Lab 7 – Robust Rotary Encoder Design
Parts A and B

Introduction

Often when working with designs that require some type of position or rotation sensing, an electro-mechanical device called an *encoder* is used. It may either encode linear position (linear encoder) or rotation (rotary encoder). In this lab you will be introduced to the latter, a *rotary encoder*: a device where using the proper electrical interface you can determine the rotation angle of a shaft. A few example applications are: a) an interface between a motor and a control circuit to provide information to control the speed of the motor; b) a volume or tuner control on a digital audio system.

The goal of this design is to use the encoder signals as inputs to a counter. We will count up or down based on the direction of the rotation. The four digit display will show the counter outputs. A high level block diagram is shown below.

This lab will be divided into two parts. In *part A* you will design the encoder datapath. In *part B* you will design the controller or *Finite State Machine* for the design.

There are many technical documents and videos on the web describing the operation of a rotary encoder. Here is one you should review before continuing these instructions:

https://www.youtube.com/watch?v=zzHcsJDV3_o&feature=emb_logo

This video demonstrates optical components as the sensing elements. We will be using a device that has mechanical contacts as the sensing elements – but the ideas are the same. Note in the second half of this video the device has two output channels: A and B. These are out of phase by 90 degrees and thus are also known as *quadrature outputs* (out of phase by a quarter of a revolution).

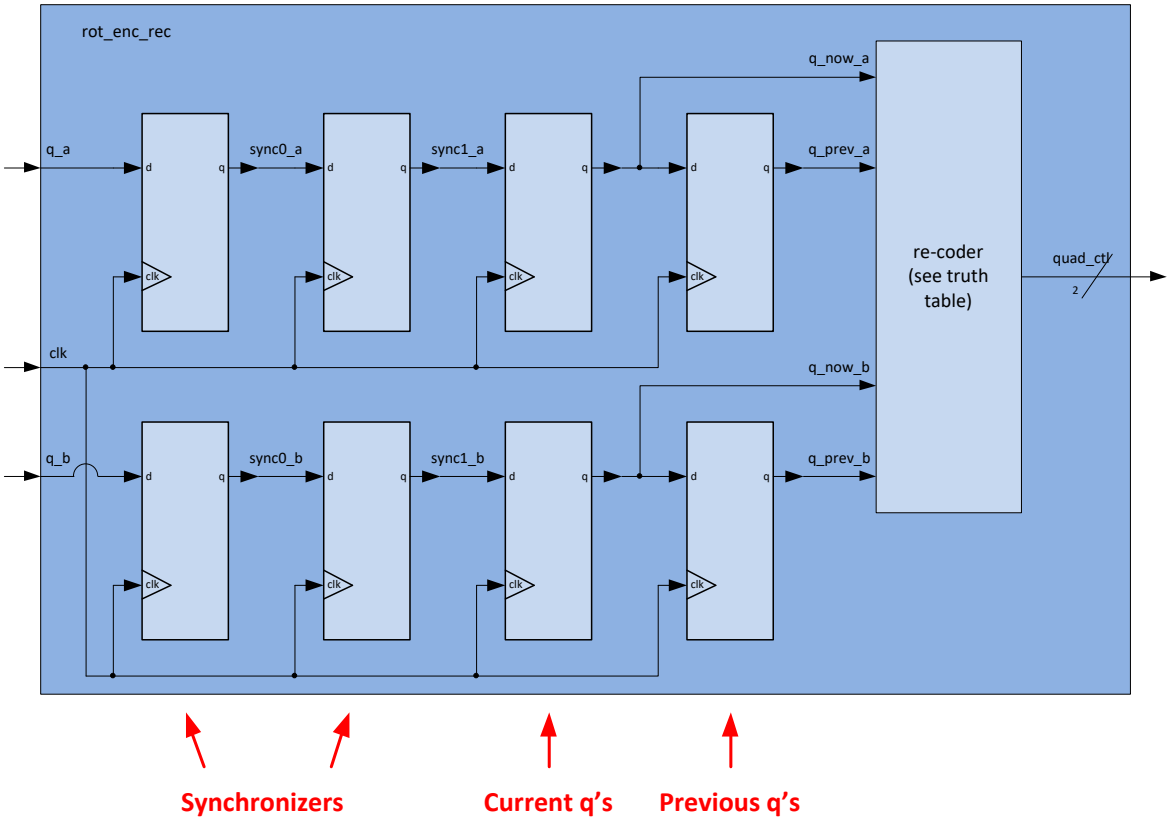
For this lab I will call these two quadrature signals: ***q_a*** and ***q_b***. There are a few things of interest with such signals:

- When rotating the shaft, direction ***a*** will have a two-bit sequence `{q_a,q_b}` of: ***2'b00, 2'b01, 2'b11, 2'b10, 2'b00, 2'b01, 2'b11, 2'b10, 2'b00, …***

- When rotating the shaft, direction ***b*** will have a two-bit sequence `{q_a,q_b}` of: ***2'b00, 2'b10, 2'b11, 2'b01, 2'b00, 2'b10, 2'b11, 2'b01, 2'b00, …***
- Note the sequences opposite each other as per each rotation direction.
- The sequence is a 2-bit *gray code* – i.e., only one bit in the sequence changes.
- In order to determine the direction of rotation, we need to store the previous 2-bit code.
    - We will use D-ff's to store the previous code.
    - Because the signals will be asynchronous with the clocks driving the D-ff's, there will often be setup and hold violations.
    - To take care of these violations, we will use two D-ff's back to back as synchronizers.
- We do not need to *debounce* the inputs, as if there is some contact or debounce noise from a mechanical switch, the design of the circuit will take care of it (another advantage of using gray code).

Detailed Specification:

Let's look at a block diagram that incorporates these synchronizing and storage ideas for the input quadrature signals.



We have the two quadrature inputs (***q_a*** and ***q_b***) and a system clock (***clk***) as inputs. For each quadrature input, we have two D-ff's back to back as synchronizers, followed by two D-ff's to store the current (***q_now***) and previous state (***q_prev***) of these signals. *The importance of these signals is that now they are synchronized to our system clock*.

Now, how should we to process these quadrature signals? Let's map out all of the possibilities of these four signals (***q_prev_a, q_prev_b, q_now_a, q_now_b***) in the truth table below.

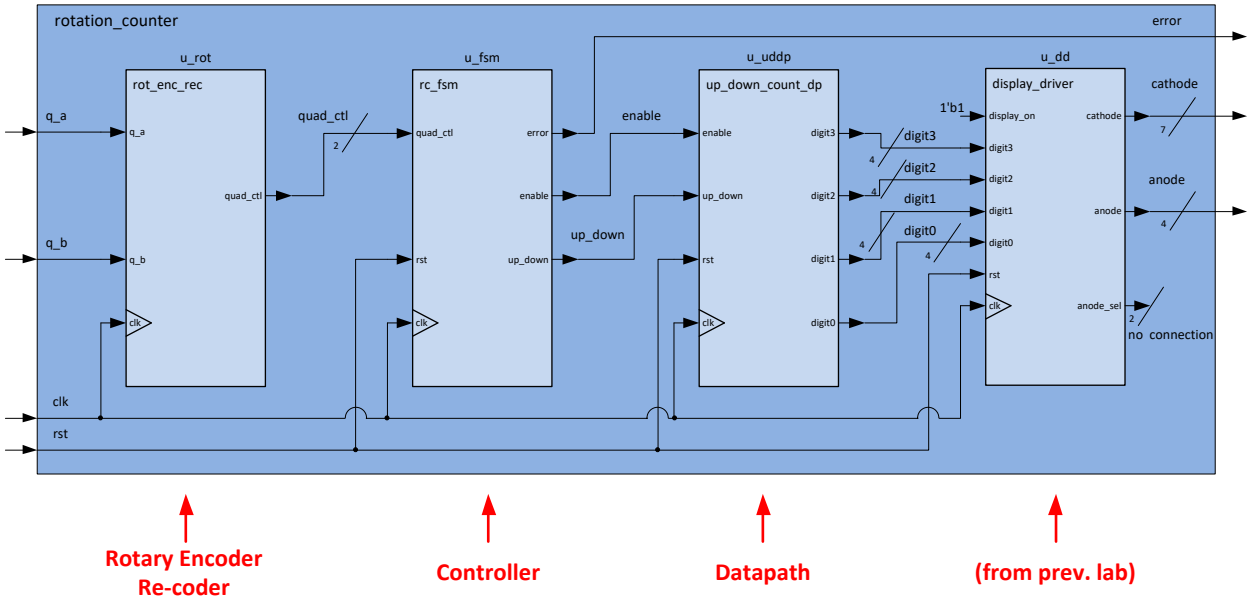
You can see they have been separated into four functional patterns:

- Hold (previous are equal to the current pattern)
- Count up (previous to current follows one gray pattern)
- Count down (previous to current follows the reverse gray pattern)
- Error (previous to current are illegal transitions – two bits transitioning)

q_prev_a	q_prev_b	q_now_a	q_now_b	quad_ctl	function
0	0	0	0	2'b00	Hold
0	1	0	1		
1	0	1	0		
1	1	1	1		
0	0	0	1	2'b01	Count up
0	1	1	1		
1	0	0	0		
1	1	1	0		
0	0	1	0	2'b10	Count down
0	1	0	0		
1	0	1	1		
1	1	0	1		
0	0	1	1	2'b11	Error
0	1	1	0		
1	0	0	1		
1	1	0	0		

The 2-bit output ***quad_ctl*** signal reflects these four possible conditions. Is this design *encoding* or *decoding*? I call it *re-coding*, since it takes an encoded signal and *codes* it into another code (***quad_ctl***) – or *re-codes* it. In the previous block diagram the combinational block called ***re-code*** provides this functionality. The top level re-coder module incorporating all of these specifications is called: ***rot_enc_rec***.

Now that we've covered the rotary encoder input flow, let's look at the design from the top level to create the functionality defined in the introduction. I call this top level block: ***rotatation_counter***, shown in the diagram below.

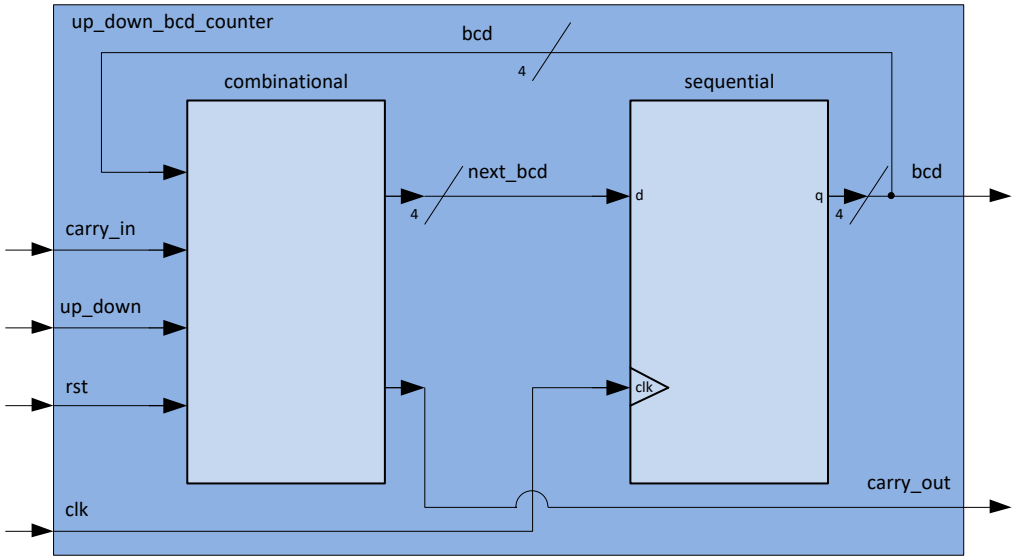


The input signals are the two quadrature signals (***q_a***, ***q_b***), the clock (***clk***) and a reset (***rst***) signal. The outputs are the cathode (***cathode***) and anode (***anode***) signals that drive the four digit display hardware. On the right you can see the instance of the ***display_driver*** that contains the seven-segment decoder and anode decoder you designed in a previous lab. An error signal (***error***) will also be connected to an LED. On the left side of the figure, you see the rotary re-coder instance discussed previously.

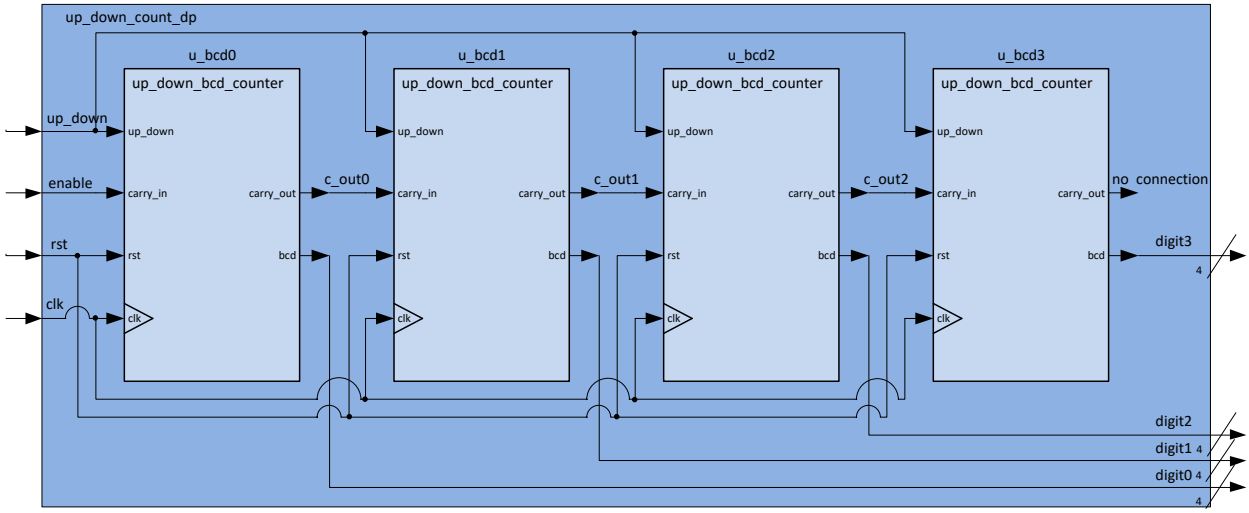
The other two components are a *datapath* and a *controller*. The controller (***rc_fsm***) is a *finite state machine* (FSM) that will be discussed in detail later. It basically takes the ***quad_ctl*** signals as inputs and provides two control output signals (***enable***, ***up_down***) to the counter datapath (***up_down_count_dp***). The counter datapath contains a four decimal digit up down counter. Let's discuss this module next.

Inside this datapath we need four binary coded decimal digits. A block diagram below shows a block diagram of *one* of these digits, called ***up_down_bcd_counter***. The specifications for the combinational logic are summarized in the table below. Note it counts up/down and rolls over at the nine/zero transitions.

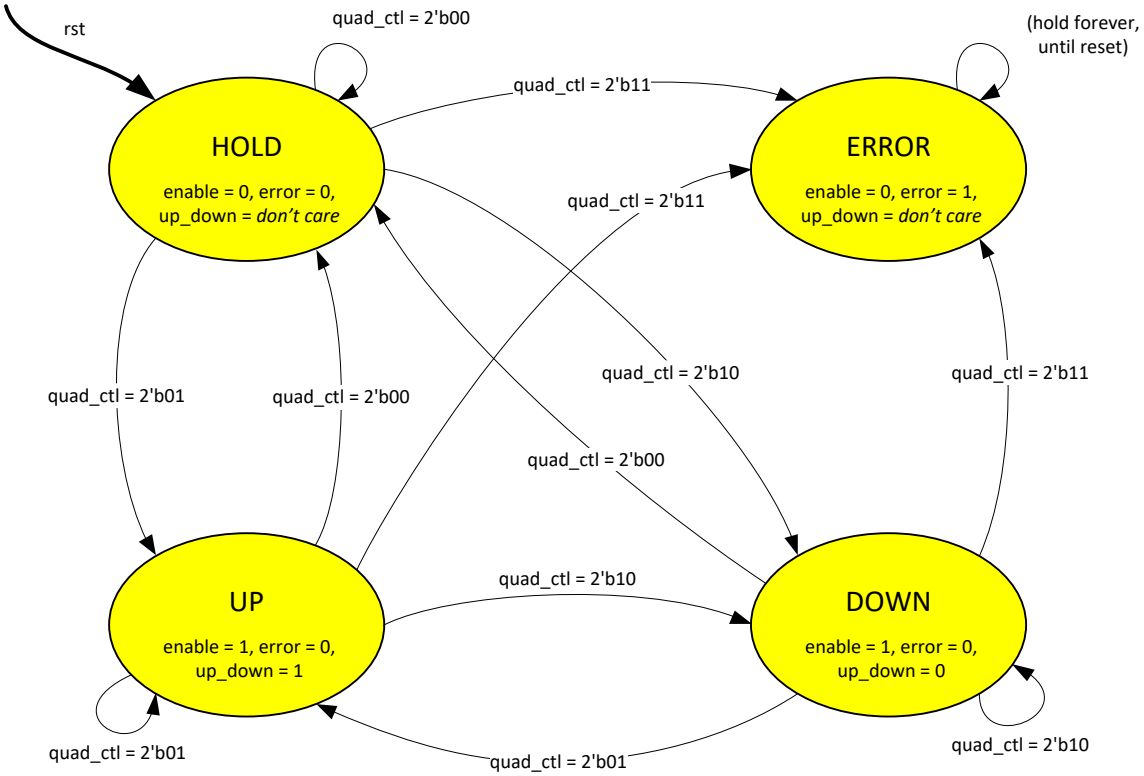
rst	*carry_in*	*up_down*	*bcd*	*next_bcd*	*carry_out*	**Function**
1	x	x	x	4'd0	x	*Reset*
0	0	x	4'd0 to 9	hold bcd	0	*Hold*
0	1	1	4'd0 to 8	bcd + 1	0	*Count up*
0	1	1	4'd9	4'd10	1	
0	1	0	4'd1 to 9	bcd - 1	0	*Count down*
0	1	0	4'd0	4'd9	1	



The up down count datapath module (***up_down_count_dp***) is relatively straightforward. We need one instance for each digit with the carry outs connected to the carry ins. (Counting down these become borrow outs/ins.) The up down, reset and clock signals are each wired together. These details are illustrated in the block diagram below.



Finally, we cover the specification for the *rotation counter fsm* module (***rc_fsm***). You can see its top level definition from its instantiation in the block diagram for ***rotation_counter***. There are three input signals (***clk***, ***rst*** and ***quad_ctl***) and four output signals (***error***, ***enable*** and ***up_down***). Their functionalities are defined in the following state diagram.



Note the four states that are controlled by the input signals ***quad_ctl***. Each state then controls the control output signals (***error***, ***enable***, and ***up_down***). Also note that to leave any state, all of the possible combinations of ***quad_ctl*** are defined. Finally, if you reach the ***ERROR*** state, there is no escape except for resetting.

Part A – Datapath

Design Steps

The rotary encoder re-coder (***rot_enc_rec***) module will be already provided to you in your design path. Review it and compare it to the block diagram and truth table provided in the specification section. Note the input flip-flop strings to synchronize and store the quadrature signals. Also note the re-coder combinational logic defined in the truth table, implemented as a case statement.

You need to complete the design of two modules:

- up down BCD counter (module ***up_down_bcd_counter***)
  Open the file: **up_down_bcd_counter.sv** and compete the design as per the specifications in the previous section. This is a single digit *binary coded decimal* counter that counts up or down from 0 to 9.
- up down count datapath (module ***up_down_count_dp***)
  Open the file **up_down_count_dp.sv** and complete the design as per the specifications in the previous section. You will be instantiating four of the *bcd* counters so that this datapath counts up or down from 0000 to 9999 in decimal.

Simulation/Verification

There is only one testbench for Part A, **tb_up_down_count_dp.sv**. You will be testing both modules with this testbench. The testbench reads the text file, **tb_up_down_count_dp.txt** and applies these test vectors in the sequence shown – one line per clock cycle. If you have mismatches, they should help you identify your error. It could be either in your counter or datapath where you are connecting the counters together. Remember, you can use *gtkwave* to see all of the signals from the simulation.

Synthesis

There is no synthesis for Part A. Just be sure your simulation passed without errors as these blocks will be used for Part B.

Part B – Controller

Design Steps

You need to complete the design of the rotation counter finite state machine (**rc_fsm.sv**). A skeleton module will be provided to you. You should follow the recipe for FSM design described in the class videos and summarized in an Appendix (*Controller Design Process*) at the end of this document.

Part of the rotation counter (***rotation_counter***) module will be already provided to you in your design path. You need to edit **rotation_counter.sv**. Instantiate and connect your datapath (***up_down_count_dp***) and FSM (***rc_fsm***) modules as per the block diagram of the rotation counter shown previously in this document. The other two modules (***rot_enc_rec*** and ***display_driver***) will already be instantiated.

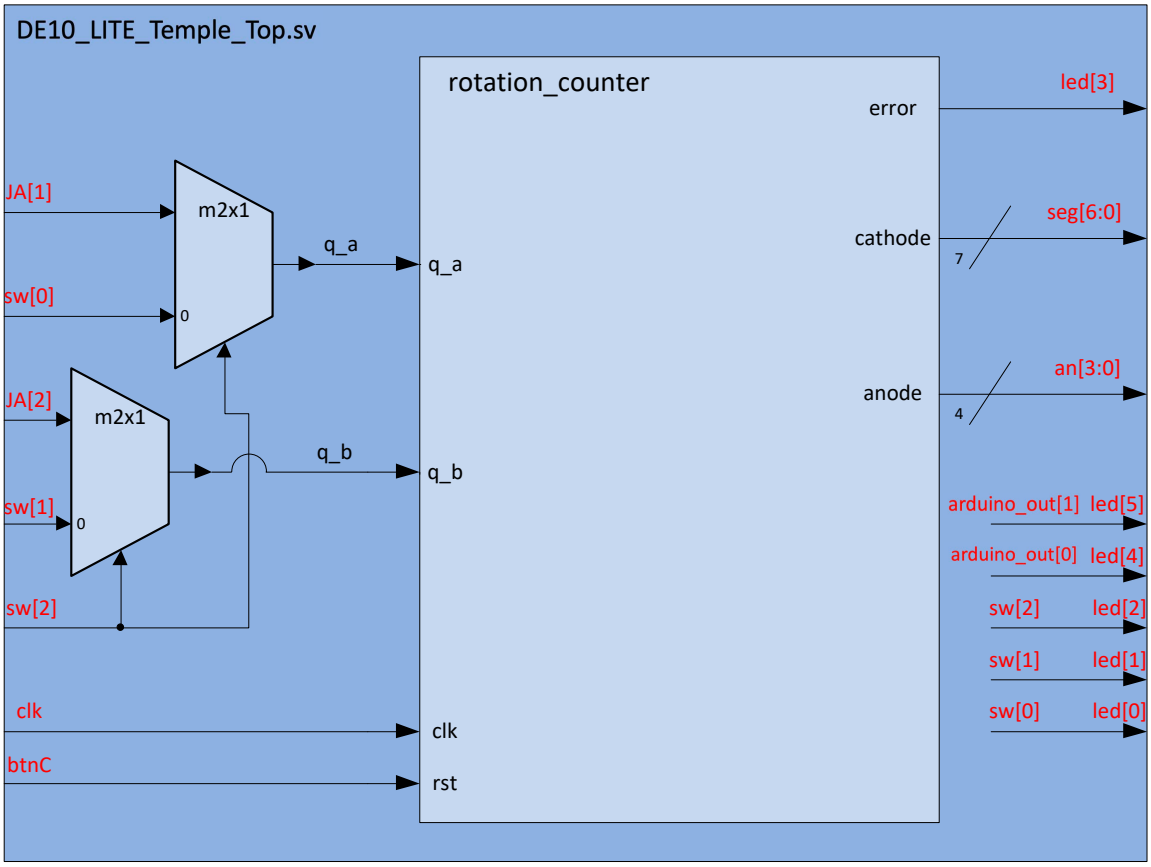
Simulation/Verification

There are two testbenches for Part B. The first testbench tests your FSM design (**tb_rc_fsm.sv**). This testbench reads text file, **tb_rc_fsm.txt**, one vector per clock cycle. If you have mismatches, the text file is well documented as to the sequence of steps the test is taking. You should be able to diagnose your design errors with this information.

The second testbench tests the top level rotation counter (**tb_rotation_counter.txt**). You should not have any problems with this as you are provided a completed ***rotation_counter*** design. If you do have mismatches, it could be due to your FSM design – since the FSM testbench does not test full functionality.

Synthesis and Hardware Test

A top level i/o wrapper is provided: **DE10_LITE_Temple_Top.sv**. A block diagram is shown below. Synthesize it (**lab7_top.qsf**).



As you can see from this design there are two paths to the quadrature inputs, ***q_a*** and ***q_b***. These are selected by ***sw[2]***. If ***sw[2]*** is off, the quadrature signals come from switches ***sw[1:0]***. If ***sw[2]*** is on, the quadrature signals come from the Arduino Uno R3 expansion header: ***arduino_out[1:0]***. You can test the

design using either of these inputs. As you switch between these inputs, it is possible you will get to the error state (LED 3 on). Then reset the state machine (center pushbutton). Note the other LEDs show the states of all of the input signals.

The rotary encoder should plug directly into the Arduino header. Identify the connector on your hardware. The rotary encoder contains only switches connected with resistors, so you shouldn't damage anything if you don't connect it correctly.

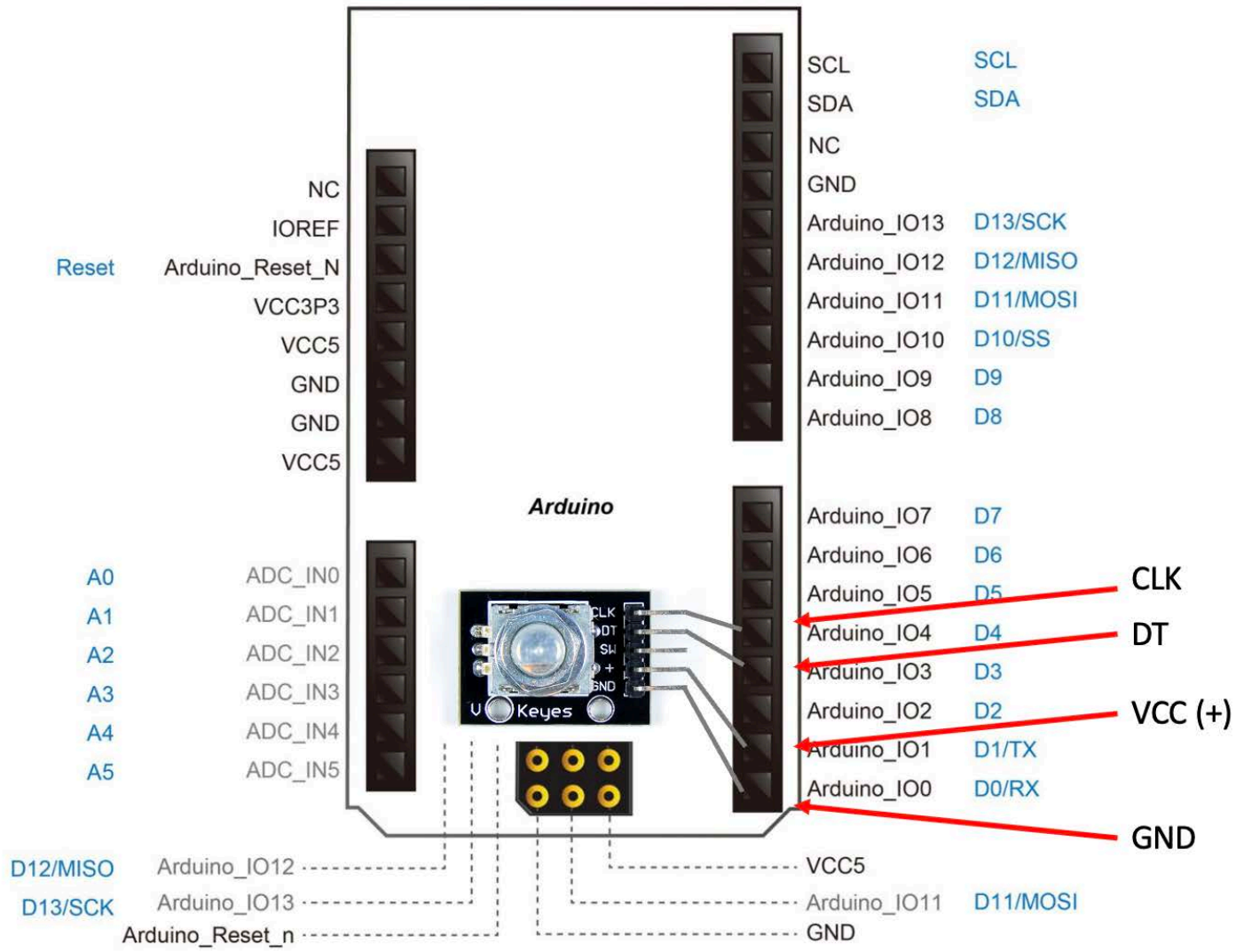


Figure 3-19 lists the all the pin-out signal name of the Arduino Uno connector. The blue font represents the Arduino pin-out definition.

It should plug directly into 4 pins as shown above (the ***sw*** connector on the encoder is not connected).

Each detent on the rotary encoder will go through four counts. There are 20 detents per rotation, or 80 counts per 360 degrees. You should be able to move carefully between the detents and see single digit changes.

Appendix – Controller Design Process

Step 1: Capture the behavior

- Create an FSM diagram describing this behavior
    - List all possible states, giving each an appropriate name;
    - Define all *state to state* transitions, and the conditions (inputs) for transitioning or holding;
    - Define all outputs at each state;
    - Mentally execute to verify the specifications met.

Step 2: Convert to circuit

- Create a Verilog module implementing the FSM
    - List inputs, outputs and determine the size of the state register (number of states) – define top level module i/o;
    - Define state names using **`parameter`** statement, or for SystemVerilog use the *enumerate* construct, example:
        - **`enum logic [1:0] {S0, S1, S2, S3} state, next_state;`**
    - Set up two **`always`** blocks – one sequential logic and one combinational logic (draw block diagrams if necessary);
    - Sequential logic has simple: **`state <= next_state`**;
    - Combinational logic has **`always_comb begin`**, and blocking procedural assignments;
    - Divide combinational logic into 3 sections: *default*, *main logic* and *priority logic*;
    - In *default* section define default behavior of all output signals and **`next_state`**;
    - In *priority* section (at the bottom) take care of the priority logic (typically resets);
    - In *main* logic section set up a case statement based on state, with each defined state having a matching case;
    - Insert the appropriate logic for each state in the *case* construct.