Project 9 Arithmetic and Logic Unit (ALU)

Introduction to ALU's

Introduction

Arithmetic and Logic Unit (ALU) circuits are found at the core of microprocessors. They combine the input data operands, and produce output data as directed by the computer's programming instructions. Computer programming instructions, or machine codes, contain several bits that configure the ALU to combine input data using arithmetic or logic functions. For example, an "ADD" instruction would contain bits to configure the ALU to combine input operands using an adding circuit, and an "OR" instruction would contain bits to configure the ALU to combine operands using an OR'ing circuit.

Although ALUs are seemingly complex, they are actually fairly easy to specify and straightforward to design. This project presents the design of a simple ALU.

Before you begin, you should:

- Understand the design and operation of arithmetic circuits;
- Understand how to specify and design combinational circuits;
- Be a confident and capable Vivado user.

After you're done, you should:

- Understand how ALUs function;
- Be able to describe a simple ALU in Verilog;
- Be comfortable designing more complex combinational circuits.

Requirements

☑ 1. ALU

Create an 8-bit ALU in Verilog that supports the functions shown in the following table. Connect data registers to both ALU inputs. Through a testbench, give different values to the ALU data and control inputs. Demonstrate all ALU functions in simulation and observe the ALU output.

OpCode	Description	Output F
000	Addition	$A+ B$
001	Increment	$A+ 1$
010	Subtract	$A— B$
011	Bit-wise XOR	$A⊕ B$
100	Bit-wise OR	$A \mid B$
101	Bit-wise AND	$A \& B$

Tutorial

Arithmetic and Logic Units

Arithmetic and Logic Units (or ALUs) are found at the core of microprocessors, where they implement the arithmetic and logic functions offered by the processor (e.g., addition, subtraction, AND'ing two values, etc.). **An ALU is a combinationalcircuit that combines many common logic circuits in one block.** Typically, ALU inputs are comprised of two N-bit busses,a carry-in, and M select lines that select between the 2^M ALU operations. ALU outputs include an N-bit bus for function output and a carry-out.

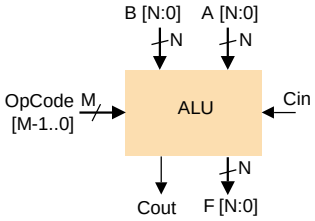

Figure 1. Arithmetic and logic units

ALUs can be designed to perform a variety of different arithmetic and logic functions. Possible arithmetic functions include addition, subtraction, multiplication, comparison, increment, decrement, shift, and rotate; possible logic functions include AND, OR, XOR, XNOR, INV, CLR (for clear), and PASS (for passing a value unchanged). All of these functions find use in computing systems, although a complete description of their use is beyond the scope of this document. An ALU could be designed to include all of these functions, or a subset could be chosen to meet the specific needs of a given application. Either way, the design process is similar (but simpler for an ALU with fewer functions).

As an example, we will consider the design of an **ALU that can perform one of eight functions on 8-bit data values**. Thisdesign, although relatively simple, is not unlike many of ALUs that have been designed over the years for all sizes and performance ranges of processors. Our ALU will feature **two 8-bit data inputs, an 8-bit data output, a carry-in and a carry-out,** and three function select inputs (**S2, S1, S0) providing selection between eight operations (three arithmetic, four logic, and a clear or '0')**.

Targeted ALU operations are shown in the operation table below. The three control bits used to select the ALU operation are called the operation code (or opcode), because if this ALU were used in an actual microprocessor, these bits would come from the opcodes (or machine codes) that form the actual low-level computer programming code. (Computer software today is typically written in a high-level language like 'C', which is compiled into assembler code. Assembler code can be directly translated into machine codes that cause the microprocessor to perform particular functions).

OpCode	**Description**	**Function**
000	Addition	$F = A + B$
001	Increment	$F = A + 1$
010	Subtract	$F = A - B$
011	No Operation	$F = 0$
100	Bit-wise XOR	$F = A \oplus B$
101	Bit-wise Inversion	$F = \overline{A}$
110	Bit-wise OR	$F = A \mid B$
111	Bit-wise AND	$F = A \ \& B$

Since ALUs operate on binary numbers, the bit-slice design method is indicated. ALU design should follow the same process as other bit-slice designs: first, define and understand all inputs and outputs of a bit slice (i.e., prepare a detailed block diagram of the bit slice); second, capture the required logical relationships in some formal method (e.g., a truth table); third, find minimal circuits (by using K-maps or espresso) or write Verilog code; and fourth, proceed with circuit design and verification.

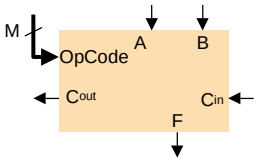

Figure 2. ALU bit-sliced block.

A block diagram and operation table for our ALU example are shown above. Note in the operation table, that entered variables are used to define the functional requirements for the two outputs (F and Cout) of the bit-slice module. If entered variables were not used, the table would have required 64 rows. Since Cout is not required for logic functions, it can be assigned '0' or don't care for those rows. A circuit diagram for an 8-bit ALU based on the developed bit slice is shown in Fig. 3.

OpCode	**Description**	C_{out}
000	Addition	$A\cdot B+ C_{\text{in}}\cdot (A\oplus B)$
001	Increment	$C_{\text{in}}\cdot A$
010	Subtract	$\overline{(A\oplus B)}\cdot C_{\text{in}} + \overline{A}\cdot B$(A ⊕ B)
011	No Operation	00
100	Bit-wise XOR	00
101	Bit-wise Inversion	00
110	Bit-wise OR	00
111	Bit-wise AND	00

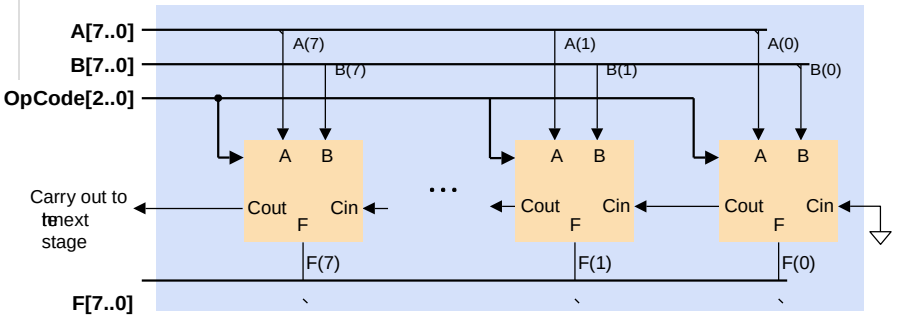

Figure 3. Block diagram of ALU composed of bit-sliced blocks.

Once the ALU operation table is complete, a circuit can be designed following any one of several methods: K-maps can be constructed and minimal circuits can be looped; muxes can be used (with an 8:1 mux for F and a 4:1 mux for Cout); the information could be entered into a computer-based minimizer and the resulting equations implemented directly; or we could bypass all the difficult and error-prone structural work and create a behavioral description using Hardware Description Language (such as Verilog or VHDL).