determined if an error occurred by looking at my generated output.

For fault1.sv, using the vector {1100} I detected:

- **a** possibly SA1
- **w1** possibly SA0
- **b** possibly SA1

For fault2.sv, using the vector {1110} I detected that **w3** was SA1.

For fault3.sv, using the vectors {0011}, {1100}, and {1111} I detected that **x** was SA0 due to contradictions in the other errors I detected.

For fault4.sv, using the vectors {0111}, {1110}, and {0101} I detected that **x** was SA1 due to contradictions in the other errors I detected.

For fault5.sv, I detected no faults.

4.2.1 Outputs

fault1.sv

```
VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b1100
X VAL: 0
A VAL: 0
B VAL: 0
C VAL: 1
D VAL: 1
POSSIBLE w0 SA1
POSSIBLE w1 SA0
POSSIBLE w2 SA1

VEC: 0b1110
X VAL: 1
A VAL: 0
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w3 SA1

VEC: 0b111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 0

VEC: 0b11
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 0
D VAL: 0

VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b101
X VAL: 0
A VAL: 1
```

```
B VAL: 0
C VAL: 1
D VAL: 0
```

fault2.sv

```
VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b1100
X VAL: 1
A VAL: 0
B VAL: 0
C VAL: 1
D VAL: 1

VEC: 0b1110
X VAL: 1
A VAL: 0
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w3 SA1

VEC: 0b111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 0

VEC: 0b11
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 0
D VAL: 0

VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b101
X VAL: 0
A VAL: 1
B VAL: 0
C VAL: 1
D VAL: 0
```

fault3.sv

```
VEC: 0b1111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w0 SA0
POSSIBLE w1 SA1
POSSIBLE w2 SA0
POSSIBLE w3 SA0

VEC: 0b1100
X VAL: 0
A VAL: 0
B VAL: 0
C VAL: 1
D VAL: 1
POSSIBLE w0 SA1
POSSIBLE w1 SA0
POSSIBLE w2 SA1

VEC: 0b1110
X VAL: 0
A VAL: 0
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 0

VEC: 0b11
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 0
D VAL: 0
POSSIBLE w4 SA1
POSSIBLE w5 SA0

VEC: 0b1111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w6 SA0
POSSIBLE w7 SA0
POSSIBLE w8 SA0

VEC: 0b101
X VAL: 0
A VAL: 1
B VAL: 0
C VAL: 1
D VAL: 0
```

fault4.sv

```
VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b1100
X VAL: 1
A VAL: 0
B VAL: 0
C VAL: 1
D VAL: 1

VEC: 0b1110
X VAL: 1
A VAL: 0
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w3 SA1

VEC: 0b111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 0
POSSIBLE w4 SA0
POSSIBLE w5 SA1
POSSIBLE w6 SA1
POSSIBLE w7 SA1

VEC: 0b11
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 0
D VAL: 0

VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b101
X VAL: 1
A VAL: 1
B VAL: 0
```

```
C VAL: 1
D VAL: 0
POSSIBLE w8 SA1
```

fault5.sv

```
VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b1100
X VAL: 1
A VAL: 0
B VAL: 0
C VAL: 1
D VAL: 1

VEC: 0b1110
X VAL: 0
A VAL: 0
B VAL: 1
C VAL: 1
D VAL: 1
POSSIBLE w3 SA1

VEC: 0b111
X VAL: 0
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 0

VEC: 0b11
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 0
D VAL: 0

VEC: 0b1111
X VAL: 1
A VAL: 1
B VAL: 1
C VAL: 1
D VAL: 1

VEC: 0b101
X VAL: 0
A VAL: 1
B VAL: 0
C VAL: 1
D VAL: 0
```

5 Task 5

1. Two other faults that could occur during manufacturing are a crossover fault and a short. A crossover fault could be detected by driving one of the signals affected and seeing if the other signal matches. A short would be detected in a similar way to how we detect stuck-at faults, since they are modelling the same failure I believe.

2. Downsides of having a scan-chain include excess logic, leading to more area, and a lower possible maximum frequency, which can affect timing. Upsides include the ability to test your physical circuit for proper functionality and to determine whether specific intermediate wires have a fault, since you can now feed values into specific points in your circuit.

3. To accomplish this, you can implement a BIST (built-in self test). Circuitry that simply outputs whether the design is working or not and does not require external input nor the need to probe the design's actual outputs to test its funcionality.

4. I would assume that unexpected things could happen if running optimization after having tested the design with a scan chain. Optimization could create a new type of fault that wasn't detected in the unoptimized design, so one must insert a scan chain once again to test. Essentially, the unoptimized and optimized designs could be completely different, so the testing of one is completely unrelated to the testing of another besides their test vectors.

6 Task 6

I have not gotten into writing much RTL yet. I've mostly just read through Section 11 of the USB 1.1 specification, which is the section regarding USB Hub design. I've ironed out some confusion I had and better understand the datapaths and FSMs that are presented in the spec now. At this point, I believe that I'm ready to being writing RTL since I now have a firm grasp on the architecture and module hierarchy needed to accomplish this design.