Bruno E. Gracia Villalobos

EE 4513

Assignment # 2

4-bit and 8-bit Carry Lookahead Adder Design

September 10, 2019

Introduction

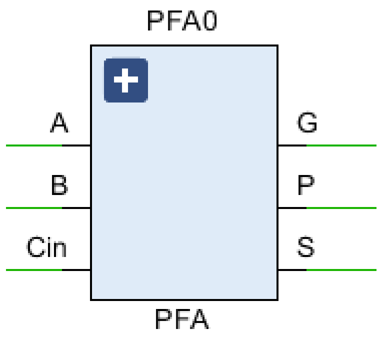
The design incorporates an N-bit Carry-Lookahead-Adder (CLA) as the top module to incorporate both 4-bit and 8-bit adders. The operands are N-bit augend and addends, as well as a 1-bit carry in. The amount of adder bits is defined at compilation and passed with *parameter* data type to the submodules of the instantiated top module. Not only does this approach increase the modularization of the design—for use in higher-order adders such as 16 or 32 bits—the N-bit approach also allows the engineer to have greater control over the efficacy of the design.

The design is created at a *structural* level of abstraction with a *bottom-up* approach to have control over the number of gates synthesized. A Partial-Full-Adder (PFA) and Carry Generator are developed as submodules using *assign* statements to specify the logic. A Carry-Lookahead-Logic (CLL) module is developed as a wrapper to instantiate N-bit carry generators. Most importantly, the CLA top module instantiates N-bit PFA's and the CLL to output the desired sum and carry.

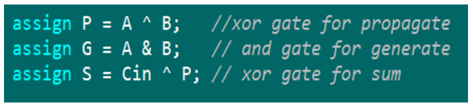
The design has been simulated, synthesized, and verified at *RTL* and is the subject of the subsequent sections.

Procedure

Partial-Full-Adder (PFA)



First, the PFA is designed. The main purpose of the PFA is to output the *propagate* and *generate* bits which dictate the module's behavior as a tunnel for the *previous* bit's carry and as a creator of the *next* bit's carry (*previous* and *next* terms are indicated as 1 bit lower in significance and 1 bit higher in significance respectively). In other words, the propagate and generate bits serve as a communication avenue between other PFA's.



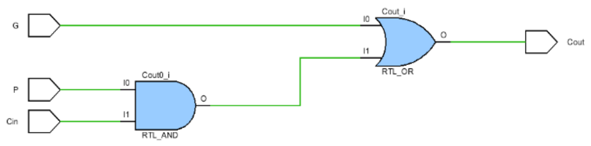
```
assign P = A ^ B;   //xor gate for propagate
assign G = A & B;   // and gate for generate
assign S = Cin ^ P; // xor gate for sum
```

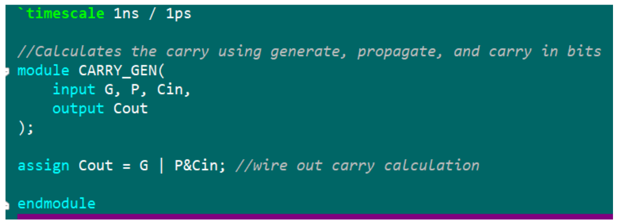
The module takes in 3-bits: A and B operands as well as a carry-in bit. These are channeled through the gates to create the resultant *propagate* (P), *generate* (G), and sum (S) bits. Figure 1 displays the PFA port-list through the RTL schematic for better visualization.

To generate the logic for the outputs, *assign* statements are used. Figure 2 displays the Verilog code definition for the output wires. Simple reduction operators point the synthesizer to the gates needed.

Carry-Generator

The Carry Generator is designed after the PFA. As its name suggests, this module takes in the *propagate*, *generate*, and preliminary carry-in (*Cin*) bits to calculate the carry-out bit. For an N-bit CLA, there will be N carry generators. It is important to mention the 0th carry generator takes in the preliminary carry-in from the outside world, and the *next* carry-generators take in the *previous* carry-out as carry-ins. Figure 3 showcases the 2 level AND-OR network used to calculate the carry-out bit for a single Carry Generator.





```
`timescale 1ns / 1ps

//Calculates the carry using generate, propagate, and carry in bits
module CARRY_GEN(
    input G, P, Cin,
    output Cout
);

assign Cout = G | P&Cin; //wire out carry calculation

endmodule
```

The Verilog code for the Carry Generator is a module that assigns the Carry out bit using a reduction OR operator. Figure 4 is the realization of the Carry Generator in Verilog. Given its simplicity, the showcase of simulation will be skipped in the “Observation and Results” section below.

Carry Lookahead Logic (CLL)

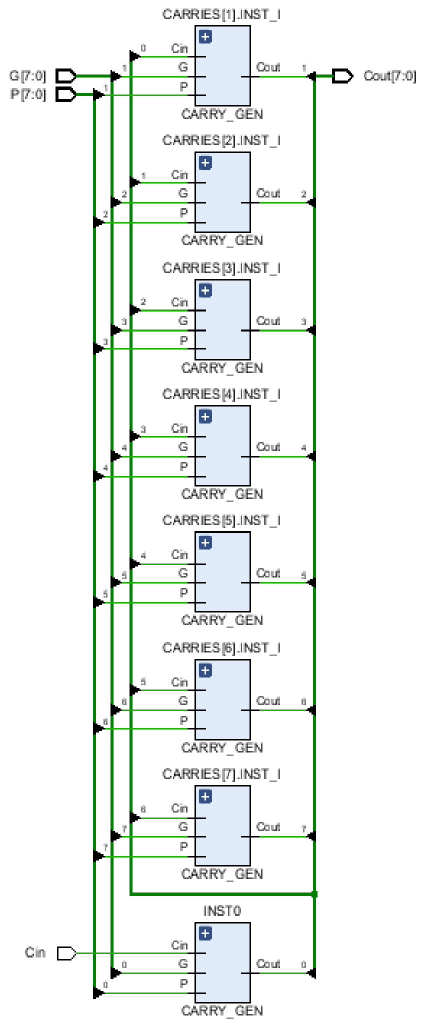
Now that the Carry Generator is developed, the CLL module is next. The CLL module is a wrapper for N carry generators and is the module that will give rise to the CLA's name: carry bits are created in parallel to reduce propagation delays of the carry bits between single bit adders. For this reason, the CLL module takes in the *propagate* and *generate* bits from the N PFA's and a single *initial carry-in bit*. The latter is passed on to the LSB PFA and the former to the rest of the PFA's. Figure 5 displays the CLL RTL schematic for the 8-bit configuration; the 4-bit schematic is omitted due to the logical deduction of 4 Carry Generators instead.

To implement the N carry generator logic, a *parameter* called *bits* specifies the number of modules to instantiate. Within the CLL module, the LSB Carry generator is instantiated as is, whilst the rest are instantiated with a *generate* block for modularity. Figure 6 displays the *generate* block for the CLL module. A significant observation is in the *named association* used to instantiate the CARRY_GEN (Carry Generator) module: port *Cin* takes in the previous Carry Generator's *Cout* and can be thought of as an internal "ripple" carry to the CLL block. However, the "ripple" happens in parallel because all the Carry Generators calculate the output concurrently.

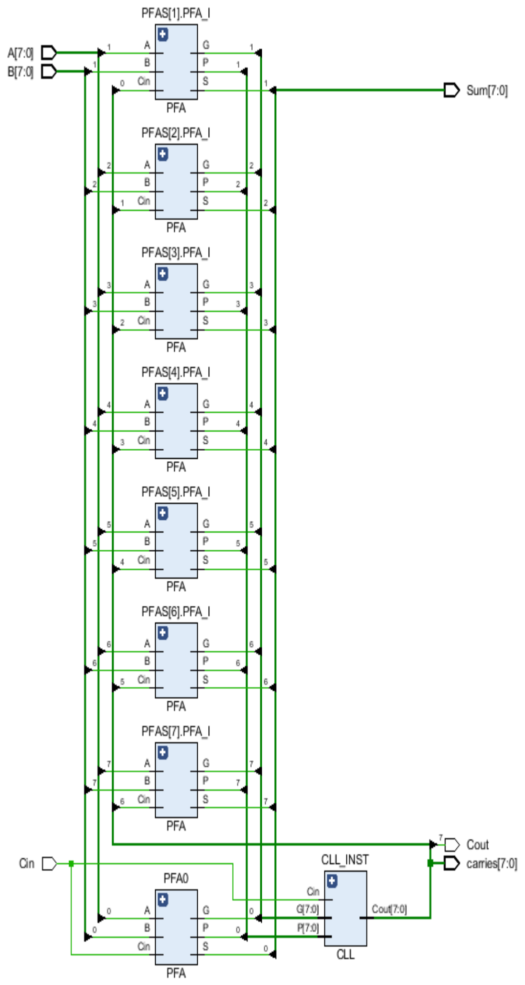
```
genvar i;
generate //Create N bits of carry generators
    for(i=1; i < bits; i=i+1) begin : CARRIES
        CARRY_GEN INST_I(
        .G(G[i]),
        .P(P[i]),
        .Cin(Cout[i-1]),

        .Cout(Cout[i])
        );

    end
```

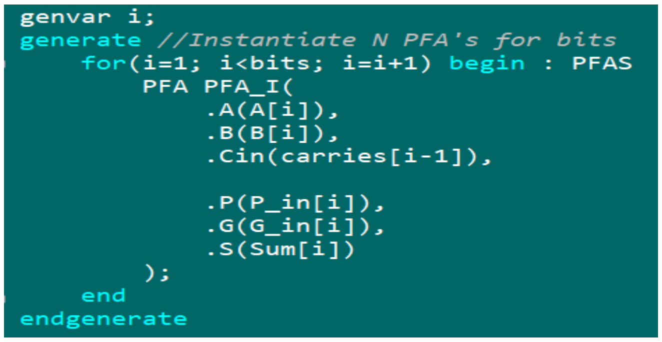


Carry Lookahead Adder (CLA)



Now with the two building blocks completed—PFA and CLL—the CLA is ready to be developed. Essentially, the CLA is a wrapper of PFA and CLL instantiations, of which their number of instantiations depends on the specified width by the *parameter* “bits” (default is 8). The CLA takes in an augend and an addend of N bits, and a 1-bit initial *Cin*. The output is an N bit sum and a 1-bit *Cout*.

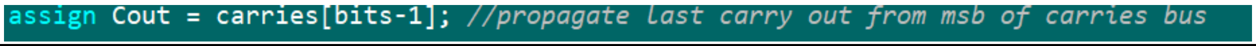
To instantiate PFA’s selectively based on the number of bits specified by the *parameter*, a *generate* block is used. The logic of the named port .Cin in the PFA instantiation requires the carry-out from the respective index as output from the CLL. Given the LSB of the adder takes in the initial *Cin*, as well as the loop logic to use the previous index value for .Cin, the loop starts at 1 and iterates up to the N bits to prevent erroneous indexing of the *carries* wire bus with -1.



```
genvar i;
generate //Instantiate N PFA's for bits
    for(i=1; i<bits; i=i+1) begin : PFAS
        PFA PFA_I(
            .A(A[i]),
            .B(B[i]),
            .Cin(carries[i-1]),

            .P(P_in[i]),
            .G(G_in[i]),
            .S(Sum[i])
        );
    end
endgenerate
```

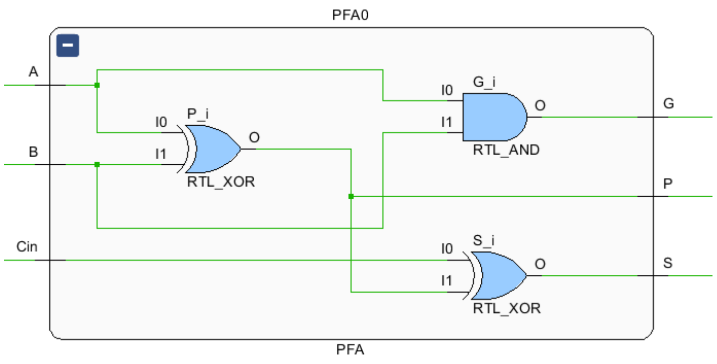
Lastly, the CLL is instantiated to provide for the carry-outs needed to output the sum. These are provided in an N bit bus (parallel) to each of the PFA’s. The collective output of all PFA’s form an N bit sum bus as the output of the CLA module. The carry out bit of the entire sum is *assigned* from the MSB of the CLL *carries* bus as shown below.



```
assign Cout = carries[bits-1]; //propagate last carry out from msb of carries bus
```

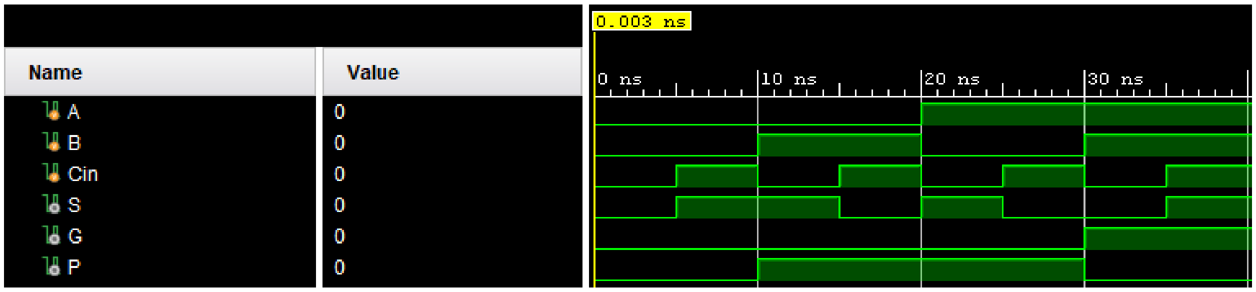
Observation and Results

Partial Full Adder (PFA)



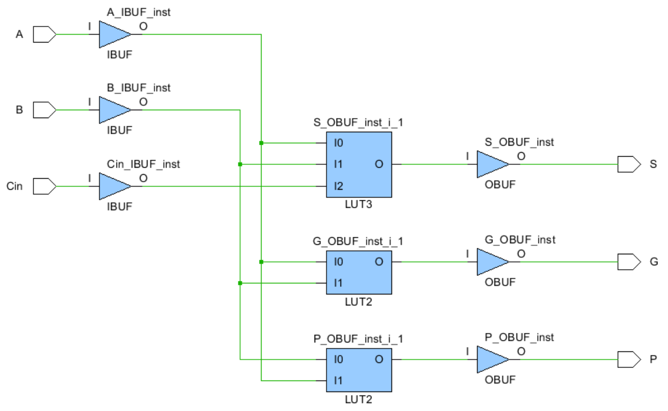
After developing the Vivado code for the PFA, the Elaborated Design function of the Vivado workflow toolbar is used to verify the intended logic at RTL. Although not as significant as synthesis, this schematic helps the engineer to quickly elaborate code without emphasis on realizing the design in hardware such as an FPGA. Figure 3 displays the schematic generated by Vivado's interpretation of the PFA code.

The next step is to simulate the design for desired operation, and Figure 4 showcases a full truth-table test of the module. Given three input bits are sent to the PFA, there are 2^3 = 8 possible combinations of outputs; these are all verified using Vivado's behavioral simulation tool.



Name	Value
A	0
B	0
Cin	0
S	0
G	0
P	0

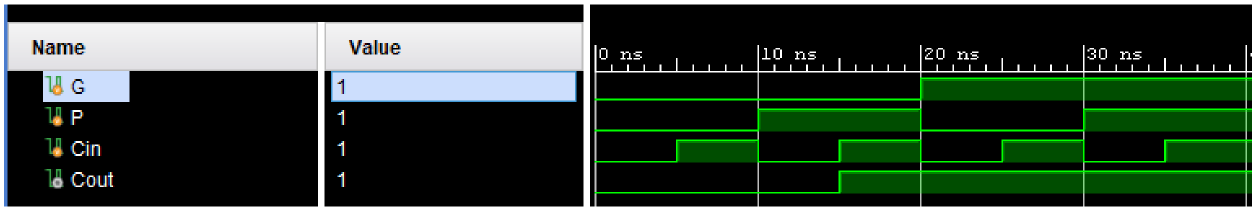
Once the simulation verified the PFA, the last step consists of synthesizing the design. As the key step of the design process for this course, synthesis serves as a measure for the realizability of the Verilog code developed by the engineer as a blueprint to the final hardware product. The *Xilinx Synthesis Technology* (XST) synthesis tool from Vivado generates this result and is displayed as a schematic in Figure 5.



As expected, the XST uses LUT's to specify the desired output of the PFA. For both the G and P bits, a LUT2 is used to specify the 2^2 = 4 possible combinations of outputs available. However, since the S bit relies on a level 2 circuit (output of level 1 gate feeds in to level 2 gate), a LUT3 is chosen by the XST to specify the output's 2^3 = 8 possible combinations.

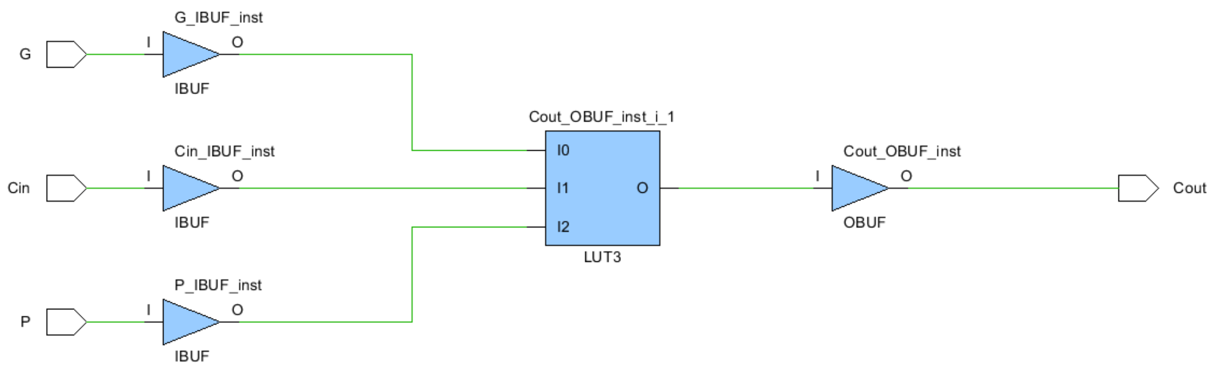
Carry Generator

The simulation of the Carry Generator consists of testing the UUT with 8 different combinations due to its 3 inputs. All of these are displayed in the Figure below.



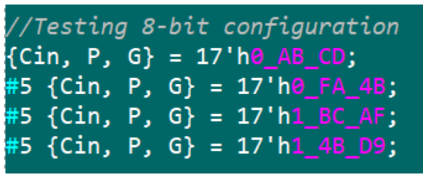
Name	Value
G	1
P	1
Cin	1
Cout	1

As expected, the XST generates a LUT3 for this design. The synthesized schematic is shown below.

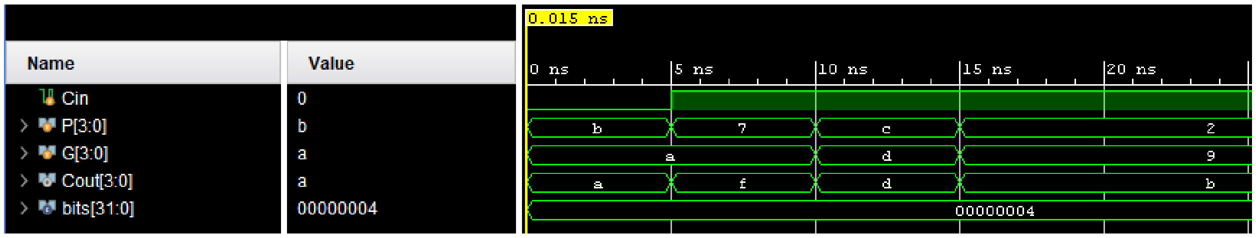


Carry Lookahead Logic (CLL)

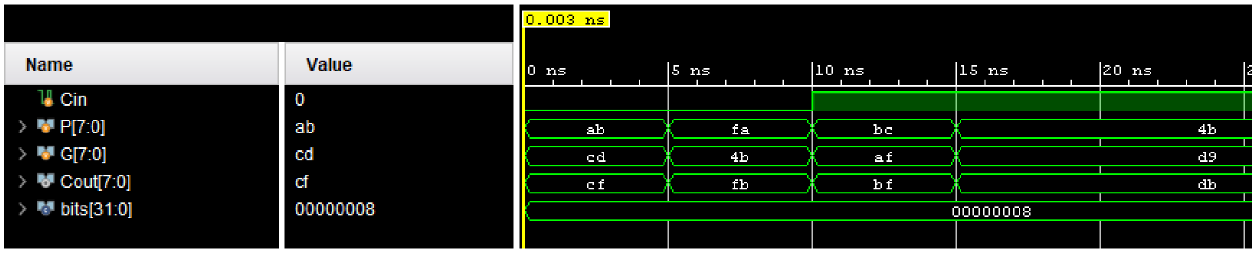
The CLL block is simulated with both 4 and 8-bit configurations as specified by the *parameter* bits. First, the 4-bit simulation is presented in the Figure below. It is difficult to assess the internal carries between Carry Generators, but the output is confirmed correct. Likewise, the 8-bit also cannot show the internal carries between Carry Generators, but the output is presented given A, B, and Cin inputs. For both simulations, each of the test inputs are separated by a 5ns delay. To facilitate the elaboration of the testbench, the concatenation operator in Verilog is used to specify a string of hex digits assigned to all three quantities in one line as shown in the Figure below.



```
//Testing 8-bit configuration
{Cin, P, G} = 17'h0_AB_CD;
#5 {Cin, P, G} = 17'h0_FA_4B;
#5 {Cin, P, G} = 17'h1_BC_AF;
#5 {Cin, P, G} = 17'h1_4B_D9;
```

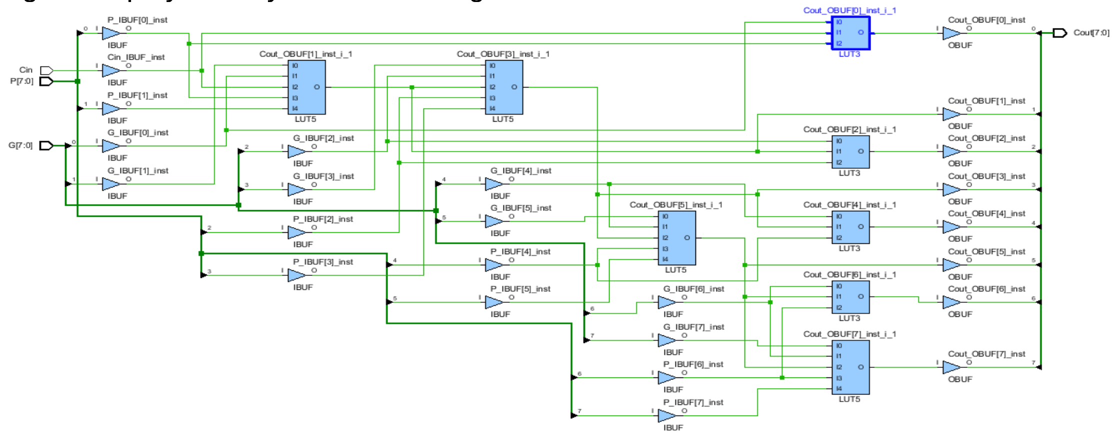


```
Name         Value       0.015 ns
                         0 ns      5 ns      10 ns     15 ns     20 ns
Cin          0
P[3:0]       b           b         7         c         2
G[3:0]       a           a                   d         9
Cout[3:0]    a           a         f         d         b
bits[31:0]   00000004    00000004
```



```
Name         Value       0.003 ns
                         0 ns      5 ns      10 ns     15 ns     20 ns
Cin          0
P[7:0]       ab          ab        fa        bc        4b
G[7:0]       cd          cd        4b        af        d9
Cout[7:0]    cf          cf        fb        bf        db
bits[31:0]   00000008    00000008
```

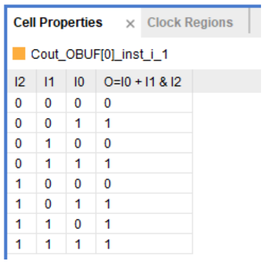
Next, the synthesis of the CLL is elaborated to analyze the realization by the XST. The 8-bit CLL is presented first. As expected, the XST chooses LUT's to realize the design given the limitations of the CLB block in the specified Artix-7 FPGA for the Vivado project. The following Figure displays the synthesized design.



For the author's interest, the truth table of the Cout_OBUF[0]_inst_i_1 cell is examined (highlighted in the schematic), of which corresponds to the LSB bit instantiation of the Carry Generator module. First, the internal pins of the LUT are translated to the external nets. The Figure below shows the mapping. As expected, the truth table equation for the cell pin "O" or output confirms the defined equation in the Carry Generator module:

assign Cout = G | P&Cin;

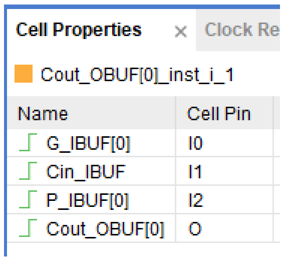
To verify the results of simulation, let's examine the LSB of the Carry out 8-bit bus. The figures below do confirm the LUT's contents. Although the XST can be certainly 100% trusted for the accuracy of the results, this observation is conducted as a learning exercise for the author.



Cell Properties × Clock Regions

🟧 Cout_OBUF[0]_inst_i_1

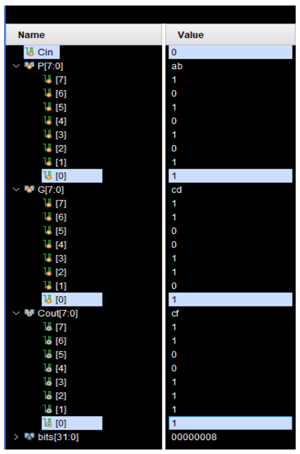
I2	I1	I0	O=I0 + I1 & I2
0	0	0	0
0	0	1	1
0	1	0	0
0	1	1	1
1	0	0	0
1	0	1	1
1	1	0	1
1	1	1	1



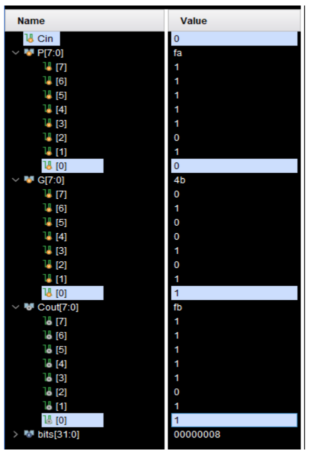
Cell Properties × Clock Re

🟧 Cout_OBUF[0]_inst_i_1

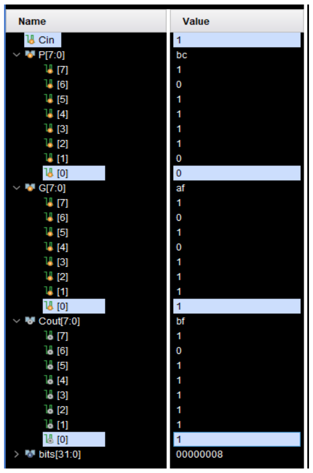
Name	Cell Pin
G_IBUF[0]	I0
Cin_IBUF	I1
P_IBUF[0]	I2
Cout_OBUF[0]	O



Name	Value
Cin	0
P[7:0]	ab
[7]	1
[6]	0
[5]	1
[4]	0
[3]	1
[2]	0
[1]	1
[0]	1
G[7:0]	cd
[7]	1
[6]	1
[5]	0
[4]	0
[3]	1
[2]	1
[1]	0
[0]	1
Cout[7:0]	cf
[7]	1
[6]	1
[5]	0
[4]	0
[3]	1
[2]	1
[1]	1
[0]	1
bits[31:0]	00000008

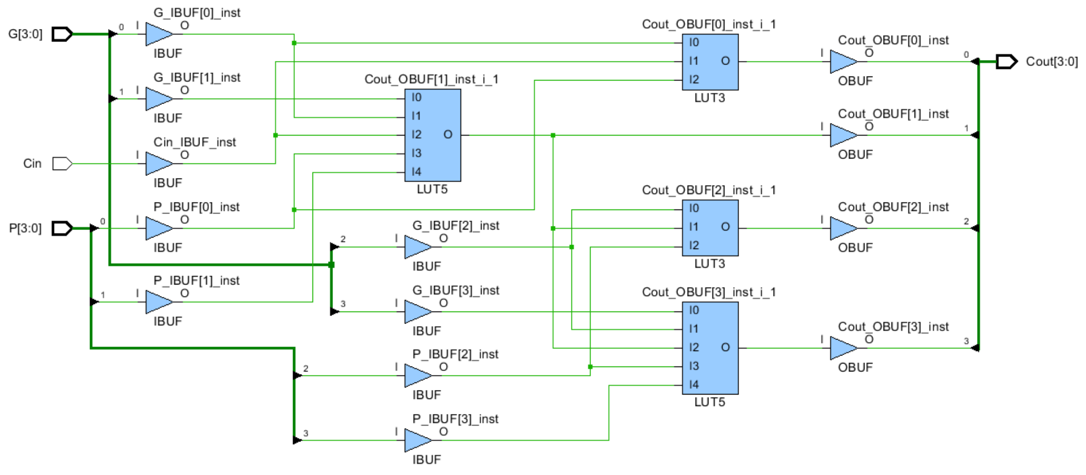


Name	Value
Cin	0
P[7:0]	fa
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	0
[1]	1
[0]	0
G[7:0]	4b
[7]	0
[6]	1
[5]	0
[4]	0
[3]	1
[2]	0
[1]	1
[0]	1
Cout[7:0]	fb
[7]	1
[6]	1
[5]	1
[4]	1
[3]	1
[2]	0
[1]	1
[0]	1
bits[31:0]	00000008



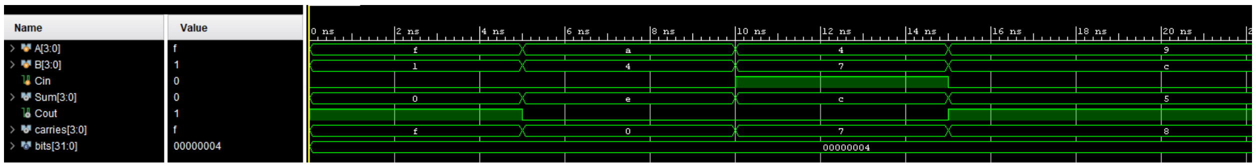
Name	Value
Cin	1
P[7:0]	bc
[7]	1
[6]	0
[5]	1
[4]	1
[3]	1
[2]	1
[1]	0
[0]	0
G[7:0]	af
[7]	1
[6]	0
[5]	1
[4]	0
[3]	1
[2]	1
[1]	1
[0]	1
Cout[7:0]	bf
[7]	1
[6]	0
[5]	1
[4]	1
[3]	1
[2]	1
[1]	1
[0]	1
bits[31:0]	00000008

For the 4-bit CLL, the following Figure represents the synthesized design. Less LUTs are used by the XST for realization because of the smaller data width.



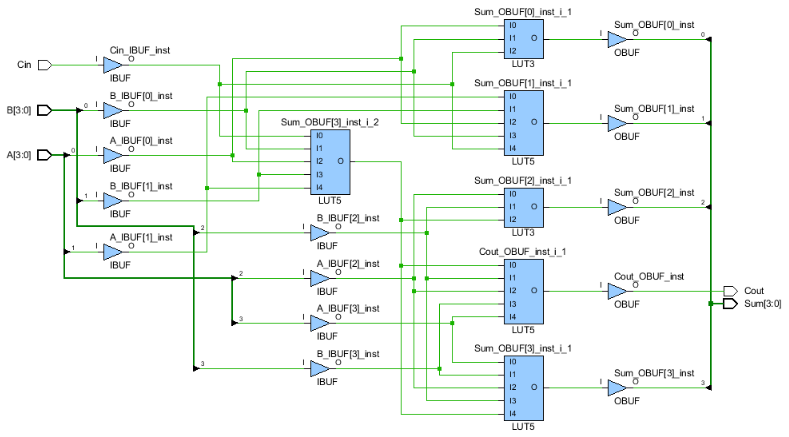
Carry Lookahead Adder (CLA)

Both 4 and 8-bit CLA's are simulated and synthesized. The 4-bit configuration is presented first. In the simulation, four different additions are performed, including the overflow case as a good benchmark shown at t=0ns to 5ns. The rest of the additions prove correct.

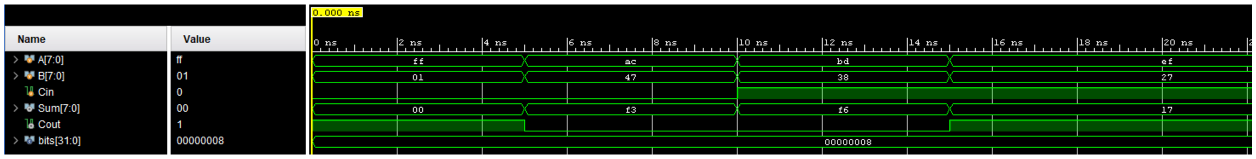


Name	Value
A[3:0]	f
B[3:0]	1
Cin	0
Sum[3:0]	0
Cout	1
carries[3:0]	f
bits[31:0]	00000004

The Figure below displays the 4-bit CLA as created by the XST. Like the rest of the modules, LUTs are used to define the combinational circuits.

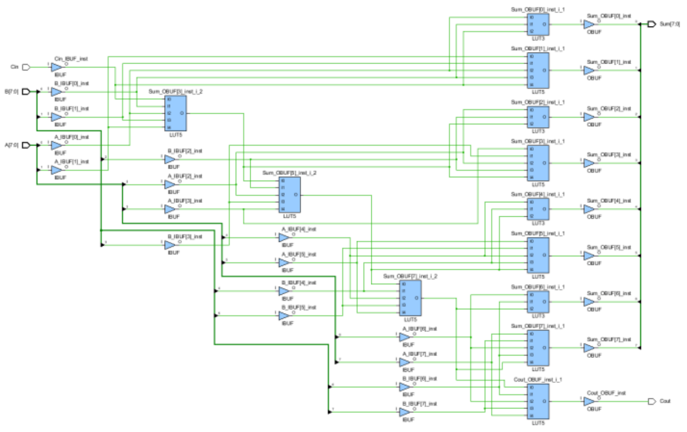


The 8-bit configuration is presented next. Again, the overflow case is used a good benchmark for verifying correct operation. The rest of the cases are also correct.



Name	Value
A[7:0]	ff
B[7:0]	01
Cin	0
Sum[7:0]	00
Cout	1
bits[31:0]	00000008

The XST output of the 8-bit CLA is shown below. The number of LUTs compares similarly to the addition of 8 PFA's and the 8-bit CLA.



Conclusion

It was nice to visualize the implementation of the LUT based synthesized designs. Surely the logic is better to visualize when in logic gate format—AND, OR, XOR, etc—but the XST outputs a schematic with the components it will use to map, place, and route in the FPGA, and this proves it difficult to understand how a combinational design with a hundred gates is implemented inside LUTs. Further research online helped to learn how to analyze cells in the synthesized schematic and this was a helpful learning exercise.

Designing both 4 and 8-bit CLAs using a bottom up approach enabled me to learn how Verilog constructs the modules one by one, from gate level up to RTL. This approach is great for an Introduction to VLSI Design class because it is important to understand digital logic at a lower level of abstraction before optimizations can be made at a higher level. Surely, the CLA could be designed using a behavioral level of abstraction, but gate level optimizations such as parallel carry are difficult to specify and can be ambiguous to the XST when synthesis comes around. Above all, it is a tradeoff between control and efficiency—gate level provides the highest level of control of a design, but behavioral saves time and can leave optimizations to the XST.

Code

PFA

```
`timescale 1ns / 1ps


//Calculates the propagate, generate, and sum bits
module PFA(
    input A, B, Cin,
    output P, G, S
);


    assign P = A ^ B;   //xor gate for propagate
    assign G = A & B;   // and gate for generate
    assign S = Cin ^ P; // xor gate for sum


endmodule
```

PFA Testbench

```
`timescale 1ns / 1ps
module PFA_SIM();
reg A, B, Cin;
wire S, G, P;



PFA UUT(
.A(A),
.B(B),
.Cin(Cin),
.S(S),
.G(G),
.P(P)

);

initial begin

{A, B, Cin} = 3'd0;
#5 {A, B, Cin} = 3'd1;
#5 {A, B, Cin} = 3'd2;
```

```
#5 {A, B, Cin} = 3'd3;

#5 {A, B, Cin} = 3'd4;

#5 {A, B, Cin} = 3'd5;

#5 {A, B, Cin} = 3'd6;

#5 {A, B, Cin} = 3'd7;


end


endmodule
```

Carry Generator

```
`timescale 1ns / 1ps

//Calculates the carry using generate, propagate, and carry in bits

module CARRY_GEN(

    input G, P, Cin,

    output Cout

);

assign Cout = G | P&Cin; //wire out carry calculation

endmodule
```

CLL

```
`timescale 1ns / 1ps


//This module handles the carry outs for each partial full adder

//Acts as a wrapper to individual carry_gen modules

module CLL #(parameter bits = 8) (

    input Cin,

    input [bits-1: 0] P, G,

    output [bits-1: 0] Cout

);


//Create carry generator block for first bit

CARRY_GEN INST0(

        .G(G[0]),

        .P(P[0]),

        .Cin(Cin),


        .Cout(Cout[0])

);
```

```
genvar i;
generate //Create N bits of carry generators
    for(i=1; i < bits; i=i+1) begin : CARRIES
        CARRY_GEN INST_I(
        .G(G[i]),
        .P(P[i]),
        .Cin(Cout[i-1]),

        .Cout(Cout[i])
        );

    end
endgenerate
endmodule
```

CLL Testbench

```
`timescale 1ns / 1ps
module CLL_SIM();
parameter bits = 8;

reg Cin;
reg [bits-1:0] P,G;
wire [bits-1:0] Cout;

CLL UUT(
.Cin(Cin),
.P(P),
.G(G),

.Cout(Cout)
);



initial begin
/*
//Testing 4-bit configuration
{Cin, P, G} = 9'h0_B_A;
```

```
# 5 {Cin, P, G} = 9'h1_7_A;

# 5 {Cin, P, G} = 9'h1_C_D;

# 5 {Cin, P, G} = 9'h1_2_9;

*/



//Testing 8-bit configuration

{Cin, P, G} = 17'h0_AB_CD;

#5 {Cin, P, G} = 17'h0_FA_4B;

#5 {Cin, P, G} = 17'h1_BC_AF;

#5 {Cin, P, G} = 17'h1_4B_D9;




end




endmodule
```

CLA

```
`timescale 1ns / 1ps


//Wrapper for Carry-Lookahead-Logic and Partial Full Adders

module CLA #(parameter bits = 8) ( //parameter to specify number of bits for adder

input [bits-1:0] A, B, //augend and addend

input Cin,  //first carry in


output [bits-1:0] Sum, //sum output

output Cout //carry out bit


//output [bits-1:0]carries //for testbench

);


wire [bits-1:0]P_in, G_in; //wire bus for propagate and generate bits

wire [bits-1:0]carries; //wire bus for carry bits


assign Cout = carries[bits-1]; //propagate last carry out from msb of carries bus
```

```
//create partial full adder instance for the first bit to incorporate Cin
PFA PFA0(
.A(A[0]),
.B(B[0]),
.Cin(Cin),

.P(P_in[0]),
.G(G_in[0]),
.S(Sum[0])

);

genvar i;
generate //Instantiate N PFA's for bits
    for(i=1; i<bits; i=i+1) begin : PFAS
        PFA PFA_I(
            .A(A[i]),
            .B(B[i]),
            .Cin(carries[i-1]),

            .P(P_in[i]),
            .G(G_in[i]),
            .S(Sum[i])
        );
    end
endgenerate

//Instantiate Carry-Lookahead-Logic block
CLL #(.bits(bits)) CLL_INST(
.Cin(Cin),
.P(P_in),
.G(G_in),

.Cout(carries)
);
```

```
Endmodule
```

CLA Testbench

```
`timescale 1ns / 1ps

module CLA_SIM();

parameter bits = 8;


reg [bits-1:0] A,B;

reg Cin;

wire [bits-1:0] Sum;

wire Cout;



//wire [bits-1:0] carries;



CLA #(bits) UUT(

.A(A),

.B(B),

.Cin(Cin),


.Sum(Sum),

.Cout(Cout)

//.carries(carries)

);


initial begin


/*

//4bit tests

{Cin, A, B} = 9'h0_F_1; //overflow

#5 {Cin, A, B} = 9'h0_A_4;

#5 {Cin, A, B} = 9'h1_4_7;

#5 {Cin, A, B} = 9'h0_9_C;

*/



//8 bit tests
```

```
{Cin, A, B} = 17'h0_FF_01; //overflow

#5 {Cin, A, B} = 17'h0_AC_47;

#5 {Cin, A, B} = 17'h1_BD_38;

#5 {Cin, A, B} = 17'h1_EF_27;




end



endmodule
```