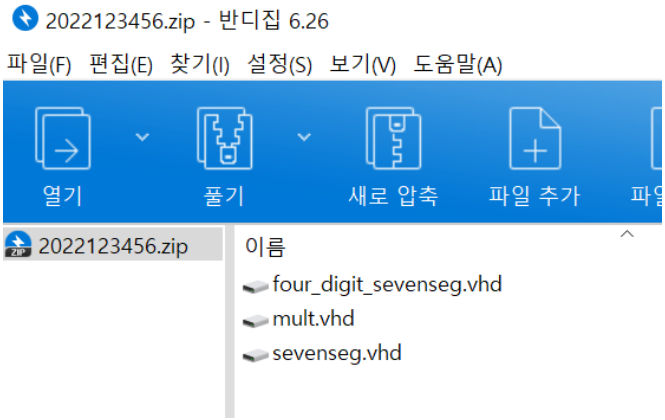
## HW#6. Due 11/12 Fri 11:59 PM

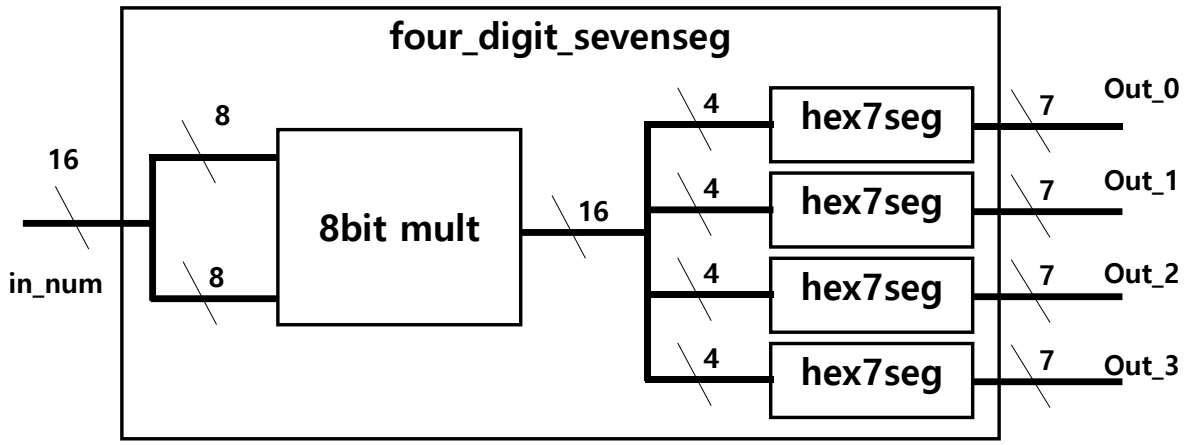
In this assignment, you will design your circuit using VHDL. The circuit is similar to what you have designed in HW5, but more complicated. Thanks to VHDL, it would be easier to create and debug (once you get used to it)

- Submit a zipped VHDL(.vhd) files for the problem.
- The name of the zip file should be your_id.zip
- **Stick to the given port names**. We will use an automatic grader and your answer will be wrong if you use a wrong port name. Case-insensitive.
- The only library you can use is **ieee.std_logic_1164.all. anything else is strictly prohibited**
- Example: 'four_digit_sevenseg.vhd' must be there. The rest are optional. They are only needed if you used other components.




-

**[Final Goal]**



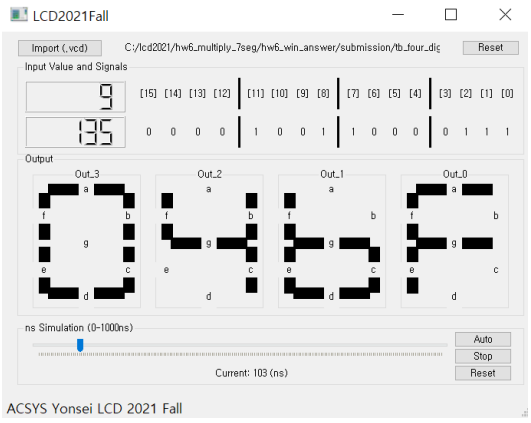

The circuit above describes what you need to create.

- It takes *in_num*(15 downto 0) as input
- It outputs four seven segment outputs, named *out_0* through *out_3*
- The *in_num* is split into upper 8 bits and lower 8 bits.
- The 8-bit multiplier multiplies the two numbers, outputing a 16-bit number.
- The 16-bit number is split into 4 digits of hexadecimal numbers.
- Each hexadecimal number is visually output using 7seg display.

A skeleton vhd file with the top entity is given. Do not modify the entity. Just fill in the architecture.

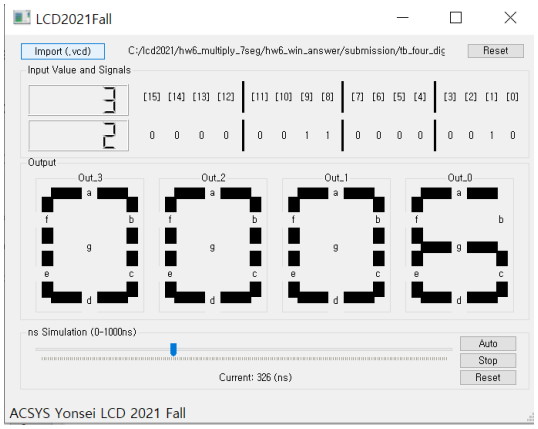
Here are some examples of the behavior:

- 0x0987: 0x09 * 0x87 (135) = 0x04bF

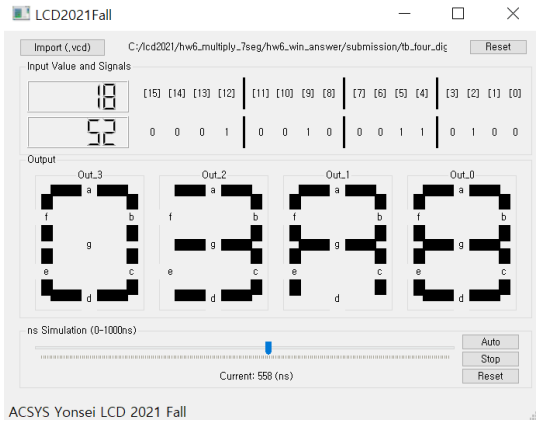



**(caution: this is not 046f, but is 04bf)**

- 0x0302: 0x03*0x02 = 0x0006




- 0x1234: 0x12 (18) * 0x34 (52) = 0x03A8



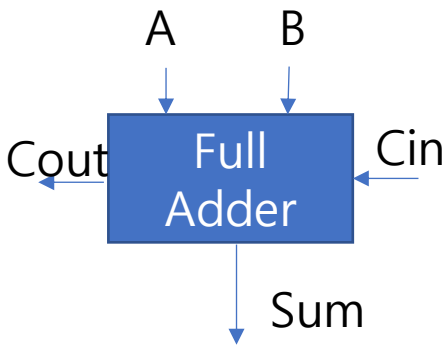

As in HW5, if you know what to do, please go ahead. You now have all the information for HW6. If you don't know where to start, below are some obvious step by step guidelines on

## 1. 8-bit Combinational Multiplier

As the first step, let's build a combinational multiplier. You know what to do, right?

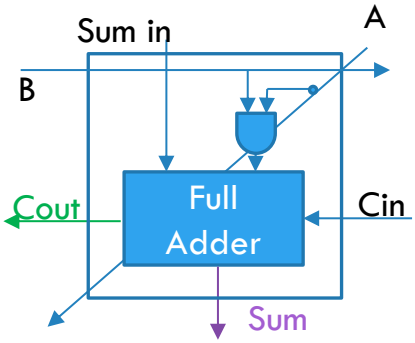
You must have had an experience of building a 4-bit multiplier from HW5. You just need to extend that to 8 bits, using VHDL. Below is one example on how you do it.

a. Build a full adder. This will be your first vhdl file. It is up to you whether you write it as a whole module, or by integrating two half adders.




b. [Unit test] Are you sure that your design works? Even if you are, you have to verify it module by module. At some point, you will face your circuit not working as intended, and you want to be able to perfectly trust your smaller building blocks. Same as in other programming. So let's perform a unit test. Write a testbench, either using waveform editor, or using VHDL (we strongly suggest using VHDL, because we are trying to leave the schematic world now).

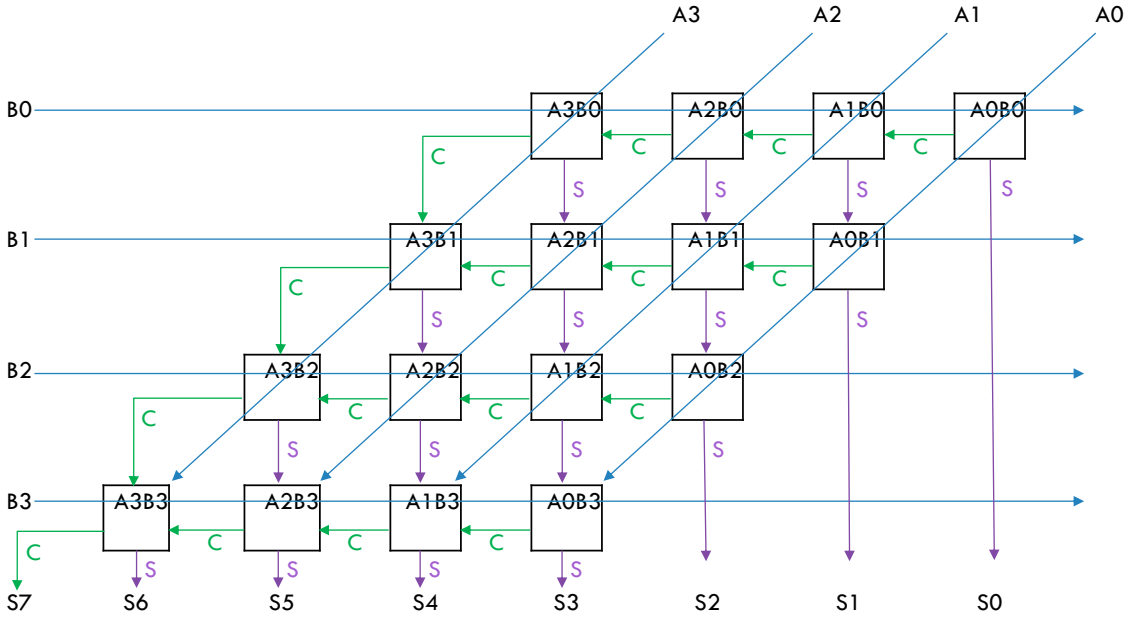
c. If your full adder works well, use it as a component to build a processing element for the multiplier.




d. Of course, write a testbench to verify your design!

e. Now it's time to build a multiplier. Our goal is to build an 8-bit multiplier, but we probably

want to do it in a smaller scale first. Build a 2-bit or 4-bit multiplier first. (or you can go from 2->4->8. This might be better if you are not sure about your understanding of the multiplier.)




f. Of course, write a testbench and verify it.

g. Let's build an 8-bit multiplier. It is just an extension of the what we have seen from the textbook (or the slides). For this, you will need to use 64 instances of the PE. There are three ways I can think of

   i. Just write down 64 instances. They are not that many as you think. But be careful for mistakes.

   ii. If you look at the figure above, each row looks almost identical. Write a module for a row of 8 PEs, and put 8 instances of such row modules.

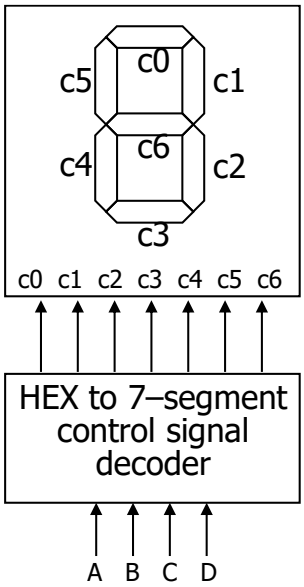
   iii. If you want to get a little bit advanturous, you can try the generate statements. There are many materials, including: (http://web.engr.oregonstate.edu/~traylor/ece474/vhdl_lectures/essential_vhdl_pdfs/essential_vhdl61-76.pdf)

h. Of course, write a testbench and verify it.

i. Congratulations! You are done! Optionally you can try to reduce the unnecessary resources as we have learned from the class. Of course it is not mandatory.

## 2. Hex to 7seg controller

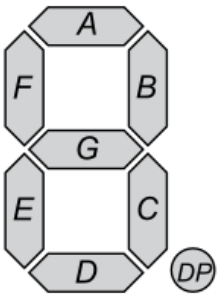
At some point, I think it has been mentioned that the 7seg is not only for numbers, but it can be used to display certain set of alphabets too. Luckily, it can show all of a-f, making it possible to display hexadecimal numbers. Let's refer to the wikipedia page for the truth table.




https://en.wikipedia.org/wiki/Seven-segment_display#Hexadecimal

Hexadecimal encodings for displaying the digits 0 to F[22][23]

| Digit | Display | p | a | b | c | d | e | f | g | pabcdefg | hex pabcdefg | pgfedcba | hex pgfedcba |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | | on | on | on | on | on | on | | 01111110 | 0x7E | 00111111 | 0x3F |
| 1 | 1 | | | on | on | | | | | 00110000 | 0x30 | 00000110 | 0x06 |
| 2 | 2 | | on | on | | on | on | | on | 01101101 | 0x6D | 01011011 | 0x5B |
| 3 | 3 | | on | on | on | on | | | on | 01111001 | 0x79 | 01001111 | 0x4F |
| 4 | 4 | | | on | on | | | on | on | 00110011 | 0x33 | 01100110 | 0x66 |
| 5 | 5 | | on | | on | on | | on | on | 01011011 | 0x5B | 01101101 | 0x6D |
| 6 | 6 | | on | | on | on | on | on | on | 01011111 | 0x5F | 01111101 | 0x7D |
| 7 | 7 | | on | on | on | | | | | 01110000 | 0x70 | 00000111 | 0x07 |
| 8 | 8 | | on | on | on | on | on | on | on | 01111111 | 0x7F | 01111111 | 0x7F |
| 9 | 9 | | on | on | on | on | | on | on | 01111011 | 0x7B | 01101111 | 0x6F |
| A | A | | on | on | on | | on | on | on | 01110111 | 0x77 | 01110111 | 0x77 |
| b | b | | | | on | on | on | on | on | 00011111 | 0x1F | 01111100 | 0x7C |
| C | C | | on | | | on | on | on | | 01001110 | 0x4E | 00111001 | 0x39 |
| d | d | | | on | on | on | on | | on | 00111101 | 0x3D | 01011110 | 0x5E |
| E | E | | on | | | on | on | on | on | 01001111 | 0x4F | 01111001 | 0x79 |
| F | F | | on | | | | on | on | on | 01000111 | 0x47 | 01110001 | 0x71 |



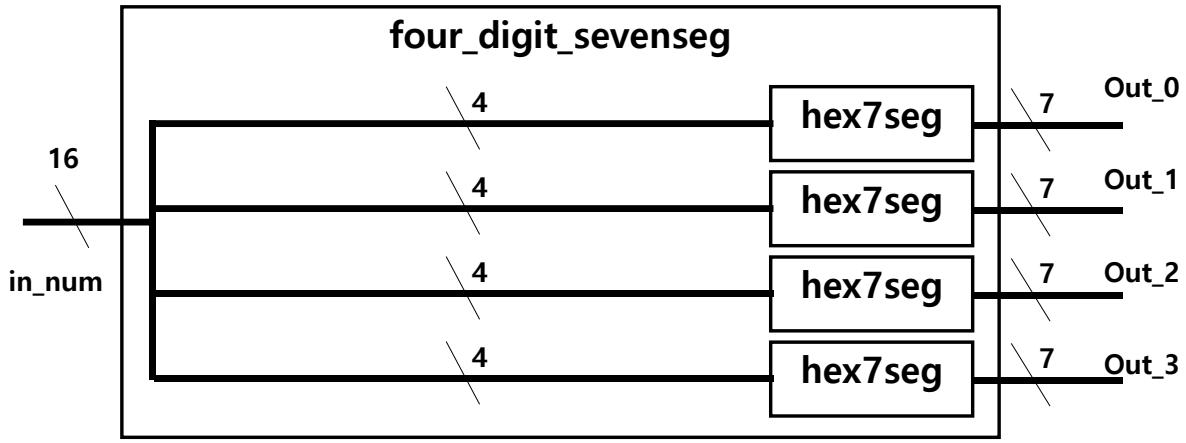

- As you see, in our notation, c6=g, c5=e, …, c0=a.
  Therefore, we would probably want to use the **gfedcba** column of the table (we don't use *p* in our 7seg).
- In this assignment, you have all the freedom to name your ports for the 7seg controller, unlike HW5. You can use the same specifications from HW5 or define your own.
- One way to implement this in VHDL is to build a 2-level circuit as in HW5.
- Another way is to simply state all possible cases, such as conditional signal assignment or if..else statements within processes.
- After you are done, make sure it works by unit test with your own testbench.

## 3. Putting them all together

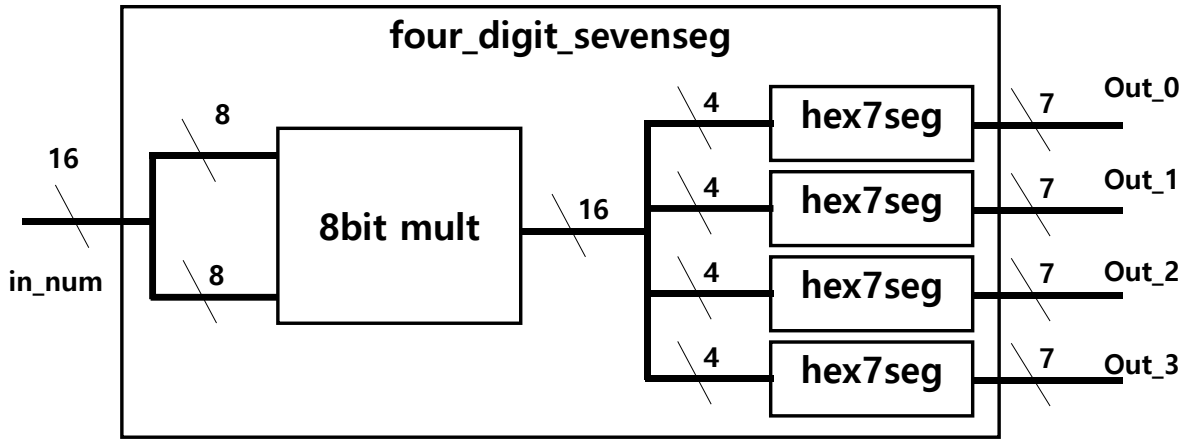
Now is the time to use the multiplier and the hex_to_7seg for the final circuit. Again, what I have described below is just one suggested way of implementing the final circuit. If you are confident about what you do, just follow your guts. You're fine as long as it works.

a. Declare the components (multiplier and the 7seg controller)

b. You probably want to check the 7seg display first. Put four instances of the 7seg controller component into the top entity (four_digit_sevenseg). Then, split the 16-bit input into four and connect each 4-bit signal to the controllers. Use the new 7seg visualizer explained below to check if your 7seg works well (if necessary, write your own testbench).




c. If all four of your 7seg display work well, you are ready to put the multiplier instance. Split the 16-bit input to two 8-bit signals. Input each to the multiplier. Finally, connect the output to the 7seg controllers.




d. Use the provided testbench to confirm that your design works. If needed, write your own testbench for more extensive verification.

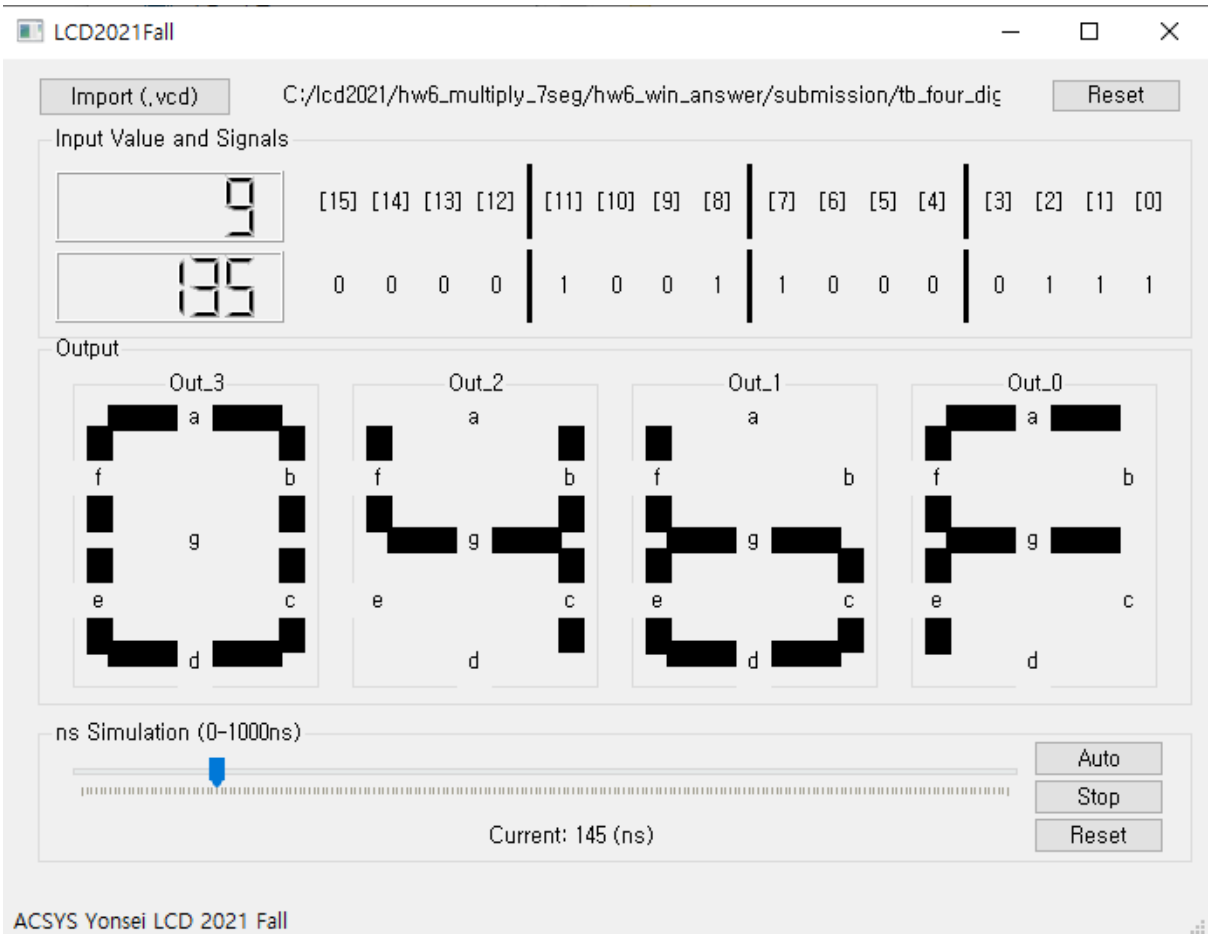
# 4digit 7segment visualizer (Windows)

- *Because we suspect that no one is using the linux version, we will distribute the windows version. Linux version is still available and will be distributed upon a separate request.*

For this assignment, we provide a modified visualizer for displaying two input numbers in decimal and output number in hex. It is not mandatory, but again, we believe it is nearly the only way to ensure that your design is correct. We strongly suggest that you use this tool.

The visualizer takes two 8bit numbers(16bit), which are the inputs of given top entity. You should not modify the entity definition.

```
-- 4 digit 7 segment entity
entity four_digit_sevenseg is
    port(
        in_num: in std_logic_vector(15 downto 0); -- input value
        out_0:  out std_logic_vector(6 downto 0); -- 0th digit
        out_1:  out std_logic_vector(6 downto 0); -- 1th digit
        out_2:  out std_logic_vector(6 downto 0); -- 2nd digit
        out_3:  out std_logic_vector(6 downto 0) --3rd digit
    );
end four_digit_sevenseg;
```




Above is what you're supposed to see. Notice the two input numbers being displayed on the top

left. The instructions are the same as HW5.

1. You should have Quartus and modelsim_ase installed in your computer.
2. Copy unzipped attachments into **only English** path. Non-english dir path is known to corrupt the simulation.
3. Add **"quartus"** and **"modelsim_ase"** into path environment variables. (Google on "windows 10 path variable setting" or similar keywords:

   e.g., https://www.architectryan.com/2018/08/31/how-to-change-environment-variables-on-windows-10/ )

   Edit on "Path" variable > "New" > add paths for "quartus" and "modelsim" like below. Actual path can vary depending on where you installed them.

4. We provide a benchmark for you, written using VHDL. You can modify it if you want, but will will perform tests based on our own testbench.

```
37    process
38    begin
39        -- 0000
40        tb_in_num <= "0000000000000000";
41        wait for 100ns;
42        -- 0987
43        tb_in_num <= "0000100110000111";
44        wait for 100ns;
45        -- 0061
46        tb_in_num <= "0000000001100001";
47        wait for 100ns;
48        -- 0302
49        tb_in_num <= "0000001100000010";
50        wait for 100ns;
51        -- 0005
52        tb_in_num <= "0000000000000101";
53        wait for 100ns;
54        -- 1234
55        tb_in_num <= "0001001000110100";
56        wait for 100ns;
57        -- 9876
58        tb_in_num <= "1001100001110110";
59        wait for 100ns;
60        -- 0005
61        tb_in_num <= "0000000000000101";
62        wait for 100ns;
63        -- 0061
64        tb_in_num <= "0000000001100001";
65        wait for 100ns;
66        -- 0302
67        tb_in_num <= "0000001100000010";
68        wait for 100ns;
69    end process;
```

   Modify attached "./testbench/**tb_four_digit_sevenseg.vhd**" in unzipped attachments.

   If you want, modify this file to do self-testbench.

   (This program just shows 1000ns, so you can do 1000ns simulation)

5. We will start performing a simulation. Put your **"four_digit_sevenseg.vhd"** and **your VHDL files** into "./submission/".

   Make sure "**four_digit_sevenseg" folder does not exist** and only **"four_digit_sevenseg.zip"** exist in execution directory.

   Run **"simulator.exe"** for auto simulation. (It will take about 1min.)

   Result file(.vcd) will be generated as **"./submission/tb_four_digit_sevenseg.vcd".**

   (If you want, copy this file to directory you want.)

6. Finally, Run **"visualizer.exe"**.

   Import copied result file (.vcd).

   Check your testbench simulation results 😊.

   (Tip: Our automatic grader is based on these programs, so check your own testbenches.)