Lab 4: Arithmetic Logic Unit (ALU)

Introduction

The heart of every computer is an Arithmetic Logic Unit (ALU). This is the part of the computer which performs arithmetic operations on numbers, e.g. addition, subtraction, etc. In this lab use the Verilog language to implement an ALU having 10 functions. Use of the case structure will make this job easy.

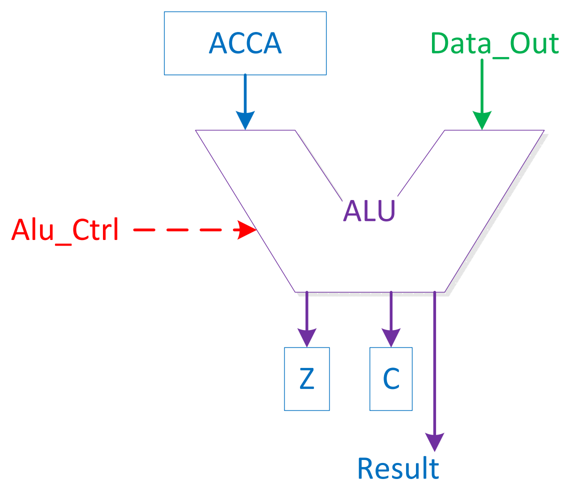

Figure 1: Arithmetic Logic Unit (ALU)

The ALU that you will build (see Figure 1) will perform 10 functions on 8-bit inputs (see Table 1). Please make sure you use the same variable name as the ones used in this lab. Do **NOT** make your own. The ALU will generate an 8-bit result (`Result`), a one bit carry (`C`), and a one bit zero-bit (`Z`). To select which of the 10 functions to implement, you will use `Alu_Ctrl` as the selection lines.

1 Prelab

1.1. Fill out Table 1 (Give *unique* values to each instruction.) How many bits should `Alu_Ctrl` be?

1.2. Write code to implement the ALU.

2 Lab

2.1. Write a Verilog program based off of your code written in the Prelab to implement the ALU.

2.2. Design the ALU using Verilog. **Make sure you deal with any unused bit combinations of the `Alu_Ctrl` lines.** (Hint: review `default` cases)

2.3. Simulate the ALU and test different combinations of DATA and ACCA. **Test ALL of the instructions.**

2.4. Create another program that will call your ALU module. In this module, have ACCA and DATA as external inputs as well as Alu_Ctrl. Output your results on two 7-segment displays. (Pinouts are included in Table 2)

2.5. Program your ALU code into your FPGA.

Table 1: Arithmetic Logic Unit Instructions

Alu_Ctrl	**Instruction**	**Operation (Mnemonic)**
	LDDA	Loads ACCA with the value on the Data bus. Z changes to 1 if Result == 0. (Load ACCA from Data)
	ADDA	Adds the value on the Data bus to the value in ACCA and saves the result in ACCA. C is the carry (out) from addition and Z is set if the result is 0. (Add ACCA and Data)
	SUBA	Subtracts the value on the Data bus from the value in ACCA and saves the result in ACCA. C is the carry (in) from subtraction and Z is set if the result is 0. (Subtract value in Data from ACCA)
	ANDA	Perform a bitwise AND of the value on the Data bus with the value in ACCA. Save the result in ACCA. C should be the logical AND of the value on the Data bus with the value in ACCA. Z is set if the result is 0. (AND of ACCA and value on Data)
	ORAA	Perform a bitwise OR of the value on the Data bus with the value in ACCA. Save the result in ACCA. C should be the logical OR of the value on the Data bus with the value in ACCA. Z is set if the result is 0. (OR of ACCA and value on Data)
	COMA	Replace the value in ACCA with its one's complement. C is set to 1 and Z is set if the result is 0. (Compliment ACCA)
	INCA	Increment value in ACCA. Z is set if the result is 0. (INCA ACCA)
	LSLA	Logical shift left of ACCA. C is set to the previous MSB of ACCA and Z is set if the result is 0. (Logical shift left ACCA)
	LSRA	Logical shift right of ACCA. C is set to the previous LSB of ACCA and Z is set if the result is 0. (Logical shift right ACCA)
	ASRA	Arithmetic shift right of ACCA. C is set to the previous LSB of ACCA and Z is set if the result is 0. (Arithmetic shift right ACCA)
	ZERO	Zero the value of ACCA. C is set to 0 and Z is set to 1. (Zero ACCA)
	RST	Reset ACCA to 0xFF. C is set to 0 and Z is set to 0. (Reset ACCA)

Table 2: CMOD-S6 DIP Assignments

DIP Pin	**FPGA Pin**	Wire	Wire	**FPGA Pin**	DIP Pin
1	**P5**	PIO01	PIO48	**M2**	48
2	**N5**	PIO02	PIO47	**M1**	47
3	**N6**	PIO03	PIO46	**L2**	46
4	**P7**	PIO04	PIO45	**L1**	45
5	**P12**	PIO05	PIO44	**K2**	44
6	**N12**	PIO06	PIO43	**K1**	43
7	**L14**	PIO07	PIO42	**J2**	42
8	**L13**	PIO08	PIO41	**J1**	41
9	**K14**	PIO09	PIO40	**G2**	40
10	**K13**	PIO10	PIO39	**G1**	39
11	**J14**	PIO11	PIO38	**H2**	38
12	**J13**	PIO12	PIO37	**H1**	37
13	**H14**	PIO13	PIO36	**F2**	36
14	**H13**	PIO14	PIO35	**F1**	35
15	**F14**	PIO15	PIO34	**E2**	34
16	**F13**	PIO16	PIO33	**E1**	33
17	**G14**	PIO17	PIO32	**D2**	32
18	**G13**	PIO18	PIO31	**D1**	31
19	**E14**	PIO19	PIO30	**C1**	30
20	**E13**	PIO20	PIO29	**B1**	29
21	**D14**	PIO21	PIO28	**A2**	28
22	**D13**	PIO22	PIO27	**B3**	27
23	**C13**	PIO23	PIO26	**A3**	26
24		VU	GND		25

3 Supplement: Verilog (3)

3.1 Parameterization

3.1.1 Macros

Listing 1: Macros in Verilog

```
‘define Rst_Addr 8’hFF // Gets Expaned
assign data = ‘Rst_Addr; // Tic is Necessary
```

3.1.2 Parameters

Listing 2: Parameters in Verilog

```
parameter num = 8;
```

Parameters are constants, not variables.

3.2 Operators

3.2.1 Ternary Operator

Listing 3: Ternary Operator in Verilog

```
assign y = sel ? a : b;
```

If `sel` is true, `y` is assigned to `a`, otherwise it is assigned to `b`.

3.2.2 Concatenation

Listing 4: Concatenation in Verilog

```
{a, b, c}
```

Bits are concatenated using { }.

3.2.3 Comparison

Listing 5: Comparison in Verilog

```
if(a > b) y = a;
```

Compare `a` to `b`, if true set `y` equal to `a`. Other comparisons are listed in Listing 6.

Listing 6: Comparison Operators

```
>     // Greater than
<     // Less than
>=    // Greater than or equal to
<=    // Less than or equal to
==    // Equality
===   // Equality including X and Z
!=    // Inequality
!==   // Inequality including X and Z
```

3.2.4 Logical Operators

Listing 7: Logical Operators

```
!     // Logical negation
&&    // Logical and
||    // Logical or
```

3.2.5 Binary Arithmetic Operators

Listing 8: Binary Arithmetic Operators

```
+     // Addition
-     // Subtraction
*     // Multiplication
/     // Division (truncated)
%     // Modulus
```

3.2.6 Unary Arithmetic Operators

Listing 9: Unary Arithmetic Operators

```
-     // Change the sign of the operand
```

3.2.7 Bitwise Operators

Listing 10: Bitwise Operators

```
~     // Bitwise negation
&     // Bitwise AND
|     // Bitwise OR
^     // Bitwise XOR
~^    // Bitwise XNOR
^~     // Bitwise XNOR (also)
```

3.2.8 Unary Reduction Operators

- Produce a single bit result by applying the operator to all the bits of the operand.

Listing 11: Unary Reduction Operators

```
~     // Bitwise negation
&     // Bitwise AND
|     // Bitwise OR
‘&    // Reduction NAND
~|    // Reduction NOR
^     // Bitwise XOR
~^    // Bitwise XNOR
^~     // Bitwise XNOR (also)
```

3.2.9 Shift Operators

- Left operand is shifted by the number of bit positions given by the right operand.
- Zeros are used to fill vacated bit positions.

Listing 12: Shift Operators

```
<<    // Logical left shift
>>    // Logical right shift
```

3.2.10 Precedence

Listing 13: Precedence

```
/* Highest Precedence */
  !, ~
  *, /, %
  +, -
  <<, >>
  <, <=, >, >=
  &
  ^, ^~
  |
  &&
  ||
  ?:
/* Lowest Precedence */
```