Oblig 10: System on chip integration –

Combining a software PID controller with PWM in FPGA fabric.

IN3160/IN4160

The main goal for this assignment is to learn and perform the basic steps needed to implement a system-on-chip (SoC) consisting of both hardware modules written in an HDL and a processing system running software written using C code. As a side note, we will also present the basic function of a PID control system.

The HDL system that shall be coupled with the processing system is the control system written in oblig 8. The HDL code will only undergo minor adjustments to provide the necessary IO to be connected with the IP-core that can be accessed by the processing system. To connect the HDL system to the processor bus, we will use an IP that handles all the bus protocols. The C code that will run on the processor in this assignment is pre-programmed. The processing system will provide PID regulated values based on the output of the HDL module.

Making a full system on chip can be a daunting task the first time it is performed. Several process steps will have to be performed in order to create the necessary environment that can be used to create nearly any combination of software and hardware. To aid this process we will use four tutorials that will take us through all the necessary steps to build our system. We will follow Xilinx own tutorials, modified to function with our setup and infrastructure.

Before we dive into creating our SoC, we shall have a brief look at what we have, and what we are about to create.

PID control

In Oblig 8, we introduced a general control system as shown in the figure below.

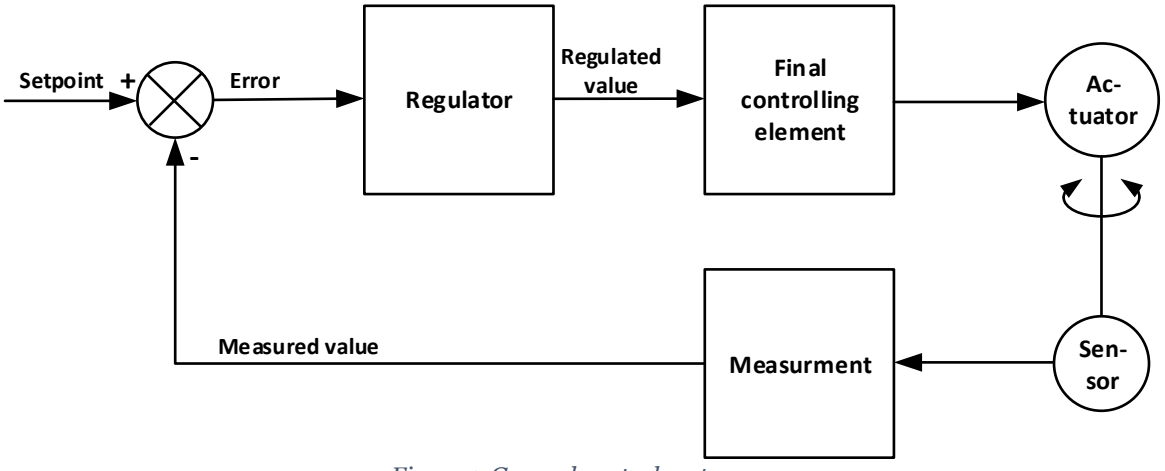

Figure 1. General control system.

In that exercise, you made the final controlling element, the pulse-width module, and used it to set the output directly. In this exercise, the regulated value shall be provided by a PID module provided as C code that can run on the Zynq processor once the system is properly set up and programmed.

A PID controller uses the current error along with the error history to provide a regulated value. The steps involved in generating PID control is relatively simple: On a regular basis, you provide the system with updated error values. The current error is multiplied by a constant (k_P) to form the proportional (P) part of the PID control signal. To form the integral part of the control signal (I), you take the previous sum of errors, add the current error, then multiply that sum with a constant k_I. The derivative (D) part is found by taking the current error and subtracting the previous error and multiplying the result with a constant k_D. At last, you add all three values (P+I+D) to form the PID output value.

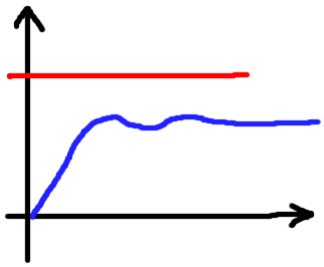

Figure 2. P-alone

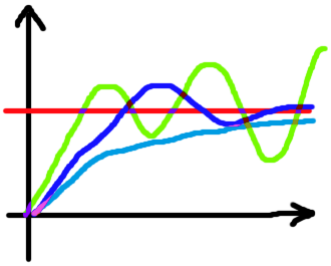

Figure 3. P + I variants

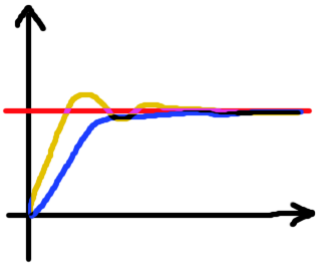

Figure 4. Tuned PID

The reason for using the integral I, is that P alone will not bring you to the setpoint. A system that is put under some sort of load (e.g. a motor will have to use energy to maintain speed) will never be able to reach the setpoint using P alone, since 0 error will give 0 output (e.g. a motor will slow down until the error gets big enough that the power is big enough to maintain speed). The integral part will continue to rise as long as our error is positive, and it will shrink when the error is negative, thus it may bring the error to 0.

In some cases, having P and I will be sufficient. However, a system using P and I, may also be either too slow in reaching the setpoint, or generating overshoot that makes the system unstable. To allow for fast control and dampen overshoot, the derivative part is introduced, as a brake when the error is changing too quickly.

```
// error calculations, all scaled up by SETPOINT_SCALE
previousError = currentError;
velocity      = readVelocity(reverseDirection);
currentError  = mySetPoint - velocity;
sumError      = sumError + currentError;
deltaError    = currentError - previousError;

// Calculate P,I,D
P = myKp*currentError;
I = myKi*sumError;
D = myKd*deltaError;

// Calculate and scale PID output
PID = (P+I+D)/VELOCITY_TO_SETPOINT_SCALE;
```

Figure 5. PID calculation in c code.

The PID calculation routine for this exercise is given in the text box above (except for declarations and truncation to 8 bit signed values), and it will be run with a constant time interval. The software will, after initial testing, compensate for direction being wrong, since this can be induced by a number of factors, such as reversed readout, reversed motor polarity, etc. (hence the reverseDirection identifier). The scaling (VELOCITY_TO_SETPOINT_SCALE) is used for allowing calculations to be done using integers without sacrificing too much fidelity.

Once you get the system up and running, you can study how different PID settings will work on this system. Note that finding the "best" constants for the PID routine can be challenging. What is best having one specific workload may be much worse in another.

There are many ways of tuning PID control systems. The preset values in this exercise has been hand tuned, and may cause values outside our predefined range of +-127 particularly when the load is high.

FPGA system design

An overview of the system to be implemented is shown in Figure 6. This is further broken down into components we will use in Figure 7.

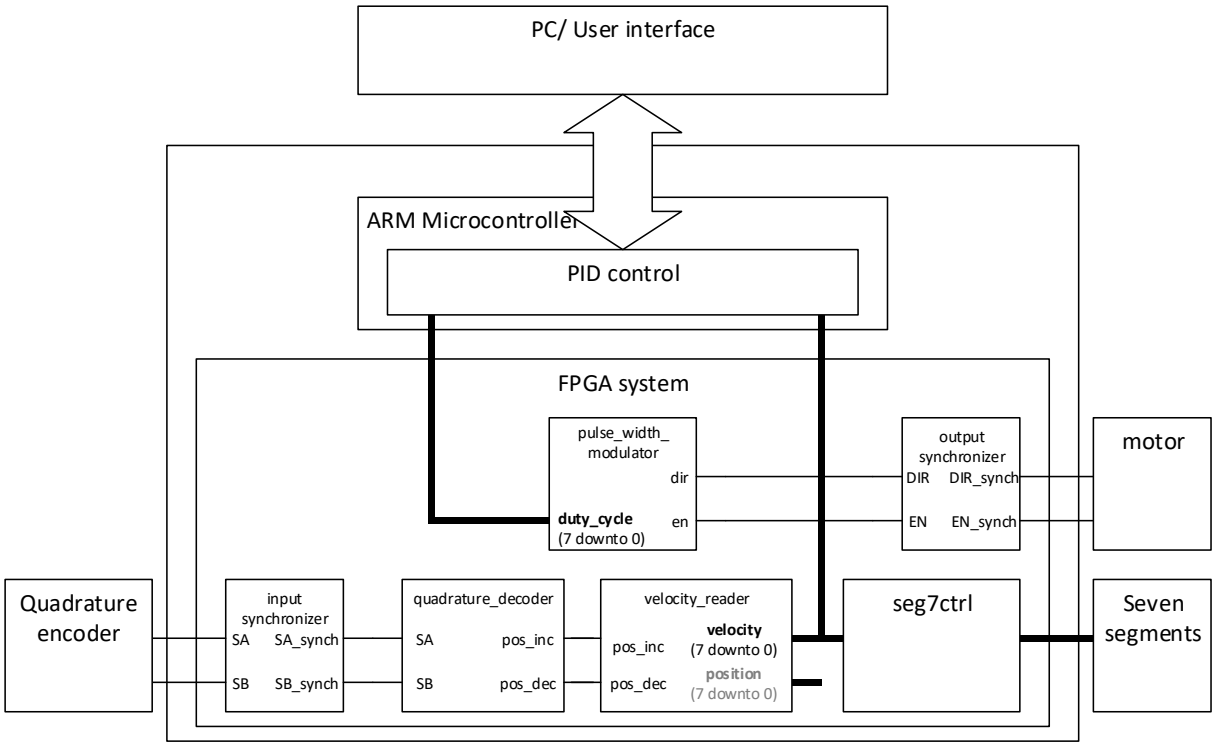

Figure 6. An overview of the control system.

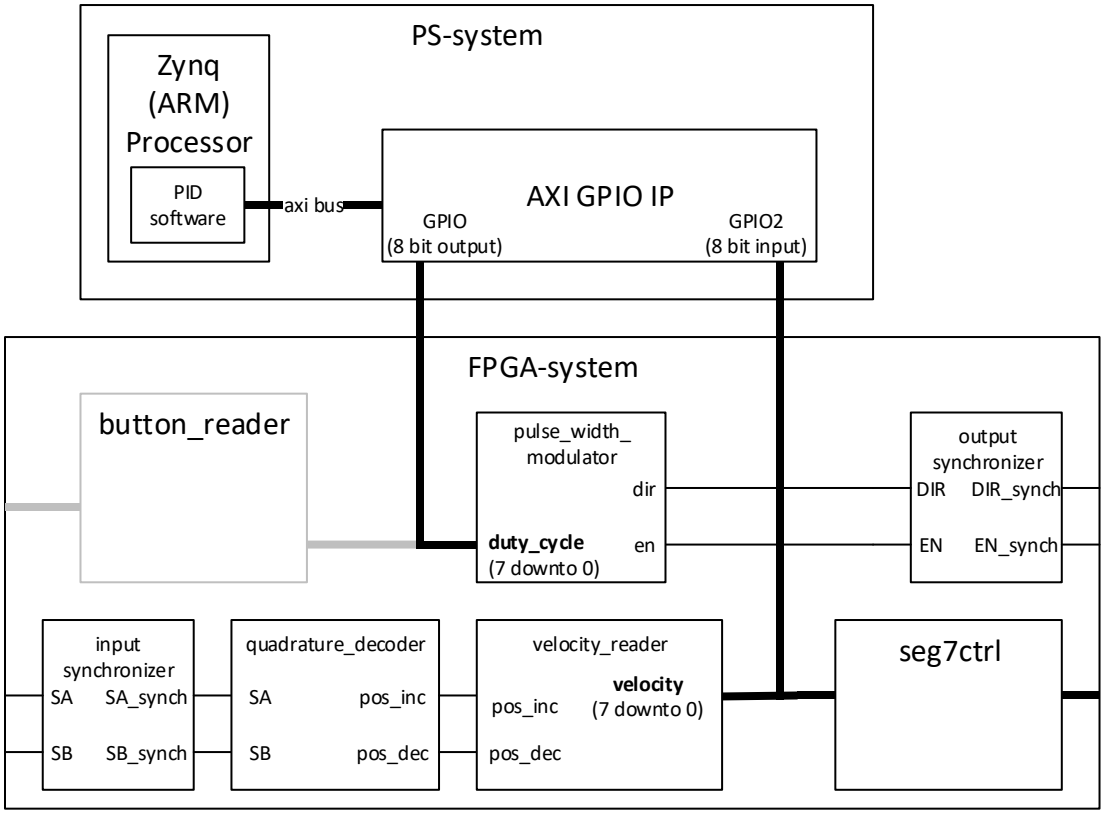

Figure 7. System on Chip

In addition to the Zynq processor and the C code that will run on it, we will use one of Xilinx' IP cores to connect our system from oblig 8 to the processor bus.

The ARM processor will be set up to uses the AXI (Advanced eXtensible Interface[1]) bus to communicate with the FPGA periphery. The IP core we connect will handle all the bus transactions needed, so we can utilize the processor output and set the processor input whenever we would like.

The button_reader (or the self test module) module from oblig 8 is grayed out, because it is not needed for this exercise. You can decide for yourself whether you would like to keep or remove that part.

Modify the the top module from oblig 8, to achieve an entity configuration as shown for the FPGA-system in Figure 7.

Extra: Note that the AXI GPIO IP is not necessarily in the same clock domain as our FPGA system. Thus, in order to ensure correct values are being read on both sides of the system you will have to *make sure that all outputs are buffered, and all inputs to the FPGA-system is synchronized* (as described in oblig 8). Create all logic that is needed to achieve this in a separate module inside the FPGA system.

[1] The AXI bus is a part of ARM Advanced Microcontroller Architecture (AMBA) which is an open-standard interconnect specification for SoCs
https://en.wikipedia.org/wiki/Advanced_eXtensible_Interface
https://en.wikipedia.org/wiki/Advanced_Microcontroller_Bus_Architecture

1: Creating an 'IP' that can be connected to the Zynq processing system

To enable the use of our module together with other IP (Intellectual Property – here other pre-made modules and the Zynq processor) in Vivado, we need to make an IP-package for our own project.

When Vivado creates a package it will essentially create an XML file with description of information Xilinx uses when connecting IPs. It will also generate a wrapper HDL entity, probably to ensure names follow Xilinx own ruleset. In this project, we will try not to modify these, however there may be cases where this is a necessity.

One thing to note about the current (2019.1 and earlier) version of Vivado, is that it doesn't support storing VHDL2008 as a feature. This means that packaging projects using VHDL2008 has to be done carefully, as synthesis of a packaged project normally will not work. There are several ways to work around this issue, and it is likely that you will encounter other methods when you start working as a hardware designer. Methods for going around vendor specific issues is not an intended learning goal for this task, thus we will only focus on one way to get you through the process of creating a functional IP for Vivado.

Follow these steps to create an IP from your already working project files (Oblig 8)

1. Create a new project in vivado called "oblig10_IP", using your updated top module.
    a. Use the updated top module (the new entity)
    b. Include all the other project files
    c. Make sure VHDL 2008 is selected for all VHDL files
    d. Turn off any constraints file (we should not bind pins at this stage).
2. Run synthesis
3. Select Open Synthesized Design when synthesis is complete
4. Verify that synthesis has completed without critical warnings
    a. If there are critical errors, correct them before continuing.
5. From the "*Tools*" menu select "*Create and package New IP*"
    a. Next
    b. Select "Package your current project"
    c. Select a location for your IP
    d. Next/Finish
6. Note that the packager will give critical warnings that VHDL 2008 is not supported for every VHDL 2008 module included. However, as we are not re-synthesizing the design, these warnings can be ignored at this point. This also means that from this point, we should not make changes in our VHDL code without starting over from the top. *Make sure there are no other critical warnings at this point.*
7. Review the "*Package IP*" tab,
    a. Check that the names make sense for your IP. Vivado will create names according to the VHDL entity name, so it should be recognizable. Consider if that name is a good one if you were about to re-use this design in a different setting and make changes if that makes sense to you.
    b. Click through the "*Packaging steps*" from "*Identification*" to "*Customization GUI*". No changes should be required.
    c. Click "*Review and Package*" then "*Package IP*"
    d. Click "OK" when Finished packaging your IP
8. Close the project to avoid making changes that would induce re-synthesizing of the package. (*File -> Close project…*)

At this point, the IP should be ready to fetch and use in Vivado. We will get back to this later when "4: Configuring and running our SoC control system".

2: Configuring an embedded system that can use the Zynq processor

To be able to use an embedded or softcore processor with an FPGA, we need to set up the working environment to be able to handle such a system. To accomplish this, we will use Xilinx own tutorials with some minor adjustments according to our local hardware.

"UG1165" is a collection of tutorials made by Xilinx to enable users to create various designs using their processor cores and other IPs. In this part, we will use the first tutorial, starting at page 13-26.

After having set up the environment, the next step will be to launch a basic "Hello world application" that runs C code on the processor.

3: Launching an 'Hello World' application on the Zedboard

Example project 'Running the "Hello World" application' on page 27-32 in "UG1165" will take you through the process of creating and running a c application that will run on the zynq arm processor core. This part must be done before proceeding to the next step.

4: Configuring and running our SoC control system

When you have the Hello World application up and running, there are few limits to what you can tell the processor to do. The next step is to create the full system containing both the processing system, the AXI GPIO IP and your own IP together with a dedicated C program that implements a PID controller. This tutorial will also be modified to accommodate our system. Start the modified tutorial "Using the GP Port in Zynq Devices" (page 32-44).

When the full system is up and running, demonstrate to the lab supervisor that you have made the system up and running for approval. *Normally this should be done in the lab, when not possible, creating a short video showing what happens will do. The video can be either screen capture or by using a cell phone to show what happens on your screen*[2].

Note: When modifying a SoC design, it may happen that the board support package is not created correctly. If important header files (such as *xgpio.h*) is missing when trying to compile c-code- this is a sign that the hardware must be exported again. To make sure this is done properly; try save the FPGA system (not the software) as a new project, and rerun the steps from "synthesizing the design" page 24, through "Exporting Hardware to SDK" before starting the "Working with SDK" tutorial on page 41. Make sure that the last generated HW platform is used.

Extra:

Modify the AXI GPIO module and the constraints to be able to read (SW 0-7).

Modify the C code to run the PID routine continuously after the initial direction test, using the switches as velocity setpoint. (The direction test must be performed). The setpoint should be between -40 and +40 using the eight switches (SW0-7) on the Zedboard.

[2] Screen capture can be done with programs such as Zoom or Open Broadcast Studio. The latter gives much more configurability, but requires more time to set up. Preferably use a relatively low framerate (10 FPS) and a resolution that will allow the screen text to be read, but not much more.