Designed By: Gandhi Puvvada

Bubble Sort
(RTL Coding Lab -- Mapping algorithm to hardware)

Objectives:

1. To carefully formulate a state diagram and arrive at state transition conditions and RTL for the bubble sort algorithm.
2. To understand the important differences between the sequential nature of software execution and the concurrent nature of hardware.

Description

The five parts of the lab (2 parts for EE457 and 3 parts for EE560):

1. Part 0 is based on the most inefficient bubble sort algorithm which does all (n-1) passes, each pass performing (n-1) comparisons. A completed design in Verilog is provided to students to form as a base design.
2. Part 1 will progressively reduce the number of comparisons made in a pass by reducing the loop bounds. The number of comparisons made would be (n-1) in the first pass, (n-2) in the 2nd pass, eventually 1 comparison in the last pass. Part 1 will terminate the passes early when a pass does not involve any swaps (SF == 0; Swap Flag remains at zero).
3. Part 1A (for the EE560 class): This part is an improvement over Part 1. It avoids unnecessary writes into the memory, there by saving energy.
4. Part 2 further improves Part 1 (Part 1, not Part 1A) in special cases when the pass just ended, has a single swap and the single swap is due to the swap of _________________ (any pair, the first pair, or the last pair). It uses a Single Swap Flag (SSF) besides the earlier Swap Flag (SF)
5. Part 3 further improves Part 1 (Part 1, not Part 1A) and reduces the number of passes further by making outer loop counter (pass counter) "jump" instead of just "decrement" when possible. This is for the EE560 class.
6. Part 4 improves Part 3 by reversing the direction of passes alternately to speed up the “turtles” (for the EE560 class).

Bubble Sort algorithm:

1. Animation of bubble sort operation: Google the web and you can find a number of animations.
   http://www.cs.oswego.edu/~mohammad/classes/csc241/samples/sort/Sort2-E.html
   http://www.youtube.com/watch?v=P00xJgWzz2c&NR=1&feature=fvwp
2. Though the Bubble Sort algorithm is a well-known simple algorithm, we need to translate it carefully for implementation in hardware. Here we assume that the array has **n** elements in locations 0 to (n-1). In hardware implementation we assume n is either 16 or less and hence we use three 4-bit counters for (a) the EPL (End of Pass Limit) going from 15 to 1, (b) the I counter going from 0 to 14, and (c) the In (In = I next) counter going from 1 to 15.
3. The Part 0 is based on the following most inefficient algorithm. It performs (n-1) passes (the EPL-for-loop) each performing (n-1) comparisons (the I-for-loop)

```
for (EPL = n-1; EPL >= 1; EPL--) // EPL runs from n-1 to 1
 for (I = 0; I <= n-2; I++)
  if (M[I] > M[In]) // In (=I next) is always equal to I+1
   swap (M[I], M[In]); // In (=I next) is always equal to I+1
```

4. The Part 1 changes the high-lighted portion of the above algorithm from **(n-2)** to **(EPL-1)** as shown below. It performs (n-1) passes (the EPL-for-loop) each performing progressively less and less comparisons (the I-for-loop).

```
for (EPL = n-1; EPL >= 1; EPL--) // runs from n-1 to 1
 for (I = 0; I <= EPL-1; I++) // same as for (In = 1; In <= EPL; In++)
  if (M[I] > M[In]) // In (=I next) is always equal to I+1
   swap (M[I], M[In]); // In (=I next) is always equal to I+1
```

   Part 1 implementation does more than the above pseudo code. For example if there are no swaps in a pass, we can skip the remaining passes. A flag called SF (for Swap Flag) is initially reset at the beginning of *every* pass and is set whenever we swap a pair of array elements in the pass. At the end of a pass, if we find that *we did not swap any pair*, we skip the remaining passes and conclude the sorting operation. There are further improvements in special cases in part 2. This is discussed later.

5. You are given three incomplete state diagrams in this document, for Parts 1 and 2 (EE457) and for Part 3 (EE560) .

6. Part 2 is an improvement over Part 1. Part 2 saves one pass in certain cases over and above Part 1. A simple improvement to Part 1 is that if a pass (that just ended) had a single swap and that single swap happens to be the swap of ______________________ (the very firs pair / any pair / the very last pair) (you figure it out), no more passes are needed. To implement this we have an SSF (Single Swap Flag) initiated to zero in the INITIAL (INI) state. Based on your analysis, you may want to view it as FSSF (First Single Swap Flag) or ASSF (Any Single Swap Flag) or LSSF (Last Single Swap Flag).

7. In Part 3 (for the EE560 class), we make an observation that, in the current pass, if the ***latest*** swap occurred was at M[k] and M[k+1], and the rest of the comparisons in the pass (up to and including the last pair) **did not** result in any more swaps, then the "EPL" counter can be made to jump to "k" (EPL <= k) instead of just decrementing (instead of doing EPL <= EPL -1). If you are jumping anyway, perform the same jump even if the jump is by 1 step instead of decrementing, (there by avoiding separate decrementation of EPL). The variable "k" here is renamed as "new_EPL" in state diagram as well as in the HDL code.

8. Part 4 (for the EE560 class) is a variation of Part 3 by making alternate passes proceed in ***reverse*** direction, there by providing a chance to the "turtles" to become "rabbits". See http://en.wikipedia.org/wiki/Bubble_sort#Rabbits_and_turtles . In parts 1 or 2 or 3, in a pass, the heaviest item (the largest number) will surely sink to the bottom (the rabbit quickly finishes its journey), but the lightest item (the turtle) will only bubble up one step. So if we change the direction of the pass, the lightest item becomes the rabbit this time. See the illustration of this bubble sort method (they call it Cocktail Sort) at http://www.algolist.com/Bubble_sort . The improvement may be limited, but here our interest is *more* in the state diagram design challenge posed by this method.

Implementation:

1. **Datapath unit**: Though the elements of datapath unit are inferred by the synthesis tool, it is important for a hardware engineer to visualize the hardware that is inferred.

   1.1. The 16x8 RAM, holding the array to be sorted, is basically an array of registers (like a register file in a CPU). It is a **single-ported RAM** (i.e. it has only one address input) but with separate data-in and separate data-out connections. This restriction of single-port requires you to access the two memory elements M[I] and M[In] in two separate states (meaning two separate clocks). We need a clock to read a memory location and we will not have time to compare what we read with another item previously read in the same clock. Hence comparison is done in a separate state. However, we were told that writing is fast and that we can compare and if needed write to the RAM in the same clock.

   1.2. We have in the data path, two registers, **`First`** and **`Second`**, to hold a pair of data items from the memory, a comparator to compare them, and a multiplexer to select between them for swapping and writing back into the RAM.

   1.3. We have the three counters **`EPL`**, **`I`** and **`In`**. EPL, the **End of Pass Limit counter**, is a decrementing counter, (which can be loaded with a jump value in Part 3 for EE560). **`I`** and **`In`** (**In** for **I next**) are incrementing counters to access the pair of elements, M[I] and M[In]. You may assume that enough comparison units are available for you to compare "In" with "EPL", "EPL" with "0" or "1" as needed.

2. **State Diagram for Part 0**:

Six states:
INI: Initial,
RF: Read First; First <= M[I],
RS: Read Second; Second <= M[In],
CS: Compare and Store minimum at M[I], If Comparison needs swapping, go to EOP.
EOP: End of Pass operation state. It is better to call it **Store_Max State** in Part 0. M[In] <= Max;
DONE: Sorting process is done. The array is now in ***ascending*** order with smallest at M[0]

3. **State Diagram for Part 1**:

3.1. Design plan: Part 1 improves Part 0, by reducing the number of passes using the SF flag and reducing the number of comparisons with in a pass by progressively decrementing the EPL (End of Pass Limit).

Part 1 also improves Part 0 by reducing the number of reads performed on the memory array M.
Part 0 reads 15 pairs of the 16-element array in its first pass (total 30 reads, taking 30 clocks). Part 1 reads every element only once (total 16 reads taking 16 clocks). Consider the 4-element array below, for example.

	M
0	
1	
2	
3	

Part-0 reads	Part-1 reads
M[0], M[1]	M[0], M[1]
M[1], M[2]	M[2]
M[2], M[3]	M[3]

Part 1 also improves Part 0 by reducing the number of writes performed on the memory array M.
If the array is sorted in the wrong order to start with, Part 0 writes 15 pairs of the 16-element array in its first pass (total 30 writes). In this case of reverse-sorted array, in its first pass, Part 1 writes every element only once (total 16 writes). Consider the 4-element array below, for example.

	M
0	8
1	7
2	6
3	5

Part-0 writes	Part-1 writes
7 in M[0] in CS, 8 in M[1] in EOP	7 in M[0] in CS,
6 in M[1] in CS, 8 in M[2] in EOP	6 in M[1] in CS,
5 in M[2] in CS, 8 in M[3] in EOP	5 in M[2] in CS,
	8 in M[3] in EOP

Note: EOP is actually STORE_MAX state in Part 0

After every comparison, does Part 0 end-up writing into 1 or 2 memory locations? What about Part 1? Part-0's plan is to skip writing if there is no need to swap and write *both* elements if they need to be swapped. So, if the array is reverse-sorted at the beginning, Part 0 ends up writing 15 pairs (total 30 writes) in its first pass. On the other hand, if the array is correctly-sorted at the beginning, Part 0 ends up writing 0 pairs (total 0 writes) in its first pass. However, since clock-cycles wise, a write is *free* if it takes place at the end of the comparison in the CS state. So Part 1 writes the **minimum** of the two elements being compared into M[I] **unconditionally** (*irrespective of whether swap is needed or not*) in the CS state and carries the **maximum** of the two in the "`First`" register for the next comparison. In the *final* comparison (the last comparison at the end of the current pass), after writing the minimum in M[I] in the CS state, if that final comparison warrants swapping, then you go into EOP state to store in M[In] the maximum of the two (which is the largest number of the entire array).
Mr. Bruin says: It is wasteful, though free, to write the minimum in M[I] in the CS state if does not warrant swapping. You can save energy, if not clocks, by avoiding unnecessary writes to the M[I]. If swap is not needed, isn't what you are writing into M[I] in CS state, already in M[I]? In the case of an array which is correctly-sorted at the beginning, Part 0 writes 0 pairs (total 0 writes) in its first pass. You (Trojans) write wastefully to 15 locations in Part1. Please help Mr. Bruin understand that there is a need to write the minimum into M(I) even in the clock when you do not swap, by using the example below:

	M
0	8
1	7
2	11
3	9

```
Mr. Bruin's Part-1 First Pass
compares (8,7), writes 7 in M[0] in CS since swap needed
compares (8,11), does note write in M[1] as no swap is needed
compares (11,9), writes 9 in M[2] in CS, 11 in M[3] in EOP
                                  8 is lost and 7 is replicated!
```

M after 1st pass

	M
0	7
1	7
2	9
3	11

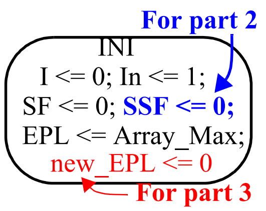
Note for the EE560 students: Part of what Bruin said is true. If the array is initially sorted in the correct order, we should be able to avoid unneeded writes. In Part 1A, try to reduce writes. You are not allowed to use an extra multi-bit comparator, like an 8-bit comparator here. Hint: Use a flag called "swap_in_last_clock"

Six states in Part 1:
INI: Initial,
RF: Read First; First <= M[I],
RS: Read Second; Second <= M[In],
CS: Compare and Swap;
Store minimum at M[I], Make sure First has the needed value for the next comparison, as we may not be going to RF state before the next comparison. Do First <= Second if needed
EOP: End of Pass operation. If an element needs to be stored in the last location of the pass, we do it here.
DONE: Sorting process is done. The array is now in ***ascending*** order with smallest at M[0]

3.2. All state transition arrows are shown. RTL operations associated with the four states INI, RF, RS, and DONE are given. State transition conditions associated with the state transition arrows diverging from these four states are also given.

3.3. You are responsible for completing the RTL operations to be performed in the two states CS and EOP. You are also responsible for the state transition conditions associated with the state transition arrows diverging from these two states.

3.4. **EPL** (End of Pass Limit): What initialization is performed at the beginning (in INI state) vs. what re-initialization is done at the end of the pass (either in CS or in EOP states)? At the beginning of the sort operation, in INI state, besides initializing I, In, SF, we also initialize EPL. But at the end of the pass, you re-initialize I, In, SF, but only adjust EPL (EPL <= EPL -1;).



3.5. **SF flag** (Swap Flag): In the bubble sort, you make several passes through the array. At the end of the first pass through the array (0 to n-1), the largest element is brought to the (n-1)th position (i.e. that is the last position) and EPL is decremented by 1 from Array_Max [i.e. from (n-1) to (n-2)]. The array eventually shrinks to a 2-element array. However, if there was **no swap** performed in an entire pass, then there is **no need** to perform any more passes. An SF flag is cleared at the beginning of every pass and set whenever a swap happens during the pass, so that at the end of a pass, you can decide whether you need more passes or not.
SF is initially cleared in the INI state. You need to clear it again in the CS or the EOP state at the end of the current pass. You need to set it if you are swapping a pair (except for the last pair of the pass, when you are getting ready to clear it). So your decision to conclude the passes shall be based on SF not being a 1 (meaning no *previous* comparisons resulted in swapping) and the *current* comparison is also not requiring a swap. It is important not to make the common mistake of looking at SF *only* as it reflects that there were no *previous* comparisons.

4. **State Diagram for Part 2**:

4.1. Note that the incomplete state diagram for the Part 1 and Part 2 are identical. The only difference is the **SSF** usage. Since we are using one common test bench for the parts 1 and 2, we have cleared the SSF in the INI state for Part 1 also but did not do anything to SSF in the rest of the states.

4.2. Question: In a pass, you perform comparisons of several pairs of data and swap them if necessary. If only one pair of data got swapped, would you need to perform another pass? Perhaps you would say, YES. What if the only pair swapped is the last pair in the pass? Or what if the only pair swapped is the first pair in the pass? Does it make any difference? Carefully consider this and exploit this in your design for part 2 to reduce the total number of passes by 1 (and there by reduce the total number of clocks) in certain cases (for some data in the array to be sorted). We added another flag called **SSF (Single Swap Flag)** and initialized it to zero in the INI state. You may want to consider it as **FSSF** (First Single Swap Flag) or **ASSF** (Any Single Swap Flag) or **LSSF** (Last Single Swap Flag)

5. **State Diagram for Part 3 (for EE560)**:

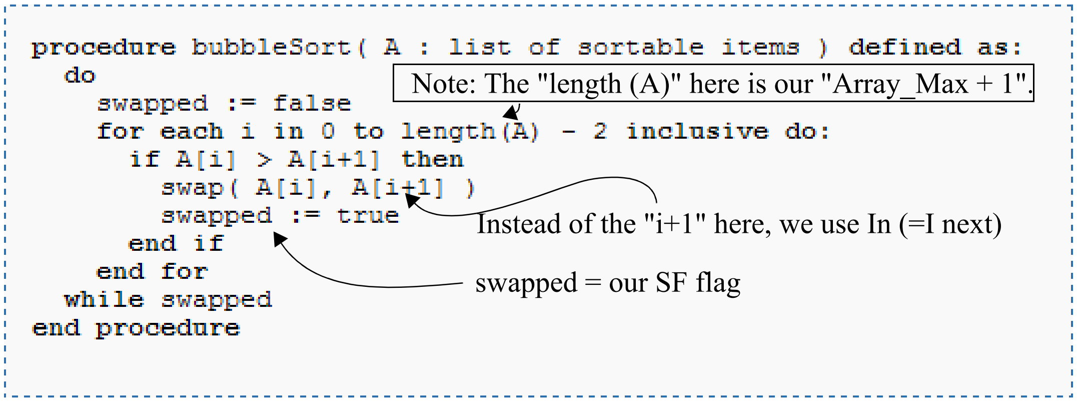
5.1. Part 3 is a variation of Part 1 by more generalizing the improvement achieved through SSF in part 2.
Part 2 basically says that if the only swap made was _______________ (*the very first / any / the very last*) swap, the complete array is in sorted condition. Hence there is no need for further passes.
In part 3, we want to generalize the statement, "the rest of the array is already in sorted state". If substantial portion of the tail-end of the array is already in sorted state, we need to sort on the remaining beginning portion of the array. We make an observation that, in the current pass, if the latest swap occurred is the swap of M[k] and M[k+1], and the rest of the comparisons in the pass (up to and including M[EPL] with M[EPL - 1]) did not result in any more swaps, then the tail portion of the array from M[k+1] to M[EPL] is already in sorted state and contains

the largest (EPL-k) members of the [0:EPL]-sized array. Hence the array portion remaining to be sorted is the first portion containing the first (k+1) members, namely M[0:k].
Note: Originally it had Array_Max+1 elements [0:Array_Max]. If we remove (Array_max-k) elements at the tail end, we are left with (k+1) elements [0:k]. So EPL can be changed directly to k instead of just decrementing to EPL-1.

5.2. Reference for the algorithm in this part is the Wikipedia page on Bubble Sort: http://en.wikipedia.org/wiki/Bubble_sort An extract is reproduced below.

Pseudocode implementation

The algorithm can be expressed as:



```
procedure bubbleSort( A : list of sortable items ) defined as:
  do
    swapped := false
    for each i in 0 to length(A) - 2 inclusive do:
      if A[i] > A[i+1] then
        swap( A[i], A[i+1] )
        swapped := true
      end if
    end for
  while swapped
end procedure
```

Note: The "length (A)" here is our "Array_Max + 1".

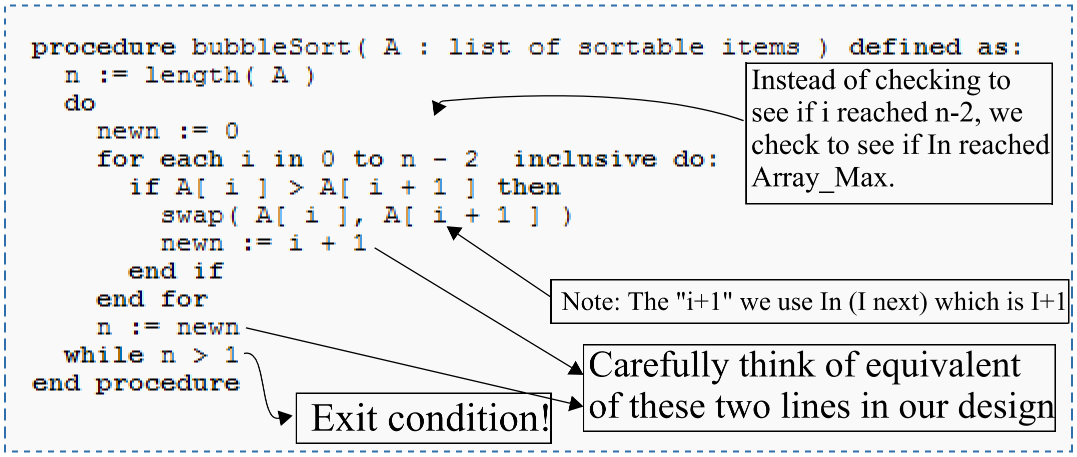
Instead of the "i+1" here, we use In (=I next)

swapped = our SF flag

Optimizing bubble sort

The bubble sort algorithm can be easily optimized by observing that the largest elements are placed in their final position in the first passes. Or, more generally, after every pass, all elements after the last swap are sorted, and do not need to be checked again. This not only allows us to skip over a lot of the elements, but also skip tracking of the "swapped" variable. This results in about a worst case 50% improvement in iteration count, but no improvement in swap counts.

To accomplish this in pseudocode we write the following:



```
procedure bubbleSort( A : list of sortable items ) defined as:
  n := length( A )
  do
    newn := 0
    for each i in 0 to n - 2  inclusive do:
      if A[ i ] > A[ i + 1 ] then
        swap( A[ i ], A[ i + 1 ] )
        newn := i + 1
      end if
    end for
    n := newn
  while n > 1
end procedure
```

Instead of checking to see if i reached n-2, we check to see if In reached Array_Max.

Note: The "i+1" we use In (I next) which is I+1

Carefully think of equivalent of these two lines in our design

Exit condition!

5.3. Again the incomplete state diagram for Part 3 looks same as the incomplete state diagrams of the Part 1 and Part 2. We did not provide a partial state diagram for Part 4 as EE560 students can arrive at it by themselves :)

6. **Verilog testbench** to test your design:

6.1. The testbench initializes the RAM with a given sequence of numbers and then activates START signal.

6.2. Your DUT then moves from the initial (I) state and performs the sorting operation.

6.3. At the end of sorting, your DUT goes into DONE state and goes back to the initial state.

6.4. The testbench waits for the Qdone signal and then reads back the contents of the memory and reports the same by writing into the text files, for example `ee457_bsort_results_P1.txt` and `additional_ee457_bsort_results_P1.txt` (for Part 1).

6.5. Sample output of `ee457_bsort_results_P1.txt`

```
 Test number   1:
 Initial memory contents:
 f4  85  07  99  02  01  03  00
 Final memory contents:
 00  01  02  03  07  85  99  f4
    Design entered DONE state from  EOP   .
    Total number of passes =            7 .
    Total number of comparisons =          28 .
    Total number of memory writes =          35 .
          Clocks taken for this test = 70.
```

6.6. The testbench provided (`ee457_bsort_improved_tb.v`) produces the above results file (`ee457_bsort_results_P1.txt`) and also an additional text file recording the details of passes. See example of an extract from the `additional_ee457_bsort_results_P1.txt`

```
After the array elements, number of comparisons, number of memory writes and number of clocks
taken are listed.

Test number  1:
8-element unsorted array
Initial contents:   f4  85  07  99  02  01  03  00            0   0   0
At end of Pass#1:   85  07  99  02  01  03  00  f4            7   8  16
At end of Pass#2:   07  85  02  01  03  00  99  ||            6   7  14
At end of Pass#3:   07  02  01  03  00  85  ||  ||            5   6  12
At end of Pass#4:   02  01  03  00  07  ||  ||  ||            4   5  10
At end of Pass#5:   01  02  00  03  ||  ||  ||  ||            3   4   8
At end of Pass#6:   01  00  02  ||  ||  ||  ||  ||            2   3   6
After LastPass#7:   00  01  ||  ||  ||  ||  ||  ||            1   2   4
Final sorted array: 00  01  02  03  07  85  99  f4           28  35  70
Sorting for Test# 1 completed.
```

6.7. You would soon notice that, for any reasonably complex design, manual analysis of lengthy waveform is laborious and time-consuming and that a well-written HDL testbench is necessary to aid design and functional verification. EE students planning to become hardware engineers should go through the testbenches and learn to write testbenches.

Procedure for Part 1:

1. Import the .zip file `ee457_bsort.zip` into your `C:\ModelSim_projects` directory.

   Unzip to form `C:\ModelSim_projects\ee457_bsort` directory containing the following files:

   Part 0 and Part 1 files
   `ee457_bsort_P0.v (complete)` `ee457_bsort_P1.v` (incomplete core design to be completed by you)
   `ee457_bsort_improved_tb.v` (complete testbench, good for all four parts, 1, 1A, 2, and 3)
   `additional_ee457_bsort_results_P1_SOLUTION.txt`, `additional_ee457_bsort_results_P2_SOLUTION.txt`
   `ee457_bsort.do` (a .do file to invoke simulator, setup waveforms, ...). It does not compile the Verilog files though.
   `ee457_bsort_wave.do` (a wave.do file called by the main .do file)

2. Read all files provided. Two test results filse (SOLUTIONs) are given for you to compare with what you produce.
3. Create a modelsim project called ee457_bsort with the project directory C:\Modesim_projects\e457_bsort. Add `ee457_bsort_P0.v` and the common testbench `ee457_bsort_improved_tb.v` to the project. Compile the two verilog files. Simulate Part 0, using the common .do file (do `ee457_bsort.do`). Look at the waveform. Zoom in a couple of passes and make sure you agree with the state sequence, etc. Look at the `ee457_bsort_results_P0.txt` and also `additional_ee457_bsort_results_P0.txt` created by `ee457_bsort_improved_tb.v` in your project directory. If you do not simulate until the end of the simulation to execute the $fclose to close the text files being generated, the text files may remain unpopulated (may contain zero bytes). The text files produced for part 0 are useful to you to later compare with the text files produced for other parts. The P1 and P2 results files would differ with the P0 results files only in the number of passes, the number of comparisons, number of memory writes, and the number of clocks taken but should not differ in the final contents of the sorted array! You can display the files to be compared side-by-side in the Notepad++ window and use Plugins => Compare => Compare to compare the text files. After finishing comparison, do Plugins => Compare => Clear Results to revert to normal display. If you do not find the above menu commands on your Notepad++ menu bar, you need to download the Plugins from the Notepad++ site:
http://sourceforge.net/projects/npp-plugins/
4. Complete the incomplete state diagram for Part 1. It is a good idea to write boolean equations for conditions such as end_of_pass, last_pass, swap (= swap needed for the current pair) and use these in arriving at the state transition conditions and also in specifying conditional mealy RTL operations with-in the state circles. You are welcome to print the diagram on 11"x17".
5. Complete the incomplete `ee457_bsort_P1.v`. Add this file to the modelsim project created earlier and manually compile it. Note: The .do file and the testbench file are common to all four parts (1, 1A, 2, and 3). Hence the .do file does not perform the "compile" step. Manually compile any Verilog files. You can compile a testbench and a core design (Part 1 or or 1A or 2 or 3) and then run the do file (do `ee457_bsort.do)` to set up waveform and run the design for 75us. Debug as needed. Do the same for Part 2.
Running the simulation for 75us is actually needed in Part 0 as it is very inefficient. The Part 1 and Part 2 take far less time. However, running for 75us uniformly is harmless. So the .do file has 75us. If your design has a flaw and is not reaching the DONE state, you may not see the message, "All tests concluded. Inspect the text file ee457_bsort_results_P0.txt " in the trascript window even after 75us simulation.

What to turn-in for Part 1:

1. Submit online the following files (one submission per team).
`ee457_bsort_P1.v ee457_bsort_results_P1.txt names.txt`
2. Submit hardcopy of the completed state diagram for Part 1 (one submission per team). There are no end-of-lab-questions to answer.

Procedure for Part 1A (EE560):

1. Make a copy of your completed state diagram of Part 1 and revise it to make it your Part 1A.
2. Make a copy of `ee457_bsort_P1.v` and name it as `ee457_bsort_P1A.v`. Change **26** to **27** and **P1** to **P1A** (note P1A has one extra character) in the following line:
```
reg [26*8:1] file_results_string = "ee457_bsort_results_P1.txt"; //
Revised line:
reg [27*8:1] file_results_string = "ee457_bsort_results_P1A.txt"; //
```
Revise the design as needed and simulate with the given testbench (and using the given .do file). Debug as needed.

What to turn-in for Part 1A:

1. Submit online the following files (one submission per team).
`ee457_bsort_P1A.v ee457_bsort_results_P1A.txt names.txt`
2. Submit hardcopy of the completed state diagram for Part 1A (one submission per team). No answers to questions needed.

Procedure for Part 2:

1. Complete the incomplete state diagram for Part 2.
2. Make a copy of `ee457_bsort_P1.v` and name it as `ee457_bsort_P2.v`. Change **P1** to **P2** in the following line:
   `reg [26*8:1] file_results_string = "ee457_bsort_results_P1.txt"; //`
   Revise the design as needed and simulate with the given testbench (and using the given .do file). Debug as needed.

What to turn-in for Part 2:

1. Submit online the following files (one submission per team).
   `ee457_bsort_P2.v` `ee457_bsort_results_P2.txt` `names.txt`
2. Submit hardcopy of the completed state diagram for Part 2 (one submission per team). No answers to questions needed.

Procedure for Part 3: (for EE560)

1. Complete the incomplete state diagram for Part 3.
2. Make a copy of `ee457_bsort_P1.v` and name it as `ee457_bsort_P3.v`. Change P1 to P3 in the following line:
   `reg [26*8:1] file_results_string = "ee457_bsort_results_P1.txt"; //`
   Revise the design as needed and simulate with the given testbench (and using the given .do file). Debug as needed.

What to turn-in for Part 3: (for EE560)

1. Submit online the following files (one submission per team).
   `ee457_bsort_P3.v ee457_bsort_results_P3.txt names.txt`
2. Submit hardcopy of the completed state diagram for Part 3 (one submission per team). No answers to questions needed.

Procedure for Part 4: (for EE560)

1. Make a suitable state diagram for Part 4 to help you with coding. The state diagram need not have complete RTL if it is difficult to get it in the limited space. I would take a 11"x17" sheet. It is not needed for submission.
2. It may be a good idea to redesign the Part 3 where passes proceed from the other end (in the reverse direction first before alternating directions.
   2.1 (optional but highly recommended) Make a copy of `ee457_bsort_P3.v` and name it as `ee457_bsort_P3_reverse_direction.v`. Change the following line:
   ```
   reg [26*8:1] file_results_string = "ee457_bsort_results_P3.txt"; //
   Revised line:
   reg [44*8:1] file_results_string = "ee457_bsort_results_P3_reverse_direction.txt"; //
   ```
   2.2 Make a copy of `ee457_bsort_P3.v` and name it as `ee457_bsort_P4.v`. Change P3 to P4 in the following line:
   `reg [26*8:1] file_results_string = "ee457_bsort_results_P4.txt"; //`
   Revise the design as needed and simulate with the given testbench (and using the given .do file). Debug as needed.

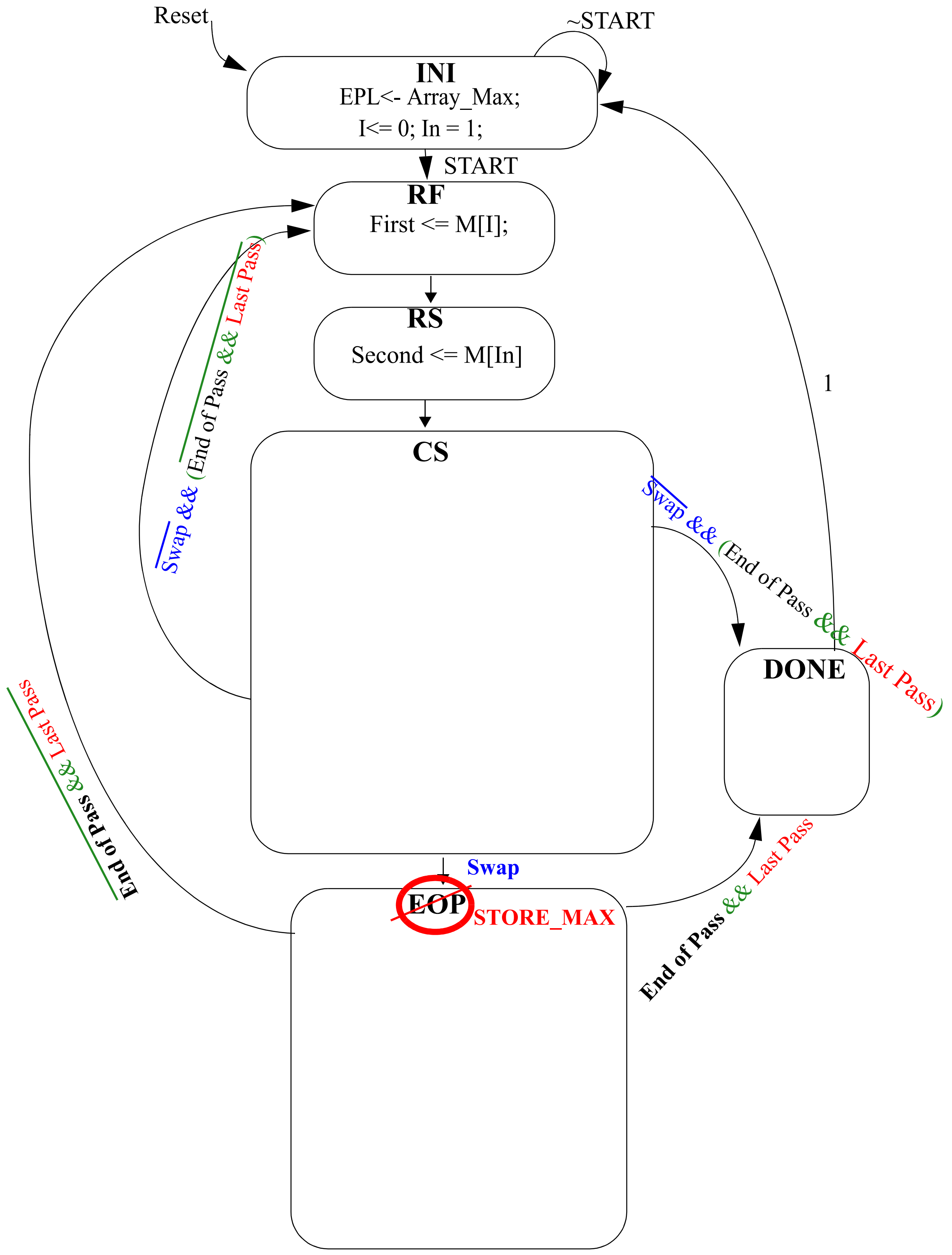
What to turn-in for Part 4: (for EE560)

1. Submit online the following files (one submission per team).
   `ee457_bsort_P4.v ee457_bsort_results_P4.txt names.txt`
2. No state diagram needs to be submitted for Part 4. No answers to questions needed.

<>**Feedback**: Please provide feedback on usefulness and propriety of this lab for our ee457class. Also please tell me if the write-up can be improved. Thanks! Gandhi

Incomplete State Diagram for Bubble Sort (Part 0)

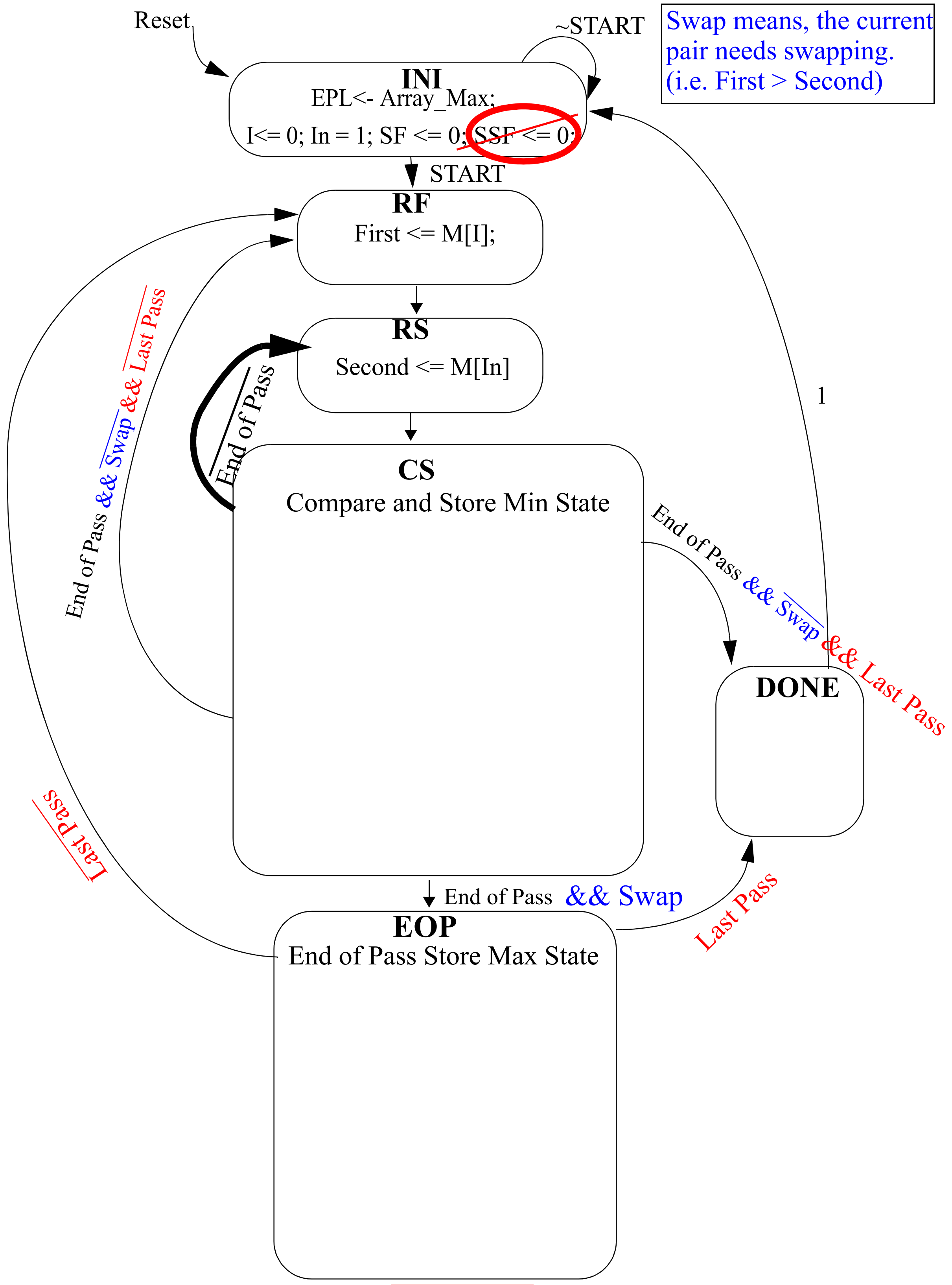
Inefficiencies: Always perform all n-1 passes each n-1 comparisons. Always read the next pair of elements afresh. Unlike in Part 1 or 2, there is no transition from CS to RS. You may like to complete this diagram. No need to submit this.



Reset
~START
INI
EPL<- Array_Max;
I<= 0; In = 1;
START
RF
First <= M[I];
RS
Second <= M[In]
CS
1
$\overline{\text{Swap}}$ && (End of Pass && Last Pass)
$\overline{\text{Swap}}$ && (End of Pass && $\overline{\text{Last Pass}}$)
DONE
End of Pass && $\overline{\text{Last Pass}}$
Swap
~~EOP~~
STORE_MAX
End of Pass && Last Pass

Incomplete State Diagram for Bubble Sort (in preparation to do Part 1)

State Diagram with descriptive state transition conditions not considering both SF and SSF.
Please complete the next page taking into consideration SF also.

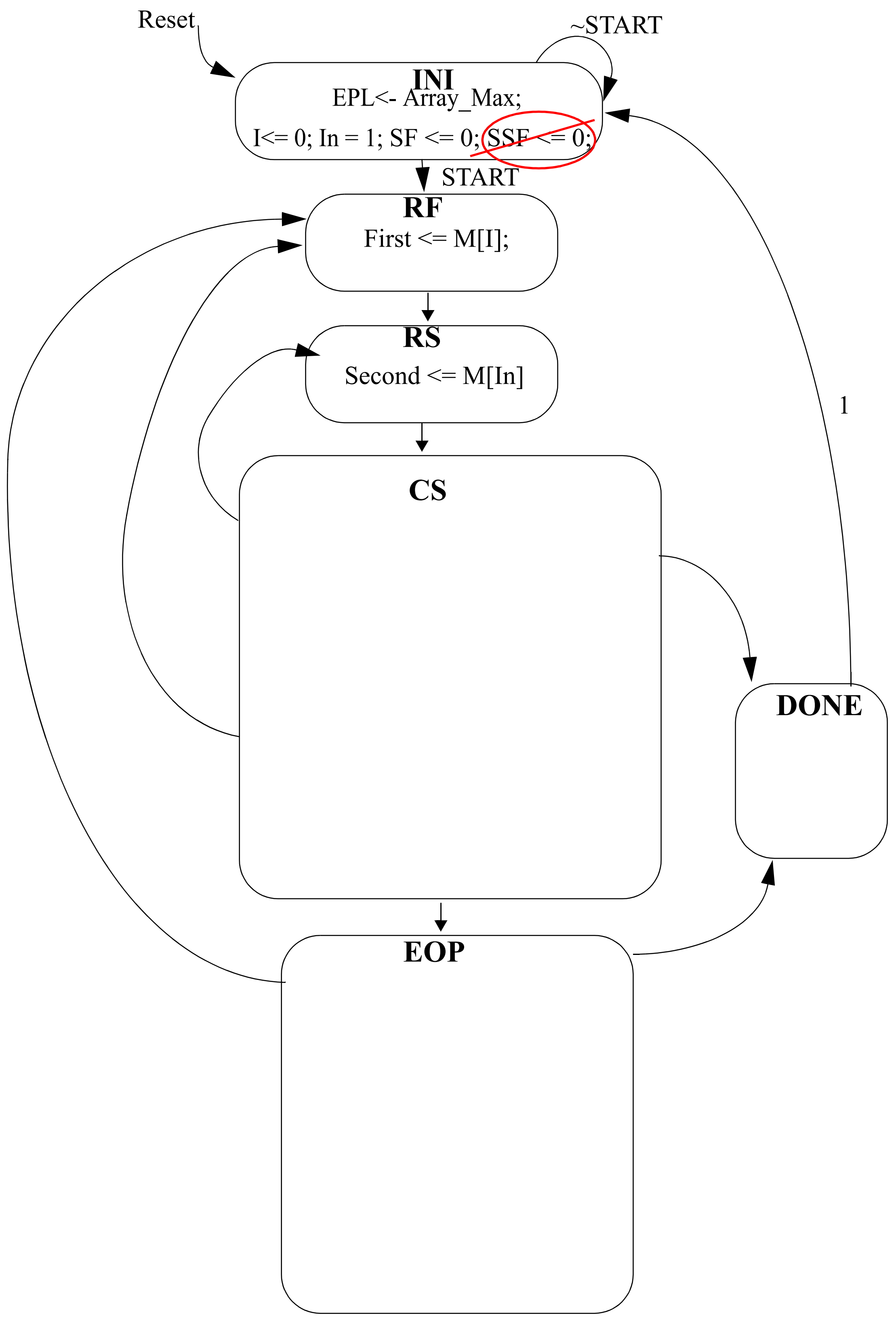


Reset
~START
Swap means, the current pair needs swapping. (i.e. First > Second)
INI
EPL<- Array_Max;
I<= 0; In = 1; SF <= 0; ~~SSF <= 0;~~
START
RF
First <= M[I];
RS
Second <= M[In]
1
End of Pass && Swap && Last Pass
End of Pass
CS
Compare and Store Min State
End of Pass && Swap && Last Pass
DONE
Last Pass
End of Pass && Swap
EOP
End of Pass Store Max State
Last Pass

Incomplete State Diagram for Bubble Sort (Part 1)

EXERCISE

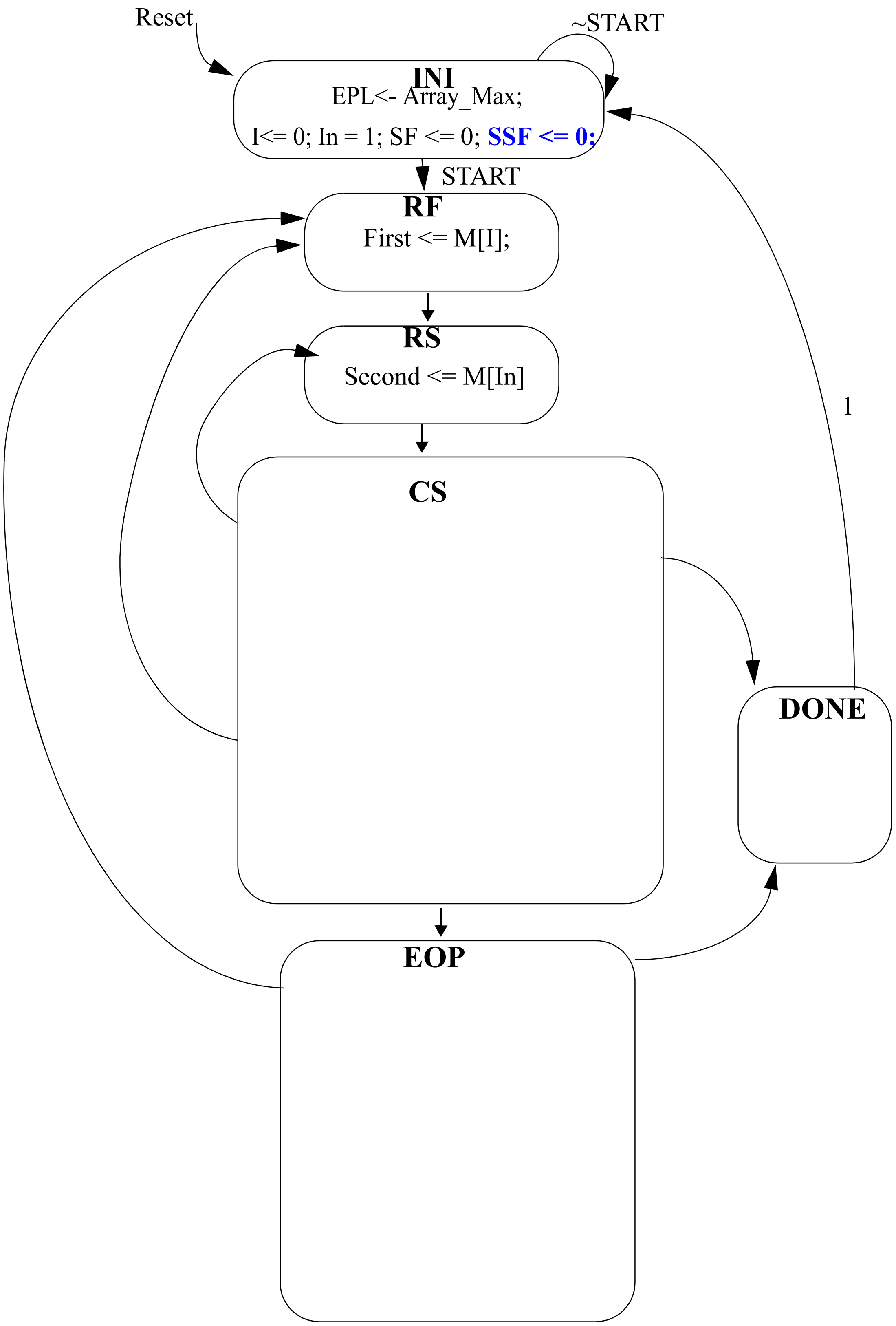
Take SF also into consideration (but not SSF), complete and submit (one per team)



Incomplete State Diagram for Bubble Sort (Part 2)

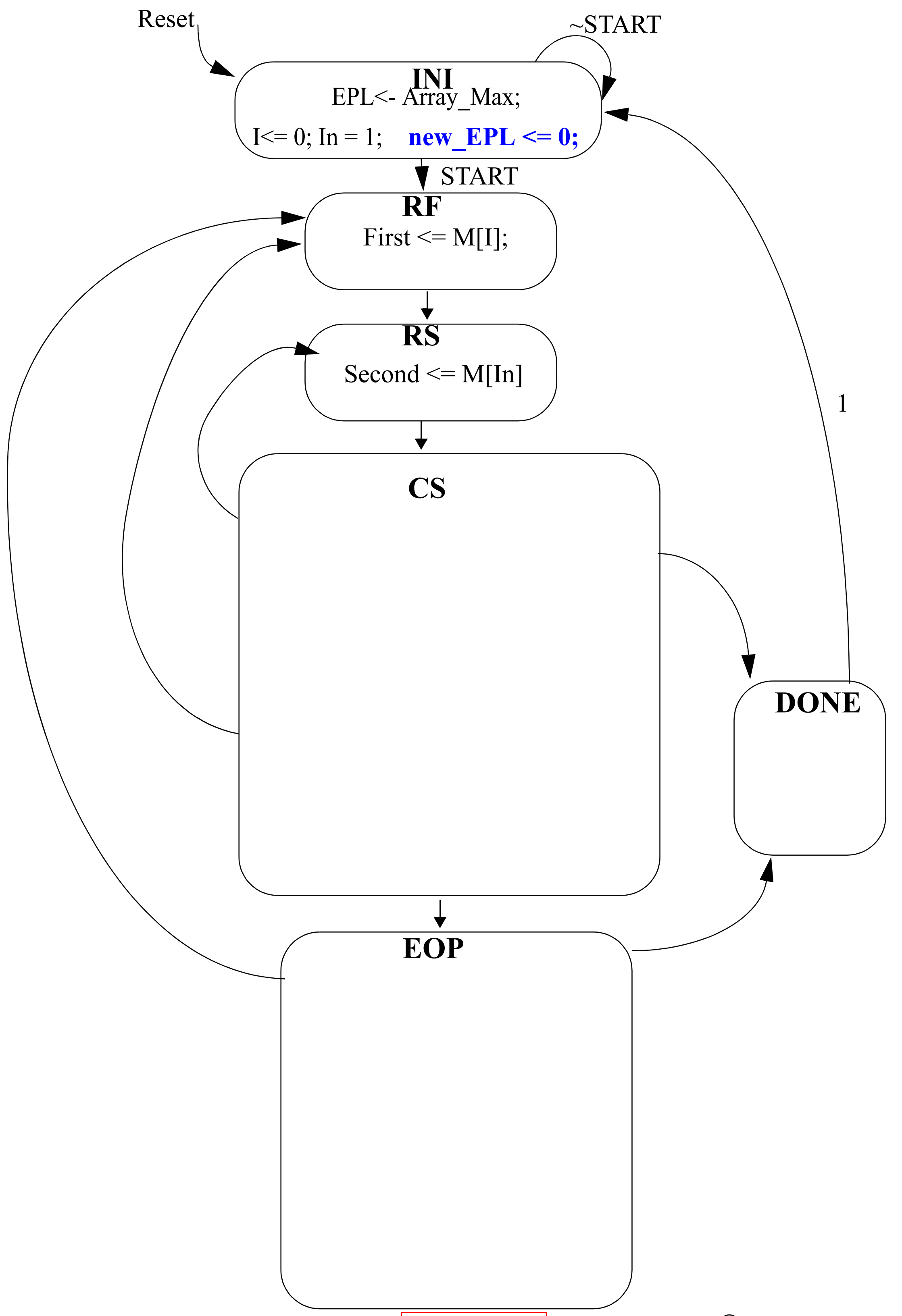
EXERCISE Now take into consideration SSF also to improve the design.

Complete and Submit (one per team)



Incomplete State Diagram for Bubble Sort (Part 3) (EE560)

EXERCISE Here you want to accelerate EPL movement by making it to "jump"!.
Complete and Submit (one per team)



Reset
~START
INI
EPL<- Array_Max;
I<= 0; In = 1;
new_EPL <= 0;
START
RF
First <= M[I];
RS
Second <= M[In]
1
CS
DONE
EOP