Lab 1.0

PWM Control Software

# Goal

The goal of this lab is for you to get acquainted with the basics of writing control software for hardware programmable interfaces (peripherals). The concepts we will see here are building blocks for all future systems software which manipulate hardware directly. Systems software is quite broad: it can range from a simple bare-metal application for an embedded system to a complex linux device driver for a high-speed PCIe interface ☺

The specific topic we will focus on during this lab is the concept of addressing a programmable interface through a bus. The peripheral you will be using is a Pulse Width Modulation (PWM) generator. This peripheral is routinely used to control rotating objects such as robotic arms or airplane flaps. We will use 2 such PWM peripherals for a much simpler goal: moving a pan-tilt module.

This first lab focusses entirely on software: we provide you with a black-box implementation of a system containing a PWM generator. We will also give you a software programming library for the PWM generator with a few empty code segments in the middle. The goal is for you to fill in these code segments after having understood how a peripheral is addressed. At the end of this lab, your code will be able to rotate a pan-tilt module in a nice sweeping motion, just as a wall-mounted security camera does.

# Theory

## System Schematic

When programming any system, hardware or software, the first thing to get your hands on is the system’s overall schematic. Without this information, it would be difficult to know what the system is composed of and how the various subcomponents interact with each other. Figure 1 shows the block-level schematic of our system. As we can see, the system is composed of 5 distinct components:

* Clock & Reset
* Nios II CPU
* On-chip memory
* JTAG UART
* 2 PWM generators

In our system, the CPU is the only master, and the other components (on-chip memory, JTAG UART, and 2 PWM generators) are all slaves. The Clock & Reset component is neither a master, nor a slave, as it is not addressable (no “address” is reserved for accessing this component).

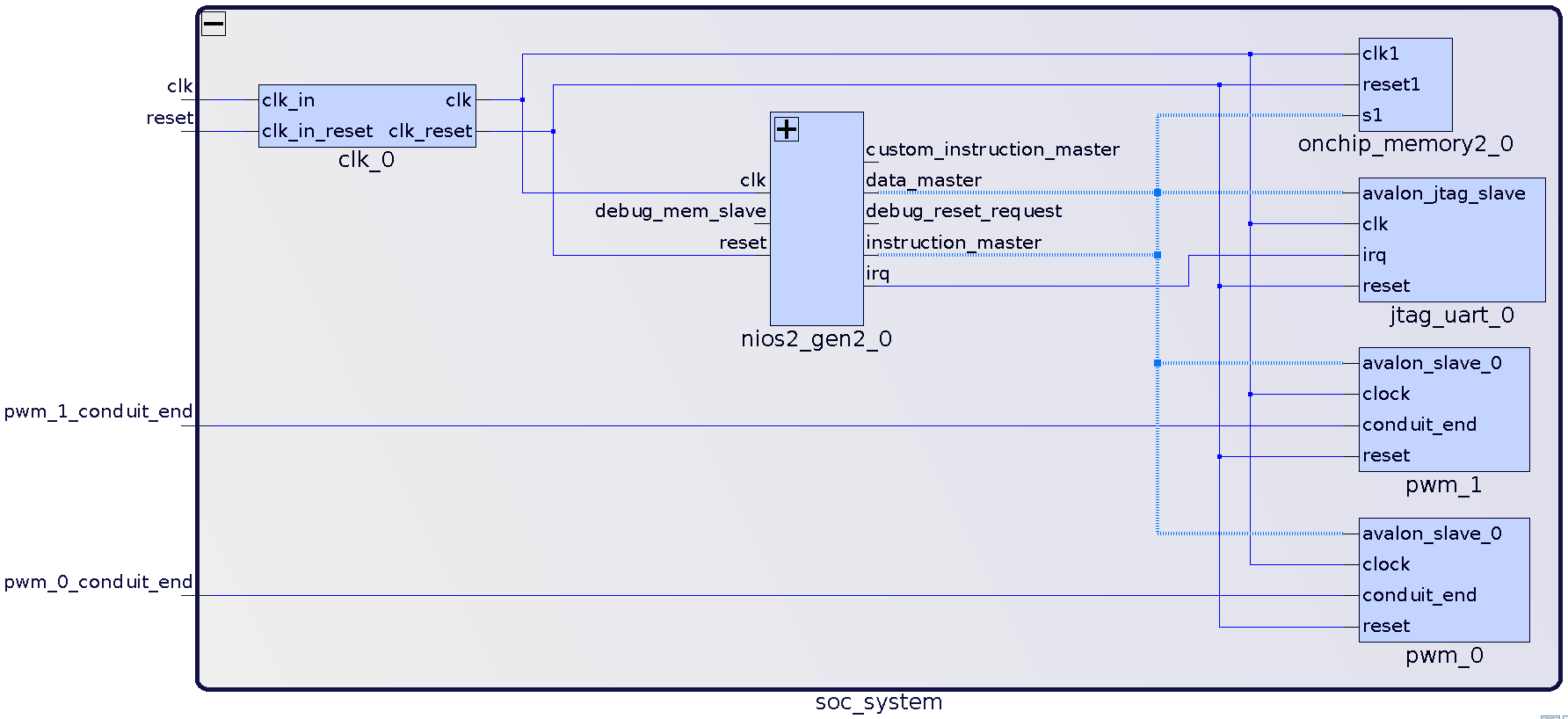


Figure 1. System Schematic

## Programmable Interfaces

Programmable interfaces are circuits that form the basis for specialized functionality in a system. A programmable interface is an instance of a slave which is accessible through a bus by a master, and which the master configures to perform a specific task.

* A master is a device that initiates transactions on a bus.
* A slave is a device that responds to transactions initiated by a master.

When you add components to a system and interconnect them, a bus is generally constructed through which all the components can communicate. Many bus designs exist in industry, and Altera FPGAs use the bus called the Avalon bus. You will learn more about the details of the Avalon bus in the next labs, but what we need to be concerned with at this point is how masters “see” slaves through this bus.

The Avalon bus uses memory-mapped I/O. This means that the same address bus is used to address memory and I/O devices. With this information, we can now say that masters “see” slaves at specific addresses in the master’s address space. For example, suppose that a peripheral is visible at address 0x1000 in a master’s address space, then the master can read/write to the interface by reading/writing at address 0x1000.

Table 1 shows the address map of the system. An address map concisely describes the how the address space of all masters is partitioned among the various programmable interfaces in a system.

|  |  |  |
| --- | --- | --- |
|  | Masters | |
| Slaves | nios2\_gen2\_0.data\_master | nios2\_gen2\_0.instruction\_master |
| onchip\_memory2\_0.s1 | 0x000­2\_0000 – 0x0003\_ffff | 0x000­2\_0000 – 0x0003\_ffff |
| jtag\_uart\_0.avalon\_jtag\_slave | 0x0004\_1020 – 0x0004\_1027 |  |
| pwm\_0.avalon\_slave\_0 | 0x0004\_1010 – 0x0004\_101f |  |
| pwm\_1.avalon\_slave\_0 | 0x0004\_1000 – 0x0004\_100f |  |

Table 1. Address Map

In addition to the system-wide address map, each programmable interface has a device-specific register map. An address map says how different address ranges are reserved for addressing specific peripherals, but it does not say anything about what the different addresses of a specific peripheral mean. For example, the address map in Table 1 shows that PWM\_0 is accessible to the Nios II CPU’s data master port between addresses 0x0004\_1010 – 0x0004\_101f. However, what do each of the 16 bytes (0x…10 – 0x…1f) in this interval specifically mean for this peripheral? The register map of the PWM generator specifies this information. Essentially, the register map is the programming interface exposed to software engineers so they can write code which can correctly use the peripheral.

Note some terminology: generally-speaking, we call the first address in a peripheral’s reserved address range its base address. For example, the base address of PWM\_0 is 0x0004\_1010. This term is used often in various APIs, so remember this word!

Finally, there remains the question of the addressing granularity. What do the addresses in the address map of Table 1 represent? Does each address represent a byte, or a word? It is essential to know this information, as software would not be able to correctly read/write from programmable interfaces without it. The answer to this question is bus-specific, so it isn’t possible to give a universal answer. In the case of the Avalon bus, the bus specification makes a distinction between the point of view from which addresses are used:

* A master uses byte-addressing (each address corresponds to a byte).
* A slave uses word-addressing (each address corresponds to a word, where the size of a word is peripheral-specific). The bus takes care of automatically translating an emitted byte address into a word address before it arrives at a peripheral, so peripherals don’t need to handle this conversion internally (this is not relevant to the current lab, but it will be useful for the next ones where you will be writing the VHDL code of specific portions of various programmable interfaces).

# Practice

Enough said! You now know the theory behind the concept of addressing in order to do what is needed in this lab. You will now program the 2 PWM generators in the system to do something useful ☺

## Launching the Nios II Software Build Tools (SBT)

We are using a Nios II CPU in our design, so we need to use the Eclipse-based "Nios II Software Build Tools" (SBT) to program our CPU.

To launch Nios II SBT, we need to first start a Nios II Command Shell. This shell defines some environment variables that are needed for SBT to work correctly.

* On Windows systems, you can launch the Nios II Command Shell from the Start menu.
* On Linux systems, you can launch the Nios II Command Shell by typing the following command in your shell: “<altera\_install\_dir>/<version>/nios2eds/nios2\_command\_shell.sh”. Remember to replace “<altera\_install\_dir>” and “<version>” according to how the tools were installed on your machine.

Once the command shell is open, enter the following command to launch Nios II SBT:

* eclipse-nios2 &

## Programming the FPGA

It is time to download the hardware design on the FPGA. Here are the steps:

1. Plug your FPGA into your computer.
2. Open the Quartus Programmer from **Nios II > Quartus Prime Programmer...** in the menu bar.
3. Click on the "Auto Detect" button on the left-hand side of the Quartus Programmer.
4. Choose 5CSEMA4.
5. Once you get back in the Quartus Programmer's main window, you will see 2 devices listed in the JTAG scan chain. One of them corresponds to the HPS (ARM CPU), and the other to the FPGA.
6. Right-click on the FPGA entry, and go to **Edit > Change File**.
7. Select the compiled “lab\_1\_0.sof” file in the "hw/quartus/output\_files" directory.
8. Enable the "Program/Configure" checkbox for the FPGA entry, then click on the "Start" button on the left-side menu.

## Creating the software project

Now that the FPGA is programmed with the black-box hardware system, we can create a software project for our design.

1. Go to **File > New > Nios II Application and BSP from Template**.
2. Select *<project\_dir>/hw/quartus/soc\_system.sopcinfo* as the SOPC Information File name.
3. Name your software project "lab\_1\_0".
4. We invite you to uncheck the "Use default location" checkbox and to choose *<project\_dir>/sw/nios/application* instead. We encourage this practice to properly separate software from hardware design files.
5. Choose "Blank Project" as the Project Template.
6. Click **Finish**.
7. Right-click on *app.c* and *pwm.c* in the **Project Explorer** and select **Add to Nios II Build**.
8. You can now write/compile/run your software.

## PWM Control Interface

The provided system includes two PWM generators. Both are 32-bit Avalon Memory-Mapped Slave interfaces. They are mapped in memory at addresses *PWM\_0\_BASE* and *PWM\_1\_BASE,* respectively. These macros can be found in the auto-generated *system.h* header file that you need to include in your C code (already done in the provided files).

The register map of the PWM is shown below:

|  |  |  |  |
| --- | --- | --- | --- |
| **Offset**  **(as viewed**  **by the NIOS)** | **Name** | **Access** | **Description** |
| 0 | PERIOD | R/W | The period of the PWM. The period unit is 50-Mhz clock period. For instance, writing 2 in this register will generate a 25-Mhz PWM signal. |
| 4 | DUTY | R/W | A value between 0 and PERIOD indicating the duty cycle of the PWM. |
| 8 | CTRL | WO | Writing 0 (resp. 1) to the register stops (resp. starts) the PWM. |

To write these registers you will want to use the *IOWR\_32DIRECT(BASE, OFFSET, DATA)* macro available in the *io.h* header file.

## Exercise

Fill in the various /\* TODO \*/ markers in *pwm.c*. This file describes the implementation of the API we provide to software developers who want to use the PWM generator. You can then test your implementation by using the *main()* function in *app.c*. If you did everything correctly, then you should see a periodic top-down, left-right sweeping motion on the pan-tilt peripheral.

Note: an important aspect of this course is one’s ability to be autonomous and persistent when faced with problems. In the embedded world, it is rare to find well-written component datasheets. Most often than one would like, datasheets are written with so much missing information that you would wonder why they are called datasheets. This is where one’s ability to infer a components behavior based on what one observes while debugging is essential. So you need to understand how the whole pipeline works, i.e. how does an instruction emitted by the CPU translate to an abstract transaction on a bus which somehow arrives at a peripheral to be processed. We insist that you read every single line of code which we provide you to see how the system is stitched together. This is especially relevant about the C source files we provide you (less so for the VHDL).

## Connecting the Pan-Tilt

We will use a specially-designed extension board (thanks René!) for the DE0-Nano-SoC throughout this course. It has custom ports for all peripherals we will use in the course to avoid having to manually use wires to connect peripherals to the FPGA (trust us, we did it before, and the extension board is much cleaner).

To keep things interesting, we will not tell you where to exactly plug in the pan-tilt module into the extension board. What we will tell you is that you should have a look at the extension board’s [schematic](https://wiki.epfl.ch/prsoc/documents/PCB_PrSOC_All.PDF) to figure this out (hint: check out page 10 and look for the corresponding pins “Jxx” on the physical board).

Happy hunting ☺.

