Practicum 3

[Home] [Back]

Contents

Introduction

[Contents]

In this practicum you will learn how to drive a simple **7-segment display** in order to display a **4-bit Binary-Coded Decimal (BCD) word** as a human-readable **base-10 decimal number**.

For this purpose a **binary to 7-segment display decoder** is required. Despite very popular dedicated integrated circuits exist for this task (such as the **CMOS 4511** or **TTL 7447**) you will implement the decoder in FPGA using the Verilog HDL.

Practicum aims

[Contents]

This practicum should exercise the following concepts:

- review the Binary Coded Decimal (BCD) code
- introduce the 7-segment display electronic component
- learn the difference between common-anode and common-cathode displays
- write a suitable combinational block in Verilog to drive a 7-segment display
- implement the circuit on breadboard
- debug the firmware on real hardware
- avoid to infer unwanted latches when writing combinational code

Navigate to the practicum directory

[Contents]

As a first step, open a **terminal** window and change to the practicum directory:

```
% cd Desktop/lae/fpga/practicum/3_seven_segment_display
```

List the content of the directory:

```
% ls -l
```

Explore datasheets

[Contents]

Datasheets for the 7-segment display available in the lab have been placed in the `doc/datasheets` directory:

```
% ls -l doc/datasheets
```

Open the required PDF according to the 7-segment display module you are working with. Understand the working principle of the 7-segment display module.

> QUESTION
>
> The 7-segment display you are working with is a **common-anode (CA)** or a **common-cathode (CC)** module ?
>
> ____

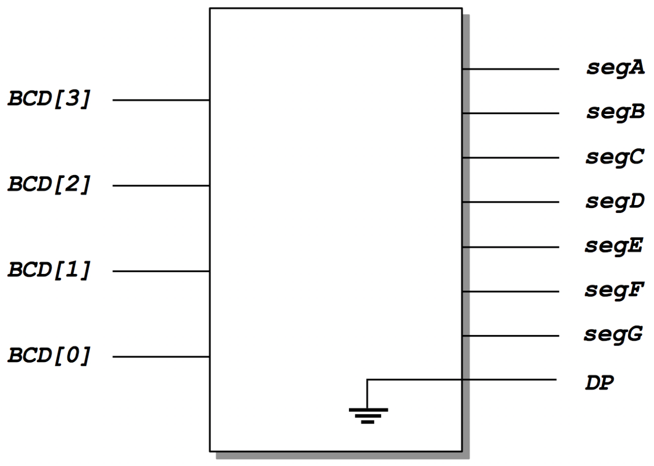
RTL coding

[Contents]

Create with your **text-editor** application a new **Verilog file** named `SevenSegmentDecoder.v` as follows:

```
% gedit SevenSegmentDecoder.v &   (for Linux users)
% n++ SevenSegmentDecoder.v       (for Windows users)
```

The `SevenSegmentDecoder` Verilog module that you are going to implement has **4-inputs** corresponding to a 4-bit binary code interpreted as BCD, e.g. `BCD[3:0]` or `Bin[3:0]` and **8-outputs** to drive the LEDs of the 7-segment display. e.g. `seg[7:0]` or `segA` , `segB` etc. If you want you can also decide to simply tie-down the decimal point (DP) LED of the display.



As an example, the main `module` declaration could be the following:

```
module SevenSegmentDecoder (

   input wire [3:0] BCD,

   output wire DP,
   output ... segA,
   output ... segB,
   output ... segC,
   output ... segD,
   output ... segE,
   output ... segF,
   output ... segG ) ;

   assign DP = 1'b0 ;


   ...
   ...

endmodule
```

The **digital functionality** that we want to implement is to **map** a 4-bit Binary-Coded Decimal (BCD) word `0000` , `0001` , ... , `1001` into another 8-bit binary code (or 7-bit if you decide to tie-down the decimal point) corresponding to the on/off status of the 7-segment display LEDs. Therefore the block is a **decoder**.

Try to **write yourself** synthesizable Verilog code that implements this functionality. Remind that you can **check for syntax errors** at any time by **compiling the code** with `xvlog` at the command line:

```
% xvlog SevenSegmentDecoder.v
```

HINT

> The block is a pure **combinational circuit**, therefore you can use a **truth-table**. Alternatively, from the truth-table you can also write a **Karnaugh map** for each decoder output and derive **logic equations** to drive the LEDs of the 7-segment display.

Design constraints

[Contents]

In order to map the Verilog code on real FPGA hardware you also need to write a **constraints file** using a **Xilinx Design Constraints (XDC) script**.

Create with your **text-editor** application a second source file named `SevenSegmentDecoder.xdc` as follows:

```
% gedit SevenSegmentDecoder.xdc &    (for Linux users)

% n++ SevenSegmentDecoder.xdc        (for Windows users)
```

Copy from the `.solutions/` directory the reference XDC file for the Arty board:

```
cp .solutions/arty_all.xdc
```

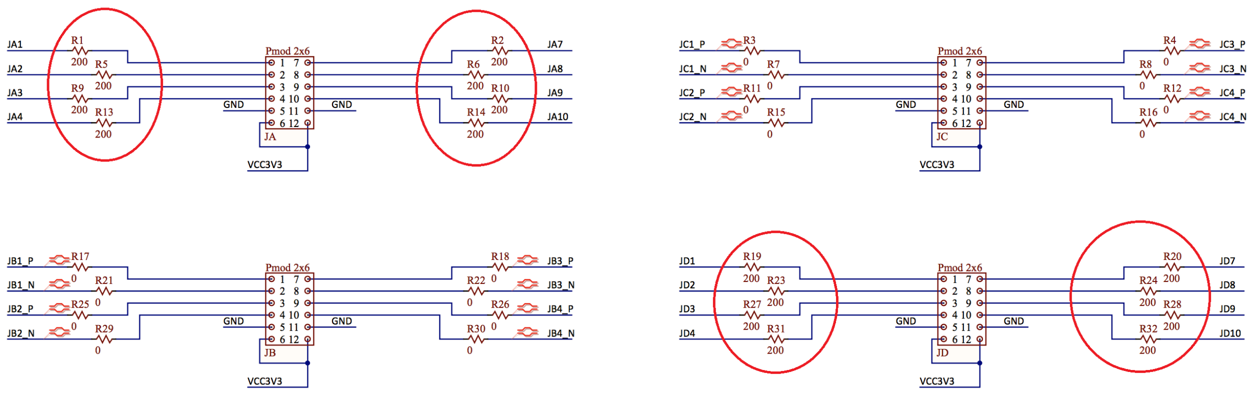
Try to **write yourself** design constraints required to implement the design on real hardware. Use the main `arty_all.xdc` file as a reference starting point for the syntax. In the following you can find additional information to help you in writing the code.

PHYSICAL CONSTRAINTS (PORT MAPPING)

Map decoder **inputs** to **slide-switches** `SW0`, `SW1`, `SW2` and `SW3` available on the board:

```
set_property -dict { PACKAGE_PIN A8   IOSTANDARD LVCMOS33 } [get_ports {BCD[0]} ] ;  ## SW0
set_property -dict { PACKAGE_PIN C11  IOSTANDARD LVCMOS33 } [get_ports {BCD[1]} ] ;  ## SW1
set_property -dict { PACKAGE_PIN C10  IOSTANDARD LVCMOS33 } [get_ports {BCD[2]} ] ;  ## SW2
set_property -dict { PACKAGE_PIN A10  IOSTANDARD LVCMOS33 } [get_ports {BCD[3]} ] ;  ## SW3
```

For decoder **outputs** you can use for instance a **Peripheral MODule (PMOD) pin header**, but feel free to choose **ChipKit pins** instead. Please, remind that both **JA** and **JD** pin headers already have a **200 ohm resistor** placed in series, while **JB** and **JC** doesn't.



> **IMPORTANT !**
>
> Remind that at the end you are driving LED devices! Be sure that a **current-limiting series resistor** is always placed on the current path of a LED! If you don't use PMODs that already have resistors in series it's **UP TO YOU** to place approx. 100-200 ohm resistors on 7-segment display input pins!
>
> **WITHOUT LIMITING RESISTORS YOU WILL DESTROY THE 7-SEGMENT DISPLAY MODULE !**

ELECTRICAL CONSTRAINTS

Incorrect voltage supply for the configuration interfaces on board can result in configuration failure or device damage. Vivado has a **Design Rules Check (DRC)** tool that verifies if the configuration interfaces of the device have correct voltage support based on the **Configuration Bank Voltage Select** `CFGBVS` , `CONFIG_VOLTAGE` , and the `CONFIG_MODE` properties settings.

Add these electrical constraints in order to instruct Vivado how the device configuration interfaces are used and connected on board.

```
set_property CFGBVS VCCO        [current_design]
set_property CONFIG_VOLTAGE 3.3 [current_design]
```

Ref. also to:

- *https://www.xilinx.com/support/answers/55660.html*
- *https://forums.xilinx.com/t5/Other-FPGA-Architecture/set-property-CFGBVS-set-property-CONFIG-VOLTAGE/td-p/782750*

TIMING CONSTRAINTS

Our decoder is a pure combinational block, therefore **timing** is not of primary importance. That is, we can assume that after a certain amount of propagation time all outputs settle to **static logic values** by changing the inputs.

Nevertheless for pure combinational timing paths you can define a **maximum delay constraint** between an input and an output using the `set_max_delay` constraint.

As an example, define a max. 10 ns delay between all decoder inputs and all decoder outputs:

```
set_max_delay 10 -from [all_inputs] -to [all_outputs]
```

You can also play with the delay value and verify the effect in Vivado **timing reports**.

Alternatively you can also **disable all timing checks** with `set_false_path` as follows:

```
set_false_path -from [all_inputs] -to [all_outputs]
```

Run the Vivado implementation flow targeting the Arty board

[Contents]

Once ready, **launch Vivado in graphic mode** and try to run the FPGA implementation flow in *Project Mode* up to bitstream generation.

For Linux users:

```
% vivado -mode gui &
```

For Windows users:

```
% echo "exec vivado -mode gui &" | tclsh -norc
```

Create a new project attached to the **xc7a35ticsg324-1L** device. To save time you can skip the *New Project* wizard and use the `create_project` Tcl command as follows:

```
create_project -force -part xc7a35ticsg324-1L SevenSegmentDecoder -verbose
```

Add Verilog and XDC sources to the project and try to run the FPGA implementation flow up to bitsreeam generation.

If you run out of time you can also run the *Project Mode* flow using a Tcl script. Copy from the `.solutions/` directory the `project.tcl` script already prepared for you:

```
% cp .solutions/project.tcl .
```

You can then `source` the script from the Vivado Tcl console:

```
source project.tcl
```

> **QUESTION**
>
> Inspect the **post-synthesis gate-level schematic** and the **post-synthesis utilization report**. Is your circuit a pure combinational block as expected ? What is the output of the `all_latches` command ? In case **latches** have been inferred for you in the design where is the source of the mistake ?
>
> ____

Implement the circuit on breadboard

[Contents]

Plug the 7-segment display module on your breadboard and use **jumper wires** to make all necessary connections between the breadboard and the FPGA according to output pins that you use in XDCs. Use the datasheet to understand the pinout of the module.

Debug your firmware on real hardware

[Contents]

Program the FPGA using the Vivado *Hardware Manager*. Debug the functionality of the firmware on real hardware.

> **QUESTION**

> In case your design contains unwanted latches, which is the effect on real hardware ? Fix the RTL code to solve the problem
> and regenerate the bitstream. Re-program the FPGA and debug the new firmware.
>
> ____

Further readings

[Contents]

* *https://www.electronics-tutorials.ws/blog/7-segment-display-tutorial.html*
* *https://www.electronics-tutorials.ws/counter/7-segment-display.html*
* *https://www.geeksforgeeks.org/seven-segment-displays*
* *https://www.electronicshub.org/seven-segment-displays*
* *https://www.electronicshub.org/bcd-7-segment-led-display-decoder-circuit*