Bu-Ali Sina University

“ ALU ”

Digital Logic Circuit

Final Project

Dr.Abdoli

Fall 2024

Students : Sahba Mirshahi , Mohammad Matin Soleimani

Student number : 40212358040 40212358021

Design Functions And Their Test Bench

for each function we have input , output and flags (zero , carry , borrow ...)

here is the description of each module and the explanation :

Addition : add two input numbers and produce a result .

the overall view of the module is as follows :

```
ENTITY Generic_Adder is
   GENERIC ( size : INTEGER := 4  );
   Port (
      A  : IN  STD_LOGIC_VECTOR (size-1 downto 0);
      B  : IN STD_LOGIC_VECTOR (size-1 downto 0);
      Cin : IN  STD_LOGIC ;
      Sum : OUT STD_LOGIC_VECTOR (size-1 downto 0);
      Zero_flag : OUT STD_LOGIC ;
      Carry_flag : OUT STD_LOGIC ;
      Borrow_flag : OUT STD_LOGIC ;
      Overflow_flag : OUT STD_LOGIC ;
      Greater_flag : OUT STD_LOGIC ;
      Equal_flag : OUT STD_LOGIC
   );
end Generic_Adder ;
```

in architecture behavioral of adder we have two signals :

```
SIGNAL temp_sum : STD_LOGIC_VECTOR  (size-1 downto 0);
SIGNAL Cin_unsigned : unsigned (size-1 downto 0);
```

The first signal is created to temporarily compute the general calculations and then assign it to the final answer .

The second signal is for converting carry_in to n_bit vector .

we calculate the result in this way :

```
temp_sum <= STD_LOGIC_VECTOR (unsigned(A) + unsigned(B) + Cin_unsigned);
Sum <= temp_sum ;
```

we have no borrow flag so set it to zero .

and calculate carry flag and overflow flag in this way :

```
Overflow_flag <= (A(size-1) AND B(size-1)) OR (A(size-2) AND B(size-2));
Carry_flag <= (A(size-1) AND B(size-1)) OR (A(size-2) AND B(size-2));
```

Finally , we use the NUMERIC library to compare A and B together and determine greater_than and equality flag .

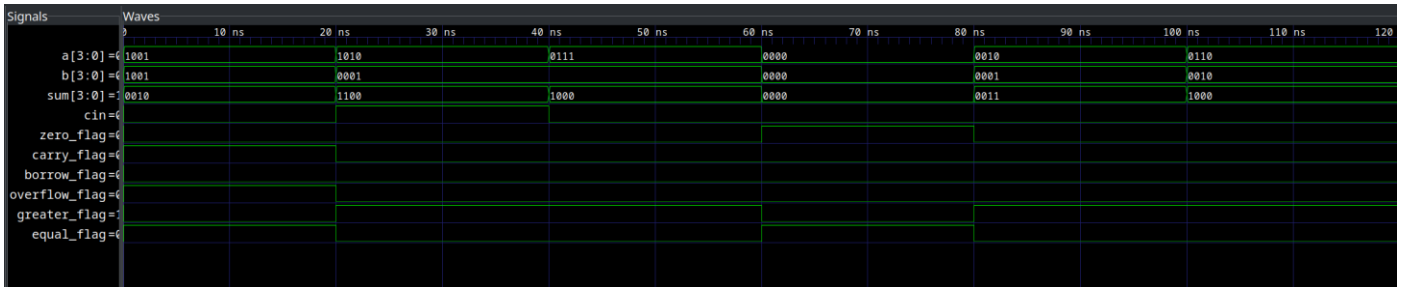
TestBench : Adder

First of all , we should create a signal for each input and output in our module .

to have an easier compile , we write the COMPONENT of the module in the test bench .

After writing the Generic Port and Port Map , we begin the stimulation process . We give A and B desired quantities .

Waveform :



Subtraction : subtract one input from the other and produce a result .

the overall view of the module is as follows :

```
ENTITY Generic_Subtractor IS
   GENERIC ( size : integer := 4 );
   PORT (
      A , B : IN STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
      Bin : IN STD_LOGIC ;
      Diff : OUT STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
      Zero_flag : OUT STD_LOGIC ;
      Carry_flag : OUT STD_LOGIC ;
      Borrow_flag : OUT STD_LOGIC ;
      Overflow_flag : OUT STD_LOGIC ;
      Greater_flag : OUT STD_LOGIC ;
      Equal_flag : OUT STD_LOGIC
   );
END Generic_Subtractor ;
```

in architecture behavioral of adder we have one signal :

```
SIGNAL tempSub : STD_LOGIC_VECTOR ( size DOWNTO 0 );
```

The signal is created to temporarily compute the general calculations and then assign it to the final answer .

we calculate the result in this way :

```
tempSub <= STD_LOGIC_VECTOR
(unsigned ('0' & A) - unsigned ('0' & B) - unsigned ('0' & std_logic_vector'(0 => Bin)));
Diff <= tempSub (size - 1 downto 0);
```

The signal is a (n+1) bit vector , so we should convert all inputs and the carry_in into (n+1) bit vectors . For doing this , we should use AND between the input and ‘0’ .

The amount of flags calculate as follows :

Carry_flag <= NOT tempSub (size);

Borrow_flag <= tempSub (size);

Overflow_flag <= (A(size-1) XOR B(size-1)) AND (A(size-1) XOR tempSub(size-1));

Finally , we use the NUMERIC library to compare A and B together and determine greater_than and equality flag .

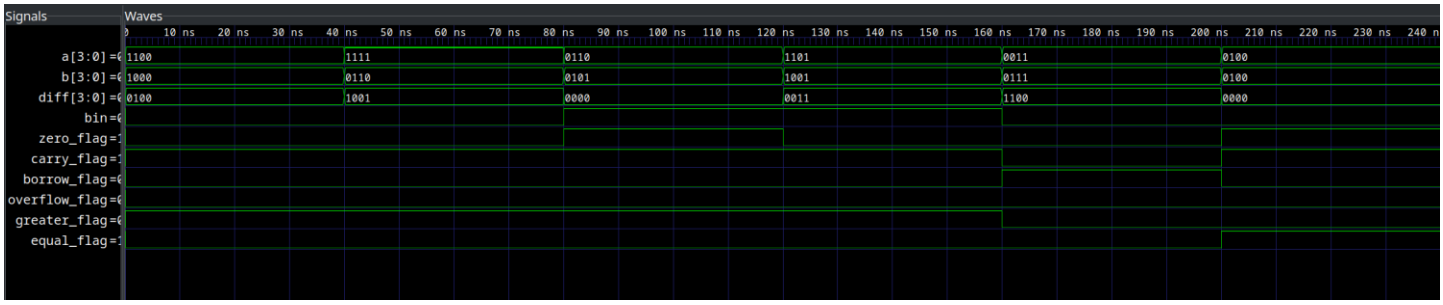
TestBench : Subtractor

First of all , we should create a signal for each input and output in our module .

to have an easier compile , we write the COMPONENT of the module in the test bench .

After writing the Generic Port and Port Map , we begin the stimulation process . We give A and B desired quantities .

Waveform :



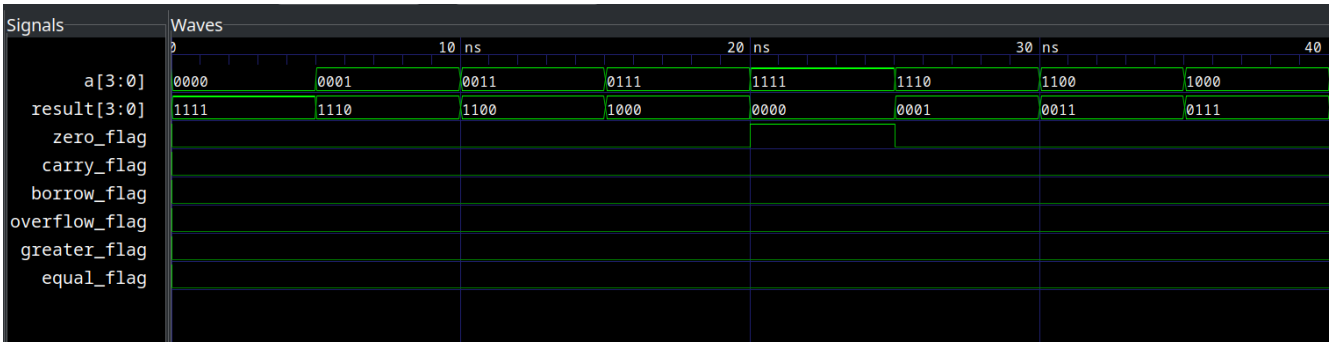
NOT : perform a bitwise NOT operation on one input .

the overall view of the module is as follows :

```
ENTITY generic_not IS
    GENERIC ( size : INTEGER := 4 );
    PORT ( A : IN STD_LOGIC_VECTOR (size-1 DOWNTO 0);
          Result : BUFFER STD_LOGIC_VECTOR (size-1 DOWNTO 0);
          Zero_flag : OUT STD_LOGIC ;
          Carry_flag : OUT STD_LOGIC ;
          Borrow_flag : OUT STD_LOGIC ;
          Overflow_flag : OUT STD_LOGIC ;
          Greater_flag : OUT STD_LOGIC ;
          Equal_flag : OUT STD_LOGIC
    );
END generic_not ;
```

In architecture behavior we have a process to NOT the input A . All flags are zero (they are not applicable in NOT operation) except the Zero_flag .

Waveform :



AND : perform a bitwise AND operation on two inputs.

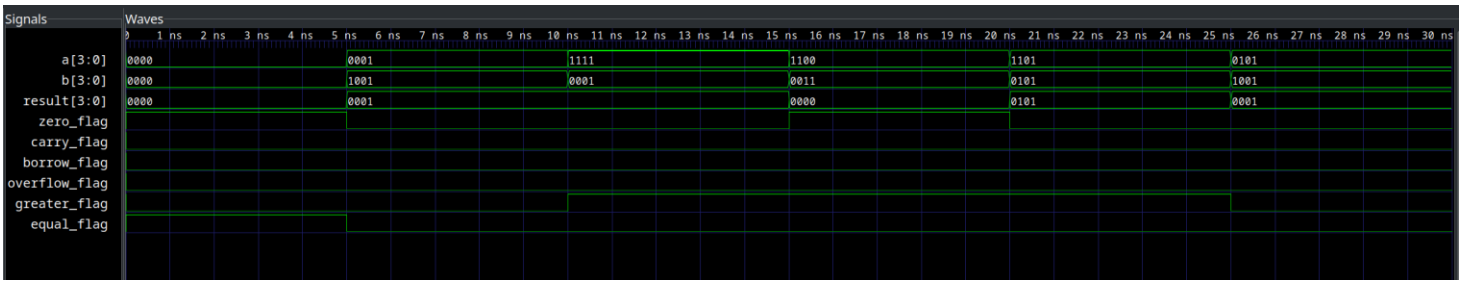
the overall view of the module is as follows :

```
ENTITY GENERIC_AND IS
    GENERIC (size: INTEGER := 4);
    PORT (
        A : IN STD_LOGIC_VECTOR (size-1 downto 0);
        B : IN STD_LOGIC_VECTOR (size-1 downto 0);
        Result : BUFFER STD_LOGIC_VECTOR (size-1 downto 0);
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC
    );
END GENERIC_AND ;
```

In architecture behavior we have a process to AND the input A . Some flags are zero (they are not applicable in AND operation) including carry , borrow and overflow .

We check other flags like before .

Waveform :



OR : perform a bitwise OR operation on two inputs .

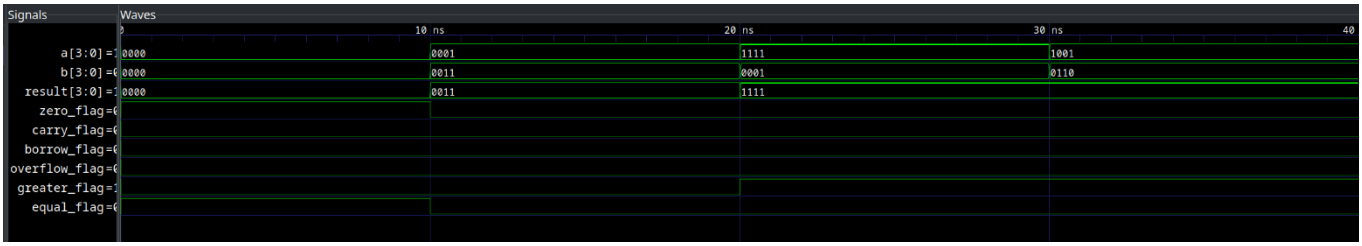
the overall view of the module is as follows :

```
ENTITY GENERIC_OR IS
    GENERIC ( size: INTEGER := 4 );
    PORT (
        A : IN STD_LOGIC_VECTOR (size-1 downto 0);
        B : IN STD_LOGIC_VECTOR (size-1 downto 0);
        Result : BUFFER STD_LOGIC_VECTOR (size-1 downto 0);
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC
    );
END GENERIC_OR;
```

In architecture behavior we have a process to OR the input A . Some flags are zero (they are not applicable in OR operation) including carry , borrow and overflow .

We check other flags like before .

Waveform :



XOR : perform a bitwise XOR operation on two inputs .

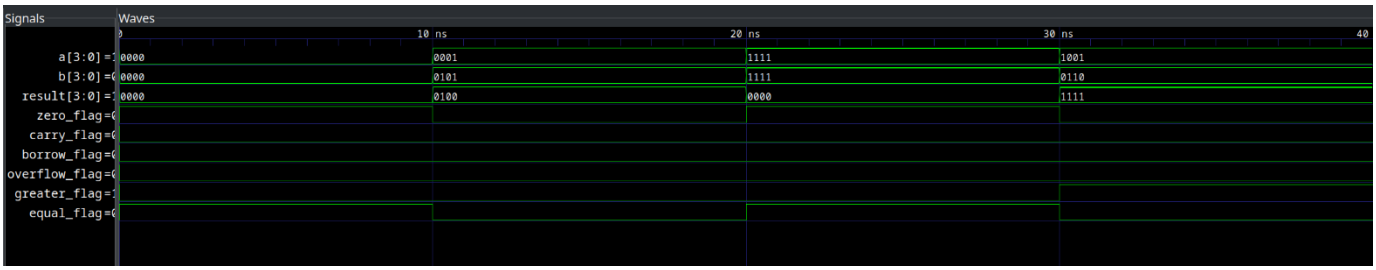
the overall view of the module is as follows :

```
ENTITY GENERIC_XOR IS
    GENERIC ( size: INTEGER := 4 );
    PORT (
        A : IN STD_LOGIC_VECTOR (size-1 downto 0);
        B : IN STD_LOGIC_VECTOR (size-1 downto 0);
        Result : BUFFER STD_LOGIC_VECTOR (size-1 downto 0);
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC
    );
END GENERIC_XOR ;
```

In architecture behavior we have a process to XOR the input A . Some flags are zero (they are not applicable in XOR operation) including carry , borrow and overflow .

We check other flags like before .

Waveform :



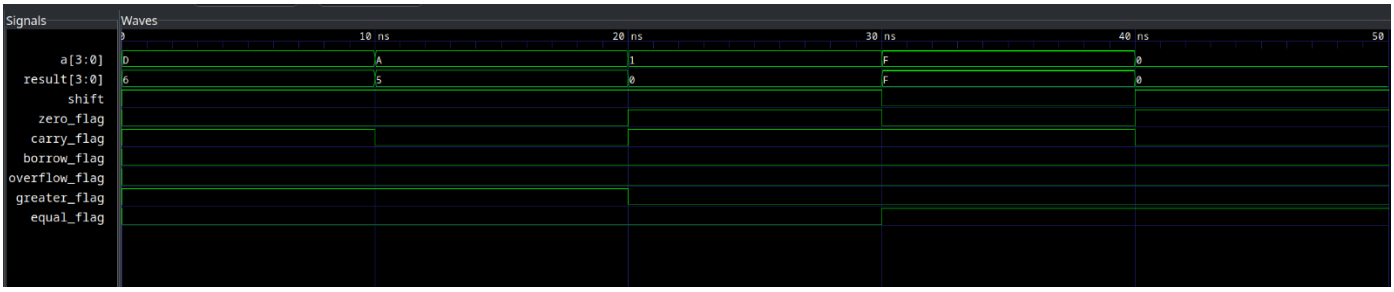
Right Shift : Shift the input bits to the right .

the overall view of the module is as follows :

```
ENTITY Right_Shift is
    GENERIC (
        size : integer := 4
    );
    PORT (
        A : IN  STD_LOGIC_VECTOR (size-1 DOWNTO 0);
        Shift : IN  STD_LOGIC ;
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC ;
        Result : OUT  STD_LOGIC_VECTOR (size-1 DOWNTO 0)
    );
END Right_Shift ;
```

In architecture behavior we have a process to do the right shift and another process to determine the amount of flags .

Waveform :



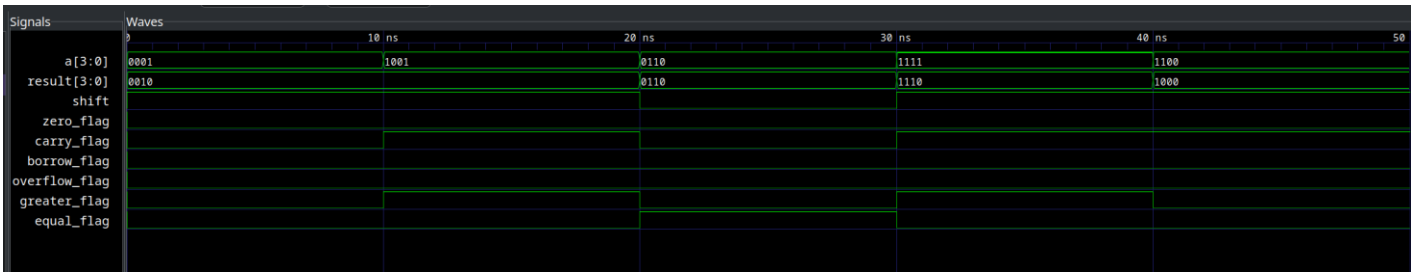
Left Shift : Shift the input bits to the left .

the overall view of the module is as follows :

```
ENTITY Left_Shift is
    GENERIC (
        size : integer := 4
    );
    PORT (
        A : IN  STD_LOGIC_VECTOR (size-1 DOWNTO 0);
        Shift : IN  STD_LOGIC ;
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC ;
        Result : OUT  STD_LOGIC_VECTOR (size-1 DOWNTO 0)
    );
END Left_Shift ;
```

In architecture behavior we have a process to do the left shift and another process to determine the amount of flags .

Waveform :



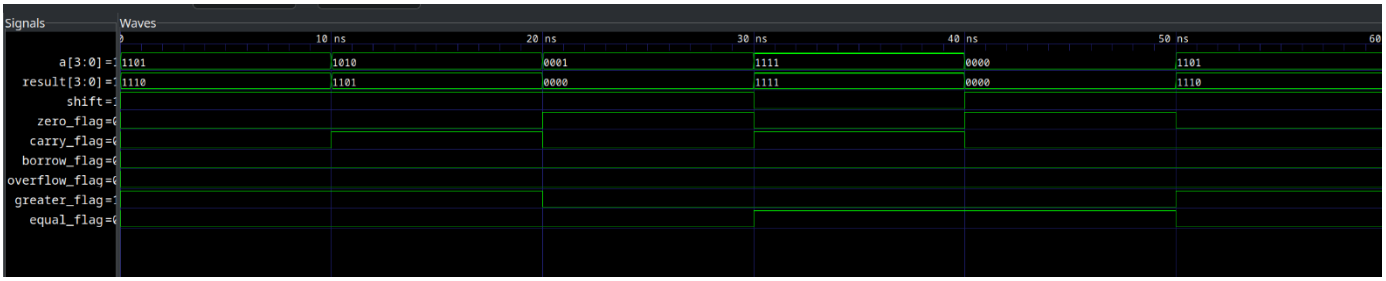
Arithmetic right shift : Shift the input bits to the right , retaining the sign bit for signed operations .

the overall view of the module is as follows :

```
ENTITY Arithmetic_Right_Shift is
    GENERIC (
        size : integer := 4
    );
    PORT (
        A : IN  STD_LOGIC_VECTOR (size-1 DOWNTO 0);
        Shift : IN  STD_LOGIC ;
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC ;
        Result : OUT  STD_LOGIC_VECTOR (size-1 DOWNTO 0)
    );
END Arithmetic_Right_Shift ;
```

In architecture behavior we have a process to do the left shift and another process to determine the amount of flags .

Waveform :



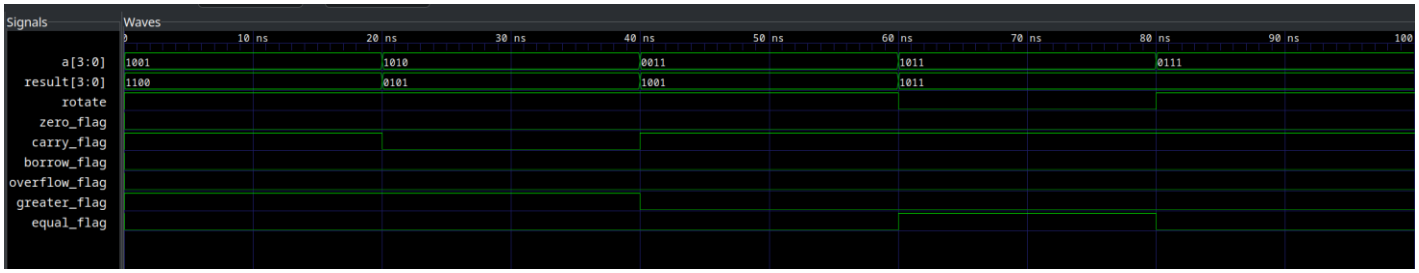
Rotate left : Rotate the bits of the input to the left .

the overall view of the module is as follows :

```
ENTITY Right_Rotate IS
   GENERIC ( size : INTEGER := 4 );
   PORT (
      Rotate : IN STD_LOGIC ;
      A : IN STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
      Result : OUT STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
      Zero_flag : OUT STD_LOGIC ;
      Carry_flag : OUT STD_LOGIC ;
      Borrow_flag : OUT STD_LOGIC ;
      Overflow_flag : OUT STD_LOGIC ;
      Greater_flag : OUT STD_LOGIC ;
      Equal_flag : OUT STD_LOGIC
   );
END Right_Rotate ;
```

In architecture behavior we have a process to do the right rotate and another process to determine the amount of flags . we use a signal to save the temp result and assign it to result at the end .

Waveform :



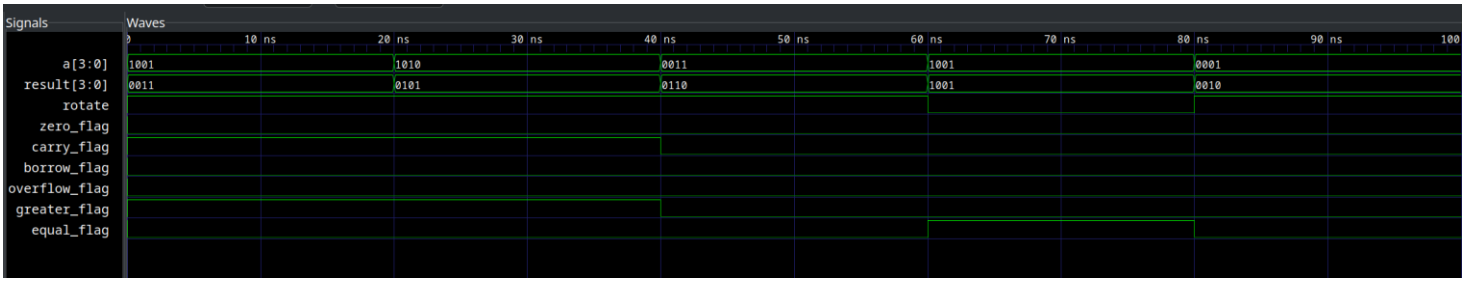
Rotate left : Rotate the bits of the input to the left .

the overall view of the module is as follows :

```
ENTITY Left_Rotate IS
    GENERIC ( size : INTEGER := 4 );
    PORT (
        Rotate : IN STD_LOGIC ;
        A : IN STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
        Result : OUT STD_LOGIC_VECTOR ( size - 1 DOWNTO 0 );
        Zero_flag : OUT STD_LOGIC ;
        Carry_flag : OUT STD_LOGIC ;
        Borrow_flag : OUT STD_LOGIC ;
        Overflow_flag : OUT STD_LOGIC ;
        Greater_flag : OUT STD_LOGIC ;
        Equal_flag : OUT STD_LOGIC
    );
END Left_Rotate ;
```

In architecture behavior we have a process to do the right rotate and another process to determine the amount of flags . we use a signal to save the temp result and assign it to result at the end .

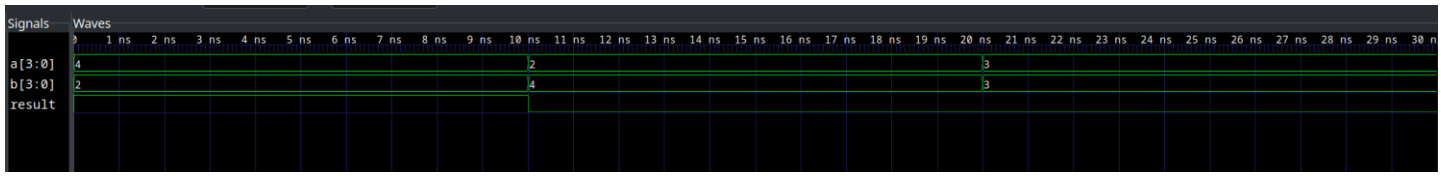
Waveform :



for these two modules we just have two inputs (A and B) and a result .

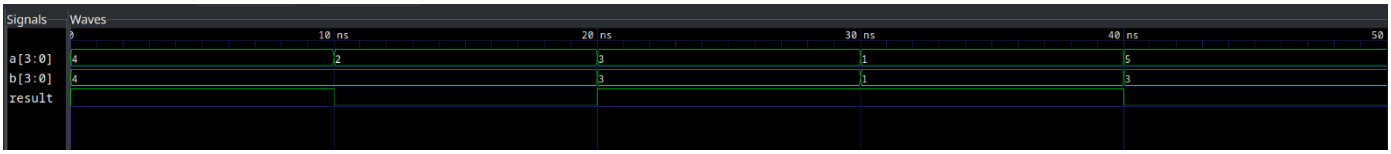
Comparison : Output 1 if input A is greater than input B ; otherwise , output 0 .

Waveform :



Equality check : Output 1 if input A is equal to input B ; otherwise , output 0 .

Waveform :



Test benches : in all test benches we define a COMPONENT . make an instance and port it to our desired module .

Control Unit

in this ENTITY we control all operations . we have an input named Opcode that shows the operation number .

for example if Opcode = “0001” the control unit do addition between the inputs A and B and so on …

first of all we should write all COMPONENTS that we designed before . after that , we make 13 instances for each operation (Add , Sub , And …) and port them to their module .

Test Bench :

In the test bench we write one test case for each operation .