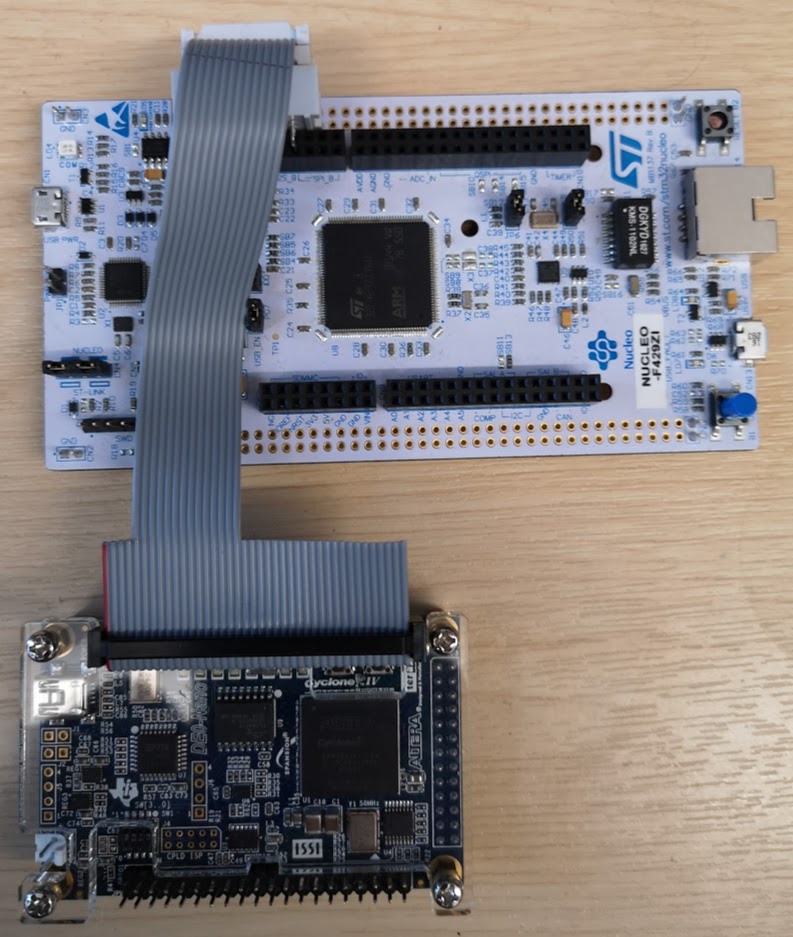
Lab 02-03 Digital Systems Design

For this lab, you will need the following:

* FPGA board
* Nucleo F429ZI Microcontroller board
* Ribbon connector
* Lab02-03.zip

# Getting Set Up

You first need to connect your MCU to the FPGA as shown in the figure below:



# Task 01 – Internal Bus Structures

In the lecture, we talked about the internal architecture of a CPU. Central to this was the transfer of data from register to device and back to register over a common bus. This is in contrast to multiplexing data from component to component.

An internal data bus requires support for internal tri-state logic. This is a strategy commonly used in ASIC devices to save space, but we cannot assume this is directly achievable in an FPGA. To illustrate this important point, let us first consider a simple example of two registers with outputs sharing a common bus:

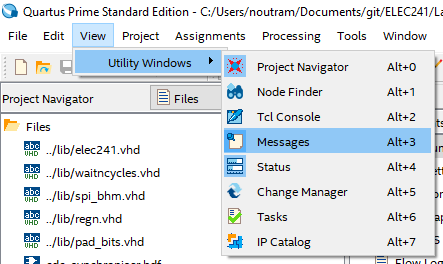
* Download Task-01.zip and extract into a location with no spaces in the path.
* Open the project in Task-01
* Note any warnings – especially those relating to “multiplexers”
* Build the project and deploy to FPGA

We are now going to take a closer look at what was actually synthesised.

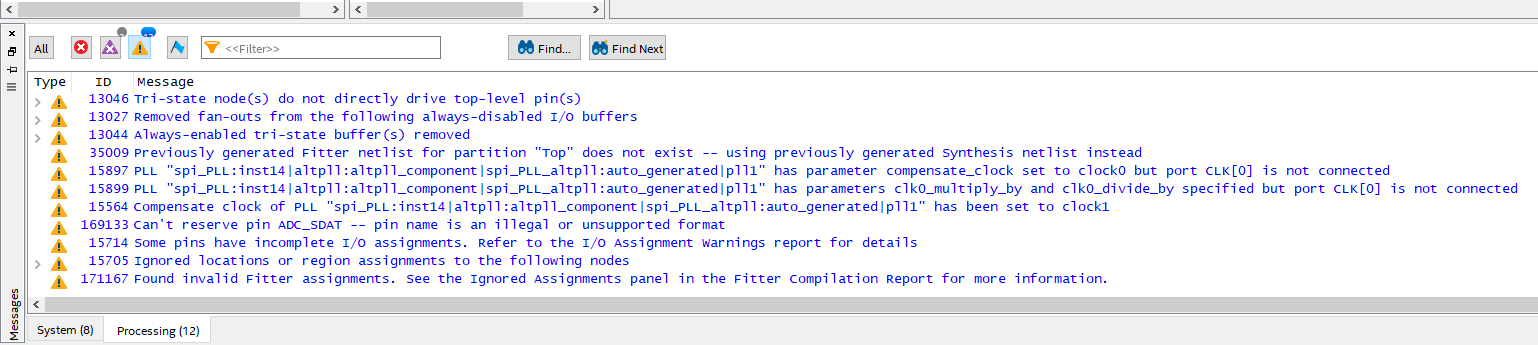
* Click Tools -> Netlist Viewers -> RTL Viewer (see Figure 1)

You should see a schematic similar to that in Figure 2. Note the TRI BUS component is actually a multiplexer. The compiler should warn you about this.

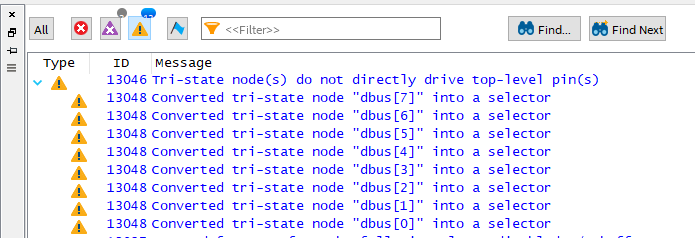
* Make sure the message windows is visible (see below).



* In the messages window (below), click the warning triangle.



* Expand the message that reads *“Tri-state node(s) do not directly drive top-level pins(s)”*



This is an implied behaviour and creates uncertainty for the designer. To illustrate, let us now make a serious design error by creating a bus contention. This is achieved by allowing two devices to assert the ‘bus’ at the same time.

* In controller.vhd, change lines 54 and 56 to read as follows

R1\_OE <= '1'; --line 54

R2\_OE <= '1'; --line 56

This now means that the outputs of two connected devices are **both** enabled. *This is an illegal condition with a bus structure* (and could have serious consequences in terms of power and power-supply stability). For a MUX based design, this makes little logical sense, as two signals cannot be routed to the output at the same time.

* Without deploying to the FPGA, simple re-build the project and look at the warnings
* Look at the warnings. Do any of the warnings direct you towards the error?

It is interesting that there is a warning that the LEDs are stuck at ground. This could have you looking for some time!

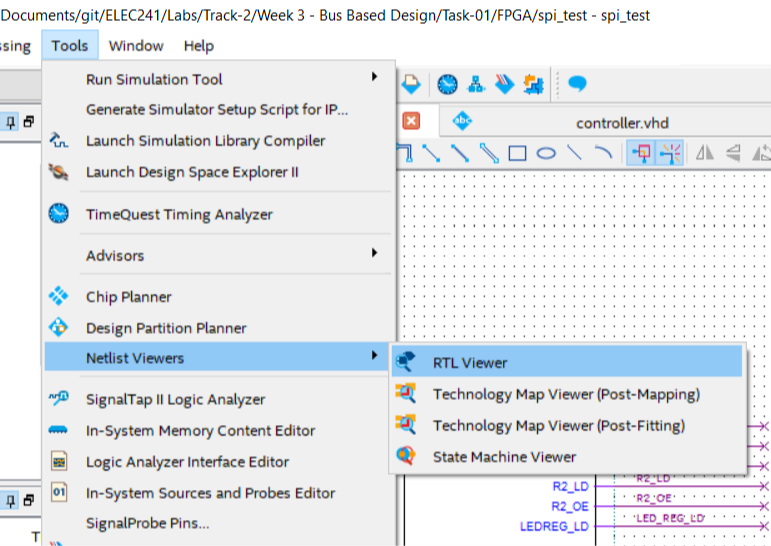


Figure 1. Examinging the FPGA synthesis

As bus structures are converted to multiplexer based designs by the synthesis tools, it is much simpler to design (and debug!) systems that explicitly use multiplexers.

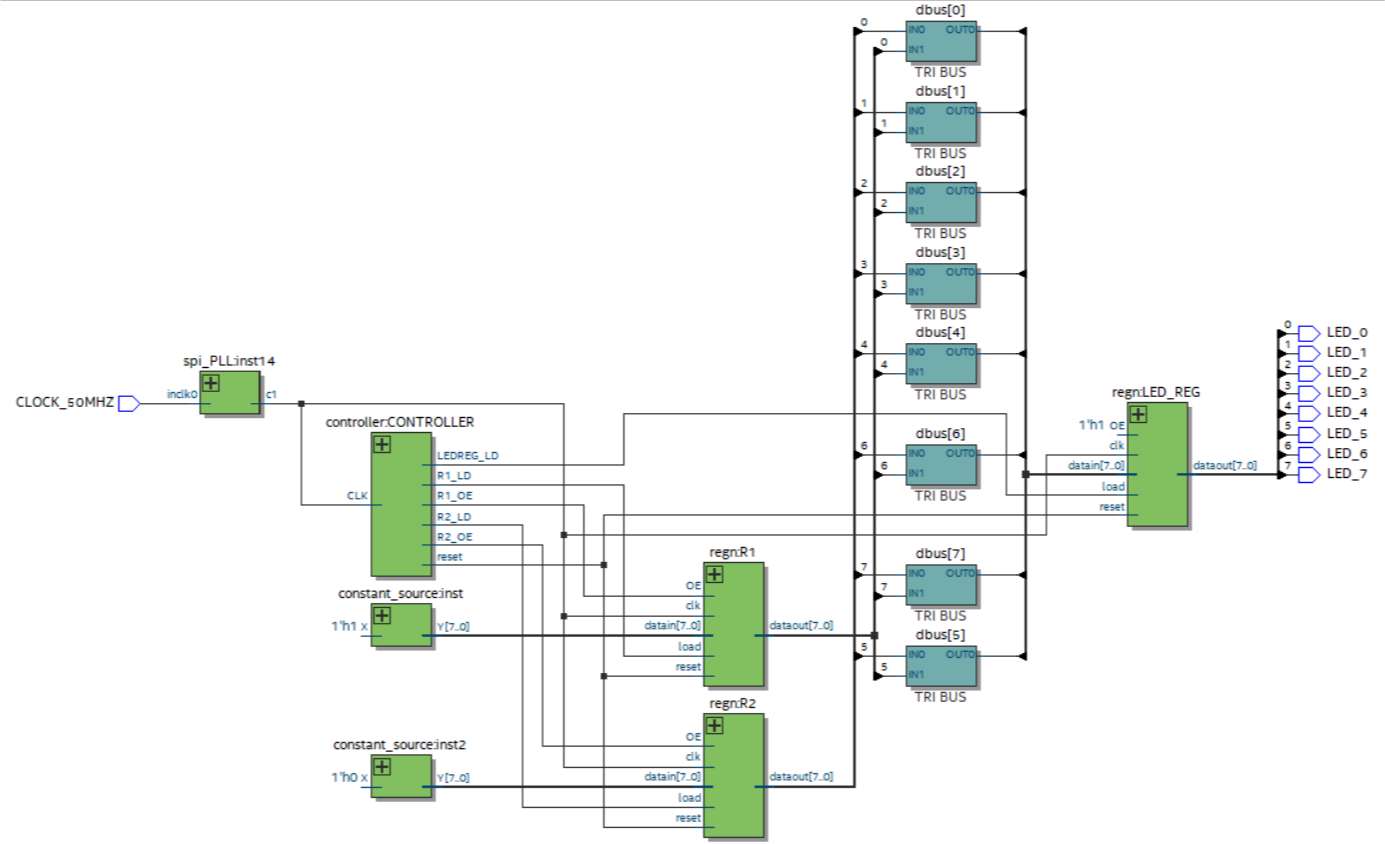


Figure 2. Upon closer inspection, we see that the synthesis tools have converted the internal bus into a multiplexer tree. The TRI BUS components are actually 2-1 multiplexers.

## TASK 02

Convert the design to use multiplexers instead of a bus structure.

The datapath and controller should resemble that shown in Figure 3 and Figure 4 respectively.

For this, you should do the following:

* Modify the register component so that it no longer has an output enable. Outputs should be zero when reset.
* Add a multiplexer to the top-level design (mux\_21.vhd is included in the project).
* Modify the controller to output a select line instead of “output enable” signals.

Note that a solution is provided in Task-02-solution.zip.

A significant advantage of this approach is that the architecture is explicit (the only implied behaviour is the latching of the registers). Note how the output signals of registers feed into a MUX. The controller choses which of the sources feeds into the LED register. Therefore, there is no possibility of bus contentions (as there is no bus!).

This highlights a disadvantage of an FPGA – the number of gates required to multiplex signals can become very large.

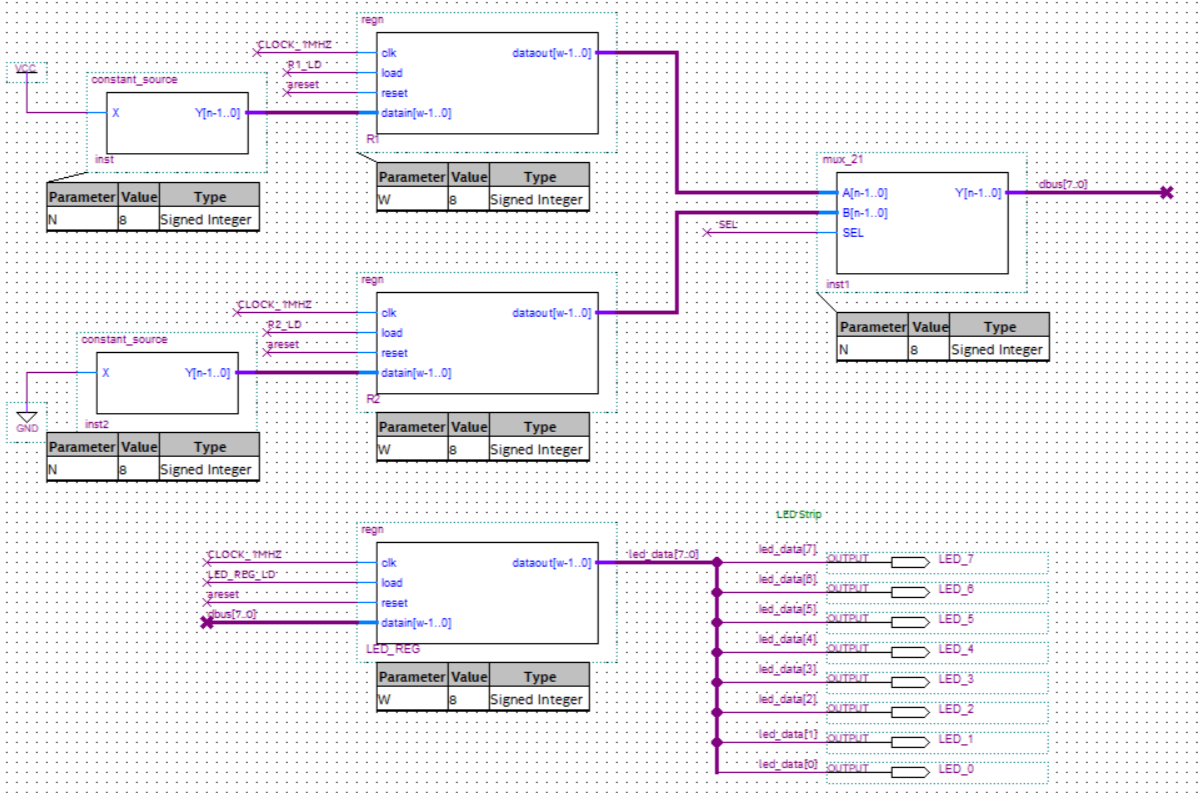


Figure 3. The datapath components of the MUX based design (from the top level schematic)

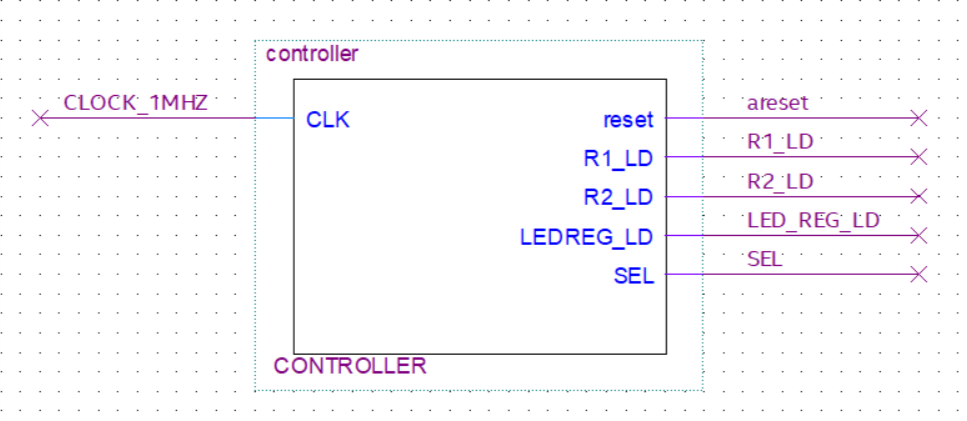


Figure 4 The controller component of the MUX based design (from the top level schematic)

## Task 03 – Register Based Design (advanced)

Unzip and open Task-03.zip

* Open the MCU project in Keil uVision. Build and deploy this code to your microcontroller.
* Open the FPGA project in Quartus. Build and deploy to the FPGA.
* You should see a moving light pattern.
* Open PuTTY and view the output from the microcontroller.

*Take time to familiarise yourself with the top-level schematic of this design*. This is a fairly complex system which needs to be studied before you can progress in the tasks below.

## Datapath

There are number of **datapath** components:

* Switches + Switch Register
* LEDs + LED register
* SPI Slave component
* SPI Transmit register (holds value to return to the MCU)
* SPI Receive register (holds value set by the MCU)
* Clock synchronisation logic

A very important component is the multiplexer (MUX) block. This takes input from all possible data sources and routes them to a single output. The MUX output is then connected to all possible destinations. Only destinations with the LD pin asserted HIGH will read this data.

Note which register outputs are connected to the multiplexer.

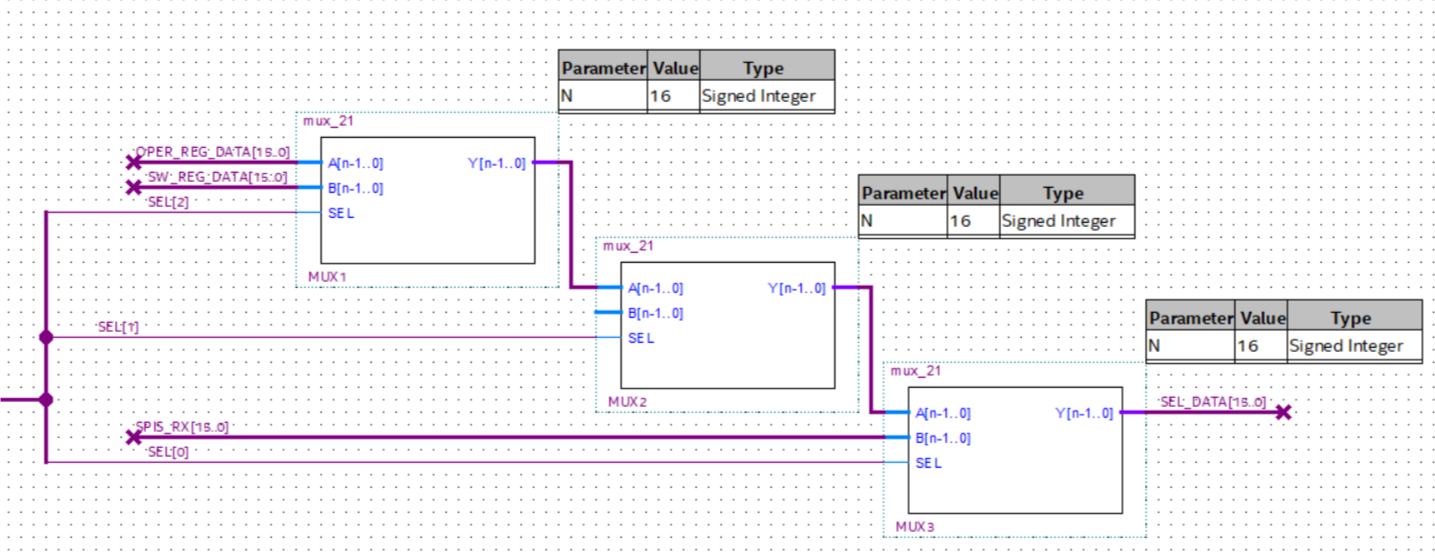


Figure 5. The MUX tree, showing the three existing data sources (OPER\_REG\_DATA, SW\_REG\_DATA and SPIS\_RX). MUX2 has a spare (currently unassigned) input. The output SEL\_DATA is routed to a number of destinations.

## Controller

There is a single **controller** in this system. It is the task of the controller to listen to the SPI system, examine the data set from the microcontroller and sequence the MUX select lines and the LD inputs of all destinations.

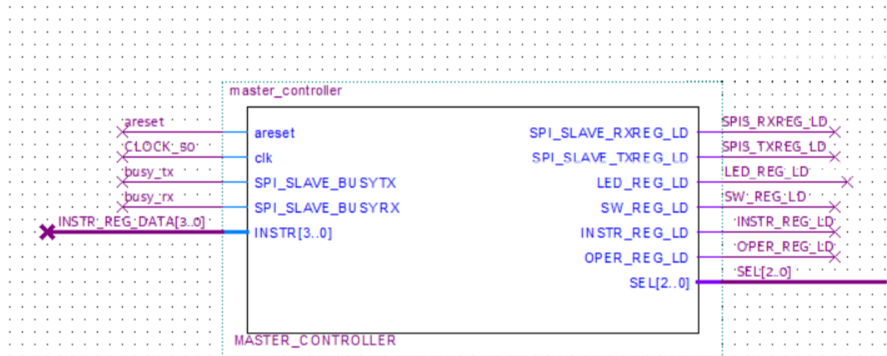


Figure 6. Controller component that manages the MUX selector and register load (LD) signals.

Central to this design is the SPI slave. The MCU will send 16-bit data over an SPI interface in the following format:

[-- INSTRUCTION -- | --------- OPERAND --------- ]

The instruction is 4-bits wide and the operand (data) is 12-bits. The operand is purely data. The instruction tells the controller what operations to perform.

An SPI word is detected when the following sequence is received:

BUSY\_RX -> HIGH

BUSY\_TX -> HIGH

BUSY\_RX -> LOW

BUSY\_TX -> LOW

Once this pattern is detected, it is safe to read and write data from/to the SPI slave, at least until one of these signals goes HIGH again.

* The SPI interface is running at 1MHz
* The MCU is running at 50MHz

*Therefore you have at least 50 clock cycles to perform all actions. In reality, it is much more.*

## TASK

Study the controller component in the project. Sketch an ASM chart for the controller[[1]](#footnote-1)

Add an additional instruction “increment” to return the “operand + 1”

Note that the MUX tree has a spare input for this purpose

1. State minimisation has not yet been performed on this, so you may find some redundant states [↑](#footnote-ref-1)