

Lab 5: Arithmetic Logic Unit (ALU)

October 10, 2008

Contents

1 Prelab	4
2 Lab	4
3 Supplementary Material	6
3.1 Verilog	6
3.1.1 Parameters	6
3.1.2 Operators	6
3.1.3 8-bit Adder	8

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 you will use the Verilog language to implement an ALU having 10 functions. Use of the `case` structure will make this job easy.

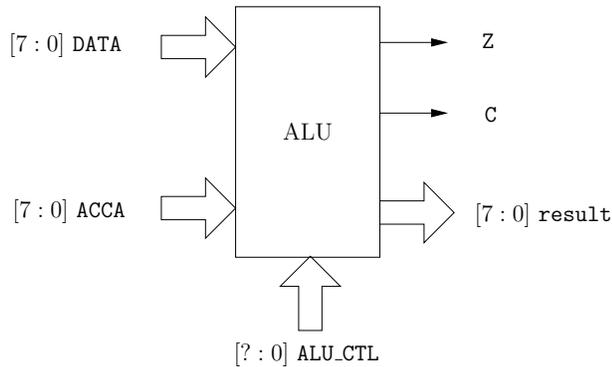


Figure 1: ALU block diagram

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. Don't 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_CTL** as the selection lines.

Table 1: ALU Functions

ALU_CTL	Mnemonic	Description
	Load	(load DATA into result) $DATA \Rightarrow result$ C is a don't care $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	ADDA	(add DATA to ACCA) $ACCA + DATA \Rightarrow result$ C is carry from addition $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	SUBA	(subtract DATA from ACCA) $ACCA - DATA \Rightarrow result$ C is borrow from subtraction $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	ANDA	(logical AND DATA with ACCA) $ACCA \& DATA \Rightarrow result$ C is a don't care $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	ORAA	(logical OR DATA with ACCA) $ACCA DATA \Rightarrow result$ C is a don't care $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	COMA	(complement of ACCA) $\overline{ACCA} \Rightarrow result$ $1 \Rightarrow C$ $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	INCA	(increment ACCA by 1) $ACCA + 1 \Rightarrow result$ C is a don't care $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	LSRA	(logical shift right of ACCA) Shift all bits of ACCA one place to the right: $0 \Rightarrow result[7]$, $ACCA[7 : 1] \rightarrow result[6 : 0]$ $ACCA[0] \Rightarrow C$ $1 \rightarrow Z$ if $result == 0$, $0 \rightarrow Z$ otherwise
	LSLA	(logical shift left of ACCA) Shift all bits of ACCA one place to the left:

		$0 \Rightarrow \text{results}[0], \text{ACCA}[6 : 0] \rightarrow \text{result}[7 : 1]$ $\text{ACCA}[7] \Rightarrow \text{C}$ $1 \rightarrow \text{Z}$ if $\text{result} == 0, 0 \rightarrow \text{Z}$ otherwise
	ASRA	(Arithmetic shift right of ACCA) Shift all bits of ACCA one place to the right: $\text{ACCA}[0] \Rightarrow \text{results}[7], \text{ACCA}[7 : 1] \rightarrow \text{result}[6 : 0]$ $\text{ACCA}[0] \Rightarrow \text{C}$ $1 \rightarrow \text{Z}$ if $\text{result} == 0, 0 \rightarrow \text{Z}$ otherwise

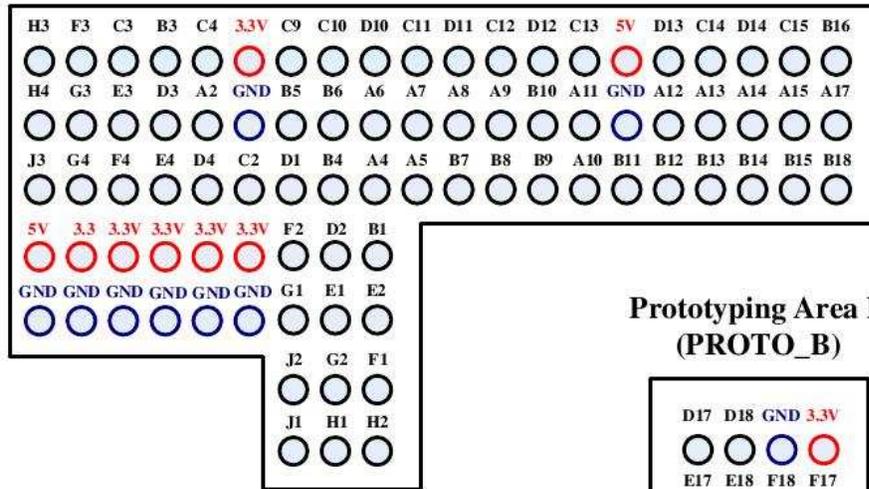
1 Prelab

1. Fill out Table 1.
2. Write a Verilog program to implement the ALU.

2 Lab

1. Design the ALU using Verilog. (Make sure you deal with any unused bit combinations of the ALU_CTL lines).
2. Simulate the ALU and test different combinations of DATA and ACCA.
3. Program your ALU code into your CPLD.
4. Create another program that will call your ALU module. In this module read external inputs for ACCA and DATA as well as the ALU_CTR. Output your results on two 7-segment displays (Pinout of the MAX II micro board is shown in Figure 2).

**Prototyping Area A
(PROTO_A)**



**Prototyping Area B
(PROTO_B)**

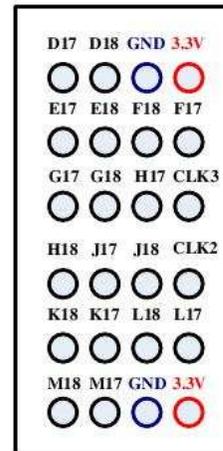


Figure 2: I/O map of prototyping areas

3 Supplementary Material

3.1 Verilog

3.1.1 Parameters

Parameters are constants and not variables.

```
parameter num = 8;
```

3.1.2 Operators

?:Construct

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

If `sel` is true, then `y` is assigned `a`, else it is assigned `b`.

Concatenations In Verilog it is possible to concatenate bits using `{.}`.

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

is equivalent to

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

Comparison Operators

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

assign `y` to `a` if `a>b` and assign it to `b` otherwise. Table 2 shows a list of comparison operators.

Table 2: Comparison Operators

Operator	Description
>	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

- for `==` and `!=` the result is `x`, if either operand contains an `x` or `z`.

Table 3: Logical Operators

Operator	Description
!	logical negation
&&	logical AND
	logical OR

Logical Operators Table 3 shows a list of logical operators.

- Evaluation is performed left to right.
- x if any of the operands has unknown x bits.

Binary Arithmetic Operators Table 4 shows a list of arithmetic operators.

Table 4: Arithmetic Operators

Operator	Description
+	addition
-	subtraction
*	multiplication
/	division (truncates any fractional part)
%	equality

Unary Arithmetic Operators Table 5 shows a list of unary arithmetic operators.

Table 5: Unary Arithmetic Operators

Operator	Description
-	Change the sign of the operand

Bitwise Operators Table 6 shows a list of bitwise operators.

Unary Reduction Operators Table 7 shows a list of unary reduction operators. They produce a single bit result by applying the operator to all of the bits of the operand.

Shift Operators Table 8 shows a list of 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.

Table 6: Bitwise Operators

Operator	Description
\sim	Bitwise negation
$\&$	Bitwise AND
$ $	Bitwise OR
$\sim \&$	Bitwise NAND
$\sim $	Bitwise OR
\sim^{\wedge} or $\wedge \sim$	Equivalence

Table 7: Unary Reduction Operators

Operator	Description
\sim	Bitwise negation
$\&$	Bitwise AND
$ $	Bitwise OR
$\sim \&$	Bitwise NAND
$\sim $	Bitwise OR
\sim^{\wedge} or $\wedge \sim$	Equivalence

Operator Precedence Rule Table 9 shows a list operator precedence rules.

3.1.3 8-bit Adder

Program 1 shows how to implement an 8-bit adder.

Program 1 An example of an 8-bit adder.

```

wire [7:0] sum, a, b;
wire cin,cout;

assign {cout,sum} = a+b+cin;

```

Table 8: Shift Operators

Operator	Description
<<	left shift
>>	right shift

Table 9: Precedence Rules

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