

VLSI LAB Experiment-3

Harshavardhan Alimi

18EC10021

1. 4:1 multiplexer using conditional operators

Aim :-

The aim of this experiment is to Implement a 4 to 1 multiplexer using Conditional Operator without using direct logic equation (&, | operators).

Schematic diagrams and labels :-

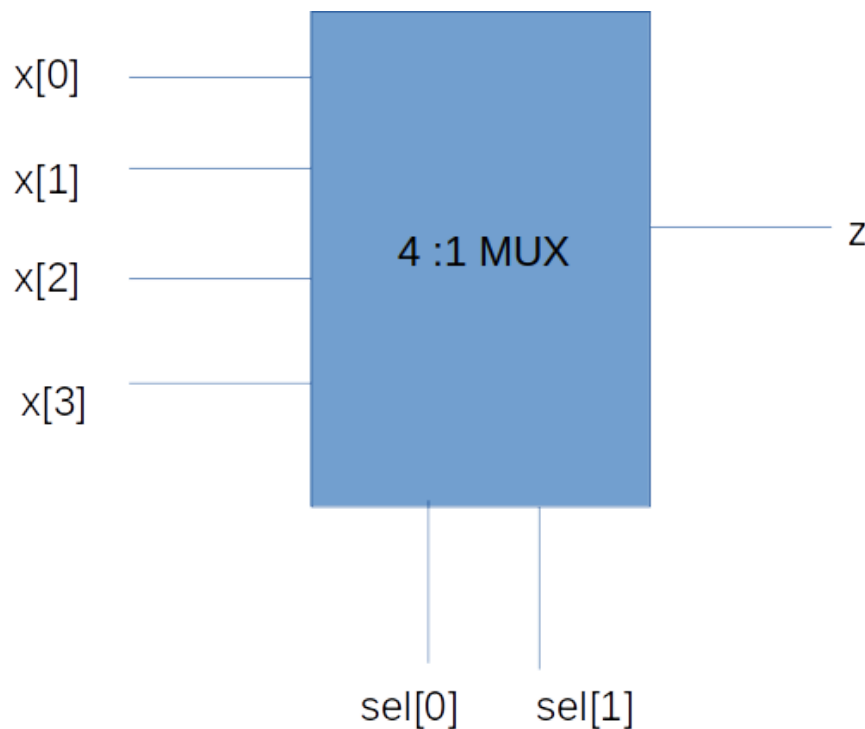


Fig.1:- Block diagram for implementation of 4:1 multiplexer

Description :-

- ❖ Below is the code for implementation of 4:1 multiplexer, In the code:-
- ❖ 4 - bit Input : x
- ❖ 2-bit Select line : sel.
- ❖ Output displayed : z
- ❖ One of these 4 inputs will be connected to the output based on the combination of inputs present at these two selection lines. Truth table of 4x1 Multiplexer is shown below.

select lines		output
sel[1]	sel[0]	z
0	0	x[0]
0	1	x[1]
1	0	x[2]
1	1	x[3]

- ❖ We have done a data flow modelling of the 4x1 MUX using conditional operators
 - $z = \text{sel}[1] ? (\text{sel}[0] ? x[3] : x[2]) : (\text{sel}[0] ? x[1] : x[0]) ;$

Verilog codes :-

```

module design_4_1_MUX (
    input [3:0] x,
    input [1:0] sel,
    output z
);

    assign z = sel[1] ? (sel[0] ? x[3] : x[2]) : (sel[0] ? x[1] : x[0]) ;
endmodule

module design_4_1_MUX_tb;

```

```
// Inputs
reg [3:0] x;
reg [1:0] sel;

// Outputs
wire z;

// Instantiate the Unit Under Test (UUT)
design_4_1_MUX uut (
    .x(x),
    .sel(sel),
    .z(z)
);

initial begin
    $dumpfile("test.vcd");
    $dumpvars(0,design_4_1_MUX_tb);
end

initial begin
    // Initialize Inputs
    #0 x = 4'b1101;sel=2'b00 ;
    #10 sel=2'b01 ;
    #10 sel=2'b10 ;
    #10 sel=2'b11 ;
    #10;
end

initial begin
    $monitor("t=%3d : x=%b,sel=%b,z=%b \n",$time,x,sel,z);
```

```
end  
endmodule
```

Runtime log :-

VCD info: dumpfile test.vcd opened for output.

t= 0 : x=1101,sel=00,z=1

t= 10 : x=1101,sel=01,z=0

t= 20 : x=1101,sel=10,z=1

t= 30 : x=1101,sel=11,z=1

Results :-



Discussion :-

- ❖ The gate level implementation of 4-to-1 MUX would require four 3-input AND gates and one 4-input OR gate.
- ❖ In some cases gate level implementation of 4-to-1 MUX, which is quite complex, is not required.
- ❖ In such cases we implement the circuit at an abstraction level using a data flow modelling scheme where we can just use conditional operators to implement the logic circuit.
- ❖ This logic level abstraction can then further be used as a structural block in implementation of more complex circuits.

2.Full adder - data flow modelling and carry look ahead

Aim :-

The aim of this experiment is To implement a 4-bit carry look ahead adder using data flow operators.

Schematic diagrams and labels :-

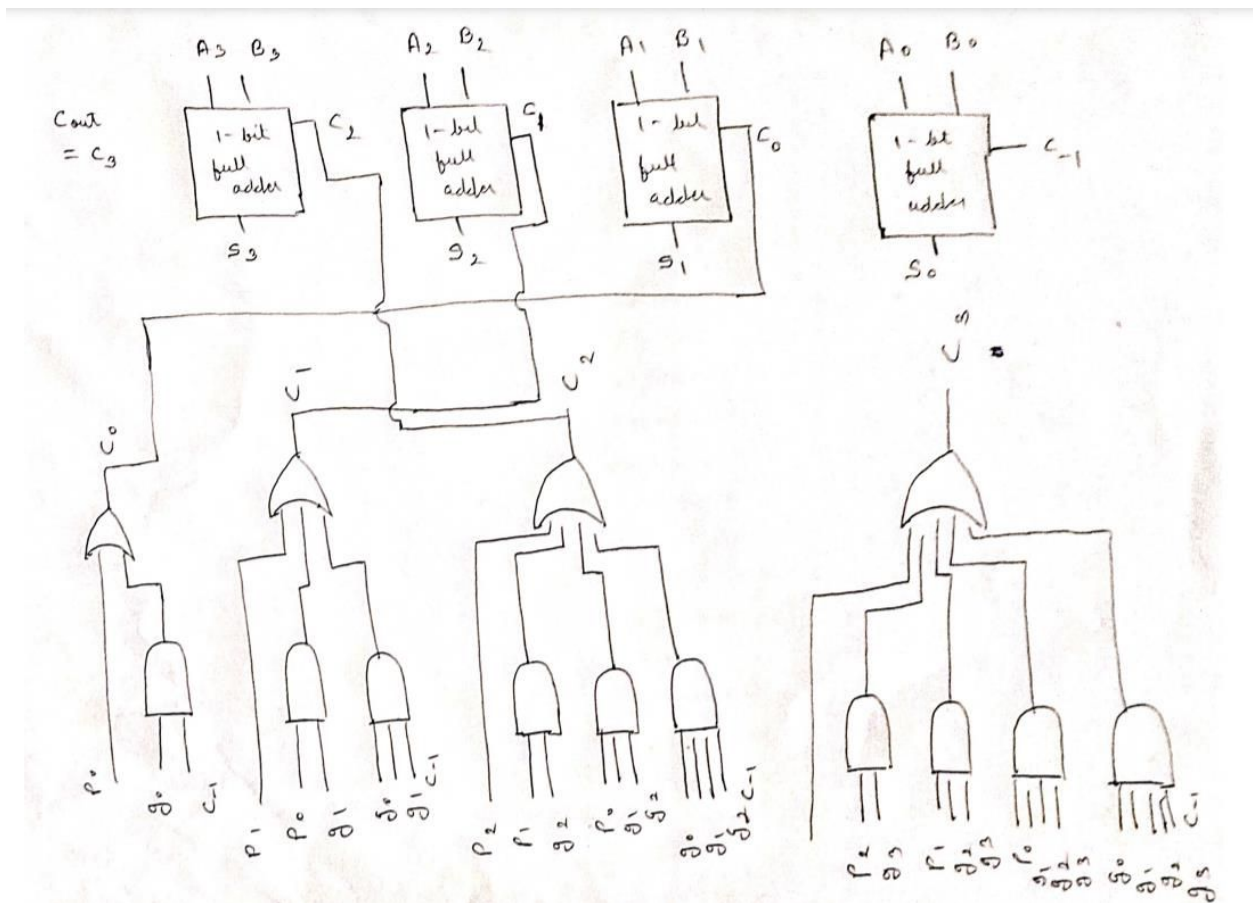


Fig2:- Block diagram for the implementation of carry lookahead full adder using data flow operators.

Description :-

- ❖ Below is the code for implementation of carry lookahead adder, In the code:-
- ❖ 4 - bit Input : a
- ❖ 4 - bit Input : b
- ❖ Carry in input : cin
- ❖ 4 - bit sum Output displayed : sum1 (direct implementation of 4-bit full adder in a single line in data flow model).
- ❖ Carry out bit : cout1 (direct implementation of 4-bit full adder in a single line in data flow model).
- ❖ 4 - bit sum Output displayed : sum2 (carry lookahead 4-bit full adder implementation).
- ❖ Carry out bit : cout2 (carry lookahead 4-bit full adder implementation).
- ❖ Data flow single line implementation of the full adder :-
 - `assign {cout1,sum1} = a + b + cin;`
- ❖ The circuit of the Carry Look Ahead (CLA) adder can be divided into two parts: one consisting the circuitry for carry generation and other for generating the sum.
- ❖ Assigning $p = a \oplus b$ (propagate term) and $g = a \& b$ (generate term)
- ❖ we can define carry $C[3:0]$ as:
 - $c[0] = cin$
 - $c[1] = g[0] \mid (p[0] \& cin)$
 - $c[2] = g[1] \mid (p[1] \& g[0]) \mid (p[1] \& p[0] \& cin)$
 - $c[3] = g[2] \mid (p[2] \& g[1]) \mid (p[2] \& p[1] \& g[0]) \mid (p[2] \& p[1] \& p[0] \& cin)$
- ❖ And carry out is defined as :
 - $cout = g[3] \mid (p[3] \& g[2]) \mid (p[3] \& p[2] \& g[1]) \mid (p[3] \& p[2] \& p[1] \& g[0]) \mid (p[3] \& p[2] \& p[1] \& p[0] \& cin);$
- ❖ The circuitry was implemented in data flow modelling scheme by using bitwise operators $\&$ for 'AND' and \mid for 'OR'.
- ❖ The circuitry generating sum was also generated following data flow modelling scheme is defined below :-
 - $sum = p \oplus c$

Verilog codes :-

```
module full_adder_data_flow (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] sum,
    output cout
);
    assign {cout,sum} = a + b + cin;
endmodule

module full_adder_carry_lookahead (
    input [3:0] a,
    input [3:0] b,
    input cin,
    output [3:0] sum,
    output cout
);
    wire [3:0] p;
    wire [3:0] g;
    wire [3:0] c;

    assign p = a ^ b; // propagate
    assign g = a & b; // generate

    assign c[0] = cin;

    assign c[1] = g[0] | (p[0] & cin);

    assign c[2] = g[1] | (p[1] & g[0]) | (p[1] & p[0] & cin);
```

```
    assign c[3] = g[2] | (p[2] & g[1]) | (p[2] & p[1] & g[0]) | (p[2] &
p[1] & p[0] & cin);

    assign cout = 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] & cin);

    assign sum = p ^ c;
endmodule

module full_adder_tb;

    // Inputs

    reg [3:0] a;

    reg [3:0] b;

    reg cin;

    // Outputs

    // full adder data flow(concatenation)

    wire [3:0] sum1;

    wire cout1;

    // full adder carry look ahead

    wire [3:0] sum2;

    wire cout2;

    // Instantiate the Unit Under Test (UUT) {for data flow full adder}

    full_adder_data_flow uut1 (

        .a(a),

        .b(b),

        .cin(cin),

        .sum(sum1),

        .cout(cout1)

    );
```



```
// Instantiate the Unit Under Test (UUT) {for carry look-ahead full
adder}

full_adder_carry_lookahead uut2 (

    .a(a),

    .b(b),

    .cin(cin),

    .sum(sum2),

    .cout(cout2)

);

initial begin

    $dumpfile("test.vcd");

    $dumpvars(0,full_adder_tb);

end

initial begin

    // Initialize Inputs

    #0  a=1011;b=0011;cin=0;

    #10 b=0100;

    #10 b=0101;

    #10 b=0110;

    #10 b=0111;

    #10 cin=1;b=0011;

    #10 b=0100;

    #10 b=0101;

    #10 b=0110;

    #10 b=0111;

    #10;
```

```

end

initial begin

    $monitor("\nt=%3d : a=%b | b=%b | cin=%b\nData flow full
adder:-\n\tsum=%b | cout=%b \nCarry look-ahead full adder:-\n\tsum=%b |
cout=%b ", $time, a, b, cin, sum1, cout1, sum2, cout2);

end

endmodule

```

Runtime log :-

VCD info: dumpfile test.vcd opened for output.

t= 0 : a=0011 | b=1011 | cin=0

Data flow full adder:-

sum=1110 | cout=0

Carry look-ahead full adder:-

sum=1110 | cout=0

t= 10 : a=0011 | b=0100 | cin=0

Data flow full adder:-

sum=0111 | cout=0

Carry look-ahead full adder:-

sum=0111 | cout=0


t= 20 : a=0011 | b=0101 | cin=0

Data flow full adder:-

sum=1000 | cout=0

Carry look-ahead full adder:-





sum=1000 | cout=0

t= 30 : a=0011 | b=1110 | cin=0

Data flow full adder:-

sum=0001 | cout=1

Carry look-ahead full adder:-

sum=0001 | cout=1

t= 40 : a=0011 | b=1111 | cin=0

Data flow full adder:-

sum=0010 | cout=1

Carry look-ahead full adder:-

sum=0010 | cout=1

t= 50 : a=0011 | b=1011 | cin=1

Data flow full adder:-

sum=1111 | cout=0

Carry look-ahead full adder:-

sum=1111 | cout=0

t= 60 : a=0011 | b=0100 | cin=1

Data flow full adder:-

sum=1000 | cout=0


Carry look-ahead full adder:-

sum=1000 | cout=0

t= 70 : a=0011 | b=0101 | cin=1

Data flow full adder:-

sum=1001 | cout=0



sum=1001 | cout=0

Data flow full adder:-

Carry look-ahead full adder:-

Data flow full adder:-

Carry look-ahead full adder:-

Results :-



Discussion :-

- ❖ In the second part of the experiment we implemented a carry look ahead (CLA) adder following data flow modelling scheme.
- ❖ In ripple carry adder (RCA), the 1-bit full adder waits for the carry generated from the previous 1-bit full adder to generate its output sum and carry bits. Thus we have a high propagation delay if the number of input bits are high. Alternatively we can say RCA has a time complexity of $O(n)$.
- ❖ A CLA adder on the other hand pre-computes all the carry bits of each stage and uses them to generate the output. Thus its time complexity is $O(1)$.
- ❖ But the complexity of the CLA circuit increases with increase in number of bits which is a main disadvantage.
- ❖ We implemented a 4-bit CLA adder by using data flow modelling to avoid the complex and gate level implementation but still we could observe its various characteristics.
- ❖ The main disadvantage of carry lookahead 4-bit full adder is the number of gates used in implementation(uses high number of gates).

Drive links for verilog codes :-

[Part_A](#)

[Part_B](#)