LABORATORY # 4
L A B M A N U A L

Datapath Components – Adders
(EDA Playground)

Objectives

1. Design of adders, synthesis and implementation;
2. Design of special purpose registers

Equipment

- PC or compatible

Software

- EDA Playground

Parts

- N/A

Background - Adders Design

In this FPGA application development assignment, we will implement a calculator that does just one thing – adds two 4-bit numbers.

4-bit lookahead adder

An N-bit adder adds two N-bit numbers plus a carry-in bit, resulting in an N-bit sum and a carry-out bit. A block diagram of a 4-bit adder appears in **Figure L6-1**.

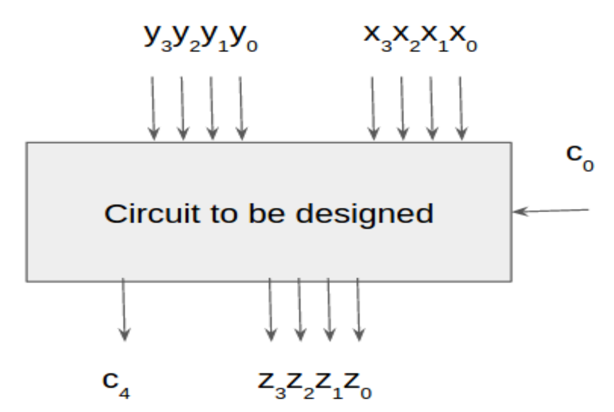

Figure L6-1. Block Diagram of 4-bit adder

Although we could design a 4-bit adder's circuit using the combinational logic design process, the resulting circuit would be rather large. Let's assume that we are adding two n-bit numbers $x_{n-1} .. x_1 \; x_0$ and $y_{n-1} .. y_1 y_0$. The result is $z_{n-1} .. z_1 z_0$. From the class notes, recall that in a ripple carry adder we used full adders to add x_i, y_i and c_i and get as result z_i and c_{i+1}. The equations for these quantities are as follows

$c_{i+1} = (x_i \,\&\, y_i) \;|\; (x_i \,\&\, c_i) \;|\; (yi \,\&\, ci)$

$z_i = (x_i \wedge y_i \wedge c_i)$

With simple boolean algebra, we can rewrite c_{i+1} as

$c_{i+1} = (x_i \,\&\, y_i) \;|\; c_i(x_i \,|\, y_i) ;$

Remember from lecture it takes N full-adder delays for the carry to propagate through the carry-ripple adder. To avoid this, we can use a different design approach which targets speed.

In this lab you will be designing a carry look-ahead adder. Here the carry bits (c_n .. c_2 c_1) are pre-calculated using a separate module and fed into each full-adder. The full-adder in turn just calculates the result bits (z_{n-1} .. z_1 z_0)

Note, from the previous equation c_{i+1} can be rewritten as $c_{i+1} = g_i + p_i c_i$ where $g_i = (x_i \,\&\, y_i)$ and $p_i = x_i \mid y_i$. As result c_1 and c_2 can be written as

$$c_1 = g_0 + p_0 c_0$$

$$c_2 = g_1 + p_1 c_1 = g_1 + p_1(g_0 + p_0 c_0) = g_1 + p_1 g_0 + p_1 p_0 c_0$$

Likewise, derive the equations for c3 and c4. You will be needing these later.

At this point we have equations to compute all c_i. Now, to compute z_i we can use the equation $z_i = (x_i \wedge y_i \wedge c_i)$ where the c_i's are as above. Now connect four full-adders to create a 4-bit adder, as shown in **Figure L6-2**. The figure does not show all the connections of the inputs and outputs to the full-adders, but you should be able to determine those connections easily.

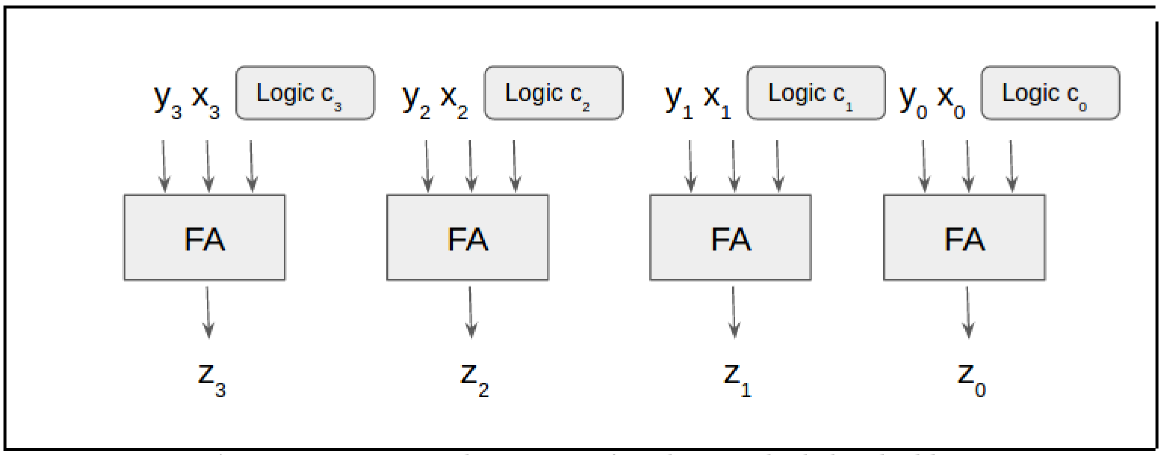

Figure L6-2. General structure of a 4-bit carrylookahead adder

Simulate the system and observe the outputs. Try adding some small numbers, like 0000+0000+0 (which should result in 0 0000) and 0001+0001+0 (which should equal 0 0010). Also try adding some larger numbers, like 1111+1111+0 (which should equal 1 1110), and 1111+1111+1 (which should equal 1 1111).

Question: Is it possible for two 4-bit numbers and a carry-in to result in a number too big to represent using 4 sum bits and a carry-out bit?

Note that the above adder assumes the inputs are unsigned numbers (i.e., the inputs are *not* in two's complement form).

Specification

Create and test a Full adder using structural verilog. In order to simplify your design we suggest creating four components. One component to implement the logic of a full adder (code given) and one for an N-bit register (code given) . Another component to implement the logic of the carry unit (part of the code given). Finally, create a 4-bit carry look-ahead structure module that uses the components already created (set this as your top level module if you are creating separate Verilog files for each module). The following are the interfaces of the modules we suggest.

```
module falogic(
 output r,  // We label our output as r instead of z
 input x,
 input y,
 input cin
   );
wire t1;
xor cx1 ( t1, x,y  );
xor cx2 ( r, t1, cin  );

endmodule

module register_logic(
  input clk,
  input enable ,
  input [4:0] Data ,
  output reg [4:0]  Q ) ;

// on real FPGA board which has clk signal, we use the following always statement:
// always @(posedge clk )
// begin
//   if ( enable) begin
//   Q = Data;
// end
// end
// endmodule

// for simulation, we force the statement to execute without clk signal:
always @(*) begin
   if ( enable) begin
       Q = Data;
```

```
   end
end

endmodule
```

```
module carrylogic(
output [3:0] cout ,
input cin,
input [3:0] x,
input [3:0] y
  );

// Computing all gx

wire g0, g1, g2, g3 ;

assign g0 = x[0] & y[0] ;
assign g1 = x[1] & y[1] ;
assign g2 =// Your code ;
assign g3 = //Your code ;

// Computing all px
wire p0, p1, p2, p3 ;

assign p0 = x[0] + y[0] ;
assign p1 = x[1] + y[1] ;
assign p2 = //Your code ;
assign p3 = //Your code ;

// Computing all carries

assign cout[0] = g0 | ( p0 & cin) ;
assign cout[1] = g1 | ( p1 & ( g0 | (p0 & cin) )  ) ;
assign cout[2] = // Your code ;
assign cout[3] =  //Your code ;

endmodule
```

```
module carrylookahead_st(
 input clk ,
 input cin,
 input [3:0] x,
 input [3:0] y,
 output cout,
 output [3:0] r
  );
```

```
wire [3:0] c;
wire [3:0] ir1 ;
wire [4:0] ir2 ;

// Compute Carries
carrylogic cx1 ( c, cin, x, y ) ;

// Compute R
falogic cx6 ( ir1[0], x[0], y[0], cin   ) ;
// Your code (3 more full adders)

// Register
register_logic cx10 ( clk, 1'b1, {c[3],ir1}, ir2 )  ;

// Results
assign r = ir2[3:0] ;
assign cout = ir2[4] ;

endmodule
```

If required, please rename cx1 to avoid confusion with falogic cx1 as they are different functions.

The testbench code is given below.

```
`timescale 1ns / 1ps

////////////////////////////////////////////////////////////////////////////
/////
////////////////////////////////////////////////////////////////////////////
/////

module carrylookahead_tb;
```

```
        // Inputs
        reg cin;
        reg [3:0] x;
        reg [3:0] y;
        reg clk;
        // Outputs
        wire cout;
        wire [3:0] r;

        reg [3:0] rx;

        integer index ;
        //always @(*) // no sensitivity list, so it always executes
        //      begin
        //              clk = 1; #100; clk = 0; #100; // 10ns period
        //      end
        // Instantiate the Unit Under Test (UUT)
        carrylookahead_st uut (
                .clk(clk),
                .cin(cin),
                .x(x),
                .y(y),
                .cout(cout),
                .r(r)
        );
```

For the CLK to function as expected,
generate clock signal with a 10ns period.

```
        initial begin
      $dumpfile("dump.vcd"); $dumpvars;
                // Initialize Inputs
                cin = 'd0;
                y = 'd0;
                // r = x + 0 ;  cout = 0;
          $display("TC11 ");
          for (index=0; index < 15; index = index + 1) begin
                        x = index ;
                        #100;
                        if ( r != x ) $display  ("Result is wrong");
                        if ( cout != 1'b0 ) $display  ("Result is wrong -
Carryout ");
                end

           // r = x + 1 ;
```

```
                cin = 1'b1;
                y = 4'b0;
     $display("TC12 ");

          for (index=0; index < 15; index = index + 1) begin
                                x = index ;
                                #100;
                                if ( r != (x + 'd1) ) $display  ("Result is
wrong %b %b" , r, (x+1) );
                                if ( cout != 1'b0 ) $display  ("Result is wrong -
Carryout ");
                end

                // r = x + y + 1 ;
     cin = 1'b1;
     $display("TC13 ");
          for (index=0; index < 8; index = index + 1) begin
                                x = index ;
                                y = index ;
                                #100;
                                if ( r != (x + y +1 ) ) $display  ("Result is
wrong %b %b" , r, (x+y) );
                                if ( cout != 1'b0 ) $display  ("Result is wrong -
Carryout ");

                end

                // r = x + y + 1 ;
                cin = 1'b1;
     $display("TC14 ");
          for (index=8; index < 16; index = index + 1) begin
                                x = index ;
                                y = index ;
                                rx = x + y + cin ;
                                #100;
                                if ( r != rx ) $display  ("Result is wrong %b %b" , r,
rx );

                                if ( cout != 1'b1 ) $display  ("Result is wrong -
Carryout ");

                end
```

Here, if the rx does not show the accurate values,
Adding a small delay after x,y,cin would help
for the rx computation to complete before assigning.

```
        end

endmodule
```

Demonstration

Demonstrate that the application performs according to specs.

Presentation and Report

Must be presented according to the general EE/CS120A lab guidelines.

Prelab

1. Review Chapter 4 Lecture (particularly the section on Adders);
2. Try to answer all the questions, prepare logic truth tables, do all necessary computations