EE 231 Lab 5

Arithmetic Logic Unit

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.

1. Prelab

1.1. Fill out Table 1.

1.2. Write a Verilog program to implement the ALU.
### Table 1: ALU functions

<table>
<thead>
<tr>
<th>ALU_CTL</th>
<th>Mnemonic</th>
<th>Description</th>
</tr>
</thead>
</table>
| LOAD    | Load DATA into RESULT  
DATA => RESULT  
C is a don't care  
1 → Z if RESULT == 0, 0 → Z otherwise |
| ADDA    | Add DATA to ACCA  
ACCA + DATA => RESULT  
C is carry from addition  
1 → Z if RESULT == 0, 0 → Z otherwise |
| SUBA    | Subtract DATA from ACCA  
ACCA - DATA => RESULT  
C is borrow from subtraction  
1 → Z if RESULT == 0, 0 → Z otherwise |
| ANDA    | Logical AND DATA with ACCA  
ACCA & DATA => RESULT  
C is a don't care  
1 → Z if RESULT == 0, 0 → Z otherwise |
| ORAA    | Logical OR DATA WITH ACCA  
ACCA | DATA => RESULT  
C is a don't care  
1 → Z if RESULT == 0, 0 → Z otherwise |
| COMA    | Compliment of ACCA  
ACCA => RESULT  
1 → C  
1 → Z if RESULT == 0, 0 → Z otherwise |
| INCA    | Increment ACCA by 1  
ACCA + 1 = RESULT  
C is a don't care  
1 → if RESULT == 0, 0 → Z otherwise |
| LSRA    | Logical shift right of ACCA  
Shift all bits of ACCA one place to the right:  
0 → RESULT[7], ACCA[7:1] → RESULT[6:0]  
ACCA[0] → C  
1 → Z if RESULT == 0, 0 → Z otherwise |
| LSLA    | Logical shift left of ACCA  
Shift all bits of ACCA one place to the left:  
0 → RESULT[0], ACCA[6:0] → RESULT[7:1]  
ACCA[7] → C  
1 → Z if RESULT == 0, 0 → Z otherwise |
| ASRA    | Arithmetic shift right of ACCA  
Shift all bits of ACCA one place to the right:  
ACCA[0] → RESULT[7], ACCA[7:1] → RESULT[6:0]  
ACCA[0] → C  
1 → Z if RESULT == 0, 0 → Z otherwise |

2. **Lab**

2.1 Design the ALU using Verilog. **Make sure you deal with any unused bit combinations of the ALUT_CTL lines.**
2.2 Simulate the ALU and test different combinations of DATA and ACCA.

2.3 Program your ALU code into your FPGA

2.4 Create another program that will call your ALU module. In this module read external inputs for ACCA and DATA as well as the ALU_CTL. Output your results on two 7-segment displays (the pinout for the GPIO-0 is shown in Figure 2).

Figure 2: Pinout for GPIO-0 expansion area for the DE0-NANO

3. Supplementary Material

Verilog

3.1 Parameters

Parameters are constants and not variables:

\[\text{parameter num} = 8;\]

3.2 Operators

3.2.1 ?: Construct
assign\ y = \text{sel}\?a\!?:b;\\
\text{if\ } sel\ \text{is true, then}\ y\ \text{is assigned to}\ a, \text{else it is assigned} \ b.\\

3.2.2 Concatenation

In Verilog it is possible to concatenate bits using \{\}:\\

\{a, b, c, a, b, c\}\\
is equivalent to:\n
\{2\{a, b, c\}\}\n
3.2.3 Comparison Operators

assign\ y = a\!>\!b?a\!?:b;\\

assign\ y\ \text{if}\ a > b\ \text{and assign it to} \ b\ \text{otherwise. Table 2 shows a list of comparison operators.}\\

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Operator} & \textbf{Description} \\
\hline
\textgreater & Greater than \\
\textless & Less than \\
\geq & Greater than or equal to \\
\leq & Less than or equal to \\
\textgreater\textless & Equality \\
\textless\textless & Equality including x and z \\
\text{!} & Inequality \\
\text{!}\textless\textless & Inequality including x and z \\
\hline
\end{tabular}
\caption{Comparison Operators}
\end{table}

\begin{itemize}
\item For == and != the result is x, if either operand contains an x or z.
\item Evaluation is performed left to right.
\item x if any of the operand has unknown x bits
\end{itemize}

3.2.4 Logical Operators

\begin{table}[h]
\centering
\begin{tabular}{|c|c|}
\hline
\textbf{Operator} & \textbf{Description} \\
\hline
\text{!} & Logical negation \\
\&\& & Logical AND \\
\text{|} & Logical OR \\
\hline
\end{tabular}
\caption{Logical Operators}
\end{table}
3.2.5 Binary Arithmetic Operators

**Table 4:** Binary Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>+</td>
<td>Addition</td>
</tr>
<tr>
<td>−</td>
<td>Subtraction</td>
</tr>
<tr>
<td>*</td>
<td>Multiplication</td>
</tr>
<tr>
<td>/</td>
<td>Division (truncates any fractional part)</td>
</tr>
<tr>
<td>%</td>
<td>equality</td>
</tr>
</tbody>
</table>

3.2.6 Unary Arithmetic Operators

**Table 5:** Unary Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-</td>
<td>Change the sign of the operand</td>
</tr>
</tbody>
</table>

3.2.7 Bitwise Operators

**Table 6:** Bitwise Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

3.2.8 Unary Reduction Operators: Produce a single bit result by applying the operator to all of the bits of the operand.

**Table 7:** Unary Arithmetic Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>~</td>
<td>Bitwise negation</td>
</tr>
<tr>
<td>&amp;</td>
<td>Bitwise AND</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>~&amp;</td>
<td>Bitwise NAND</td>
</tr>
<tr>
<td>~</td>
<td></td>
</tr>
</tbody>
</table>

~^ or ^~ (equivalent)
3.2.9 Shift Operators

Table 8: Shift Operators

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;&lt;</td>
<td>Left shift</td>
</tr>
<tr>
<td>&gt;&gt;</td>
<td>Right shift</td>
</tr>
</tbody>
</table>

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

3.2.10 Operator Precedence Rule

Table 9: Precedence Rules

<table>
<thead>
<tr>
<th>Operator</th>
<th>Highest Precedence</th>
</tr>
</thead>
<tbody>
<tr>
<td>!, ~</td>
<td></td>
</tr>
<tr>
<td>* , / , %</td>
<td></td>
</tr>
<tr>
<td>+ , -</td>
<td></td>
</tr>
<tr>
<td>&lt;&lt; , &gt;&gt;</td>
<td></td>
</tr>
<tr>
<td>&lt;= , &gt;=</td>
<td></td>
</tr>
<tr>
<td>+= , -= , == , !=</td>
<td></td>
</tr>
<tr>
<td>&amp;</td>
<td></td>
</tr>
<tr>
<td>^ , ^ ~</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>&amp;&amp;</td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>?:</td>
<td>Lowest Precedence</td>
</tr>
</tbody>
</table>

Program 1: Example of an 8-bit Adder

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

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