Lab #2 The only time you have to do math

Hardware/Software Design of Embedded Systems

Iftidar Miah

7 March 2019

1. Purpose

   Create a single bit full adder component implemented to create a 4-bit ripple-carry adder. Design an Arithmetic Logic Unit (ALU) that will perform addition, subtraction, increments, decrements, negation, greater than logic, sll, srl, sra, logical nots, ands, ors, xors, xnors, nands, and nors. Implement the ALU design on the Zybo board with buttons, switches, clock, and LEDs.

2. Prelab

   i. Theory

   Single-bit full adder logic equations for sum and carryout are given by

   $\mathbf{S} = (\mathbf{A} \text{ XOR } \mathbf{B}) \text{ XOR } \mathbf{C_{IN}} = \mathbf{A} \oplus \mathbf{B} \oplus \mathbf{C}$

   $\mathbf{C_{OUT}} = ((\mathbf{A} \text{ XOR } \mathbf{B}) \text{ AND } \mathbf{C}) \text{ OR } (\mathbf{A} \text{ AND } \mathbf{B}) = (\mathbf{A} \oplus \mathbf{B})\,\mathbf{C} + \mathbf{AB}$

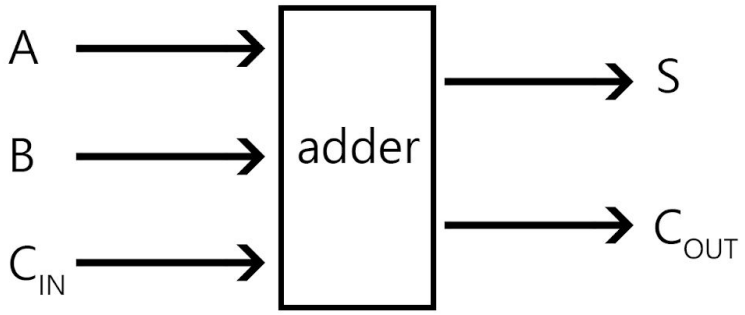
   ii. Truth Table

Single bit full adder

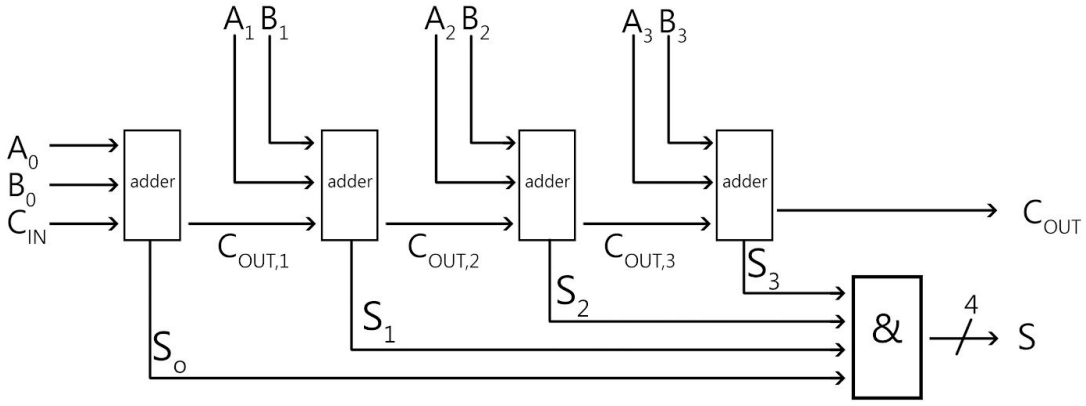
| **A** | **B** | $\mathbf{C_{IN}}$ | $\mathbf{(A \oplus B)}$ | $\mathbf{S = (A \oplus B) \oplus C}$ | $\mathbf{(A \oplus B)\,C}$ | **AB** | $\mathbf{C_{OUT} = (A \oplus B)\,C + AB}$ |
|---|---|---|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| 0 | 1 | 0 | 1 | 1 | 0 | 0 | 0 |
| 0 | 1 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 1 | 0 | 0 | 0 |
| 1 | 0 | 1 | 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 0 | 0 | 1 | 1 |
| 1 | 1 | 1 | 0 | 1 | 0 | 1 | 1 |

iii. Schematic

Single Bit Full Adder Black Box




4-Bit Ripple Carry Adder Block Diagram




3. Part 1

a. Theory

A single bit adder will add 3 bits (A, B, C_in) and output S and C_out. The logic functions for both are given by the following logic functions and behaves according to the truth table below.

**S** = (**A** XOR **B**) XOR $\mathbf{C_{IN}}$ = **A** $\oplus$ **B** $\oplus$ **C**

$\mathbf{C_{OUT}}$ = ((**A** XOR **B**) AND **C**) OR (**A** AND **B**) = (**A** $\oplus$ **B**) **C** + **AB**

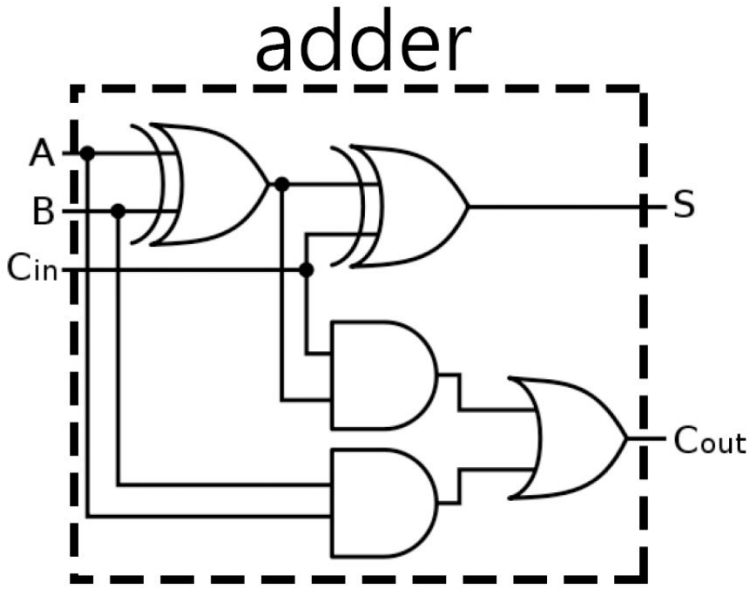
Using a structural model, we can create 4 instances of adder and chain them together to make a ripple carry adder that adds two 4 bit numbers and a single-bit carry in. The end result is a single bit carry out and 4-bit sum that successfully adds 4-bit numbers with a carry in.

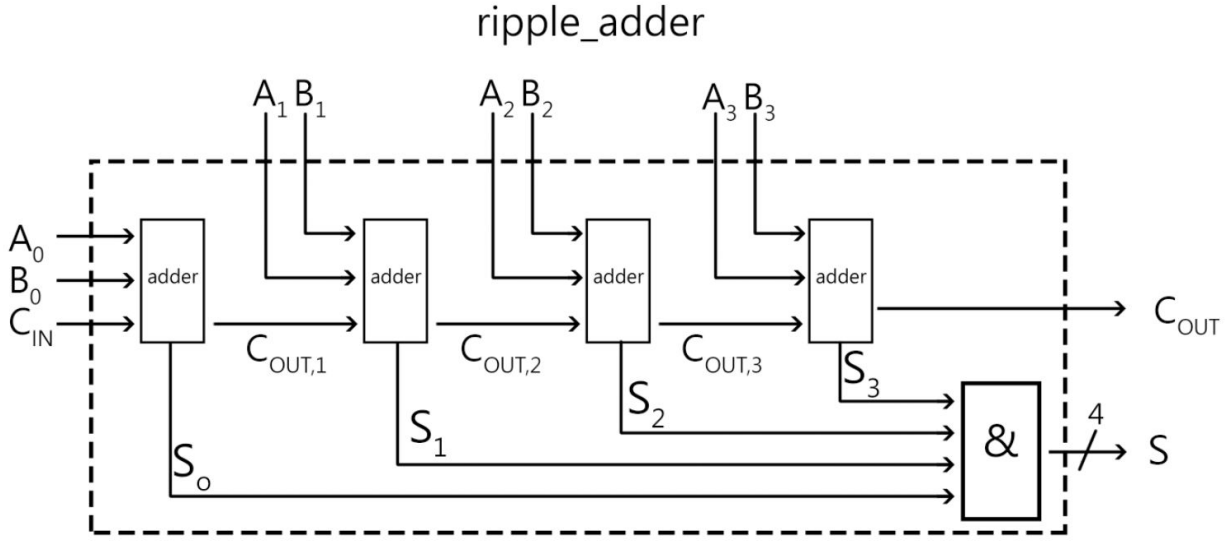
b. Truth Table

For a single bit adder

| **A** | **B** | **$C_{IN}$** | **$S = (A \oplus B) \oplus C$** | **$C_{OUT} = (A \oplus B)\, C + AB$** |
|---|---|---|---|---|
| 0 | 0 | 0 | 0 | 0 |
| 0 | 0 | 1 | 1 | 0 |
| 0 | 1 | 0 | 1 | 0 |
| 0 | 1 | 1 | 0 | 1 |
| 1 | 0 | 0 | 1 | 0 |
| 1 | 0 | 1 | 0 | 1 |
| 1 | 1 | 0 | 0 | 1 |
| 1 | 1 | 1 | 1 | 1 |

c. Schematic




d. Design

Adder.vhd (single bit full adder)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity adder is
    port(   A, B, C_in  : in    std_logic;
            S, C_out    : out std_logic);
end adder;

architecture adder_arch of adder is
```

```
begin
   S <= (A XOR B) XOR C_in;
   C_out <= ((A XOR B) AND C_in) OR (A AND B);
end adder_arch;
```

ripple_adder.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ripple_adder is
   port(   C_in    : in std_logic;
           A, B    : in std_logic_vector(3 downto 0);
           C_out   : out std_logic;
           S       : out std_logic_vector(3 downto 0));
end ripple_adder;

architecture ripple_adder_arch of ripple_adder is
   signal C    : std_logic_vector(2 downto 0); -- Carry ins/outs inside black box

   component adder
      port(   A, B, C_in  : in    std_logic;
              S, C_out    : out   std_logic);
   end component;

begin

   ADD0: adder
      Port Map(   A => A(0),
                  B => B(0),
                  C_in => C_in,
                  S => S(0),
                  C_out => C(0)
      );

   ADD1: adder
      Port Map(   A => A(1),
                  B => B(1),
                  C_in => C(0),
                  S => S(1),
                  C_out => C(1)
      );

   ADD2: adder
      Port Map(   A => A(2),
                  B => B(2),
                  C_in => C(1),
                  S => S(2),
                  C_out => C(2)
     );

   ADD3: adder
```

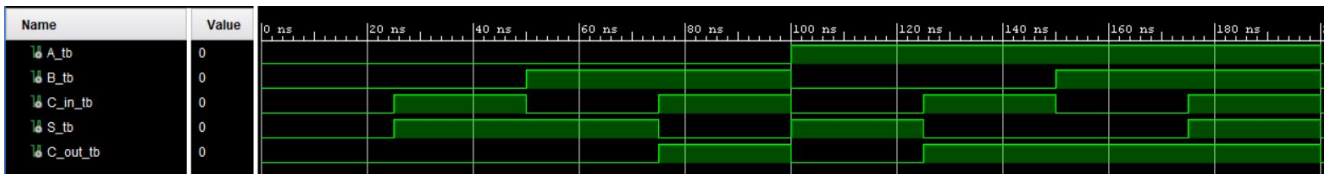
```
    Port Map(   A => A(3),
                B => B(3),
                C_in => C(2),
                S => S(3),
                C_out => C_out
    );

end ripple_adder_arch;
```

e. Test

Adder_tb.vhd (for single bit full adder)

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std;

entity adder_tb is
end adder_tb;

architecture adder_arch_tb of adder_tb is
   signal A_tb, B_tb, C_in_tb  : std_logic := '0';    -- Input test bench signal declarations
   signal S_tb, C_out_tb       : std_logic;    -- Output test bench signal declarations

   component adder
      port(   A, B, C_in  : in    std_logic;
              S, C_out    : out std_logic);
   end component;

begin
   A_proc: process
   begin
      wait for 100ns;
      A_tb <= '1';
      wait for 100ns;
      A_tb <= '0';
   end process A_proc;

   B_proc: process
   begin
      wait for 50ns;
      B_tb <= '1';
      wait for 50ns;
      B_tb <= '0';
   end process B_proc;

   C_in_proc: process
   begin
      wait for 25ns;
      C_in_tb <= '1';
      wait for 25ns;
      C_in_tb <= '0';
   end process C_in_proc;
```

```
    DUT: adder
        Port Map(   A => A_tb,
                    B => B_tb,
                    C_in => C_in_tb,
                    S => S_tb,
                    C_out => C_out_tb
        );

end adder_arch_tb;
```

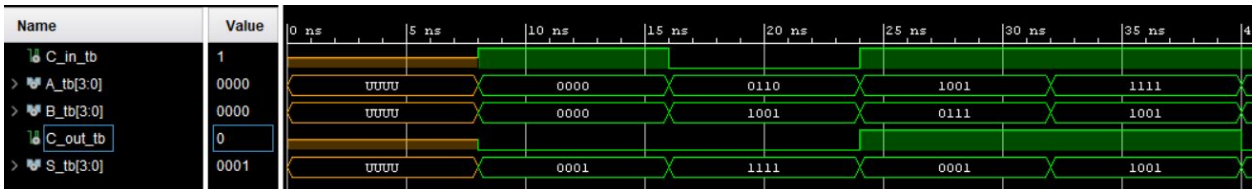



Simulation for single bit adder matches truth table from prelab for a single adder.

Ripple_adder_tb.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity ripple_adder_tb is
end ripple_adder_tb;

architecture ripple_adder_arch_tb of ripple_adder_tb is
    signal C_in_tb      : std_logic;
    signal A_tb, B_tb   : std_logic_vector(3 downto 0);
    signal C_out_tb     : std_logic;
    signal S_tb         : std_logic_vector(3 downto 0);

    component ripple_adder
        port(   C_in    : in std_logic;
                A, B    : in std_logic_vector(3 downto 0);
                C_out   : out std_logic;
                S       : out std_logic_vector(3 downto 0));
    end component;

begin

    A_proc: process
    begin
        wait for 8ns;
        A_tb <= "0000";
        wait for 8ns;
        A_tb <= "0110";
        wait for 8ns;
        A_tb <= "1001";
        wait for 8ns;
        A_tb <= "1111";
    end process A_proc;
```

```
    B_proc: process
    begin
        wait for 8ns;
        B_tb <= "0000";
        wait for 8ns;
        B_tb <= "1001";
        wait for 8ns;
        B_tb <= "0111";
        wait for 8ns;
        B_tb <= "1001";
    end process B_proc;

    C_in_proc: process
    begin
        wait for 8ns;
        C_in_tb <= '1';
        wait for 8ns;
        C_in_tb <= '0';
        wait for 8ns;
        C_in_tb <= '1';
        wait for 8ns;
        C_in_tb <= '1';
    end process C_in_proc;

    DUT: ripple_adder
        Port Map(   C_in => C_in_tb,
                    A => A_tb,
                    B => B_tb,
                    C_out => C_out_tb,
                    S => S_tb
        );

end ripple_adder_arch_tb;
```




Simulation for ripple adder shows 4 simple additions.

$0000 + 0000 + 1 = 0001$, $C_{OUT} = 0$

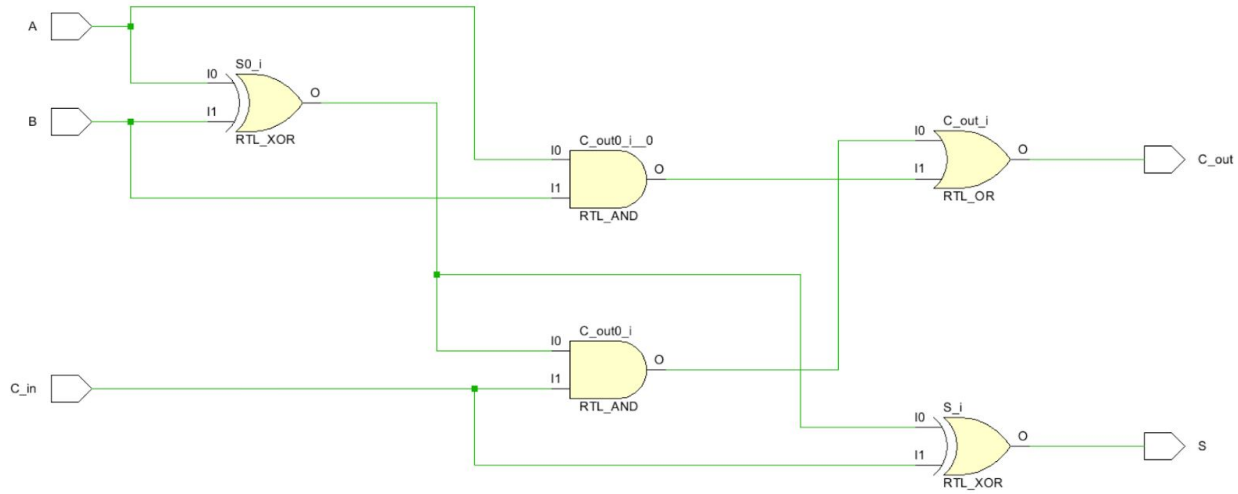
$0110 + 1001 + 0 = 1111$, $C_{OUT} = 0$

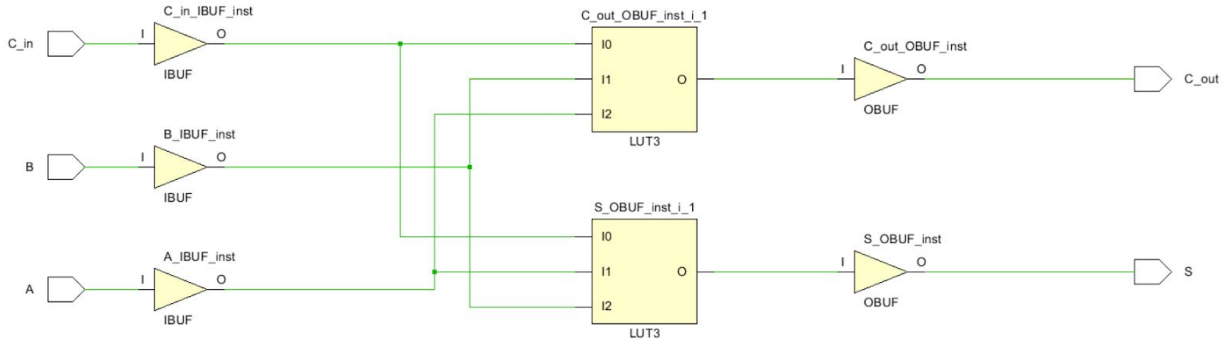
$1001 + 0111 + 1 = 0001$, $C_{OUT} = 1$

$1111 + 1001 + 1 = 1001$, $C_{OUT} = 1$

f. Implementation

## Single Bit Full Adder



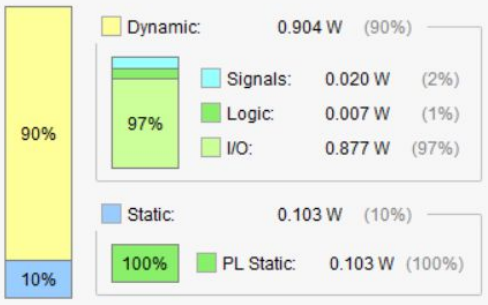




**Utilization** **Post-Synthesis** | Post-Implementation

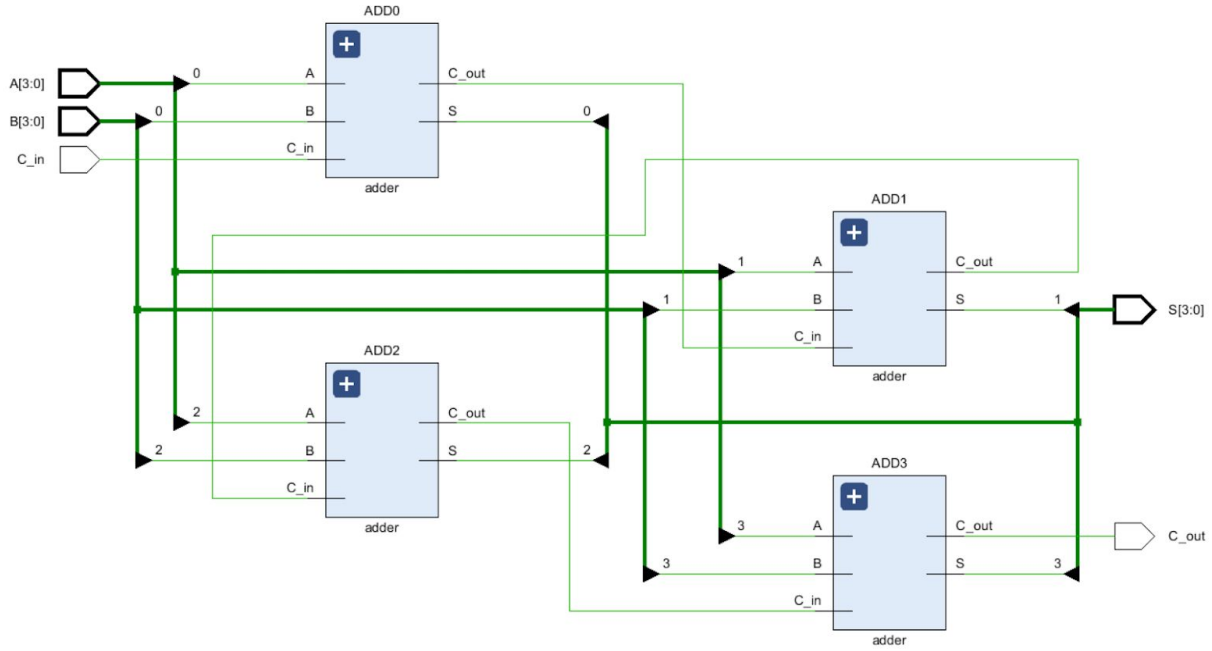
Graph | **Table**

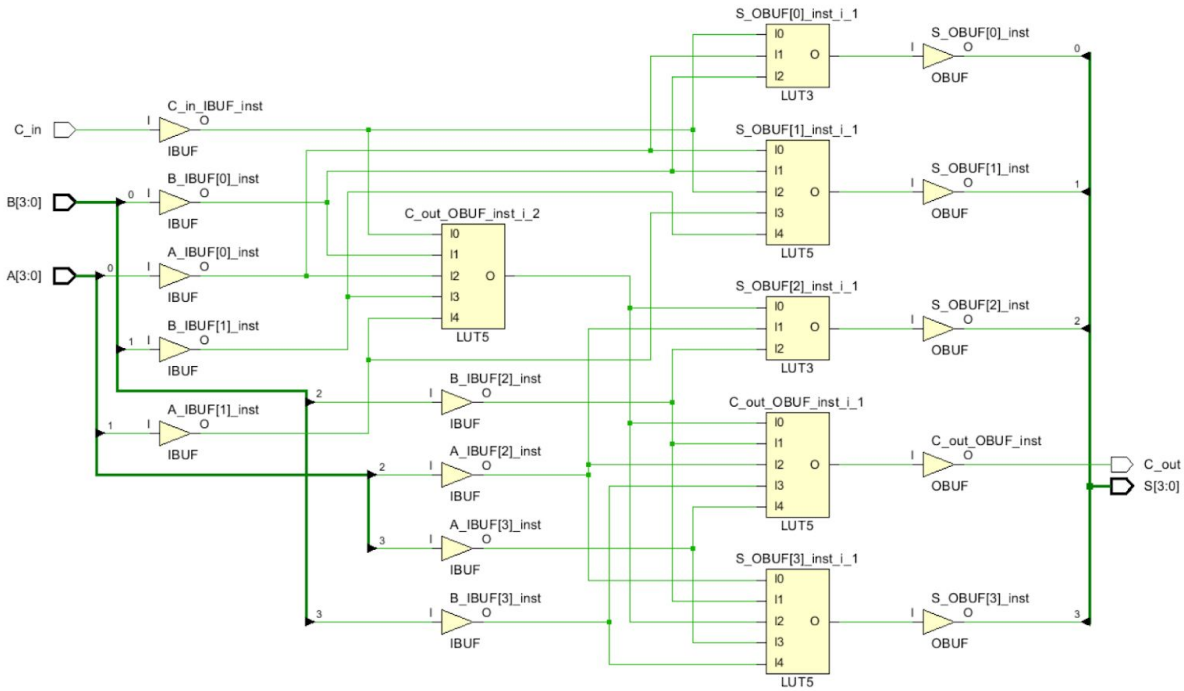
| Resource | Estimation | Available | Utilization % |
|---|---|---|---|
| LUT | 1 | 17600 | 0.01 |
| IO | 5 | 100 | 5.00 |

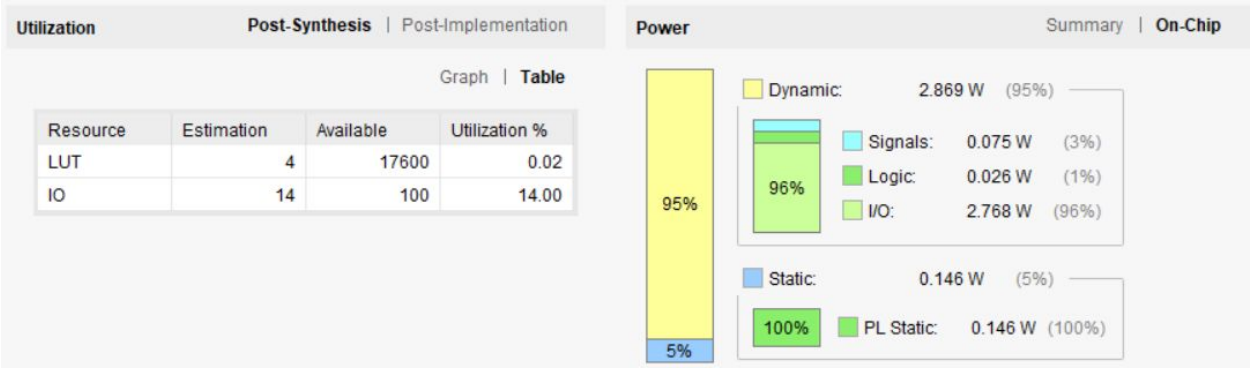
**Power** Summary | **On-Chip**




## 4-bit Ripple Carry Adder


ADD0
A[3:0]
B[3:0]
C_in
A
B
C_in
C_out
S
adder
ADD1
ADD2
ADD3
S[3:0]
C_out


C_in
C_in_IBUF_inst
B[3:0]
B_IBUF[0]_inst
B_IBUF[1]_inst
B_IBUF[2]_inst
B_IBUF[3]_inst
A[3:0]
A_IBUF[0]_inst
A_IBUF[1]_inst
A_IBUF[2]_inst
A_IBUF[3]_inst
IBUF
C_out_OBUF_inst_i_2
LUT5
S_OBUF[0]_inst_i_1
S_OBUF[1]_inst_i_1
S_OBUF[2]_inst_i_1
S_OBUF[3]_inst_i_1
LUT3
C_out_OBUF_inst_i_1
S_OBUF[0]_inst
S_OBUF[1]_inst
S_OBUF[2]_inst
S_OBUF[3]_inst
C_out_OBUF_inst
OBUF
C_out
S[3:0]




4. Part 2
    a. Theory

    A 16 function 4-bit ALU uses a 4-bit OPCODE to decide one of 15 possible arithmetic operations to implement. The operands A and B are 4-bits as well. The 4 switches on a Zybo board will set the value of the OPCODE, A, or B on the next rising edge of a clock when buttons 2, 1, or 0 are pressed respectively. The result of the operation is displayed on the 4 LEDs. Button 3 clears the values of OPCODE, A, and B and sets them to the default of "0000." The operations are

| opcode | function | opcode | function |
|---|---|---|---|
| x"0" | $A + B$ | x"8" | $A >>> 1$ (right shift arithmetic) |
| x"1" | $A - B$ | x"9" | not A |
| x"2" | $A + 1$ | x"A" | $A$ and $B$ |
| x"3" | $A - 1$ | x"B" | $A$ or $B$ |
| x"4" | $0 - A$ | x"C" | $A$ xor $B$ |
| x"5" | $A > B$ (greater than - see note) | x"D" | $A$ xnor $B$ |
| x"6" | $A << 1$ (left shift logical) | x"E" | $A$ nand $B$ |
| x"7" | $A >> 1$ (right shift logical) | x"F" | $A$ nor $B$ |

    b. Schematic

    c. Design

    my_alu.vhd

```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity my_alu is
```

```
   port(   clk     : in    std_logic;
           A, B    : in    std_logic_vector(3 downto 0);
           Opcode  : in    std_logic_vector(3 downto 0);
           ALU_out : out   std_logic_vector(3 downto 0));
end my_alu;

architecture my_alu_arch of my_alu is
   signal funct    : std_logic_vector(3 downto 0) := "0000"; -- Intermediate signal for
ALU_out
begin

   ALU_out <= funct;   -- funct will hold values from alu operations

   ALU_proc: process(clk)
   begin
      if rising_edge(clk) then
         case (to_integer(unsigned(Opcode))) is
            when 0  => funct <= std_logic_vector(unsigned(A) + unsigned(B));
            when 1  => funct <= std_logic_vector(unsigned(A) - unsigned(B));
            -- Case 2: convert int +1 to length 4 unsigned then add/sub to unsigned A
            -- Then convert all to std_logic_vector before assigning to funct
            when 2      => funct <= std_logic_vector((unsigned(A) + to_unsigned(1,4)));
            when 3      => funct <= std_logic_vector((unsigned(A) - to_unsigned(1,4)));
            when 4      => funct <= std_logic_vector((to_unsigned(0,4) - unsigned(A)));
            when 5      =>
               if (unsigned(A) > unsigned(B)) then
                  funct <=    "0001";
               else
                  funct <= (others => '0');
               end if;
            when 6      => funct <= std_logic_vector(unsigned(A) sll 1);
            when 7      => funct <= std_logic_vector(unsigned(A) srl 1);
            when 8      =>
               if (A(3) = '1') then
                  funct <= ('1' & std_logic_vector(unsigned(A(3 downto 1))));
               else
                  funct <= std_logic_vector(unsigned(A) srl 1);
               end if;
            when 9      => funct <= std_logic_vector(not unsigned(A));
            when 10     => funct <= std_logic_vector(unsigned(A) AND unsigned(B));
            when 11     => funct <= std_logic_vector(unsigned(A) OR unsigned(B));
            when 12     => funct <= std_logic_vector(unsigned(A) XOR unsigned(B));
            when 13     => funct <= std_logic_vector(unsigned(A) XNOR unsigned(B));
            when 14     => funct <= std_logic_vector(unsigned(A) NAND unsigned(B));
            when 15     => funct <= std_logic_vector(unsigned(A) NOR unsigned(B));
            when others => funct <= (others => '0');
         end case;
      end if;
   end process ALU_proc;
end my_alu_arch;
```

Debounce.vhd

```
library IEEE;
```

```
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity debounce is
   port(   clk, btn    : in    std_logic;
           dbnc        : out   std_logic);
end debounce;

architecture debounce_arch of debounce is
   signal count    : std_logic_vector(2 downto 0) := (others => '0'); -- 3 bit counter signal
   signal shift    : std_logic_vector(1 downto 0) := (others => '0'); -- 2 shift register

begin

   debounce: process(clk)
   begin
      if rising_edge(clk) then
         shift <= shift(0) & btn;    -- Shift left 1, put btn at shift(0)

         if ((shift(1) = '1') AND (unsigned(count) /= 4)) then      -- Check if shift(1) = 1 then
check and increment counter
            count <= std_logic_vector(unsigned(count) + 1);     -- Increment count when
shift(0) = 1 and count < 4
         elsif ((shift(1) = '1') AND (unsigned(count) = 4)) then
            dbnc <= '1';        -- When count = 4, debounce output = 1. Now count and dbnc
won't change until next shift(0) = 0
         else
            dbnc    <= '0';
            count <= (others => '0');          -- Reset count if shift(0) = 0
         end if;
      end if;
   end process debounce;

end debounce_arch;
```

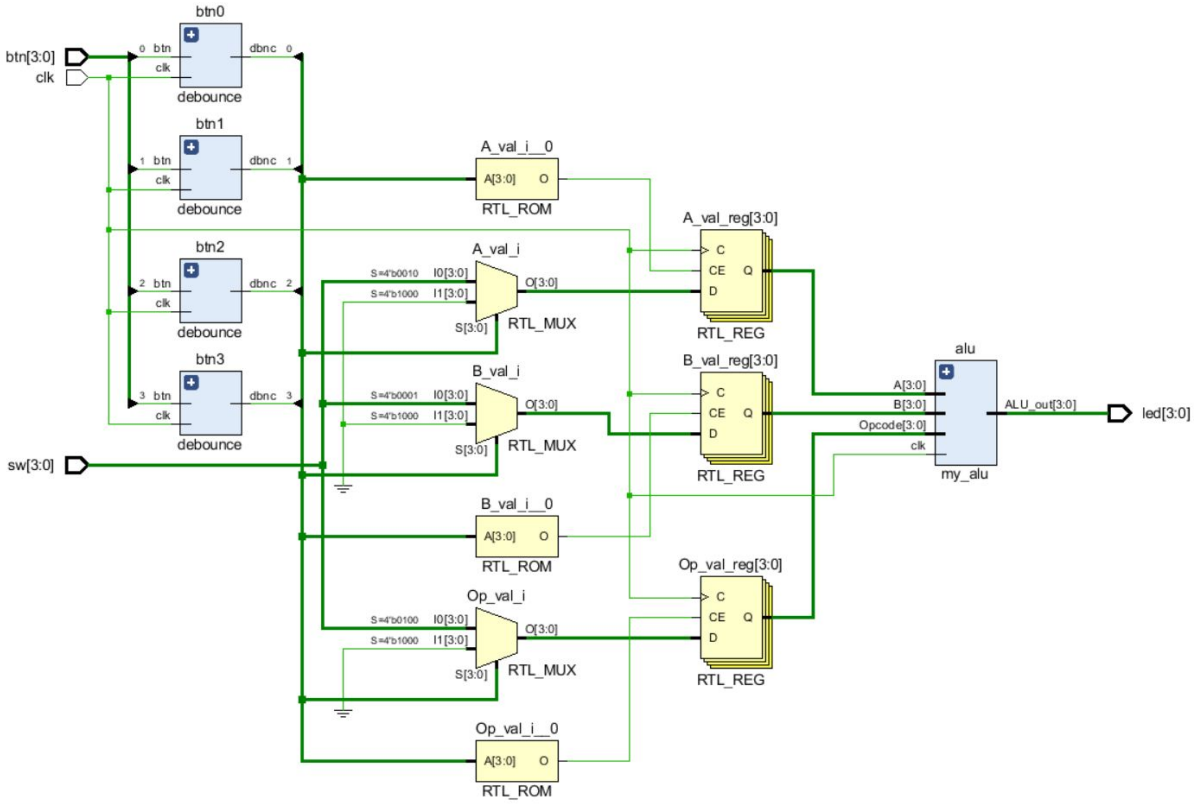
Alu_tester.vhd

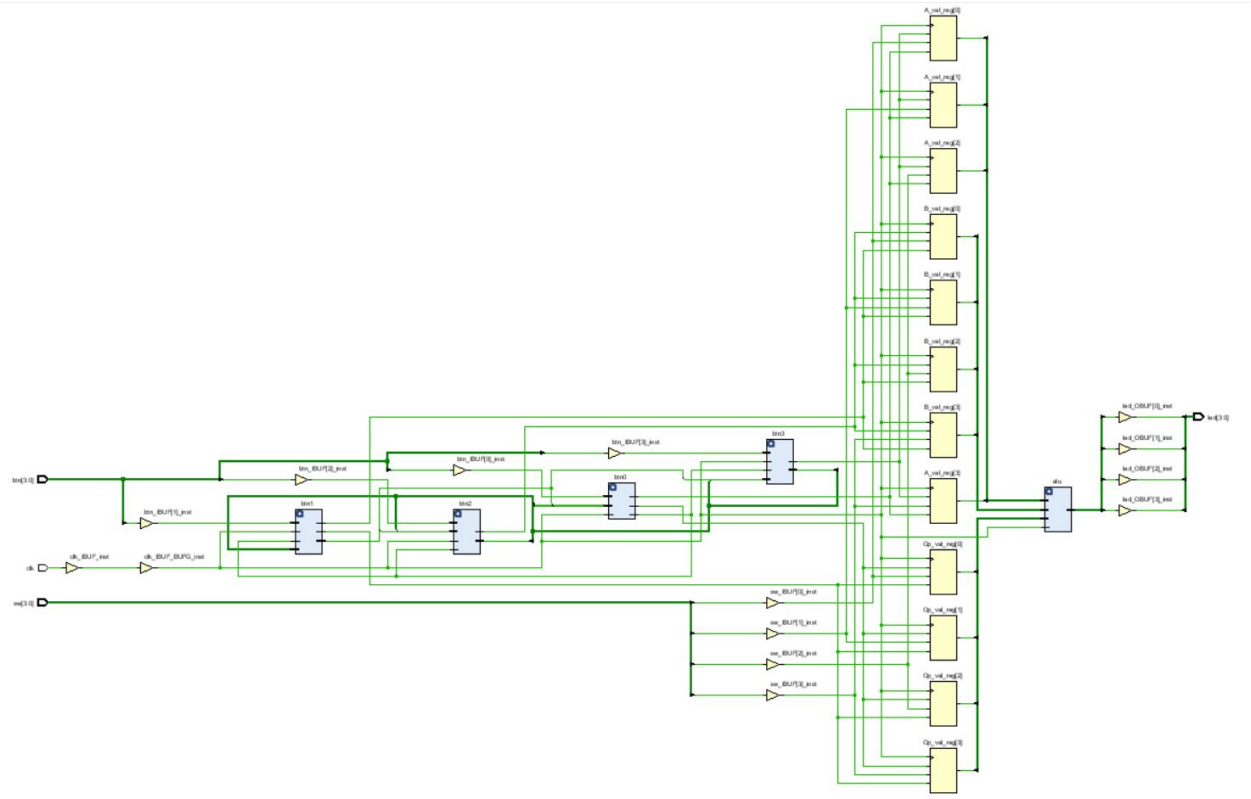
```
library IEEE;
use IEEE.std_logic_1164.all;
use IEEE.numeric_std.all;

entity alu_tester is
   port(   clk         : in    std_logic;
           btn, sw     : in    std_logic_vector(3 downto 0);
           led         : out   std_logic_vector(3 downto 0));
end alu_tester;

architecture alu_tester_arch of alu_tester is
   signal btn_val  : std_logic_vector(3 downto 0) := "0000";
   signal A_val    : std_logic_vector(3 downto 0) := "0000";
   signal B_val    : std_logic_vector(3 downto 0) := "0000";
   signal Op_val   : std_logic_vector(3 downto 0) := "0000";
```

```
    component my_alu
       port(   clk     : in    std_logic;
               A, B    : in    std_logic_vector(3 downto 0);
               Opcode  : in    std_logic_vector(3 downto 0);
               ALU_out : out   std_logic_vector(3 downto 0));
    end component;

    component debounce
       port(   clk, btn   : in    std_logic;
               dbnc       : out   std_logic);
    end component;

begin

    alu_test_proc: process(clk)
    begin
       if rising_edge(clk) then
          case (btn_val) is
             when "0001" => B_val <= sw;
             when "0010" => A_val <= sw;
             when "0100" => Op_val <= sw;
             when "1000" =>
                Op_val  <= (others => '0');
                A_val   <= (others => '0');
                B_val   <= (others => '0');
             when others =>
          end case;
       end if;
    end process alu_test_proc;

    -- 4 debounce circuits for 4 buttons
    btn0:   debounce
       Port Map(
                clk => clk,
                btn => btn(0),
                dbnc => btn_val(0)
       );

    btn1:   debounce
       Port Map(
                clk => clk,
                btn => btn(1),
                dbnc => btn_val(1)
       );

    btn2:   debounce
    Port Map(
               clk => clk,
               btn => btn(2),
               dbnc => btn_val(2)
    );

    btn3:   debounce
```

```
Port Map(
        clk => clk,
        btn => btn(3),
        dbnc => btn_val(3)
);

alu:    my_alu
Port Map(   clk     => clk,
        A       => A_val,
        B       => B_val,
        Opcode  => Op_val,
        ALU_out => led
);


end alu_tester_arch;
```

d. Implementation



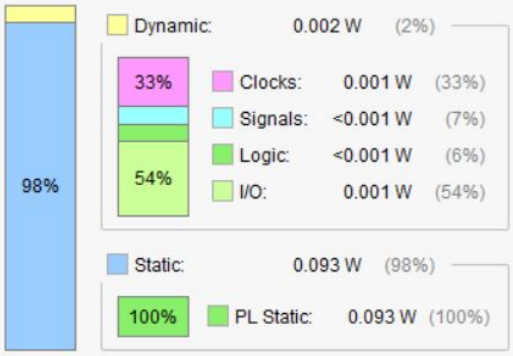


**Utilization** **Post-Synthesis** | Post-Implementation

Graph | **Table**

| Resource | Estimation | Available | Utilization % |
|---|---|---|---|
| LUT | 35 | 17600 | 0.20 |
| FF | 40 | 35200 | 0.11 |
| IO | 13 | 100 | 13.00 |
| BUFG | 1 | 32 | 3.13 |

**Power** Summary | **On-Chip**



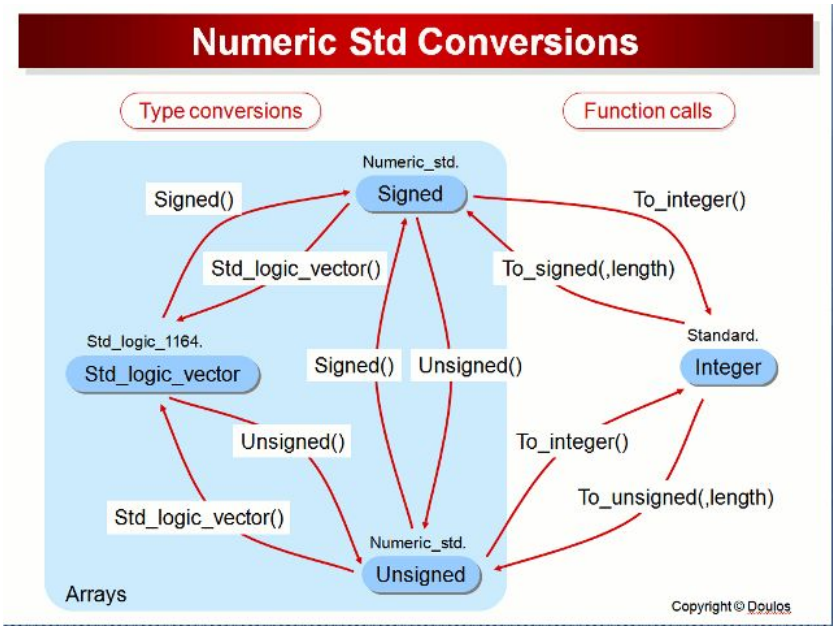

On Zybo_Master.xdc, I uncommented the lines for the clock, 4 LEDs, Switches, and Buttons so I can use them to assign values to A, B, $C_{IN}$ and view the output of the ALU.

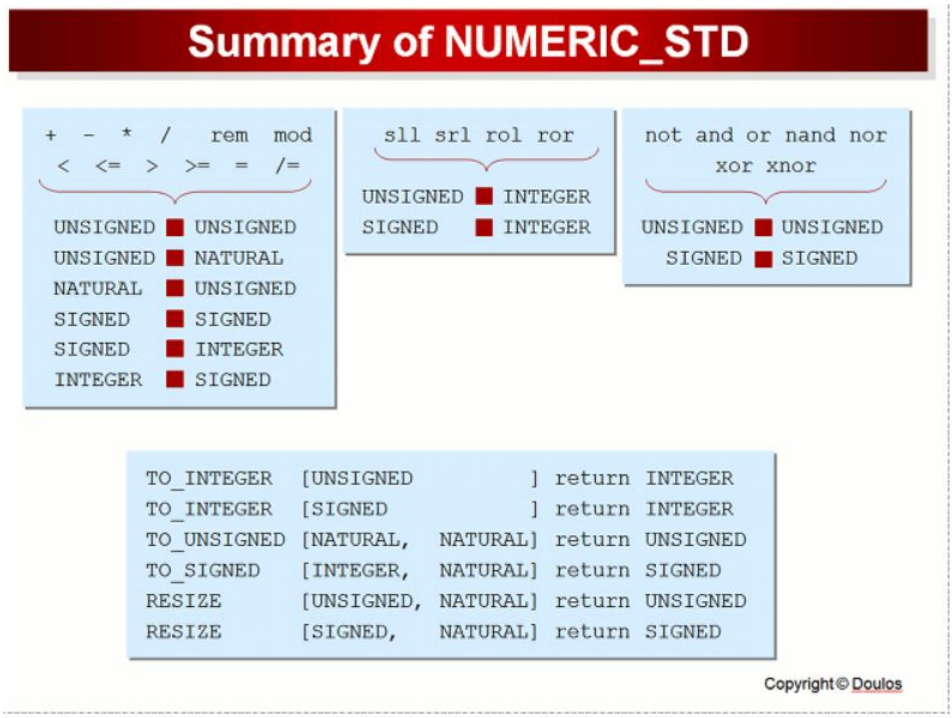
5. Discussion
    a. Observation/Discoveries

        The first part is just more practice with structural models for the first part. The adder isn't designed to be synchronous so concurrent assignments can be made

(data-flow model) for a single bit adder. Then the structural model is used to tie them together into a 4-bit ripple carry adder.

For the ALU, a lot of data type casting has to be done using the numeric_std functions. Certain numeric_std operations can only be done between certain data types so we have to cast it into that type before performing any arithmetic.



I can also hover over signals and it would tell me the data type in Vivado. There was a situation with the buttons where I only had to do something in the cases “0001”, “0010”, “0100”, and “1000.” When using a case statement for them, although I wanted nothing to happen for any other button inputs, I had to put an “ when others =>” line to get the code to work. I didn’t put any operations or assignments for that case and left it blank.

b. Questions/Follow Up

The first part is straightforward and I have no questions about logical operations and structural models. I also understand the need to cast data into different types. But I feel that keeping track of which signals are what type is tedious.