SIMPLE ARITHMETIC LOGIC UNIT (ALU)

DSD Project Report

Submitted in partial fulfilment of the requirements for the degree of

BACHELOR OF TECHNOLOGY in

ELECTRONICS AND COMMUNICATION ENGINEERING

by

SAI ASWIN MADHAVAN

(211EC244)

SANGANABASU M HERUR

(211EC245)

DEPARTMENT OF ELECTRONICS AND COMMUNICATION ENGINEERING,

NATIONAL INSTITUTE OF TECHNOLOGY KARNATAKA, SURATHKAL,

MANGALORE - 575025

January, 2023

ABSTRACT

This project presents a simple arithmetic logic unit (ALU) implemented in Verilog. The ALU takes in a bit stream and a clock signal as inputs, and performs a simple operation on two 4-bit values, "a" and "b". The "out" output is connected to the "out_reg" register and will change on the rising edge of the clock signal.

The project utilises a state machine to control the operation, where the current state is stored in the "cur_state" register and the next state is stored in the "next_state" register. The state machine has 11 possible states, represented by the parameters "S0" to "S15". The "count" register is used to keep track of the number of bits received in the bit stream.

The operation performed is determined by the current state and the bit stream input. The possible operations include loading values to the registers and performing operations like adding, subtracting, bitwise OR and bitwise AND .The final result is stored in the "buffer" register and is assigned to the "out_reg" register when the operation is complete.

Overall, this project provides a basic understanding of how to implement a simple ALU in Verilog and how to use state machines to control the operations. It can be used as a starting point for more complex digital systems that require arithmetic operations.

CONTENTS

INTRODUCTION

An Arithmetic Logic Unit (ALU) is a fundamental building block of a central processing unit (CPU) in a computer. It is responsible for performing arithmetic and logical operations on binary numbers. The ALU is a digital circuit that performs operations such as addition, subtraction, multiplication, division, bitwise operations like AND, OR, NOT, and XOR, and many others.

The ALU is an essential part of a CPU, as it is responsible for executing the instructions of a computer program. Without an ALU, a computer would not be able to perform basic calculations or make logical decisions. The ALU is also important for performing floating-point operations, which are essential for many applications such as scientific simulations, computer graphics, and machine learning.

In addition, the ALU plays a critical role in the control unit of the CPU. It receives instructions from the control unit and performs the appropriate operation, and then sends the result back to the control unit. The control unit then uses this result to determine the next instruction to be executed.

This project aims to build a simpler version of an ALU with a lesser number of operations like addition, subtraction, bitwise OR and bitwise AND. This module is a SIPO module taking inputs from a bitstream and gives a 4-bit parallel output.

DESIGN

The initial thoughts on the project were directed towards deciding on a format to send the bitstream. The following is the decided bitstream format.

Start Bit	Op Bit 1	Op Bit 2	Op Bit 3	Data Bit 1	Data Bit 2	Data Bit 3	Data Bit 4	Stop Bit
1	X	X	X	X	X	X	X	X

Bitstream Format

The next step was to create an operation table to decide the unique bitstream code for each operation. The following is the decided operation table.

Operation	Start Bit	Op Bit 1	Op Bit 2	Op Bit 3	Data Bit 1	Data Bit 2	Data Bit 3	Data Bit 4	Stop Bit
Load A	1	0	0	X	A[3]	A[2]	A[1]	A[0]	X
Load B	1	0	1	X	B[3]	B[2]	B[1]	B[0]	X
A + B	1	1	0	0	X	X	X	X	X
A - B	1	1	0	1	X	X	X	X	X
A & B	1	1	1	0	X	X	X	X	X
A | B	1	1	1	1	X	X	X	X	X

Operation Table

To identify each code, we need to design pattern detectors using state machines. Each state detects a particular code.

S0 = Rest State

S1 = 1

S2 = 10

S3 = 11

S4 = 100

S5 = 101

S6 = 110

S7 = 111

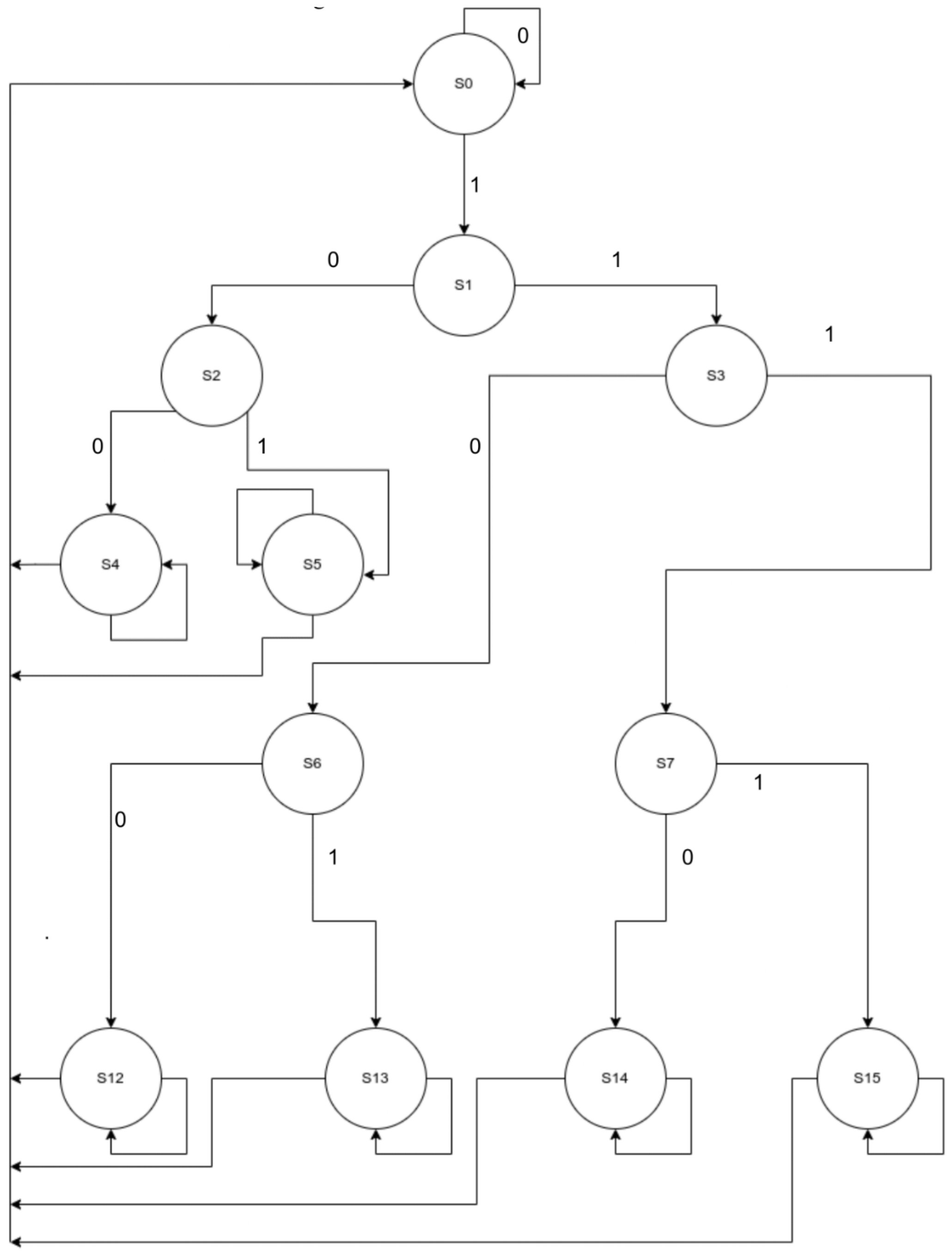
S12 = 1100

S13 = 1101

S14 = 1110

S15 = 1111

Shown below is the state diagram.



State Diagram

States S4, S5. S12. S13, S14, S15 are output states. They stay in the same state until the count goes back to zero. Once count returns to zero, out_reg is updated and we go back to the rest state.

IMPLEMENTATION

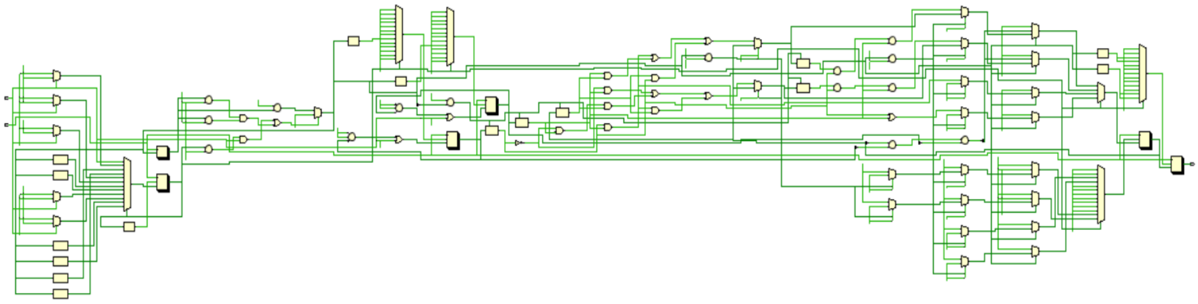
The above module was implemented using verilog. The code is given below.

```
`timescale 1ns / 1ps


module simple_alu(
   input bit_stream,clk,
   output [3:0] out
   );

   reg [3:0] out_reg = 0;
   reg [3:0] a = 0;
   reg [3:0] b = 0;
   reg [3:0] buffer = 0;
   assign out = out_reg;

   reg [3:0] cur_state = 0;
   reg [3:0] next_state = 0;

   parameter [3:0] S0 = 0;
   parameter [3:0] S1 = 1;
   parameter [3:0] S2 = 2;
   parameter [3:0] S3 = 3;
   parameter [3:0] S4 = 4;
   parameter [3:0] S5 = 5;
   parameter [3:0] S6 = 6;
   parameter [3:0] S7 = 7;
   parameter [3:0] S12 = 12;
   parameter [3:0] S13 = 13;
   parameter [3:0] S14 = 14;
   parameter [3:0] S15 = 15;

   reg [3:0] count = 0;




   always @ (posedge clk)begin
      cur_state = next_state;
      if((cur_state == S0 && bit_stream == 1) || (count >= 1 && count < 8)) count = count
+ 1;
```

```
else count = 0;
case(cur_state)
  S0:begin
    if(bit_stream) next_state = S1;
    else next_state = S0;
  end
  S1:begin
    if(~bit_stream) next_state = S2;
    else next_state = S3;
  end
  S2:begin
    if(~bit_stream) next_state = S4;
    else next_state = S5;
  end
  S3:begin
    if(~bit_stream) next_state = S6;
    else next_state = S7;
  end
  S4:begin
    case(count)
      0: next_state = S0;
      default:begin
        a = (a << 1) | bit_stream;
        next_state = S4;
      end
    endcase
  end
  S5:begin
    case(count)
      0: next_state = S0;
      default:begin
        b = (b << 1) | bit_stream;
        next_state = S5;
      end
    endcase
  end
  S6:begin
    if(~bit_stream) next_state = S12;
    else next_state = S13;
  end
  S7:begin
    if(~bit_stream) next_state = S14;
    else next_state = S15;
```

```
        end
        S12:begin
          case(count)
            0: begin
              out_reg = buffer;
              buffer = 0;
              next_state = S0;
            end
            8:begin
              buffer[count-5] = buffer[count-5] + a[count-5] + b[count-5];
              next_state = S12;
            end
            default: begin
              buffer[count-4] = (a[count-5] & b[count-5]) | (b[count-5] & buffer[count-5])
| (buffer[count-5] & a[count-5]);
              buffer[count-5] = buffer[count-5] + a[count-5] + b[count-5];

              next_state = S12;
            end
          endcase
        end
        S13:begin
          case(count)
            0: begin
              out_reg = buffer;
              buffer = 0;
              next_state = S0;
            end
            8:begin
              buffer[count-5] = buffer[count-5] + a[count-5] + b[count-5];
              next_state = S13;
            end
            default: begin
              buffer[count-4] = (~a[count-5] & b[count-5]) | (~a[count-5] & ~b[count-5]
& buffer[count-5]) | (a[count-5] & b[count-5] & buffer[count-5]) ;
              buffer[count-5] = buffer[count-5] + a[count-5] + b[count-5];
              next_state = S13;
            end
          endcase
        end
        S14:begin
          case(count)
            0: begin
```

```
                out_reg = buffer;
                buffer = 0;
                next_state = S0;
            end
            default: begin
                buffer[count-5] = a[count-5] & b[count-5];
                next_state = S14;
            end
        endcase
    end
    S15:begin
        case(count)
            0: begin
                out_reg = buffer;
                buffer = 0;
                next_state = S0;
            end
            default: begin
                buffer[count-5] = a[count-5] | b[count-5];
                next_state = S15;
            end
        endcase
    end
  endcase
end

endmodule
```



Elaborated Design from Vivado tool

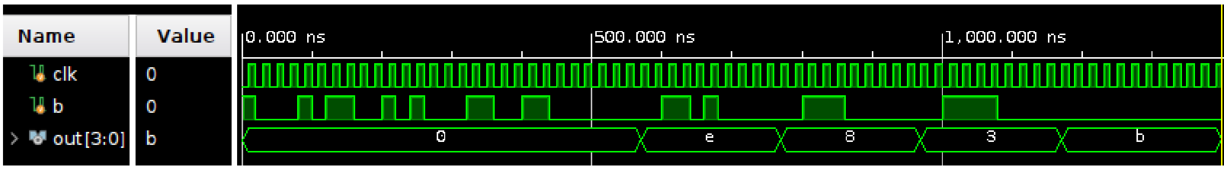
RESULTS

Test bench:

```
`timescale 1ns / 1ps

module alu_tb();
   reg clk,b;
   wire [3:0] out;


   simple_alu alu(b,clk,out);

   initial begin
      clk = 0;
      forever #10 clk = ~clk;
   end

   initial begin
      b = 1;#20;
      b = 0;#20;
      b = 0;#20;
      b = 0;#20;
      b = 1;#20;
      b = 0;#20;
      b = 1;#20;
      b = 1;#20;

      b = 0;#40;

      b = 1;#20;
      b = 0;#20;
      b = 1;#20;
      b = 0;#20;
      b = 0;#20;
      b = 0;#20;
      b = 1;#20;
      b = 1;#20;

      b = 0;#40
```

```
b = 1;#20;
b = 1;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;

b = 0;#40;

b = 1;#20;
b = 1;#20;
b = 0;#20;
b = 1;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;

b = 0;#40;

b = 1;#20;
b = 1;#20;
b = 1;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;

b = 0;#40;

b = 1;#20;
b = 1;#20;
b = 1;#20;
b = 1;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;
b = 0;#20;

b = 0;#40;
```

```
    end
endmodule
```



Output simulation of above testbench

The above testbench loads the values 11 and 3 to registers “a” and “b” respectively and computes the four operations on them. The results can be seen from the output waveform.

CONCLUSION

The project provides a basic understanding of how to implement a simple ALU in Verilog and how to use state machines to control the operations and creates a basic understanding of the working of ALUs.. It can also serve as a starting point for more advanced digital systems that require arithmetic operations. This design has many flaws like inefficient timing, incompatible input and output, etc that can be corrected with minimal changes to the design. Furthermore, this project could be enhanced by adding more advanced operations such as multiplication, division, and floating point operations.