

Lab 1.1

PWM Hardware Design

Lab 1.0 – PWM Control Software (recap)

In lab 1.0, you learnt the core concepts needed to understand and interact with simple systems. The key takeaways were the following:

- Hardware components are categorized as *masters* or *slaves*, and communicate through a *bus*.
- Masters *initiate* transactions on the bus, and slaves *respond* to transactions.
- Masters instruct a bus to route transactions towards a specific slave by means of *addressing*.
- Each *master* has an *address map* that indicates where a slave can be found in its address range (of course, only if the slave is connected to the master).
- Each *slave* has a *register map* that indicates what each byte in the slave's addressable space actually corresponds to.

Finally, you put all these concepts together by writing control software for a simple slave programmable interface: a PWM generator.

Lab 1.1 – PWM Hardware Design

Goal

In lab 1.0, we provided you with a black-box implementation of a complete FPGA system containing a CPU, an on-chip memory, a UART, and 2 PWM generators. The goal of lab 1.0 was to tackle one end of the system, i.e. the control *software* for the PWM generators (given their register map).

Now that you have a functional application to control the PWM generators, it is time to tackle the other end of the system, i.e. the *hardware* that makes up the PWM generator itself. Therefore, the goal of this lab is for you to write the VHDL code of the PWM generator.

Note that we are looking at the two endpoints of the system at this stage, but that there is a huge gap in the middle which we have skipped: the bus which interconnects the masters and slaves. For simplicity, let's assume for now that the bus somehow exists, but we do not know how it is created. We will come back to how one creates this bus more in detail in lab 2.1.

Theory

Interconnect motivation

Any system must provide a way to connect components together, otherwise the system wouldn't be able to do much of anything. The question is *how* do we perform this interconnection? In his "fundamental theorem of software engineering", David Wheeler states that "all problems in computer science can be solved by another layer of indirection". Let's see how to apply this theorem to the hardware realm.

On one hand, each hardware component solves a very specific problem with custom hardware, so each hardware component has its own unique set application-specific ports. On the other hand, hardware components must communicate with each other, so their ports must somehow match up. This hardware interconnection problem is solved by separating a hardware component's ports into *application-specific* ports, and *interconnection* ports. The application-specific ports allow each component to do its custom task, whereas the interconnection ports are used to connect various components together through a new layer of indirection: the *bus*.

When hardware components want to communicate, they do not talk directly to each other, but instead talk exclusively to the bus. The bus then takes care of relaying messages from one component to the other according to the *bus protocol*. This decouples all hardware components from each other, so hardware component X doesn't need to know any implementation details of hardware component Y with which it wants to communicate. All X needs to know is that Y shares the same bus interface, so they can communicate through that medium. Essentially, all hardware components conceptually look as in Figure 1.

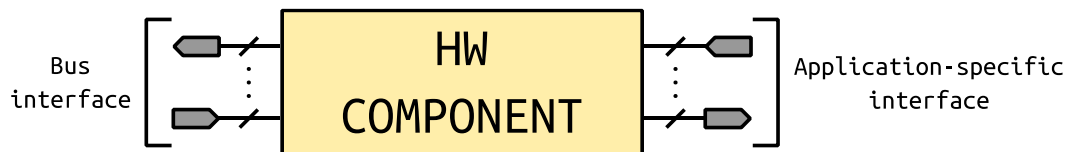


FIGURE 1. ABSTRACT HARDWARE COMPONENT

Now that we know a bus is needed, let's look into some desirable properties. An interconnect should ideally have high throughput and low latency in order to minimize communication overheads and maximize computation time. An easy solution for FPGAs would be to use the same high-performance interconnects that are present in commodity systems, and, as a matter of fact, this is the case in some FPGAs today (Xilinx FPGAs use the *AXI4* bus: a high-performance interconnect designed by ARM for use with their processors). However, such high-performance interconnects come at a large area cost. This may be justifiable today as FPGAs have become huge and can easily absorb the area cost of these interconnects, however, it is easy to imagine this was not the case in the early days of FPGAs when the devices did not have many resources available.

Altera solved this problem back then by introducing the *Avalon bus*: a simple and small bus targeted specifically at their FPGA product line. Despite its simplicity, the bus is able to perform I/O at around 300 MB/s, which is not bad at all considering its simplicity.

Avalon bus

Introduction

The Avalon bus specification supports 7 different types of interfaces, but in this course we will only be interested in 4 of them:

- Avalon *Clock* interface
- Avalon *Reset* interface
- Avalon *Conduit* interface
- Avalon *Memory-Mapped* (MM) interface

The clock and reset interfaces literally describe what they correspond to, so they don't need any explanations. The interesting interfaces are the conduit and memory-mapped interfaces.

- An Avalon *conduit* interface groups an *arbitrary* collection of signals. You can specify any role for conduit signals. However, when you connect conduits, the roles and widths must match and the directions must be opposite. An Avalon Conduit interface can include input, output, and bidirectional signals. Conduit interfaces typically used to drive off-chip device signals (e.g. a PWM generator's output). There is nothing much more that can be said about conduit interfaces.
- An Avalon *memory-mapped* interface specifies a *standard* collection of signals that can be used to implement a read/write interface between masters and slaves. The interface can be customized to be write-only or read-only if needed by omitting the corresponding signals. The minimum set of ports needed for a slave to use a read/write Avalon-MM interface are as follows:
 - address
 - read
 - write
 - readdata
 - writedata

The width of the readdata and writedata ports specify the component's *word size* (*very important!*), and both ports must have the same width. Finally, note that all signals in an Avalon-MM interface are synchronous to the hardware component's clock interface.

To summarize, a generic hardware component with an Avalon-MM *slave* interface would look like Figure 2.

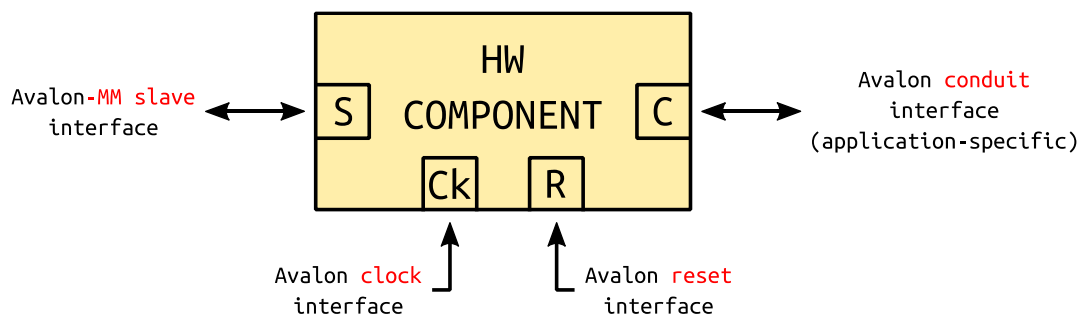


FIGURE 2. AVALON HARDWARE COMPONENT

Avalon-MM slave example

All this talk is nice, but we need an example to seal the deal ☺. Figure 3 shows the interface schematic of an **adder** implemented as an Avalon-MM slave unit. Note that this example is excessively simple and is here for educational purposes only: the design is actually quite inefficient as the area used for the bus logic is larger than the actual application logic (just an adder). **BUT**, it is compatible with the Avalon-MM interface!

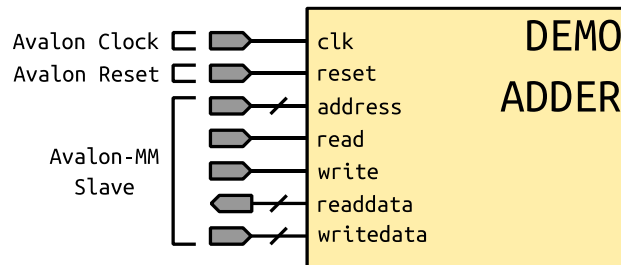


FIGURE 3. DEMO ADDER

The complete VHDL code used to implement this Avalon-MM slave unit is shown in Figure 4. Please take the time to read it in detail and understand **every line**. You will need to understand how the bus works in order to implement your own Avalon-MM slave PWM generator later on for this lab.

```

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

entity demo_adder is
  port(
    -- Avalon Clock interface
    clk : in std_logic;

    -- Avalon Reset interface
    reset : in std_logic;

    -- Avalon-MM Slave interface
    address : in std_logic_vector(1 downto 0);
    read : in std_logic;
    write : in std_logic;
    readdata : out std_logic_vector(31 downto 0);
    writedata : in std_logic_vector(31 downto 0)
  );
end demo_adder;

architecture rtl of demo_adder is

  -- Register map
  -- +-----+-----+-----+-----+
  -- | RegNo | Name      | Access | Description |
  -- +-----+-----+-----+-----+
  -- | 0      | INPUT_1   | R/W     | First input of the addition. |
  -- +-----+-----+-----+-----+
  -- | 1      | INPUT_2   | R/W     | Second input of the addition. |
  -- +-----+-----+-----+-----+
  -- | 2      | RESULT    | RO      | Result of the addition. Writing to |
  -- |         |           |         | this register has no effect. |
  -- +-----+-----+-----+-----+
  constant REG_INPUT_1_OFST : natural := 0;
  constant REG_INPUT_2_OFST : natural := 1;
  constant REG_RESULT_OFST : natural := 2;

  signal reg_input_1 : unsigned(writedata'range);

```

```

    signal reg_input_2 : unsigned(writedata'range);
begin
    -- Avalon-MM slave write
    process(clk, reset)
    begin
        if reset = '1' then
            reg_input_1 <= (others => '0');
            reg_input_2 <= (others => '0');
        elsif rising_edge(clk) then
            if write = '1' then
                case to_integer(unsigned(address)) is
                    when REG_INPUT_1_OFST =>
                        reg_input_1 <= unsigned(writedata);

                    when REG_INPUT_2_OFST =>
                        reg_input_2 <= unsigned(writedata);

                    -- RESULT register is read-only
                    when REG_RESULT_OFST => null;

                    -- Remaining addresses in register map are unused.
                    when others => null;
                end case;
            end if;
        end if;
    end process;

    -- Avalon-MM slave read
    process(clk, reset)
    begin
        if rising_edge(clk) then
            if read = '1' then
                case to_integer(unsigned(address)) is
                    when REG_INPUT_1_OFST =>
                        readdata <= std_logic_vector(reg_input_1);

                    when REG_INPUT_2_OFST =>
                        readdata <= std_logic_vector(reg_input_2);

                    when REG_RESULT_OFST =>
                        readdata <= std_logic_vector(reg_input_1 + reg_input_2);

                    -- Remaining addresses in register map are unmapped => return 0.
                    when others =>
                        readdata <= (others => '0');
                end case;
            end if;
        end if;
    end process;
end architecture rtl;

```

FIGURE 4. DEMO_ADDER.VHD

Practice

Enough reading, more doing! It's time for you to work! The goal of this lab is for you to write the VHDL code for the PWM generator we had provided you in lab 1.0 as a black box. The box will just be a lot more transparent this time as you'll be implementing it ☺.

Before continuing, you should download and extract the lab template ([lab 1 1 template.zip](#)) from the course website. **REMEMBER THAT THERE MUST BE NO SPACES IN THE PATH LEADING TO THE PROJECT! WE WILL REMIND YOU THIS UNTIL YOU MEMORIZE IT.**

PWM generator interface schematic

Figure 5 shows the interface schematic of the unit you need to develop. Note that, unlike the demo adder shown in Figure 3, your PWM unit contains an additional Avalon conduit interface as the PWM output signal of your slave is an application-specific signal.

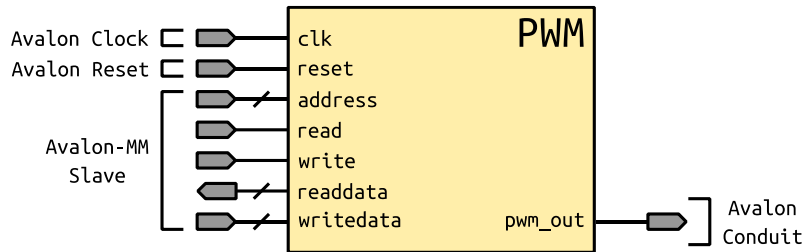


FIGURE 5. PWM GENERATOR

PWM generator register map

Table 1 shows the PWM unit's register map (identical to the one shown in lab 1.0). Pay close attention to the semantics of the period and duty cycle registers, specifically regarding the **current** and **new** period values.

Byte offset (from base)	Name	Access	Description
0	PERIOD	RW	<p>Period in clock cycles ($2 \leq \text{period} \leq 2^{32} - 1$).</p> <p>This value can be read/written while the unit is in the middle of an ongoing PWM pulse. To allow safe behaviour, one cannot modify the period of an ongoing pulse, so we adopt the following semantics for this register:</p> <ul style="list-style-type: none"> Writing a value in this register indicates the new period to apply to the next pulse. Reading a value from this register indicates the current period of the ongoing pulse.
4	DUTY_CYCLE	RW	<p>Duty cycle of the PWM ($1 \leq \text{duty cycle} \leq \text{period}$)</p> <p>This value can be read/written while the unit is in the middle of an ongoing PWM pulse. To allow safe behaviour, one cannot modify the duty cycle of an ongoing pulse, so we adopt the following semantics for this register:</p> <ul style="list-style-type: none"> Writing a value in this register indicates the new duty cycle to apply to the next pulse. Reading a value from this register indicates the current duty cycle of the ongoing pulse.
8	CTRL	WO	<ul style="list-style-type: none"> Writing 0 to this register stops the PWM once the ongoing pulse has ended. Writing 1 to this register starts the PWM. Reading this register always returns 0.

TABLE 1. PWM REGISTER MAP

Draw PWM generator schematic on paper

Every RTL design you write during this course should be drawn on paper *before* you touch the VHDL code as it helps sharpen your digital design skills and greatly helps when debugging design errors (design errors are difficult to spot in VHDL, but easy on a diagram). In any case, you will have to include the RTL diagrams in the reports you write, so you will eventually have to do it (so better do it now ☺).

Implement PWM generator schematic in VHDL

Once your RTL diagram is ready, you can proceed to implement it in VHDL. The VHDL file you need to complete is “hw/hdl/pantilt/hdl/pwm.vhd”. Some constants are defined in “hw/hdl/pantilt/hdl/pwm_constants.vhd” for your convenience (i.e. use them!).

Use a text editor to edit the file and fill in the implementation of a PWM generator with an Avalon-MM slave interface that satisfies the register map shown in Table 1. You’ll need to apply what you learnt in the previous sections, especially the sample VHDL code for the demo adder.

Compiling the hardware design

You must now compile your VHDL to see if it is synthesizable.

- Launch Quartus Prime then go to **File > Open Project...** and open file “hw/quartus/lab_1_1.qpf”. At this point, you should see the following files in the Project Navigator. Pay attention to the fact that the VHDL files of your PWM unit are *not* visible in the Project Navigator, but are rather hidden somewhere within the “soc_system.qsys” file (we will see why in lab 2.1, you can ignore this for now).

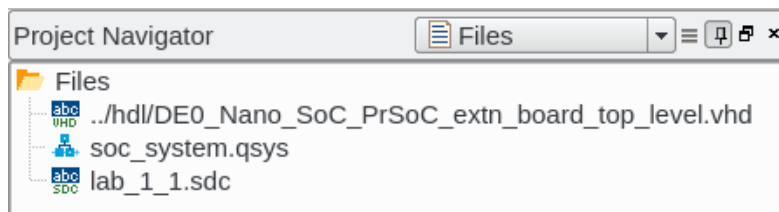



FIGURE 6. QUARTUS PROJECT NAVIGATOR

- Compile the project by going to **Processing > Start Compilation**, or by pressing the  button. If you receive errors about your design while compiling, you can double-click on the log entry in the messages view to open the file at the position of the error. **ATTENTION:** for reasons we will see in lab 2.1, the VHDL files of your PWM generator are *copied* to a temporary directory inside your Quartus project’s working directory before being compiled. As such, the errors flagged by Quartus will point to the *copied* file, not to the original. Be sure to modify your *original* VHDL file each time you fix an error, otherwise you won’t be correcting anything as Quartus recopies your original over the copy each time it compiles! This leads to much frustration if you are not aware of this “feature”.

Testing the PWM generator with a testbench

If all went well and no VHDL errors were reported during synthesis, you must now check if the PWM unit adheres to the specification of its register map. There are 2 ways to do this:

- You are a *lazy* engineer who feels super lucky, so you proceed to “try” your luck by running your control software from lab 1.0 directly on your design and seeing if the servomotors work as expected.
- You are a *hardcore* engineer and decide to simulate your circuit with a testbench to see if it respects the specifications of its register map.

For *obvious* reasons, we are going with the second option (if the reasons are not obvious, please ask the course staff so we can guide you back to the light ☺). For this first VHDL lab of the course, we provide you with a short (non-exhaustive) testbench which checks the functionality of your PWM generator. You can find the testbench in “hw/hdl/pantilt/tb/tb_pwm.vhd” and can run it in ModelSim.

If you are unfamiliar with writing testbenches in VHDL or using ModelSim, we highly recommend you read the [VHDL Testbench Tutorial](#) available on the course website prior to continuing. Indeed, an engineer’s job does not end after having found a “solution” to a problem, but he/she must be able to demonstrate, to various degrees of certitude, that the solution is *correct*. This is especially valid in the hardware industry as components can generally not be fixed once delivered to customers.

Furthermore, we will not be providing testbenches for the various components in the future labs, so you will have to write them yourselves if you need them, so it is essential you grasp the concept early to ease your life later.

Programming the FPGA

If all is good and no assertion failures are reported in the testbench, you can finally proceed to program your FPGA with your custom design.

1. Plug your FPGA to your computer with a USB Blaster cable.
2. Open the Quartus Programmer.
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_1.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

The FPGA is now programmed with your custom design. You can now create a software project for your design. The software is intended to run on the Nios II CPU.

1. Copy the source file you completed in lab 1.0 (“lab_1_0/sw/nios/application/pwm/pwm.c”) to the same location in lab 1.1. We will use this to test if your implementation of the PWM generator functions as expected with the same application code as in lab 1.0.

2. Launch the Nios II Software Build Tools.
3. Go to **File > New > Nios II Application and BSP from Template**.
4. Select “hw/quartus/soc_system.sopcinfo” as the SOPC Information File name.
5. Name your software project “lab_1_1”.
6. We invite you to uncheck the "Use default location" checkbox and to choose “sw/nios/application” instead. We encourage this practice to properly separate software from hardware design files.
7. Choose "Blank Project" as the Project Template.
8. Click **Finish**.
9. Right-click on app.c and pwm.c in the **Project Explorer** and select **Add to Nios II Build**.
10. You can now run your software and see if the PWM unit behaves as expected.

