The EE457 lab on FIFO has two parts: (i) Designing a single-clock FIFO and (ii) An application using the FIFO in an RTL design.
The second part is an interesting RTL design but we do not have time in the semester to complete this. For several semesters, the part #2 was not being assigned.

FIFO Design and Application

You need to read only the uncrossed parts of the two pages, page 1/6 and 5/6.

Objective:

- Design a single-clock FIFO in two ways
- ~~Use the FIFO in a design~~

Description:

We provide you one design for the single-clock FIFO and ask you to design the same in a slightly different way.

~~Then you use the FIFO *given by us* in an application to complete an RTL design.~~

Single-Clock FIFO Design:

You first go through the single-clock and the two-clock FIFO design lecture posted on the Black-board (FIFO_1.pdf, FIFO_1.wmv). As you would have noticed from the lecture, one can use (n+1)-bit counters as write pointer (WP) and read pointer (RP) and easily derive depth as [WP – RP]mod 2**(n+1). For example, for a 16 location FIFO, even though 2**4 is 16, we go for 5-bit counters for WP and RP and produce depth by doing [WP-RP] mod 32. This is the way to produce FULL and EMPTY inferences in the 2-clock FIFO and this can also be used in the single-clock FIFO also. Besides this method, in the case of a single-clock FIFO, it is possible to use just n-bit pointers. The problem is that, with n-bit pointers, [WP-RP] can become zero in two totally different situations: (i) when the FIFO is EMPTY and (ii) when the FIFO is FULL. To deal with this situation, as discussed in the FIFO lecture, we use a FLAG flip-flop to remember/record if the FIFO was *most recently* (before this ambiguous situation occurs) running *almost full* or *almost empty*. If the FIFO was running *almost full* recently, we interpret [WP-RP] == 0 as indicating the "FULL" state. Similarly, if the FIFO was running *almost empty* recently, we interpret [WP-RP] == 0 as indicating the "EMPTY" state. You were given the (n+1)-bit-pointers based FIFO design and you were asked to complete a n-bit-pointers based FIFO design. One of the aspects we want you to learn in coding the FIFO is parameterized design.

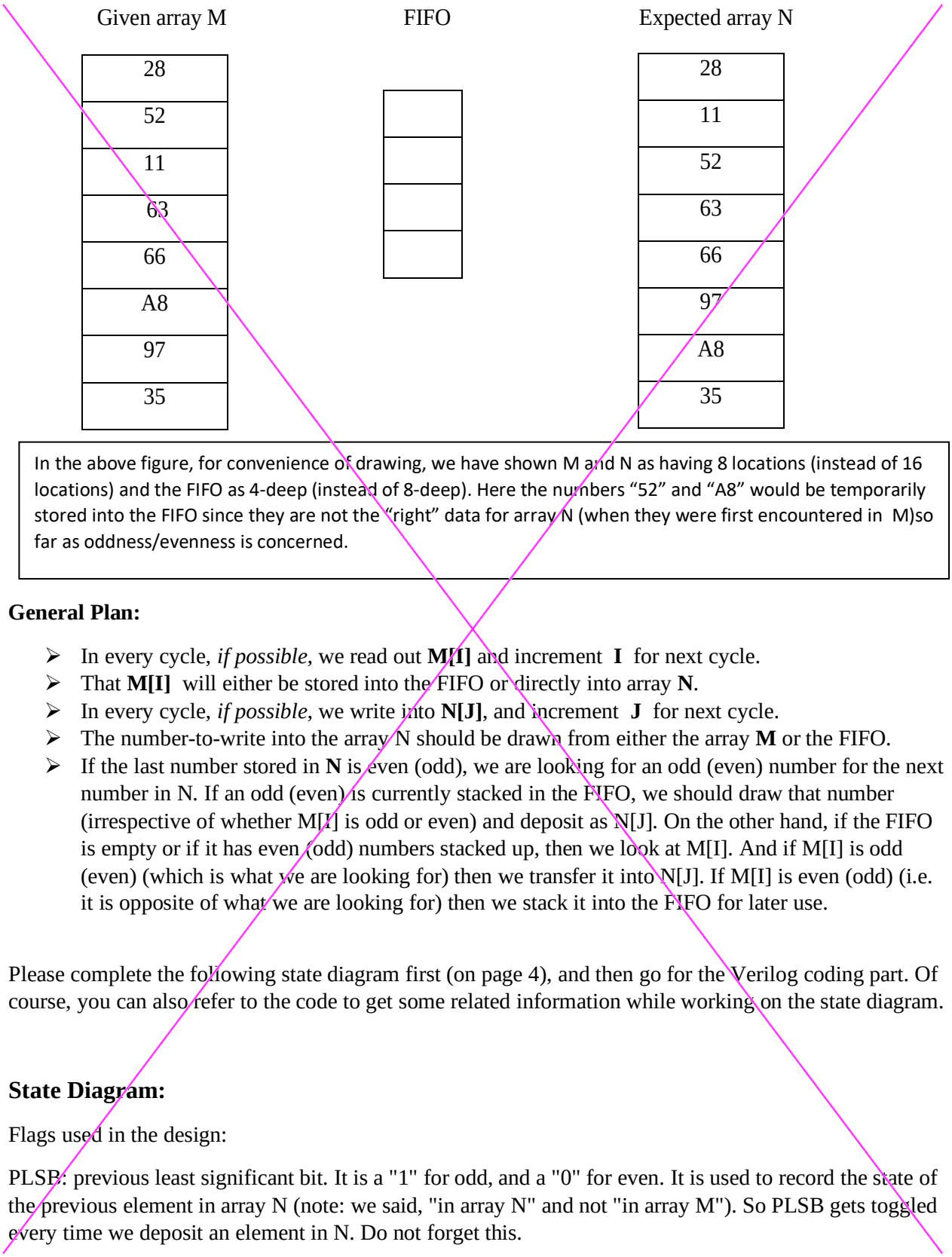
~~FIFO application in a RTL design~~:

An array **M** of 16 numbers is given and we were told that there are 8 even numbers and 8 odd numbers in the array, but they may not be alternating. Our job is to transfer them to the array **N** in such way that they are alternating between odd and even. We need to do so without disturbing the relative order among the even numbers and similarly without disturbing the relative order among the odd numbers. It may not be obvious at the first sight why a FIFO is needed and how it can be used.

Let us use an 8-location FIFO to temporarily store the excess of even numbers or excess of odd numbers. Note that both even and odd cannot be in excess at the same time. e.g.

```
Given array M:          28 51 33 32 86 78 63 11 89 81 33 66 5C 50 2F 8E
8-loc FIFO has O/E:           O        E        O  O  O  O  O  O  O

Expected array N:       28 51 32 33 86 63 78 11 66 89 5C 81 50 33 8E 2F
```



In the above figure, for convenience of drawing, we have shown M and N as having 8 locations (instead of 16 locations) and the FIFO as 4-deep (instead of 8-deep). Here the numbers "52" and "A8" would be temporarily stored into the FIFO since they are not the "right" data for array N (when they were first encountered in M)so far as oddness/evenness is concerned.

General Plan:

- In every cycle, *if possible*, we read out **M[I]** and increment **I** for next cycle.
- That **M[I]** will either be stored into the FIFO or directly into array **N**.
- In every cycle, *if possible*, we write into **N[J]**, and increment **J** for next cycle.
- The number-to-write into the array N should be drawn from either the array **M** or the FIFO.
- If the last number stored in **N** is even (odd), we are looking for an odd (even) number for the next number in N. If an odd (even) is currently stacked in the FIFO, we should draw that number (irrespective of whether M[I] is odd or even) and deposit as N[J]. On the other hand, if the FIFO is empty or if it has even (odd) numbers stacked up, then we look at M[I]. And if M[I] is odd (even) (which is what we are looking for) then we transfer it into N[J]. If M[I] is even (odd) (i.e. it is opposite of what we are looking for) then we stack it into the FIFO for later use.

Please complete the following state diagram first (on page 4), and then go for the Verilog coding part. Of course, you can also refer to the code to get some related information while working on the state diagram.

State Diagram:

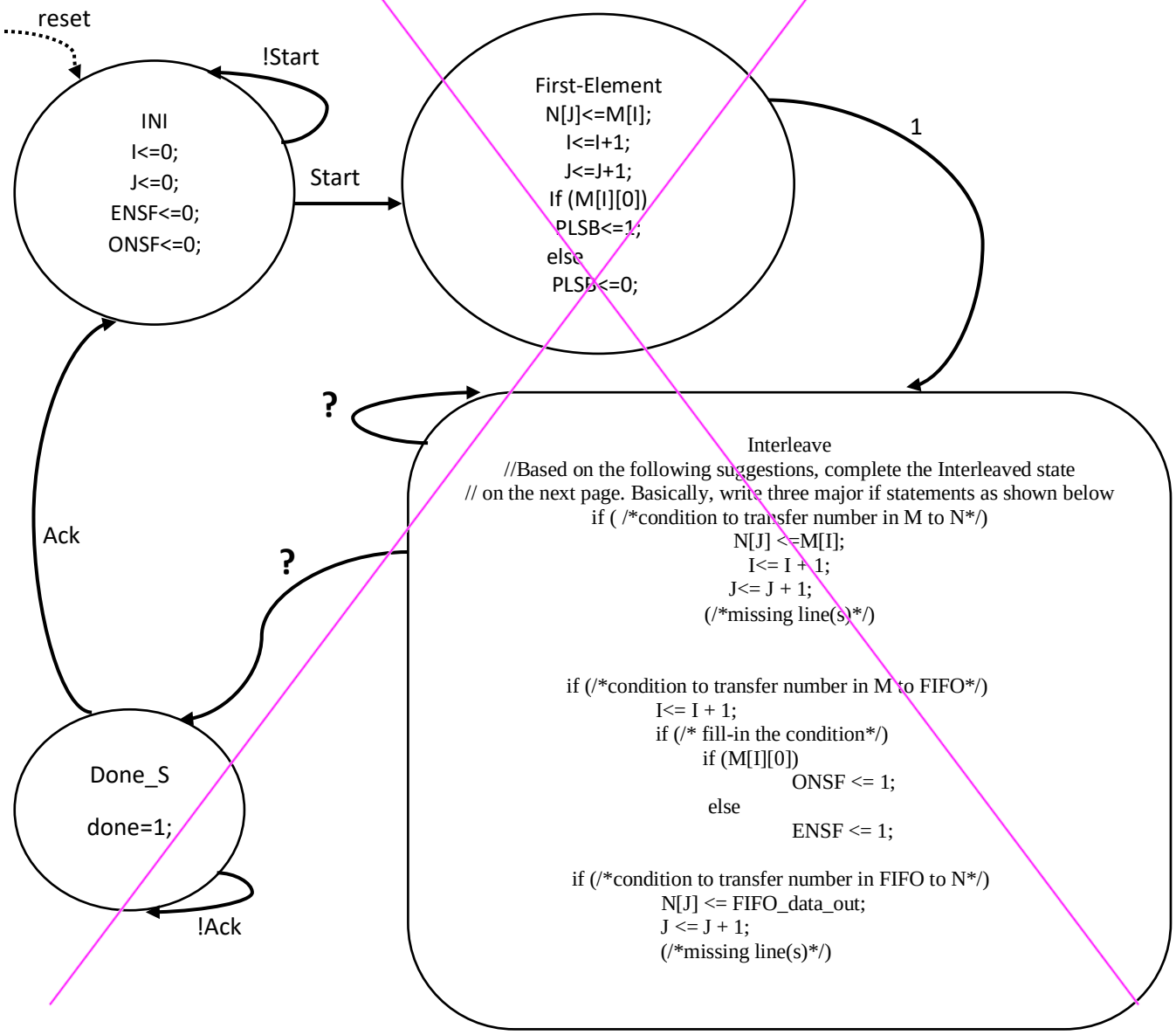
Flags used in the design:

PLSB: previous least significant bit. It is a "1" for odd, and a "0" for even. It is used to record the state of the previous element in array N (note: we said, "in array N" and not "in array M"). So PLSB gets toggled every time we deposit an element in N. Do not forget this.

ENSF (Even Numbers Stacked in the FIFO), ONSF (Odd Numbers Stacked in the FIFO): These flags indicate the evenness/oddness of number(s) stacked in FIFO. Both can be zero together (if the FIFO is empty) but both cannot be 1 simultaneously. Whenever the FIFO becomes empty, do not forget to reset either the flag which was set (or for simplicity reset both the flags). Set a flag to 1 either once when you are depositing the first element in the FIFO (or for simplicity on every deposition of an element in the FIFO).

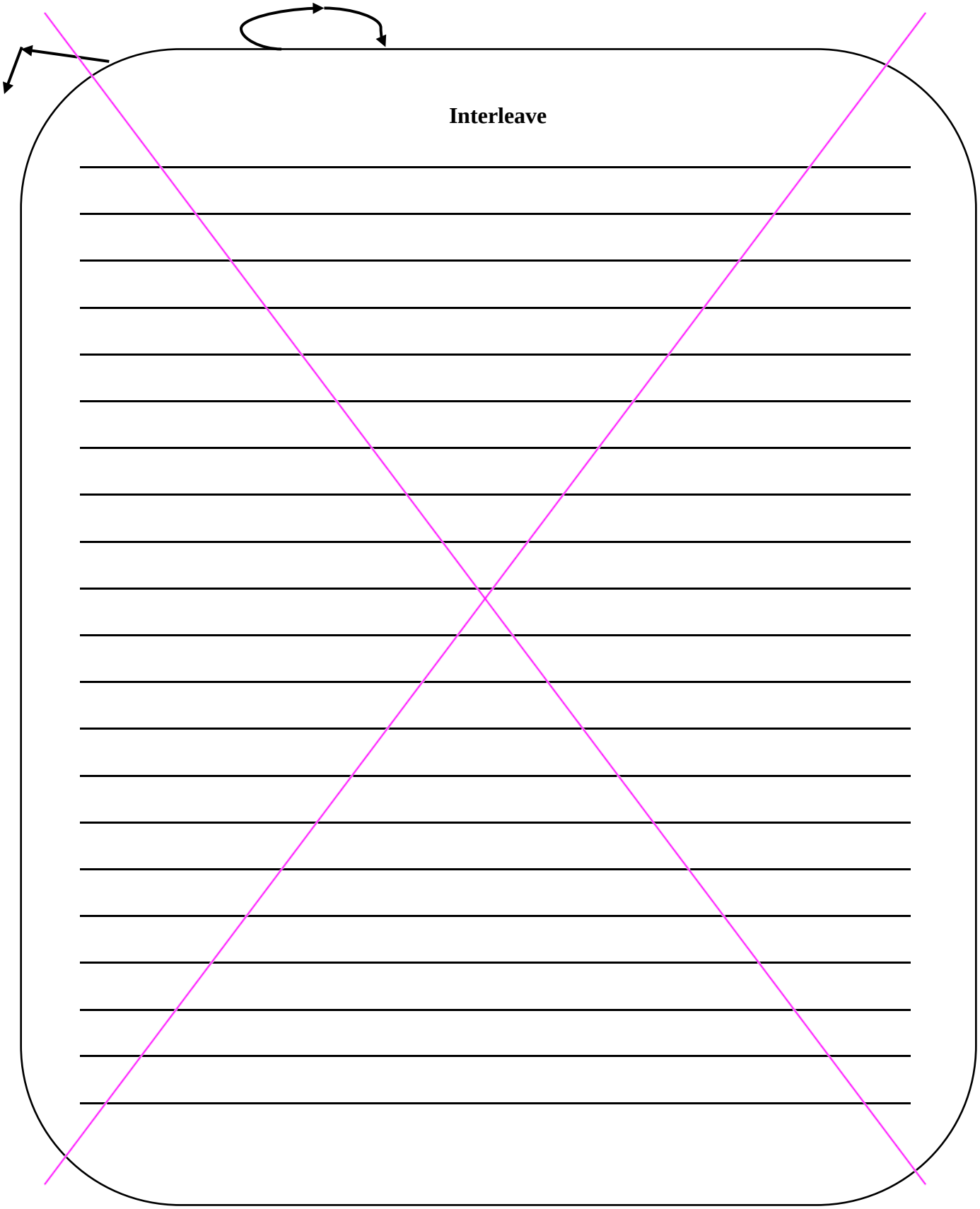
On this page, we have already given the RTL and state transition arrows with conditions for three of the four states. We also gave hints for you to arrive at the RTL for the Interleave state. Complete the RTL and state transition conditions associated with this Interleaved state on the next page and submit just the next page with your name and your partner's name and both email IDs. Here, we use pseudo-RTL. For combinational outputs such as FIFO_wen and FIFO_ren, you can use "=" instead of "<=" as shown below.

FIFO_wen = 1'b1; // combinational output



Name #1 and email ID:

Name #2 and email ID:



Procedure for Part 1:

1. Read and understand the (n+1)-bit pointer FIFO design `fifo_reg_array_sc_n_plus_1_bit_pointers.v`. Notice that the depth is a 5-bit wide item for a 16-location FIFO and covers all 17 values (zero depth corresponding to empty to full depth of 16 corresponding to full status). So, when (depth == 5'b00000) the FIFO is empty and when (depth === 5'b10000) the FIFO is full.

2. Make a directory `EE457_FIFO` under `C:\Modelsim_projects`.
Make two subdirectories `part1` and `part2`.
Into part1 subdirectory, import the files provided to you through the zip file: `EE457_FIFO_part1.zip`
The .zip file contains the following files:
Files specific to the (n+1)-bit FIFO design:

```
fifo_reg_array_sc_n_plus_1_bit_pointers.v
fifo_reg_array_n_plus_one_bit_pointers_sc_tb.vhd
fifo_reg_array_n_plus_one_bit_pointers_sc.do
fifo_reg_array_sc_n_plus_one_bit_wave.do
```

Files specific to the (n)-bit FIFO design:

```
fifo_reg_array_sc_n_bit_pointers_exercise.v
fifo_reg_array_n_bit_pointers_sc_tb.vhd
fifo_reg_array_n_bit_pointers_sc.do
fifo_reg_array_sc_n_plus_one_bit_wave.do
```

Files common to the (n+1)-bit and (n)-bit FIFO designs:

```
producer.vhd
consumer.vhd
ee457_producer_data.txt
ee457_received_data_golden.txt
```

3. Start modelsim and create a new project
Project Name: `EE457_FIFO` Project Location: `C:\Modelsim_projects\EE457_FIFO\part1`
Add to the project the following files:

```
fifo_reg_array_sc_n_plus_1_bit_pointers.v
fifo_reg_array_n_plus_one_bit_pointers_sc_tb.vhd
producer.vhd
consumer.vhd
```

4. Simulate the design using the provided .do file (`do fifo_reg_array_n_plus_one_bit_pointers_sc.do`)
Compare the results `ee457_received_data.txt` with `ee457_received_data_golden.txt`.

5. Complete the (n)-bit pointer design `fifo_reg_array_sc_n_bit_pointers_exercise.v` based on the EE254L FIFO lecture. Add it to the project along with the given testbench: `fifo_reg_array_n_bit_pointers_sc_tb.vhd` . And simulate it (the testbench) using the given .do file (`do fifo_reg_array_n_bit_pointers_sc.do`).
Compare the results `ee457_received_data.txt` with `ee457_received_data_golden.txt`.
Submit the completed Verilog file, received data file and names.txt

```
submit -user ee457lab -tag puvvada_FIFO_part1
fifo_reg_array_sc_n_bit_pointers_exercise.v ee457_received_data.txt names.txt
```

Procedure for Part 2:

1. Read and understand the given incomplete state diagram

1. Complete the state diagram on page 4.

2. Complete the design `even_odd_number_interleaving_exercise.v`.

3. Into the previously created part2 subdirectory `C:\Modelsim_projects\EE457_FIFO\part2` , import the files provided to you through the zip file: `EE457_FIFO_part2.zip`
The .zip file contains the following files:

```
even_odd_number_interleaving_exercise.v
even_odd_number_interleaving_tb.v
even_odd_number_interleaving.do
even_odd_number_interleaving_wave.do
Golden_results.txt
```

To the above directory add a copy of your complete part1 design.
Copy `fifo_reg_array_sc_n_bit_pointers_exercise.v` from part1 directory and paste it to part2 directory.

4. Start modelsim and create a new project
Project Name: `EE457_FIFO_app` (app = application)
Project Location: `C:\Modelsim_projects\EE457_FIFO\part2`
Add to the project the design and the testbench in the above step and simulate using given .do file (`do even_odd_number_interleaving.do`).
Compare results produced by you `output_results.txt` with the golden results `Golden_results.txt` .
If needed, debug.

5. Submit the design and the results file.

```
submit -user ee457lab -tag puvvada_FIFO_part2
even_odd_number_interleaving_exercise.v output_results.txt names.txt
```

6. Submit separately your completed paper design (completed Interleaved state on page 4).

7. Celebrate your success!