LABORATORY # 5
LAB MANUAL

Datapath Components - Adders

Objectives

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

Equipment

- PC or compatible
- Digilent's Basys Spartan-3E FPGA Evaluation Board

Software

- Xilinx ISE Design Software Suite
- ModelSim XE III modeling software
- Digilent's Adept ExPort Software

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**.

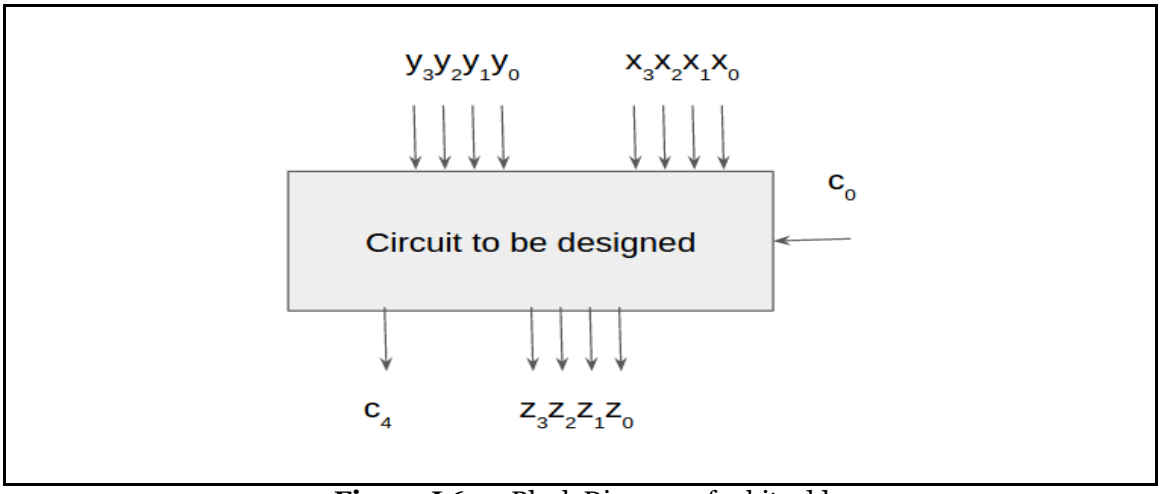

Figure L6-1. Block Diagram of 4-bit adder

Although we could design a 4-bit adder's circuit using the combinational logic design process, the resulting circuit would be rather large. Instead, we can use a different design approach which target speed. Let's assume that we are adding two n-bits numbers $x_{n-1} .. x_1 \; x_0$ and $y_{n-1} .. y_1 \; y_0$. The result is $z_{n-1} .. z_1 \; z_0$. For 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

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

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

Now that c_{i+1} can be written also as

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

Moreover 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

$c_1 = g_0 + p_0 c_0$

$c_2 = g_1 + p_1c_1 = g_1 + p_1(g_0 + p_0c_0) = g_1 + p_1g_0 + p_1p_0c_0$

Notice that when calculating c_2 we don't need c_1. As this point we have equations to compute all c_i. 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.

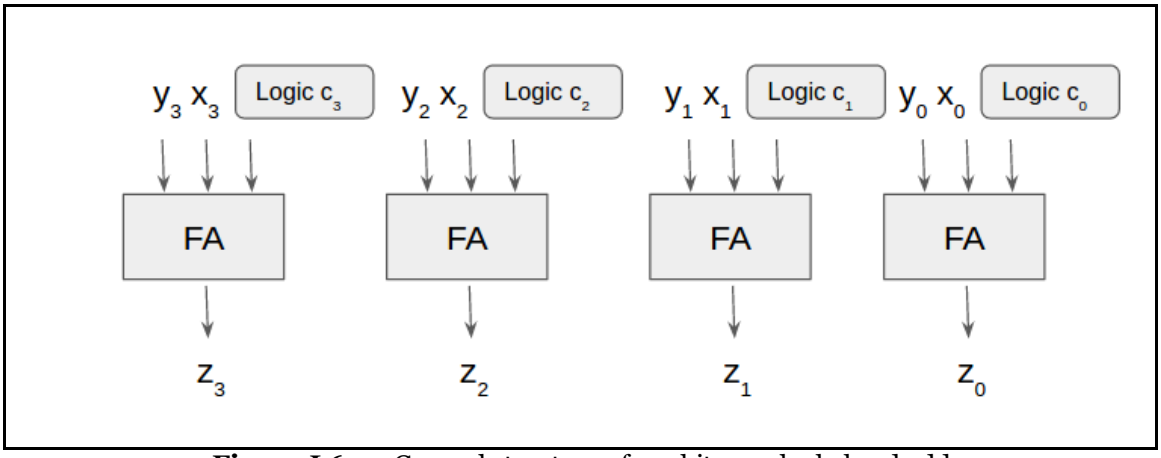

Figure L6-2. General structure of a 4-bit carrylookahead adder

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). 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

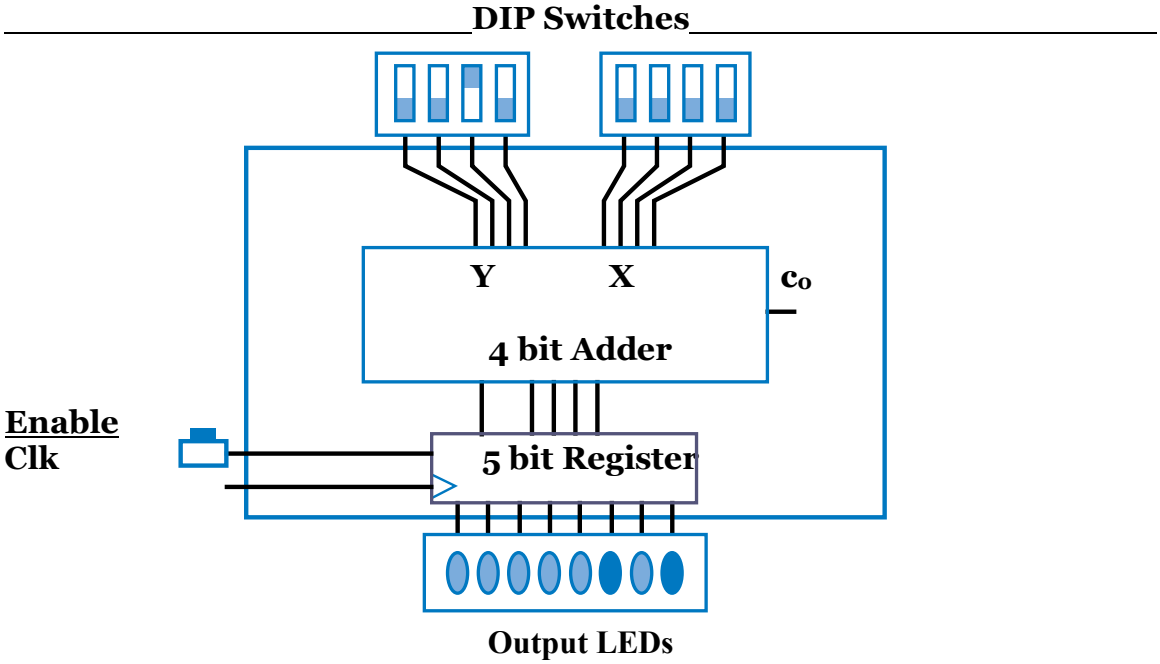

Figure L6-3. Adder Structure and Basys Board implementation hint

Implement a 4-bit adder system on the Basys development board. Use two 4-position DIP switches for the two 4-bit inputs, a single switch for the carry-in input, and five LEDs for the give outputs. Then try different combinations of the inputs and observe the outputs. Use one of the buttons to load the results to the LEDs. See **Figure L6-3** for details.

Part A

Create and test the Full Adder as a separate schematic, and implement in the top module as a schematic symbol. The 5-bit register with load[1] should also be realized in a separate schematic and brought in as a symbol.

Part B

Create and test a Full adder using structural verilog. In order to simplify your design we suggest to create four components. One component to implement the

[1] Load functions to update the register. Until load is HIGH, the registry will not update its value.

logic of a full adder (code given) and 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 carrylookahead module that uses the components already created. The following are the interfaces of the modules we suggest.

```
module falogic(
 output r,
 input x,
 input y,
 input cin
   );

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 ) ;

  always @(posedge clk )
  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 = x[2] & y[2] ;
assign g3 = x[3] & y[3] ;

// 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 enable ,
  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
```

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

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

endmodule
```

Implementation Utilities and Additional Hints

The following CLK configuration must be used in the constraints file:

```
//Inputs

NET "clk" LOC = "P54" ;
NET "enable" LOC = "XX" ;

// Xs and Ys

NET "x[0]" LOC = "P38" ;
NET "x[1]" LOC = "P36" ;
NET "x[2]" LOC = "P29" ;
NET "x[3]" LOC = "P24" ;

NET "y[0]" LOC = "P18" ;
NET "y[1]" LOC = "P12" ;
NET "y[2]" LOC = "P10" ;
NET "y[3]" LOC = "P6" ;

// Outputs

NET "r[0]" LOC = "P15" ;
NET "r[1]" LOC = "P14" ;
NET "r[2]" LOC = "P8" ;
NET "r[3]" LOC = "P7" ;
NET "cout" LOC = "P5" ;
```

Demonstration

Demostrate that the application performs according to specs.

Procedures

1. Xilinx ISE Design and Synthesis environment;
2. Creation of Configuration files;
3. Usage of Adept ExPort download software;

Presentation and Report

Must be presented according to the general EE120A lab guidelines posted in iLearn.

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