Function specification: Describe in technology agnostic language how the function operates. Note that this step should provide enough clarity such that someone unfamiliar with toUpper() could build the function correctly.

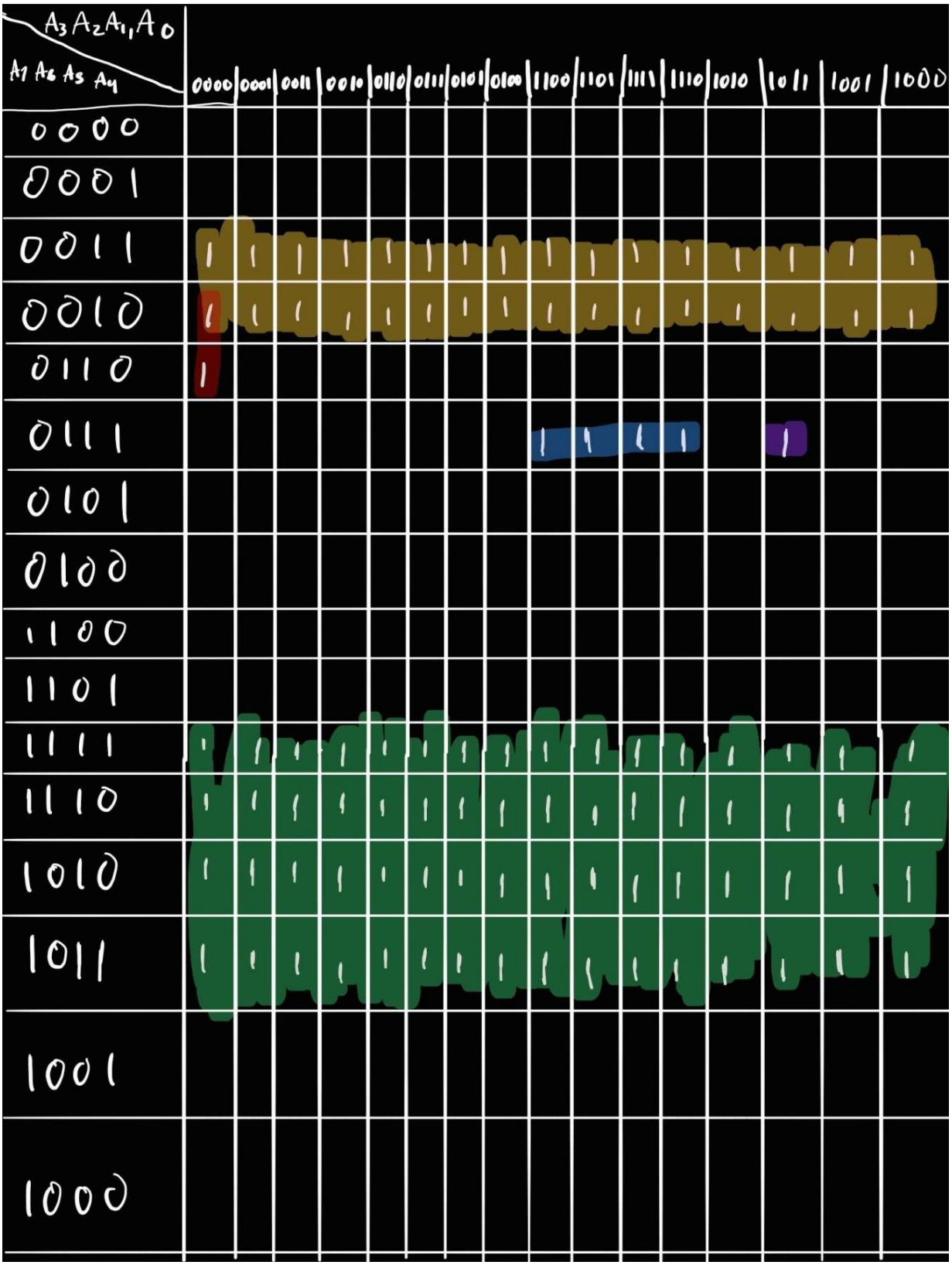
The toUpper() function turns a lowercase letter into its uppercase form. This function is seen in many programming languages and works similarly when compared to the implementation in Verilog. For the Verilog implementation, we give the function an 8-bit ASCII value (char) and through the function it outputs an 8-bit ASCII value. The output that is returned from the function is like the input but what changes is the 5th bit while the rest 0-4 and 6-7 are kept the same. The reason why only the 5th bit is changed is that in ASCII the only difference between a lowercase letter and the uppercase form is the 5th bit with the lowercase letter having a 1 in the 5th bit and the uppercase form having a 0. If the input is not a lowercase letter, then the output will be the same as the input with no changes made.

Circuit design:

For this circuit, I wanted the function to output directly to the 5th bit as mentioned above, because the only thing that changes between the lower and uppercase of a letter is the 5th bit. However, ASCII isn't just letters as depending on the 8-bit value it could mean symbols or characters that aren't letters like $ or +. To design the circuit, it has to:

- Read the input
- Check if the input is a lowercase letter
    - The way it can check is since the lowercase letters are in the range 01100001 to 01111010
- If the input was a lowercase letter, then the function will flip the 5th bit from 1 to 0
- Otherwise, rest of the bits will pass through without any changes applied to them.

To create a Boolean expression for the function I created an 8 var Karnaugh map which serves as the check, 1 is indicated for the values that have their 5th bit as 1 except a-z as they need to change to 0 in order to be uppercase.



The empty space in the kmap boxes are 0

In canonical minterm form this is shown as,

F = ∑m(32 to 63, 96, 123 to 127, 160 to 191, 224 to 255)
F is the function represented by the kmap
Yellow: A7'A6'A5
Red: A7'A5A4'A3'A2'A1'A0'
Blue: A7'A6A5A4A3A2
Purple: A7'A6A5A4A3A2'A1A0
Green: A7A5

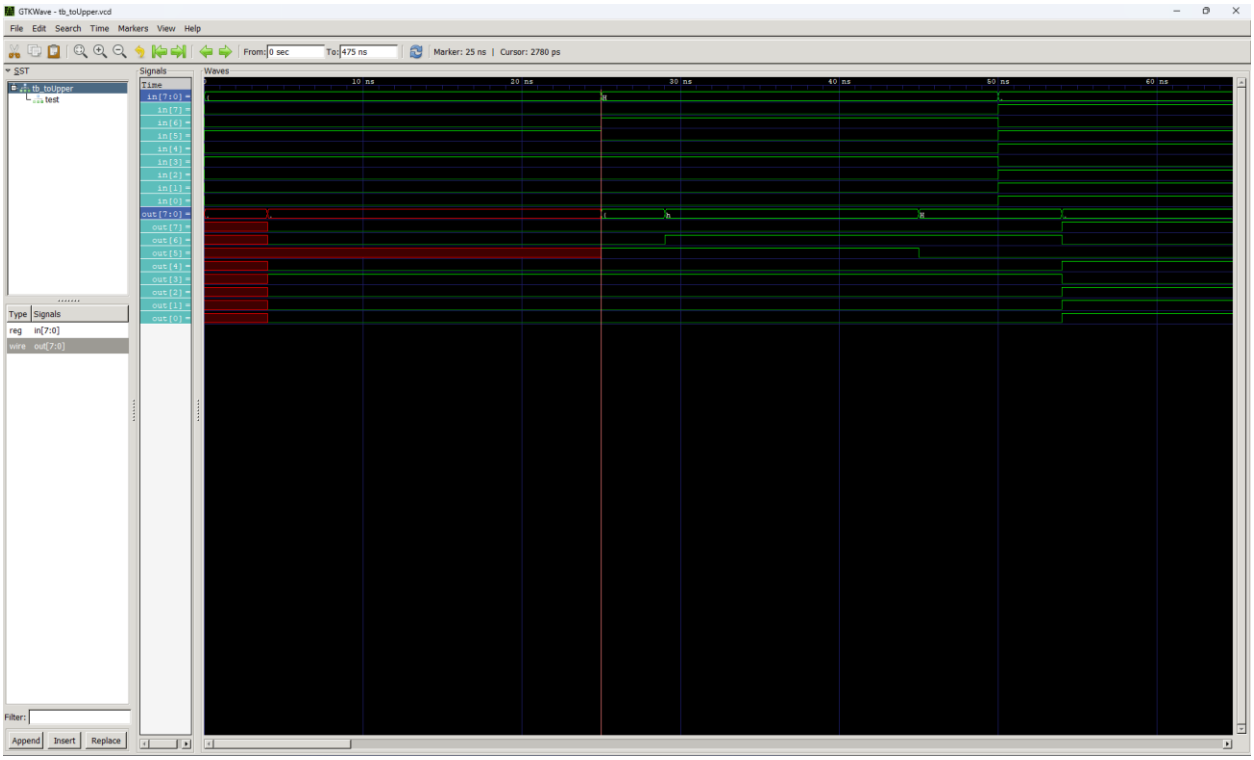
With these groupings we get the Sum Of Products form:
A7'A6'A5 + A7'A5A4'A3'A2'A1'A0' + A7'A6A5A4A3A2 + A7'A6A5A4A3A2'A1A0 + A7A5

For the minimization process it was done through the kmap shown above using the canonical minterm form and making the minimized SOP form through the five groupings from the kmap.

Stress testing and smallest valid inter-input delay and for greatest invalid inter-input delay

Porting over the sum of products expression into Verilog, I saw that it should take no longer than 25ns as the not gates delay are 5ns, and gates were 10ns, and lastly or gate being 10 ns. Although there were more than one not gates they were working in parallel and the same can be said about the and gates. Shown through the total time in the console output and the GTKwave

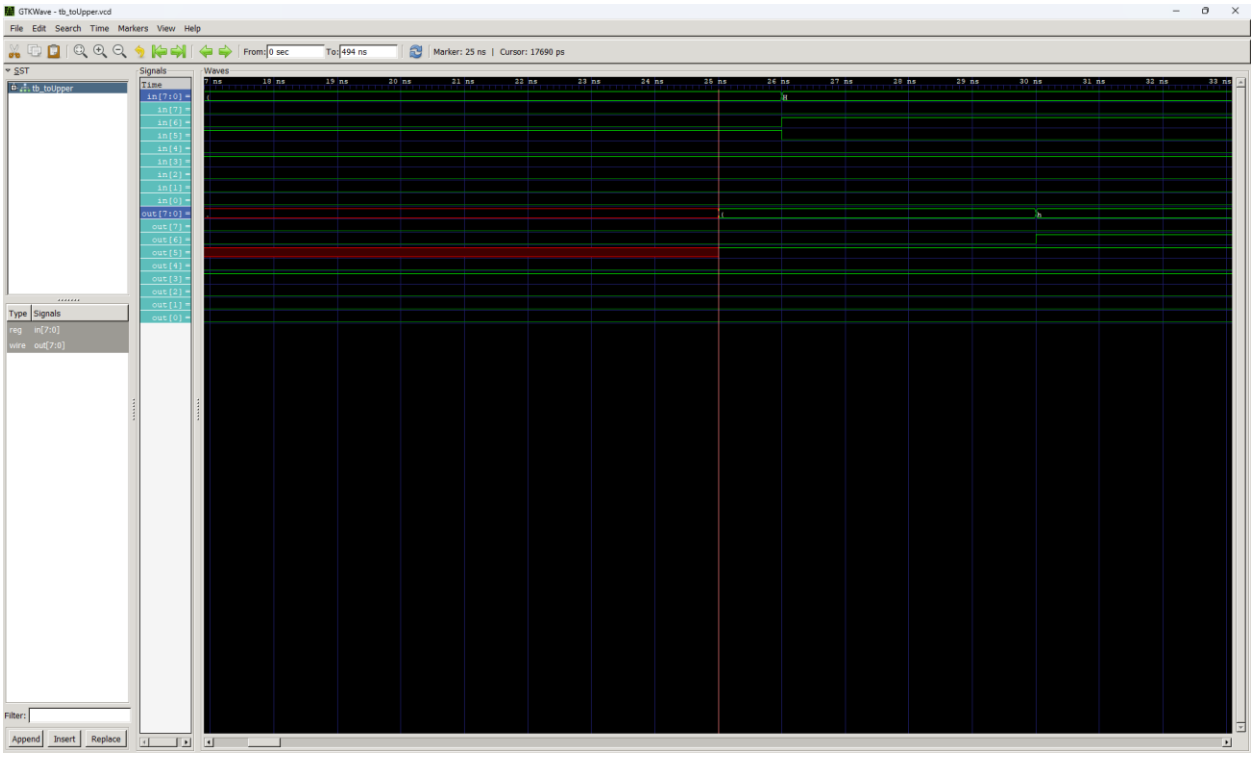
```
Decimal Binary  Ascii_IN        Ascii_OUT
VCD info: dumpfile tb_toUpper.vcd opened for output.
 40     00101000        (
 72     01001000        H       H
183     10110111        ╖       ╖
131     10000011        â       â
124     01111100        |       \
 20     00010100
235     11101011        δ       δ
 97     01100001        a       A
 65     01000001        A       A
122     01111010        z       Z
 71     01000111        G       G
109     01101101        m       M
146     10010010        Æ       Æ
 48     00110000        0
207     11001111        ╧       ╧
 58     00111010        :       Ϛ
123     01111011        {       {
148     10010100        ö       ö
127     01111111                _
tb_toUpper.v:32: $finish called at 475000 (1ps)
```



As you can see it finished at 25ns but what stands out from the console output and the GTKwave is that it doesn't exhibit the correct behavior with the console output not showing the expected output for five testcases showing that it needs a longer delay for a proper output. For the GTKWave although it finished at 25ns we don't actually see it until right after 25ns. From this it is valid to claim that the **greatest invalid inter-input delay** is **25 nanoseconds**.

Adding 1ns to the invalid inter-input delay does actually changes what the console outputs and what the GTKwave looks like.

```
Decimal Binary  Ascii_IN       Ascii_OUT
VCD info: dumpfile tb_toUpper.vcd opened for output.
 40     00101000       (       (
 72     01001000       H       H
183     10110111       ╖       ╖
131     10000011       â       â
124     01111100       |       |
 20     00010100
235     11101011       δ       δ
 97     01100001       a       A
 65     01000001       A       A
122     01111010       z       Z
 71     01000111       G       G
109     01101101       m       M
146     10010010       Æ       Æ
 48     00110000       0       0
207     11001111       ╧       ╧
 58     00111010       :       :
123     01111011       {       {
148     10010100       ö       ö
127     01111111
tb_toUpper.v:32: $finish called at 494000 (1ps)
```



The console now shows all the correct outputs for each of the testcases and we see that through the GTKwave. For the (test case, the correct output is between 25 and 30ns and with display running 26ns after the input which was (is set and in that blue line that

represents 26ns (is shown in the output. From this it is valid to claim that the **smallest valid inter-input delay** is **26 nanoseconds**.

Provide a screenshot of the waveform visualization of the output: