Aeronautics Institute of Technology – ITA
Computer Science Division – IEC

EEA-25 Programmable Digital Systems

Prof. André F. Ponchet
Teaching assistant: Felipe Ferreira

Laboratory 02

Last document update: August 8, 2023

1 Objectives

1.1 General Objectives

- Familiarize yourself with the Lattice iCE40 development environment;
- Understand circuit design process using HDLs and FPGA implementation;
- Simulate and verify results with real FPGA boards.

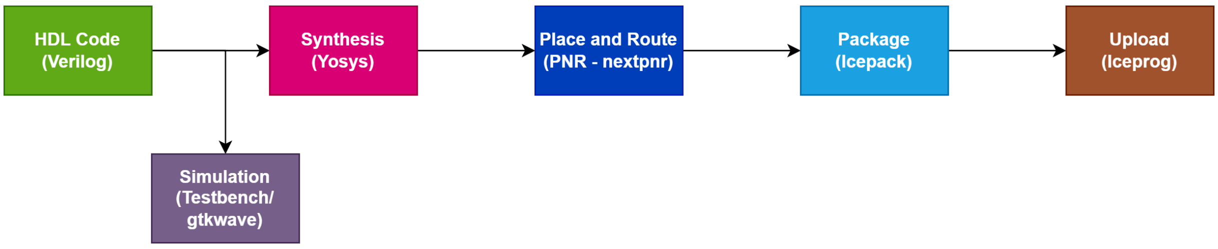
1.2 Specific Objectives

- Understand the HDL design process with Lattice iCE40.
- Implement and test a **full adder** using Verilog and FPGA;
- Focus on the use of continuous assignments to create digital logic circuits, and procedural assignments to perform simulations through testbenches.

For more: We are fortunate that the bistream for the Lattice iCE40 was reverse-engineered by **Claire Wolf**, which enabled the Project **iCEstorm toolchain** and major advancements in the open-source movement for FPGAs. By mastering the basic understanding of the iCEstorm toolchain, you can find out the underlying principles of mainstream tools, such as the QUARTUS maintained by INTEL.

2 Introduction

The process of developing circuits based on digital design can be broken down into smaller steps:



HDL design and implementation process steps.

2.1 HDL Code

In the first step, we describe the digital circuit using a Hardware Description Language (HDL) such as VHDL or Verilog. We will use the latter in the current project due to tool support.

2.2 Simulation

After the code is written and no syntax errors are found, we need to verify if the project specification and circuit description match by simulating input and output signals using a testbench. Simulation is crucial in finding behavior and specification errors before running the code on an FPGA board.

2.3 Synthesis

To work on FPGA, the code is translated into a gate-level digital representation during the synthesis process. Remember that HDL code **does not** work on a procedural level (like some programming languages *i.e.,* C).

2.4 Place and Route (PNR)

According to the HDL description, we need to use the synthesis output in a Place and Route (PNR) tool to place and connect the circuit components inside the FPGA. A ".pcf" file maps the APIO cells with the physical pins on the board. The iCEStorm framework handles such procedures.

2.5 Package

The PNR tool output is a readable ASCII file that details the pin connections inside the FPGA. To translate it into a ".bin" file that supports our board, we will employ the **icepack** tool n the iCEStorm toolkit.

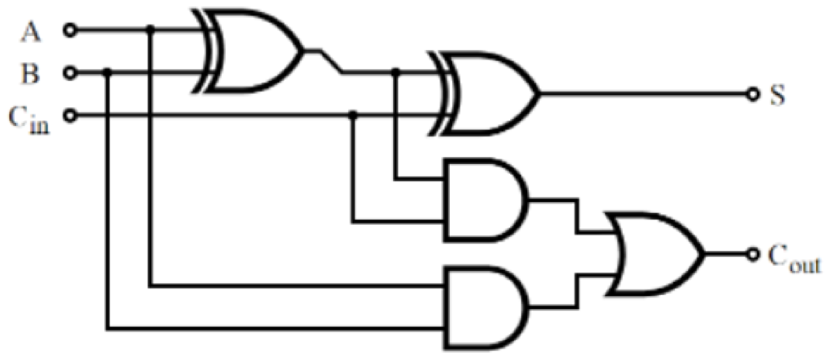
2.6 Upload

Finally, the ".bin" file is loaded into the flash memory of the FPGA board. During startup, the FPGA will read the binary data and configure its logic cells appropriately. The **iceprog** tool in the iCEStorm toolkit will upload the synthesized and routed project to the board.

3 Lab Development

3.1 Structural design analysis

1. There are multiple approaches to circuit design in Verilog. In this laboratory, we are going to use the structural design. The following image describes a full adder:



Full adder schematic.

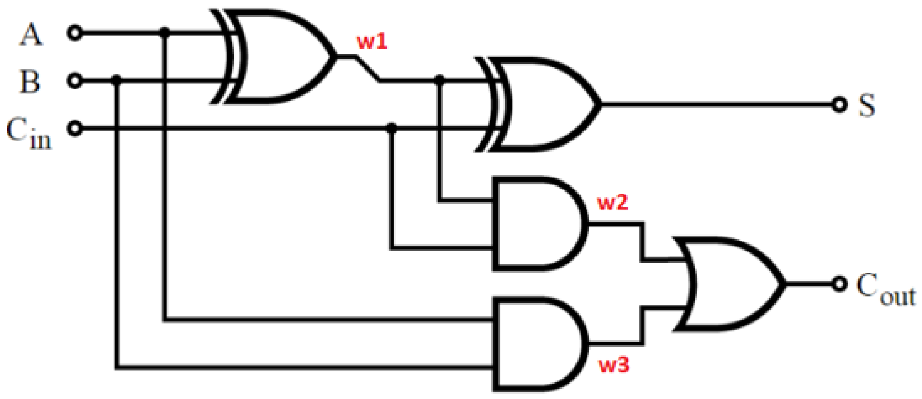
For Remember: The structural approach consists of the steps below:

- Describe the circuit in terms of logic gates, blocks and their interconnections;
- Use primitives to build circuit behavior;
- Elementary blocks (primitives) → logic gates; (AND, OR, NAND, NOR, XOR, etc) <primitive> name (link).

Thus, building the circuit using a structural approach is as simple as reference the appropriate primitives and make the connections between them.

2. Create nodes called "wires" in order to make connections between the input and output nodes, this process can be seen on the next image.

   Notice that the purpose of this type of structure is to describe the circuit based on its diagram.



Full adder schematic with "wire" inputs.

3.2 HDL code implementation

1. Create a folder called "LAB-2";
2. Create a file called "full_adder.v";
3. Inside the new file, create a module and declare the input and output ports, follow the code below:

```
  module full_adder
(
        //Port declaration
            //Inputs
            input a,
            input b,
            input cin,
            //Outputs
            output s,
            output cout
);
```

4. Declare the intermediate wires using the code below:

```
    //Declaration of intermediate wires
         wire w1, w2, w3;
```

5. Now describe the full adder using the structural design approach and declare the end of the module:

```
    //Circuit operation - structural approach
    xor u0 (w1, a, b);
    xor u1 (s, w1, cin);
    and u2 (w2, cin, w1);
    and u3 (w3, a, b);
    or u4 (cout, w2, w3);
endmodule
```

3.3 Simulation and testbench

1. A testbench is needed to simulate the full adder. So, inside the same folder create a file named "full_adder_tb.v" and map the same input and output nodes:

```verilog
// tb for full adder

`include "full_adder.v"

module full_adder_tb;

    reg A, B, CIN; // UUT inputs
    wire S, COUT; //  UUT outputs

    // full adder instance
    full_adder uut( .a(A), .b(B), .cin(CIN),
                    .s(S), .cout(COUT));

    initial begin
        $dumpfile("full_adder_tb.vcd");
        $dumpvars(0, full_adder_tb);
    end

    initial begin
        A = 1'b0;    B = 1'b0;    CIN = 1'b0; #1


        A = 1'b1;    B = 1'b1;    CIN = 1'b1; #1;
    end

endmodule
```

2. The testbench is incomplete, it only appears for the first and last possible cases. Complete the missing cases for the testbench file.
3. Open the Linux terminal inside the folder, enter the following command:

```
 iverilog full_adder_tb.v
```

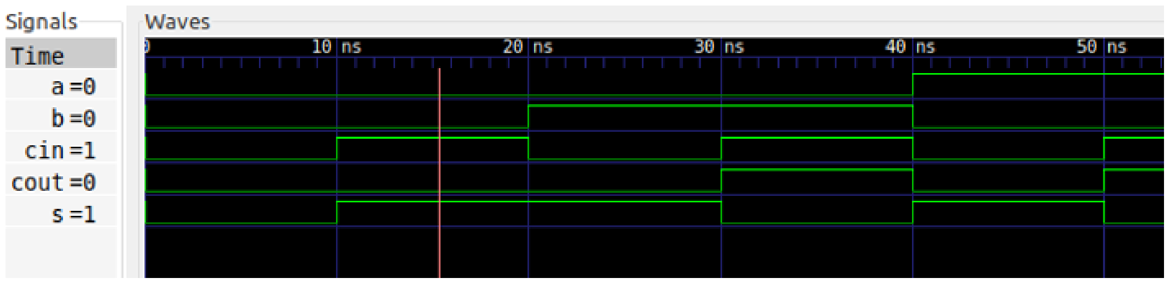
4. If no error occurs, a file named "a.out" will be created by IcarusVerilog, now enter the command:

```
vvp a.out
```

5. Open "full_adder_tb.vcd" with gtkwave:

```
gtkwave full_adder_tb.vcd
```

6. Select all input and output signals and click "append" to add to the simulation screen, next click in "Zoom fit" to adjust the window. Compare your results with the next image:



Full adder simulation.

- Write the truth table below with your answers obtained from the full adder.
- Do the values found match the expected output for a full adder?

Inputs			Outputs	
A	B	C_{in}	S	C_{out}
0	0	0		
0	0	1		
0	1	0		
0	1	1		
1	0	0		
1	0	1		
1	1	0		
1	1	1		

Truth table for the full adder.

3.4 Pin constraint file and configuration

1. Now we will map the full adder inputs and outputs to our FPGA board. Create a file named "full_adder.pcf" inside the folder. Map the inputs based on the board datasheet or implement:

```
# Pin Constraint- ICE-40LP1K

#  PMOD I/O in use - OUTPUT-FULL-ADDER
set_io  cout    C5
set_io  s       B3

#  PMOD I/O in use - INPUT-FULL-ADDER
set_io -pullup yes a    C6
set_io -pullup yes b    B4
set_io -pullup yes cin  B5

set_io --warn-no-port PMOD4 E3
set_io --warn-no-port PMOD5 E1
set_io --warn-no-port PMOD6 C2
set_io --warn-no-port PMOD7 B1
set_io --warn-no-port PMOD8 A1

set_io --warn-no-port PMODL2 A3
set_io --warn-no-port PMODL3 B6
set_io --warn-no-port PMODR1 A1
set_io --warn-no-port PMODR2 B1
set_io --warn-no-port PMODR3 D1
set_io --warn-no-port PMODR4 E2
```

3.5 Synthesis, PNR and package

1. Enter the following command in Linux terminal to make the circuit synthesis:

```
yosys -p "synth_ice40 -json full_adder.json -blif full_adder.blif"
         full_adder.v
```

2. If no mistakes were made, yosys will create both ".json" and ".blif" files, use them in the PNR process with the next command:

```
nextpnr-ice40 --lp1k --package cm36 --json full_adder.json --pcf
               full_adder.pcf --asc full_adder.asc --freq 48
```

3. A file with ".asc" extension will be made if no errors were detected.

3.6 Full adder implementation in board

1. Enter the following command in Linux terminal to create the ".bin" file:

```
icepack full_adder.asc full_adder.bin
```

2. Now connect the board USB to the computer you are using and type:

```
sudo iceprog full_adder.bin
```

3. If no errors were found, your full adder implementation was written inside the board.

3.7 Automation

Write a makefile to summarize and run the whole described process.

4 Grading

This lab does not require a report. When finishing it, make sure you enter the line for a direct assessment conducted by the professors or TAs during lab timeframe.