VLSI Engineering Lab (Digital)

Experiment 3

Utkarsh Patel (18EC30048)

1 Objective

- Implementing 4:1 MUX using only conditional operator
- Implementing 4-bit full adder
    - using concatenation operator (ripple carry adder)
    - with carry lookahead

2 Block Diagram

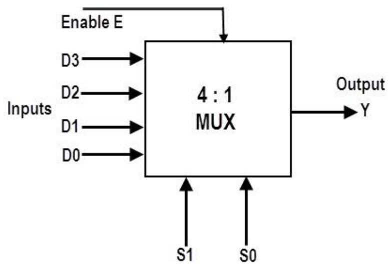


Fig. 1. ***Block diagram of 4:1 MUX:*** The circuits takes as input 4-bit integer `D[3:0]`, 2-bit select line `S[1:0]` (that control which bit will be passed on), and produces the desired output `Y`. Enable `E` is used for switching on / off the 4:1 MUX

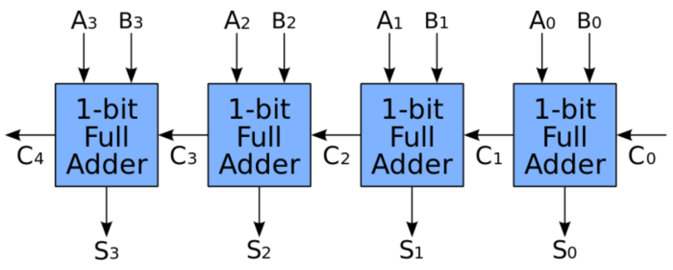


Fig. 2. ***Block diagram of 4-bit ripple carry adder:*** The circuits takes as input two 4-bit integer `A[3:0]` and `B[3:0]`, input carry `C0`, and produces the sum `S[3:0]` and output carry `C4`

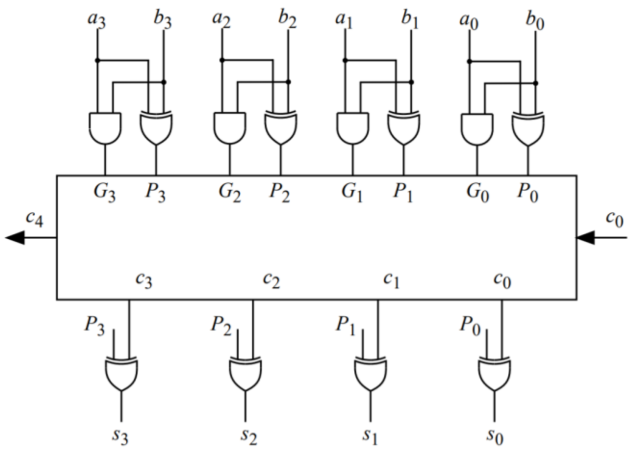


Fig. 3. ***Block diagram of 4-bit carry lookahead adder:*** The circuits takes as input two 4-bit integer `A[3:0]` and `B[3:0]`, input carry `C0`, and produces the sum `S[3:0]` and output carry `C4`. The block *B cell* generates propagation and generation bits, and *Carry-lookahead* block produces carry bits from propagation and generation bits.

3 Theory

3.1 4:1 Multiplexer

A 4:1 multiplexer consists four data input lines as D_0 to D_3, two select lines as S_0 and S_1 and a single output line Y. The particular input combination on select lines selects one of input (D_0 through D_3) to the output.

Select Data Inputs		Output
S_1	S_0	Y
0	0	D_0
0	1	D_1
1	0	D_2
1	1	D_3

Fig. 4. Truth table of 4:1 MUX

The truth table of 4:1 MUX is shown above. It can be observed that the output Y is given as

$$Y = \overline{S_1}\,\overline{S_0}D_0 + \overline{S_1}S_0D_1 + S_1\overline{S_0}D_2 + S_1S_0D_3$$

3.2 Concatenation operator in Verilog

Multi-bit Verilog wires and variables can be clubbed together to form a bigger multi-net wire or variable using concatenation operators { and } separated by commas. Concatenation is also allowed to have expressions and sized constants as operands in addition of wires and variables. Size of each operand must be known in order to calculate the complete size of concatenation.

This method is used to implement the ripple carry adder in this experiment as

```
assign {c_out, sum} = a + b + c_in;
```

where size of `c_out` and `sum` are known *a-priori*.

3.3 Ripple carry adder

The carry lookahead adder (CLA) solves the carry delay problem by calculating the carry signals in advance, based on the input signals. It is based on the fact that a carry signal will be generated in two cases: (1) when both bits a_i and b_i are 1 or (2) when one of the two bits is 1 and the carry-in is 1. Thus, one can write,

$$c_{i+1} = a_i \cdot b_i + (a_i \oplus b_i) \cdot c_i$$

$$s_i = a_i \oplus b_i \oplus c_i$$

The above two equations can be written in terms of two new signals P_i and G_i, which are shown in Fig. 5:

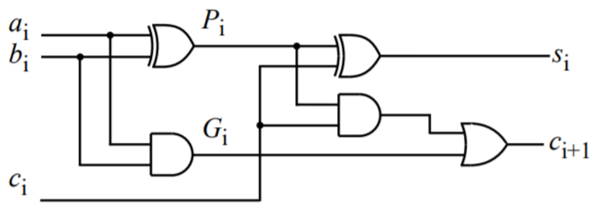


Fig. 5. Full adder at stage i with P_i and G_i shown

$$c_{i+1} = G_i + P_i \cdot c_i$$

$$s_i = P_i \oplus c_i$$

P_i and G_i are called the carry propagate and carry generate terms, respectively. Notice that the generate and propagate terms only depend on the input bits and thus will be valid after one and two gate delay, respectively. If one uses the above expression to calculate the carry signals, one does not need to wait for the carry to ripple through all the previous stages to find its proper value. For a 4-bit adder, we can have

$$\begin{aligned}
c_1 &= G_0 + P_0.c_0 \\
c_2 &= G_1 + P_1.G_0 + P_1.P_0.c_0 \\
c_3 &= G_2 + P_2.G_1 + P_2.P_1.G_0 + P_2.P_1.P_0.c_0 \\
c_4 &= G_3 + P_3.G_2 + P_3.P_2.G_1 + P_3.P_2.P_1.G_0 + P_3.P_2.P_1.P_0.c_0
\end{aligned}$$

The disadvantage of CLA is that the carry logic block gets very complicated for more than 4-bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4-bits.

4 Verilog codes

4.1 4:1 MUX using conditional operator

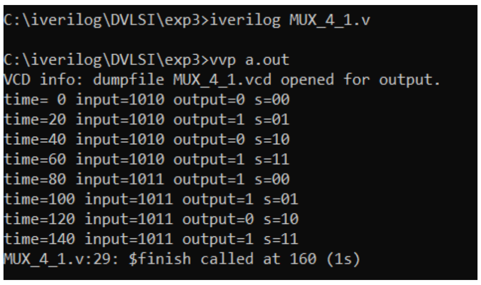
```
// 4:1 MUX using conditional operators only
module MUX_4_1(input[1:0] S, input[3:0] I, output Y);
  assign Y = (S == 2'b00) ? I[0] :
         (S == 2'b01) ? I[1] :
         (S == 2'b10) ? I[2] :
         (S == 2'b11) ? I[3] : 1'bx;
endmodule

module MUX_4_1_tb();
  reg [1:0]select;
  reg [3:0]in;
  wire out;
  MUX_4_1 mux(select,in,out);
  initial begin
    $monitor("time=%2d input=%b output=%b s=%b ",$time,in,out,select);
    #0;in=4'b1010; select=2'b00;
    #20;select=2'b01;
    #20;select=2'b10;
    #20;select=2'b11;
    #20;in=4'b1011; select=2'b00;
    #20;select=2'b01;
    #20;select=2'b10;
    #20;select=2'b11;
    #20;$finish;
  end
  initial begin
    $dumpfile("MUX_4_1.vcd");
    $dumpvars;
  end
endmodule
```

4.2 4-bit adder

```
// 4-bit ripple carry adder module using concatenation operator
module ripple_adder(input [3:0] a,input [3:0] b, input c_in, output [3:0] sum, output c_out);
  assign {c_out, sum} = a + b + c_in;  //concatenation of c_out and sum
endmodule

// 4-bit carry lookahead adder module
module carry_look_ahead(input [3:0] a,input [3:0] b,input cin, output[3:0] sum, output cout);
  wire [3:0] p,g,c;
  assign p=a^b; //propagate
  assign g=a&b; //generate
  assign c[0]=cin;
  assign c[1]= g[0]|(p[0]&c[0]);
  assign c[2]= g[1] | (p[1]&g[0]) | p[1]&p[0]&c[0];
  assign c[3]= g[2] | (p[2]&g[1]) | p[2]&p[1]&g[0] | p[2]&p[1]&p[0]&c[0];
  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]&c[0];
  assign sum=p^c;
endmodule

module adder_tb();
  reg[3:0] A;
  reg[3:0] B;
  reg Cin;
  wire[3:0] Sum_0;
  wire[3:0] Sum_1;
  wire Cout_0;
  wire Cout_1;
  ripple_adder adder_0 (A,B,Cin,Sum_0,Cout_0);
  carry_look_ahead adder_1(A,B,Cin,Sum_1,Cout_1);
  initial begin
    $monitor(" Time=%5d, A=%b, B=%b Cin=%b\n \t Cout_0=%b Sum_0=%b\n \t Cout_1=%b Sum_1=%b\n", $time,
A,B,Cin,Cout_0,Sum_0,Cout_1,Sum_1);
    // Initialize Inputs
    #0; A = 4'b0111 ; B = 4'b0111; Cin = 0;
    #10; A = 4'b0111 ; B = 4'b1111; Cin = 1;
    #20; A = 4'b0111 ; B = 4'b0001; Cin = 0;
    #40 $finish;
  end

  initial begin
    $dumpfile("adder.vcd");
    $dumpvars;
  end

endmodule
```

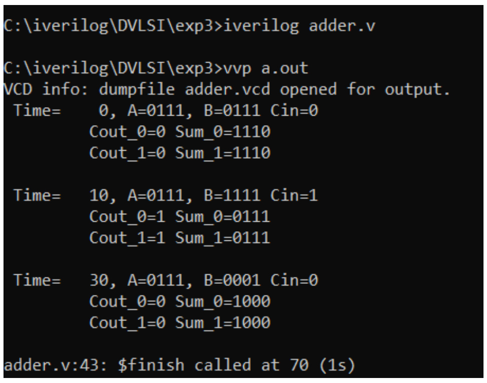
5 Output Log

5.1 4:1 MUX using conditional operator



```
C:\iverilog\DVLSI\exp3>iverilog MUX_4_1.v

C:\iverilog\DVLSI\exp3>vvp a.out
VCD info: dumpfile MUX_4_1.vcd opened for output.
time= 0 input=1010 output=0 s=00
time=20 input=1010 output=1 s=01
time=40 input=1010 output=0 s=10
time=60 input=1010 output=1 s=11
time=80 input=1011 output=1 s=00
time=100 input=1011 output=1 s=01
time=120 input=1011 output=0 s=10
time=140 input=1011 output=1 s=11
MUX_4_1.v:29: $finish called at 160 (1s)
```

5.2 4-bit adder



```
C:\iverilog\DVLSI\exp3>iverilog adder.v

C:\iverilog\DVLSI\exp3>vvp a.out
VCD info: dumpfile adder.vcd opened for output.
 Time=    0, A=0111, B=0111 Cin=0
         Cout_0=0 Sum_0=1110
         Cout_1=0 Sum_1=1110

 Time=   10, A=0111, B=1111 Cin=1
         Cout_0=1 Sum_0=0111
         Cout_1=1 Sum_1=0111

 Time=   30, A=0111, B=0001 Cin=0
         Cout_0=0 Sum_0=1000
         Cout_1=0 Sum_1=1000

adder.v:43: $finish called at 70 (1s)
```

6 Waveforms

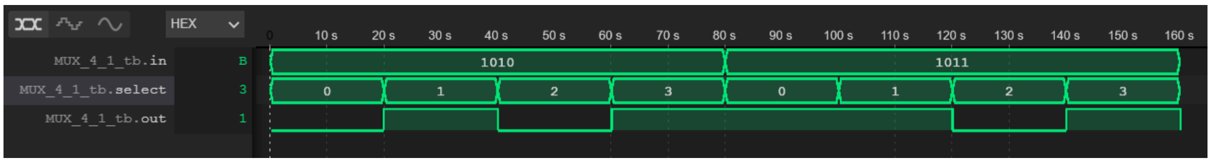


Fig. 6. ***Waveforms for 4:1 MUX:*** The circuits takes as input 4-bit integer `in[3:0]`, 2-bit select line `s[1:0]` (that control which bit will be passed on), and produces the desired output `out`.

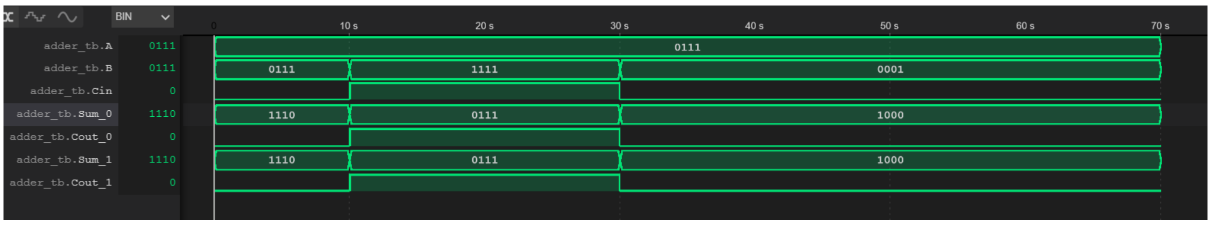


Fig. 7. ***Waveforms for 4-bit adder:*** The circuits takes as input two 4-bit integer `A[3:0]`and `B[3:0]`, input carry `Cin`, and produces the sum `Sum[3:0]` and output carry `Cout`. `Sum_0` and `Cout_0` are produced from ripple carry adder and Sum_`1` and `Cout_1` are produced from carry lookahead adder.

7 Conclusion

- In this experiment, a 4:1 MUX (Fig. 1) is designed using only conditional operator.
- A 4:1 multiplexer consists four data input lines as D_0 to D_3, two select lines as S_0 and S_1 and a single output line Y. The particular input combination on select lines selects one of input (D_0 through D_3) to the output. See Fig. 4 for the truth table.
- The output of the MUX is given by $Y = \overline{S_1}\,\overline{S_0}D_0 + \overline{S_1}S_0D_1 + S_1\overline{S_0}D_2 + S_1S_0D_3$ which can be easily implemented using conditional operator.
- From Fig. 6, it can be observed that the 4:1 MUX module is operating properly.
- Later in the experiment, 4-bit adder (Fig. 2) is designed using Verilog concatenation operator.
- Multi-bit Verilog wires and variables can be clubbed together to form a bigger multi-net wire or variable using concatenation operators { and } separated by commas. Concatenation is also allowed to have expressions and sized constants as operands in addition of wires and variables. Size of each operand must be known in order to calculate the complete size of concatenation.
- 4-bit carry-lookahead adder (Fig 3) is also designed at last.
- Carry-lookahead adder precomputes the carry bits from the carry generate and carry propagate terms as discussed in theory, which makes it substantially faster than ripple carry adders.
- The disadvantage of CLA is that the carry logic block gets very complicated for more than 4-bits. For that reason, CLAs are usually implemented as 4-bit modules and are used in a hierarchical structure to realize adders that have multiples of 4-bits.
- From Fig. 7, it can be observed that both the adder modules are implemented correctly.