Oblig 7

Diagrams and implementation

IN3160/IN4160

Version 2023-12-21

In this exercise, you will create diagrams, testbench and VHDL code for a pulse density module.

The key goal for this exercise is to get practice in creating diagrams and understanding their limitations and use in digital design. Creating VHDL and testbench is a secondary objective. Making a functional pulse density module may help in the next assignment, where the pulse density module can be used as a component or with some modifications.

This exercise is also intended to aid understanding of control systems, relevant for the study program in robotics and intelligent systems.

Approval:

In this exercise, you shall deliver diagrams according to the task text below. Diagrams shall be drawn in a capable digital tool[1], and shall be uploaded together as a single pdf file.

The python and VHDL code in task d) and e) can be delivered as two separate (.py and .vhd) files or together as a single .zip file.

Requirements:

- A pdf containing
  - the diagrams as specified in a)- c)
  - waveform from simulation of d) and e)
- Testbench code as specified in d)*
  - The testbench should compile and run (past reset and then some…).
    - A good test will help you make sure your code can be used later.
- VHDL code as implemented in e)*
  - The code should compile and be synthesizable
  - Having the VHDL pass all tests is not required in this assignment.
    - Assignment 8 *can* be solved without this code, getting this right will make it easier.
  - * If your code does not work as intended:
    - Make comments and ask two questions for the supervisors (*traversing all your code should not be necessary to aid you*)

Functioning PDM modules can be reused in assignment 8, which requires functional PWM (either ordinary PWM or PDM see text below).

[1] We recommend using https://www.drawio.com or https://app.diagrams.net (= the same tool, available for download or online use). This tool has been readied for exam usage in Silurveien (late 2023), hence we expect to use it for coming exams (2024-)

Background and motivation

Pulse density modulation

Power transistors, used in all sorts of motors and appliances of any kind, draw less power when turned fully on or off, rather than modulating analog signals. To mimic analog current values, we can rapidly switch the transistors on or off to create an average current equal to that of a steady current. The most common way to do this is to use pulse width modulation where the modulation frequency is fixed, and the pulse width is calculated once per period. In Figure 1 (below), pulse width modulation using a modulation period of T= 5 ms (f=200Hz) is shown as the top signal.

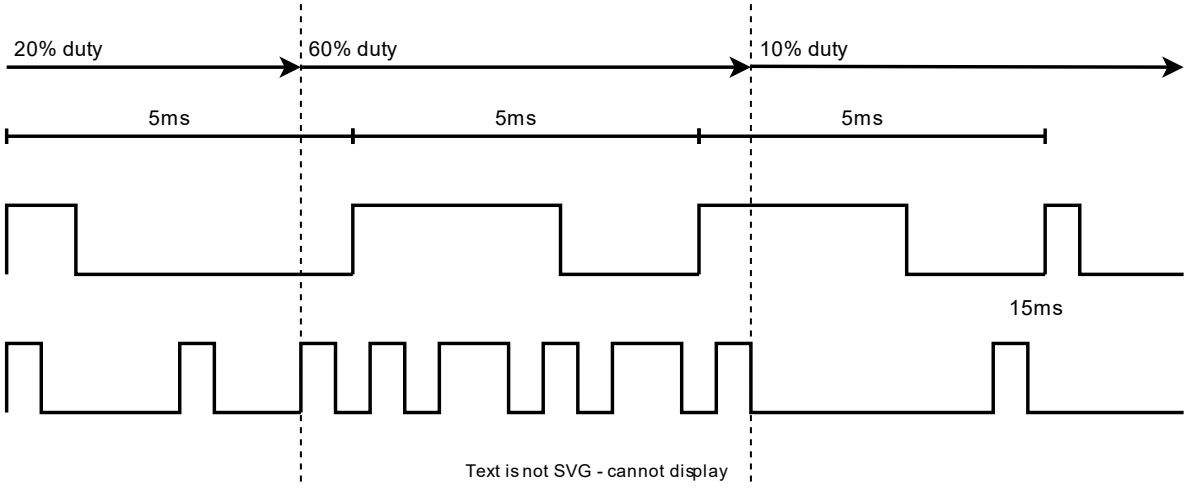

Figure 1: PWM-Pulse width modulation(top) and PDM - Pulse density modulation (bottom)

Pulse density modulation is a variant of pulse width modulation, where there are less restrictions on the modulation period. Rather than varying the pulse width at regular intervals, we calculate whether to create a minimum- duration pulse or not at each timestep. The signal at the bottom of Figure 1 (above), shows an example of pulse density modulation, compared to regular pulse width modulation, both using the same setpoint for duty cycle.

Each of these variants have their own pro's and con's when used in a control system. While the regularity of ordinary pulse width modulation is often preferred when used in combination with current measurements, pulse density modulation usually can respond faster and will provide more even currents using the same minimum pulse width.

Practical considerations for transistors

Using real world components that have capacitance (filling up charge /electrons takes time), considerations must be made to avoid modulation that is useless or cause shorts. The switching times of transistors vary with size and technology in use. If a pulse duration is shorter than the time a transistor needs to switch, the net effect will be little or virtually none to the system it is used. In low voltage digital electronics, switching times are in the ballpark of picoseconds, while higher power, higher voltage transistors switching may be in the range of nano- or microseconds.

Without going into detail for specific components, this means that any modulation *should* be made with some thought for timing considerations that apply for the components in use. That consideration *could* be modulating at a low enough frequency that we avoid considering switching issues, *or* to make a system that is fast and has the flexibility to mitigate enough of these issues. For this task, we attempt to do the latter in a reasonable manner.

Creating a pulse density module

a) Datapath diagram

Creating pulse density modulation without timing considerations can be done using an accumulator-register and adding the setpoint each clock cycle.

In the task that follows, you will build on this principle.

```
signal setpoint : unsigned(WIDTH-1 downto 0);       -- setpoint = duty cycle
signal r_acc, next_acc  : unsigned(WIDTH downto 0); -- acc=accumulator, +1bit
alias PDM_out : std_logic is acc(acc'left);         -- leftmost bit = "carry"
...
r_acc <= next_acc when rising_edge(clk);
next_acc <= ("0" & unsigned(setpoint)) + ("0" & r_acc(WIDTH-1 downto 0));
-- note that carry from the register is not used in calculation
```

Listing 1. Simple PDM VHDL implementation- A starting point for this exercise.

Create a datapath diagram for the VHDL implementation og PDM modulation shown in Listing 1 (above). Make sure to annotate all signals and their vector sizes.

b) Block diagram

```
entity pdm is
  generic( WIDTH: natural := 16 );
  port(
    clk,
    reset      : in std_logic;
    setpoint,
    min_off,
    min_on,
    max_on     : in std_logic_vector(WIDTH-1 downto 0);
    mea_req    : in std_logic;
    mea_ack,
    pdm_pulse  : out std_logic
  );
end entity pdm;
```

Listing 2. Pulse density module entity.

The pulse density module for this task shall use the entity shown in Listing 2 (above). All vectors shall be interpreted as natural numbers (*no sign bit*).

Create a block diagram showing how the pdm module connects to surrounding modules.

The surrounding modules are as follows: A control module, a measurement module, and a driver module. The connection between the surrounding modules is irrelevant to this task, only the connections to the pdm module shall be shown.

The control module shall supply the `setpoint` used for modulation, along with the `min_off`, `min_on` and `max_on` signals used to govern the timing of the pulse modulation. The measurement module supplies the `mea_req` signal. The pdm module supplies `mea_ack` flag to the measurement and the control module and `pdm_pulse` to the driver module.

The signals `clk` and `reset` do not need to be shown in the block diagram.

c) ASMD Diagram

The pdm module shall have a state machine that controls the `pdm_pulse` and the `mea_ack` signals.

The pdm module shall use two counters, one used as a timer, and one used to keep track of the pulse duration. The timer shall be used to keep track of the minimum time for the pulse being deactivated (`min_off`) and the maximum time for the pulse being active (`max_on`).

The counter shall count each cycle the most significant bit is set in the accumulator circuit (as described in Listing 1) is high, except when the pulse is active. When the pulse is active, the counter shall count down except when the most significant bit in the accumulator circuit is high. The counter shall never count past the maximum value (all bits = '1'), or below zero.

Similarly, the timer counts down each clock cycle from the set value until it reaches zero. At zero the timer shall stop counting down. The timer values shall be set to `max_on` when the pulse goes high, and `min_on` when the pulse is turned off.

Whenever the measurement system requests time for doing measurement, `mea_req` is asserted (high). The pdm module shall then respond by asserting `mea_ack` after waiting for the current pulse to finish its duration. The acknowledge signal shall remain high until the request signal is de-asserted. During the measurement, while the acknowledge flag is asserted, the pulse shall not be asserted.

When the `pdm_pulse` is low and the timer has reached zero after counting down from the minimum off time, the pulse can be asserted once the counter has reached the `min_on` value. When the pulse is high, the pulse shall be de-asserted when either the counter or timer has reached zero.

The recommended size for the state machine controlling this is three states (mealy machine), but other configurations are possible.

Create an ASMD diagram for the state machine.

Maintaining simplicity in ASMD diagrams.

Generally, when a part has a distinct function (typically data path operations) that is either the same for each state, or there are only a small portion that changes with state usage, it does not make sense to put the whole function in every state of a diagram, as it would require multiple duplications that clutter the diagram. However, functionality that does change according to state should be displayed in the state diagram. When you have a mix of those cases, it may be beneficial to add a list of default statements on the side of the diagram, rather than adding that functionality to almost every state. Then exceptions can be shown in the diagram.

For this module, the counter and the timer mostly follow the same rules regardless of state, although their signals are used for decisions in the state machine. The timer will for example always count down, except when a new value is loaded into the counter. In stead of saying "timer <- timer -1" in each state, we put that statement in a list of default statement on the side of the diagram and show when the timer is loaded properly placed in the diagram. That way the diagram is easier to work with and we maintain simplicity and readability, and thus verifiability of the diagram.

d) Python testbench

Create a Cocotb-python testbench that can be used for testing the pdm-module. After reset and setting initial values, the testbench shall always check the following features:

- That pulses never are wider than max_on + 1 (see example in Listing 3, below)
- That the pulse never is off is shorter than min_on
- That mea_ack is never asserted during a pulse
- That mea_ack is asserted within two clock cycles after mea_req is asserted when the pulse is low
- That mea_ack is de-asserted within two clock cycles after mea_req is de-asserted
- That the duty cycle is within reasonable limits (about 10%*)
  (measured from one de-assertion to the next de-assertion of the pulse)

For testing, the recommended number for min_on is 5 cycles, the min_off is 10 cycles, and max_on is 200 cycles or greater. The mea_req should be asserted at 5 random times, at least 400 cycles apart. Mea_req can be lowered 5 cycles after mea_ack is asserted. The testbench should check at least 50 random setpoints, and at least 10 of these should be tested for 3 periods or more.

Using these values, the duty cycle (should be well within 10% of the set value (where all '1's is 100%, 0 is zero %)

When should I write testbenches?

Writing testbenches, or creating a test specification, before having something to test can be a mental challenge. However, sometimes we can use this challenge to avoid falling into the traps of poor test design. One example of a trap is making a check specific point in time while the error occurs at a different time.

Quite often we can understand what we can test right after we have specified what we shall make (ie we have an idea of an entity, its connections and how it should work- possibly formulated in state- or data path diagrams. Thus, we can create checks that should be valid throughout the simulation.

It is generally a good idea to make checks that can run throughout simulation. In this task, most checks, if not all can be made as static checks in the context that they can be on from the time where the circuit is reset and until the end. Even checks on duty-cycle can be written standalone, using the duty cycle given to the dut as input.

Often it is beneficial to make testbenches in parallel with the module, in order to always test new features added, and also test that the testbench code run before it gets big and messy.

```python
from cocotb.triggers import RisingEdge, FallingEdge
from cocotb.utils import get_sim_time
CLOCK_PERIOD_NS = 10
#...
async def max_on_check(dut):
    while True:
        await RisingEdge(dut.pdm_pulse)
        start = get_sim_time('ns')
        await FallingEdge(dut.pdm_pulse)
        end = get_sim_time('ns')
        duration = end-start
        cycles = duration/CLOCK_PERIOD_NS
        assert cycles <= int(dut.max_on.value)+1, (
            f"""Pulse of {cycles} cycles greater than
            max_on: {int(dut.max_on.value)}""")
#...
cocotb.start_soon(max_on_check(dut))
```

Listing 3: Maximum pulse-width check that can run throughout an entire simulation.
Tests done independent of-, or triggered by stimuli, can often run throughout simulation.

e) VHDL implementation and test

Implement the pdm module in synthesizable VHDL and test the module using your own testbench. Test the code using your own testbench from d). The waveform created during the test should be added to the pdf file with diagrams. Comment on unexpected behavior, or clarifications you would need to finish this task. VHDL and testbench implementation is not critical for this assignment, but understanding how to get it right will be crucial for the next.

Note: Use one process for setting all registers rather than multiple one-liner-assignments. Registry reset should be synchronized to the clock signal. All registers should be set to their respective `next_<signal name>`.

Hint: Checking for maximum values (all '1') is best done using **`and`** `<signal>`

Supervision guide (for supervisors)

Checklist for approval:

Generally: Diagrams that correspond to the task. Code that compiles. If code does not work as specified, try to answer the two questions asked by the students.)

- Datapath diagram
    - Should correspond to the code in a).
        - Comment if there are parts that are unclear.
- Block diagram
    - Should match the description in the task.
        - Comment if there are minor mismatches.
- ASM diagram
    - Does the diagram comply with ASM rules?
        - *This is a requirement to pass*. Comment if it does not.
    - Does the diagram implement pdm according to the task specification?
        - This is a requirement to pass. Comment if it does not.
        - If yes, comment if there are easy ways to simplify the implementation or diagram.
- Testbench and VHDL module
    - Does the waveform look reasonable?
        - If not make a brief comment on what is missing
            - Stimuli or functionality?
    - Comment on questions asked.
    - Check that the code is handed in.

Templates are not considered a delivery = hard fail.

Getting the ASM diagram right is the main requirement for this task.

Getting code right is required in assignment 8, not here.

Being set on the right path to be able to solve the next assignment is the main goal for commenting code. *Do not spend excessive time commenting code in this task.*

Comments can be provided orally for students that are able to present their work/ and questions in lab. Please note “oral feedback is given” in canvas when doing so, to make it possible to make sure no one gets forgotten.