# Experiment 2 in Electrical engineering, ET061G

by Khaled Hamza

Mittuniversitetet

MID SWEDEN UNIVERSITY

## Purpose

The goal of this experiment is to gain experience with VHDL, enabling the programming of an FPGA board using a high-level language, rather than relying on transistors and breadboards to build circuits. Additionally, the experiment aims to develop the ability to create two different modules with the same functionality and compare the differences between the two approaches.

## Theory

The experiment begins with the implementation of a simple 1-bit adder. This adder takes three input values: A and B, which are the numbers to be added, and carry in, which is used when working with multi-bit adders. The adder produces two outputs: sum, representing the result of adding A and B, and carry out, which becomes 1 if the sum equals 2. In this case, sum will be 0; otherwise, carry out remains 0. After writing the VHDL code, the functionality of the adder can be tested on an FPGA board. This requires creating a constraints file in Vivado software, which maps the input values to specific switches on the board and the output values to LEDs. Using this setup, it is also possible to test the carry in input and confirm that the adder behaves as expected.

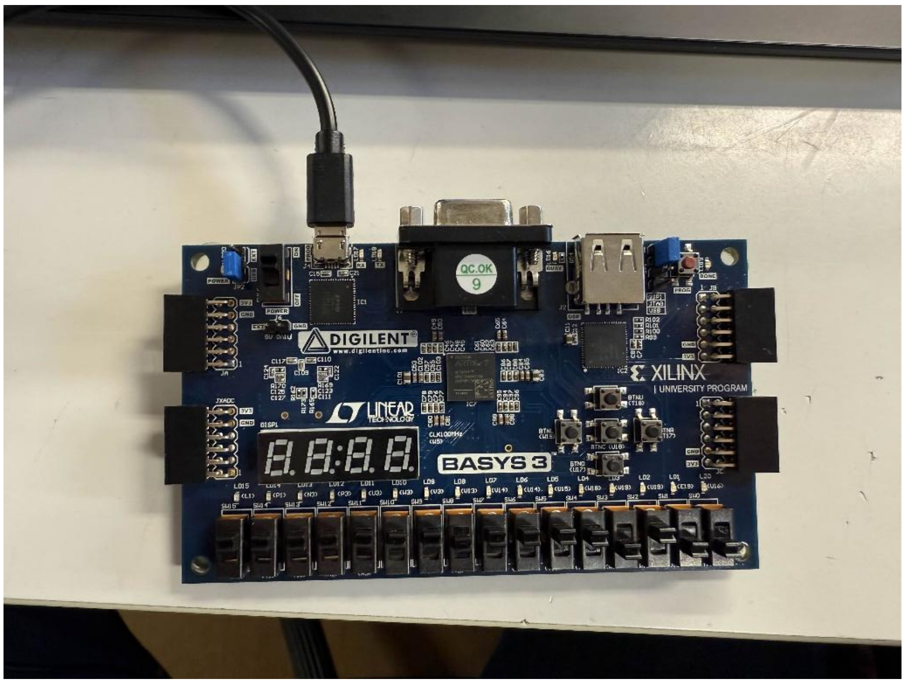


*Figure 1: An image of how an FPGA board looks like.*

## Implementation:

The implementation of a 1-bit adder in Vivado software can be achieved using basic logical operators. The **sum** output is determined by applying the XOR operator to the inputs **A**, **B**, and **carry in**. XOR produces a result of **1** if an odd number of the inputs are **1**; otherwise, the result is **0**. The **carry out** output is **1** if any two of the inputs are **1**; otherwise, it is **0**. This can be implemented using the AND operator for each pair of inputs (e.g., **A AND B**) and combining the results with the OR operator (e.g., **(A AND B) OR (A AND carry in) OR (B AND carry in)**).

As mentioned earlier, a 1-bit adder is a fundamental building block for constructing multi-bit adders. However, there are other ways to implement multi-bit adders without relying on individual 1-bit adders. For example, an 8-bit adder can be created using two different approaches. One common approach is to use two input vectors, each containing 8 bits, and process each corresponding pair of bits using a 1-bit adder. The **carry out** from each adder is passed as the **carry in** to the next adder in the sequence. For the first adder, **carry in** is set to **0** since there is no previous adder. The VHDL code used for this implementation is provided at the end of this document.

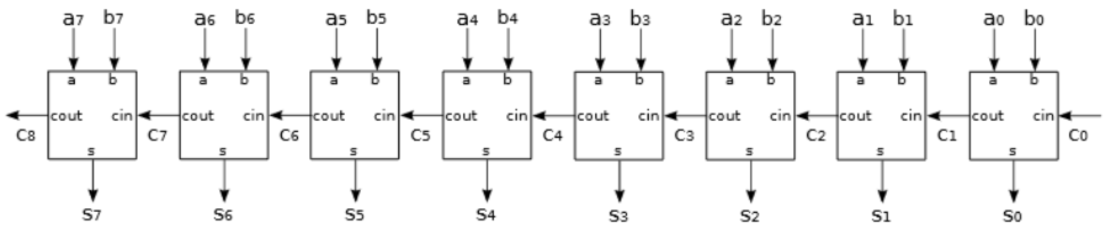
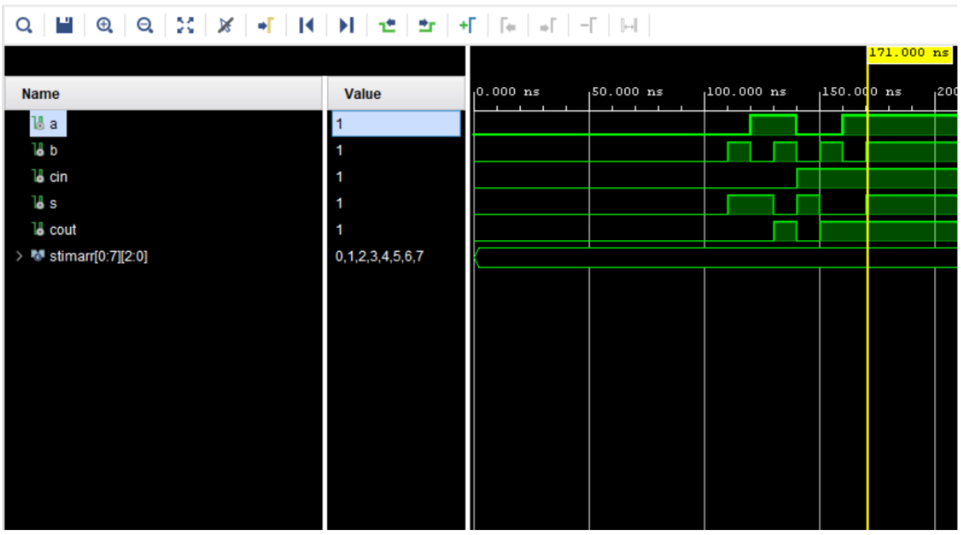


*Figure 2: A visualization of an 8-bit adder.*

Another approach to implementing an 8-bit adder is to simply use the arithmetic (**+)** operator. This method is generally preferred when working with multi-bit adders, as it offers better performance and simplifies the design process. However, manually constructing bit-level adders remains a valuable exercise for understanding the underlying hardware and how arithmetic operations are performed at a lower level.

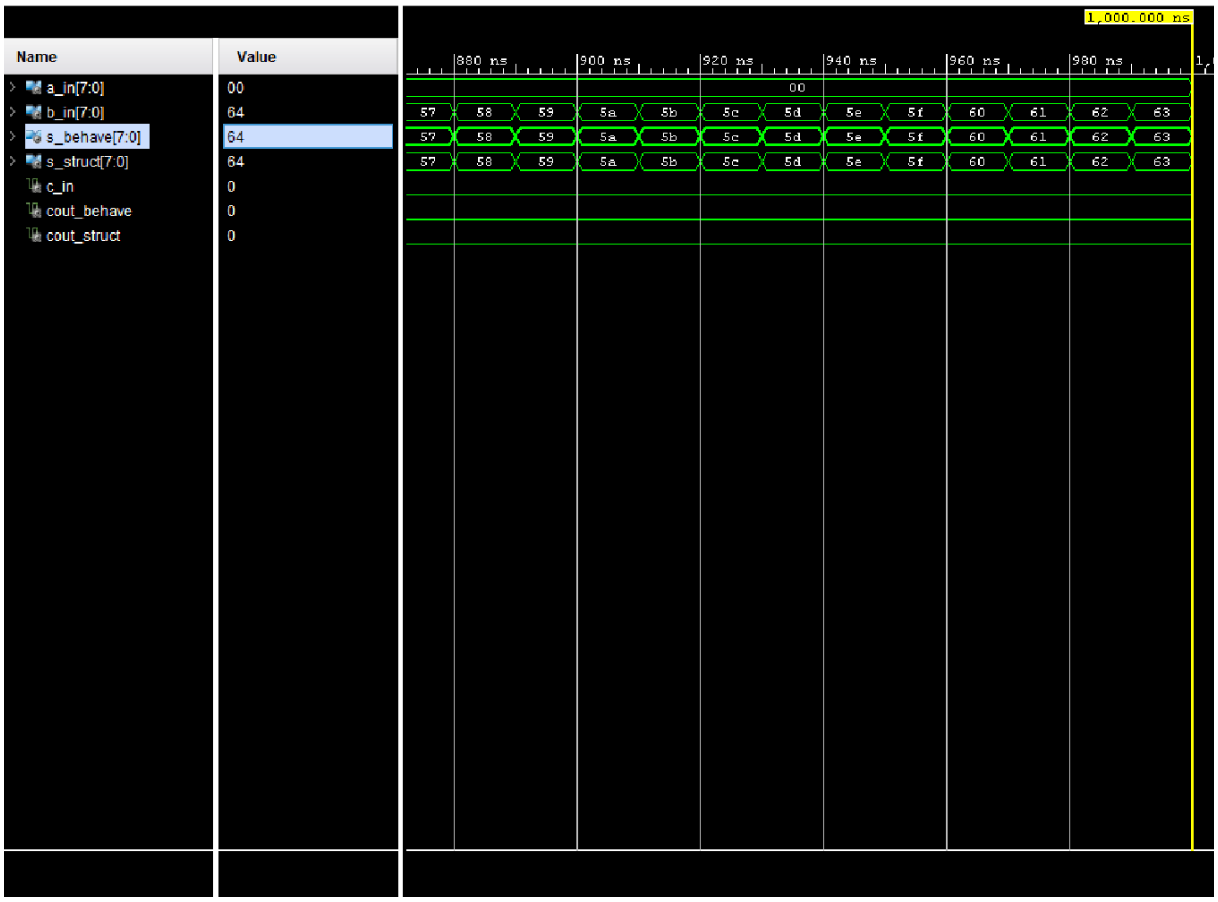
## Results and Discussions:



| Name | Value |
|---|---|
| a | 1 |
| b | 1 |
| cin | 1 |
| s | 1 |
| cout | 1 |
| stimarr[0:7][2:0] | 0,1,2,3,4,5,6,7 |

*Figure 3: Simulation result of the full adder shown in hexadecimals.*

After implementing the 8-bit adder, its functionality can be verified by simulating it using the testbench provided on the course webpage.



```
Name              Value
a_in[7:0]         00
b_in[7:0]         64
s_behave[7:0]     64
s_struct[7:0]     64
c_in              0
cout_behave       0
cout_struct       0
```

1,000.000 ns
880 ns
900 ns
920 ns
940 ns
960 ns
980 ns
00
57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63
57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63
57 58 59 5a 5b 5c 5d 5e 5f 60 61 62 63

*Figure 4: Result of the 8-bit adder shown in hexadecimals*

As shown in the image above, at the marked timestamp, **a_in**, which represents the first input value, is **24** in hexadecimal (equal to **36** in decimal), and **b_in**, the second input, is **E0** in hexadecimal (equal to **224** in decimal). The outputs **s_behave** and **s_struct** are the same, even though they were calculated using two different methods. Both outputs hold the value **4** because the sum of **a_in** and **b_in** exceeds **255**. Since the **sum** variables are only 8 bits wide, the result wraps around, causing an overflow. The actual sum is **260**, but because it exceeds the 8-bit limit, the result becomes **260 - 256 = 4**. As a result, the **cout** variables hold the value **1**, indicating an overflow, which represents **256** in decimal or **100** in binary.

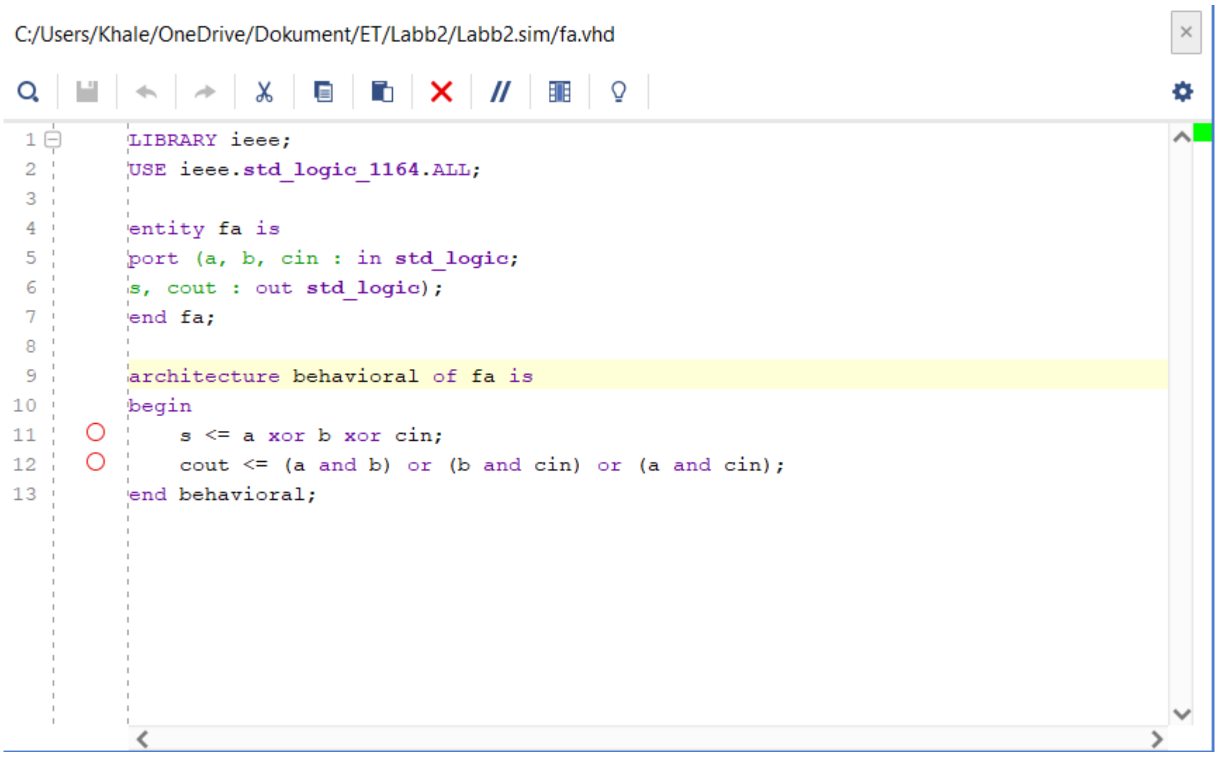
## References:

https://www.xilinx.com/support/download.html

https://docs.xilinx.com/v/u/2019.2-English/ug901-vivado-synthesis

## Attachments:

An implementation of a full adder:

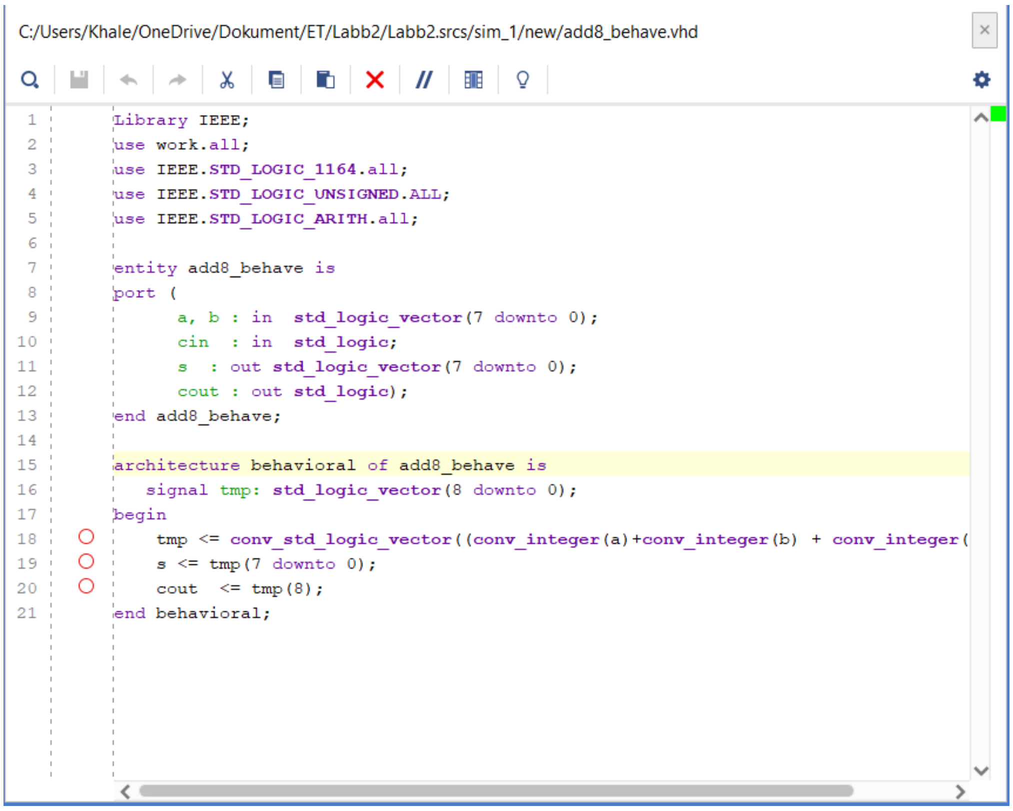

C:/Users/Khale/OneDrive/Dokument/ET/Labb2/Labb2.sim/fa.vhd

```
LIBRARY ieee;
USE ieee.std_logic_1164.ALL;

entity fa is
port (a, b, cin : in std_logic;
s, cout : out std_logic);
end fa;

architecture behavioral of fa is
begin
    s <= a xor b xor cin;
    cout <= (a and b) or (b and cin) or (a and cin);
end behavioral;
```

An implementation of an 8-bit adder:



C:/Users/Khale/OneDrive/Dokument/ET/Labb2/Labb2.srcs/sim_1/new/add8_behave.vhd

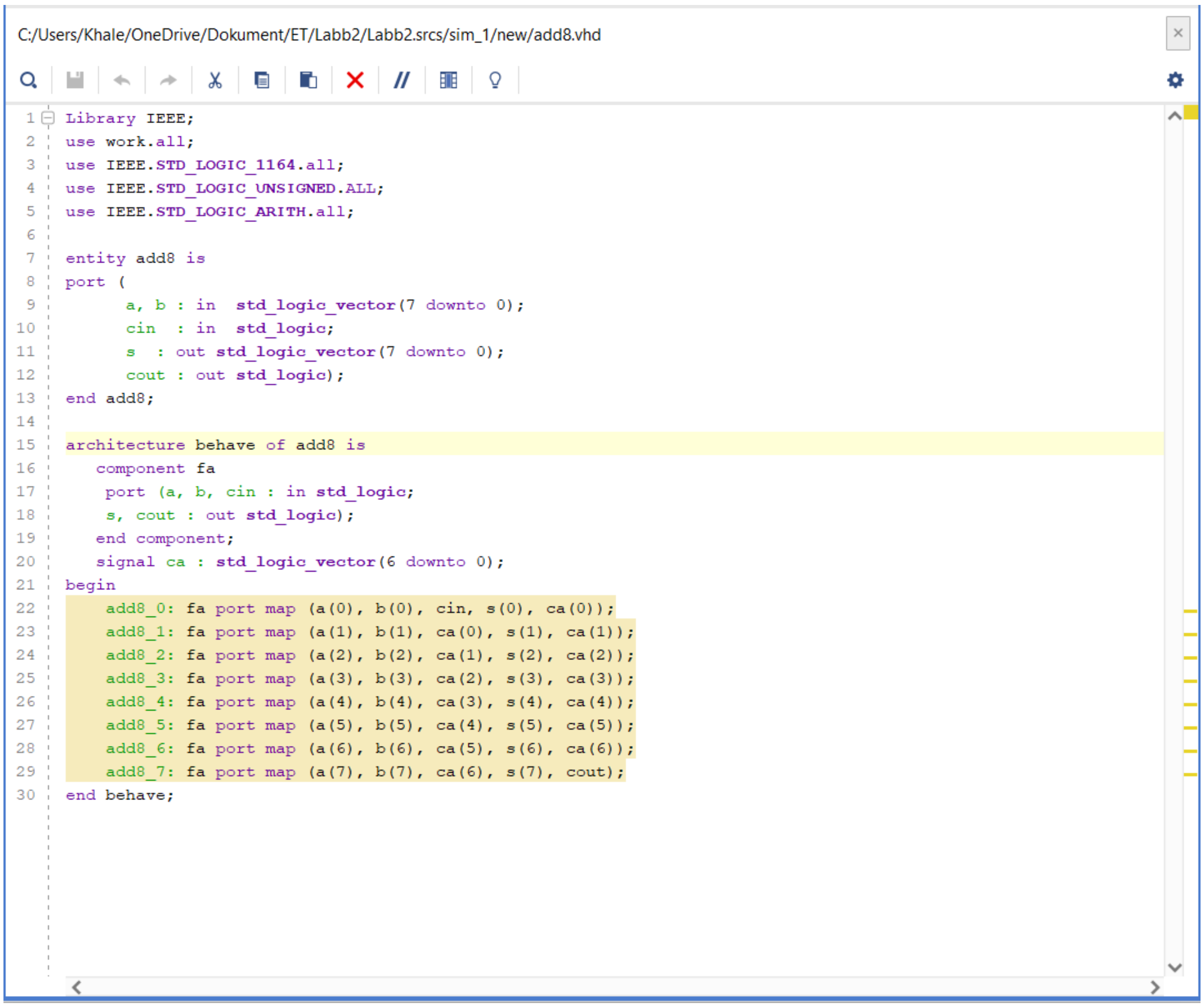
```
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.all;

entity add8_behave is
port (
        a, b : in  std_logic_vector(7 downto 0);
        cin  : in  std_logic;
        s   : out std_logic_vector(7 downto 0);
        cout : out std_logic);
end add8_behave;

architecture behavioral of add8_behave is
    signal tmp: std_logic_vector(8 downto 0);
begin
    tmp <= conv_std_logic_vector((conv_integer(a)+conv_integer(b) + conv_integer(
    s <= tmp(7 downto 0);
    cout  <= tmp(8);
end behavioral;
```

An implementation of an 8-bit adder using a full adder:

Khaled Hamza ET061G
2024-02-13


C:/Users/Khale/OneDrive/Dokument/ET/Labb2/Labb2.srcs/sim_1/new/add8.vhd

```vhdl
Library IEEE;
use work.all;
use IEEE.STD_LOGIC_1164.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use IEEE.STD_LOGIC_ARITH.all;

entity add8 is
port (
      a, b : in  std_logic_vector(7 downto 0);
      cin  : in  std_logic;
      s   : out std_logic_vector(7 downto 0);
      cout : out std_logic);
end add8;

architecture behave of add8 is
   component fa
    port (a, b, cin : in std_logic;
    s, cout : out std_logic);
   end component;
   signal ca : std_logic_vector(6 downto 0);
begin
    add8_0: fa port map (a(0), b(0), cin, s(0), ca(0));
    add8_1: fa port map (a(1), b(1), ca(0), s(1), ca(1));
    add8_2: fa port map (a(2), b(2), ca(1), s(2), ca(2));
    add8_3: fa port map (a(3), b(3), ca(2), s(3), ca(3));
    add8_4: fa port map (a(4), b(4), ca(3), s(4), ca(4));
    add8_5: fa port map (a(5), b(5), ca(4), s(5), ca(5));
    add8_6: fa port map (a(6), b(6), ca(5), s(6), ca(6));
    add8_7: fa port map (a(7), b(7), ca(6), s(7), cout);
end behave;
```