Lab Exercise 3
ASM Flow Diagram
Use of modules in a system and design of a state machine

INF3430/INF4431 Autumn 2016

Version 1.3/10.10.2016

Part 1. ASM flow diagram

Exercise 1

In this exercise, we will create ASM flow diagrams for an FSM that controls a soda vending machine (VHDL code is not to be created).

It is important to remember from Zwolinski, pages 86-87, that the character = means to set a signal to a value that lasts only for a clock period and is reset to the default value (normally 0) when the clock period is over (or alternatively, always sets the value!). The characters <– are used to set a value for a signal or a vector (i.e. many bits) at the end of a clock period (i.e. during the next clock period) and maintain the value until a new value is set (i.e. becomes a register). In the exercises below both of the two assignments *must* be used.

a)

The first version of the machine is only an initial prototype where we have simplified the control so that the soda is free. The entity for the FSM is stated below, and a *Moore FSM* is to be created that gives an active high signal `servesoda` which lasts for a clock period of 20 ns (i.e. 50 MHz) when the signal `reqserve` is active high for a clock period (the input signal `reqserve` is always active high for just a clock period at a time; any bouncing has already been eliminated). The signal `led` should be active high for 4 seconds after `reqserve` has been active.

```
entity sodamachine_simple is
  port (
    clk       : in  std_logic;          -- Clock
    rst       : in  std_logic;          -- Asynchronous reset
    reqserve  : in  std_logic;          -- Request serve (start)
    servesoda : out std_logic;          -- Serve soda (dispense)
    led       : out std_logic);         -- LED active high 4 s
end sodamachine_simple;
```

b)

In the next version of the soda vending machine, the soda will cost the amount determined by the signal `sodacost`, given in kroner. The machine shall be able to give change as 10 or 1 kroner when the signals `coinback10` and then `coinback1`, respectively, are active high for as many clock periods as the number of coins indicates (i.e. `coinback1` high for 4 periods will be 4 1-krone coins) after `led` has been illuminated for 4 seconds. We assume now that there is an infinite number of coins in the vending machine, so that it will always be possible to give change and the signals `sodacost` and `money` cannot be greater than 200 kroner, which is the maximum amount the vending machine accepts in the form of bills or coins. If too little money is inserted before `reqserve` is pressed, all of the money will be returned and the led will illuminate, but `servesoda` does not become active. The new entity is stated below, and a *Mealy FSM* is to be created now.

```
entity sodamachine_advanced is
  port(
    clk        : in  std_logic;         -- Clock
    rst        : in  std_logic;         -- Asynchronous reset
    sodacost   : in  std_logic_vector(7 downto 0);  -- Unit cost
    reqserve   : in  std_logic;         -- Request serve (start)
    money      : in  std_logic_vector(7 downto 0);  -- Money
    servesoda  : out std_logic;         -- Serve soda (dispense)
    led        : out std_logic;         -- LED active high 4 s
    coinback1  : out std_logic;         -- 1 kroner back
    coinback10 : out std_logic);        -- 10 kroner back
end sodamachine_advanced;
```

Part 2. Modules in a system and design of state machines

Introduction

The goal of this lab exercise is to give you experience with creating state machines in VHDL. For each part of the exercise you must follow the same design flow as in lab exercise 2.

The ability to control/regulate the position of a robot arm is a basic function for every robot. In this exercise, we will be creating an IP (Intellectual Property) module that will comprise the position regulator for a robot. We will be using this in a number of contexts, in this exercise and in the next, where we will use it together with a microprocessor.

As part of this process you will learn:

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

Position regulation in brief

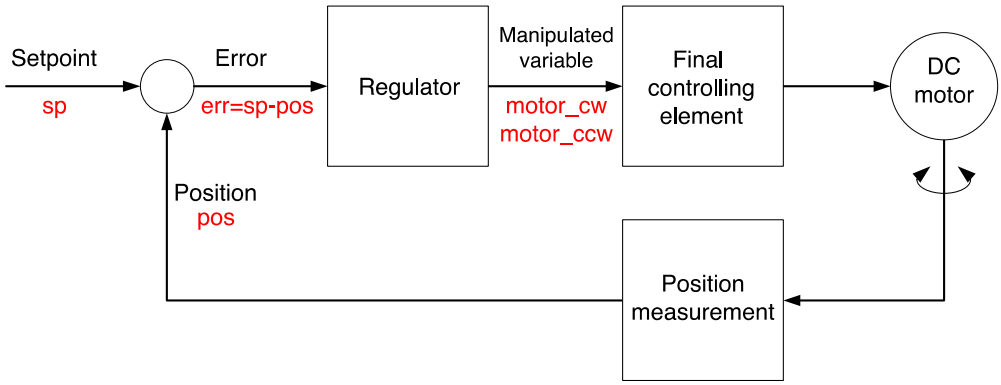

Figure 1. Regulation loop

When the position of a robot arm is to be controlled/regulated, it is common to do this by means of an electric motor at each joint. We want to create a regulation of the position. With regulation, we always have feedback with measurement of what we want to regulate. The main idea is: if you want to have control of where you are going, you have to know where you are. This feedback is lacking with control (take aim and close your eyes).

The figure above illustrates such a regulation loop, and its manner of operation is as follows:

1. We impress a desired value, `sp` (demand or setpoint) that is a desired value for the position.
2. `sp` is compared to a measured value of the position, `pos`. The difference `err=sp-pos` is called the deviation.
3. `err` is used as input for the regulator, which processes this with some mathematical formula (which is favorable for what we want to regulate) and yields a result that is called the manipulated variable. The manipulated 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 power transistors that are connected directly to

the motor. In other systems, there may be hydraulic components, pneumatic components, etc.

Position sensor

In our system, the position sensor consists of an *optical shaft encoder*. You can read more about this shaft encoder at http://www.usdigital.com/products/encoders/incremental/rotary/shaft/s4/. Shaft encoders have a shaft and a permanently mounted part. The shaft is connected to the shaft of the motor. 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`.

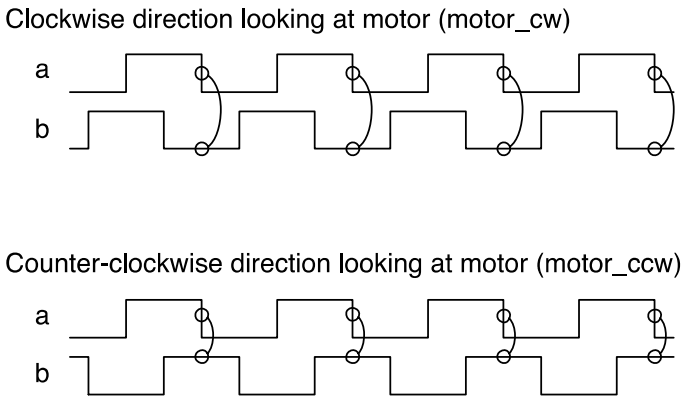

Figure 2. Position measurement. Detection of periods and direction

The encoder we use gives us 360 *bounce-free* periods for the signals `a` and `b` per revolution. If we detect the signals as illustrated in the figure above, it gives us a resolution of 1 degree per pulse. We can cover a range of 0–127 degrees by means of an 8-bit *signed* data type for the position, `pos`[1].

Regulator

Regulators of the PID (Proportional-Integral-Derivative) type are often used. The manipulated variable can be expressed then as the sum of three elements: Manipulated variable = P + I + D, where P=Kp*err, I= Ki*∑err and D = Kd*Δerr. Those who have a special interest can read more about this in the application note from Atmel, for example: http://www.atmel.com/images/doc2558.pdf

---

[1]By detecting each flank of `a` and `b`, we could achieve a resolution of 0.25 degrees/flank. For us, however, it seems appropriate to use an 8-bit number, since this gives us the opportunity to use the switches on the board in an easy way to create the setpoint, `sp`.

In our example we will be making significant simplifications, however, they function very well in practice. Our regulator should function as follows: *If the deviation is greater than zero, full speed clockwise. If the deviation is less than zero, full speed counterclockwise. If the deviation is zero, then the shaft is in the desired position and the motor should be idle.*

This functions well, because our motor has a gearbox with a gear ratio of several hundred to one, which is thus a *sluggish* regulation. Regulating in this manner corresponds to a P regulator with very high amplification.

Motor and final controlling element

In our example, we will control the motor by means of two signals, `motor_cw` (clockwise) and `motor_ccw` (counterclockwise) according to the following table:

Table 1. Motor signals

motor_cw	motor_ccw	Motor movement	Motor driver LED
0	0	Idle	Off
1	0	Clockwise	Green light
0	1	Counterclockwise	Red light
1	1	Idle (illegal comb.)	Off

These signals are sent to the expansion connector `JA` on the ZedBoard. A seven-segment display expansion board is connected to the connectors `JA` and `JB` of the ZedBoard, and has two 6-pin connectors (`J2` and `J3`) for connecting a motordriver board. These connectors have 3.3 V voltage and ground plus four signals. This means that we have just enough space for the signals `motor_cw` and `motor_ccw` for the final controlling element, and the position signals `a` and `b`. A *Motordriver* board can be connected to one of the connectors. Motordriver contains connectors to connect the position sensor, motor and a 12 V power supply. The Motordriver board has also been equipped with galvanic isolation to avoid any problems with the grounding. The manipulated variable signals are connected to a purchased, finished module, consisting of an IC with power transistors. This comprises our final controlling element. You can read more about the power transistor module at:

http://www.sparkfun.com/commerce/product_info.php?products_id=8907

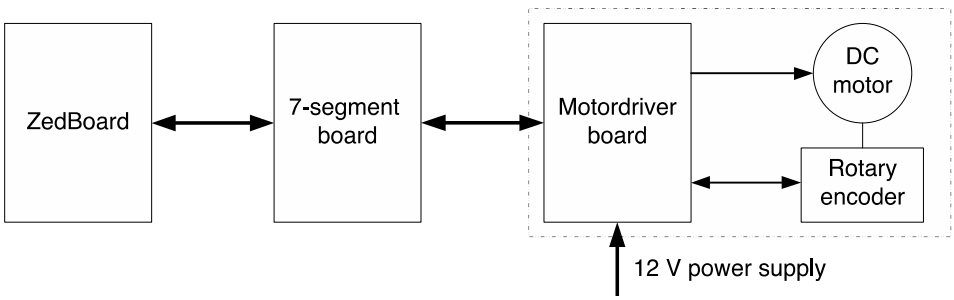

Figure 3. Overview of the system

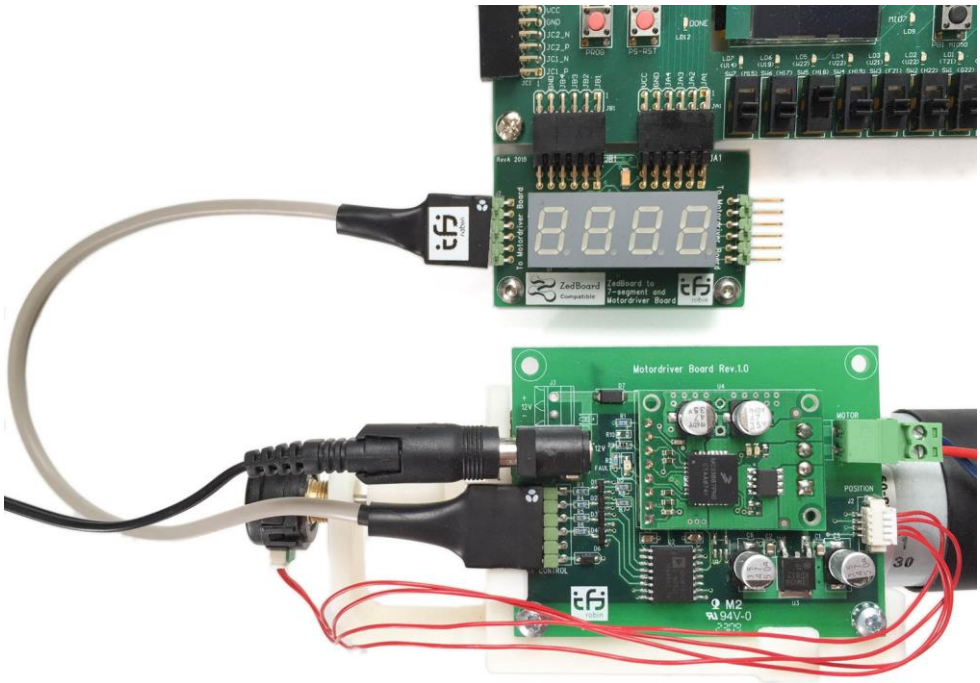

Figure 4. Connecting the motordriver board. Note the white dot on each end of the cable, indicating the location of pin number 1.

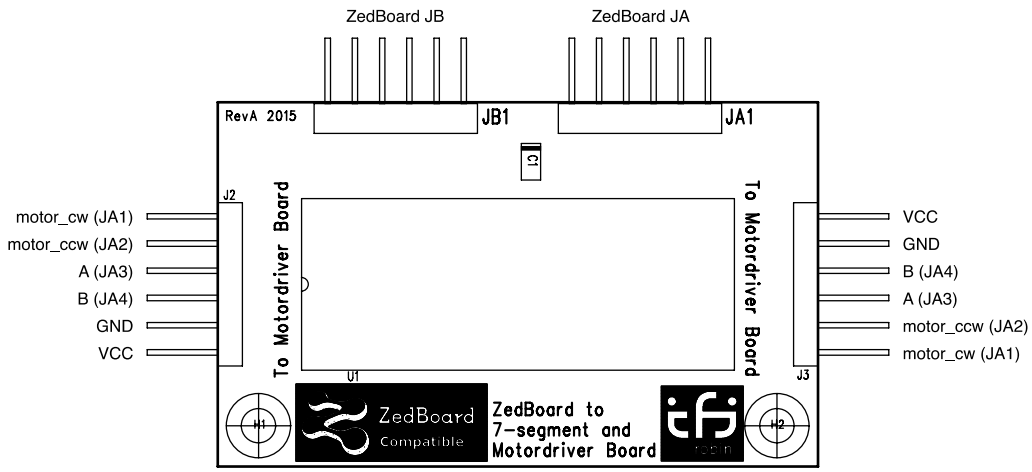

Figure 5. Pins of the 7-segment and motordriver board

Report

For each part of the exercise, you must hand in:

- VHDL source file(s)
- VHDL test bench

- *Do* file(s) for simulation in Modelsim
- `.xdc` constraints files if relevant
- Utilization report and Timing summary report

A brief report that sums up what has been done and includes problems/challenges. In addition, you must demonstrate the finished system in part 4 of the exercise for the group instructor in the lab.

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

Exercise 2

Construct a position sensor by means of a state machine or counter. Use the attached entity `pos_meas_ent.vhd`, and use the following ASM flow diagram as your starting point. Remember that the signals `a` and `b` are asynchronous and therefore require synchronization flip-flops. For asynchronous/synchronous resets, the position should be reset to `0`. The position should vary between 0–127, and you must make sure that it never goes beyond this range, as the most significant bit will be used for something else in the next exercise.

HINT: It may be useful to use the attached simulation model for motors and position sensors.

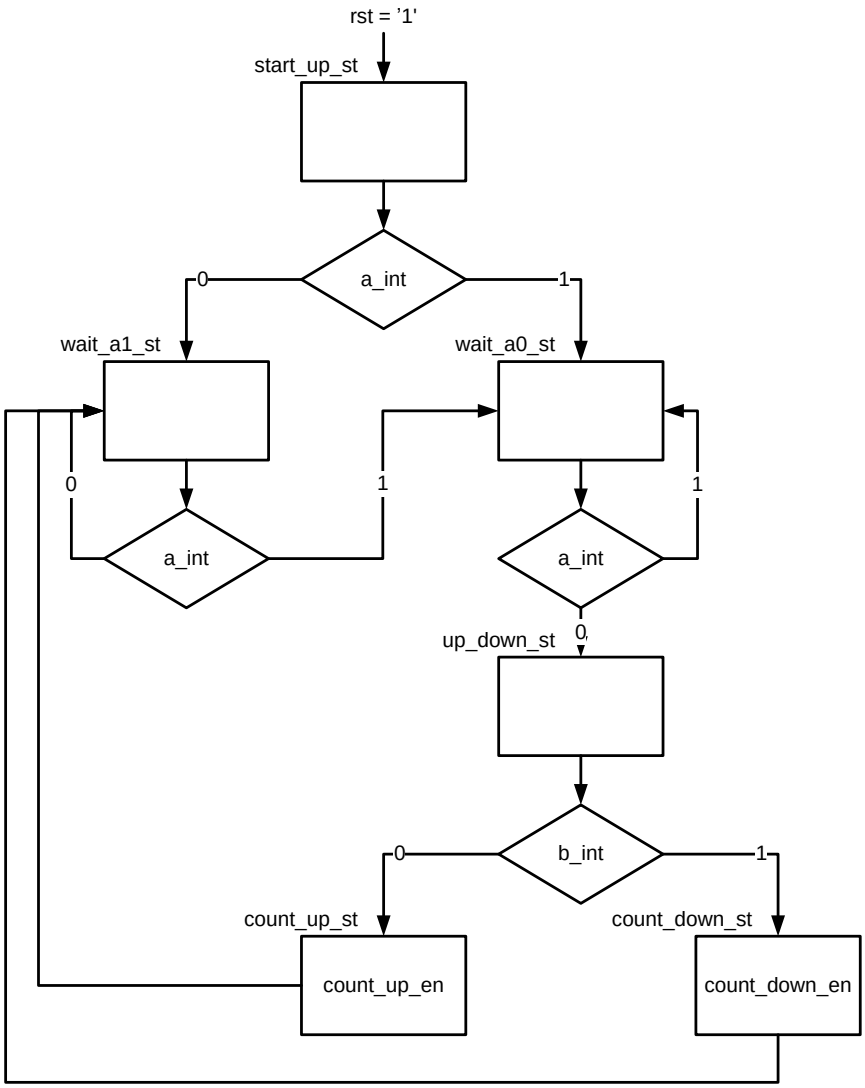

Figure 6. ASM flow diagram for position sensor

Simulate the module thoroughly first with a test bench. Then test the module on the test board. It is particularly important to never go beyond the measurement area between [0,127].

Hint: To test `pos_meas` on the board, it may be a good idea to instantiate it in a test module, `test_pos_meas`, which uses BTNL and BTNR to create movement in one direction or the other.

Exercise 3

Implement a regulator, `p_ctrl`, based on the following ASM flow diagram. Use the attached entity in `p_ctrl_ent.vhd`. Simulate the module thoroughly with a test bench. This module should only be simulated and *not* tested on the test board.

The input signals `pos` and `sp` require synchronization flip-flops, since `sp` is an external signal and `pos` is an internal signal that has a clock other than the master clock for the rest of the system in the following parts 3 and 4 of the exercise, in which `p_ctrl` is to be used.

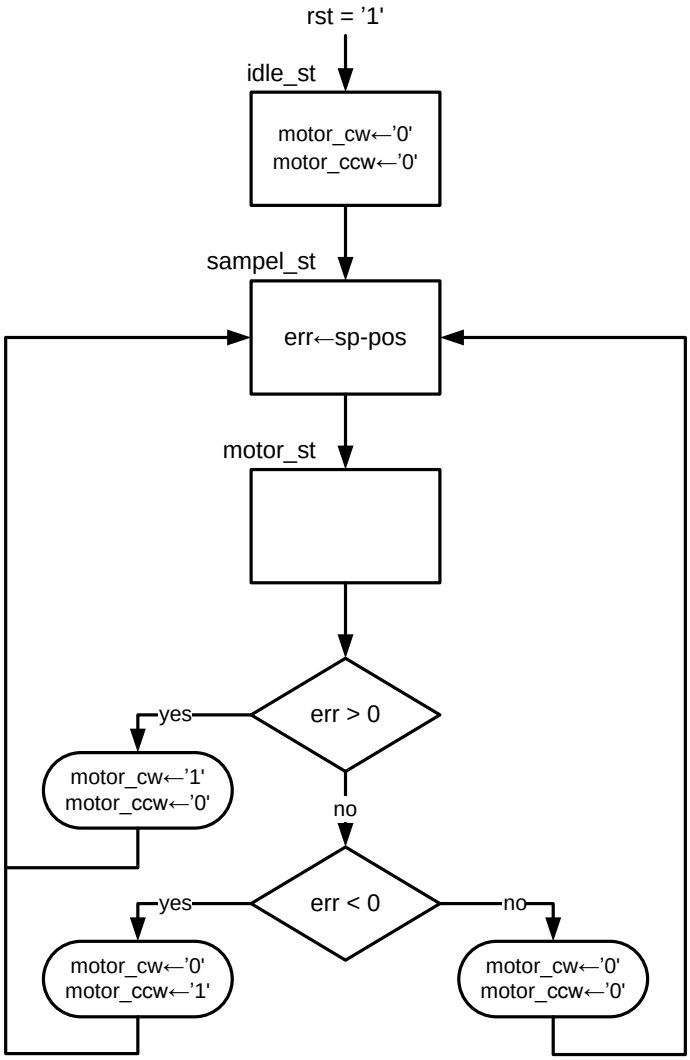

Figure 7. ASM flow diagram for `p_ctrl`

Exercise 4

Integrate the position sensor from a) and the regulator from b) in a new module, `pos_ctrl`. Use the attached entity, `pos_ctrl_ent.vhd`. Note that `p_ctrl` must have its own clock, `mclk_div`, which is not the master clock. It should be possible to force the motor to run clockwise or counterclockwise by means of two signals: `force_cw` and `force_ccw`. A multiplexer selects which motor signals apply in accordance with the following truth table:

Table 2. Forced running of motor (F1)

force_cw	force_ccw	motor_cw	motor_ccw
0	0	`cw` (from `p_ctrl`)	`ccw` (from `p_ctrl`)
0	1	0 (`force_cw`)	1 (`force_ccw`)
1	0	1 (`force_cw`)	0 (`force_ccw`)
1	1	`cw` (from `p_ctrl`)	`ccw` (from `p_ctrl`)

Forced running is important when the position sensors are to be reset in a known position, which will typically be an end position. Mask away the top bit in `sp` so that the value of `sp` never exceeds the 0–127 range, before it is connected to `p_ctrl` (`sp <= '0' & sp(6 downto 0)`).

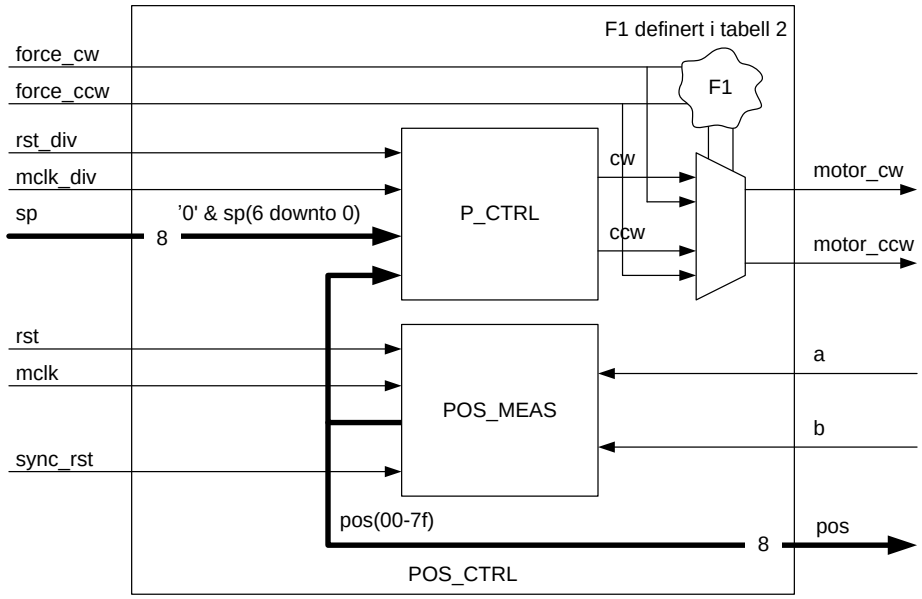

Figure 8. `pos_ctrl` IP

`pos_cntr` is not to be implemented in an FPGA, but simulated thoroughly with a test bench.

Exercise 5

Connect up a system, `pos_seg7_ctrl`, consisting of `pos_ctrl` and the seven-segment controller from lab exercise 2, as well as a Clock Reset Unit (CRU).

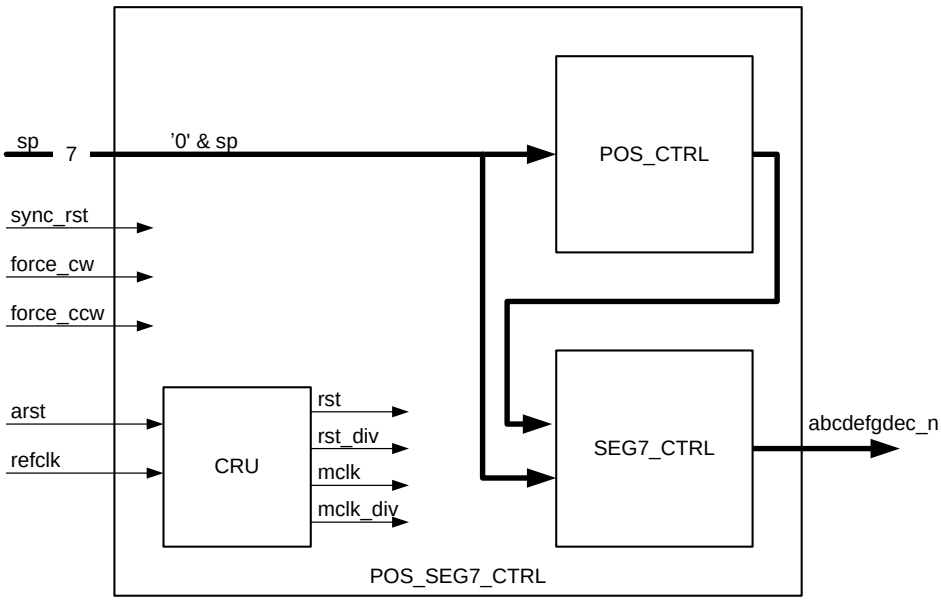

Figure 9. `pos_seg7_ctrl`

`sp` shall be displayed in DISP3 and DISP2 and the position, `pos`, shall be displayed in DISP1 and DISP0. Connect the setpoint, `sp`, to the switches SW6–SW0. Connect `force_ccw` and `force_cw` to BTNL and BTNR, respectively. Connect `sync_rst` to BTNC. Clocks and reset are taken from the Clock Reset Unit, CRU. `p_ctrl` must use the clock `mclk_div` and the reset signal `rst_div`.

Create a constraints file (`.xdc`), which, in addition to pin number constraints, has timing constraints for all input and output signals, in addition to the external clock signal `refclk` and the timing constraints of the internal clock `mclk_div`. See the lecture slides, which include an example of an `.xcf` file.

Verify that the design is constrained. Any unconstrained paths must be documented.

`pos_seg7_ctrl` must be both simulated and implemented in the FPGA. It will also be used in lab exercise 4 together with a microprocessor.

GOOD LUCK!