



# Politecnico di Torino

## DIGITAL SYSTEMS ELECTRONICS

040IHNX - A.A. 2024/2025

Prof. G. Masera

### Laboratory assignment no. 3 – Data-path elements

DUE DATE: 01/04/2025

DELIVERY DATE: 31/03/2025

GROUP 08 - Contributions:

- Daniele Becchero (308299) – 33.3%
- Bohotici Ionut Viorel (300061) – 33.3%
- Simone Viola (310779) – 33.3%

The members of the group listed above declare under their own responsibility that no part of this document has been copied from other documents and that the associated code is original and have been developed expressly for the assigned project.

## Introduction

Decimal	Signed
+7	0111
+6	0110
+5	0101
+4	0100
+3	0011
+2	0010
+1	0001
0	0000
-1	1111
-2	1110
-3	1101
-4	1100
-5	1011
-6	1010
-7	1001
-8	1000

This laboratory experience is about the design of some elements that perform the fundamental operations of sum, subtraction and multiplication of binary numbers. All these components are usually implemented in a data-path to process an input stream.

Numbers can be positive or negative, and representing them correctly is crucial. While unsigned arithmetic employs a straightforward binary format for non-negative values, signed arithmetic uses the two's complement format. This method streamlines addition and subtraction by enabling the same hardware to process both positive and negative numbers without separate circuits. It also standardizes the representation of zero, eliminating the ambiguity of having both a positive and a negative zero. Additionally, the two's complement system facilitates easy detection of overflow, which is essential for maintaining accurate calculations in computer systems. The 4-bit two's complement conversion table on the left illustrates this representation clearly.

The formula to convert a N-bit signed number into its decimal representation is the following one

$$X_{DEC} = -b_{N-1} \cdot 2^{N-1} + \sum_{i=0}^{N-2} b_i \cdot 2^i$$

where  $b_i$  is the  $i$ -th bit of the 2's complement number.

## Section 1: 4-bit sequential RCA

This section is about a signed 4-bit adder based on a 4-bit ripple-carry adder. The circuit scheme is shown below. It includes also three 4-bit registers, and a circuit which detects the overflow and store its flag in a flip-flop.

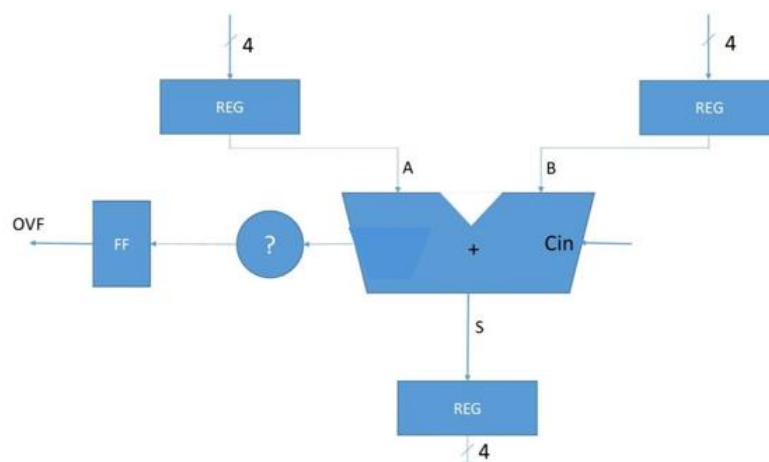


Figure 1 – 4-bit ripple-carry adder circuit structure

## Delivered files

Eight VHDL files have been delivered for the description of the signed adder and its simulation:

- *signedAdder.vhd*, VHDL code for the top-level entity, which acts as the signed adder
- *signedAdder\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder
- *rippleCarryAdder.vhd*, VHDL code for the 4-bit ripple carry adder, built using four full adders
- *regn.vhd*, which is the VHDL code for the register
- *ovfdetector.vhd*, VHDL code for the overflow detection circuit, which evaluates whether an overflow has occurred
- *flipflop.vhd*, VHDL code for a D-type flip-flop
- *fullAdder.vhd*, VHDL code for the full adder
- *mux.vhd*, VHDL code for the 2-to-1 1-bit multiplexer

These files have been used on ModelSim in order to perform the static simulation of the adder.

Two additional VHDL files were provided for testing on the FPGA. The first file, *device\_signedAdder.vhd*, serves as the new top-level entity. This file includes the *signedAdder* component modelled and tested in ModelSim, the 7-segment display decoder, and the connections between the DE1-SoC onboard devices and the *signedAdder* component that will be implemented in the FPGA.

We designed it this way to focus solely on simulating the adder component, excluding other elements, such as the decoder and displays, that might introduce additional errors during simulation. By isolating the adder in simulation, we ensure accurate testing of its functionality. After validating the adder through simulation, we incorporate it as a component in a VHDL-described circuit for practical implementation on the FPGA. With this workflow, we assume that, except for the *signedAdder* component, every other component used in the circuit, such as the decoder, is logically correct. This assumption is valid because we designed and tested them in previous laboratory experiences. **The same workflow has been used for all the next section in this report.**

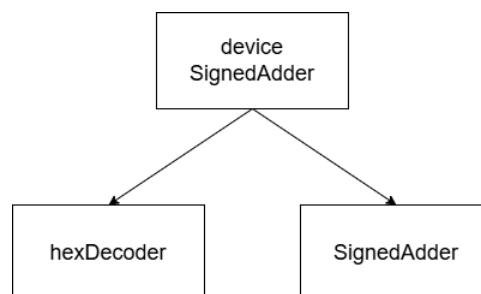


Figure 2 - VHDL model structure for 4-bit signed adder

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *Lab3\_1.sof*, and it already includes the assignment file *DE1-SoC.qsf*. The .sdc file, or Synopsys Design Constraints file, is used to define the timing constraints for the digital design.

## Design entry

### Signed adder (top-level entity)

The top-level entity is the signed adder, as described above. It has two 4-bit wide input ports named IN1 and IN2, that are the two addendums. Another input, IN3, defines is the input carry of the adder. There are two outputs. OUT1 is the operation output as a 4-bit binary value, while OUT2 is the overflow flag. Moreover, there are the CLK and the RESN inputs that provides the clock and the reset signal for the circuit. Notice that the reset signal is active-low and asynchronous. The signed adder architecture is made by the components named *regn*, *flipflop*, *rippleCarryAdder* and *ovfdetector*, connected according to Figure 1.

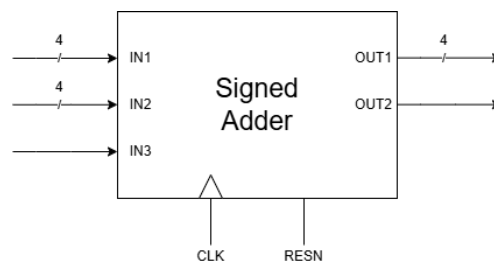


Figure 3 – 4-bit signed adder block representation

### Register and flipflop

The *regn* and *flipflop* VHDL files are the ones provided by the professor in the assignment.

### 4-bit ripple-carry adder

The *rippleCarryAdder.vhd* file defines a fully-combinational component that implements a 4-bit ripple carry adder. This component features two 4-bit wide signed input ports (representing the numbers to be added) and a 4-bit wide signed output port (representing the sum result). Additionally, it includes a 1-bit carry-in input and a 1-bit carry-out output to handle overflow or propagate carry bits.

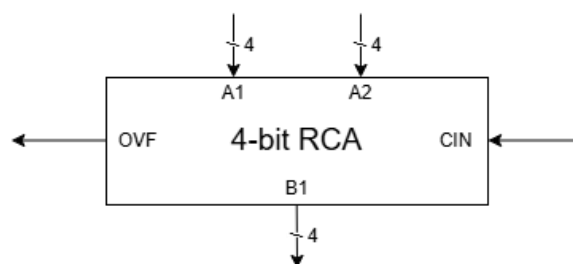


Figure 4 - 4-bit ripple-carry adder block representation

Notice that this entity has two 4-bit wide *signed* input ports, but then every bit is connected to the input ports of the corresponding full adder. The ripple-carry adder works correctly if the input ports were declared as standard-logic vector type, since a generic RCA perform bitwise addition independently from the input vector format. For newest designs it could be a better choice to change the *signed* ports of the RCA into *std\_logic\_vector* ports, in order to achieve a better interoperability of the entity with both signed and unsigned inputs, leaving to the designer the handling of the two cases signed and unsigned.

The architecture of the RCA is defined using four interconnected full adders, following the configuration specified in the assignment. Each full adder processes a single bit of the input numbers, with the carry-out of one adder feeding into the carry-in of the next, creating the "ripple" effect.

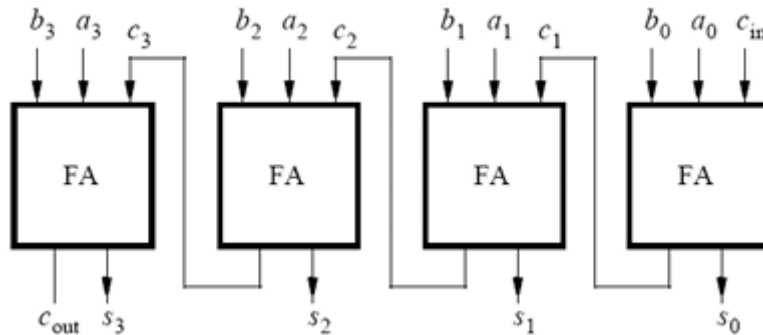


Figure 5 – 4-bit ripple-carry adder internal structure

### Full adder

The full-adder is the elementary component of the ripple-carry adder. Since the RCA is 4-bit wide, there are four full adders connected as shown above. The functionality of a single full adder can be implemented using a combinational logic circuit, as depicted in the corresponding diagram.

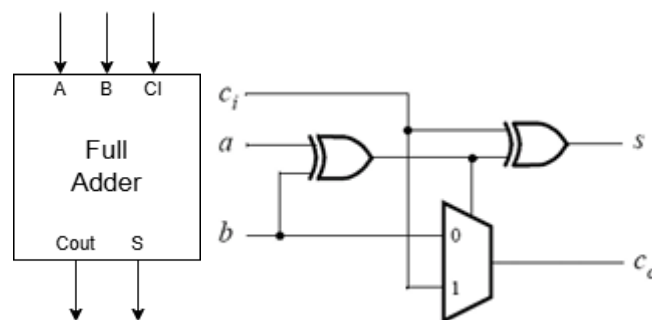


Figure 6 - Full-adder block representation and internal structure

Hence, the full adder has three input ports, named A, B, and C\_IN, and two output ports, named S and C\_OUT. Its architecture includes the multiplexer as a component, and two exclusive-or gates.

### Overflow detector circuit

The overflow detector circuit, labelled as the "?" component in Figure 1, is designed to handle overflow cases in signed arithmetic. For example, consider the addition of 0111 (+7 in decimal) and 0001 (+1). The expected result is 1000 (-8 in decimal), with a carry-out of 0. However, this result is incorrect due to an overflow condition, which occurs when the sum exceeds the representable range for 4-bit signed numbers. The overflow condition must be accurately detected and appropriately signalled by the circuit.

This operation is performed by analysing the conditions that lead to overflow. For instance, overflow occurs when adding two positive numbers produces a negative result, or when adding

two negative numbers produces a positive result. These scenarios indicate that the sum has exceeded the representable range for signed numbers, triggering the need for overflow detection. In digital logic, this is equivalent to first check if the MSB of A and B is equal (meaning same sign), by using an XNOR gate and then check also if the MSB of A (or B) and S are different (meaning the sign of A is different from the sign of S), by using of an XOR gate. This is described in VHDL as shown.

```
OVF <= (A xnor B) and (A xor S);
```

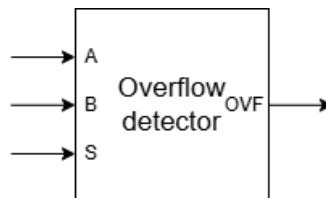


Figure 7 – Overflow detector circuit block representation

### Functional simulation

The testbench for the signed adder, named *signedAdder\_tb.vhd* is based on a process where two input values are changed every two clock cycles, which is also generated inside the process. The input values are updated every two clock cycles because the sum result is updated every two cycles, and so the interpretation of the simulation results is easier.

The waveforms result of the simulation is shown in Figure 8. It is also observed that, on the right side of the waveform, the RESN signal is driven low. As a result, the output is immediately forced to '0000', bypassing the need to wait for the next active rising edge of the clock signal. Note that the testbench stimuli have been represented in decimal form by selecting 'Decimal' in the Radix menu.

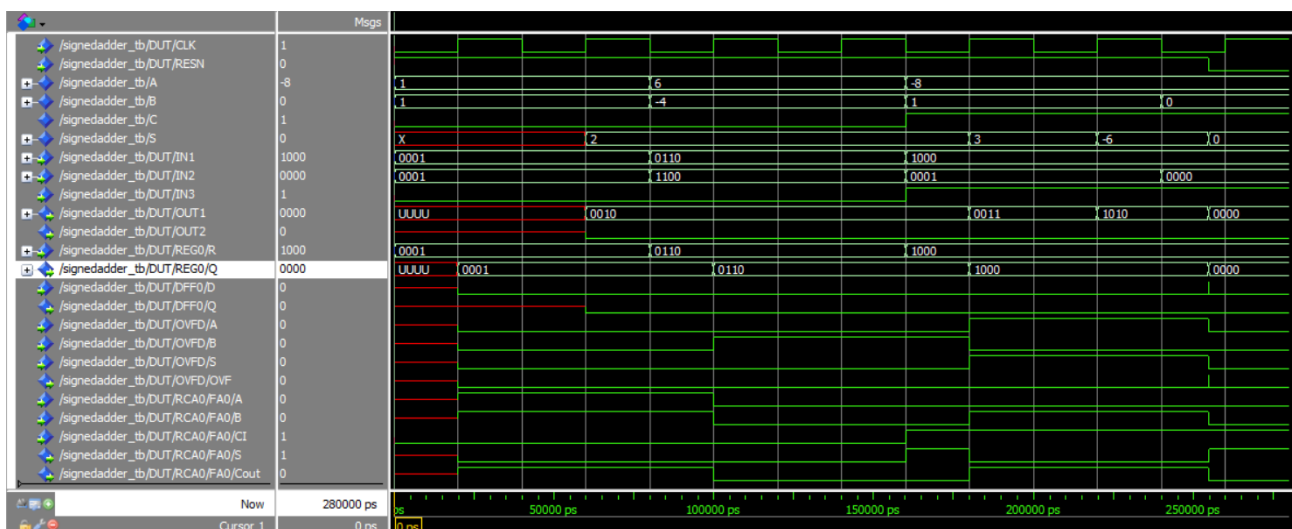


Figure 8 – 4-bit signed adder simulation results in ModelSim

Starting from the top, in the waveforms are represented the clock signal and the active-low reset. Then, the stimuli and the signed adder's signals have been represented either in logic or decimal form by selecting the correct format in the Radix menu of the signal. Below, there are



the I/O signals of a register the flipflop and the overflow detector. On the bottom, there are I/O and internal signals of the first full-adder. It is important to notice that the circuit is fully synchronous, apart from the RESN input, which is asynchronous.

### Synthesis

The VHDL code for *device\_signedAdder* is ready to be opened in a new Quartus Prime project and used to program the FPGA. First, we need to import all the VHDL codes of the components and the assignment file *DE1-SoC.qsf*. Remind that in this step the *hexDecoder* component needs to be included in the project. This file is a modification of the previous 7-segment decoder in order to show correctly a 4-bit signed number in two 7-segment displays.

Once the VHDL files have been imported, we start the compilation of the entire project. Once the compilation is complete, we check for errors or warnings. As there are no critical errors or warnings, we connect the board to the PC, launch the auto-detect tool, and upload the .sof file to the FPGA.

At this point, the FPGA on the DE1-SoC board is correctly programmed, and we are ready to verify its functionality. The first four switches (*SW0* to *SW3*) control the input A of the adder, while the second four switches (*SW4* to *SW7*) control the input B. It is important to remind that the device is handling signed numbers, so the input combination must be expressed in 2's complement format. The first pushbutton *KEY0* is used to provide the asynchronous reset signal, which is active-low, while the second pushbutton *KEY1* is providing a “manual” clock signal. The two addendums and the operation result are shown in the 7-segment displays, arranged from left to right. Every number is displayed (in decimal) on two displays: one shows the modulus, and the other indicates the minus sign, if needed. The input A is shown in *HEX5* and *HEX4* displays, the input B is shown in *HEX3* and *HEX2* displays, and the result is shown on the remaining ones. The overflow condition is signalled by *LED9*.

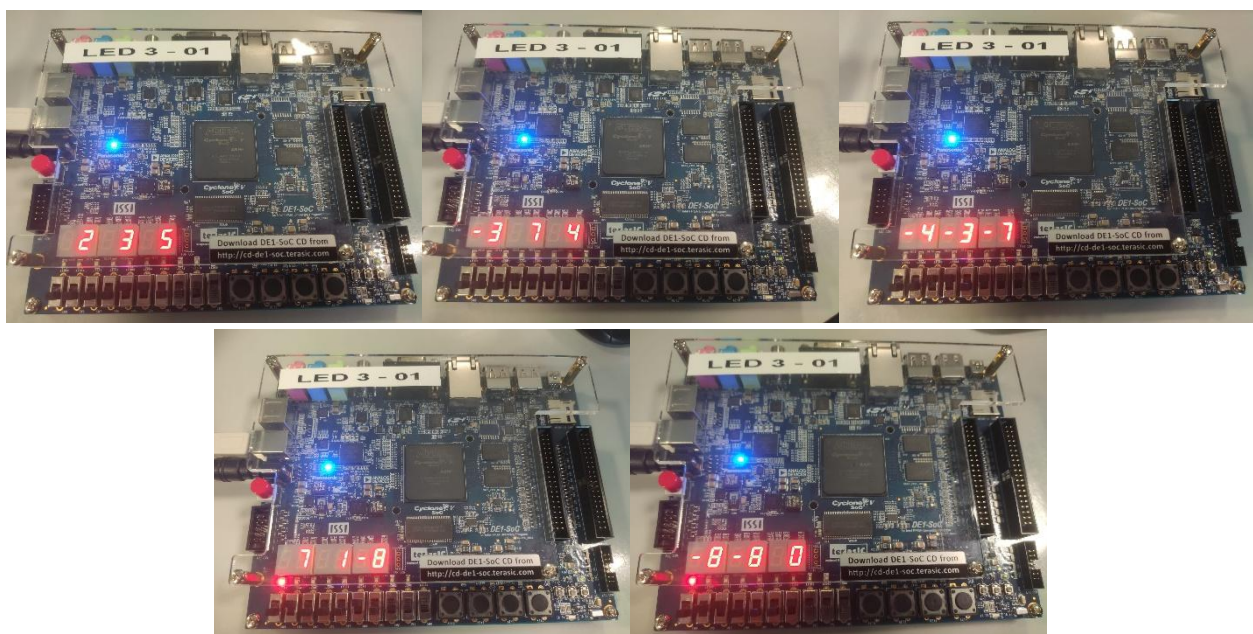



Figure 9 – Practical test of 4-bit signed adder with the DE1-SoC board

### Timing analysis

Quartus is able to perform a timing analysis after the place & route phase in order to evaluate the worst-case delay path that will limit the speed of the circuit. This analysis needs to include a so called “timing constraint” in the project. This file is the one that have the extension .sdc, which stands for Synopsys design constraint. In this file there is the definition of a clock signal with a name that corresponds to the one used in the VHDL model. Once that file has been included, it is possible to start the timing analysis by re-compiling the entire project. At the end, in the timing analysis report is possible to check the results. The most interesting parameter is the maximum clock frequency that is allowed without having errors due to delays. The analysis returns two values, specified at 0 °C and 85 °C. The lowest one is approximately 466 MHz at 0 °C.

Slow 1100mV 0C Model Fmax Summary			
 <<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	466.2 MHz	466.2 MHz	CLK


Slow 1100mV 85C Model Fmax Summary			
 <<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	469.04 MHz	469.04 MHz	CLK

Figure 10 – Timing analysis results for 4-bit signed adder

In this circuit, the critical path is defined by the signal propagation that starts at the input of full-adder number 0 and extends to the output carry of the ripple-carry adder (RCA). This path represents the longest delay, thus setting the maximum speed at which the circuit can operate.



## Section 2: 4-bit sequential adder/subtractor

This section extends the functionality of the signed adder described in *Section 1* by also handling the subtraction of 4-bit signed numbers. The circuit structure is exactly the same as the one described in *Section 1*. The only difference is in the full-adder entity, which has been edited to handle either addition or subtraction.

### Delivered files

Eight VHDL files have been delivered for the description of the signed adder/subtractor and its simulation:

- *signedAdderSubtractor.vhd*, VHDL code for the top-level entity, which acts as the signed adder/subtractor
- *signedAdderSubtractor\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder/subtractor
- *rippleCarryAdderSubtractor.vhd*, VHDL code for the 4-bit ripple carry adder/subtractor, built using four full adders
- *regn.vhd*, which is the VHDL code for the register
- *ovfdetector.vhd*, VHDL code for the overflow detection circuit, which evaluates whether an overflow has occurred
- *flipflop.vhd*, VHDL code for a D-type flip-flop
- *fullAdder.vhd*, VHDL code for the full-adder
- *mux.vhd*, VHDL code for the 2-to-1 1-bit multiplexer

These files have been used on ModelSim in order to perform the static simulation of the adder.

Two additional VHDL files were provided for testing on the FPGA. The first file, *device\_signedAdderSubtractor.vhd*, serves as the new top-level entity. This file integrates the *signedAdderSubtractor* component modelled and tested in ModelSim, the 7-segment display decoder, and the connections between the DE1-SoC onboard devices and the *signedAdderSubtractor* component implemented in the FPGA, following the same workflow described in *Section 1*.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *Lab3\_2.sof*, and it already includes the assignation file *DE1-SoC.qsf*. The .sdc file, or Synopsys Design Constraints file, is used to define the timing constraints for the digital design.

### Design entry

#### Signed adder/subtractor (top-level entity)

The top-level entity is the signed adder/subtractor, as described above. The component is an edited version of the signed adder component described in *Section 1*. The main difference is in the IN3 input, which this time is defining if the RCA is working as an adder or as a subtractor.

The signed adder/subtractor architecture is made by the components named *regn*, *flipflop*, *rippleCarryAdderSubtractor* and *ovfdetector*, connected according to Figure 1.

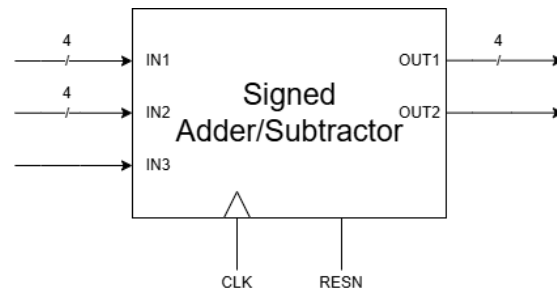


Figure 11 – 4-bit signed adder/subtractor block representation

### Register and flipflop

The *regn* and *flipflop* VHDL files are the ones provided by the professor in the assignment.

### 4-bit ripple-carry adder/subtractor

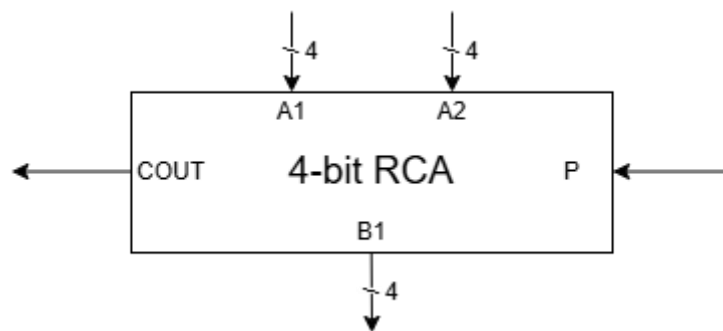


Figure 12 – 4-bit ripple-carry adder/subtractor block representation

The *rippleCarryAdderSubtractor.vhd* is an edited version of the 4-bit ripple-carry adder described in the previous section. As shown in lectures, the modification that must be done to the RCA to perform both addition and subtraction (one at a time) are first the calculation of the 2's complement of the input A2 and then adding one as input-carry, only when the device is working as a subtractor. The logic operation of subtraction becomes the following one:

$$A1 - A2 = A1 + (-A2) = A1 + (\overline{A2} + 1) = (A + \overline{A2}) + 1$$

The mode selection of the device is selected by the P input ('0' will perform addition, while '1' will perform subtraction). The logic behind the addition operation has not been modified.

The architecture of the RCA is defined using four interconnected full-adders, with no substantial modification with respect of the previous case. The main modification is that the P input (which selects the operation to be done) is provided to each full-adder and also as input-carry of the first full-adder. It is very important to notice that, apart from the logic operation  $\text{XOR}(B, P)$ , the internal structure of the full adder component has not changed.

### Full adder

The full-adder component has undergone one significant modification. The structure is very similar to the one of the full-adder described in the previous section, but the B input of the component is first inverted if a subtraction is done. This is done by computing the XOR operation between B and P **before** providing that signal to the B input of the “old” full-adder combinational circuit. Apart from that change, the rest of the combinational circuit that implements the full-adder, including the multiplexer, has not changed.

Hence, the full-adder has four input ports, named A, B, CI and P, and two output ports, named S and C\_OUT. The VHDL description of the full-adder’s architecture is shown below.

```
... - mux component instantiation
signal B2, SEL : STD_LOGIC;

begin
    B2 <= B xor P;

    MUX0 : mux
    port map(B2, CIN, SEL, Cout);

    SEL <= A xor B2;
    S <= CIN xor SEL;
-- End of architecture
```

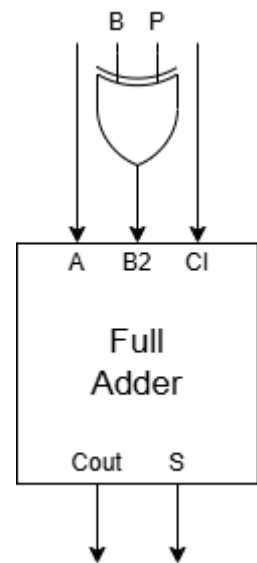


Figure 13 – Edited full-adder block representation

### Overflow detector circuit

The overflow detector circuit differs slightly from the previous one because it must handle two operations. Besides to the addition case, the subtraction case must also be considered. Overflow during subtraction occurs when a positive number is subtracted from a negative number, resulting in a positive result, or when a negative number is subtracted from a positive number, resulting in a negative result. The entity includes an additional input, labelled as MODE, which define if an addition or a subtraction is performed, in order to evaluate correctly the overflow case depending on the case. In VHDL, this component can be described as shown.

```
OVF <= (A AND B AND NOT S) OR (NOT A AND NOT B AND S) when
MODE = '0' else
    (A AND NOT B AND NOT S) OR (NOT A AND B AND S);
```

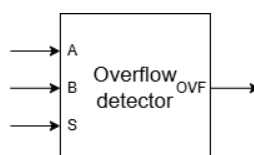


Figure 14 - Overflow detection circuit block representation

## Functional simulation

The testbench for the signed adder/subtractor is very similar to the one for the signed adder. It only includes also the input “C” that is defining what operation must be done. C = ‘0’ will perform an addition between A and B, while C = ‘1’ will perform the subtraction A-B. Again, the result is computed with a delay of two clock cycles with respect to the applied inputs.

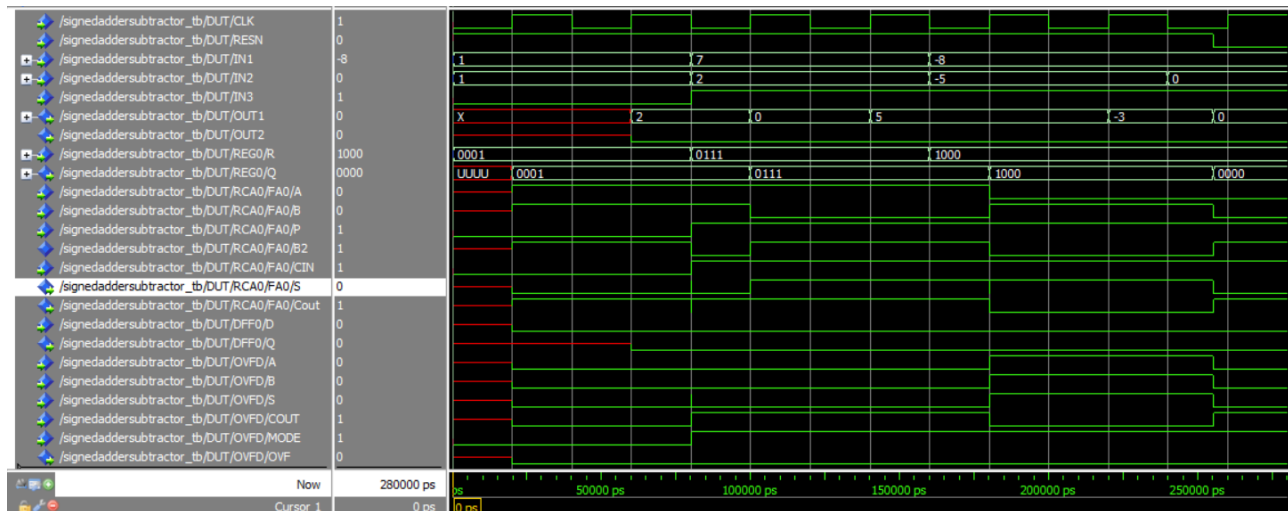


Figure 15 - 4-bit signed adder/subtractor simulation results in ModelSim

Starting from the top, in the waveforms are represented the clock signal and the active-low reset. Then, the signed adder’s signals have been represented either in logic or decimal form by selecting the correct format in the Radix menu of the signal. Below, there are the I/O signals of a register, the I/O and internal signals of the first full-adder. On the bottom, there are the signals of the flip-flop and the overflow detector. It is important to notice that the circuit is fully synchronous, apart from the RESN input, which is asynchronous.

## Synthesis

The steps to test the functionality of the device on the DE1-SoC board are the same that we have done for the previous exercises.

As in the previous section, the two input ports of the adder/subtractor are connected respectively to the switches 0 to 3 and 4 to 7. An additional switch, SW8, is used to define what operation must be done. When SW8 is ‘OFF’, the device works as an adder. In the other case, the circuit works as a subtractor. The overflow condition is shown by using the LED9. The pushbuttons KEY0 and KEY1 are used as the reset button and “manual” clock. The two addendums and the operation result are shown in the 7-segment displays, arranged from left to right to enhance the readability.

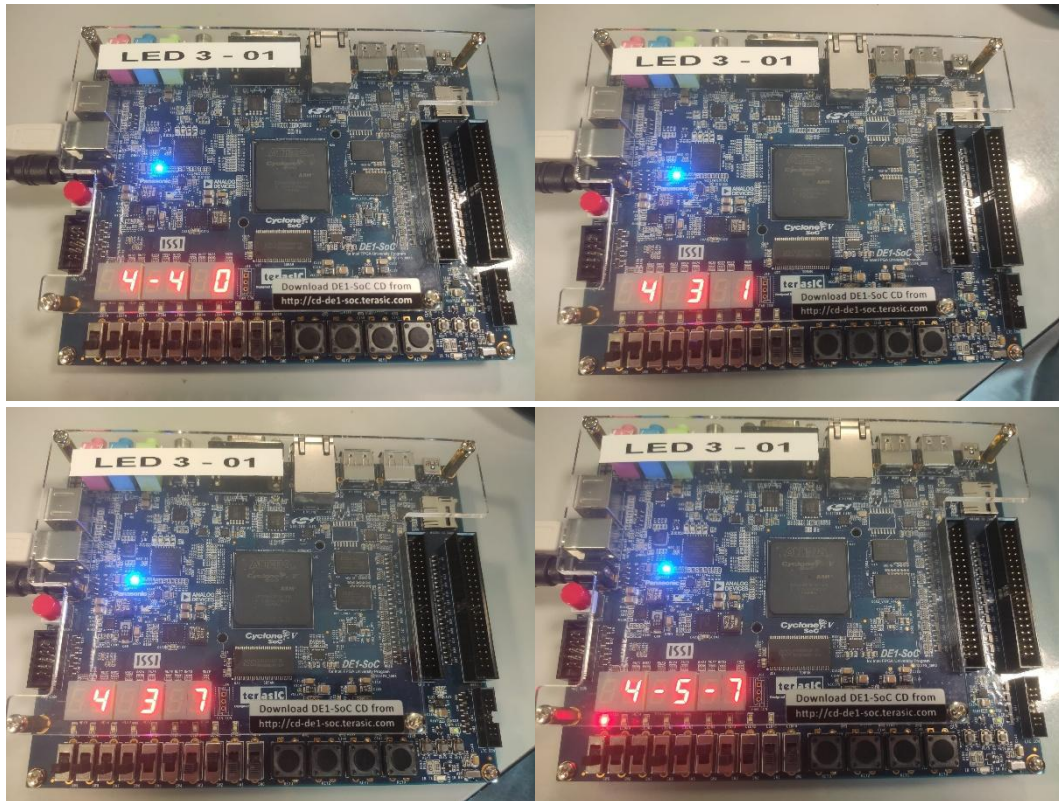


Figure 16 – Practical test of 4-bit signed adder/subtractor with the DE1-SoC board

### Timing analysis

For this model, the timing analysis returns two maximum frequencies that are slightly lower than the ones for the previous circuit. This is due to an additional, relatively minor delay due to the bit inversion required before addition. The critical path for delay is again the one starting from any input of the first full-adder and ends at the output-carry.

Slow 1100mV OC Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	418.59 MHz	418.59 MHz	CLK
Slow 1100mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	426.62 MHz	426.62 MHz	CLK

Figure 17 – Timing analysis results for 4-bit signed adder/subtractor

## Section 3: 16-bit RCA, Carry-Bypass Adder and Carry-Select Adder

This section is about the design of a 16-bit adder based on three different architectures and their comparison. The first one is the ripple-carry adder, which works as the circuit in *Section 1*. Then, we are asked to design a carry-bypass adder. Its internal architecture is different from the RCA, but thanks of the modular approach of VHDL, we have just to re-design the adder block, without modifying all the rest of the circuit and the blocks' interconnections. In the third subsection, we are asked to design a carry-select adder, and we have proceeded with the same workflow.

### Section 3.1: 16-bit RCA

This section is about the design of a 16-bit ripple-carry adder. The circuit is basically an extension of the circuit described in *Section 1*.

#### Delivered files

Eight VHDL files have been delivered for the description of the component and its simulation:

- *signedAdder16bit.vhd*, VHDL code for the top-level entity, which acts as the 16-bit signed adder
- *signedAdder16bit\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder
- *rippleCarryAdder16bit.vhd*, VHDL code for the 16-bit ripple carry adder, built using 16 full-adders
- *regn.vhd*, which is the VHDL code for the register
- *ovfdetector.vhd*, VHDL code for the overflow detection circuit, which evaluates whether an overflow has occurred
- *flipflop.vhd*, VHDL code for a D-type flip-flop
- *fullAdder.vhd*, VHDL code for the full-adder
- *mux.vhd*, VHDL code for the 2-to-1 1-bit multiplexer

In addition to the mentioned files, also the .sdc file with timing constraints have been delivered.

#### Design entry

The entities *signedAdder16bits*, *flipflop*, *regn* and the *overflowdetector* are exactly the same of the ones described in *Section 1*, and they are arranged in the same way. The only modification is in the instantiation of the “adder” component and in the dimensions of the signals of the components, because this time the inputs IN1 and IN2 and the output OUT1 are 16-bit wide instead of 4.

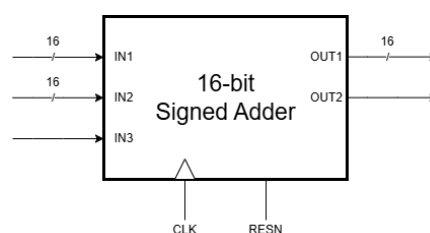


Figure 18 – 16-bit signed adder block representation



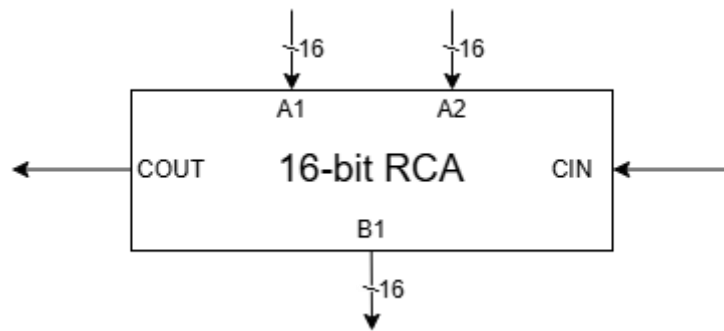
*16-bit ripple carry adder*

Figure 19 – 16-bit ripple-carry adder block representation

The 16-bit ripple-carry adder has the same style as the 4-bit one, except for the number of full-adders that have been allocated. This time 16 bits have to be handled, so 16 full-adders are needed, so we decided to generate them with the `for... generate` statement to speed up the writing of VHDL code.

```
-- Architecture body
... - mux component instantiation

-- 15 intermediate carry plus the carry_out
signal C : STD_LOGIC_VECTOR((15 + 1) downto 0);

begin
    C(0) <= CIN;
    COUT <= C(16);

    -- Generate multiple instances of the full adder
    gen_block : for i in 0 to 15 generate
        FA : fullAdder
        port map(
            A    => A1(i),
            B    => A2(i),
            CI   => C(i),
            S    => B1(i),
            Cout => C(i + 1)
        );
    end generate gen_block;
```

The full adder and the multiplexer entities are the same used in previous sections.

**Functional simulation**

The testbench for this circuit is straightforward, and the process statement is very similar to the previous ones. The key difference is that integer numbers are used in the testbench and then applied to the corresponding ports of the 16-bit adder. Before connecting the integer signals to the ports of the component, they must undergo a cast operation from integer to signed to ensure compatibility with the component described in the design entry. This

approach allows us to quickly verify the functionality of the component, as working with decimal integers is far more intuitive for us compared to performing additions with two 16-bit binary numbers, which can be a more challenging task to do mentally.

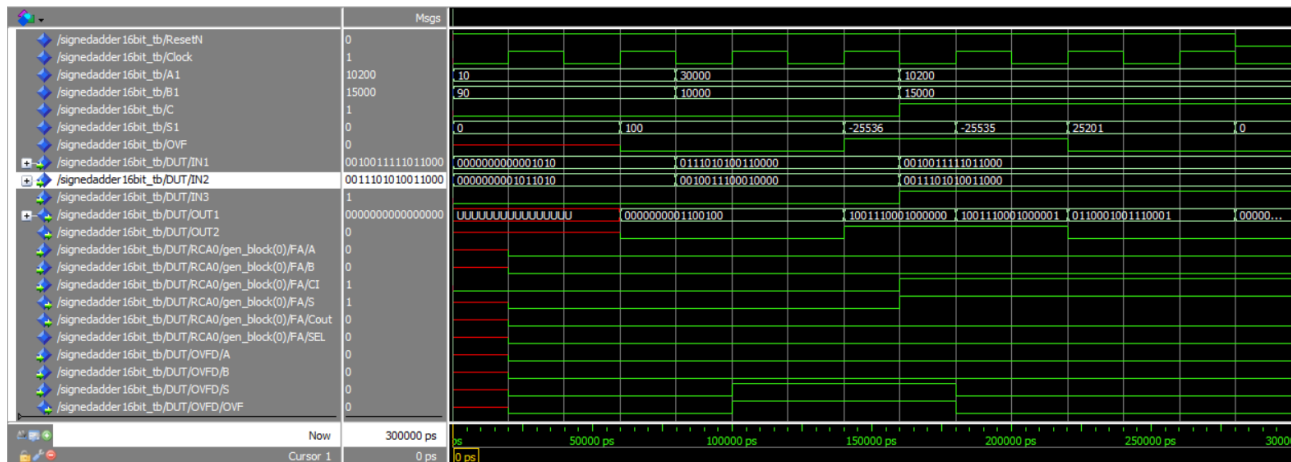


Figure 20 – 16-bit signed adder simulation results in ModelSim

Starting from the top, in the waveforms are represented the active-low reset signal and the clock. Then, the testbench stimuli signals have been represented in decimal form, as described above. The DUT signals have been represented either in logical form or as 16-bit signed values. Below, there are the I/O and internal signals of the first full-adder. On the bottom, there are the signals of the overflow detector. It is important to notice that the circuit is fully synchronous, apart from the RESN input, which is asynchronous.

### Synthesis and timing analysis

For this model, practical implementation on the DE1-SoC board cannot be tested due to the insufficient number of available switches required to generate two 16-bit inputs, but the project is still synthesizable. In the compilation process, Quartus will perform all synthesis steps but will not assign any signals to the I/O ports of the top-level entity. Hence, the *signedAdder16bit* is fit in the FPGA, but it is basically useless. However, the synthesis process is useful to perform the timing analysis of the circuit. After including the timing constraints file, the tool returns two maximum frequencies that are significantly lower than the ones for the *Section 1*. This is due to the increased number of full-adders required to handle 16-bit operations. The critical path for delay is the one starting from any input of the first full-adder and ends at the output-carry.

Slow 1100mV 0C Model Fmax Summary				Slow 1100mV 85C Model Fmax Summary			
<<Filter>>				<<Filter>>			
	Fmax	Restricted Fmax	Clock Name		Fmax	Restricted Fmax	Clock Name
1	196.93 MHz	196.93 MHz	CLK	1	189.39 MHz	189.39 MHz	CLK

Figure 21 – Timing analysis results of 16-bit signed adder based on RCA internal architecture

### Section 3.2: 16-bit carry-bypass adder

In this section, the ripple-carry adder have been changed with a carry-bypass adder, also known as carry-skip adder (CSKA). Since the RCA is a just component of the 16-bit signed adder, this project can be derived from the previous one.

#### Delivered files

Ten VHDL files have been delivered for the description of the component and its simulation:

- *signedAdder16bit.vhd*, VHDL code for the top-level entity, which acts as the 16-bit signed adder
- *signedAdder16bit\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder
- *carryBypassAdder\_16bit.vhd*, VHDL code for the 16-bit carry-bypass adder, built using four 4-bit RCA, four 2-to1 multiplexers and four “Carry propagation” components
- *rippleCarryAdder4bit.vhd*, VHDL code for the 4-bit ripple-carry adder, built using four full adders
- *carryPropagation.vhd*, VHDL code for the entity that will compute if the carry can be provided to the next RCA without waiting for its evaluation
- *regn.vhd*, VHDL code for the register
- *ovfdetector.vhd*, VHDL code for the overflow detection circuit, which evaluates whether an overflow has occurred
- *flipflop.vhd*, VHDL code for a D-type flip-flop
- *fullAdder.vhd*, VHDL code for the full adder
- *mux.vhd*, VHDL code for the 2-to-1 1-bit multiplexer

In addition to the mentioned files, also the .sdc file with timing constraints have been delivered.

#### Design entry

The entities *signedAdder16bit*, *flipflop*, *regn* and the *overflowdetector* are exactly the same of the ones described in the previous case, and they are arranged in the same way. The carry-skip adder is declared as a component used in the *signedAdder16bit* entity (which is the top-level entity of the entire project).

#### 16-bit carry-bypass adder

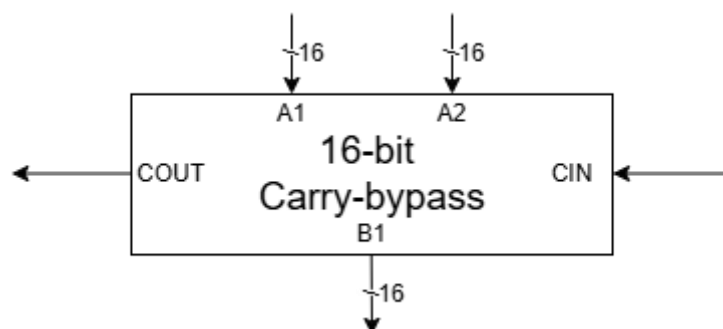


Figure 22 – 16-bit carry-bypass adder block representation

The carry-bypass adder is a circuit composed of several RCAs and multiplexers. In a 16-bit RCA, the main issue lies in the critical path delay, as the output-carry is determined by signals propagating all the way from the first full-adder to the last. To address this, the carry-skip adder is designed to perform this operation more efficiently. It consists of four blocks, each made up of a 4-bit RCA and a multiplexer. While each RCA receives the input-carry and computes the sum of its block's bits (a familiar operation apart from the reduced RCA size), the key innovation lies in handling the output-carry. Each block assesses whether the carry can propagate uninterrupted across all its bits using a propagation condition. If the condition holds true, the output-carry of the RCA equals the input-carry. Otherwise, the output-carry must be calculated by the RCA itself. A multiplexer selects the appropriate path for the output-carry, which is then passed to the next RCA block.

The 16-bit CSKA entity has the same I/O ports as the 16-bit RCA, but its structure has changed. It is made by four blocks, as described above, so in VHDL it has been described with a `for...generate` statement. Each block is made by a 4-bit RCA, a 2-to-1 multiplexer, and a carry propagation selection circuit, connected as shown in pictures.

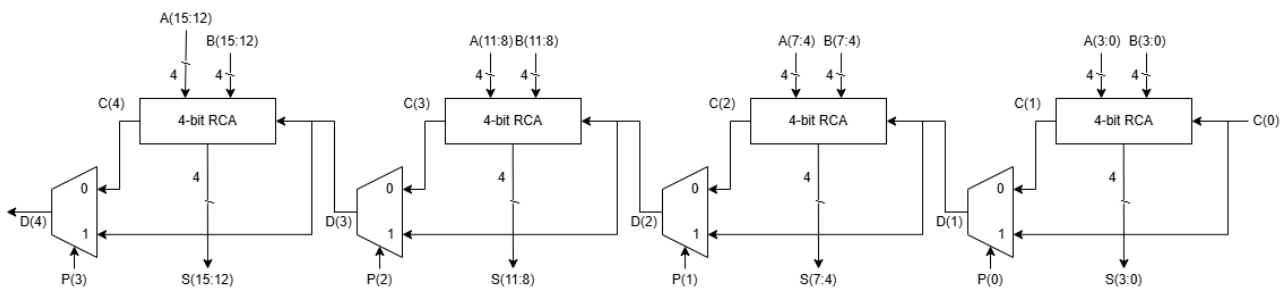


Figure 23 – 16-bit carry-bypass adder internal architecture

The 4-bit RCA and the multiplexer are the same components used in *Section 1*.

### Carry propagation circuit

The carry propagates without delay through the RCA when the inputs, A and B, of each full-adder within the RCA are different. In example, for the first RCA the propagation condition is equivalent to the logical operation

$$P = \prod_{i=0}^{4-1} p_i, \quad p_i = a_i \oplus b_i$$

Hence, this entity has two 4-bit wide input ports and one 1-bit output port. The VHDL description of the component comes from the logical operation above.

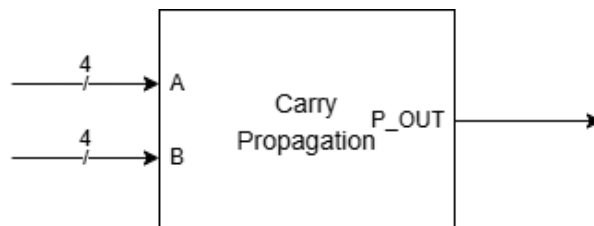


Figure 24 – Carry propagation detector block representation

### Functional simulation

Since this circuit remains a 16-bit signed adder, the testbench used for this circuit is identical to the one in *Section 3.1*. While the internal architecture differs, its behaviour is unchanged, with the expectation of delivering enhanced performance.

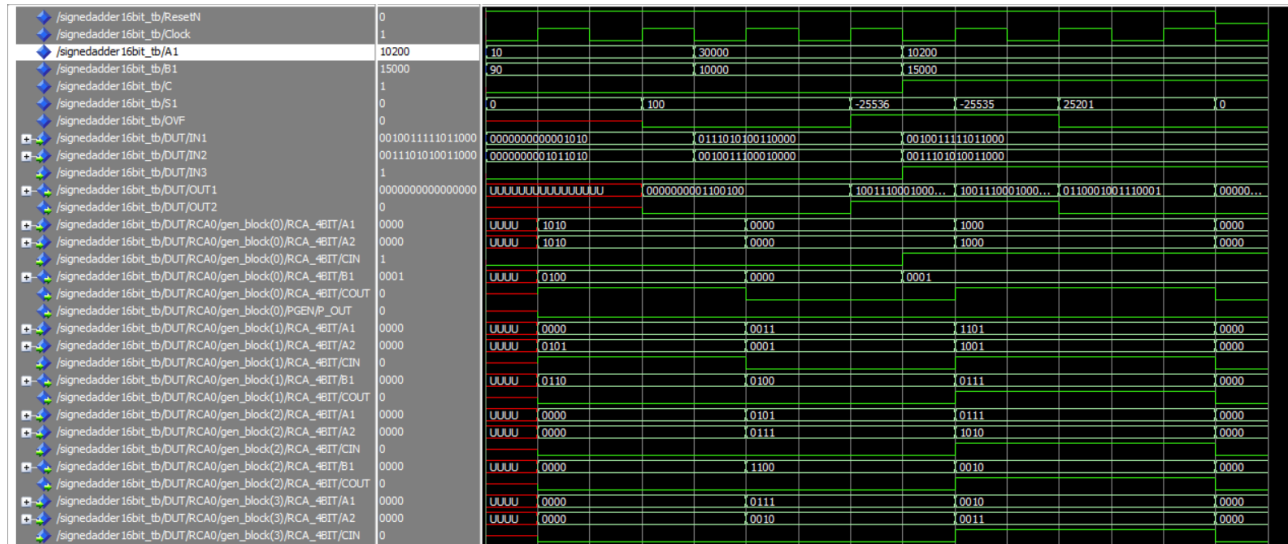


Figure 25 – 16-bit signed adder based on CSKA simulation results in ModelSim

Starting from the top, in the waveforms are represented the active-low reset signal and the clock. Then, the testbench stimuli signals have been represented in decimal form, and the DUT signals have been represented either in logical form or as 16-bit signed values. Below, there are the I/O and internal signals of some RCAs.

### Synthesis and timing analysis

As in the previous case, the practical implementation on the DE1-SoC board cannot be tested due to the insufficient number of available switches, but it is possible to perform the timing analysis. After including the timing constraints file, the tool returns two maximum frequencies. These are slightly lower than the ones of the 16-bit RCA and significantly lower than the ones of 4-bit RCA. This is due to the fact that the tool is evaluating the maximum operating frequency in the worst-case scenario, which in the CSKA is when the signal must travel from the LSB to the output-carry, passing from the multiplexers. The overall performance in standard conditions is higher in terms of achievable clock speed.

Slow 1100mV 0C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	165.29 MHz	165.29 MHz	CLK
Slow 1100mV 85C Model Fmax Summary			
<<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	162.44 MHz	162.44 MHz	CLK

Figure 26 – Timing analysis results for 16-bit signed adder based on CSKA

### Section 3.3: 16-bit carry-select adder

The carry-select adder is another optimized design for implementing a signed adder. It uses 4-bit RCAs and multiplexers, similar to the previous architecture. In this case, each block of bits is equipped with two RCAs: one with an input-carry of '0' and the other with an input-carry of '1'. This approach ensures that partial results are calculated for both possible input-carry conditions. A 2-to-1 4-bit multiplexer then selects the correct result based on the output-carry from the preceding RCA.

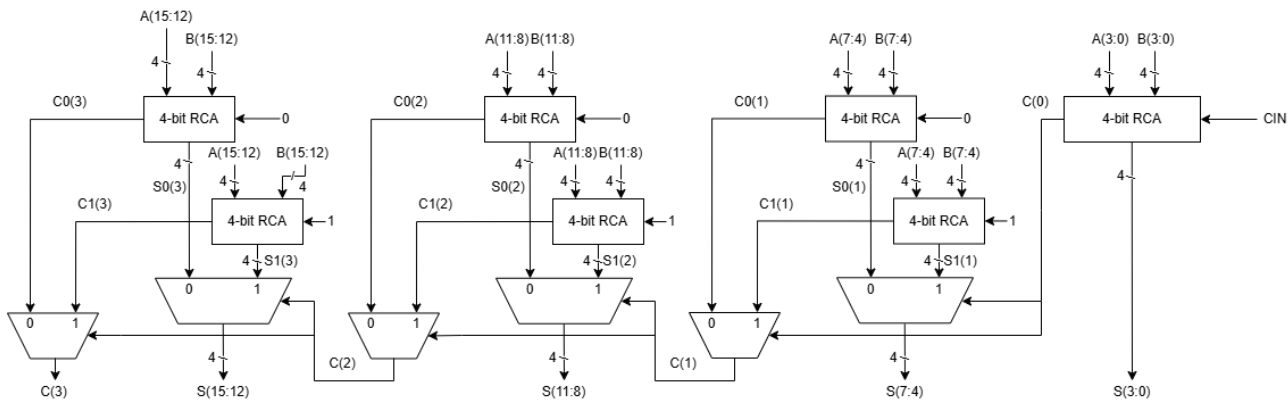


Figure 27 – 16-bit carry-select adder internal architecture

#### Delivered files

Ten VHDL files have been delivered for the description of the component and its simulation:

- *signedAdder16bit.vhd*, VHDL code for the top-level entity, which acts as the 16-bit signed adder
- *signedAdder16bit\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder
- *rippleCarryAdder\_CS\_16bit.vhd*, VHDL code for the 16-bit carry-select adder, built using four 4-bit RCA, four 2-to1 1-bit wide multiplexers and four 2-to-1 4-bit wide multiplexers
- *rippleCarryAdder4bit.vhd*, VHDL code for the 4-bit ripple carry adder, built using four full adders
- *mux4bit.vhd*, VHDL code for the 2-to-1 4-bit multiplexer
- *regn.vhd*, VHDL code for the register
- *ovfdetector.vhd*, VHDL code for the overflow detection circuit, which evaluates whether an overflow has occurred
- *flipflop.vhd*, VHDL code for a D-type flip-flop
- *fullAdder.vhd*, VHDL code for the full adder
- *mux.vhd*, VHDL code for the 2-to-1 1-bit multiplexer

#### Design entry

The top-level entity has not changed since the previous section. The only thing that has changed is the component that performs the sum operation. The *carryBypassAdder16bit* has been replaced with the *carrySelectAdder16bit*.



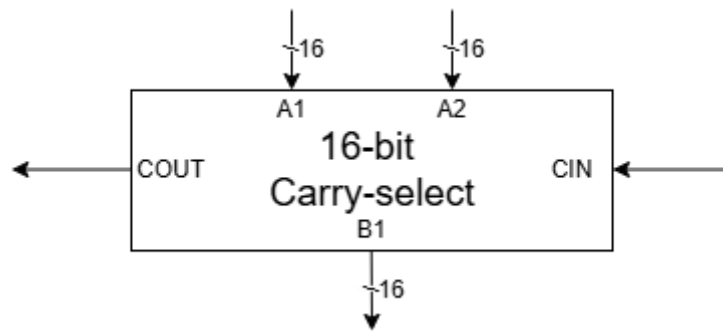
*16-bit carry-select adder*

Figure 28 – 16-bit carry-select adder block representation

The carry-select adder is constructed using a series of components organized into four blocks, with each block processing 4 bits, as shown in Figure 27. Each block consists of two 4-bit ripple-carry adders (RCAs), a 4-bit multiplexer, and a 1-bit multiplexer. The configuration was described in VHDL using the `for...generate` statement. Intermediate signals were structured using vectors and matrices for efficient representation. Specifically:

- C0 and C1: These are 4-bit wide vectors, where the label "0" or "1" indicates the input-carry of the RCA. Each element at position  $i$  represents the output-carry produced by the  $i$ -th block of RCAs, given an input-carry of either '0' or '1'.
- C: This is also a 4-bit wide vector, storing the output-carry of the  $i$ -th block, as determined by the multiplexer.
- S0 and S1: These are matrices defined using the `type` directive. Similar to the carry vectors, the labels "0" and "1" refer to the input-carry of the RCA. The matrices are organized as three rows of 4-bit wide vectors, where the rows represent the sum outputs generated by the  $i$ -th block of RCAs.

It is important to note that for the first block (handling bits 0 through 3), no carry selection is needed. This is because the input carry for the first block is always zero, and as a result, the second RCA in this block is missing.

*Functional simulation*

The simulation of the circuit has been done exactly as described in the previous sections, with the same testbench.


Starting from the top, in the waveforms are represented the active-low reset signal and the clock. Then, the testbench stimuli signals have been represented in decimal form, as described above and the DUT signals have been represented either in logical form or as 16-bit signed values. Below, there are the I/O and internal signals of the first RCA and the second RCA with input-carry equal to '0'. On the bottom, there are the signals from the 4-bit multiplexer.



Figure 29 - 16-bit signed adder based on CSEA simulation results in ModelSim

## Synthesis and timing analysis

The maximum operating frequencies for this circuit are shown in the figure below. These are higher than the ones of the 16-bit RCA and the 16-bit CSKA. This is due to the fact that the critical path for delay is starting to the first full-adder in the first RCA and then it goes through three levels of multiplexers, that is very fast with respect to the other cases, in terms of delay.

Slow 1100mV OC Model Fmax Summary			
 <<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	199.52 MHz	199.52 MHz	CLK


Slow 1100mV 85C Model Fmax Summary			
 <<Filter>>			
	Fmax	Restricted Fmax	Clock Name
1	201.61 MHz	201.61 MHz	CLK

Figure 30 – Timing analysis results for 16-bit signed adder based on CSEA

## Section 4: Multiplier

This section is about a multiplier that, given two signed 4-bit input, compute the 8-bit product between them. The architecture is based on the paper-and-pencil algorithm.

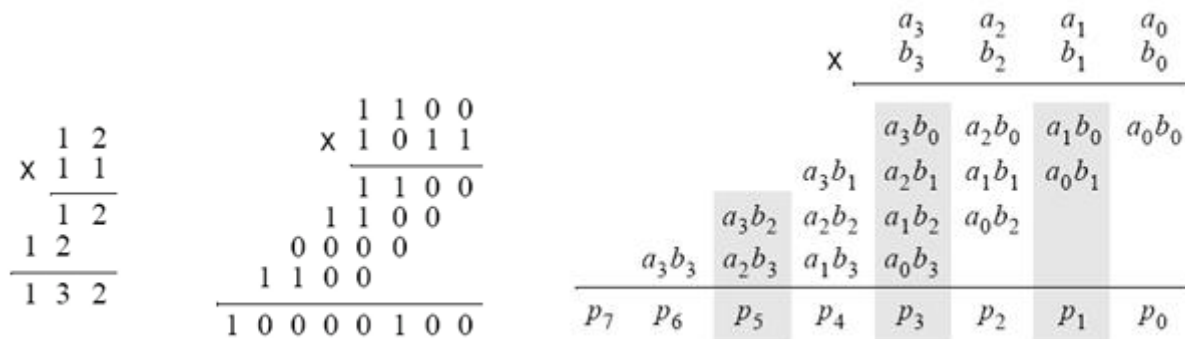


Figure 31 – Paper-and-pencil multiplication and 4-bit array multiplier implementation

Each intermediate result can be represented in a table, while the product final result is the sum of each column. This process involves basically and gates for computing the intermediate results and an array of full-adders to perform the column sum. The architecture becomes the following one.

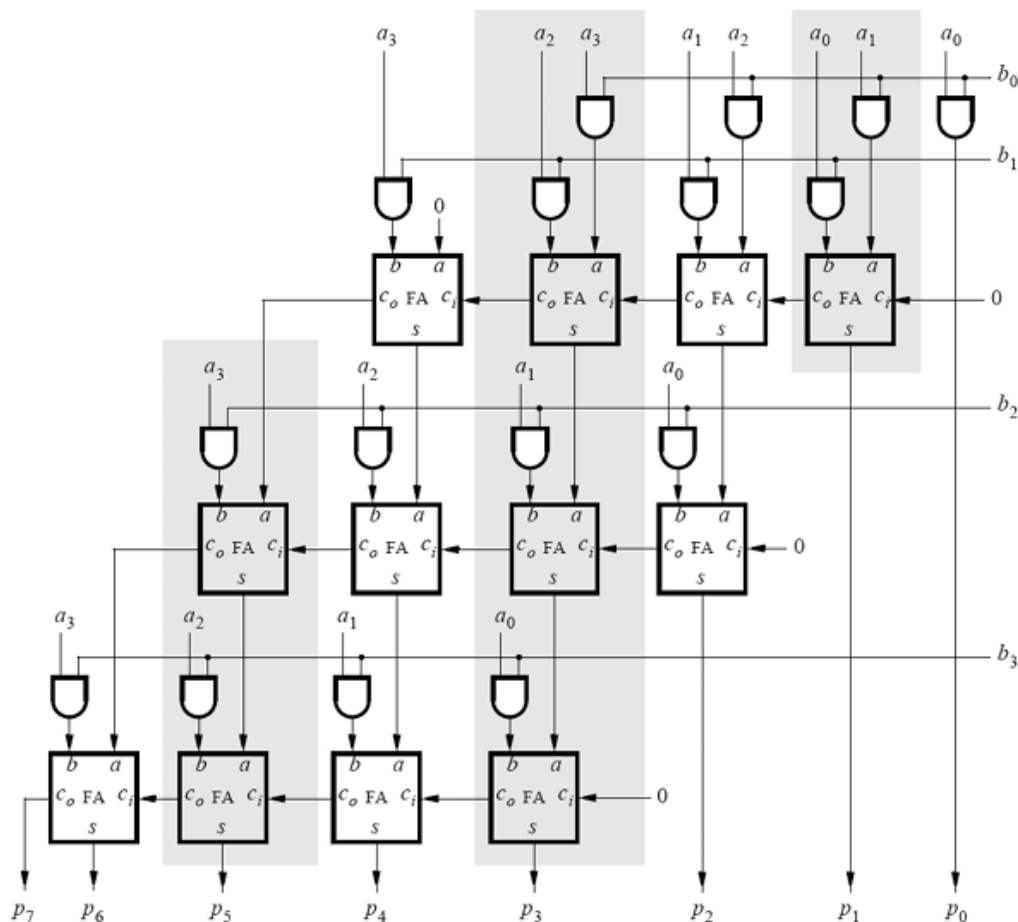


Figure 32 – 4-bit array multiplier internal structure

## Delivered files

Four VHDL files have been delivered for the description of the component and its simulation:

- *multiplier4bit.vhd*, VHDL code for the top-level entity, which acts as the 4-bit inputs array multiplier
- *multiplier4bitbit\_tb.vhd*, VHDL code for the testbench used to validate the functionality of the adder
- *fullAdder.vhd*, which is the VHDL code for the full adder
- *mux.vhd*, which is the VHDL code for the 2-to-1 1-bit multiplexer

These files have been used on ModelSim in order to perform the static simulation of the multiplier.

Two additional VHDL files were provided for testing on the FPGA. The first file, *device\_multiplier4bit.vhd*, serves as the new top-level entity. This file integrates the *multiplier4bit* component modelled and tested in ModelSim, the 7-segment display decoder, and the connections between the DE1-SoC onboard devices and the multiplier component implemented in the FPGA, following the same workflow described in section 1.

The output file generated by Quartus for the Cyclone V 5CSEMA5F31C6 FPGA is named *device\_multiplier.sof*, and it already includes the assignation file *DE1-SoC.qsf*.

## Design entry

### *Multiplier (top-level entity)*

The main entity of the multiplier has two 4-bit wide input ports and one 8-bit wide output port. Its architecture is implemented as an array of full-adders combined with elementary logic gates. Since the full-adders are arranged in a structured manner, we have used two levels of `for...generate` statements: the first level generates an entire row of full-adders, while the second level generates each individual row consisting of four full-adders.

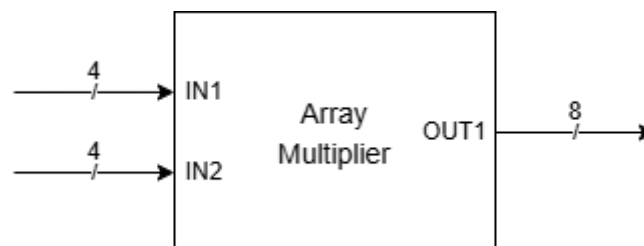


Figure 33 – 4-bit array multiplier block representation

Some auxiliary internal signals have been declared as matrices using the `type` directive. Specifically:

- Signals A1, B1, and S1 are 3x4 matrices. Each cell represents the value associated with a specific input or output of a single full-adder (e.g., B1(2)(1) represents the B input of the full-adder located at row 2 and column 1, with column indexing starting from right to left).

- Signal C1 is a 3x5 matrix that represents the carry outputs of all the full-adders. The indexing of C1 follows the same pattern as the other signals, but it has an additional column to handle the output carry of each row properly.

### Full-adder and multiplexer

The full-adder and the multiplexer entities are the one used in previous sections, without any difference.

### Functional simulation

The testbench for this component is implemented using a process that updates the input values every 20 ns. For simplicity, the stimuli are initially defined as integers and then cast to the appropriate type before being applied to the multiplier.

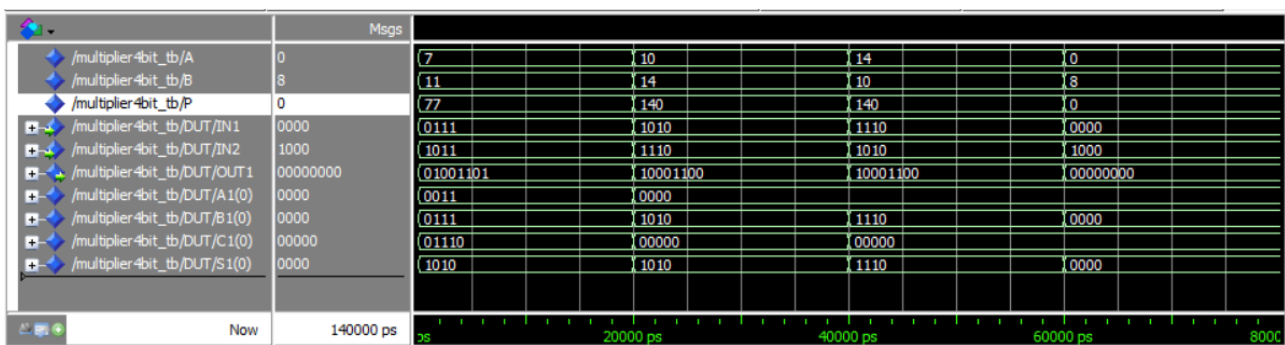


Figure 34 – 4-bit array multiplier simulation results in ModelSim

### Synthesis

The steps to test the functionality of the device on the DE1-SoC board are the same that we have done for the previous exercises.

The two inputs are defined respectively by the position of the switches 0 to 3 and 4 to 7. The hexadecimal value of the two inputs and the result are shown in the 7-segment displays, arranged from left to right. The input A is displayed to HEX5, B is displayed on HEX3, while the result is shown on HEX1 and HEX0, since an 8-bit binary number needs 2 digits to be represented in hexadecimal.

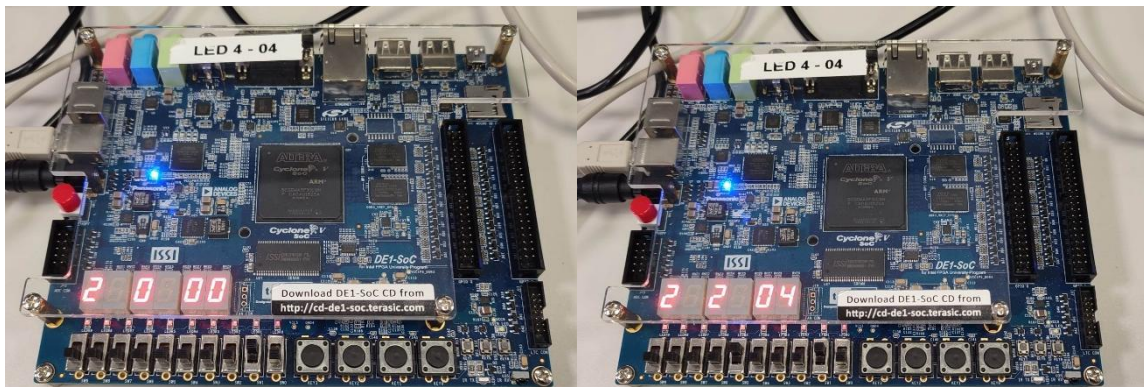


Figure 35 – Practical test of 4-bit array multiplier with the DE1-SoC board. On the left, the two inputs A and B are 0010 (2 in decimal) and 0000 (0) and the corresponding result is 00000000 (0). On the right side, the two inputs are both 0010 (2) and the result is 00000100 (4).



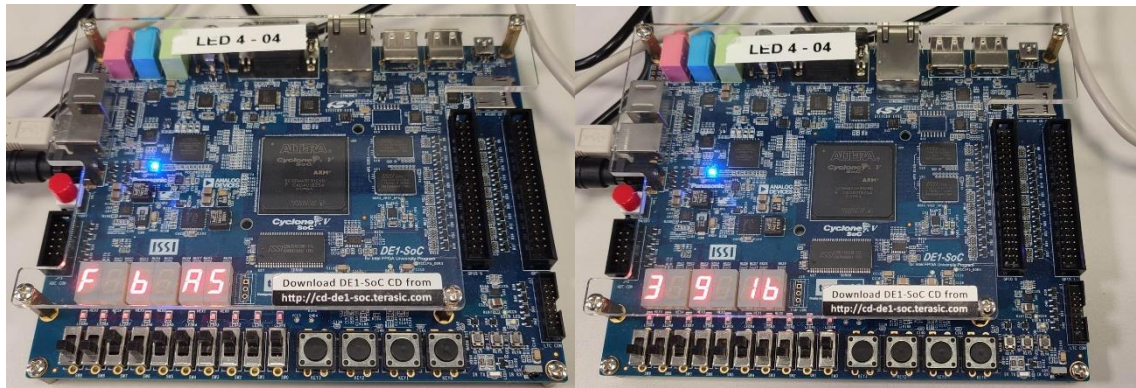


Figure 36 – Other examples. On the left, the two inputs A and B are 1111 (15 in decimal) and 1011 (11) and the corresponding result is 10100101 (165). On the right side, the two inputs are 0011 (3) and 1001 (9) and the result is 00011011.

## Conclusions

This laboratory experience has been very challenging due to the increasing complexity of the exercises. We have gained a deep understanding of how to organize each project in a hierarchical manner and how to edit or improve it by focusing solely on the component of interest, as illustrated in *Section 3*. We have also understood the importance of adopting a modular design approach to enable efficient hardware descriptions using `for...generate` statements and `generic` directives. We encountered no significant issues during the design, validation and synthesis phases.

Timing analysis has proven to be a fundamental tool for determining the maximum achievable speed of the design in advance, allowing for early-stage modifications if the timing requirements are not met.

## Sources and notes

For all the VHDL files provided, an online tool has been used to format the code to ensure coherence between all the different files. The tool is free to use, and it is available at this link: [VHDL Beautifier](#).

For the block diagrams, the online tool used in this document can be reached at this link: [draw.io](#).

The VHDL files were developed using course materials produced by Prof. G. Masera (available on the course webpage) and the following eBooks recommended by the professor:

1. Mealy, Bryan, and Fabrizio Tappero. *Free Range VHDL*. 2016.
2. Ashenden, Peter J. *The VHDL Cookbook*. 1990.