Synthesizable VHDL Design for FPGAs

Eduardo Augusto Bezerra
Djones Vinicius Lettnin

Universidade Federal de Santa Catarina
Florianópolis, Brazil

August 2013

Contents

Chapter 10. Arithmetic Circuits

In previous chapters, the ‘+’ VHDL operator is used in the adder design. In this chapter the adder component is conceived using logic gates, at the structural level. At the end of the chapter, the students should be able:

- to understand the design process of adders using structural VHDL;
- to simulate the calculator case study using the new adder;
- to prototype the calculator case study in an FPGA board.

10.1 Half-Adder, Full-Adder, Ripple-Carry Adder

The adder is a fundamental circuit in digital systems. In an *n* bits adder, the main problem is the delay needed for the carry propagation. The carry propagation is a concern also in a multiplier circuit. The concepts behind the design of adders and multipliers are employed in the solution of several similar digital system problems. In Figure 10.1, a four bits value ($A_{3..0}$) is added to another four bits value ($B_{3..0}$) resulting in a four bits summation ($S_{3..0}$). All bit position sum operations result also in a carry ($C_{3..0}$).

$$\begin{array}{cccccccl}
 & C_4 & C_3 & C_2 & C_1 & & \leftarrow & \texttt{carry}\\
 & & A_3 & A_2 & A_1 & A_0 & \leftarrow & \texttt{1}^{\texttt{st}}\ \texttt{operand}\\
+ & & B_3 & B_2 & B_1 & B_0 & \leftarrow & \texttt{2}^{\texttt{nd}}\ \texttt{operand}\\
\hline
 & & S_3 & S_2 & S_1 & S_0 & \leftarrow & \texttt{sum}
\end{array}$$

Figure 10.1. A four bits sum operation.

Two different components can be used in the design of this four bits adder. The half-adder component, shown in

Figure 10.2, has two inputs (A0 and B0) and two outputs (S0 and C1). The full-adder component, shown in

Figure 10.3, has three inputs (Ci, Ai and Bi) and two outputs (Si and Ci+1). The half-adder component is used to perform the least significant bits (LSB) addition, and the full-adder component for the remaining bits.

A_0	B_0	S_0	C_1
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

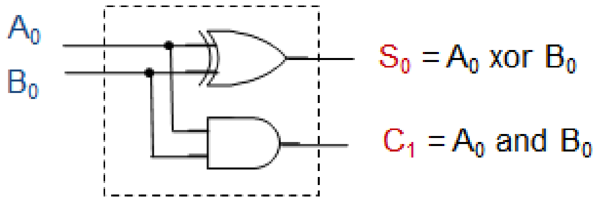


Figure 10.2. Half-adder (HA) component. Truth table, schematic, and logic equations.

C_i	A_i	B_i	S_i	C_{i+1}
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

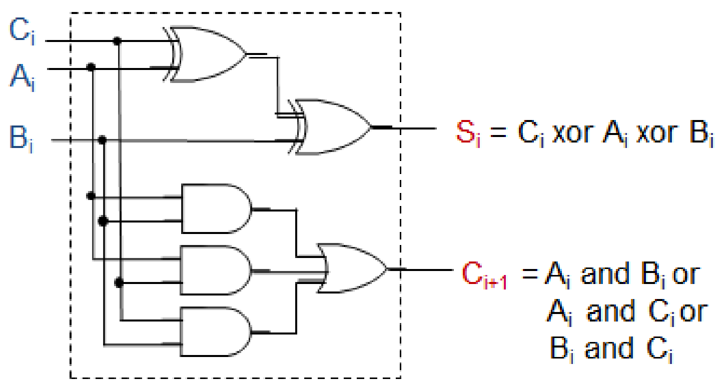

C_i
A_i
B_i
S_i = C_i xor A_i xor B_i
C_{i+1} = A_i and B_i or A_i and C_i or B_i and C_i

Figure 10.3. Full-adder (FA) component. Truth table, schematic, and logic equations.

In an FPGA based implementation, as there are no actual logic gates available in the device, an *n* bits adder can be implemented using only full-adders with no major resource usage penalties. The adder functionality will be implemented using the FPGA's look-up tables (LUTs), and the only concern is the amount of input signals, as there is no difference between a two input function performing a single AND operation (C_1 in the half-adder), or three ANDs and two ORs (C_{i+1} in the full-adder). Both functions will employ exactly the same amount of FPGA resources (LUTs).

There are several topologies available for implementing parallel adders. Figure 10.4 shows a four bits Ripple-Carry Adder (RCA), which can be used to implement the operation described in Figure 10.1. In this circuit, full-adders (FA) are used in all bit positions, including the LSB, where a half-adder could have been used. Having an FA in the LSB position is interesting also in order to have a component that can be used as a building block for larger adders or subtractors.

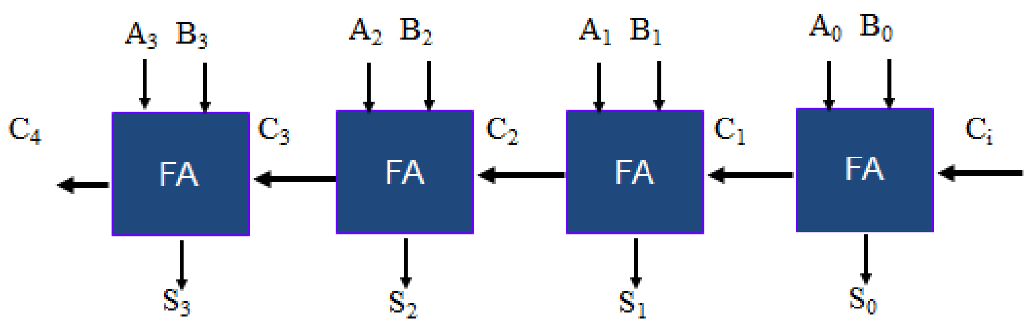


Figure 10.4. A four bits Ripple-Carry Adder (RCA).

As shown in Figure 10.1, each bit of the result (S_i) can only be obtained after the previous adder stage has finished its carry (C_{i-1}) calculation. Consequently, an RCA component provides a result ($S_{n..0}$) only after all carry bits ($C_{n..1}$) are stable.

When working with signed values, an RCA component can be adapted in order to implement an add / subtract circuit. In Figure 10.5, four multiplexers and four inverters are used in the B inputs of the RCA. When C_i is '1' the circuit works as a subtractor. In this case, the circuit performs an add operation between A and the two's complement of B:

S = A + B' + 1
S = A + (-B)
S = A – B

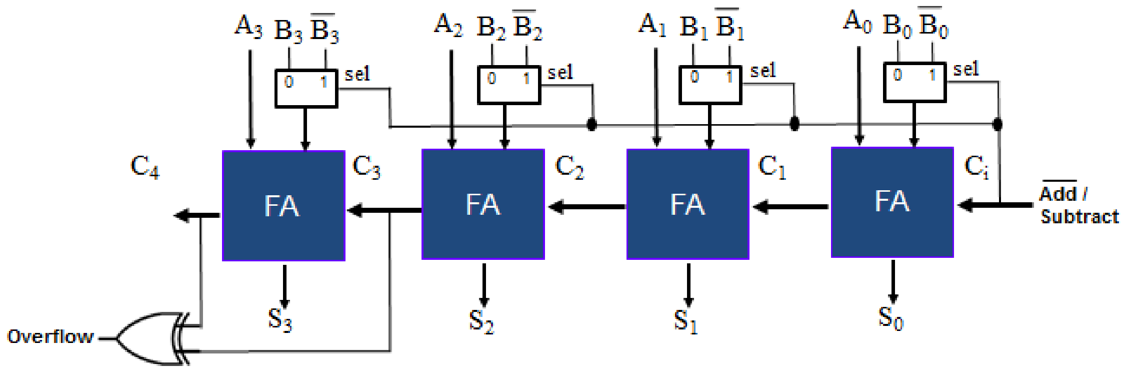

A_3 B_3 $\overline{B_3}$
A_2 B_2 $\overline{B_2}$
A_1 B_1 $\overline{B_1}$
A_0 B_0 $\overline{B_0}$
0 1 sel
C_4
C_3
C_2
C_1
C_i
FA
$\overline{\text{Add}}$ / Subtract
S_3
S_2
S_1
S_0
Overflow

Figure 10.5. Add / subtract circuit based on an RCA component.

The RCA delay to provide a valid (stable) result depends on the input data. In order to have a better understanding of the carry propagation delay problem, a *time unit* is used as a performance figure. A *time unit* is the period required by a component to evaluate one level of logic gates. For instance, the half-adder component shown in Figure 10.2 has just one level of logic gates, and all operations in this component are evaluated in one time unit. The full-adder shown in Figure 10.3 has two levels of logic gates, and the operations are evaluated in two time units.

Considering A = 0101 and B = 1010 (with no carries), the circuit in Figure 10.4 provides a valid result in two time units (2t). In the worst case scenario, 8t is the maximum delay acceptable for a four bits RCA. Larger adders would result in unacceptable delay figures. An 8 bits RCA should have a maximum delay of 16t, a 32 bits RCA the delay should be 64t, and so on.

Proposed solutions to decrease the delay usually perform the carry calculation in parallel, instead of the sequential approach employed in the RCA topology. The *Carry Lookahead Adder* (CLA) uses the functions *propagate* and *generate* in order to perform the parallel calculation of the carry. CLAs are faster than RCAs for 4 bits operands max. For longer length operands, the CLA carry generation logic complexity make it slower than the RCA. The *Carry Select Adder* (CSA) uses two FAs for each bit operation and, as a consequence, the carry generation is faster than the RCA and CLA, when considering larger operands.

Figure 10.6 shows a perform analysis between the adders. CSA and CLA present similar performance figures when for operands up to 10 bits long. For larger operands, the CSA circuit is the fastest adder. For instance, for 64 bits long operands, RCA takes more than 30 ns to provide a result, CLA takes more than 15 ns, and CSA provides the result in just 5 ns.

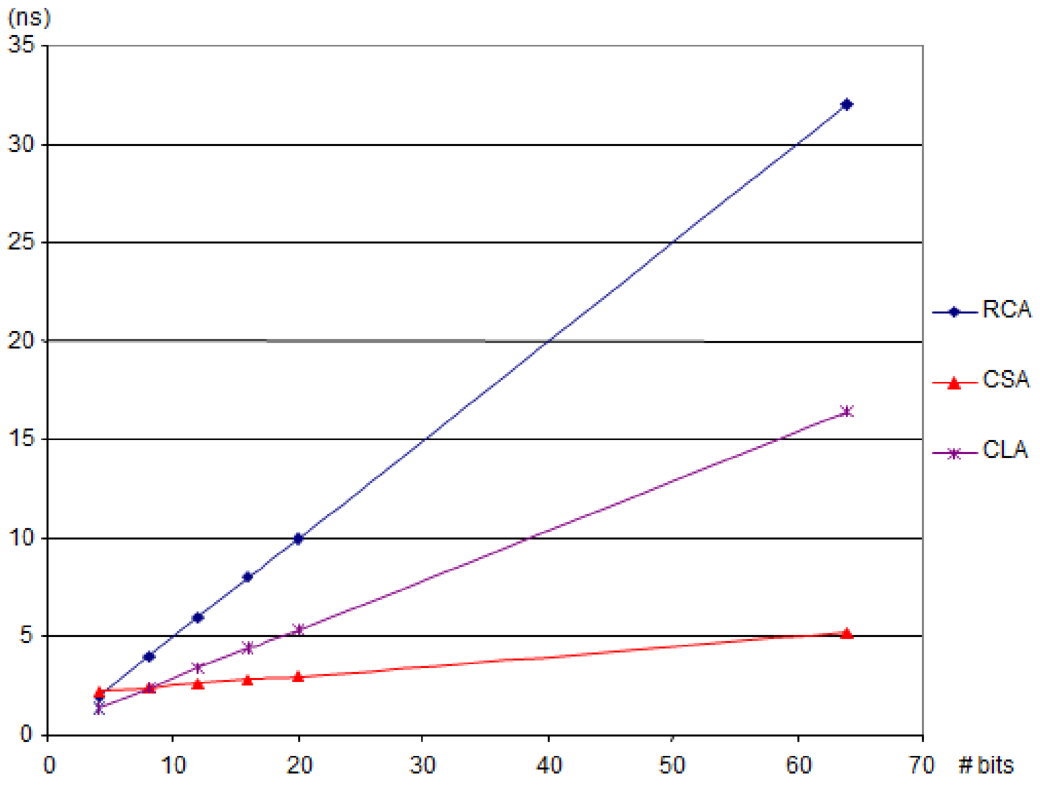
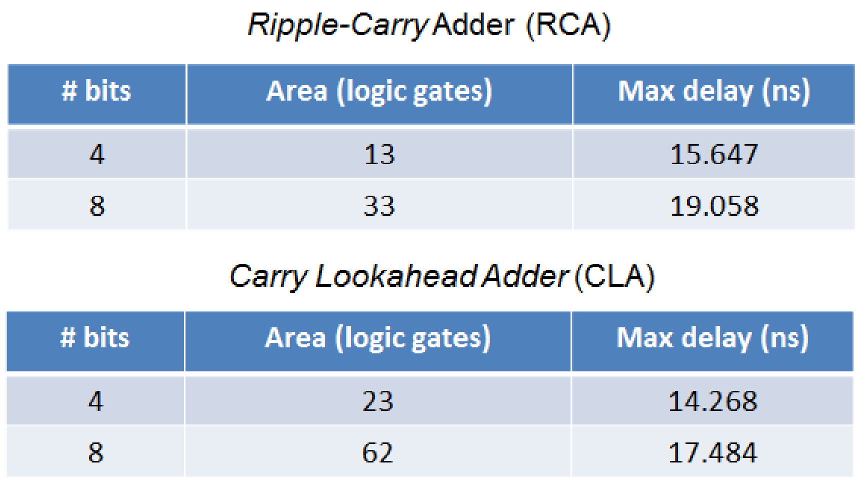


Figure 10.6. Carry propagation delay for the three adders.

The drawback is that faster adders require more hardware resources to be implemented. In Figure 10.7 it is possible to observe that the synthesized CLA takes almost twice the area than the RCA. On the other hand, CLA is about 10% faster than RCA.



Ripple-Carry Adder (RCA)

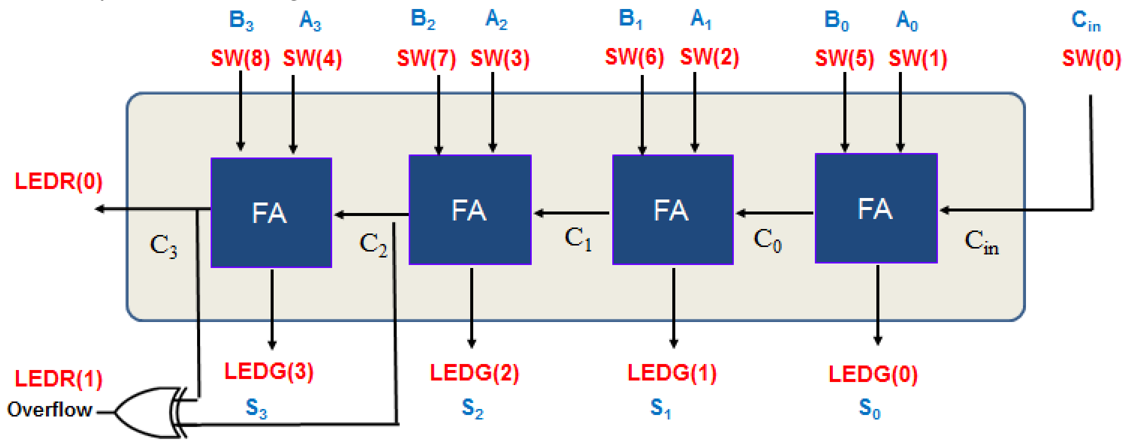
# bits	Area (logic gates)	Max delay (ns)
4	13	15.647
8	33	19.058

Carry Lookahead Adder (CLA)

# bits	Area (logic gates)	Max delay (ns)
4	23	14.268
8	62	17.484

Figure 10.7. Synthesis results summary for an RCA and a CLA implementation.

RCA is the slowest adder, but it has the less complex implementation. Two VHDL implementations of the RCA are discussed next, and compared to the synthesis tool built-in adder alternative. Figure 10.8 shows the block diagram of the 4 bits RCA case study. This circuit considers two's complement signed numbers. The red LEDs 0 and 1 are used to indicate carry out and overflow, respectively. The green LEDs 0 to 3 provide sum results. The inputs are provided by the SW switches, including A and B operands, and also the carry in. Internal signals are used to connect all carries between FAs.



B3 A3 B2 A2 B1 A1 B0 A0 Cin
SW(8) SW(4) SW(7) SW(3) SW(6) SW(2) SW(5) SW(1) SW(0)
LEDR(0)
FA
FA
FA
FA
C3 C2 C1 C0 Cin
LEDR(1)
Overflow
LEDG(3) LEDG(2) LEDG(1) LEDG(0)
S3 S2 S1 S0

Figure 10.8. Block diagram of the 4 bits RCA circuit implemented in VHDL.

In the first implementation, the A and B internal signals are used just to make the code more readable. On lines 13 and 14 in

Figure 10.9, the FPGA board switches SW4..1 and SW8..5 are assigned to the internal signals A3..0 and B3..0, respectively.

```
 1  LIBRARY ieee;
 2  USE ieee.std_logic_1164.all;
 3  ENTITY RCA IS
 4      PORT (
 5          SW : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
 6          LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
 7          LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
 8      );
 9  END RCA;
10  ARCHITECTURE stru OF RCA IS
11      signal carry, A, B: std_logic_vector(3 downto 0);
12  BEGIN
13      A <= SW(4 downto 1);
14      B <= SW(8 downto 5);
15      LEDG(0)  <= ((A(0) xor B(0)) xor SW(0));
16      carry(0) <=  (A(0) and B(0)) or (A(0) and SW(0)) or (B(0) and SW(0));
17      LEDG(1)  <= ((A(1) xor B(1)) xor carry(0));
18      carry(1) <=  (A(1) and B(1)) or (A(1) and carry (0)) or (B(1) and carry(0));
19      LEDG(2)  <= ((A(2) xor B(2)) xor carry(1));
20      carry(2) <=  (A(2) and B(2)) or (A(2) and carry(1)) or (B(2) and carry(1));
21      LEDG(3)  <= ((A(3) xor B(3)) xor carry(2));
22      carry(3) <=  (A(3) and B(3)) or (A(3) and carry(2)) or (B(3) and carry(2));
23      LEDR(0)  <= carry(3);                -- carry out
24      LEDR(1)  <= carry(3) xor carry(2);   -- overflow
25  END stru;
```

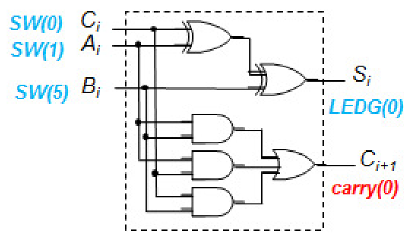
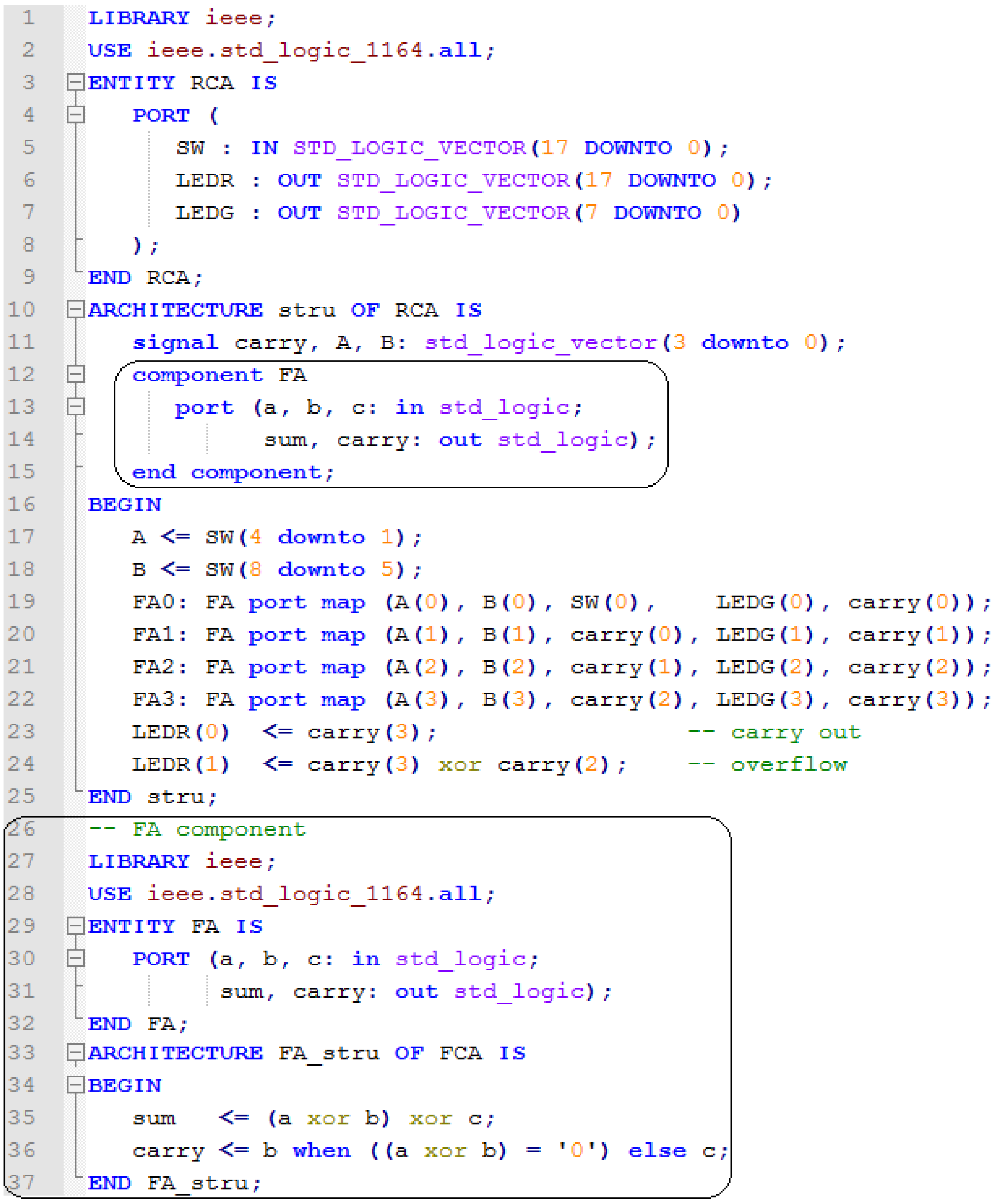


Figure 10.9. Solution I - RCA implementation using logic equations.

In the second implementation, the FA circuit is written as a VHDL component, making the RCA design more organized and easier to follow. The FA entity / architecture start on line 27 in Figure 10.10. The four FA instances are created and interconnected on lines 19 through 22.



```
1   LIBRARY ieee;
2   USE ieee.std_logic_1164.all;
3   ENTITY RCA IS
4      PORT (
5         SW : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
6         LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
7         LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
8      );
9   END RCA;
10  ARCHITECTURE stru OF RCA IS
11     signal carry, A, B: std_logic_vector(3 downto 0);
12     component FA
13        port (a, b, c: in std_logic;
14              sum, carry: out std_logic);
15     end component;
16  BEGIN
17     A <= SW(4 downto 1);
18     B <= SW(8 downto 5);
19     FA0: FA port map (A(0), B(0), SW(0),    LEDG(0), carry(0));
20     FA1: FA port map (A(1), B(1), carry(0), LEDG(1), carry(1));
21     FA2: FA port map (A(2), B(2), carry(1), LEDG(2), carry(2));
22     FA3: FA port map (A(3), B(3), carry(2), LEDG(3), carry(3));
23     LEDR(0)  <= carry(3);                 -- carry out
24     LEDR(1)  <= carry(3) xor carry(2);    -- overflow
25  END stru;
26  -- FA component
27  LIBRARY ieee;
28  USE ieee.std_logic_1164.all;
29  ENTITY FA IS
30     PORT (a, b, c: in std_logic;
31           sum, carry: out std_logic);
32  END FA;
33  ARCHITECTURE FA_stru OF FCA IS
34  BEGIN
35     sum   <= (a xor b) xor c;
36     carry <= b when ((a xor b) = '0') else c;
37  END FA_stru;
```

Figure 10.10. Solution II - RCA implementation using component / port map.

The VHDL coding style adopted in Solution II results in a more organized circuit also when using the Netlist Viewer option available in Quartus II tool. In Figure 10.11, the four FA components used for the RCA implementation are explicitly shown in the Solution II netlist view. In order to see the FA logic gates implementation, the designer has to double-click in one of the FA components. On the other hand, in Solution I circuit view, the adder functionality understanding is not as straightforward as in Solution II.

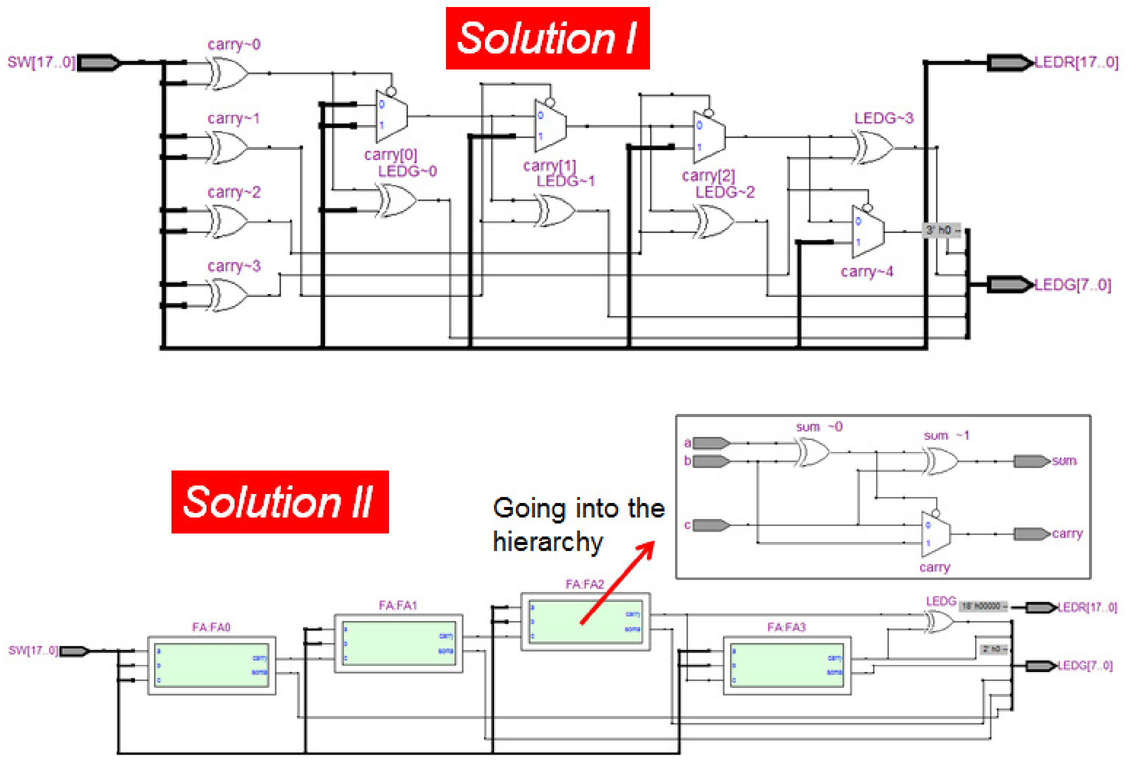
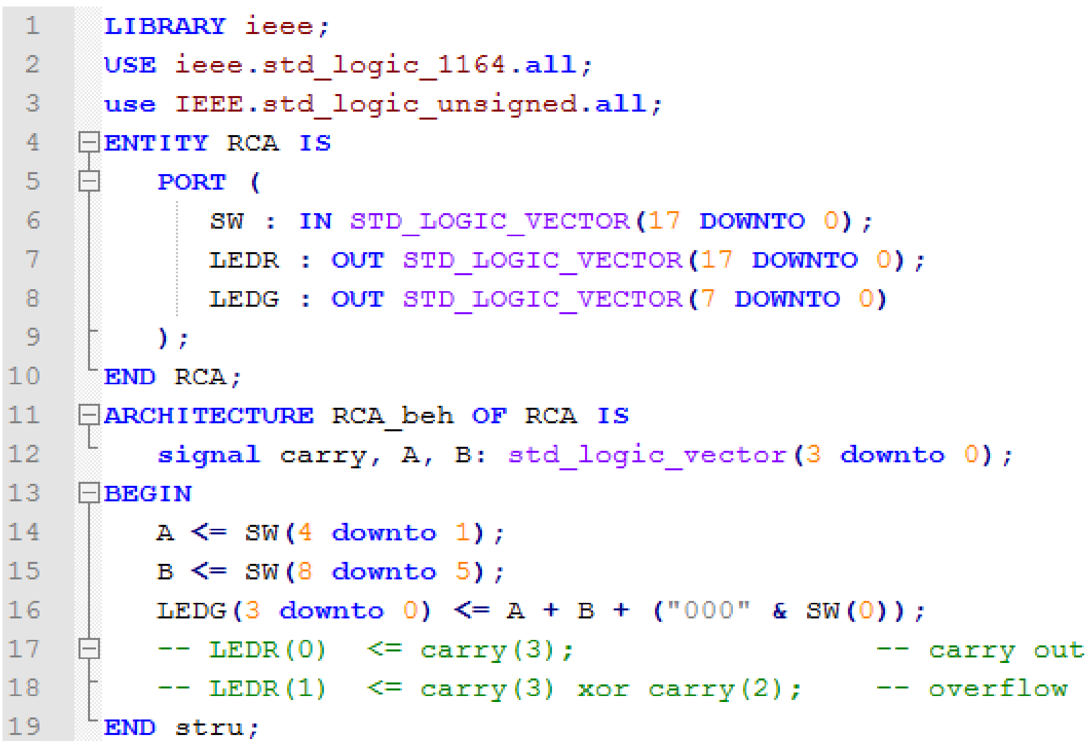


Figure 10.11. Quartus II netlist view for the RCA circuit.

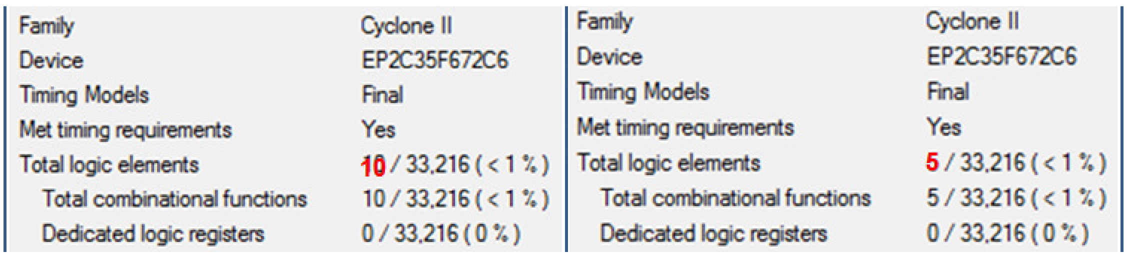
In previous chapters, the adder used in the calculator design was implemented using the VHDL '+' operator. This is a much easier way to implement an adder, but the drawback is the loss of circuit's observability and controllability. In Figure 10.12, the adder is written in a higher abstraction level, and the designer does not know which topology the synthesis tool will select for the implementation. For instance, there is no way to force the synthesis tool to implement the circuit using an RCA topology. In this example, with the lower observability degree, it is not possible to extract the overflow and carry out data.



```
LIBRARY ieee;
USE ieee.std_logic_1164.all;
use IEEE.std_logic_unsigned.all;
ENTITY RCA IS
    PORT (
        SW  : IN STD_LOGIC_VECTOR(17 DOWNTO 0);
        LEDR : OUT STD_LOGIC_VECTOR(17 DOWNTO 0);
        LEDG : OUT STD_LOGIC_VECTOR(7 DOWNTO 0)
    );
END RCA;
ARCHITECTURE RCA_beh OF RCA IS
    signal carry, A, B: std_logic_vector(3 downto 0);
BEGIN
    A <= SW(4 downto 1);
    B <= SW(8 downto 5);
    LEDG(3 downto 0) <= A + B + ("000" & SW(0));
    -- LEDR(0)  <= carry(3);                -- carry out
    -- LEDR(1)  <= carry(3) xor carry(2);   -- overflow
END stru;
```

Figure 10.12. Adder implementation using the VHDL ‘+’ operator.

In Figure 10.13 there is a comparison between the synthesis results for the RCA implementation listed in Figure 10.10 (left hand side), and the adder implementation using the ‘+’ operator listed in Figure 10.12 (right hand side). As expected, the ‘+’ operator alternative, resulted in less FPGA resources usage (5 logic elements), as the synthesis tool will select the best algorithm available for the adder implementation.



Family	Cyclone II	Family	Cyclone II
Device	EP2C35F672C6	Device	EP2C35F672C6
Timing Models	Final	Timing Models	Final
Met timing requirements	Yes	Met timing requirements	Yes
Total logic elements	10 / 33,216 (< 1 %)	Total logic elements	5 / 33,216 (< 1 %)
Total combinational functions	10 / 33,216 (< 1 %)	Total combinational functions	5 / 33,216 (< 1 %)
Dedicated logic registers	0 / 33,216 (0 %)	Dedicated logic registers	0 / 33,216 (0 %)

Figure 10.13. Quartus II synthesis summary for solution II and the ‘+’ operator alternative.

However, in applications where a higher degree of controllability/observability is required, the RCA implementations described before as Solution I and Solution II are a better alternative for an adder design than the ‘+’ VHDL operator.

The *carry out* and *overflow* signals observed in solutions I and II are also known as *flags*. A flag is a strategy widely used in digital systems to indicate the status or a condition happened in the last arithmetic or logic operation. In the discussed examples, two flags have been implemented. One of them is used to indicate the event of a carry out in the operation, and the other one to indicate that an overflow has happen in the last operation. In this case, the user (or another circuit) can use this information in order to decide whether or not to use the result provided by the adder.

10.2 Laboratory Assignment

The laboratory objectives are:

- to understand the design of a structural RCA component in VHDL;
- to replace the adder used in the calculator by the new component.

Laboratory Session

The tasks to be performed in this laboratory session are as follows:

- Using the VHDL implementation of Solution II listed in Figure 10.10, make the necessary modifications in order to have an 8-bits RCA component.
- The new component should have two 8-bits inputs, a 1-bit *carry in* input, an 8-bits sum output, and four 1-bit flags output.
- The flags should indicate the following events happened in the last operation performed by the adder: a *carry out*, an *overflow*, a *negative result*, and a *zero result*.
- Assume that the calculator makes operations in two's complement signed numbers.
- The *negative* flag can be obtained straight from the result (most significant bit).
- The *zero* flag should be on when all bits of the result are equal to zero.
- The four flags can be connected to the green LEDs, as some of the red LEDs are already in use by the calculator.
- It could be interesting to perform a timing simulation, instead of the functional simulation, in order to observe the delay needed by the RCA component to propagate the internal carry signals
- Replace the C1 component used in previous chapters, by the new RCA component.
- The new design should be carefully designed, in order to avoid major changes in the remaining calculator components.
- Perform a final simulation in the design as a whole, this time with the new component added to the calculator.

Figure 10.14 shows the calculator design with the new RCA component, replacing the C1 component used in previous chapters. The new adder has an additional 4-bits output connected to the green LEDs (flags), and also a 1-bit input that is not connected (*carry in* signal).

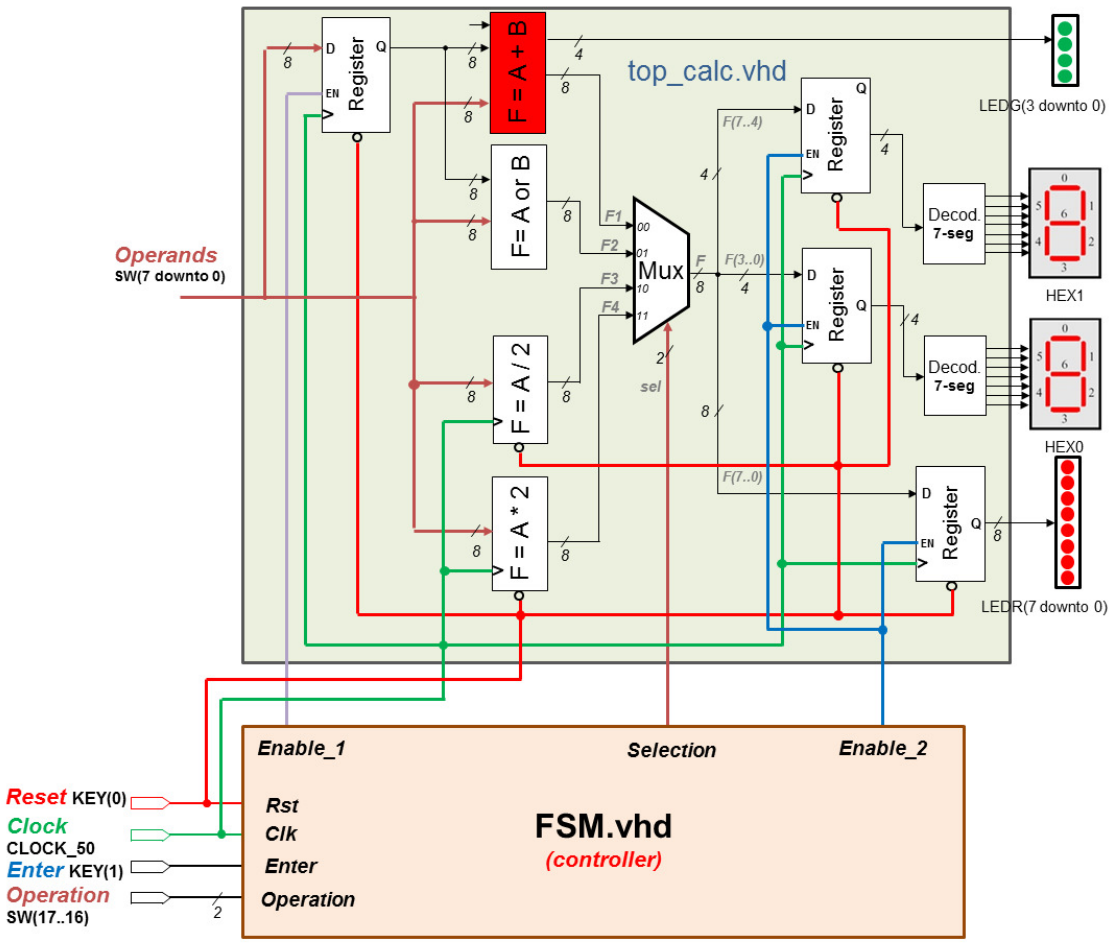


Figure 10.14. The final calculator design, with the new RCA component.