CS232 Week3 Q2

Vedang Dhirendra Asgaonkar 200050154

January 2022

1 Design Overview

The ALU is designed with separate components for each arithmetic and logical operation. A demultiplexer takes the selection *sel* as input, and passes the 4 bit input signals a, b to one of the 8 components (operations). The component produces an 8 bit output signal. A multiplexer uses *sel* to choose between the output signals of the 8 components (since some components may have a default voltage level even when the input signal does not reach them) and passes it the 8 bit output signal *result*.

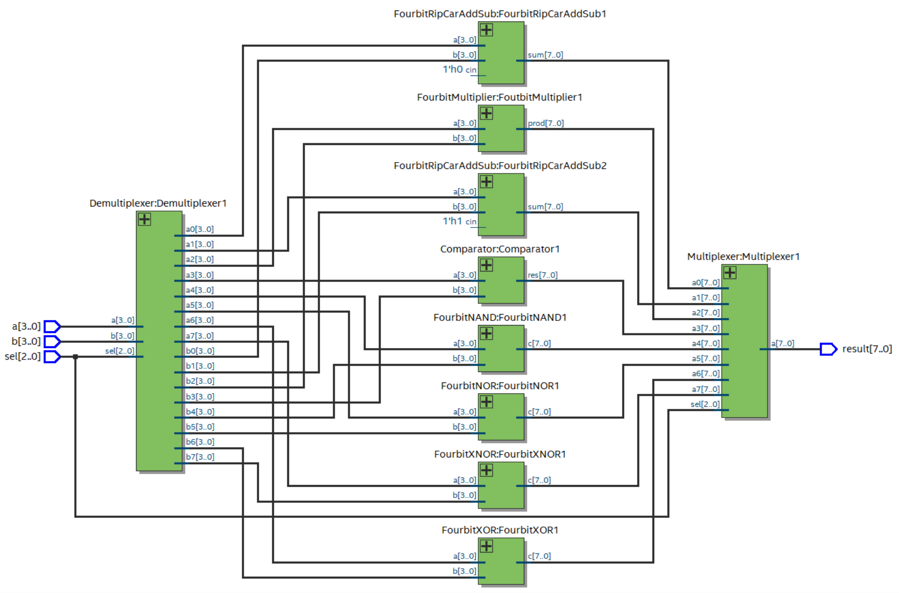

Figure 1: Design overview

2 Component Design

2.1 Demultiplexer

It takes as input a, b, two 4 bit vectors and a 3 bit *sel* selection. It has 16 output terminals, 8 for a and 8 for b. Each terminal is a 4 bit vector. It transfers the signals a and b to the correct output pin and keeps other outputs 0000.

For any particular output terminal a_i, b_i, it checks whether *sel* is equal to i. It first applies xor to $\overline{sel}$ and i, and then applies AND to the three bits produced to get output x_i. Then it assigns $a_i = a \cdot x_i$, $b_i = b \cdot x_i$

2.2 Multiplexer

It takes as input 8 8-bit vectors a_i and the selection *sel* as input and gives one 8-bit output a depending on the selection. For any particular input terminal a_i, it checks whether *sel* is equal to i. It first applies xor to $\overline{sel}$ and i, and then applies AND to the three bits produced to get output x_i. Then it assigns $c_i = a \cdot x_i$. The output a is or over all the c_i's.

2.3 Four bit adder/subtractor

We first design a four bit ripple adder. For each position in the sum, we add together the corresponding positions from a, b and the carry from the previous addition using the one bit full adder.

$$\begin{aligned}
sum(0), carry(0) &= \text{OneBitFullAdd}(a(0), b(0), cin) \\
sum(1), carry(1) &= \text{OneBitFullAdd}(a(1), b(1), carry(0)) \\
sum(2), carry(2) &= \text{OneBitFullAdd}(a(2), b(2), carry(1)) \\
sum(3), cout &= \text{OneBitFullAdd}(a(3), b(3), carry(2))
\end{aligned}$$

In case of subtraction, we note the following:

$$\begin{aligned}
&a - b \quad mod\ 2^n \\
&= a - b + 2^n \quad mod\ 2^n \\
&= a + (2^n - 1 - b) + 1 \quad mod\ 2^n \\
&= a + \bar{b} + 1 \quad mod\ 2^n \quad (*)
\end{aligned}$$

where $\bar{b}$ is produced by inverting all bits of b. Thus for addition, we consider a signal $b_2 = b$ and for subtraction $b_2 = \bar{b}$. Thus, if $cin = 0$ for addition and $cin = 1$ for subtraction

$$b_2(i) = b(i) \oplus cin \quad \forall\ i$$

Now, we add b_2 and a using the Four Bit Ripple Adder, taking the carry bit as 0 for addition and 1 for subtraction, thus accounting for the +1 in $(*)$.

$$sum, cout = \text{FourBitRippleAdder}(a, b, cin)$$

The *sum* is concatenated to the *cout* to get a five bit output.

2.4 Four bit Multiplier

It first calculates the four partial products of a with each bit of b. This is done by using AND gate. Then, carry-save addition is used to add these suitably shifted partial products to get the final result. During this process, partial sums and carries are produced which are stored in signals and added using one bit full adder. Finally in the result we need to use full adders at the bits 4,5,6,7 to add the partial sums with the previous carries. The carry of the full adder at bit 7 is stored in bit 8.

2.5 Comparator

The comparator traverses the bits of a and b from left to right comparing them. Thus,

$$\begin{aligned} p = a < b \text{ if } & (a(3).\overline{b(3)}) \\ & + ((a(3) = b(3)).(a(2).\overline{b(2)})) \\ & + ((a(3) = b(3)).(a(2) = b(2)).(a(1).\overline{b(1)})) \\ & + ((a(3) = b(3)).(a(2) = b(2)).(a(1) = b(1)).(a(0).\overline{b(0)})) \end{aligned}$$

$$\begin{aligned} q = b < a \text{ if } & (b(3).\overline{a(3)}) \\ & + ((a(3) = b(3)).(b(2).\overline{a(2)})) \\ & + ((a(3) = b(3)).(a(2) = b(2)).(b(1).\overline{a(1)})) \\ & + ((a(3) = b(3)).(a(2) = b(2)).(a(1) = b(1)).(b(0).\overline{a(0)})) \end{aligned}$$

If none of these is true, then $a = b$.

The outputs of these are computations are stored in two boolean variables p, q. Then if the result vector is res, we have

$$\begin{aligned} res(0) &= p.\overline{q} \\ res(1) &= \overline{p}.\overline{q} \\ res(2) &= \overline{p}.q \end{aligned}$$

2.6 Bitwise NAND, NOR, XOR, XNAND

The single bit NAND, NOR, XOR, XNAND gates are designed using AND, OR and NOT gates. Then, each bit of the result is computed by applying these operations on the corresponding bits of the inputs a, b.