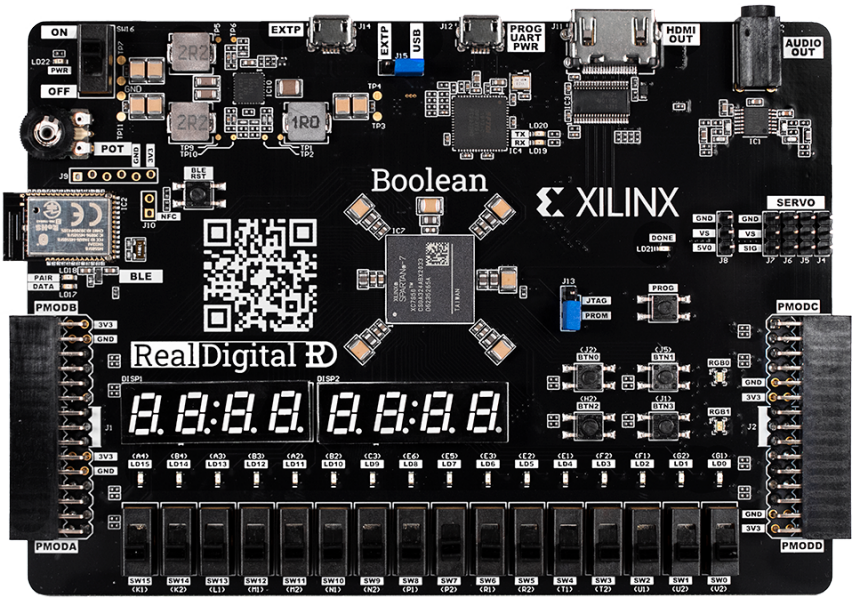
# Digital Electronics – Lab 2

D-type flip-flop / register

Kent Abrahamsson, Andreas Axelsson

2026-01-18



# 1 Revision history

| Date | Author | Description |
|---|---|---|
| 2023-03-27 | Kent Abrahamsson | First edition |
| 2023-04-01 | Kent Abrahamsson | Minor typo fix |
| 2026-01-10 | Andreas Axelsson | From Altera to Xilinx |

# Contents



# 2 General

## 2.1 Background

This document is intended to be used as Lab instructions for the flip-flop / register lab lab in the VHDL course.

## 2.2 Intended outcome

In this assignment you shall implement a couple of simple logic functions to gain basic knowledge about how to describe various logic functions using VHDL which is synthesized to hardware.

All outputs shall be synchronized (driven) with a D-type flip-flop/register. The clock source for the register shall be taken from one of the pushbuttons called "BTN*" on the development board.

By using a push button to generate the clock signal the student can study the code, synthesis result and cycle by cycle understand how the signals flow trough the register(s) in the design.

The lab is split up in three different lab parts.

## 2.3 Checkpoints

In order to Pass the lab assignment the student shall be able to show that the following tasks have been fulfilled.

- Different Vivado projects have been created for each lab part. Each project shall be stored in its own project folder. Clone the repo from GitHub Classroom, and use create_project.tcl to create each project.
- Properly written VHDL implementation for all three projects. Use proper indentation, spaces are cheap! Add comments when you feel it's suitable.
- Student have understood how pins are mapped to the signal names in the design entity via the constraint file.
- A simulation shall (of course) be performed for all parts but it is enough to show Simulation for one of the parts.
- The RTL (Netlist) viewer shall be viewed and analyzed for all parts.

**Tip :** Make sure you have a look at the testbench overview in the appendix.

# 3 Lab instructions

## 3.1 General

To ease implementation and understanding of your code it is a good idea to keep your top level design entity in line with the signals on the development board.

If we want to use some signals from e.g. a key-button we can easily create an internal signal with a name that makes more sense for us in the current lab. By doing this way we can also reuse the top level entity for several projects in the future.

By naming the signal names in the top level entity we can understand directly where these are expected to be controlled from just by looking at the development board.

These are easily "pin-mapped" to the correct pin by looking at the development board User Manual.

In the architecture, before "begin", internal signals are declared with a name that reflects their functionality or purpose. After "begin" in the architecture we assign the internal signals from the entity signals. Just by looking at the code we can understand that the development board "BTN0" pushbutton is used as clock signal, and "BTN1" is used as an active high reset signal.

**Study the schematic of the Boolean board for the BTN push buttons to understand how these work.**

Typically we have a clk coming in as an input in the entity. But since we want to use a BTN input to act as the clock we can create an internal clk signal in our architecture and map for instance btn[0] to it. Make sure to comment on the purpose of this in your code so it is easier to understand as you come back to the project at a later stage. Also note that the reset signal also comes from a BTN input as described earlier. As for clk we need to create an internal signal reset that is tied to for instance btn[1].

*NOTE: Do you need to change anything in the constraint file and the the entity port mapping to be able to access btn[1] as well as btn[0]?*

## 3.2 Lab Part 1

Implement an adder circuit that adds two 3-bit numbers. The input signals can be connected to 6 switches on the Boolean board. The output from the adder shall be presented as a binary number on 4 LEDs. You must use vectors and the implementation shall be in one single process.

Use an asynchronous reset of the output register.

Use one of the pushbuttons as the reset input.

Please see the course lecture notes for tips on how to implement this adder, or google it.

Hint: use the "+" sign and the strategy to assign internal signals from the switches as shown above.

## 3.3 Lab Part 2

Implement a 4-1 MUX (Multiplexer).

A 4-1 multiplexer has 4 inputs (channels) and a two bit channel selector inputs and one output. Use the switches as inputs and one LED as output.

You must use vectors and a case-when structure.

Use an asynchronous reset of the output register. Use one of the pushbuttons as the reset input.

## 3.4 Lab Part 3

Implement a comparator which compares two 4-bit vectors called A and B below (use any name you wish). Your design shall use three outputs, which are connected to three LEDs on the Boolean board. The following functionality shall be implemented.

If A $>$ B the output LEDs shall be set to "100" (1 means on, 0 means off)
If A $<$ B the output LEDs shall be set to "001"
If A $=$ B the output LEDs shall be set to "010"

Use the switches as inputs and three LEDs as output. Use an asynchronous reset of the output register. Use one of the pushbuttons as the reset input.

### 3.4.1 Signed / Unsigned comparison

Try to change the code in Lab Part 3 above to do a signed interpretation of the input values, or have one signed and one unsigned vector. Notice any difference in the behaviour?

# A VHDL Testbench Overview

A testbench is a *simulation-only* VHDL design used to verify that a circuit behaves as intended before it is implemented in hardware.

## Purpose of a Testbench

A testbench serves the following purposes:

- Instantiates the design under test (DUT)
- Generates stimulus signals (inputs)
- Observes and verifies output signals
- Is used exclusively for simulation and is never synthesized

Simulation using a testbench is one of the most important verification steps in digital design.

## High-Level Structure

A VHDL testbench typically consists of the following elements:

1. Library and package declarations
2. Testbench entity (without ports)
3. Internal signal declarations
4. Instantiation of the design under test (DUT)
5. One or more stimulus processes
6. Optional result checking and assertions

Each element is described in more detail below.

## Library and Package Declarations

Testbenches generally use the same standard IEEE libraries as synthesizable VHDL designs:

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

Additional packages may be used for text output, randomization, or advanced verification, but are optional.

## Testbench Entity

A testbench entity has no ports, since it does not connect to external hardware.

```
entity tb_top is
end entity;
```

All interaction with the DUT takes place via internal signals declared in the testbench architecture.

## Internal Signals

Signals are declared to connect the testbench to the DUT. These signals typically correspond directly to the DUT ports.

```
signal sw  : std_logic_vector(5 downto 0);
signal btn : std_logic_vector(0 downto 0);
signal led : std_logic_vector(3 downto 0);
```

## Instantiation of the Design Under Test

The DUT is instantiated in the testbench using a normal component or entity instantiation.

```
uut : entity work.top
  port map (
    sw  => sw,
    btn => btn,
    led => led
  );
```

## Stimulus Process

One or more stimulus processes are used to drive input signals and define test scenarios. Stimulus processes typically use `wait for` statements to control simulation time. A process without a sensitivity list will automatically restart if it doesn't end with a `wait` (see clk_process below).

```
stim : process
begin
  sw <= "010001";
  btn(0) <= '1';
  wait for 10 ns;
  btn(0) <= '0';
  wait;
end process;
```

Stimulus processes execute sequentially and may model clocks, button presses, resets, or data changes.

## Clock Generation (Optional)

If the DUT uses a clock signal, the testbench may generate a periodic clock:

```
clk_process : process
begin
  clk <= '0';
  wait for 5 ns;
  clk <= '1';
  wait for 5 ns;
```

```
end process;
```

In simpler demonstrations, clocks may instead be generated manually using stimulus events.

## Checking Results and Assertions

Testbenches may include assertions to automatically verify correct behavior:

```
assert led = "0011"
  report "Unexpected output value"
  severity error;
```

Assertions improve test quality by making failures explicit during simulation and are ignored during synthesis.

## End of Simulation

A testbench process typically ends with a `wait;` statement, which halts execution and allows waveform inspection:

```
wait;
```

## Key Characteristics of Testbenches

- Testbenches are not synthesizable
- Timing control using `wait for` is allowed
- Assertions and file I/O may be used
- Testbenches can be reused across multiple design iterations

## Summary

A VHDL testbench is a structured simulation environment whose purpose is to drive inputs, observe outputs, and verify correctness before hardware implementation. Students are **strongly** encouraged to simulate all designs using a testbench prior to synthesis and implementation.