Oblig 8
Modules in a system, state machine design and synchronization.

IN3160/IN4160

Version 4/2024-01

Introduction

***Read* the entire introduction and the module specifications *before starting* to code!**
The goal of this lab exercise is to give you experience creating state machines and synchronization modules using HDL and integrating modules in a larger design. For each part of the exercise, you must follow the same design flow as in previous exercises.

The ability to control/regulate the position of a robot arm is a basic function for every robot. In this exercise, you will be creating an "IP" (Intellectual Property) module that will comprise the speed regulator for a motor. This will be used in both in this exercise and in oblig 10, together with a microprocessor.

As part of this process, you will learn:

- to create a large system
- to synchronize external and internal signals
- to use predefined simulation models in a test bench

Regulation systems in brief

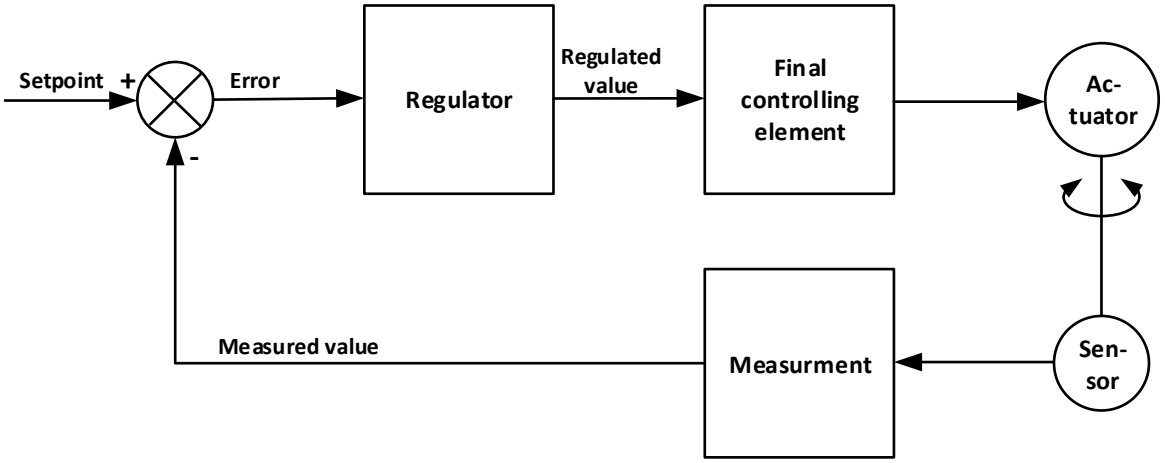

Figure 1: Regulation loop

Generally, a regulation system consists of an actuator we want to control and a sensor that measures the actuator output. The measured output is compared to our setpoint, which is the desired output for the system. The comparison results in an error signal (deviation between the setpoint and the measured value) that is fed back to the regulator. The regulator uses the error to calculate a regulated value. The regulated value is then translated by the final controlling element into a signal that will actuate the actuator.

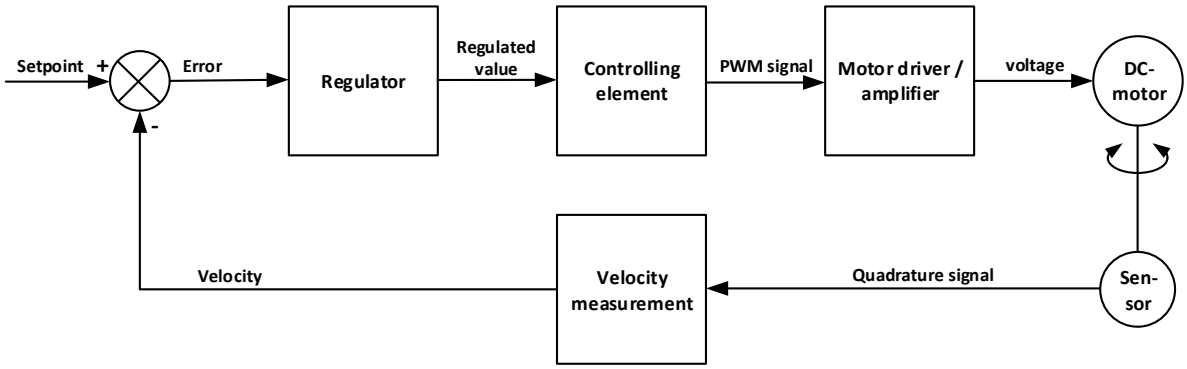

Figure 2: Regulation loop for this exercise

In this exercise, you will create a system that control the velocity of a motor. In the end, the system will consist of a PID regulator written in software that will run on the Zynq ARM core, which will pass a regulated value that will be used by the controlling element written in HDL. The controlling element will provide a pulse-width-modulated signal that can be sent to an H-bridge (PModHB3) that will drive the motor. The motor shaft is connected to an optical quadrature sensor (YUMO E6A2-CW3C) that sends its output back to the FPGA through the Pmod HB3 module. The quadrature output shall be be used to calculate a velocity value using HDL modules. The velocity value shall be displayed on seven segments used in earlier exercises, and it shall be used together with our setpoint to generate the error signal that can be fed into the PID regulator.

Figure 2 illustrates the regulation loop, and its manner of operation is as follows:

1. We impress a desired value, `setpoint` that is a desired value for the velocity.
2. `setpoint` is compared to a measured `velocity`, which gives the difference or deviation: `error = setpoint - velocity`.
3. `error` is used as input for the regulator, which processes this with some mathematical formula (favorable for what we want to regulate) and yields a result that is called the manipulated or regulated variable. The regulated variable is used to control the final controlling element.
4. The final controlling element is often a power transmission unit, which is controlled from the regulator, and exerts power sufficient to create movement. In our case, this consists of a module generating a PWM signal that are connected to the motor through an H-bridge. In other systems, there may be hydraulic components, pneumatic components, etc.

A block diagram of the modules that will be used or created in the system is given in Figure 3, below.

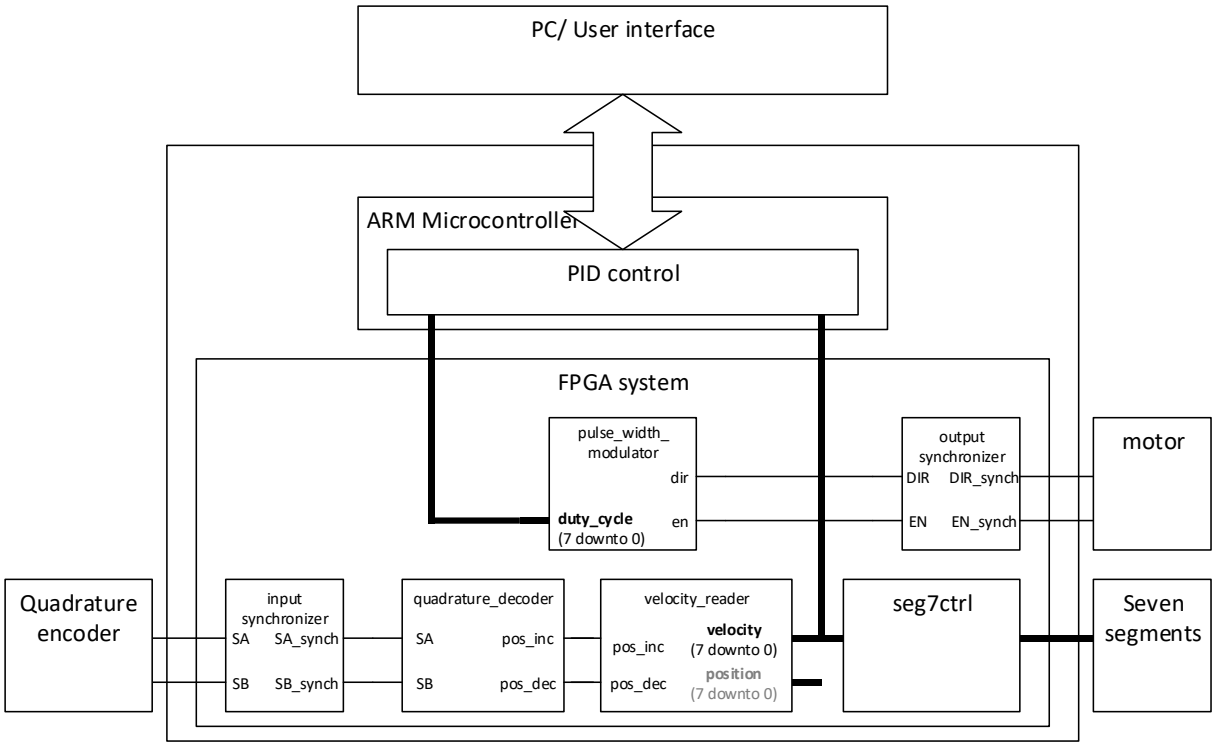

Figure 3: Long-term goal: To implement a control loop with a soft PID controller combined with hardware for generating PWM and capturing sensor data.

Creating this entire system is a tall order before finishing the first course in digital design, so we will circle in on specific parts, making the process manageable. One thing to note before doing that, is that when first starting a project, you will normally only have an idea of the major modules that will play a part in your design. In this design, there are several modules, some that were not identified initially. Thus, the initial block diagram may need to be refined several times during a design depending on the complexity.

For this design, we have aimed for having two levels of HDL code- one structural level, and one level with modules written at the register transfer level. If the complexity were to grow much more, we might want to add another structural level to make the top level easier to comprehend. Doing so would need to be thought through carefully, to avoid creating a messy structure where RTL code exists at multiple levels. Having the same type of code at each level is advisable to keep the system read- and maintainable.

Another thing to note about this design is that we have broken the design into modules that you may or may not want to merge or split. We could add modules to provide reuse of signals and thus reducing the overall design size. Questions that have come up are for example:

- Do we need separate modules for input and output synchronization?
- Would it be wise to create a “heartbeat” module to reduce the number of counters in the system?

You may have other questions, both now and after finishing the design, but for now, the structure is laid down before you, and we shall use it as it is.

Process for this exercise

In this exercise, you will go through the process of designing and testing the FPGA part of the overall system, with a few modifications to allow for some testing before we get into the system integration, which will be covered in a later exercise. The system for this task is shown in Figure 4, below.

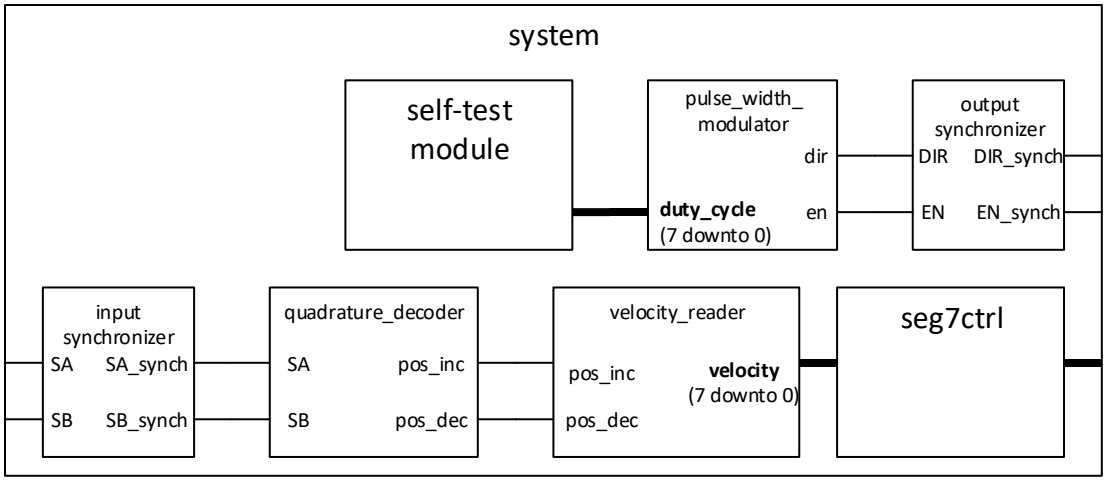

Figure 4: In this exercise, you will create a system consisting of several modules that can be used to test the functionality, before you will connect them to the ARM core in the next exercise.

a) The first task for this exercise is to create a self-test module that will provide an 8 bit signed duty cycle signal to the pulse width modulator (PWM module). The self test module shall provide at least 20 different values which shall be displayed for 3 seconds each. The values should be stored in a ROM after being read from a separate text file during synthesis. (This will make it easier to change values when testing the system). In this pattern, full speed in each direction shall be tested, and the motor direction should visibly turn at least twice. The last value sent to the motor shall be zero, to stop the PWM output. The test shall not repeat after finishing the sequence.

   Optional: *Create a way to have the self-test include either a linear or sinusoidal sweep of PWM values, lasting 10 seconds before the 20 values that are required.*

b) The next task for this exercise will be to create and simulate the PWM-module (pulse_width_modulator.vhd) that will be used to control the motor speed. The test bench for this simulation is pre-made (tb_pwm.py).
   - You can either create a new pwm module or
   - Modify the pdm module from assignment 7 or
   - Use the pdm_module from assignment 7 as a submodule.

c) After the pwm-module is tested and verified, you can create the in- and output synchronizing modules and test the pwm-module and the output synchronizer together with the self-test module. You will need to decide for yourself which modifications will be needed to achieve this.

d) Create and simulate the quadrature decoder. In this case, you will be creating the testbench. (see note on type names in cocotb)

e) The quadrature decoder shall be connected to the input synchronization module and the pre-made velocity reader. The velocity reader shall be connected to the seg7ctrl module. You will have to decide for yourself if you should modify the seg7ctrl module for this task. Note that the velocity reader output is an 8 bit signed. An absolute value of this may be the best option to display on the seven-segments.

   Hint: It is OK to only use Di(3 downto 0) for each display.

When all these modules are put together, as shown in Figure 4, the system shall run a the self-test, which will show how the motor responds to the duty-cycle numbers coded in the test pattern. While the motor is running, it should be possible (in a comprehensible manner) to read how fast it spins on the seven-segment display.

Report

Please hand in:

- VHDL source file(s)
- Python test bench for c), d)
- ROM content file for the self-test module.
- *Makefiles*
- `.xdc` constraints files
- Utilization report and Timing summary report
- Bit programming file for the final task.
- A brief report that sums up what has been done, and thoughts on problems/challenges.

All the submitted VHDL files should follow the naming rules for VHDL files and indenting guidelines as described in the cookbook.

Optional: The system can be run and shown to the lab supervisors for approval.

Note 1: cocotb does not have access to custom types, like state types.

"`dut.state.value`" will return a binary value, just as any other signal in the dut. To make sense of what state the dut is in, use a translation in the testbench. The binary values have a pattern determined by how you define the type. The first state has a value of 0, the second 1, the third 2, and so on. A python dictionary will be the best data structure for this translation. Eg.:

```
state_conv_table = {
    0: 'state1',
    1: 'state2',
    ...}
```

Specifications on the PWM-, synchronization and Quadrature modules follows:

pulse_width_modulator

In this task, you will create a module "`pulse_width_modulator`" that can control the motor speed by using a pulse with modulated signal. The motor is connected to an H-bridge board (PmodHB3).

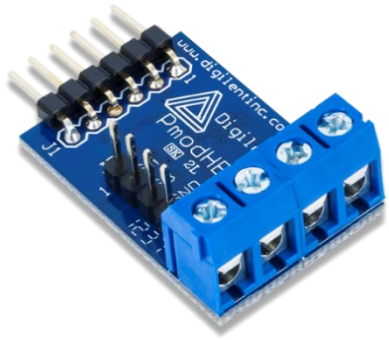

Figure 5: PmodHB3

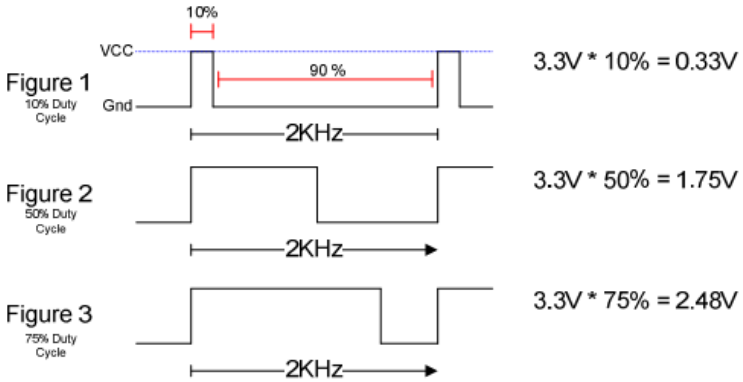

Figure 6: Pulse width modulation example

The H-bridge is controlled by two signals, DIR for direction and EN to enable output. Switching DIR will switch the polarity of the output, thus driving the motor in the opposite direction. **DIR Must never be changed while enable is high, since this will cause a short in the board.** Thus to avoid destroying the motor driver circuit we must provide a module that does not allow this to happen.

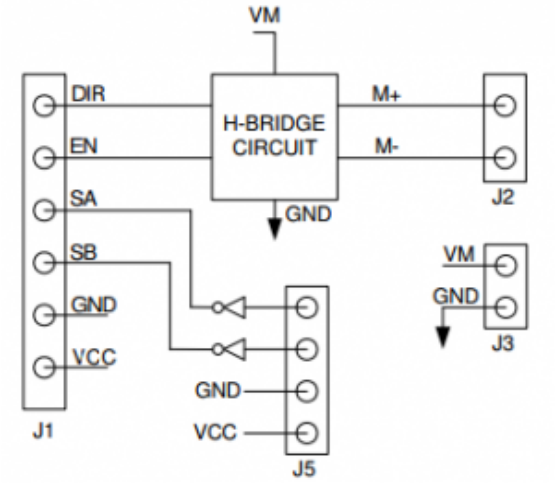

Figure 7: H bridge connection diagram

Pin	Signal	Description
1	DIR	Direction pin
2	EN	Enable pin
3	SA	Sensor A feedback pin
4	SB	Sensor B feedback pin
5	GND	Power Supply Ground
6	VCC	Positive Power Supply (3.3/5V)

Table 1: PmodHB3 pin description

The EN signal shall be pulse with modulated. The pulse frequency will affect how well the pwm signal is able to drive the motor with no load. The data sheet refers to 2 kHz, which should be "safe", but well within the audible spectrum. Selecting frequencies in the upper part of the audible spectrum (5-25 kHz) may yield unpleasant sound, going higher seems to heat up the PMod board while the motor impedance increases, thus you should keep the overall frequency of the output below 7 kHz.

The H-bridge is connected to the top row of the pmod port JB1 on the Zedboard. (When used remote, this can be hard to spot).

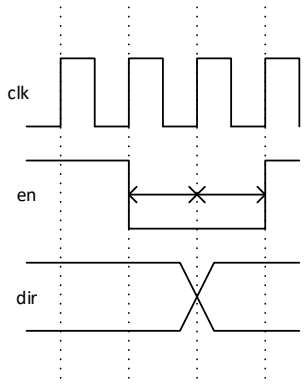

Figure 8: en is required to be low before and after dir changes

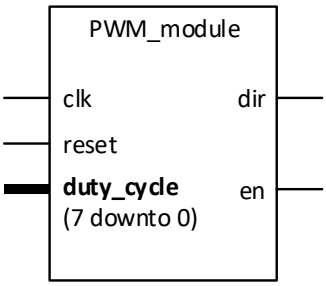

Figure 10: PWM module entity

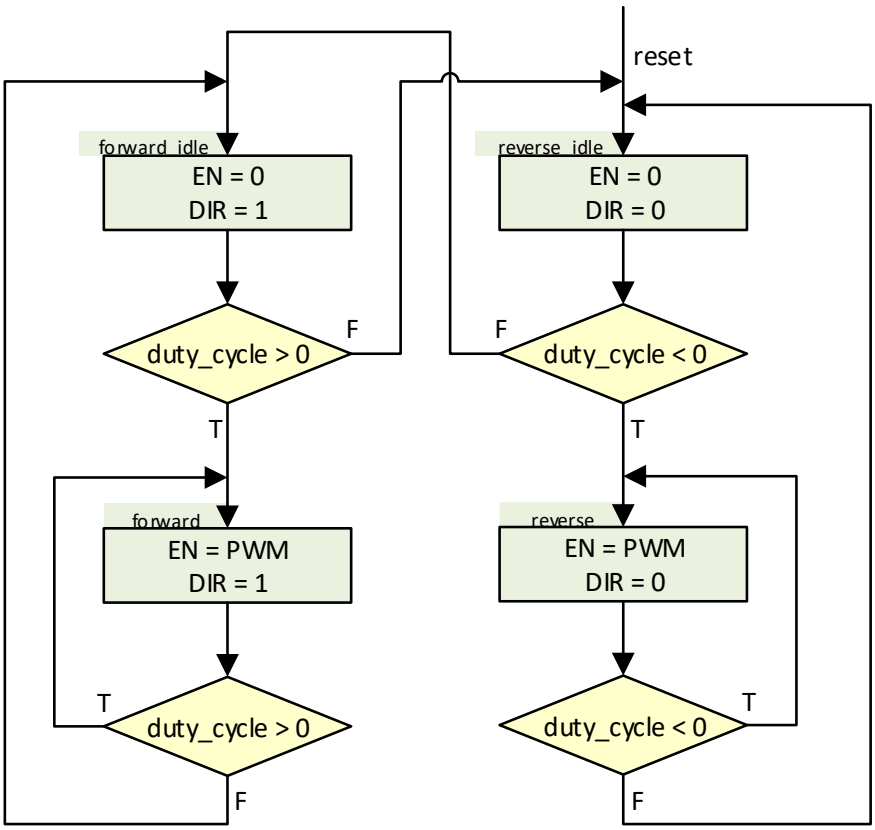

Figure 9: PWM module state machine

Implement the PWM module using the entity and state machine diagram in Figure 9 and Figure 10. The `duty_cycle` signal will be sent as a two-complement coded 8 bit std_logic_vector.

Use the test bench tb_pwm_module.py to test that the operation of the module is safe.

Do not use your module for implementation before the testbench can be run without errors or failures.

When the test is passed, you can implement the module (you will do that later anyway).

Since we are working with mechanical parts that wear out over time, please make sure the motor is off or turn off the board when you have confirmed operation.

Hints for pulse width modulator:

Type convert to `signed`, when using `duty_cycle` in calculations.

Using the pdm module from assignment 7, either modified or as a submodule `mea_req` can be tied to zero, and the following constants are recommended to ensure safe operation:

- `WIDTH: 20 bit`
- `MIN_OFF: x"000FF"`
- `MIN_ON: x"00FF0"`
- `MAX_ON: x"FFF00"`

If you are not using the pdm module from assignment 7:

Pulse width modulation can be achieved by comparing an n-bit counter to the `duty_cycle` absolute value. To get a frequency lower than 7kHz use the most significant bits on an n-bit counter. The absolute value can be found using the **`abs`** function. The keyword **`alias`** can be used to give an alias to a signal or a part of a vector.

```
function "abs" (X : signed) return signed;
     -- Result subtype: signed(X'length-1 downto 0).
     -- Result: Returns the absolute value of a signed vector X.
```

```
alias <new_name> : <type><(range)> is <signal_name><(range)>;
-- example
signal frukt : std_logic_vector(12 downto 2);
alias pærer : std_logic_vector(3 downto 0) is frukt(frukt'low + 3 downto frukt'low)
```

Synchronization modules

When we construct modules that both read asynchronous data, and set data that will be read outside our clock domain, we have to be aware of two basic hazards:

1: Input from external sources may change at random, thus creating metastability and unpredictable behavior in our registers.

2: Our output, even those coming from stable sources as a Moore machine, may be the source to glitches, due to combinational output based on our registers.

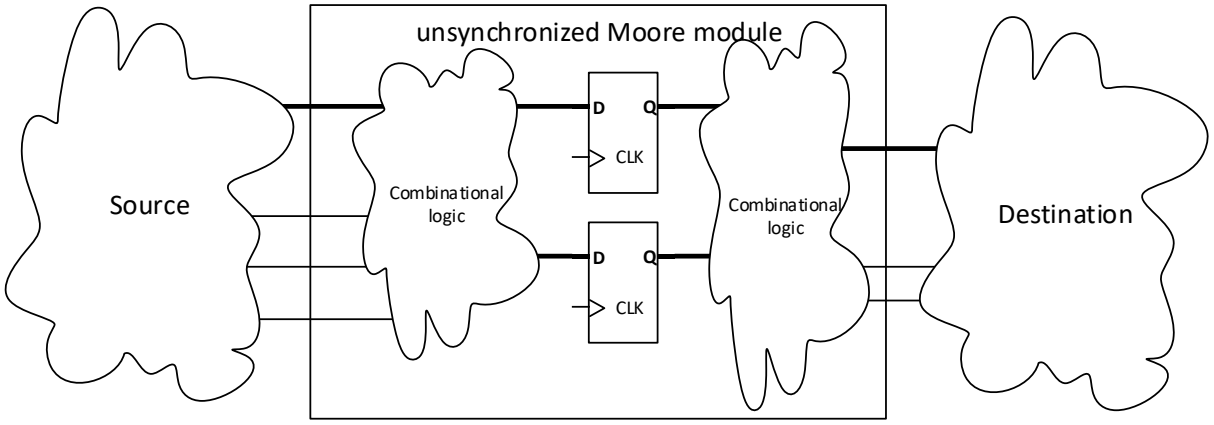

Figure 11: Moore type design in an unsynchronized environment.

To avoid metastability when reading input sources, we need specialized synchronizers, such as FIFOs or brute force synchronizers, depending on the input type.

To avoid glitches from the combinational output we may have two options- either create glitch free combinatorial output, our simply put flipflops on all our outputs. The latter will be preferable if it can be permitted in the design, as it is easy to design and verify. In some cases, however, we may have to construct specialized combinatorial logic, to ensure proper behavior without resorting to output buffers.

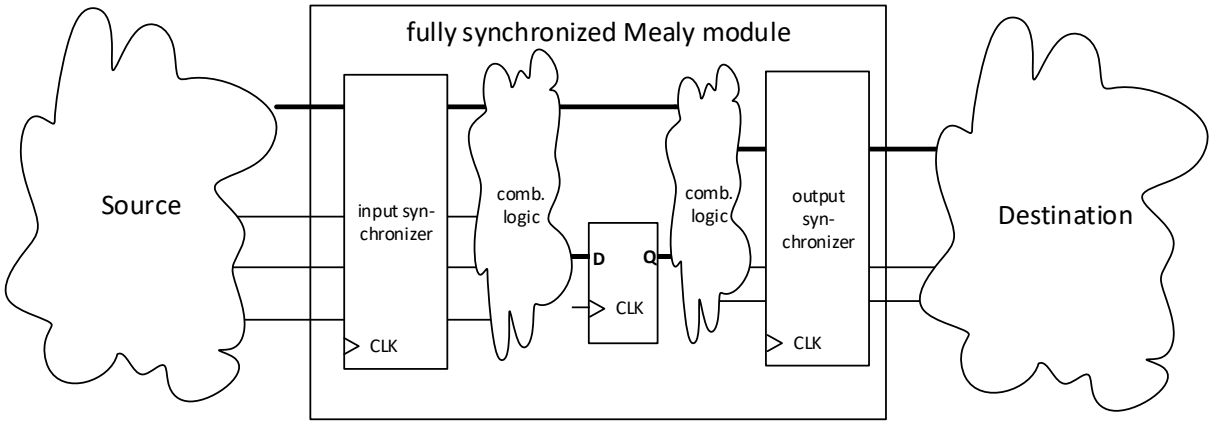

Figure 12: A fully synchronized Mealy type design.
Note that once we fully synchronize both inputs and outputs, Mealy type FSMs will generally be preferable, unless pipelining is required to meet timing requirements. When looking at the whole system, it can be seen as a Moore machine, but it is often beneficial to make the distinction between a system with a Mealy machine and output synchronization, compared to a Moore machine alone. Both Mealy and Moore machines may need output synchronization to ensure that the outputs are glitch free.

output_synchronizer

When it comes to synchronizing the seven-segment LED output, the human eye will likely not detect to short glitches as long as the output frequency is low compared to the overall clock frequency. Thus, we do not require output synchronization on the seven-segment output.

When it comes to the motor output, it is essential to avoid output glitches that may cause short-circuits in the driving circuit.

Create a module containing flipflops for DIR and EN to avoid creating output glitches on these signals.

input_synchronizer

Without using buttons, only the input of the quadrature encoder is asynchronous. Synchronizing these will be the task for this exercise.

Since the quadrature encoder output is gray code, we can use brute force synchronizers on the input (combined with faulty state recognition in the quadrature-decoder module).

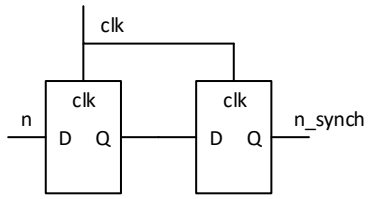

Figure 13: 1 bit brute force synchronizer.

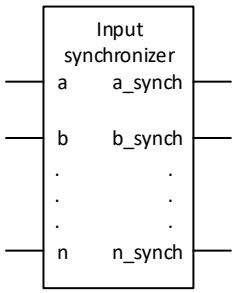

Figure 14: General synchronizer

Create a brute force synchronization module for the SA and SB signals from the quadrature encoder.

Quadrature_decoder

In this task, you will use the signals from the quadrature encoder to create pulses that show when the motor position has been either incremented or decremented one position.

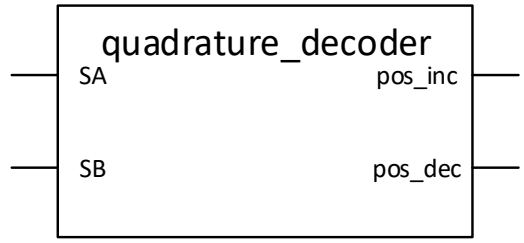

Figure 16: Quadrature decoder

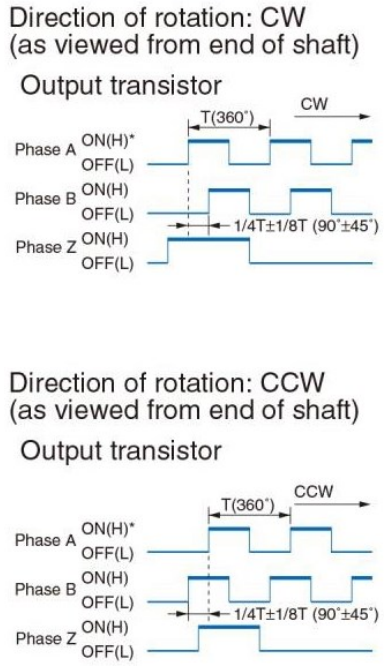

Figure 15: Quadrature encoder operation

The increment and decrement pulses shall be 1 clock cycle long. We shall have an increment or decrement pulse each time Phase A or B changes from the quadrature encoder.

Note that SA and SB are not synchronized with our clock signal, thus the output from the synchronization module from the previous task must be used as input to the quadrature decoder. Clock and reset signals shall be present, although omitted in the entity drawing.

The (internal) signal "err" is for declaring an erroneous state change. Normally this will never happen, as the encoder should be able to read all available speeds of the geared down motor shaft.

Table 2: State table for the quadrature decoder

AB	S_reset	S_init		S_0		S_1		S_2		S_3		current state
00	S_init	S_0	-	S_0	-	S_0	dec	S_reset	err	S_0	inc	next_ state/ output
01		S_1	-	S_1	inc	S_1	-	S_1	dec	S_reset	err	
11		S_2	-	S_reset	err	S_2	inc	S_2	-	S_2	dec	
10		S_3	-	S_3	dec	S_reset	err	S_3	inc	S_3	-	

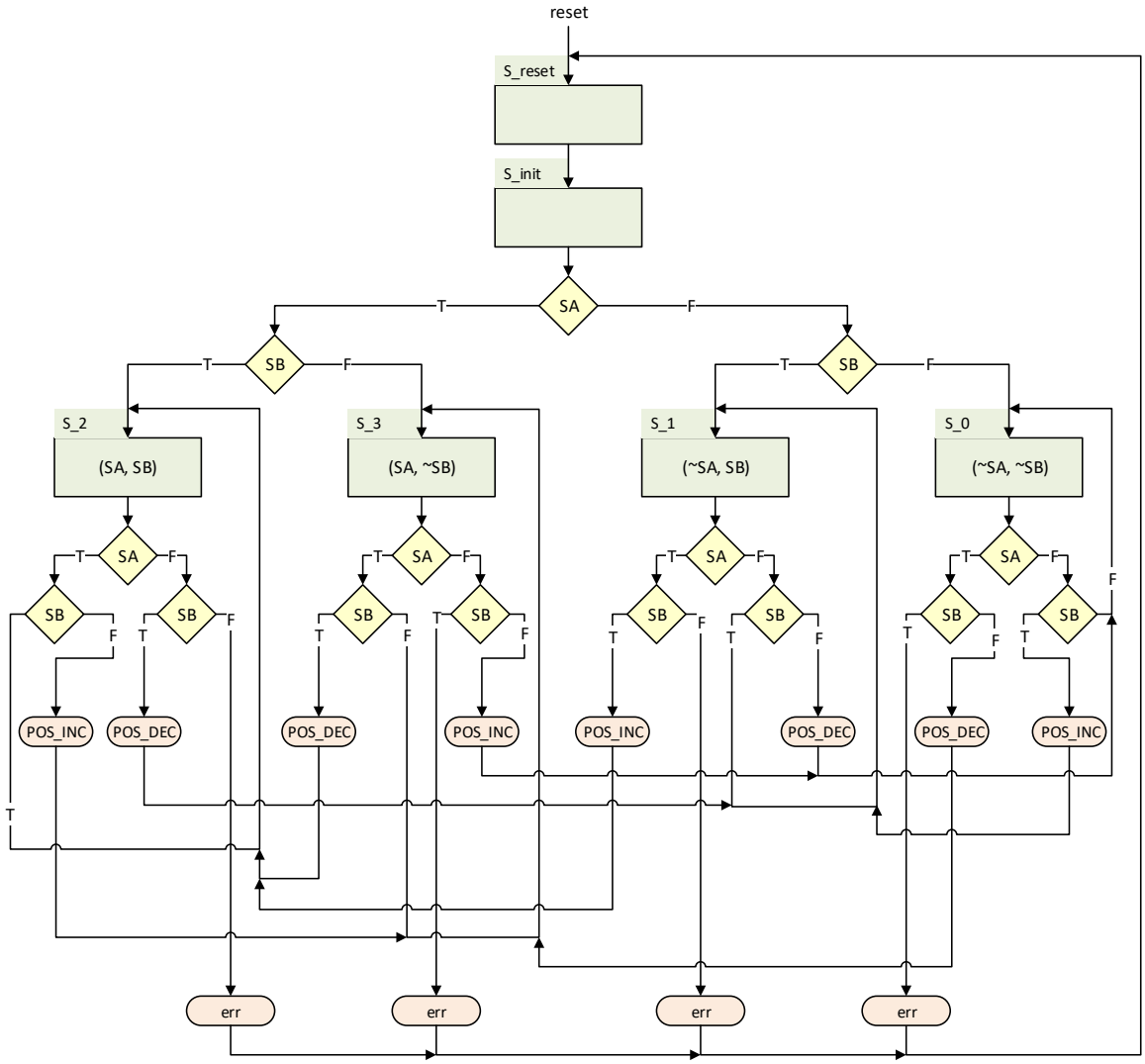

Figure 17: ASM diagram for Quadrature decoder[1].

Implement an FSM for the quadrature decoder as described above.

Tip: **Sometimes**, **a state table will be** both faster to make and **easier to read** than an ASM diagram. Multiple paths between states is a sign that this might be the case. (An ASM diagram will infer priority, whether that is necessary or not). *In this case, creating a bubble diagram for S0, S1, S2 and S3 (set out in a circle), omitting S_reset and S_init, might help visualizing the valid quadrature transitions. Including the S_reset and S_init will likely make the diagram harder to read, and it is strictly not needed, since the visualization is only a supplement to the state table.*

Tip: Using `case`, rather than nested `if`-statements may result in code that is easier to read and maintain.

---

[1] **The state machine shall not set SA and SB**, the parenthesis in the state boxes show the combination of SA and SB that lead to the state. Using parenthesis like this is strictly not ASM (which does not state any use of parenthesis in this manner), but it may save you some time determining which arrow goes where. Please also note also that this diagram is not intended to be 100% perfect – It showcases where an ASM diagram is a poor solution compared to bubble diagrams or a state table. Drawing your own bubble diagram is recommended.

About the quadrature encoder and the motor speed.

In our system, the position sensor consists of an *optical shaft encoder* (Yumo E6A2-CW3C). Shaft encoders have a shaft and a permanently mounted part. The shaft is connected to the shaft of the motor, and the electrical output is connected to the feedback pins in the PMOD H-bridge. A and B can thus be read from SA and SB pin according to Figure 7. Two pulses are received from the encoder, `a` and `b`, which have a 90 degree phase displacement. Which signal comes first depends on the rotational direction of the shaft. In our set-up we have the following relationship between the direction and phase displacement between the signals `a` and `b`.

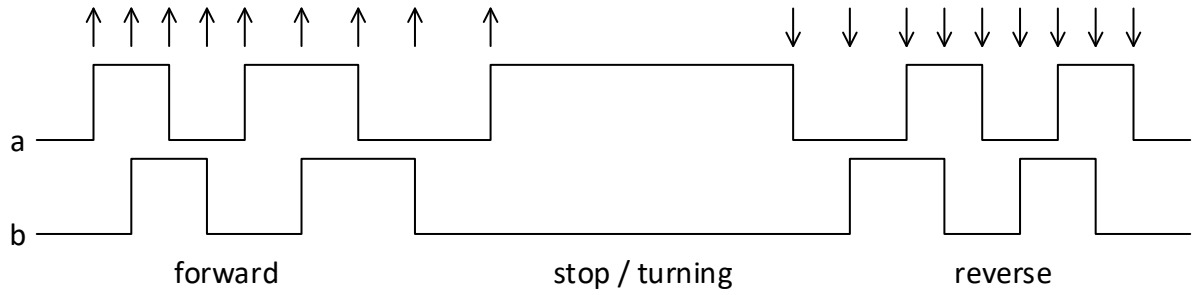

Figure 18: Position measurement. Arrow shows detected of transitions and direction

The encoder we use gives 200 pulses per round for each phase, which in total should enable us to read 800 positions for each round. That is we have 4 transitions of either a or b for each pulse period.

By implementing the state machine described in the quadrature decoder task, the FPGA will generate one pulse for each change in position, as shown in Figure 18.

Note that the the maximum rotation speed of the encoder is 5000 RPM which in turn may give 4 000 000 transitions (or 1 000 000 pulses for each phase) per minute. This corresponds to a switching frequency of 16,7kHz and a transition frequency of 66,7kHz, which both are well below our master clock frequency of 100MHz.

When connecting the motor directly to a 5V source, while unloaded, we can get around 300 RPM on the geared down shaft.

The velocity module does provide velocity in terms of rounds per 10s (coded as an 8 bit signed vector), given that the quadrature decoder provides increment and decrement signals according to the task specification. The format of rounds per 10s was chosen to enable all usable values to fall within the range of an 8 bit unsigned (between -128 and 127).