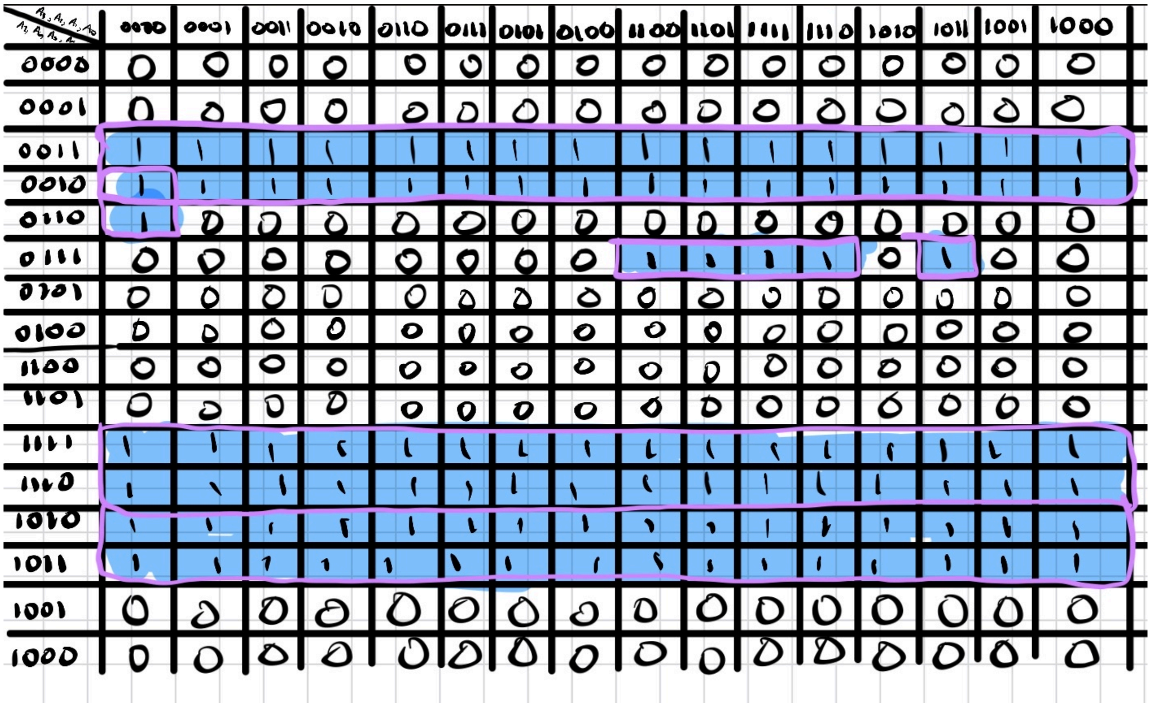
Abdullah Altamir
CS 211 Project 1: ToUPPER
11/4/2025

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 function that we're working with is called toUpper(), which changes a lowercase letter into an uppercase one. The way this function works is that when implemented in Verilog, it takes an 8-bit input in ASCII and outputs a 8-bit value in response. When looking at both input and output, we notice that the A5 bit is the one that changes. Lowercase letters have the fifth bit(A5) turned on, so the function turns off A5(going from 1 to 0) in order for the lowercase letter to become uppercase(which has the A5 bit at 0, which is turned off). If the read input already has the A5 bit turned off, then nothing is changed as that letter is already uppercase.

Circuit Implementation:

- In order to make this circuit, we need an expression to be able to implement it in Verilog. To derive this expression, the ASCII values help determine the difference between a-z and A-Z. When looking at the ASCII values, I noticed that the A5 bit is the only one that is changed between the lowercase and uppercase letters. So the circuit would revolve around detecting the A5 bit and turning it off if it was reading a lowercase letter and not changing the 8 bit value at all if it was reading an uppercase letter. To make this expression, I used an 8 variable Karnaugh map that centered around the A5 bit. If the A5 bit was on, the value was given a 1 in the map. If the value represented a-z, the value 0 was given in the map. The following k-map was constructed(see below):



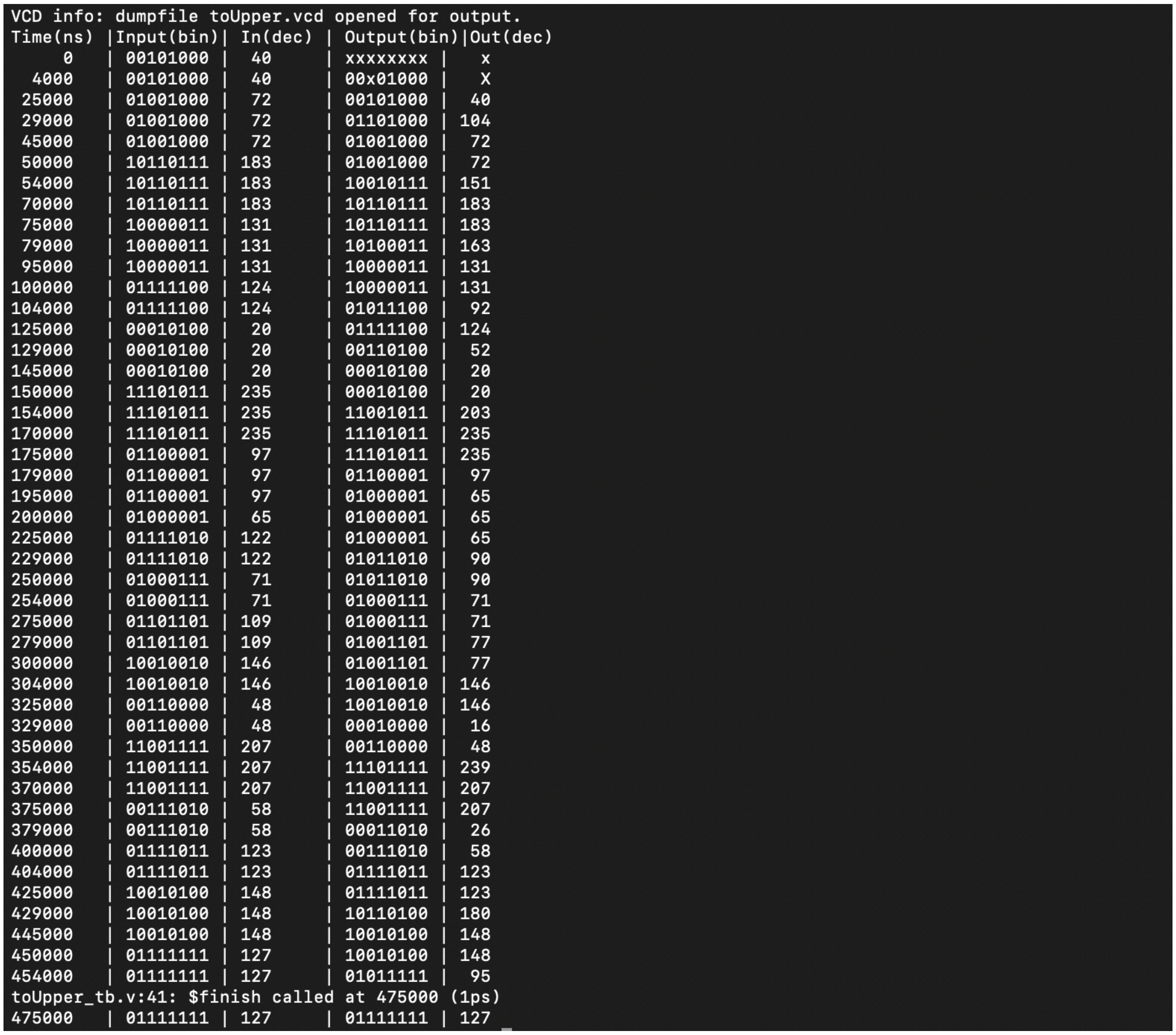
A3, A2, A1, A0 / A7, A6, A5, A4	0000	0001	0011	0010	0110	0111	0101	0100	1100	1101	1111	1110	1010	1011	1001	1000
0000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
0110	1	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0111	0	0	0	0	0	0	0	0	1	1	1	1	0	1	0	0
0101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
0100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1100	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1101	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1111	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1110	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1010	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1011	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1	1
1001	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1000	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

Figure 1: 8-Variable Karnaugh Map for toUpper()

We can represent this Karnaugh map in its canonical minterm form, which is:
F = Σm(32-63, 96, 123-127, 160-191, 224-255). This represents all the 1s in the k-map and we can simplify this into a boolean expression:
SumOfProducts: A7’A6’A5 + A7’A5A4’A3’A2’A1’A0’ + A7’A6A5A4A3A2 + A7’A6A5A4A3A2’A1A0 + A7A5
As shown, this expression was derived from usage of the 8 variable Karnaugh Map, by grouping up the terms and creating an expression. We can now implement our expression into Verilog, and create toUpper().

Stress Testing Function and Finding Smallest Possible Delay

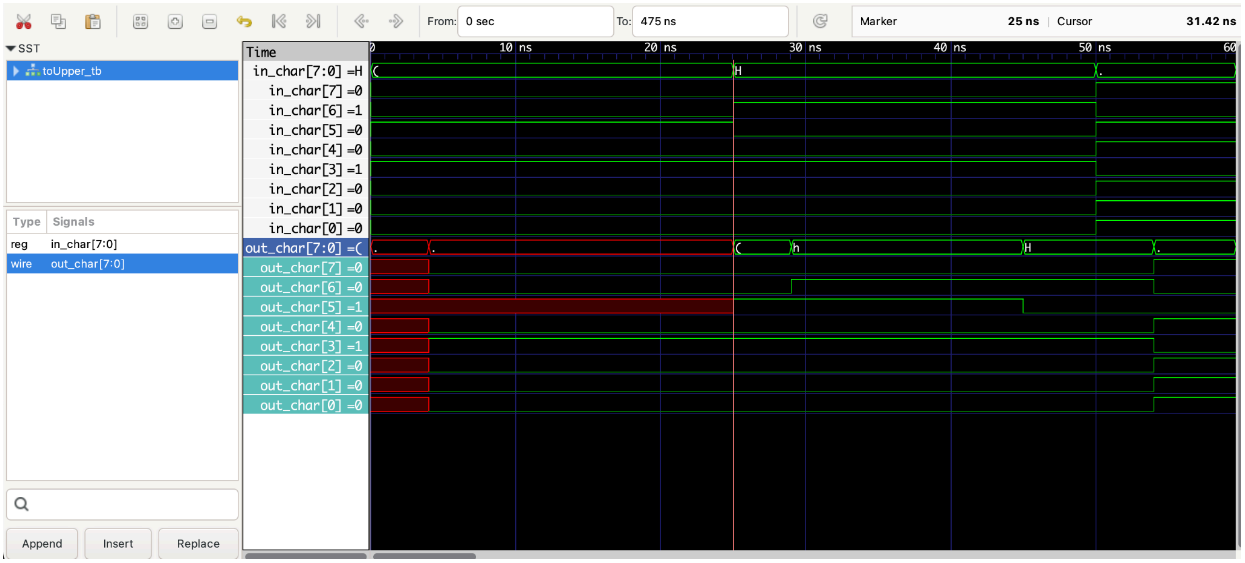
- After implementing the following boolean expression into Verilog, it was stress tested to ensure that it would work for specific inputs. By adding up the delay for the gates used(and not and or), the delay was counted to be 25ns, and the results are shown below:



```
VCD info: dumpfile toUpper.vcd opened for output.
Time(ns) |Input(bin)| In(dec) | Output(bin)|Out(dec)
     0   | 00101000 |  40     | xxxxxxxx |   x
  4000   | 00101000 |  40     | 00x01000 |   X
 25000   | 01001000 |  72     | 00101000 |  40
 29000   | 01001000 |  72     | 01101000 | 104
 45000   | 01001000 |  72     | 01001000 |  72
 50000   | 10110111 | 183     | 01001000 |  72
 54000   | 10110111 | 183     | 10010111 | 151
 70000   | 10110111 | 183     | 10110111 | 183
 75000   | 10000011 | 131     | 10110111 | 183
 79000   | 10000011 | 131     | 10100011 | 163
 95000   | 10000011 | 131     | 10000011 | 131
100000   | 01111100 | 124     | 10000011 | 131
104000   | 01111100 | 124     | 01011100 |  92
125000   | 00010100 |  20     | 01111100 | 124
129000   | 00010100 |  20     | 00110100 |  52
145000   | 00010100 |  20     | 00010100 |  20
150000   | 11101011 | 235     | 00010100 |  20
154000   | 11101011 | 235     | 11001011 | 203
170000   | 11101011 | 235     | 11101011 | 235
175000   | 01100001 |  97     | 11101011 | 235
179000   | 01100001 |  97     | 01100001 |  97
195000   | 01100001 |  97     | 01000001 |  65
200000   | 01000001 |  65     | 01000001 |  65
225000   | 01111010 | 122     | 01000001 |  65
229000   | 01111010 | 122     | 01011010 |  90
250000   | 01000111 |  71     | 01011010 |  90
254000   | 01000111 |  71     | 01000111 |  71
275000   | 01101101 | 109     | 01000111 |  71
279000   | 01101101 | 109     | 01001101 |  77
300000   | 10010010 | 146     | 01001101 |  77
304000   | 10010010 | 146     | 10010010 | 146
325000   | 00110000 |  48     | 10010010 | 146
329000   | 00110000 |  48     | 00010000 |  16
350000   | 11001111 | 207     | 00110000 |  48
354000   | 11001111 | 207     | 11101111 | 239
370000   | 11001111 | 207     | 11001111 | 207
375000   | 00111010 |  58     | 11001111 | 207
379000   | 00111010 |  58     | 00011010 |  26
400000   | 01111011 | 123     | 00111010 |  58
404000   | 01111011 | 123     | 01111011 | 123
425000   | 10010100 | 148     | 01111011 | 123
429000   | 10010100 | 148     | 10110100 | 180
445000   | 10010100 | 148     | 10010100 | 148
450000   | 01111111 | 127     | 10010100 | 148
454000   | 01111111 | 127     | 01011111 |  95
toUpper_tb.v:41: $finish called at 475000 (1ps)
475000   | 01111111 | 127     | 01111111 | 127
```

Figure 2: Terminal output results for toUpper()

As shown, the first two outputs are recorded to be x due to signal instability, which is then fixed after those two outputs. We get a total of 475000 ns, which is correct in our case since 25 ns times 19 inputs gives us 475 ns. This means all tests cases were completed successfully. Here is the gtkwave screenshot that shows that 25ns ran successfully:



From: 0 sec To: 475 ns Marker 25 ns | Cursor 31.42 ns

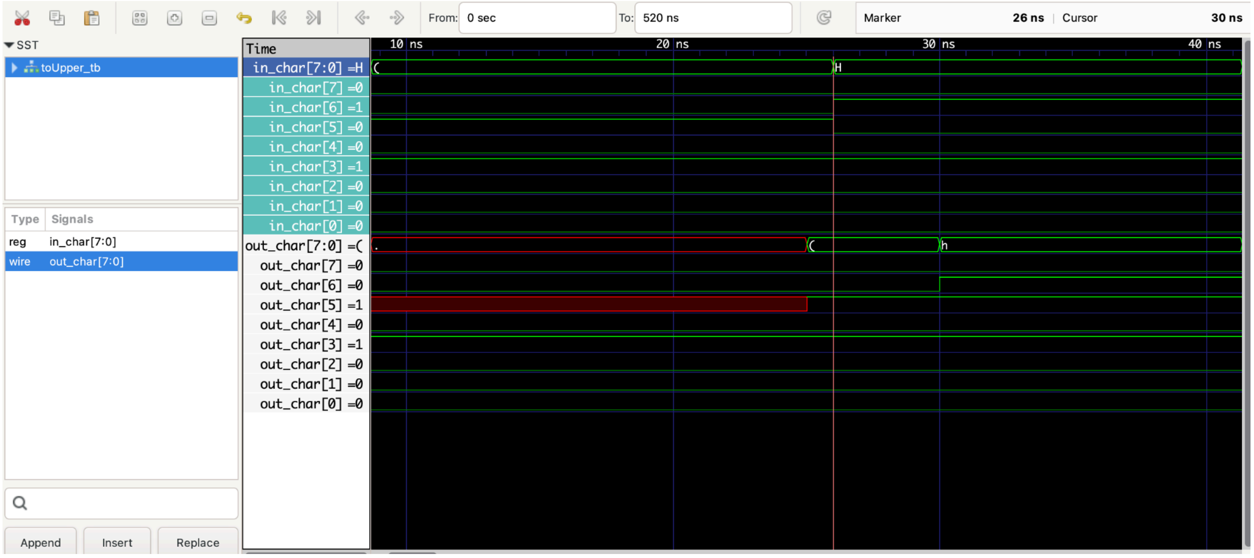
```
SST
  toUpper_tb

Type  Signals
reg   in_char[7:0]
wire  out_char[7:0]
```

```
Time
in_char[7:0] =H
  in_char[7] =0
  in_char[6] =1
  in_char[5] =0
  in_char[4] =0
  in_char[3] =1
  in_char[2] =0
  in_char[1] =0
  in_char[0] =0
out_char[7:0] =(
  out_char[7] =0
  out_char[6] =0
  out_char[5] =1
  out_char[4] =0
  out_char[3] =1
  out_char[2] =0
  out_char[1] =0
  out_char[0] =0
```

Figure 3: GTKWAVE screenshot showing results for 25ns

As shown, the circuit has successfully completed in 25ns. However, the issue here is that 25ns does not give us the correct total outputs(meaning in some cases the test cases are not successful). Therefore this means that the greatest invalid input-delay is 25ns as found, and the above figure also shows the failures of those test cases. Therefore, it was determined that 26 nanoseconds was the smallest valid inter-input delay, and the GTKWAVE results correspond with this statement:



From: 0 sec To: 520 ns Marker 26 ns | Cursor 30 ns

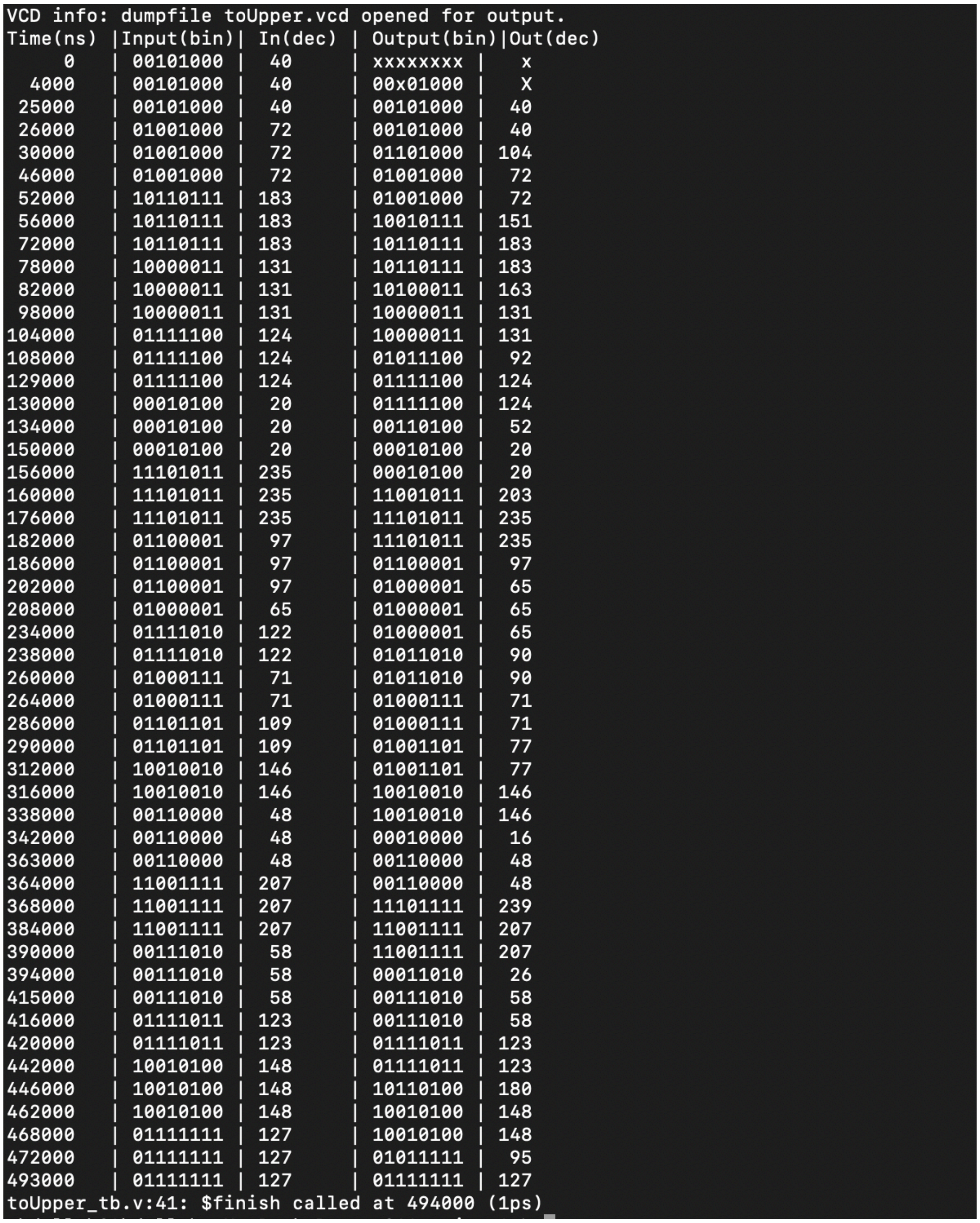
```
SST
  toUpper_tb

Type  Signals
reg   in_char[7:0]
wire  out_char[7:0]
```

```
Time
in_char[7:0] =H
  in_char[7] =0
  in_char[6] =1
  in_char[5] =0
  in_char[4] =0
  in_char[3] =1
  in_char[2] =0
  in_char[1] =0
  in_char[0] =0
out_char[7:0] =(
  out_char[7] =0
  out_char[6] =0
  out_char[5] =1
  out_char[4] =0
  out_char[3] =1
  out_char[2] =0
  out_char[1] =0
  out_char[0] =0
```

Figure 4: GTKWAVE screenshot showing results for 26ns

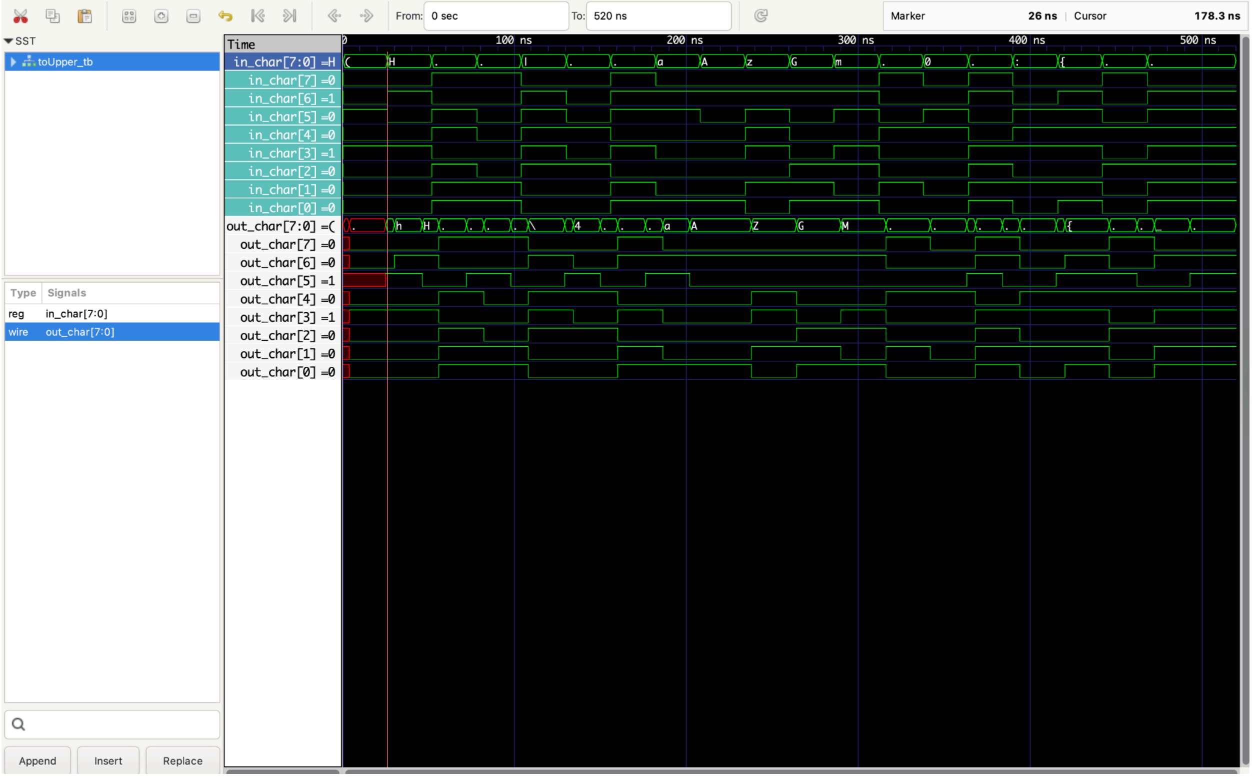
As shown, the circuit was not only successful but the test cases were also successful with 26ns, as shown in this terminal image:



```
VCD info: dumpfile toUpper.vcd opened for output.
Time(ns) |Input(bin)| In(dec) | Output(bin)|Out(dec)
     0   | 00101000 |  40     | xxxxxxxx |   x
  4000   | 00101000 |  40     | 00x01000 |   X
 25000   | 00101000 |  40     | 00101000 |  40
 26000   | 01001000 |  72     | 00101000 |  40
 30000   | 01001000 |  72     | 01101000 | 104
 46000   | 01001000 |  72     | 01001000 |  72
 52000   | 10110111 | 183     | 01001000 |  72
 56000   | 10110111 | 183     | 10010111 | 151
 72000   | 10110111 | 183     | 10110111 | 183
 78000   | 10000011 | 131     | 10110111 | 183
 82000   | 10000011 | 131     | 10100011 | 163
 98000   | 10000011 | 131     | 10000011 | 131
104000   | 01111100 | 124     | 10000011 | 131
108000   | 01111100 | 124     | 01011100 |  92
129000   | 01111100 | 124     | 01111100 | 124
130000   | 00010100 |  20     | 01111100 | 124
134000   | 00010100 |  20     | 00110100 |  52
150000   | 00010100 |  20     | 00010100 |  20
156000   | 11101011 | 235     | 00010100 |  20
160000   | 11101011 | 235     | 11001011 | 203
176000   | 11101011 | 235     | 11101011 | 235
182000   | 01100001 |  97     | 11101011 | 235
186000   | 01100001 |  97     | 01100001 |  97
202000   | 01100001 |  97     | 01000001 |  65
208000   | 01000001 |  65     | 01000001 |  65
234000   | 01111010 | 122     | 01000001 |  65
238000   | 01111010 | 122     | 01011010 |  90
260000   | 01000111 |  71     | 01011010 |  90
264000   | 01000111 |  71     | 01000111 |  71
286000   | 01101101 | 109     | 01000111 |  71
290000   | 01101101 | 109     | 01001101 |  77
312000   | 10010010 | 146     | 01001101 |  77
316000   | 10010010 | 146     | 10010010 | 146
338000   | 00110000 |  48     | 10010010 | 146
342000   | 00110000 |  48     | 00010000 |  16
363000   | 00110000 |  48     | 00110000 |  48
364000   | 11001111 | 207     | 00110000 |  48
368000   | 11001111 | 207     | 11101111 | 239
384000   | 11001111 | 207     | 11001111 | 207
390000   | 00111010 |  58     | 11001111 | 207
394000   | 00111010 |  58     | 00011010 |  26
415000   | 00111010 |  58     | 00111010 |  58
416000   | 01111011 | 123     | 00111010 |  58
420000   | 01111011 | 123     | 01111011 | 123
442000   | 10010100 | 148     | 01111011 | 123
446000   | 10010100 | 148     | 10110100 | 180
462000   | 10010100 | 148     | 10010100 | 148
468000   | 01111111 | 127     | 10010100 | 148
472000   | 01111111 | 127     | 01011111 |  95
493000   | 01111111 | 127     | 01111111 | 127
toUpper_tb.v:41: $finish called at 494000 (1ps)
```

<u>Figure 5: Terminal outputs for 26ns</u>

The pictures correspond with the statements, and thus 26 nanoseconds is the smallest inter-input delay. Here is the GTKWAVE for the entirety of the 19 inputs for 26 nanoseconds:



From: 0 sec

To: 520 ns

Marker 26 ns | Cursor 178.3 ns

Type	Signals
reg	in_char[7:0]
wire	out_char[7:0]

```
Time
in_char[7:0] =H
  in_char[7] =0
  in_char[6] =1
  in_char[5] =0
  in_char[4] =0
  in_char[3] =1
  in_char[2] =0
  in_char[1] =0
  in_char[0] =0
out_char[7:0] =(
 out_char[7] =0
 out_char[6] =0
 out_char[5] =1
 out_char[4] =0
 out_char[3] =1
 out_char[2] =0
 out_char[1] =0
 out_char[0] =0
```

SST
toUpper_tb
Append
Insert
Replace
0
100 ns
200 ns
300 ns
400 ns
500 ns
(H . . l . . a A z G m . 0 . : { . .
. h H . . . . \ 4 . . . a A Z G M . . . . . { . . _ .

<u>*Figure 6: GTKWAVE screenshot showing all 19 inputs for 26 ns</u>