Assignment-4 (CS 232) by Utkarsh Ranjan

8-bit 8x1 Multiplexer

Components
seven 8-bit 2x1 Multiplexer

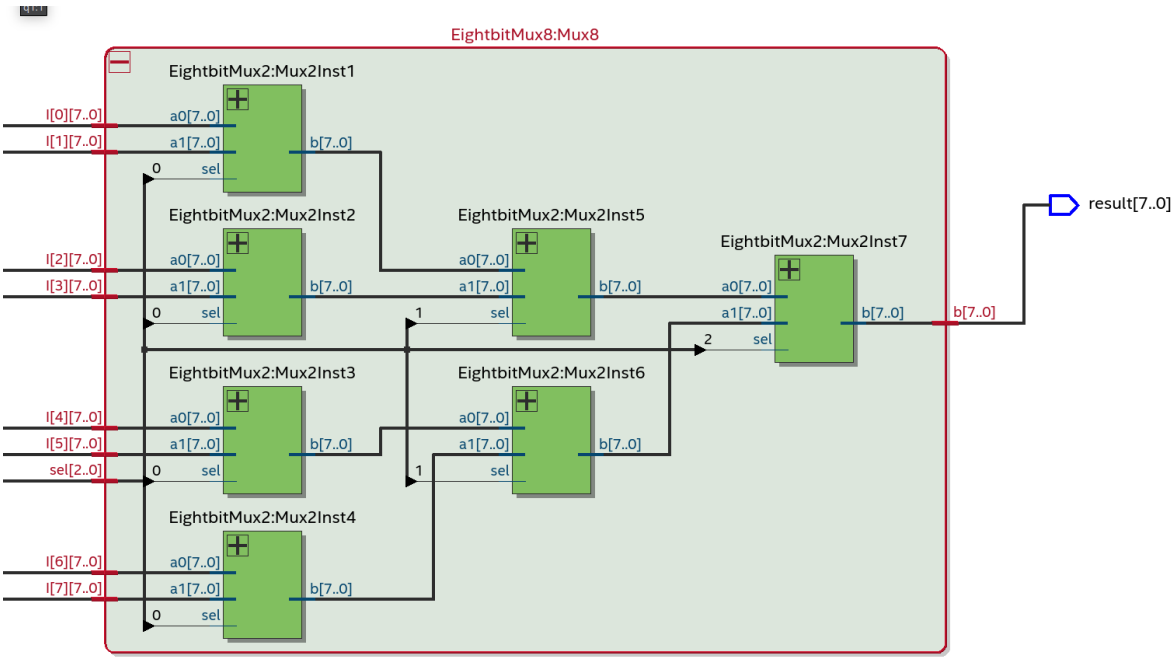
Description

- 8 Inputs :- Each 8-bit string
- 1 Output :- One 8-bit string
- The truth table for a 8x1 Multiplexer is as follows where each I_i is a 8-bit Signal.

S_2	S_1	S_0	Y
0	0	0	I_0
0	0	1	I_1
0	1	0	I_2
0	1	1	I_3
1	0	0	I_4
1	0	1	I_5
1	1	0	I_6
1	1	1	I_7

Idea

- The purpose of the multiplexer is to select one of the 8 inputs based on different values of the selector input
- To perform this, each 2x1 Multiplexer selected one singal out of two signals based on one-bit of the 3-bit selector input.
- 2x1 Multiplexers are arranged in the same way as matches in a Knockout Tournament.
- 1st round has 4 MUX, 2nd has 2 MUX , 3rd has 1 MUX. Each MUX knocking one of the eight inputs of the 8x1 Multiplexer



Carry-Look-Ahead Adder Circuit

Components

Three 4-bit XOR gate, Many AND gates, Many OR gates.

Description

- 3 Inputs: two 4-bit Strings (A , B) , one bit (Sel)
- 1 Output: one 8-bit String (Result)

Idea

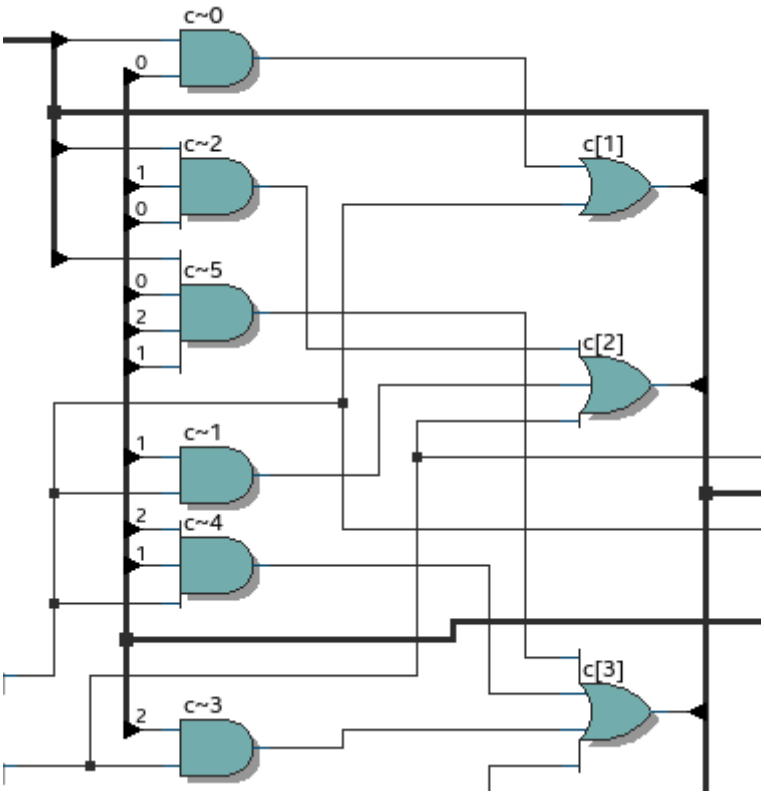
- We defined two variables 'Carry Generate (G_i)' and 'Carry Propagate (P_i)'
- The sum and carry bits are found using the following equations

$$P_i = A_i \oplus B_i$$
$$G_i = A_i B_i$$

$$C_1 = G_0 + P_0 C_{in}$$
$$C_1 = G_1 + P_1 G_0 + P_1 P_0 C_{in}$$
$$C_1 = G_2 + P_2 G_1 + P_2 P_1 G_0 + P_2 P_1 P_0 C_{in}$$
$$C_1 = G_3 + P_3 G_2 + P_3 P_2 G_1 + P_3 P_2 P_1 G_0 + P_3 P_2 P_1 P_0 C_{in}$$

$$S_i = P_i \oplus C_i$$

The subtractor Instance is made using '1' as the selector input while adder is made using '0' as the Selector input. This is hard-coded in the ALU file.



Array Multiplier Circuit

Components

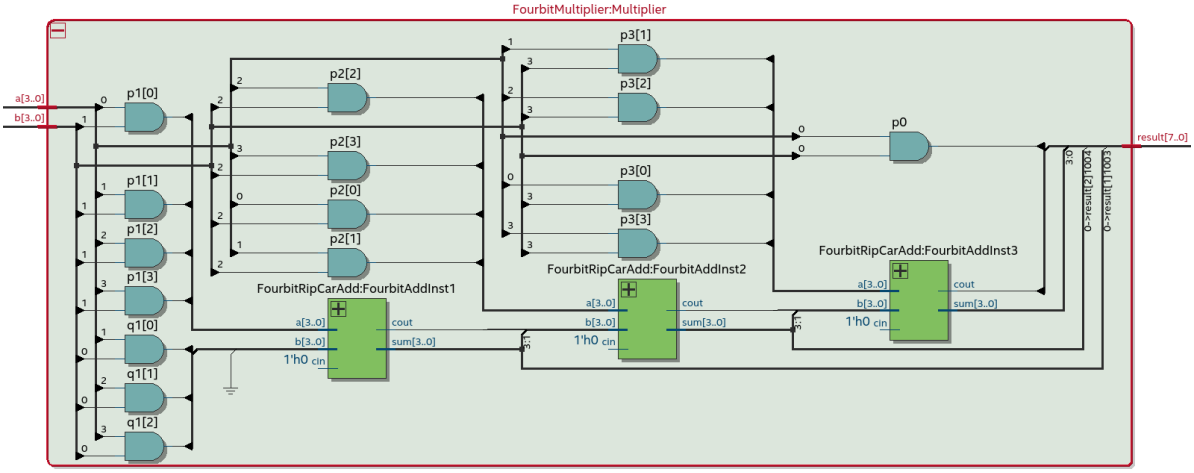
16 AND gate, *Three* 4-bit RIpple Carry Adder

Description

- 2 Inputs: two 4-bit Strings (A , B)
- 1 Output: one 8-bit String (Result)

Idea

- The idea is exactly similar to a mathematical multiplication
- Each bit of 'b' are ANDed with input 'a' and the ouput thus obtained are added to the previous sum using 4-bit ripple carry adder. The Cin of the adder is 0.



4-bit unsigned comparator

Components

16 AND gate, *Three* 4-bit RIpple Carry Adder

Description

- 2 Inputs: two 4-bit Strings (a, b)
- 1 Output: one 8-bit String (Result)

- a < b => result = xxxxx001
- a > b => result = xxxxx100
- a = b => result = xxxxx010

Idea

- The equal output is produced when all the individual bits of one number are exactly coincides with corresponding bits of another number. Then the logical expression for A=B output can be written as

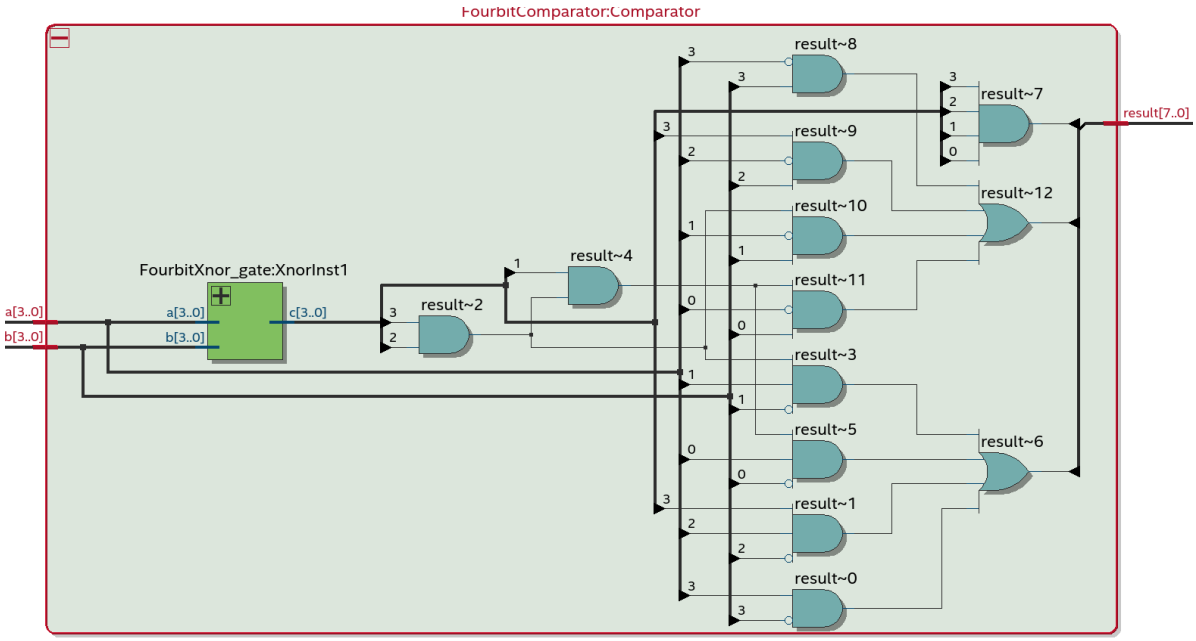
```
E = (A3 Ex-NOR B3) (A2 Ex-NOR B2) (A1 Ex-NOR B1) (A0 Ex-NOR B0)
```

- 1. If A3 = 1 and B3 = 0, then A is greater than B (A>B). Or
  2. If A3 and B3 are equal, and if A2 = 1 and B2 = 0, then A > B. Or
  3. If A3 and B3 are equal & A2 and B2 are equal, and if A1 = 1, and B1 = 0, then A>B. Or
  4. If A3 and B3 are equal, A2 and B2 are equal and A1 and B1 are equal, and if A0 = 1 and B0 = 0, then A > B.
- From the above statements, the output A > B logic expression can be written as

```
G = A3 B3' + (A3 Ex-NOR B3) A2 B2' +  (A3 Ex-NOR B3) (A2 Ex-NOR B2) A1 B1' +
(A3 Ex-NOR B3) (A2 Ex-NOR B2) (A1 Ex-NOR B1) A0 B0'
```

- Similarly the logic expression for the L or A < B output can be expressed as

```
L = A3' B3 + (A3 Ex-NOR B3) A2' B2 +  (A3 Ex-NOR B3) (A2 Ex-NOR B2) A1' B1 +
(A3 Ex-NOR B3) (A2 Ex-NOR B2) (A1 Ex-NOR B1) A0' B0
```



4-bit bitwise Operators

- Bitwise NAND, NOR, XOR, XNOR are designed in the same way as normal gates except singals (inputs and outputs) being std_logic_vector(3 downto 0) instead of std_logic

4-bit unsigned ALU

Components

one 8-bit 8x1 Multiplexer , *two* carry-look-ahead adder , *one* 4-bit unsigned multiplier , *one* 4-bit unsigned comparator , *one* 4-bit bitwise NAND gate , *one* 4-bit bitwise NOR gate , *one* 4-bit bitwise XOR gate , *one* 4-bit bitwise XNOR gate

Description

- 3 Inputs: 8-bit Strings (A,B), 3-bit String (sel)
- 1 Output: 8-bit String (result)
- A byte-array of eight elements is defined in a package, this is to handle eight singals each of which are 8-bit strings in itself.

Idea

- Once we have all the components, we can select one of the 8 output signals of each component using the 8x1 MUX.
- Inputs are fed to every component which performs different operations on the input.
- Then these 8 signals are passed through the 8x1 MUX along with the selector input to get the final output.