Min Max Finder RTL Design

Objective:

- To learn RTL design through a simple exercise
- Understand the difference between an informal flow chart (for implementing the given task in assembly language) and the formal state diagram for implementing in hardware (at RTL level)
- Understand the basic concept of reducing clocks/time to complete a computation by performing as many operations as possible simultaneously.
- Understand the trade-off between Data-Path-Unit hardware cost and performance (time/speed)
- Understand the difference between Moore and Mealy state machines
- Understand when to increment an iteration counter effectively and what terminal count to use
- Understand the beneficial use of registers with a data-enable control and counters with an count_enable control
- Understand simple Verilog coding for RTL design and be able to visualize the hardware inferred
- Understand how to interpret and analyze waveforms

Description:

Design suitable hardware to search and find the smallest (minimum) number and the largest (maximum) number from a set of 16 8-bit unsigned numbers stored in a 16x8 memory array.

This lab has three parts. Part 1 uses two comparators whereas Parts 2 and 3 use a single comparator in their Data Path Unit (DPU). The design of the Control Unit (CU) turns out to be simple in the case of Part 1 compared to other parts.

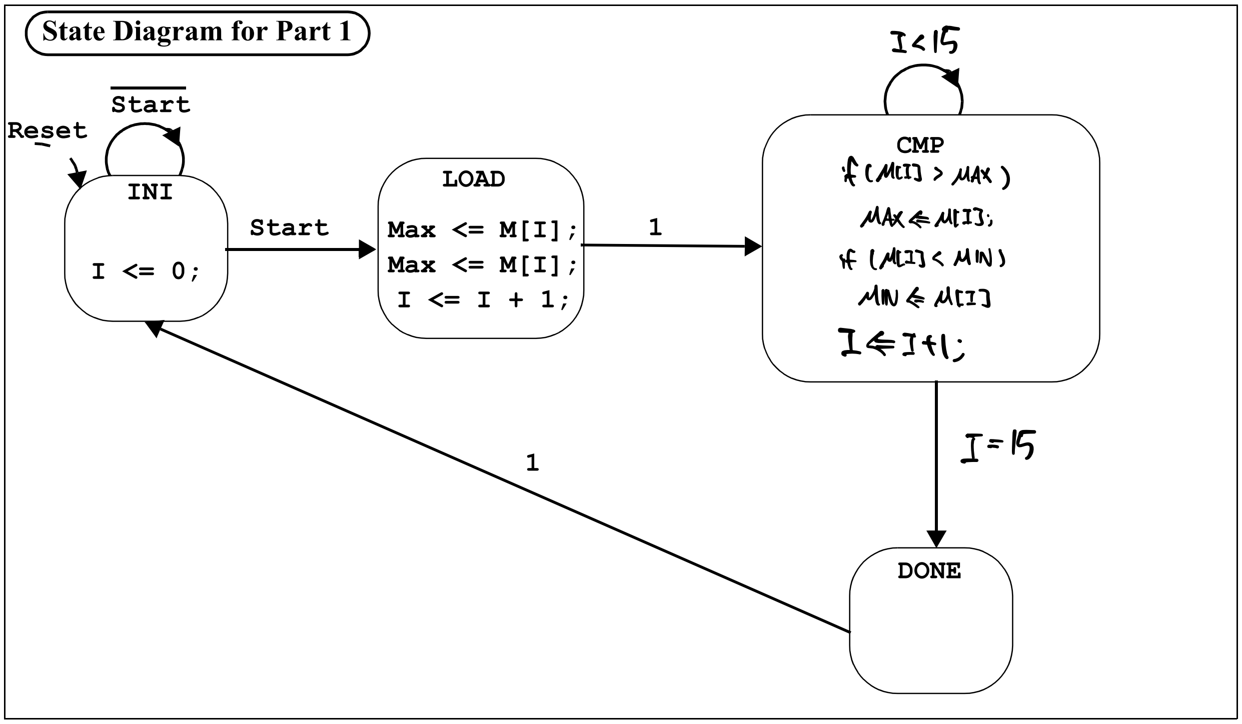
The data unit consists of two registers, a **`Max`** register holding the running maximum number and a **`Min`** register holding the running minimum number. A 4-bit counter (**`I`**) is used as an index into the 16x8 memory array (**`M`**). As stated earlier, part 1 uses two comparators to compare the **`M[I]`** with the running **`Max`** and running **`Min`**, whereas Parts 2 and 3 use a single comparator. So in Parts 2 and 3, the single comparator is used to compare **`M[I]`** with one of the two, **`Max`** or **`Min`**, first and then compare with the other **if needed**. So Parts 2 and 3 take more clocks compared to Part 1 and the number of clocks taken in Parts 2 and 3 **are data dependent**.

On reset, the control unit (CU) goes into an initial (**`INI`**) state. It clears the iteration counter **`I`** while in this state and awaits the **`Start`** control signal. When **`Start`** command is available, the CU goes into **`LOAD`** state and loads **`M[0]`** (the very first data item in **`M`**) into both the **`Max`** and the **`Min`** registers and the iteration counter is incremented to 1 (**`I <= I + 1`**). In subsequent state(s), the remaining **`M[I]`** are processed, updating **`Max`** or **`Min`** as necessary, so that, at the end, when you come into the **`DONE`** state, they (**`Max`** and **`Min)`** contain the largest and smallest numbers respectively. You move from the **`DONE`** state to the initial state **`INI`** on the very next clock without any acknowledgement.

State Diagram Design

All the states and state transition arrows are in place in the following incomplete state diagrams. Complete each of the state diagrams below, first by hand. It is common to make mistakes both in the register transfer statements (listed inside the state circles) and also in the state transition conditions associated with the state transition arrows. Later, when you complete the verilog code and simulate, you will correct your mistakes and will learn through this process *how to think in hardware*.

Part 1 (two comparators)



Part of the incomplete Verilog code given to you for part 1(**`min_max_finder_part1.v`**):

```verilog
module min_max_finder_part1
(Max, Min, Start, Clk, Reset,
Qi, Ql, Qc, Qd);

input Start, Clk, Reset;
output [7:0] Max, Min;
output Qi, Ql, Qc, Qd;

reg [7:0] M [0:15];
//reg [7:0] X;
reg [3:0] state;
reg [7:0] Max;
reg [7:0] Min;
reg [3:0] I;

localparam
INI  =   4'b0001,
LOAD =   4'b0010,
COMP =   4'b0100,
DONE =   4'b1000;

assign {Qd, Qc, Ql, Qi} = state;
```

To facilitate use of symbolic names for states and also user defined state encoding

```verilog
always @(posedge Clk, posedge Reset)

  begin  : CU_n_DU
    if (Reset)
      begin
        state <= INI;
        I  <= 4'bXXXX;
        Max  <= 8'bXXXXXXXX;
        Min  <= 8'bXXXXXXXX;
      end
    else
      begin
        case (state)
          INI :
            begin
              // state transitions
              if (Start)
                state <= LOAD;
              // RTL operations
              I <= 0;
            end
          LOAD      :    // to be completed
            begin
```

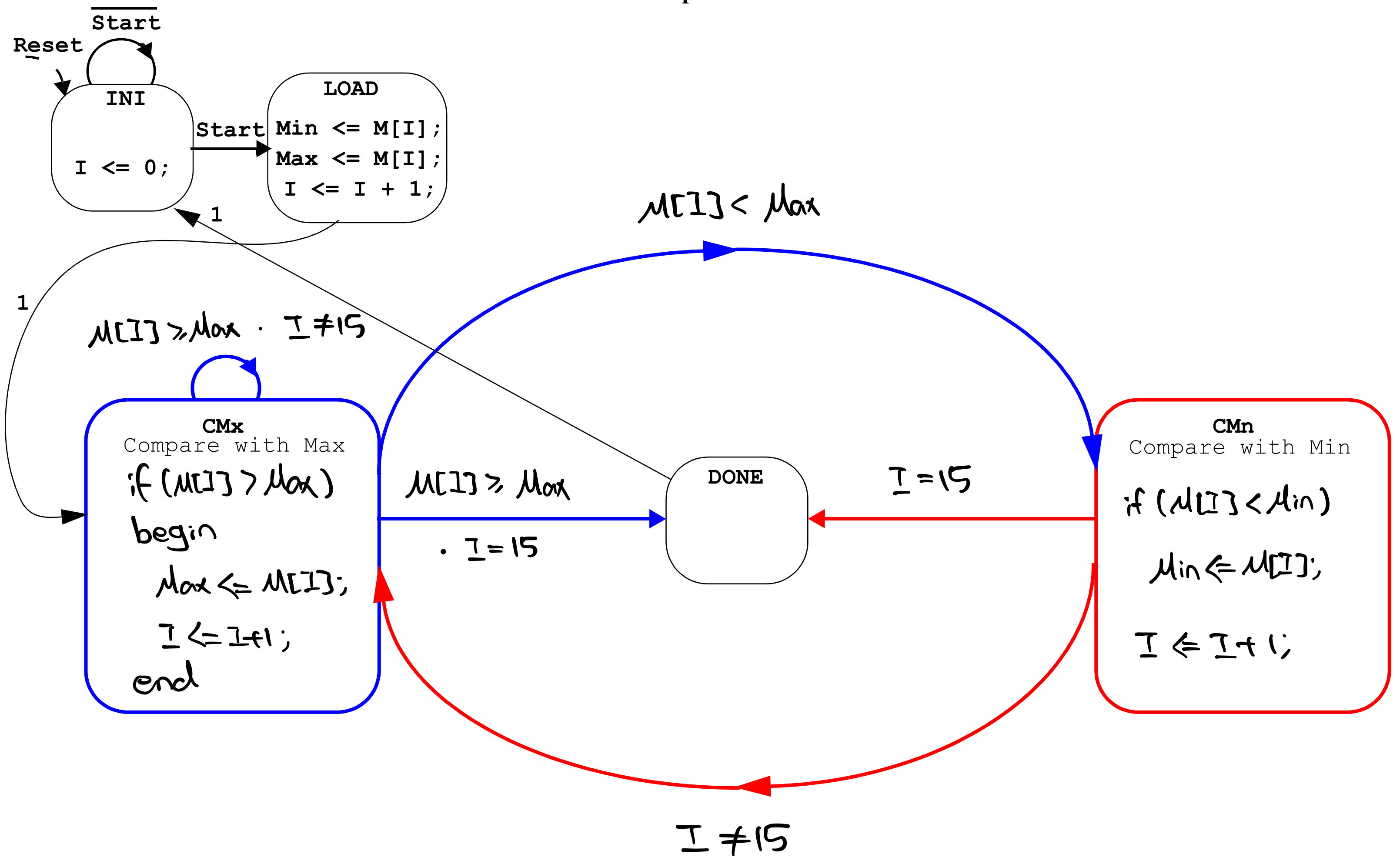
To avoid unnecessary recirculating muxes controlled by Reset

"case" statement to describe both CU and DPU

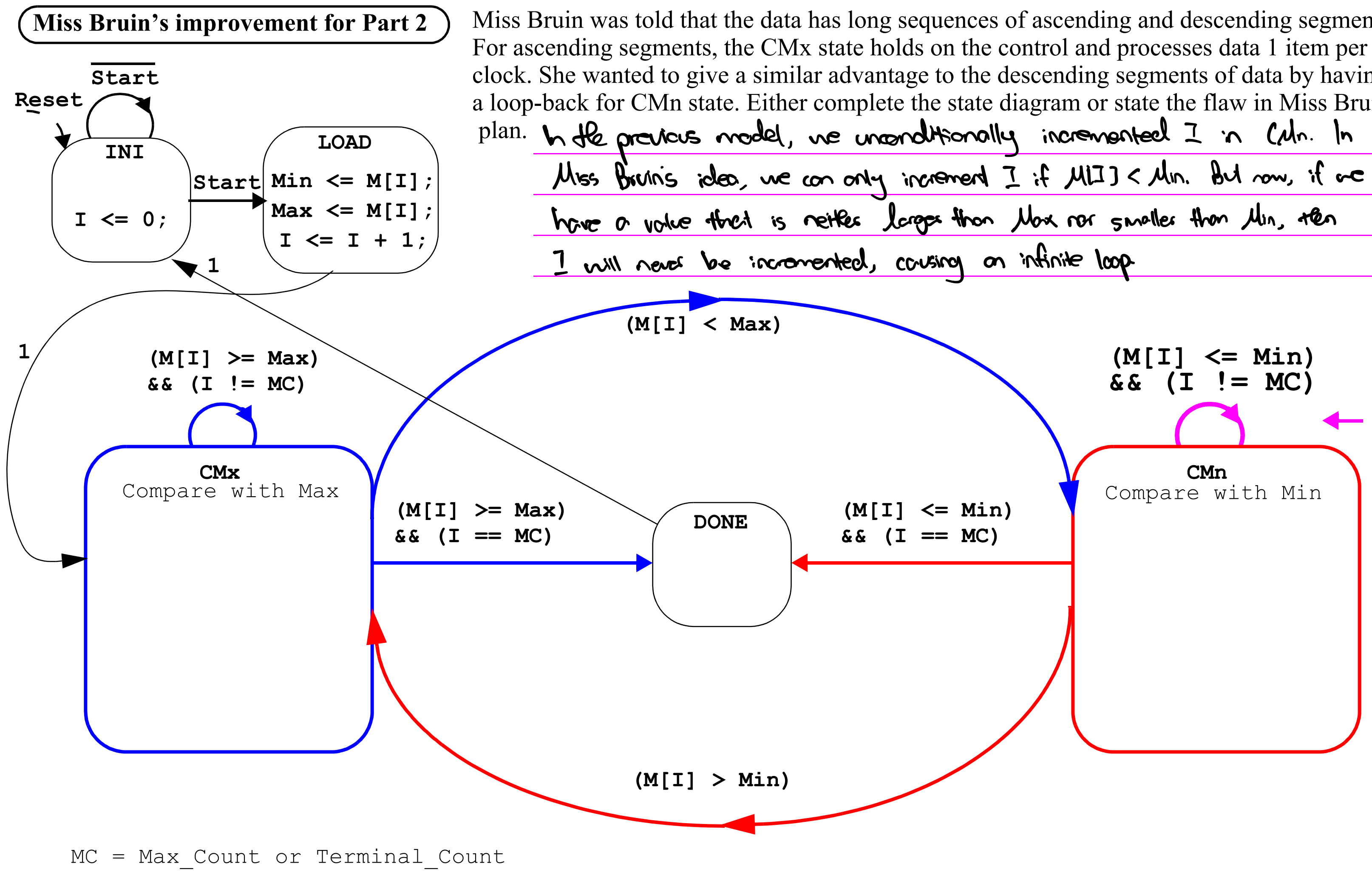
Part 2 (one comparator)

Here a new M[I] is first compared with Max in CMx and then only if, needed, it is compared with Min in CMn state.

State Diagram for Part 2 for 1 comparator



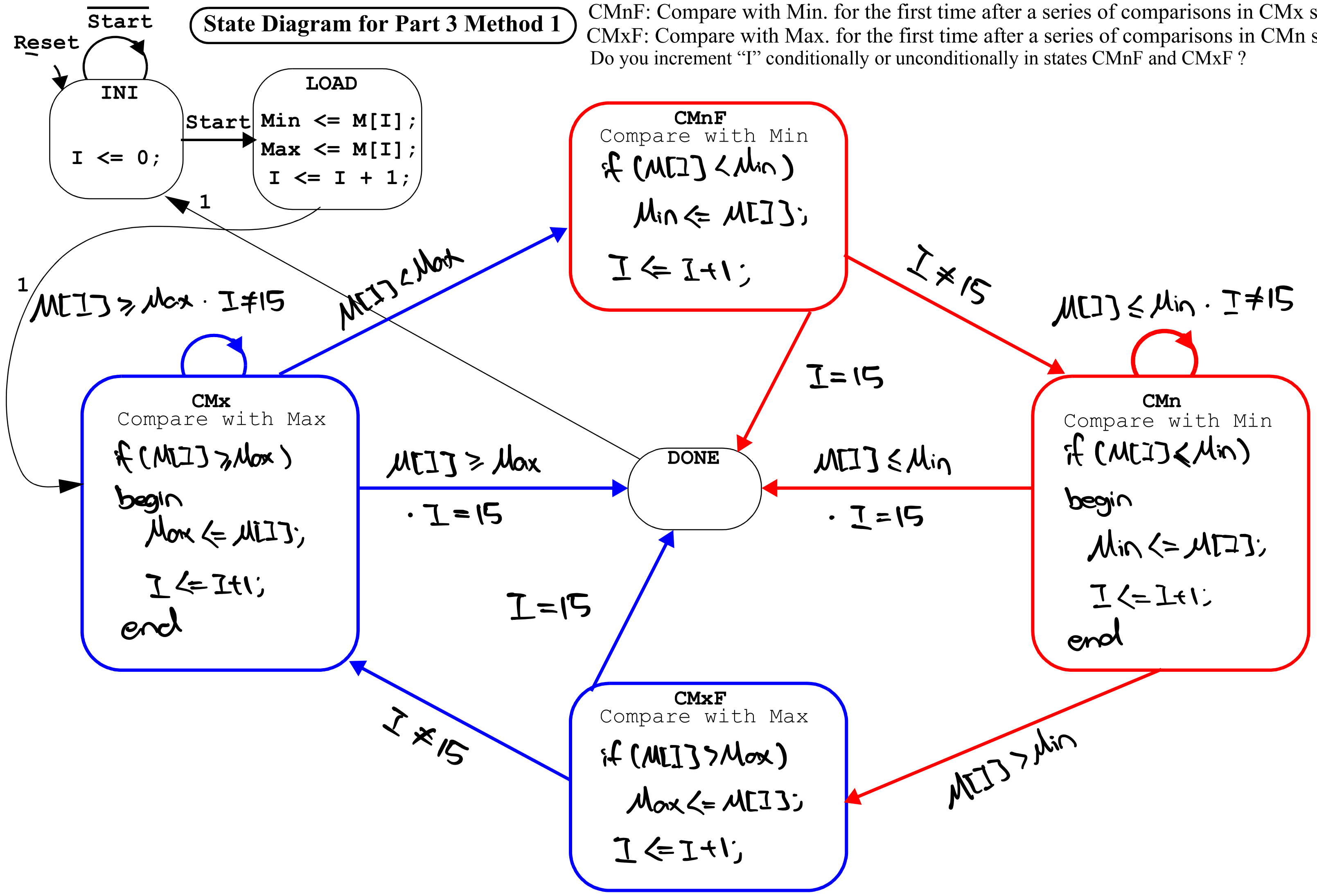
Miss Bruin was told that the data has long sequences of ascending and descending segments. For ascending segments, the CMx state holds on the control and processes data 1 item per clock. She wanted to give a similar advantage to the descending segments of data by having a loop-back for CMn state. Either complete the state diagram or state the flaw in Miss Bruin’s plan. In the previous model, we unconditionally incremented I in CMn. In Miss Bruin's idea, we can only increment I if M[I] < Min. But now, if we have a value that is neither larger than Max nor smaller than Min, then I will never be incremented, causing an infinite loop


Miss Bruin’s improvement for Part 2
Reset
Start
INI
I <= 0;
Start
LOAD
Min <= M[I];
Max <= M[I];
I <= I + 1;
1
1
(M[I] >= Max) && (I != MC)
CMx
Compare with Max
(M[I] < Max)
(M[I] >= Max) && (I == MC)
DONE
(M[I] <= Min) && (I == MC)
(M[I] <= Min) && (I != MC)
CMn
Compare with Min
(M[I] > Min)
MC = Max_Count or Terminal_Count

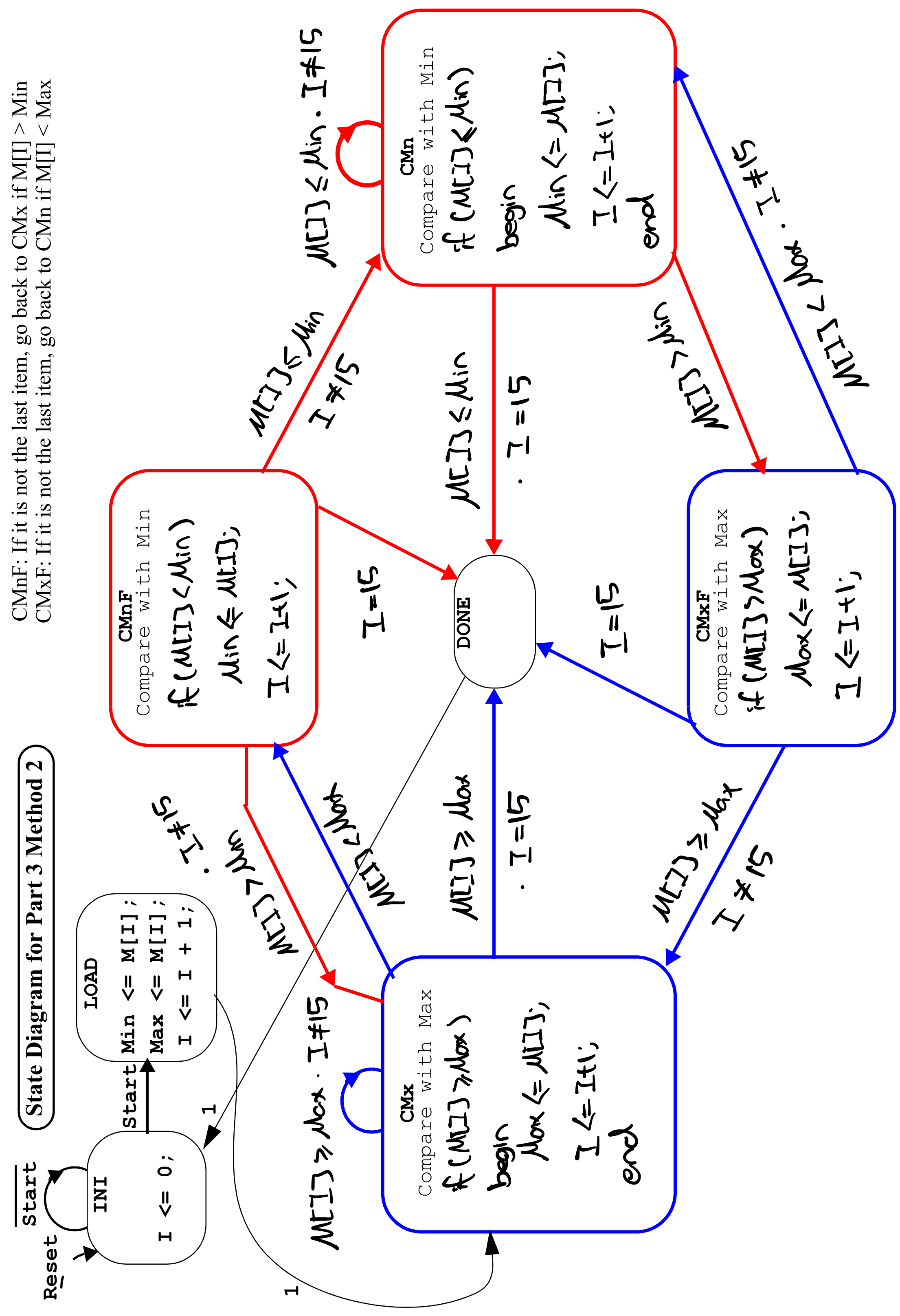
State Diagram for Part 3 Method 1

State Diagram for Part 3 Method 1

CMnF: Compare with Min. for the first time after a series of comparisons in CMx state
CMxF: Compare with Max. for the first time after a series of comparisons in CMn state
Do you increment “I” conditionally or unconditionally in states CMnF and CMxF ?



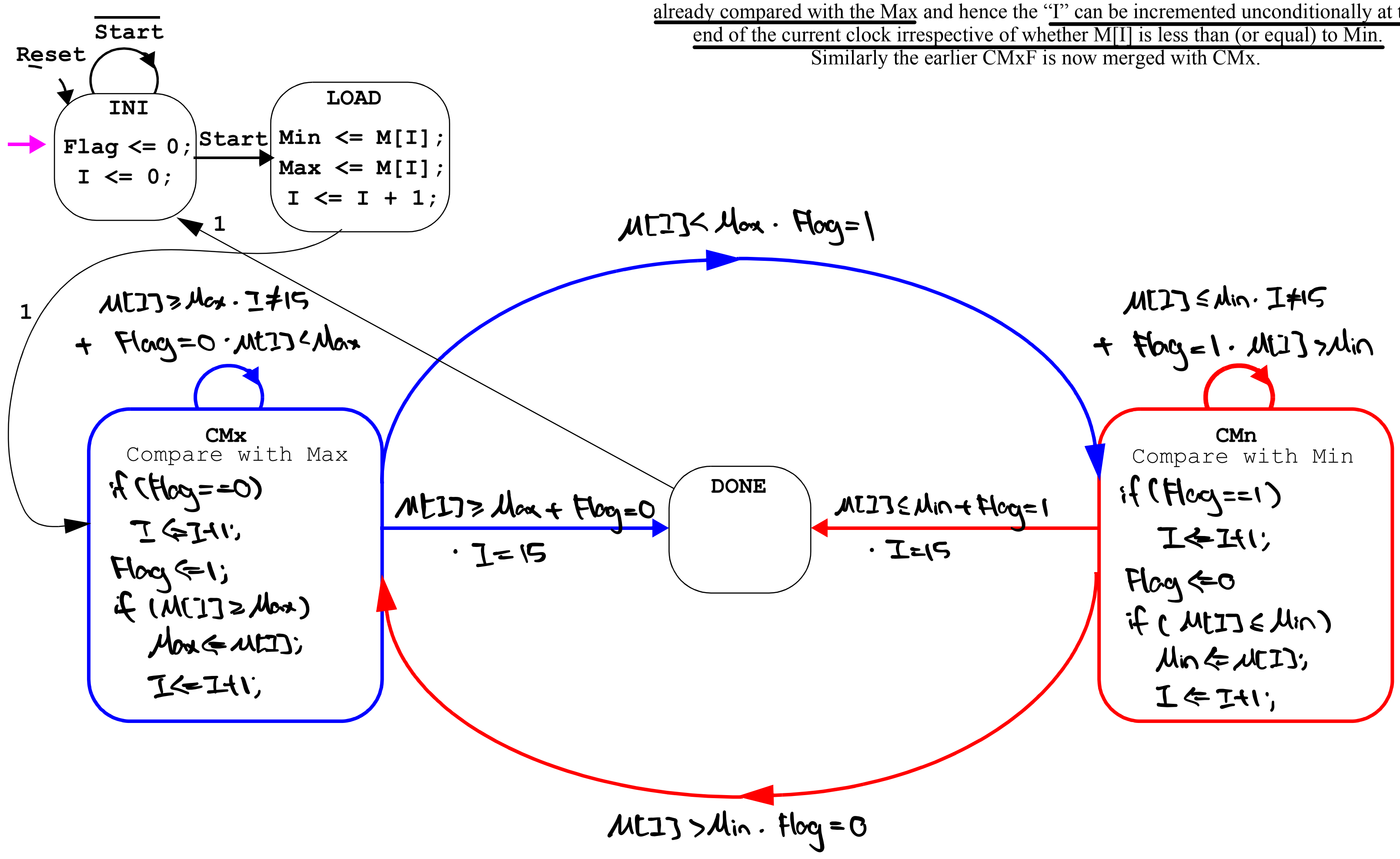
State Diagram for Part 3 Method 2



State Diagram for Part 3 Method 3 (similar to Method 1)

State Diagram for Part 3 Method 3 (similar to Method 1)

Earlier CMnF is now merged with CMn state. When control is received from CMx to CMn, the Flag is received in SET condition for the CMn state to recognize that the current M[I] is already compared with the Max and hence the "I" can be incremented unconditionally at the end of the current clock irrespective of whether M[I] is less than (or equal) to Min. Similarly the earlier CMxF is now merged with CMx.

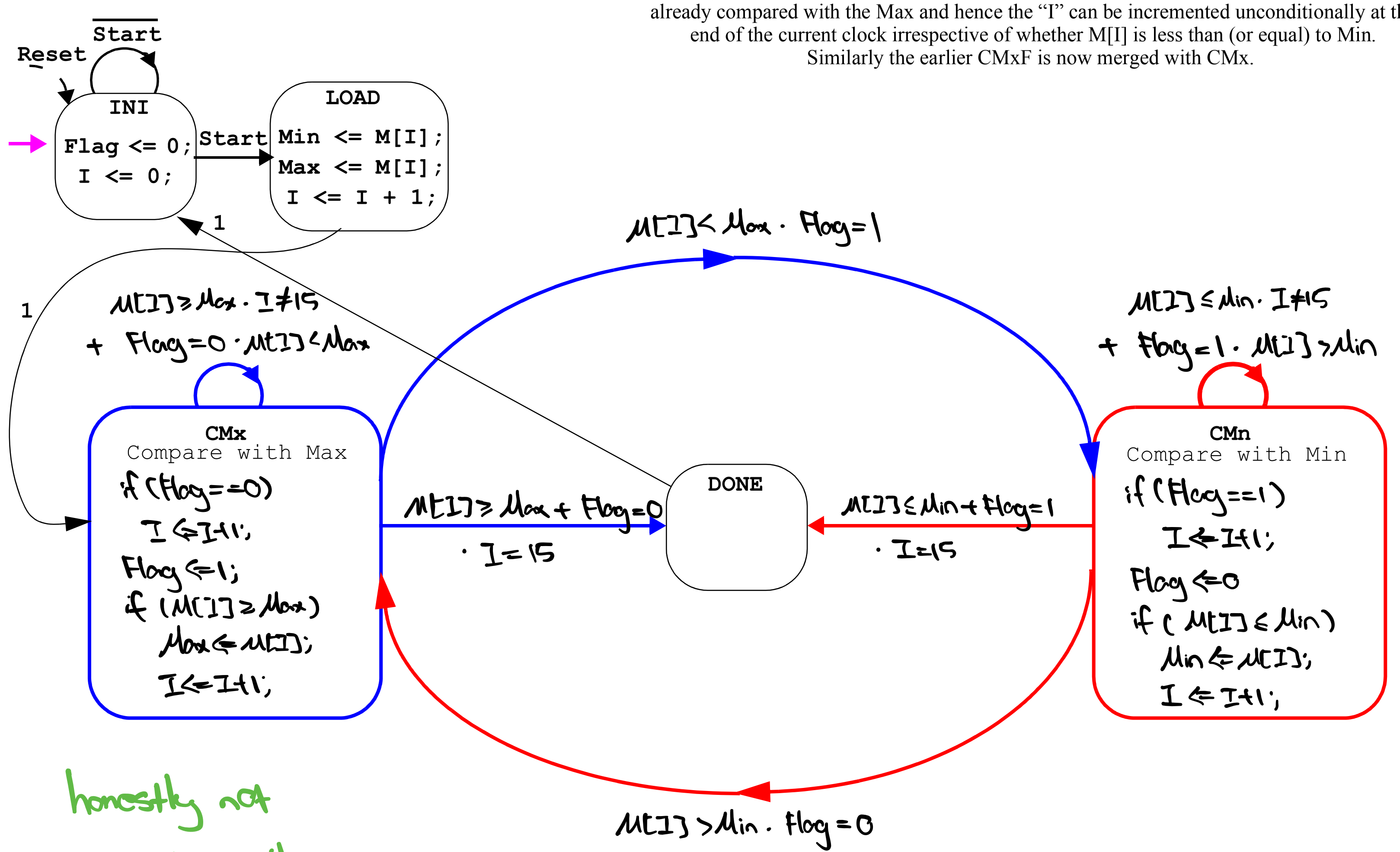


Reset
Start (overbar)
INI
Flag <= 0;
I <= 0;
Start
LOAD
Min <= M[I];
Max <= M[I];
I <= I + 1;
1
1
M[I] ≥ Max · I ≠ 15 + Flag=0 · M[I] < Max
CMx
Compare with Max
if (Flag==0) I <= I+1; Flag <= 1; if (M[I] ≥ Max) Max <= M[I]; I <= I+1;
M[I] < Max · Flag=1
M[I] ≥ Max + Flag=0 · I=15
DONE
M[I] ≤ Min + Flag=1 · I=15
M[I] ≤ Min · I ≠ 15 + Flag=1 · M[I] > Min
CMn
Compare with Min
if (Flag==1) I <= I+1; Flag <= 0 if (M[I] ≤ Min) Min <= M[I]; I <= I+1;
M[I] > Min · Flag=0

State Diagram for Part 3 Method 4 (similar to Method 2)

State Diagram for Part 3 Method 4 (similar to Method 2)

Earlier CMnF is now merged with CMn state. When control is received from CMx to CMn, the Flag is received in SET condition for the CMn state to recognize that the current M[I] is already compared with the Max and hence the "I" can be incremented unconditionally at the end of the current clock irrespective of whether M[I] is less than (or equal) to Min. Similarly the earlier CMxF is now merged with CMx.



honestly not
sure how !!

What you have to do

For each of the 6 designs (Part 1, Part 2, Part 3 Method 1, Part 3 Method 2, Part 3 Method 3, Part 3 Method 4), do the following:

1. Complete the state diagram in hand. Please try to say *loud* (as if you are explaining to someone) what you intended to do in each state and under what conditions you transit to which state.

2. Import the .zip file containing five files into your `C:\ModelSim_projects` directory and unzip it to form a sub-directory with five files. Example: for Part 1, you will have a subdirectory called
`C:\ModelSim_projects\min_max_finder_part1` containing:
A) an incomplete core design file: `min_max_finder_part1.v`
B) a testbench for the above: `min_max_finder_part1_tb.v`
C) a **`.do`** file containing ModelSim commands (to compile, start simulation, setup waveforms, and run simulation to needed length of time): `min_max_finder_part1.do`
D) an additional **`.do`** file (formatting the waveforms which is called by the previous **`.do`** file):
`min_max_finder_part1_wave.do`
E) an empty text file for the test bench to record results: `output_results_part1.txt`

3. Browse through the last four files. Complete the core design file (Example: `min_max_finder_part1.v`). It is best to use an HDL editor such as ModelSim or Notepad++

4. Invoke ModelSim and create a project. For example for Part 1, the project directory will be
`C:\ModelSim_projects\min_max_finder_part1` and project name can be `min_max_finder_part1`.

5. At the VSIM> prompt, execute the **`.do`** file by typing (for Part 1): `do min_max_finder_part1.do`

6. Inspect the console output and/or the results file and also the waveform. Debug as needed. Note that you may end up changing your state diagram design and/or verilog code a few times as this is your first assignment. Start early and seek help early if needed.

7. Finally submit online (through your unix account) one set of files for a team of two students:
`min_max_finder_part1.v, output_results_part1.txt, names.txt`
You need to use the files names *exactly* as stated and follow the submission procedure *exactly* as specified.
We use unix script files to automate grading.

8. Randomly we call upon students to explain the work submitted by them. If you submitted some lab or homework and if you can not explain your work, we infer that you cheated. Also, if your partner did all the work and you include your name in the submission, then we hold BOTH members of the team as partners in cheating.

9. Non-working lab submission: In simulation, it will be evident if your lab is not working. We discourage you from submitting a non-working lab. If you want to submit a non-working lab, ***each*** member of your team needs to send an email to ***all*** lab graders (with a copy to ***all*** TAs) stating in the subject line, "EE457 Non-working lab submission request" and obtain an approval from one of them. Also in the first line of all files submitted, you need to say, "This is a non-working lab submission". Submitting a non-working lab or a partial lab without following the above procedure is interpreted as **an intention to cheat**. Sorry to say all this, but this makes sure that the system works well.

10. We are here to help if you approach sufficiently in advance. So please start early and enjoy.

Questions:

Answers to the these end-of-lab questions (and any waveform with your analysis/justification if asked in future labs) is an **individual effort** (not a team task). For this lab, you need to submit hard copies of (a) the 6 completed state diagrams (completed in hand) and (b) answers to the following questions. No waveforms are needed.

1. Is your Part 1 control unit a Moore machine or a Mealy machine? How about Part 2? How can you tell whether it is a Moore or a Mealy machine? Which of the two options (Moore or Mealy) results in saving clock cycles? Is saving clock cycles always better, or can it possibly result in widening the clock (lowering the frequency)? Discuss briefly using a simple example.

   Part 1 is a Mealy machine because there are if statements inside the boxes of the state diagrams, and so is Part 2. A Mealy machine saves more clock cycles because it goes through less states such as the "undo" states in Moore machines.

   Saving clock cycles is not always better especially in devices where power and frequency is important, for example in GPU's where we don't mind using up more clock cycles if frequency is the priority

2. You know that it may not be a good idea to pack too many operations into one state as the total time it takes to complete all the operations may be fairly long resulting in a slow clock. Here, in this design, we assumed that the memory is a register array and hence is very fast for accessing data. Hence, in a clock, we not only access the memory **`M`** with the index **`I`**, but also compare the **`M[I]`** with **`Max`** and/or **`Min`** in the same clock. On the other hand, if we are given a *slow* memory needing a significant access time comparable to the comparison time, will it be advantageous in part 1 to split the memory access and comparing with the **`Max`** and **`Min`** into two separate states so that the clock rate is higher and overall speed is better? Discuss.

   If we are given slow memory, then it is very much advantageous and needed to split the memory access and the comparison into 2 separate steps so as to not widen the clock or to completely miss the clock period.

   Of course, the disadvantage is that now we need 1 extra clock cycle to first load M[I] into some temporary register BUF, before using the value of BUF for comparison in the next clock cycle.

3. All of you have designed the control unit for the data unit for part 2, utilizing a SINGLE comparator. Our strategy was to compare **`M(I)`** with current **`Max`** first and then, if necessary, with current **`Min`** also. We noted that the total number of clock cycles taken by our control unit is data-dependent. For example, if the data in the memory is already sorted in ascending order, it would take the least number of clock cycles. If the data is in the descending order it would take maximum number of clock cycles. As we do not have any prior knowledge of data distribution we can not optimize the control unit design.

   Now for the sake of our exercise, assume that (you know that) the data has many large chunks of numbers arranged in ascending order and similarly it has also many large chunks of numbers arranged in descending order. Then, to optimize your design, it is desirable that once you go into one of the two states, **`CMx`** (compare with **`Max`**) or **`CMn`** (compare with **`Min`**), you stay there as long as possible. Unlike in your original design where you go back to **`CMx`** from **`CMn`** (as long as it is not max-count), here you perhaps would like to stay in **`CMn`** unless the data item being compared warrants comparison with **`Max`** (the current maximum).

   Your friend, Miss Bruin, thought that she could easily do this problem and arrived at the state diagram given in previous pages. See the page #4 with her state diagram. Do you agree with her design? Is there any flaw in her thinking? Do you increment the counter **I** *conditionally* or *unconditionally* in state **`CMx`**? And similarly in State **`CMn`**? If there is no flaw in her design, complete the state diagram. If you find a flaw in her (we mean in her design), state what is the flaw and how to correct it. This exercise leads to Part 3 (methods 1, 2, 3, and 4).
   Answer this question on the state diagram sheet itself. DONE see page #4