# Lab 2.2
# Custom slave programmable interface

## Introduction

In the previous labs, you used a system integration tool (Qsys) to create a full FPGA-based system comprised of a processor, on-chip memory, a JTAG UART (the serial port used for standard output), and a simple programmable interface, the parallel input/output (PIO) port.

- In lab 2.0 you used a pre-existing *standard* PIO IP component.
- In lab 2.1 you created and used your own *custom* PIO IP component.

Now that you know how to use all the tools, we are going to shift our focus towards creating more specialized IP components.

## Goal

The goal of this lab is for you to design *1* of the following custom slave IP components:

- PWM – *P*ulse *W*idth *M*odulation — ***Easy***
- RTC – *R*eal-*T*ime *C*lock — ***Intermediate***
- WS2812 — ***Intermediate***
- UART – *U*niversal *A*synchronous *R*eceiver *T*ransmitter — ***Hard***

You will then package your custom slave module as an IP component in Qsys and add it to a base Qsys system containing the following components:

- Nios II/e processor
- 128kB on-chip memory
- JTAG UART

The processor will then access and program your IP component as needed to control an external peripheral.



René Beuchat, Sahand Kashani, Florian Depraz

# Theory

## Documentation & Block diagrams

A key part of this Embedded Systems course is learning to be autonomous. This is very important in industry as you will often be asked to implement a certain subsystem all by yourself. However, there come times when you will have to collaborate with others on your specific components, and it is therefore important to have clear *documentation* available such that the other parties can fully understand your ideas.

Three diagrams are mandatory parts of any documentation: the *block diagrams* of the

- Full system (processor, memory, interconnect, peripherals, …)
- Custom IP component
- Top-level connections between your custom IP and the development board's pins

The diagram of the full system is necessarily a high-level one, whereas the diagram of the custom IP component is a low-level one describing how the various components of your IP interact with each other.

We insist that, before you start coding *anything*, you first create these block diagrams. The reason is simple: if you are stuck and have difficulty debugging something in your design, the very first thing the TAs will ask is to see the system and custom component's block diagrams. These schematics need to be available so we can understand the design you had *in your head* and to grasp all the assumptions you have about the design.

We will *not* be able to help you unless you have these block diagrams ready and up-to-date with the code you are debugging (the same is true in industry).

## Clock Dividers

No matter which peripheral you choose to design, you would need to supply it with a slow clock signal to satisfy various a clock that is much design will require a clock frequency that is much lower than the 50 MHz oscillator available by default on the development board.

Students generally all make the mistake of generating a "clock" by using logic inside the FPGA. For example, to generate a clock that runs 3x slower than `clk_in`, they would generate the waveform shown in Figure 1.

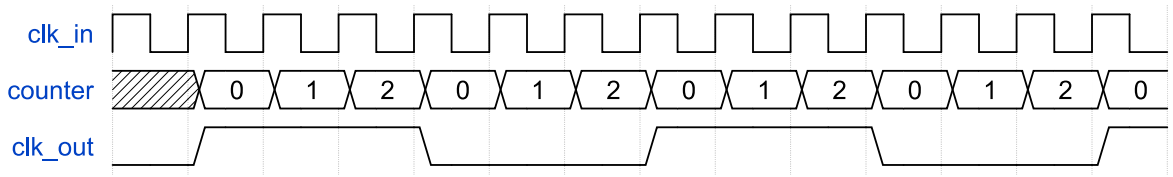


**FIGURE 1.** Incorrect **CLOCK DIVIDER (3X)**

However, this is *not* the correct way to do things on FPGAs. Indeed, clock signals are routed through special channels to guarantee that the clock arrives at all components roughly at the same time (avoids clock skew). If you instead decide to "create" a clock manually by using logic within the FPGA, your clock will not be routed on the dedicated clock channels and will suffer from clock skew, therefore causing all logic driven by the custom clock to be unstable.

The correct way to generate slow clocks in FPGAs is to *not* generate a *clock*, but to instead generate a *clock divider*. A clock divider is a component that periodically generates a *pulse* which is then fed to the slower components. Figure 2 shows an example of a correct divider that divides the clock frequency by 3.

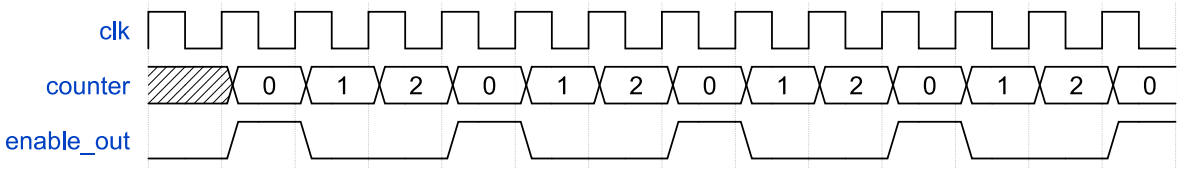


**FIGURE 2.** Correct **CLOCK DIVIDER (3X)**

Each component that requires a slow clock takes as input the standard clock of the FPGA (the one correctly routed through dedicated clock channels) along with the `enable_out` pulse generated by the clock divider. This pulse acts as an activation signal and triggers the operation of a component requiring a slow clock.

For example, you can use the clock divider make a component behave as if it had a "slow" clock by writing its VHDL as shown below:

```
process(clk)
begin
    if rising_edge(clk) then
        if enable_out = '1' then
            -- insert your slow clock logic here
        end if;
    end if;
end process;
```

# Practice

## PWM – Pulse Width Modulation

You must construct an Avalon slave peripheral that is capable of outputting a PWM signal. Your component must support a programmable

- Period
- Duty cycle
- Polarity

Figure 3 shows the relation between the period and the duty cycle.

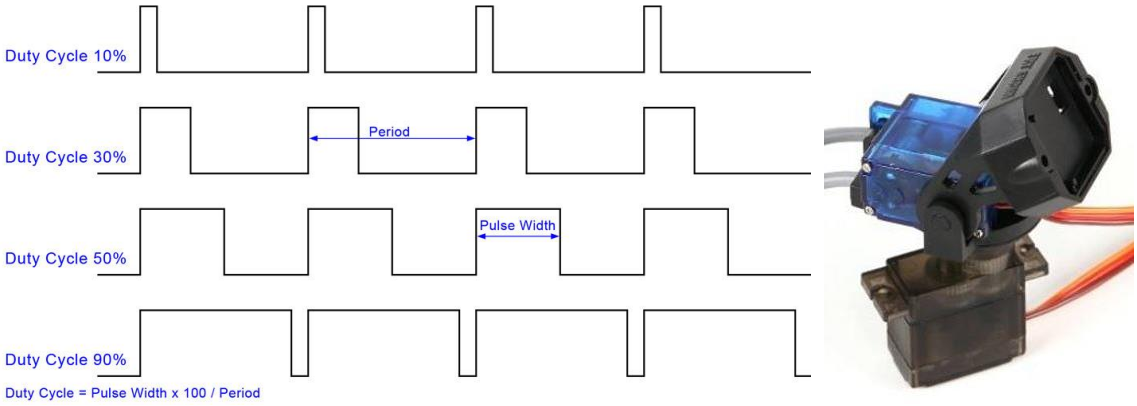


FIGURE 3. PWM WAVEFORMS [1]

You will test your PWM implementation by connecting your IP component a servomotor and programming the period, duty cycle and polarity of the module accordingly.

## RTC – Real-Time Clock

You must construct an Avalon slave peripheral that is capable of outputting an RTC on a 6-digit 7-segment display. The display we use in this lab is shown in Figure 4 and is connected to `GPIO_1` on the DE0-Nano-SoC.

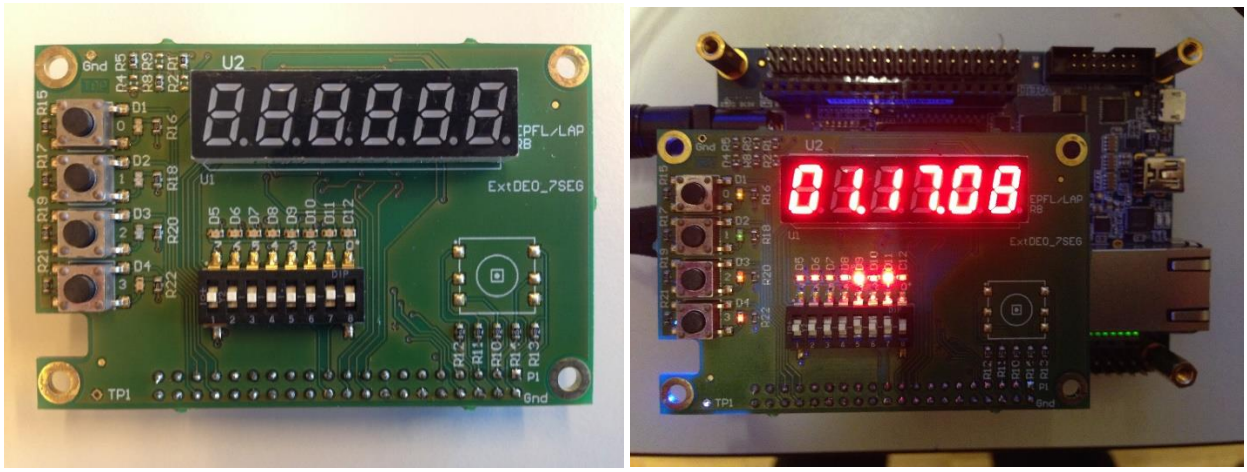


FIGURE 4. 6-DIGIT 7-SEGMENT DISPLAY

Your component must support *1* of the programmable display options below:

- Hours, minutes, seconds.
- Minutes, seconds, 1/100 seconds.

You will need the module's schematic to see how the 7-segment displays are connected to the various pins on the board. The 6 displays are controlled by 3 signals:

- `SelSeg[7 .. 0]` selects the which of the 7 segments (and 1 decimal point) available in each display are to be active. This signal is active-*high*.
- `nSelDig[5 .. 0]` selects which of the 6 displays are to be enabled. This signal is active-*low*.

- `Reset_Led` clears the accumulated charge on all displays. This signal is active-*high*.

The procedure for displaying a digit on one of the displays is as follows:

- Enable 1 display using `nSelDig`.
- Hold a bit pattern on `SelSeg` for some time to allow a charge to accumulate.
- Activate the `Reset_Led` signal to clear all displays.

Note that *only 1 display can be active at a time*, so in order to display a value correctly on the 6 displays, you will need to periodically perform the steps above for each display. The refreshing frequency of the displays needs to be sufficiently low for enough charge to accumulate in each display (to have good brightness) and high enough for the human eye not to see any flickering (due to the periodic charge reset).

A good refresh cycle is 1 ms and a good hold time is 1/6 ms.

You will find the top-level VHDL file and pin assignment script for the extension board in the project template.

## WS2812

You must construct an Avalon slave peripheral that is capable of controlling the WS2812 intelligent LED controller. This controller is used in numerous daisy-chained multi-color LEDs, some of which are shown in Figure 5.


**FIGURE 5. PRODUCTS USING THE WS2812 [2]**

The interface used by this component is too specific to detail in this lab statement, so we refer you to its datasheet (which you will have to read anyways ☺).

# UART

This is a harder exercise than the other peripherals to implement. *Warriors only*, you have been warned!

The UART protocol is a 2-pin serial communication protocol used extensively in embedded systems. For example, it is the protocol used when your Nios II processor calls the `printf()` function and you see the result transmitted back to your host PC's terminal.

A simplified introduction to the UART protocol can be found at Electric Imp.

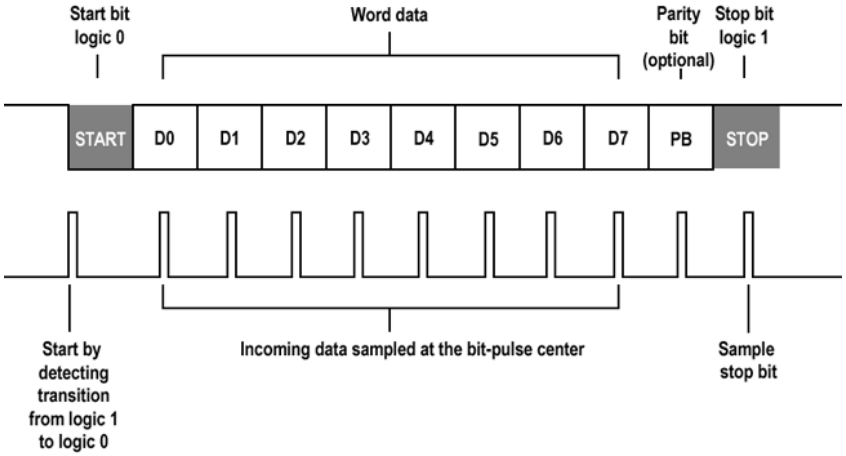


FIGURE 6. EXAMPLE UART SIGNALING [3]

For this exercise you must construct an Avalon slave peripheral that is capable of sending and receiving data through the UART protocol. To test your IP component, you will connect your host PC to your peripheral's RX and TX pins and will create a simple calculator application. Your host PC will send arithmetic expressions (additions only) to the Nios II processor through the UART peripheral and the Nios II will reply back with the answer.

To use a UART console on your host PC, you can use either Teraterm (Windows), Putty (Windows), or minicom (Linux/Mac). Don't forget to configure the terminal for the correct baud rate, parity, … as needed.

Again, *warriors only*!

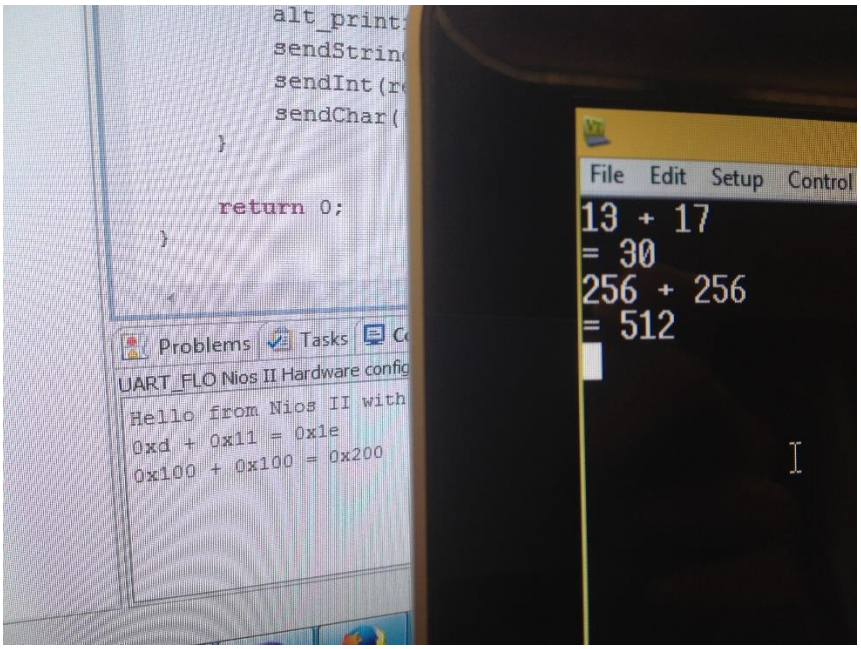


**FIGURE 7. EXAMPLE UART CALCULATOR.**

References

[1] protostack, [Online]. Available: http://d32zx1or0t1x0y.cloudfront.net/2011/06/atmega168a_pwm_02_lrg.jpg.

[2] Adafruit, [Online]. Available: https://cdn-shop.adafruit.com/970x728/1463-03.jpg.

[3] electric imp, [Online]. Available: https://electricimp.com/docs/attachments/images/uart/uart3.png.