INTRO TO PROCESSOR ARCHITECTURE

ASSIGNMENT – 1

1.BUILD AN ALU

2. BUILD MEMORY MODULE

ANIRUTH SURESH

2022102055

SAKTHIDAR PV

2022112005

1 BUILD AN ALU

OBJECTIVE :

To build a ALU unit with the following functionality :

1. ADD – 64 bits
2. SUB – 64 bits
3. AND – 64 bits
4. XOR – 64 bits

To construct a final wrapper ALU unit from where we can call the modules based on the control input .

Control 0 - ADD x and y

Control 1 – Subtract y from x

Control 2 – AND x and y

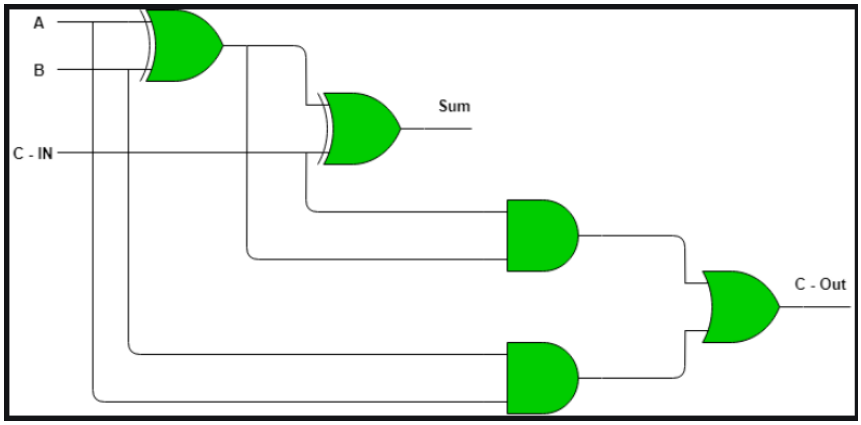
Control 3 – XOR x and y

BLOCKS :

1 ADDER BLOCK

The adder block consists of full adder module which keeps getting called for 64 times (both the inputs are 64 bits)

The following is the circuit diagram for full adder :



I/O FORMAT:

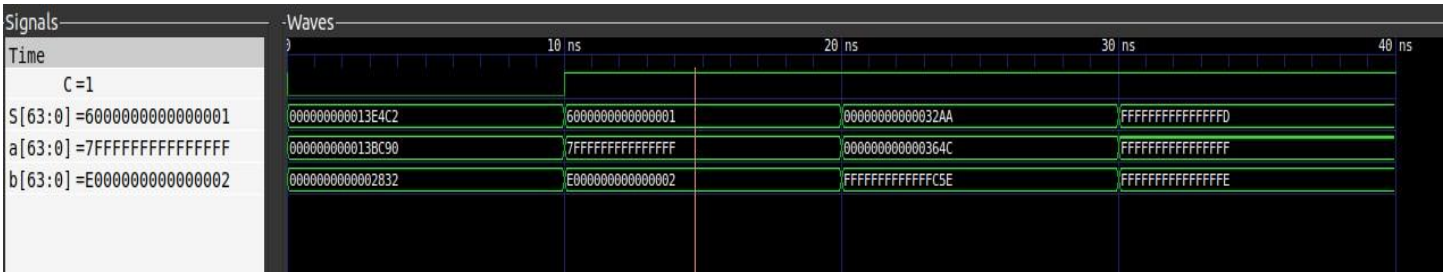
Input takes in two 64-bit binary numbers a and b . The adder will compute and the sum and carry is returned .

TESTBENCH :

The inputs in the testbenches can be given as decimal / binary numbers . The following cases were given to cover all possible combinations :

1. Both positive (no carry)
2. Both positive (carry)
3. Positive and negative (No carry possible)
4. Negative and negative (Always carry)

GTKWAVE :



OBSERVATION :

The carry bit is produced whenever the result of the addition operation is a number too big to be represented by 64 bits .

2 SUBTRACTOR BLOCK

The subtracter block uses the concept of 2's complement of a number and the adder block to perform subtraction of 2 64-bit numbers.

The 2's complement of the second input is generated which is then added to the first input which results in Inp1 – Inp2 .

The way we find the 2's complement of the input is by taking the flipping every bit by running a for loop and then using the adder to add 1 to the complemented number which results in 2's complement of a number

I/O FORMAT :

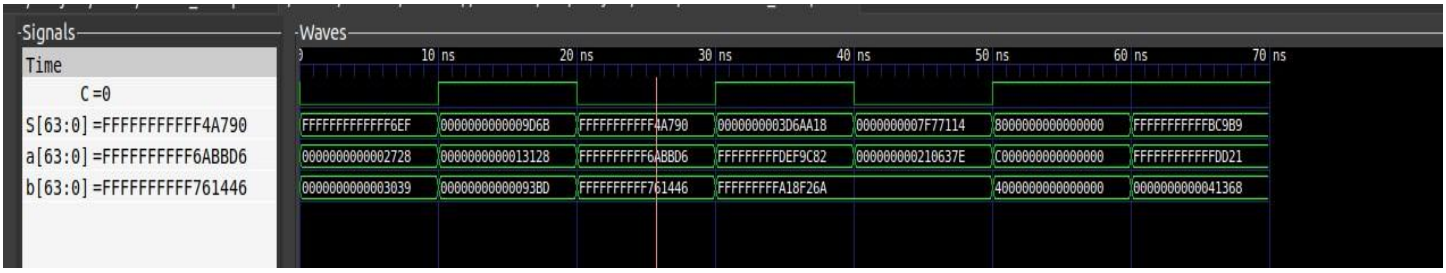
Input two 64-bit binary numbers a and b. The subtracter will output the result after performing ' a - b ' and the overflow bit .

TESTBENCH :

The inputs in the testbenches can be given as decimal / binary numbers . The following cases were given to cover all possible combinations :

1. Both positive (no carry (a<b))
2. Both positive (carry (a>b))
3. Negative and Negative (No Carry -> result = negative)
4. Negative and Negative (Carry -> result = positive)
5. Positive and Negative (No overflow -> no carry)
6. Positive and Negative (Overflow -> carry)
7. Negative and Positive (Carry -> result = negative)

GTKWAVE :



3 AND BLOCK

The and block performs bitwise and for 2 64-bit inputs. This is done by calling the and function for the corresponding bits by running a loop through every bit and performing AND on them one by one to achieve the output.

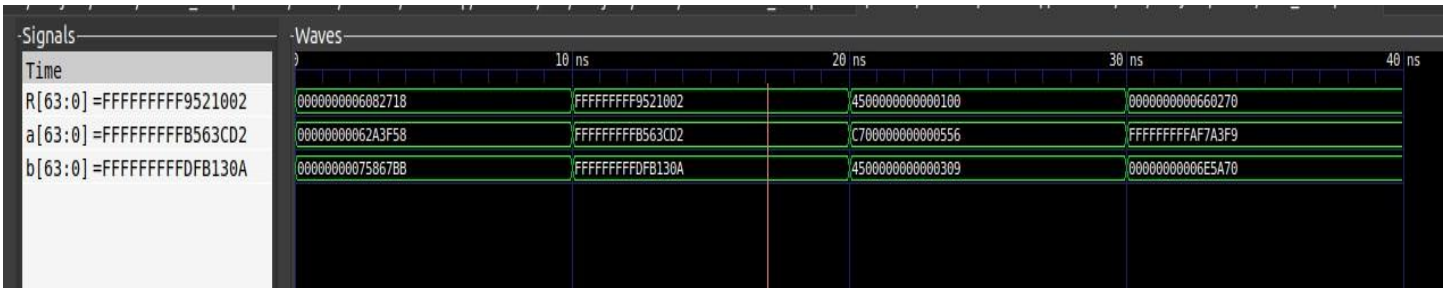
I/O FORMAT :

Input two 64-bit binary numbers a and b. The and block will output the result after performing ‘ a & b ’

TESTBENCH :

The inputs in the testbenches can be given as decimal / binary numbers . The test cases were given so as to cover all possible combinations .

GTKWAVE :



4 XOR BLOCK

The XOR block performs bitwise XOR for 2 64-bit inputs. This is done by calling the xor function for the corresponding bits by running a loop through every bit and performing XOR on them one by one to achieve the output.

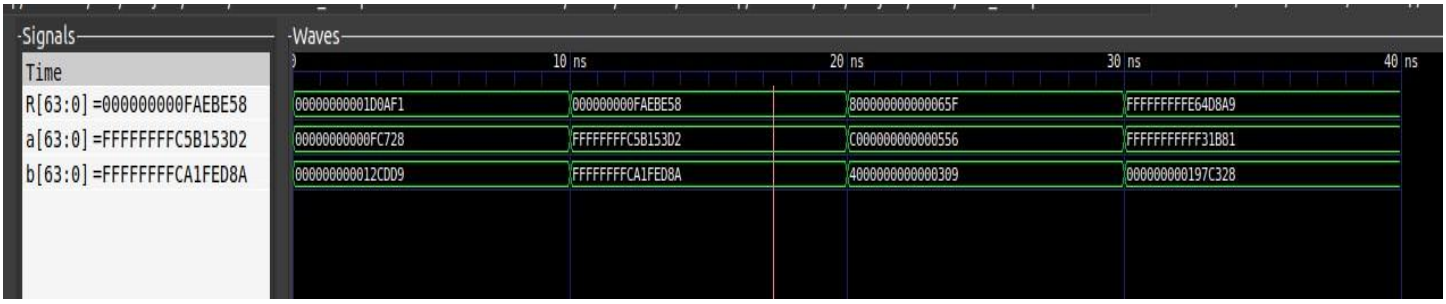
I/O FORMAT :

Input two 64-bit binary numbers a and b. The and block will output the result after performing ‘ a ^ b ’

TESTBENCH :

The inputs in the testbenches can be given as decimal / binary numbers . The test cases were given so as to cover all possible combinations

GTKWAVE PLOT :



THE ALU (WRAPPER) BLOCK

The ALU block is a module that will take the input of a control signal and chooses the operation to be performed based on the control input . (Control cases)

IMPLEMENTATION :

```
 8
 9   input [63:0] a,b;
10   input [1:0] control;
11
12
13   output reg [63:0] result;
14   wire [63:0] Sum , Diff , and_op , xor_op;
15   wire sum_carry , borrow_carry;
16   output reg carry;
17
18   add_64_bit add(a,b,Sum,sum_carry);
19   sub_64_bit sub(a,b,Diff,borrow_carry);
20   and_64_bits and_alu(a,b,and_op);
21   xor_64_bits xor_alu(a,b,xor_op);
22
23   always@(*) begin
24
25       case(control)
26
27       2'b00:  //adder call
28       begin
29       result <= Sum;
30       carry <= sum_carry;
31       end
32
33       2'b01:  //sub call
34       begin
35       result <= Diff;
36       carry <= borrow_carry;
37       end
38
39       2'b10:  //and call
40       begin
41       result <= and_op;
42       carry <= 0;
43       end
44
45       2'b11:  //xor call
46       begin
47       result <= xor_op;
48       carry  <= 0;
49       end
50
51
52       endcase
53
54
55   end
56
57
58   endmodule
```

GTKWAVE :

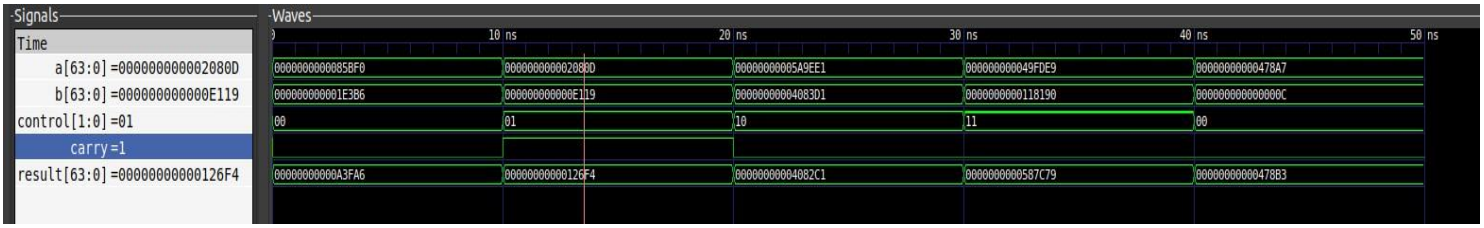
Various control signals along with inputs are given and the output is observed using GTKwave .

Control 0 - ADD x and y

Control 1 – Subtract y from x

Control 2 – AND x and y

Control 3 – XOR x and y



2 BUILD MEMORY MODULE

We need to check the validity of the memory address (64 – bit) and based on that :

1. Read from the memory
2. Write from the memory

Memory is basically a **1024 – (64-bit)** registers .

IMPLEMENTATION :

1) If the read enable = 1 and write enable = 1 , as we can't access both read and write at the same time , it will result in memory access error and sets dmem_err = 1.

2) If the read enable = 0 and write enable = 0 , we just pass it and sets dmem_err = 1 .

3) If only one of read or write enable in ON , then it performs the corresponding operation and sets dmem_err = 0 .

MEMORY – ERROR :

1. When we try to access a memory address which doesn't exist , it should flag and set dmem_err to 1 .
2. When we try to access both read and write operations .
3. When both the enables are set to 0 , just flag as dmem_err to 1 .

MAIN CODE :

```
`timescale 1ns/10ps

module dataMem(clk, Add, wEn, M_valA, rEn, m_valM, dmem_err);

    input clk, wEn, rEn;
    input [63:0] Add;
    input [63:0] M_valA;

    output reg [63:0] m_valM;
    output reg dmem_err;

    initial begin
        dmem_err = 0;
    end

    reg [63:0] memory [0:1023];

    always @(*) begin
        if (!(wEn & rEn) && ((0 <= Add) && (Add < 1024)) ) begin
            if (wEn & !rEn) begin
                memory[Add] = M_valA;
                dmem_err = 0;
            end

            if (rEn & !wEn) begin
                m_valM = memory[Add];
                dmem_err = 0;
            end

            if (!rEn & !wEn) begin
                dmem_err = 1;
                m_valM = 8'hx;
            end

        end

        else
        begin
          dmem_err = 1;
          m_valM = 8'hx; // settting the value to unknown ( cause data is not read )
        end

    end

endmodule

```

TESTBENCH :

The inputs in the testbenches can be given as decimal / binary numbers . The following test cases were given so as to cover all possible combinations :

1) Read into a empty memory
2) Read a value from the memory
3) Both the enables are 0
4) Both read and write are performed
5) When the address is invalid

GTKWAVE :

Various control signals along with inputs are given and the output is observed using GTKwave

-