Intro to Processor Architecture – Assignment 1

64-bit ALU and Memory Module

Soham Vaishnav | 2022112002

Rupak Antani | 2022102045

Memory Module:

The memory module is an eminent part of the computer architecture. It is essentially a 2D structure which is accessed mostly in a row-major fashion. In the **y86-64** bit **ISA**, we make use of, as the name suggests, registers of length 64 as well as the data stored in the memory can be 64 bit long. The memory module that we have created has a size of **32x64 bits** with 2 main functionalities – **read,** and **write**. Additionally, it makes the user aware of the possible errors such as **invalid-memory-address – read** and **write**, and **memory-conflict**, and also warns the user in case the user tries to access a preoccupied memory – **memory-location-preoccupied**. All of these functions have been explained in detail in the sections that follow.

1. **Creating memory:**

   Using Verilog, we tried to represent the memory in the form of a 2D array with 32 rows and 64 columns (**32x64 bits**). The inputs to the memory module will be **clk, rst, input_data, read_enable, write_enable, read_addr.** and **write_addr.,** and the outputs will be **read_out, invalid_w_addr., invalid_r_addr., w_r_same_addr., overflow** and **mem_loc_occupied.** The memory register (in Verilog syntax) has been created inside the module – as a part of the code. Since the memory has 32 write-able and read-able locations, out of 64 bits that the register consists of, we only make use of the least significant 5 bits (because $2^5 = 32$).

```
module RAM32×8 (
    input [63:0] write_addr,
    input [63:0] read_addr,
    input signed [127:0] input_data,
    input read_enable, write_enable,
    input clk, rst,
    output reg [63:0] read_out,
    output reg invalid_r_addr,
    output reg invalid_w_addr,
    output reg w_r_same_addr,
    output reg mem_loc_occupied,
    output reg data_overflow
);
    reg [63:0]RAM [31:0];
```

2. **Functionalities:**

   This memory module, like any other memory module, facilitates the user with 2 basic functionalities like **read** from memory and **write** into memory.

   a. **Write –** To write into a memory location, the user has to provide the address of the memory where the input data needs to be stored, as well as has to **enable** the function to make it work.

   Note – The **write** function works at the positive edge of the clock so as to ensure the placing of all the data bits into place in a synchronous manner.

```
always @ (posedge clk) begin
    if (write_enable == 1 && !w_r_same_addr && !invalid_w_addr) begin
        if (!data_overflow) begin
            RAM[write_addr] <= input_data;
        end
        //While writing the memory, regardless of the address being already occupied
        //with past data or not, new data will be over-written at that location
    end
end
```

   b. **Read –** To read, like write, the user has to provide the address of the location he/she would like to extract the memory from. Unlike **write**, **read** function does not occur at any edge of the clock. The reason behind this is to avoid the inconvenience that may arise due to the **latency** which will occur if reading is done at any edge of the clock.

```
integer i;
always @ (*) begin
    if (read_enable == 1 && !w_r_same_addr && !invalid_r_addr) begin
        read_out <= RAM[read_addr];
    end
```

3. **Errors and Warnings:**

   It is important to make the user aware of the errors that the system may encounter given a set of inputs. Furthermore, we have implemented an extra flag of warning just to enlighten the user of the consequence of going ahead with the input combinations giving rise to it. They are as follows:

```
always @ (write_enable or read_enable or input_data) begin
    mem_loc_occupied <= 0;
    w_r_same_addr <= 0;
    invalid_w_addr <= 0;
    invalid_r_addr <= 0;
    data_overflow <= 0;
    if (write_addr > 64'b11111 || write_addr < 0) begin
        invalid_w_addr <= 1;
    end
    if (read_addr > 64'b11111 || read_addr < 0) begin
        invalid_r_addr <= 1;
    end
    if ((read_enable && write_enable) && read_addr == write_addr) begin
        w_r_same_addr <= 1;
    end
    if (write_enable && RAM[write_addr] >= 0) begin
        mem_loc_occupied <= 1;
    end
    if (input_data > (2**63-1) || input_data < -(2**63-1)) begin
        data_overflow <= 1;
    end
end
```

a. **Invalid-read/write-address –** Since we have only 32 accessible memory locations. Therefore, as mentioned earlier, we use of 5 out of the 64 bits of the address register. This very approach forms the basis of the conditions that check the error for invalid address. So, if the user tries to access any location which has a value greater than 31 (because register indexing starts from 0), we raise a flag of invalid address and don't allow the demanded operation to take place. This flag is raised for both – **read** and **write** – operations.
b. **Read/write-conflict –** It may so happen that the user enables read and write options at the same time. It will cause a problem particularly when the command is to read from and write into the same location in memory. The problem arises mainly due to the redundancies in the very commands – read and writing from same memory location does not essentially make sense if the order of prioritizing the commands is not specified.
c. **Data-Overflow –** One row in memory can store signed data within 64 bits. Therefore, this flag is raised when the input number exceeds the limit.
d. **Preoccupied-Memory-Location –** This is an extra flag that we felt like raising. It is essentially a warning which is raised when the user tries to write new data into a memory location which is already preoccupied with some older chunk of data. However, this warning is just to make the user aware of the scenario in the memory and the location is anyways over-written with the new data. Currently, we haven't implemented a way to preserve the older data alongside the new one and thus the warning.

4. **Simulations and Results**

The following is the set of test-cases used to test simulate the memory module:

```
#13 clk ⩽ 1; #1
input_data ⩽ 7;

#13 clk ⩽ !clk; #1
write_enable ⩽ 1;
write_addr ⩽ 2;

#13 clk ⩽ !clk; #1
write_enable ⩽ 0;
read_enable ⩽ 1;
read_addr ⩽ 2;

#13 clk ⩽ !clk; #1
read_enable ⩽ 0;
write_enable ⩽ 1;
write_addr ⩽ 40;

#13 clk ⩽ !clk; #1
write_enable ⩽ 0;
read_enable ⩽ 1;
read_addr ⩽ 33;
```

```
#13 clk ⩽ !clk; #1
read_enable ⩽ 1;
write_enable ⩽ 1;
write_addr ⩽ 2;
read_addr ⩽ 2;

#13 clk ⩽ !clk; #1
write_enable ⩽ 0;
read_enable ⩽ 1;
read_addr ⩽ 2;

#13 clk ⩽ !clk; #1
read_enable ⩽ 0;
write_enable ⩽ 1;
write_addr ⩽ 2;

#13 clk ⩽ !clk; #1
write_enable ⩽ 0;
read_enable ⩽ 1;
read_addr ⩽ 2;
```

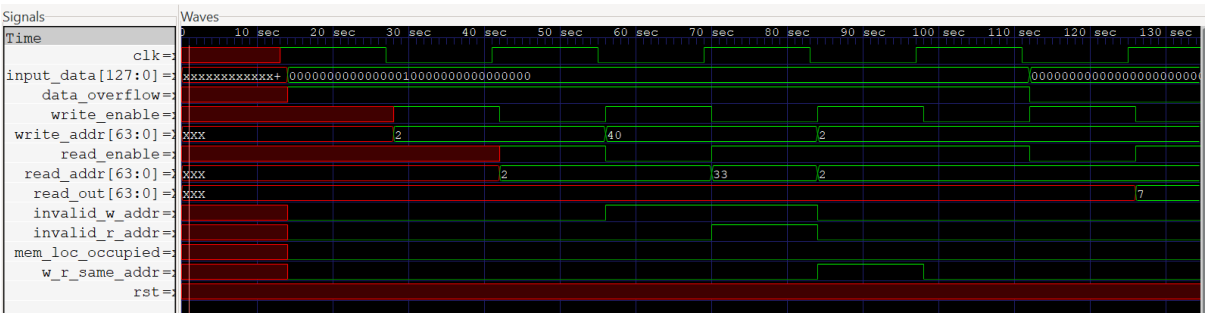
a. **Terminal Results –** For the above set of test-cases we get the following output in the terminal:

```
clk = 1 | Rst = x | DataIn = x, WAdd. = x, RAdd. = x
clk = 0 | Rst = x | DataIn = 18446744073709551616, WAdd. = x, RAdd. = x
clk = 1 | Rst = x | DataIn = 18446744073709551616, WAdd. = 2, RAdd. = x
error-data_overflow: the size of data input is too large!

clk = 0 | Rst = x | DataIn = 18446744073709551616, WAdd. = 2, RAdd. = 2
Data at address 2 is x.

clk = 1 | Rst = x | DataIn = 18446744073709551616, WAdd. = 40, RAdd. = 2
error-mem_addr: invalid memory address for writing data into!

clk = 0 | Rst = x | DataIn = 18446744073709551616, WAdd. = 40, RAdd. = 33
error-mem_addr: invalid memory address for writing data into!

error-mem_addr: invalid memory address for reading data from!

clk = 1 | Rst = x | DataIn = 18446744073709551616, WAdd. = 2, RAdd. = 2
error-data_overflow: the size of data input is too large!

error-mem_conflict: data cannot be written into and read from the same address simultaneously!

clk = 0 | Rst = x | DataIn = 18446744073709551616, WAdd. = 2, RAdd. = 2
Data at address 2 is x.

clk = 1 | Rst = x | DataIn = 7, WAdd. = 2, RAdd. = 2
Data 7 has been successfully written at address 2

clk = 0 | Rst = x | DataIn = 7, WAdd. = 2, RAdd. = 2
Data at address 2 is 7.
```

b. The following piece of code is used to produce the above outputs:

```
always @ (clk) begin
    $display("clk = %d | Rst = %d | DataIn = %0d, WAdd. = %0d, RAdd. = %0d", clk, rst, input_data, write_addr, read_addr);
    if (data_overflow == 1 && write_enable && !invalid_w_addr) begin
        $display("error-data_overflow: the size of data input is too large!\n");
    end
    if (invalid_w_addr == 1) begin
        $display("error-mem_addr: invalid memory address for writing data into!\n");
    end
    if (invalid_r_addr == 1) begin
        $display("error-mem_addr: invalid memory address for reading data from!\n");
    end
    if (w_r_same_addr == 1) begin
        if (!invalid_r_addr && !invalid_w_addr) begin
            $display("error-mem_conflict: data cannot be written into and read from the same address simultaneously!\n");
        end
    end
    if (mem_loc_occupied == 1 && !w_r_same_addr) begin
        $display("warning-data_conflict: desired location pre-occupied - older data will be lost!\n");
    end
    if (read_enable && !invalid_r_addr && !w_r_same_addr) begin
        $display("Data at address %0d is %0d.\n", read_addr, read_out);

    end
    if (clk && !invalid_w_addr && !w_r_same_addr && !data_overflow) begin
        if (write_enable == 1) begin
            $display("Data %0d has been successfully written at address %0d\n", input_data, write_addr);
        end
        else begin
            $display("Write command not enabled!\n");
        end
    end
end
```

c. **GTKWave Plots –** The following plot is obtained for the above test-cases:



The plot is in coherence with the outputs and results obtained.

ALU:

1. **Decoder Block:**

   The 64- bit ALU that is designed is capable of carrying out 4 major operations – Addition, Subtraction, AND and XOR. We take 2 select lines S0 and S1 that help us to choose which operation we want to carry. The below shown table shows the values of S0 and S1 and there corresponding operations.

S0	S1	Operation
0	0	Addition
0	1	Subtraction
1	0	AND
1	1	XOR

The below image shows the Verilog code for the decoder block:

```
module decoder_4_1(s0,s1,out0,out1,out2,out3);

input s0,s1;
output out0,out1,out2,out3;

wire w1,w2,w3,w4;

and and1(out0,~s0,~s1);
and and2(out1,~s0,s1);
and and3(out2,s0,~s1);
and and4(out3,s0,s1);

endmodule
```

2. **Enable Block:**

The enable block takes the two 64 – bit binary numbers A and B as inputs and gives two 64 – binary numbers as output. Each operation block has an enable block. The two outputs are same as A and B if the particular block is enabled based on the select line inputs given to the decoder block. If the block is not enabled, then the two output bits will be 0 and thus the output of that particular block will also be 0. The below image shows the Verilog code for enabling a single bit:

```
module enable_1_bit(e,a,out);

input e,a;
output out;

and and1(out,e,a);

endmodule
```

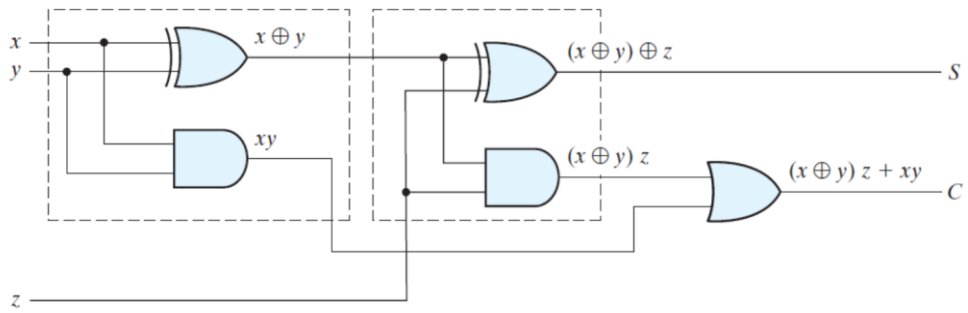
Now, in order to perform the enabling on all the 64 bits of both the numbers, we can use the generator block. In Verilog, generator block is a block which is

used to generate multiple number of modules of the same type. This helps us to carry out an operation multiple times instead of defining the modules separately. The 'for' loop in the generate block runs the module inside it 63 times in this case. The below image shows the Verilog code of the enable block:

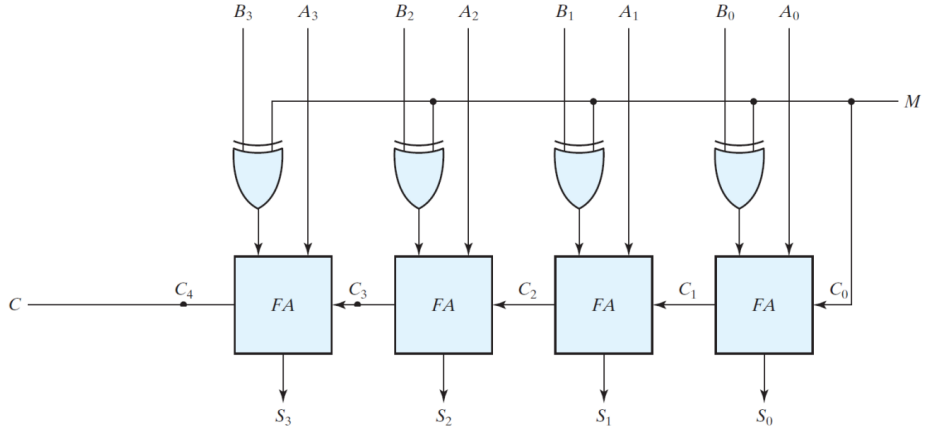
```
module enable_64_bit(e,a,b,c,d);

input e;
input [63:0] a,b;
output [63:0] c,d;

genvar i;

generate
    for (i = 0 ; i < 64 ; i = i + 1 )
    begin
        enable_1_bit u0(e,a[i],c[i]);
    end
endgenerate

genvar j;

generate
    for (j = 0 ; j < 64 ; j = j + 1 )
    begin
        enable_1_bit u0(e,b[j],d[j]);
    end
endgenerate

endmodule
```

3. Adder – Subtractor Block:

The below circuit shows the full adder circuit for adding 3 bits.



Here, x, y and z are the input bits, S represents the sum bit and C represents the carry bit. Now, to make a combined adder-subtractor circuit, the below circuit can be used.



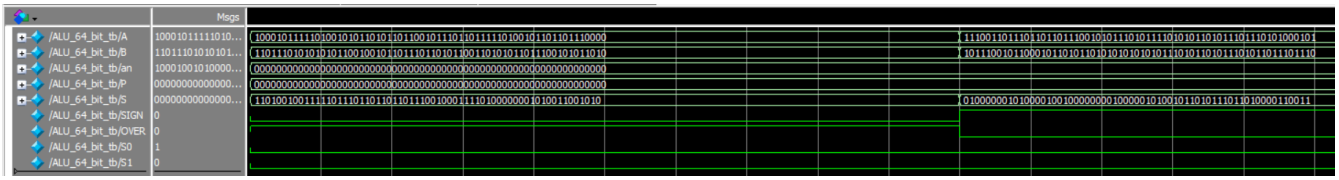
Here, A = [A3 A2 A1 A0] and B = [B3 B2 B1 B0] are two four-bit binary numbers in 2's complement form. If M = 0, addition takes place, that is, A + B, and if M=1, subtraction takes place, that is, A – B. S = [S3 S2 S1 S0] represents the 4-bit sum/difference of the 2 numbers in 2's complement form. and C is the final carry bit. The above process can be similarly performed for 63-bit numbers. The below image shows the Verilog code for a single full adder.

```
module add_sub_1_bit (a,b,c,ma,mb,m,sum,car);

input a,b,c,ma,mb,m;
output sum,car;

wire w1,w2,w3,w4,w5,w6,w7;

xor xor3(w3,b,m);
xor xor4(w4,a,w3);
xor xor5(sum,w4,c);
and and1(w5,a,w3);
and and2(w6,w4,c);
or or1(car,w6,w5);

endmodule
```

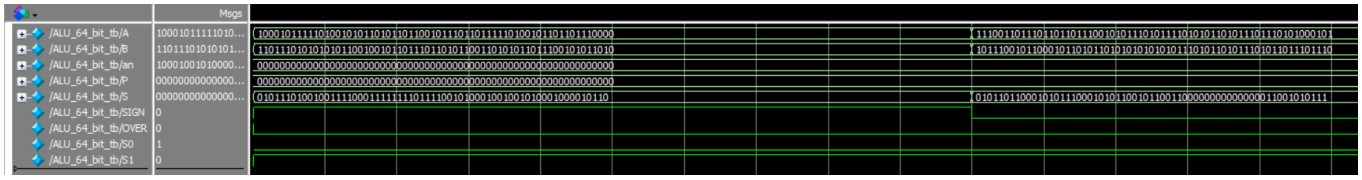
Now, the below image shows the combined 64-bit adder – subtractor.

```
module add_sub_64_bit (a,b,m,sum,over,sign);

input [63:0] a,b;
input m;
output [62:0] sum;
output over,sign;

wire j,cout;
wire [63:0] c;
wire w1,w2,w3,w4,w5,w6;
assign c[0] = m;

genvar i;

generate
    for (i=0 ; i < 63 ; i = i + 1)
    begin
        add_sub_1_bit u0 (a[i],b[i],c[i],a[63],b[63],m,sum[i],c[i+1]);
    end
endgenerate

add_sub_1_bit u1 (.a(a[63]),.b(b[63]),.c(c[63]),.ma(a[63]),.mb(b[63]),.m(m),.sum(cout),.car(j));

and and1(w1,~a[63],~b[63],cout,~m);
and and2(w2,a[63],b[63],~cout,~m);
or or1(w5,w1,w2);

and and3(w3,~a[63],b[63],cout,m);
and and4(w4,a[63],~b[63],~cout,m);
or or2(w6,w3,w4);

or or3(over,w5,w6);

and and5(sign,~over,cout);



endmodule
```

This generate block is used to generate the 63 bits of sum. Next, we make the combinational circuit which is required for to show whether there is overflow or not and to show the sign of the resultant value. The ‘OVER’ bit is 1 if there is an overflow, else 0. Also, the ‘SIGN’ bit is 0 if the result is positive or zero, else it is 1 if the result is negative. NOTE: All the inputs and outputs are in 2’s complement form. The below image shows the waveform of some of the inputs and outputs of this block:



Here, A and B are two 64-bit inputs in the 2’s complement form, where first bit represents sign and the rest 63 bits represent magnitude. Also, S represents the 64-bit sum of A and B, that is, A + B in 2’s complement form, SIGN represents the sign of the output and OVER represents whether overflow has taken place or not.



Here, A and B are two 64-bit inputs in the 2’s complement form, where first bit represents sign and the rest 63 bits represent magnitude. Also, S represents the 64-bit difference of A and B, that is, A - B in 2’s complement form, SIGN represents the sign of the output and OVER represents whether overflow has taken place or not.

4. AND Block:

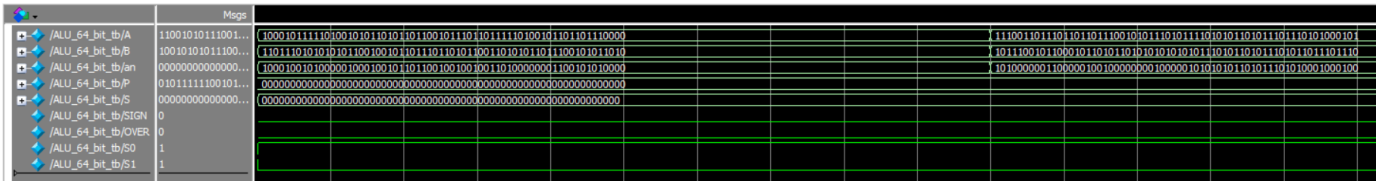
The AND block is used to perform the binary AND operation of each pair of bits of A and B. The below image shows the Verilog code for a single AND operation:

```
1  module and_1_bit (a,b,c);
2
3  input a,b;
4  output c;
5
6  and and1(c,a,b);
7
8  endmodule
```

Now, we can again use a generate block in order to perform AND operation on all the 64 bits of A and B. The below image shows the 64 – bit AND operation:

```verilog
module and_64_bit (a,b,c);

input [63:0] a,b;
output [63:0] c;

genvar i;

generate
    for (i = 0; i < 64 ;i = i + 1)
    begin
        and_1_bit u0 (a[i],b[i],c[i]);
    end
endgenerate

endmodule
```

The below image shows the waveform of some of the inputs and outputs of this block:



Here, A and B are the two 64-bit inputs and ‘an’ is the resultant output after AND operation on A and B.

5. XOR Block:

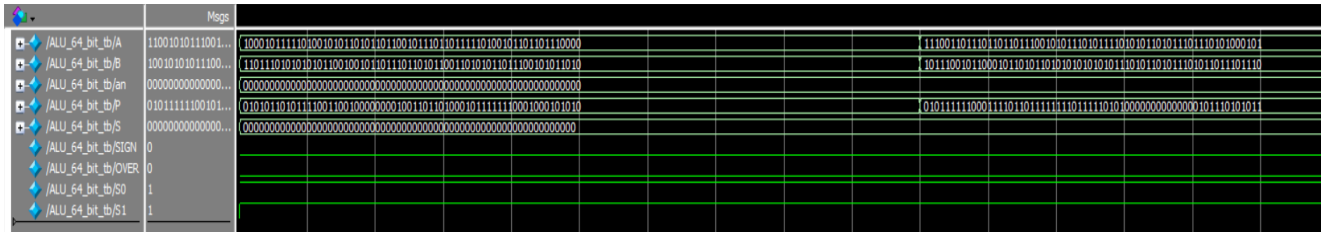
The XOR block is used to perform the binary XOR operation of each pair of bits of A and B. The below image shows the Verilog code for a single XOR operation:

```verilog
module xor_1_bit (a,b,c);

input a,b;
output c;

xor xor1(c,a,b);

endmodule
```

Now, we can again use a generate block in order to perform XOR operation on all the 64 bits of A and B. The below image shows the 64 – bit XOR operation:

```verilog
module xor_64_bit (a,b,c);

input [63:0] a,b;
output [63:0] c;

genvar i;

generate
    for (i = 0; i < 64 ;i = i + 1)
    begin
        xor_1_bit u0 (a[i],b[i],c[i]);
    end
endgenerate

endmodule
```

The below image shows the waveform of some of the inputs and outputs of this block:



Here, A and B are the two inputs and P represents the output after XOR.

6. ALU Testbench:

The role of ALU testbench is to integrate all of the individual blocks that have been used and give and take the corresponding inputs and outputs from each of the blocks in order to ensure the overall working of the ALU. The testbench is also used to give inputs to the circuit that has been designed in Verilog. The below image shows the testbench of ALU:

```
module ALU_64_bit_tb;

reg [63:0] A,B;
wire [63:0] C,D,X,Y,Z,W,an,E,P;
wire [62:0] S;
wire SIGN,OVER;
reg S0,S1;

decoder_4_1 m1(.s0(S0),.s1(S1),.out0(E[0]),.out1(E[1]),.out2(E[2]),.out3(E[3]));

enable_64_bit i1(.e(~S0),.a(A),.b(B),.c(C),.d(D));
enable_64_bit i2(.e(E[2]),.a(A),.b(B),.c(X),.d(Y));
enable_64_bit i3(.e(E[3]),.a(A),.b(B),.c(Z),.d(W));

add_sub_64_bit a1(.a(C),.b(D),.m(S1),.sum(S),.over(OVER),.sign(SIGN));

and_64_bit f1(.a(X),.b(Y),.c(an));

xor_64_bit p1(.a(Z),.b(W),.c(P));

always@(S0,S1,A,B)
begin
    $display("S0 = %d, S1= %d, A = %b, B = %b, Sum/Difference = %b, Sign= %d, Overflow = %d, AND= %b , XOR = %b",S0,S1,A,B,S,SIGN,OVER,an,P);
end

initial
begin
    S0 = 0; S1 = 0; A = 64'b1000101111101001010110101101100101110110111110100101101101110000; B = 64'b1101110101010101100100101101110110101100110101010110111001010110;
    #10 S0 = 0; S1 = 0; A = 64'b1110011011101101101110010101110101011110101010110101110110101000101; B = 64'b1011100101100010110101010101010101110101101011010110111011101110;
    #10 S0 = 0; S1 = 0; A = 64'b1100101011100101010010110101011101101011101111111101010011011; B = 64'b1100101010111001110010010110101101011101111111101010011011;
    #10 S0 = 0; S1 = 1; A = 64'b1000101111101001010110101101100101110110111110100101101101110000; B = 64'b1101110101010101100100101101110110101100110101010110111001010110;
    #10 S0 = 0; S1 = 1; A = 64'b1110011011101101101110010101110101011110101010110101110110101000101; B = 64'b1011100101100010110101010101010101110101101011010110111011101110;
    #10 S0 = 0; S1 = 1; A = 64'b1100101011100101010010110101011101101011101111111101010011011; B = 64'b1100101010111001110010010110101101011101111111101010011011;
    #10 S0 = 1; S1 = 0; A = 64'b1000101111101001010110101101100101110110111110100101101101110000; B = 64'b1101110101010101100100101101110110101100110101010110111001010110;
    #10 S0 = 1; S1 = 0; A = 64'b1110011011101101101110010101110101011110101010110101110110101000101; B = 64'b1011100101100010110101010101010101110101101011010110111011101110;
    #10 S0 = 1; S1 = 0; A = 64'b1100101011100101010010110101011101101011101111111101010011011; B = 64'b1100101010111001110010010110101101011101111111101010011011;
    #10 S0 = 1; S1 = 1; A = 64'b1000101111101001010110101101100101110110111110100101101101110000; B = 64'b1101110101010101100100101101110110101100110101010110111001010110;
    #10 S0 = 1; S1 = 1; A = 64'b1110011011101101101110010101110101011110101010110101110110101000101; B = 64'b1011100101100010110101010101010101110101101011010110111011101110;
    #10 S0 = 1; S1 = 1; A = 64'b1100101011100101010010110101011101101011101111111101010011011; B = 64'b1100101010111001110010010110101101011101111111101010011011;
end

endmodule
```

In this Verilog code, we first define all the inputs and intermediate and final outputs. Next, we call all the individual modules. First, we call the decoder module, then 3 enable block modules for the 3 operation blocks. After this, we can the adder – subtractor module, AND module and lastly the XOR module. Lastly, we give the input sequence for testing the circuit.