Task 1 & Task 2

```
#-----------------------------------------------------------------

# This function steps the clock once.

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

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

    return


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

# This function places a bit value inside FF of specified index.

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

async def input_chain_single(dut, bit, ff_index):

    dut.scan_en.value = 1
    dut.scan_in.value = bit
    await step_clock(dut)

    for i in range(ff_index):
      await step_clock(dut)

    dut.scan_en.value = 0

    return

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

# 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 i in range(len(bit_list)):
      dut.scan_in.value = bit_list[0]
      await step_clock(dut)
      # print(f"bit_list[0]: {bit_list[0]}")
      bit_list = bit_list[1:]
```

```
    for i in range(ff_index):
      await step_clock(dut)
    dut.scan_en.value = 0

    return

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

# This function retrieves a single bit value from the
# chain at specified index

async def output_chain_single(dut, ff_index):

    output = 0

    dut.scan_en.value = 1
    for i in range(CHAIN_LENGTH - ff_index):
      output = (output << 1) + dut.scan_out.value
      await step_clock(dut)
    dut.scan_en.value = 0

    return output

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

# 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):

    output = 0
    print(f"first x_out {dut.x_out.value}")

    dut.scan_en.value = 1
    for i in range(CHAIN_LENGTH - (ff_index + output_length)):
      await step_clock(dut)
      print(f"x_out {dut.x_out.value}")

    for i in range(output_length):
      output = (output << 1) + dut.scan_out.value
      await step_clock(dut)
    dut.scan_en.value = 0

    return output

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

# Your main testbench function

@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)

    bit_list = [1, 1, 1, 0, 0, 1, 1, 1, 0, 0, 0, 0, 0] # adding b = 14 and a = 7,
initializing x_out to 0
    # b_reg is first 4 bits with msb in index 0, a_reg is next 4 bits, x_out is
last 5 bits
    await (input_chain(dut, bit_list, 0))
    print(f"after input x_out {dut.x_out.value}")

    await step_clock(dut)
    outval = await (output_chain(dut, 0, 5))
    print(f'this is what 14 + 7 is: {outval}')
    assert (outval == 21)
```

Output

Testing the Adder involved shifting in different values for a and b, stepping the clock to propagate the outputs, and then seeing if the output shifted out of x_out is what we expect.

Task 3

The below code was used with the same testing framework.

```
@cocotb.test()
async def test000(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)
```

```
    bit_list = [[0, 0, 0], [0, 0, 1], [0, 1, 0], [0, 1, 1], [1, 0, 0], [1, 0, 1],
[1, 1, 0], [1, 1, 1]]

    for i in range(8):
      print(f'===== CURRENT STATE: {bit_list[i]} ===== ')
      await (input_chain(dut, bit_list[i], 0))

      dut.data_avail.value = 0
      await step_clock(dut)
      print(f"clk {dut.clk.value} data_avail {dut.data_avail.value}")
      print(f"buf_en {dut.buf_en.value} out_sel {dut.out_sel.value} out_writing
{dut.out_writing.value}")
      outval = await (output_chain(dut, 0, 3))
      print(f'next state: {outval}')
      print()

      dut.data_avail.value = 1
      await step_clock(dut)
      print(f"clk {dut.clk.value} data_avail {dut.data_avail.value}")
      print(f"buf_en {dut.buf_en.value} out_sel {dut.out_sel.value} out_writing
{dut.out_writing.value}")
      outval = await (output_chain(dut, 0, 3))
      print(f'next state: {outval}')
      print()

    assert True
```

Output:

```
===== CURRENT STATE: [0, 0, 0] =====
clk 0 data_avail 0
buf_en 1 out_sel 00 out_writing 0
next state: 0

clk 0 data_avail 1
buf_en 0 out_sel 01 out_writing 1
next state: 4

===== CURRENT STATE: [0, 0, 1] =====
clk 0 data_avail 0
buf_en 0 out_sel 11 out_writing 1
next state: 2

clk 0 data_avail 1
buf_en 0 out_sel 00 out_writing 0
next state: 7

===== CURRENT STATE: [0, 1, 0] =====
clk 0 data_avail 0
buf_en 1 out_sel 00 out_writing 0
next state: 3
```

```
===== CURRENT STATE: [1, 0, 0] =====
clk 0 data_avail 0
buf_en 0 out_sel 10 out_writing 1
next state: 1

clk 0 data_avail 1
buf_en 0 out_sel 01 out_writing 1
next state: 4

===== CURRENT STATE: [1, 0, 1] =====
clk 0 data_avail 0
buf_en 0 out_sel 00 out_writing 0
next state: 5

clk 0 data_avail 1
buf_en 0 out_sel 00 out_writing 0
next state: 7

===== CURRENT STATE: [1, 1, 0] =====
clk 0 data_avail 0
buf_en 0 out_sel 00 out_writing 0
next state: 6
```

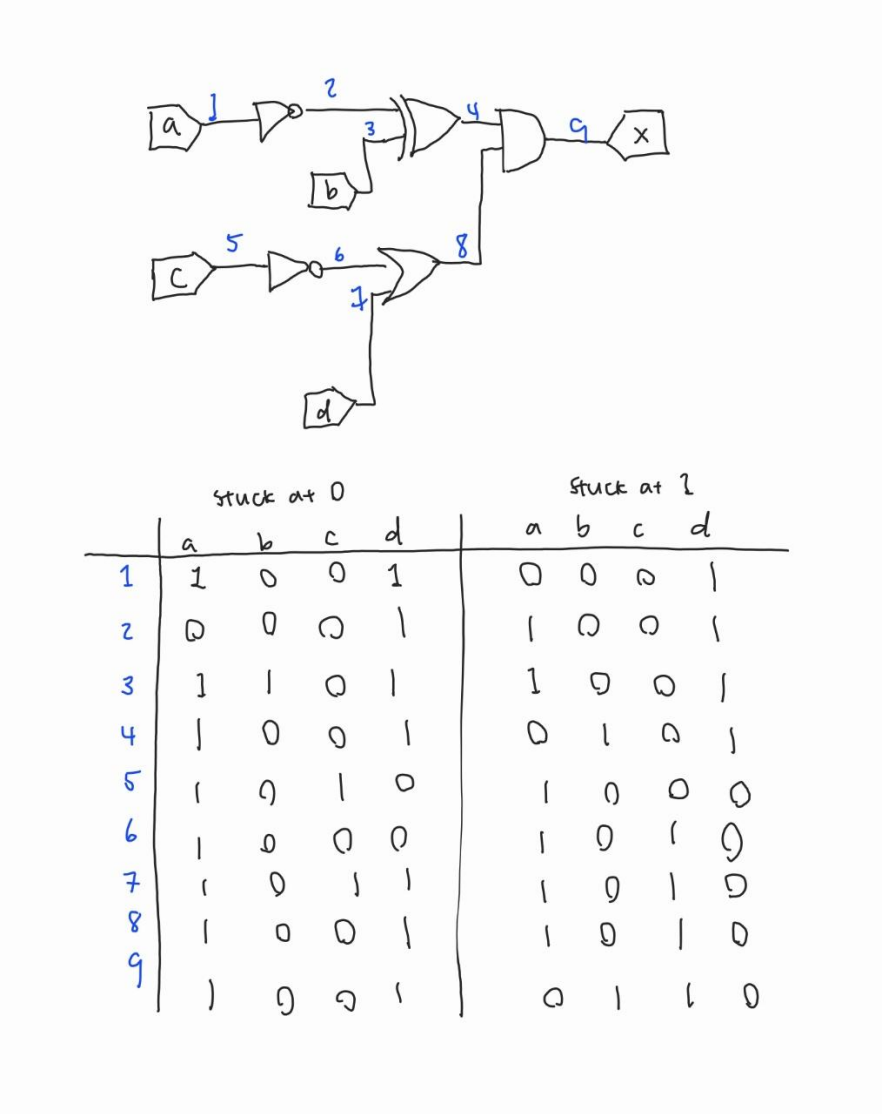
clk 0 data_avail 1 buf_en 0 out_sel 01 out_writing 1 next state: 4 ===== CURRENT STATE: [0, 1, 1] ===== clk 0 data_avail 0 buf_en 1 out_sel 00 out_writing 0 next state: 0 clk 0 data_avail 1 buf_en 0 out_sel 00 out_writing 0 next state: 7	clk 0 data_avail 1 buf_en 0 out_sel 01 out_writing 1 next state: 4 ===== CURRENT STATE: [1, 1, 1] ===== clk 0 data_avail 0 buf_en 0 out_sel 00 out_writing 0 next state: 7 clk 0 data_avail 1 buf_en 0 out_sel 00 out_writing 0 next state: 7

Note that the buf_en, out_sel, and out_writing outputs correspond with the next_state outputs as we have switched to that state after the clock step.

This process is a lot easier because we are able to use the scan chain to input the state values. If that were not an option, we would have to keep track of the entire sequence of

inputs and construct states by seeing when we have different outputs and when the sequence of different outputs is different, which is a lot more complicated.

Task 4



	Stuck at 0				Stuck at 1			
	a	b	c	d	a	b	c	d
1	1	0	0	1	0	0	0	1
2	0	0	0	1	1	0	0	1
3	1	1	0	1	1	0	0	1
4	1	0	0	1	0	1	0	1
5	1	0	1	0	1	0	0	0
6	1	0	0	0	1	0	1	0
7	1	0	1	1	1	0	1	0
8	1	0	0	1	1	0	1	0
9	1	0	0	1	0	1	1	0

```
async def test_vector(dut, vector):

  dut.a.value = vector[0]
  dut.b.value = vector[1]
  dut.c.value = vector[2]
  dut.d.value = vector[3]
  await Timer(10, units='ns')

  not_a = not vector[0]
  not_c = not vector[2]
```

```
    xor_ab_inv = (not_a ^ vector[1])
    or_cd_inv = (not_c | vector[3])
    and_result = xor_ab_inv & or_cd_inv

    print(f"vector: {vector}")
    print(f"expected {and_result}")
    print(f"actual result {dut.x.value}")
    print()

    # assert (and_result == dut.x.value)

@cocotb.test()
async def basic_test(dut):
    print("============== STARTING TEST 1 ==============")

    vectors = [
    [0, 0, 0, 1],
    [0, 1, 0, 1],
    [0, 1, 1, 0],
    [1, 0, 0, 0],
    [1, 0, 0, 1],
    [1, 0, 1, 0],
    [1, 0, 1, 1],
    [1, 1, 0, 1]
    ]

    for vector in vectors:
      await test_vector(dut, vector)

    assert True
```

I chose the test vectors by seeing which values would make it so that the wire we are testing would make a difference in the output.

Fault1.sv

```
============== STARTING TEST 1 ==============
vector: [0, 0, 0, 1]
expected 1
actual result 0

vector: [0, 1, 0, 1]
expected 0
actual result 1

vector: [0, 1, 1, 0]
expected 0
actual result 0

vector: [1, 0, 0, 0]
expected 0
actual result 0

vector: [1, 0, 0, 1]
expected 0
actual result 0

vector: [1, 0, 1, 0]
expected 0
actual result 0

vector: [1, 0, 1, 1]
expected 0
actual result 0

vector: [1, 1, 0, 1]
expected 1
actual result 1
```

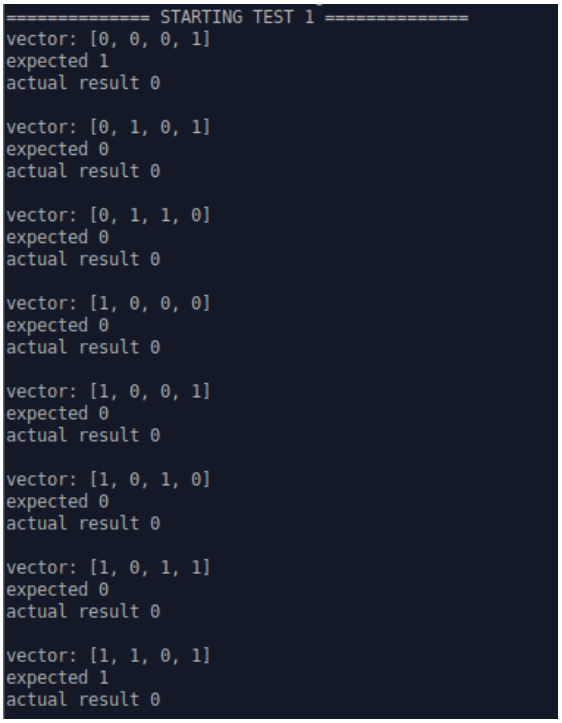
Since we have a failure for vector 0001, we might have a stuck at 1 issue at wire 1. Because of 0101, we may have stuck at 1 at wire 4.

Fault2.sv

```
============== STARTING TEST 1 ==============
vector: [0, 0, 0, 1]
expected 1
actual result 1

vector: [0, 1, 0, 1]
expected 0
actual result 1

vector: [0, 1, 1, 0]
expected 0
actual result 0

vector: [1, 0, 0, 0]
expected 0
actual result 1

vector: [1, 0, 0, 1]
expected 0
actual result 1

vector: [1, 0, 1, 0]
expected 0
actual result 0

vector: [1, 0, 1, 1]
expected 0
actual result 1

vector: [1, 1, 0, 1]
expected 1
actual result 1
```

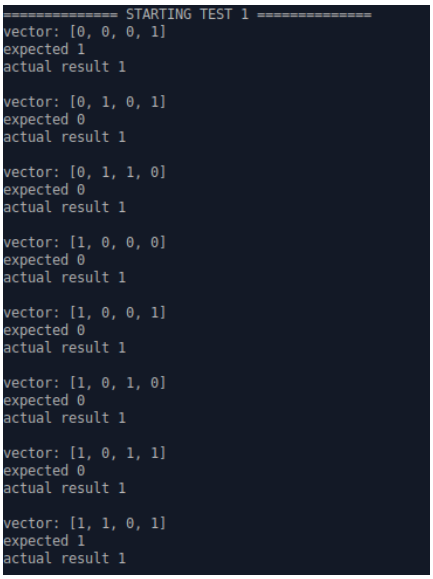
For 0101, we may have a stuck at 1 for wire 4. For 1000, we may have wire 5 stuck at 1 or wire 6 stuck at 0. For 1001, we may have wire 2 or 3 stuck at 1, or wire 8 stuck at 1. For 1011, wire 7 might be stuck at 0.

Fault3.sv



For 0001, we may have a stuck at 1 for wire 1 stuck at 0 for wire 2, or stuck at 1 for wire 8. For 1101, we may have a stuck at 0 for wire 3.

Fault4.sv



For 0101, we may have a stuck at 1 for wire 4.

For 0110, we have a stuck at 1 for wire 9. This also seems like the most likely result since all of the vectors are outputting 1.

For 1000, we may have wire 5 stuck at 1 or wire 6 stuck at 0.

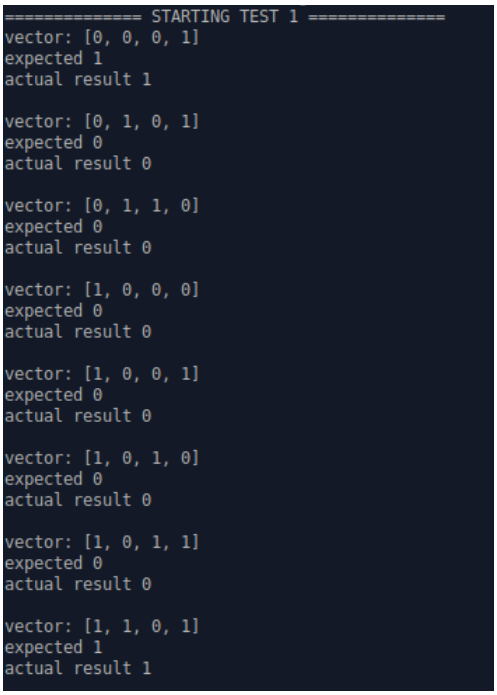
For 1001, we may have wire 2 or 3 stuck at 1, or wire 8 stuck at 0.

For 1010, we may have a stuck at 1 on 6 or 7 or stuck on 1 at wire 8.

For 1011, wire 7 might be stuck at 0.

Note that we can have one or more of these faults. For example, our wire 9 could be stuck at 1, or both wire 4 and wire 8 are stuck at 1. Since this assignment is testing for single faults, our wire 9 is stuck at 1.

Fault5.sv



There are no test vectors that output incorrect values, so there are likely no singular stuck-at issues with this circuit.

It is also possible to decode the numbers presented in the code into individual outputs for inputs {a, b, c, d} by shifting right however many times (ex: a = 0, b = 1, c = 1, d = 0, then we shift right 6 times and the output is the LSB).

Task 5

1. *Think about at least two other faults that could occur during manufacturing (other than stuck-at). How would you detect them?*

One fault that we can have is a short where we are supposed to have a gate. To test this fault, we can set the inputs to be such that the gate gives a 1 when an input is a 0, or vice versa, and seeing if the output uses the opposite value.

Another fault we can have is having multiple wires stuck. If we aren't sure if there are multiple wires stuck, we might have to run all of the input possibilities and see if the outputs match. If we are isolating two wires that are stuck, then we can set one wire to be where the other would decide the output (ex: in nand, we set one wire to 1 so that the output depends on the other), and then toggle the other wire to see if there is a change in the output, and then do the same the other way.

2. *What are some of the trade-offs involved in having a scan chain (both benefits and downsides)?*

The scan chain allows for easier testing of logic to be able to narrow down root causes of issues. It also allows for a much wider range of detection for different faults since it is able to control the internal states of flip flops. It also makes it easier to automatically generate test vectors. However, the downsides are that it adds area to the design and increases the power consumption and usually delay through the circuit because of the added muxes at each dff.

3. *Let's say you have some part of a circuit in which you don't want to include a scan chain or external testing points (for example, a cryptographic processor that you want to isolate from external access for security purposes). How can you ensure that you're not shipping potentially-broken silicon to an end-user?*

We can use built-in-self-test to include hardware that can test that small module within the chip without having to interface with an external tester. Also, we can do lots of simulation to verify that the hardware will function as expected with all the different process bounds. Lastly, we can have redundant gates / fail-safe that allow the module to be more robust and less sensitive to production defects.

4. *The designs used in this assignment had the scan-chain inserted before any major optimizations were run. Can you think of any issues this may lead to, and how might you deal with them?*

The optimizer might place importance on routing the scan chain instead of placing important critical component paths next to each other, which may affect the circuit

functionality greatly. We can deal with this by putting in the scan chain after the optimizations are done to the circuit. We can also iterate and place constraints on the scan chain so that it does not impede on our design

Task 6

I have created a basic SystemVerilog module to obtain data from a PDM microphone. This is done without oversampling and averaging. While I have written this sv file, there is still testing to be done.

```
module pdm_to_pcm(
  input clk,          // System clock
  input pdm_in,       // PDM microphone output
  output logic [7:0] pcm_out // 8-bit PCM output
);

  localparam int SAMPLING_RATE = 256;
  localparam int NUM_BITS = 8 // log2(256)

  logic [SAMPLING_RATE-1:0] pdm_buffer;
  logic [NUM_BITS-1:0] pdm_buffer_index;
  logic [7:0] accumulator;

  always_ff @(posedge clk) begin
    if (pdm_buffer_index = 8'b11111111) begin
      pcm_out = accumulator;
      accumulator = 0;
    end
    else begin
      if (pdm_in) begin
        accumulator = accumulator + 1;
      end
    end
  end

endmodule
```

Using oversampling:

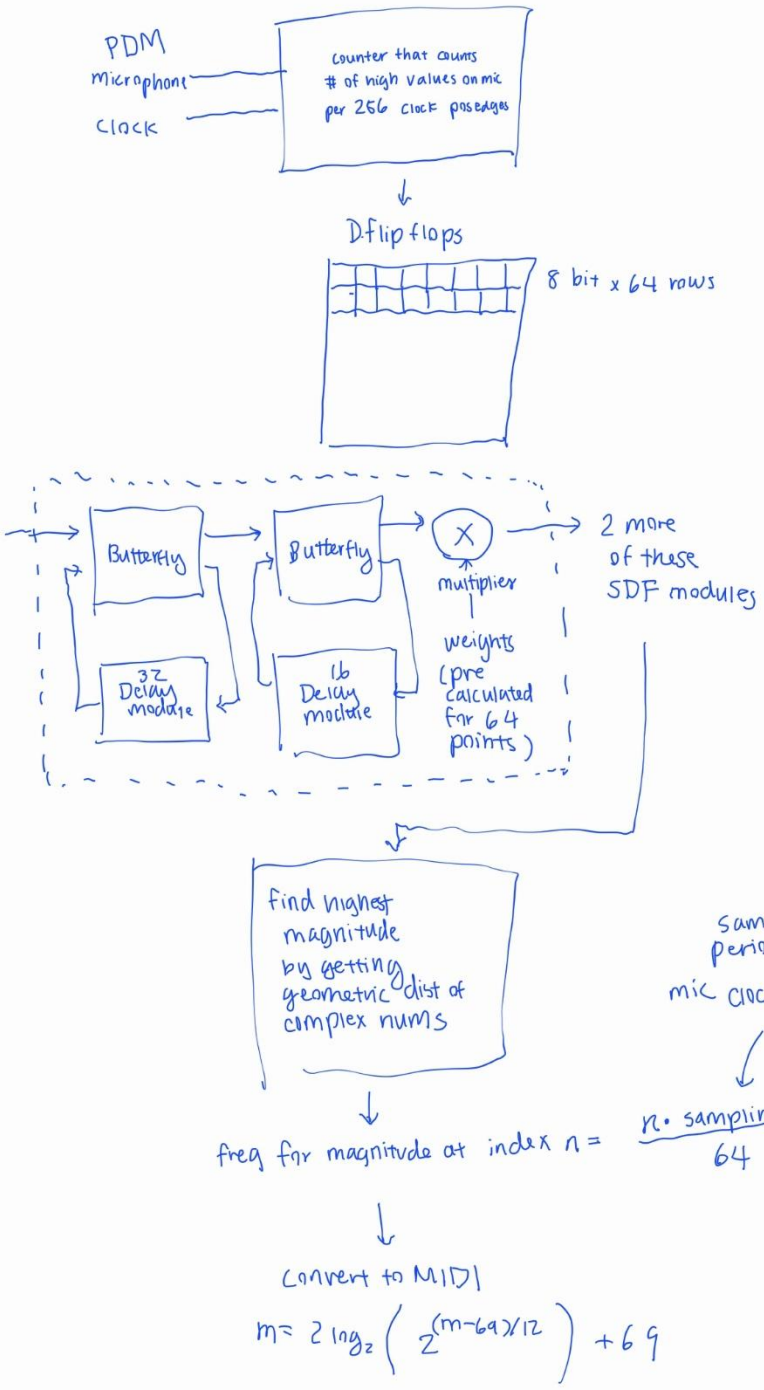
```
module pdm_to_pcm(
  input clk,          // System clock
  input pdm_in,       // PDM microphone output
  output logic [7:0] pcm_out // 8-bit PCM output
);

  localparam int SAMPLING_RATE = 256;
```

```
  localparam int NUM_BITS = 8 // log2(256)
  localparam int OVERSAMPLING_FACTOR = 4;

  logic [SAMPLING_RATE * OVERSAMPLING_FACTOR-1:0] pdm_buffer;
  logic [10-1:0] pdm_buffer_index;
  logic [10-1:0] accumulator;

  always_ff @(posedge clk) begin
    if (pdm_buffer_index = 8'b1111111111) begin
      pcm_out = accumulator / 4;
      accumulator = 0;
    end
    else begin
      if (pdm_in) begin
        accumulator = accumulator + 1;
      end
    end
  end

endmodule
```

Also for the max complex number from 64 row x 8 bit inputs.

```
module find_max_complex_magnitude(
  input clk,
  input rst_n,
  input logic signed [7:0] real_in [63:0], // Array of 64 real parts
  input logic signed [7:0] imag_in [63:0], // Array of 64 imaginary parts
  output logic [5-1:0] max_index_out,
  output logic max_valid
);

  localparam NUM_INPUTS = 64;
  localparam DATA_WIDTH = 8;
  localparam INDEX_WIDTH = 5;

  logic signed [2*DATA_WIDTH-1:0] magnitude_squared [NUM_INPUTS];
  logic signed [2*DATA_WIDTH-1:0] current_max_magnitude_squared;
  logic [INDEX_WIDTH-1:0] current_max_index;
  logic [INDEX_WIDTH-1:0] index_reg;
  assign compare_enable = (index_reg < NUM_INPUTS - 1);

  for (genvar i = 0; i < NUM_INPUTS; i++) begin : gen_magnitude_squared
    assign magnitude_squared[i] = real_in[i] * real_in[i] + imag_in[i] *
imag_in[i];
  end

  always_ff @(posedge clk or negedge rst_n) begin
```

```
    if (!rst_n) begin
      current_max_magnitude_squared <= 0;
      current_max_index <= 0;
      index_reg <= 0;
      max_valid <= 0;
    end else begin
      if (index_reg == 0) begin
        current_max_magnitude_squared <= magnitude_squared[0];
        current_max_index <= 0;
      end
      else if (magnitude_squared[index_reg] >
current_max_magnitude_squared) begin
        current_max_magnitude_squared <= magnitude_squared[index_reg];
        current_max_index <= index_reg;
      end

      if (compare_enable) begin
        index_reg <= index_reg + 1;
        max_valid <= 0;
      end
      else begin
        max_valid <= 1;
      end
    end
  end

  assign max_index_out = current_max_index;

  endmodule
```


PDM
Microphone
Clock
counter that counts # of high values on mic per 256 clock posedges
D flip flops
8 bit x 64 rows
Butterfly
Butterfly
X
multiplier
2 more of these SDF modules
32 Delay module
16 Delay module
weights (pre calculated for 64 points)
find highest magnitude by getting geometric dist of complex nums
sampling period = mic clock x 256
freq for magnitude at index n = n · sampling freq / 64
convert to MIDI
$m = 12 \log_2\left(2^{(n-69)/12}\right) + 69$