DDCO Mini Project ISA Submission

Project Title: 8 Bit Prefix Adder
Section : A
Batch Details:

Student Name	SRN
Adithya Prakash	PES1UG19CS029
Aditi D Anchan	PES1UG19CS030
Aditi Killedar	PES1UG19CS031
Aditya NG	PES1UG19CS032

Circuit Diagram:

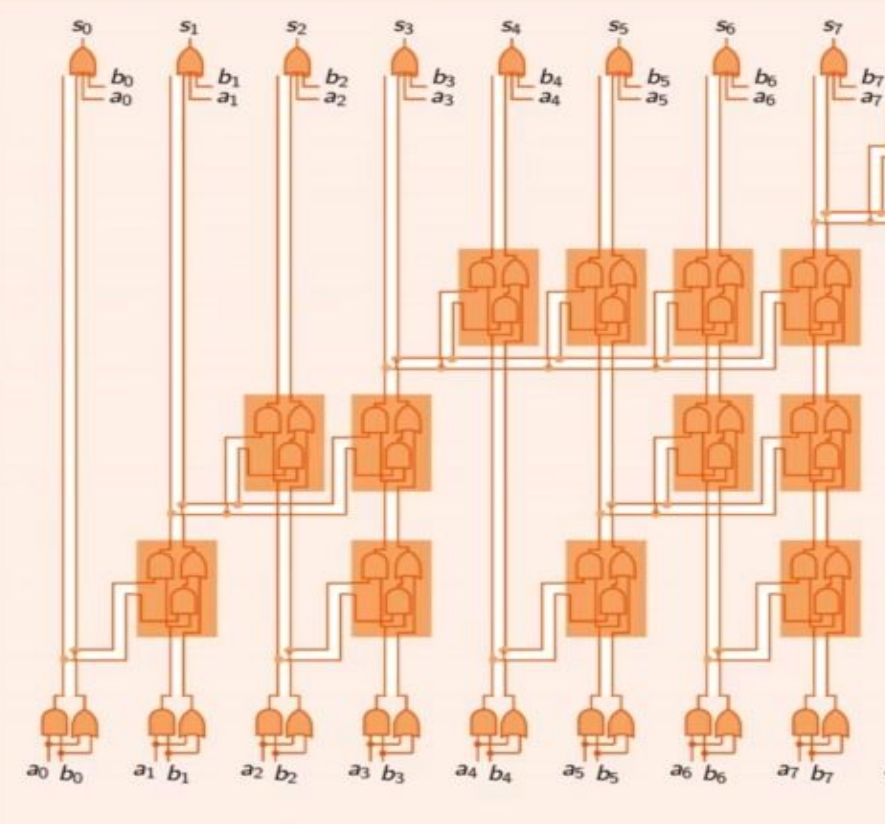


Figure 1. Circuit Diagram

Algorithm:

Define an operation $\otimes$ such that :

$(p_i, g_i) \otimes (p_j, g_j) = (g_i + p_i \cdot g_j, p_i \cdot p_j)$

It is possible to show that $\otimes$ is associative.

$((p_i, g_i) \otimes (p_j, g_j)) \otimes (p_j, g_j) = (p_i, g_i) \otimes ((p_j, g_j) \otimes (p_j, g_j))$

Consider the module shown to perform this operation $\otimes$ in Figure 2. Module

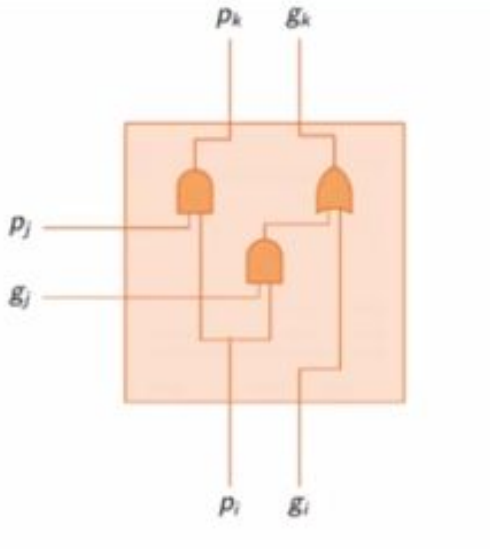


Figure 2. Module

For an 8 Bit Parallel Prefix Adder, consider 12 such modules such that :

1. 4 Modules L1a, L1b, L1c, L1d on layer 1 acting on the pairs (0,1), (2,3), (4,5), (6,7)
2. 4 Modules L2a, L2b, L2c, L2d on layer 2 acting on the pairs (L1a,2), (L1a,L1b), (L1c,6), (L1c,L1d)
3. 4 Modules L3a, L3b, L3c, L3d on layer 3 acting on the pairs (L2b,4), (L2b,L1c), (L2b,L2c), (L2b,L2d)
4. Outputs of all lines are xored with the two corresponding input bits

Refer to Figure 1. Circuit Diagram for the above implementation

Implementation (Code):

//in padder.v

```
module and_or2(input wire a, b, c, x, y, output wire d, e);
      and2 and2_d(a, b, t);
      or2 or2_d(t, c, d);
      and2 and2_e(x, y, e);
endmodule

module prefixAdder( input [7:0] a, input [7:0] b, output [8:0]
sum);
      wire [15:0] p;
      wire [15:0] g;
      wire [15:0] cp1;
      wire [15:0] cg1;
      wire cp22, cg22;
      wire cp23, cg23;
      wire cp27, cg27;
      wire cp34, cg34;
      wire cp35, cg35;
      wire cp36, cg36;
      wire cp37, cg37;
      wire cp211, cg211;
      wire cp215, cg215;
      wire cp315, cg315;
      wire cp411, cg411;
      wire cp415, cg415;
      wire cp59, cg59;
      wire cp513, cg513;

      wire [8:0] s;
      // start first level
      xor2 xor2_p0(a[0], b[0], p[0]);

      and2 a2(a[0], b[0], g[0]); //assign g[0] = a[0] & b[0];
      assign cp1[0] = p[0];
      assign cg1[0] = g[0]; //first level zero bit complete

      xor2 xor2_p1(a[1], b[1], p[1]); // assign p[1] = a[1] ^ b[1];
//first level first bit start
      and2 and2_g1(a[1], b[1], g[1]); // assign g[1] = a[1] & b[1];
      and_or2 and_or_cg1_1(p[1], g[0], g[1], p[1], g[0], cg1[1],
cp1[1]); // first level first bit complete

      xor2 xor2_p2(a[2], b[2], p[2]); // assign p[2] = a[2] ^ b[2];
//first level 2nd bit start
      and2 and2_g2(a[2], b[2], g[2]); // assign g[2] = a[2] & b[2];
      assign cp1[2] = p[2];
      assign cg1[2] = g[2]; // first level 2nd bit complete

      xor2 xor2_p3(a[3], b[3], p[3]); //first level 3rd bit start
      and2 and2_g3(a[3], b[3], g[3]);
      and_or2 and_or_cg1_3(p[3], g[2], g[3], p[3], p[2], cg1[3],
cp1[3]);
      //first level 3rd bit complete
```

```
     xor2 xpr2_p4(a[4], b[4], p[4]); //4th bit
     and2 and2_g4(a[4], b[4], g[4]);
     assign cg1[4] = g[4];
     assign cp1[4] = p[4];

     xor2 xor2_p5(a[5], b[5], p[5]); //5th bit start
     and2 and2_g5(a[5], b[5], g[5]);
     and_or2 and_or_cg1_5(p[5], g[4], g[5], p[5], p[4], cg1[5],
cp1[5]);

     xor2 xor2_p6(a[6], b[6], p[6]); //6th bit
     and2 and2_g6(a[6], b[6], g[6]);
     assign cg1[6] = g[6];
     assign cp1[6] = p[6];

     xor2 xor2_p7(a[7], b[7], p[7]); //7th bit
     and2 and2_g7(a[7], b[7], g[7]);
     and_or2 and_or_cg1_7(p[7], g[6], g[7], p[7], p[6], cg1[7],
cp1[7]);

     and_or2 and_or_cg2_2(cp1[2], cg1[1], cg1[2], cp1[2], cp1[1],
cg22, cp22);
     and_or2 and_or_cg2_3(cp1[3], cg1[1], cg1[3], cp1[3], cp1[1],
cg23, cp23);

     and_or2 and_or_cg2_7(cp1[7], cg1[5], cg1[7], cp1[7], cp1[5],
cg27, cp27);
     and_or2 and_or_cg3_4(cp1[4], cg23, cg1[4], cp1[4], cp23,
cg34, cp34);
     and_or2 and_or_cg3_5(cp1[5], cg23, cg1[5], cp1[5], cp23,
cg35, cp35);

     and_or2 and_or_cg3_6(cp1[6], cg35, cg1[6], cp1[6], cp35,
cg36, cp36);
     and_or2 and_or_cg3_7(cp27,   cg23, cg27,   cp27,   cp23,
cg37, cp37);

     and_or2 and_or_cg2_11(cp1[11], cg1[9], cg1[11], cp1[11],
cp1[9], cg211, cp211);
     and_or2 and_or_cg2_15(cp1[15], cg1[11], cg1[15], cp1[15],
cp1[11], cg215, cp215);

     and_or2 and_or_cg3_15(cp215, cg211, cg215, cp215, cp211,
cg315, cp315);

     and_or2 and_or_cg4_11(cp211, cg37, cg211, cp211, cp37, cg411,
cp411);
     and_or2 and_or_cg4_15(cp315, cg37, cg315, cp315, cp37, cg415,
cp415);

     and_or2 and_or_cg5_9(cp1[9], cg37, cg1[9], cp1[9], cp37 ,
cg59, cp59);
     and_or2 and_or_cg5_13(cp1[13], cg411, cg1[13], cp1[13],
cp411, cg513, cp513);
```

```
      assign s[0] = p[0];
      xor2 xor2_s1(p[1], cg1[0], s[1]);
      xor2 xor2_s2(p[2], cg1[1], s[2]);
      xor2 xor2_s3(p[3], cg22, s[3]);
      xor2 xor2_s4(p[4], cg23, s[4]);
      xor2 xor2_s5(p[5], cg34, s[5]);
      xor2 xor2_s6(p[6], cg35, s[6]);
      xor2 xor2_s7(p[7], cg36, s[7]);
      assign s[8] =  cg37;

      assign sum = {s[8:0]};

endmodule

-----------------------------------------------------------------
--
```

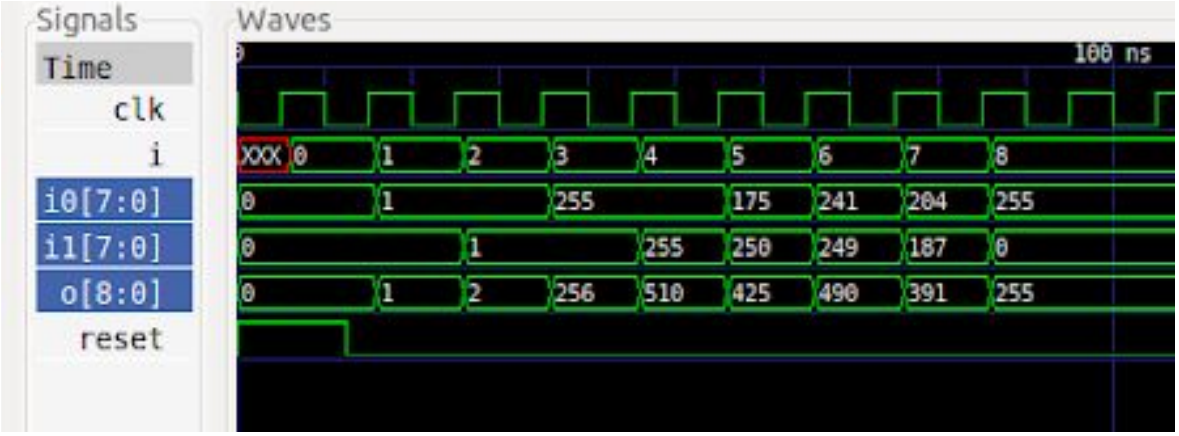
// in testBench.v

```
`timescale 1 ns / 100 ps
`define TESTVECS 8

module tb;
  reg clk, reset;
  reg [7:0] i0, i1;
  wire [8:0] o; wire cout;
  reg [31:0] test_vecs [0:(`TESTVECS-1)];
  integer i;
  initial begin $dumpfile("tb_alu.vcd"); $dumpvars(0,tb); end
  initial begin reset = 1'b1; #12.5 reset = 1'b0; end
  initial clk = 1'b0; always #5 clk =~ clk;
  initial begin
     //test_vecs[0][33:32] = 2'b00; test_vecs[0][31:8] =
16'h00;test_vecs[0][15:0] = 16'h00;

    test_vecs[0][15:8] = 16'b01;  test_vecs[0][7:0] = 16'b00;
    test_vecs[1][15:8] = 16'b01;  test_vecs[1][7:0] = 16'b01;
    test_vecs[2][15:8] = 16'hff;  test_vecs[2][7:0] = 16'b01;
    test_vecs[3][15:8] = 16'hff;  test_vecs[3][7:0] = 16'hff;
    test_vecs[4][15:8] = 16'haf;  test_vecs[4][7:0] = 16'hfa;
    test_vecs[5][15:8] = 16'hf1;  test_vecs[5][7:0] = 16'hf9;
    test_vecs[6][15:8] = 16'hcc;  test_vecs[6][7:0] = 16'hbb;
    test_vecs[7][15:8] = 16'hff;  test_vecs[7][7:0] = 16'h00;

  end
  initial {i0, i1} = 0;
  prefixAdder PA_0 (i0, i1, o);
  initial begin
    #6 for(i=0;i<`TESTVECS;i=i+1)
      begin #10 {i0, i1}=test_vecs[i]; end
    #100 $finish;
  end
endmodule
-----------------------------------------------------------------
--
```

//in lib.v

```
module invert(input wire i, output wire o1);
     assign o1 = !i;
endmodule

module and2(input wire i0, i1, output wire o2);
     assign o2 = i0 & i1;
endmodule

module and3(input wire i0, i1, i2, output wire o6);
     assign o6 = i0 & i1 & i2;
endmodule

module or2(input wire i0, i1, output wire o3);
     assign o3 = i0 | i1;
endmodule

module xor2_v2(input wire i0, i1, output wire o3);
     assign o3 = ( i0 & (!i1) ) | ( i1 & (!i0) );
endmodule

module xor2(input wire i0, i1, output wire o4);
     assign o4 = i0 ^ i1;
endmodule

module xor3(input wire i0, i1, i2, output wire o7);
     assign o7 = i0 ^ i1 ^ i2;
endmodule

module xor3E(input wire i0, i1, i2, output wire o7);
     assign o7 = ( !i0 & !i1 & i2 ) | ( !i0 & i1 & !i2 ) | ( i0 &
!i1 & !i2 );// | ( i0 & i1 & i2 );
endmodule

module or3(input wire i0, i1, i2, output wire o4);
     assign o4 = i0 | i1 | i2;
endmodule

module nand2(input wire i0, i1, output wire o5);
     wire t;
     and2 and2_0 (i0, i1, t);
     invert invert_0 (t, o5);
endmodule
```

Output:


Signals
Waves
Time
clk
i
i0[7:0]
i1[7:0]
o[8:0]
reset
100 ns
XXX 0 1 2 3 4 5 6 7 8
0 1 255 175 241 204 255
0 1 255 250 249 187 0
0 1 2 256 510 425 490 391 255