Lab 2 - Step Motor Controller

Liran Golan 311121073

Danie Mergy 342533627

Introduction

A Stepper motor is an electric motor that can rotate by crossing a sequence of unit little moves. Each unit of move is called a step. Each step is crossed by applying a specific voltage on the motor inputs. Therefore, in order to rotate in a defined direction, we have to cross a defined sequence of inputs on the motor.
More precisely, a step corresponds to a combination of voltage applied on coils that are present in the motor. Applying voltage to the coil creates a current flow that will induct a magnetic field that causes the "anchor" located in the center of the motor to rotate according to the voltage applied to the coils. This is what cause the rotation.
In this lab we used a bipolar motor, as presented in figure 1 that means that we have two coils as presented in figure 2.

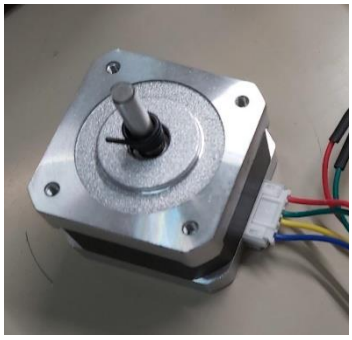


Figure 1 The bipolar motor we used

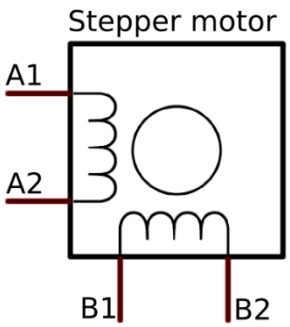


Figure1 The coils that are inside

We can control both sides of the coils to create current in opposite directions.
As we said before in order to rotate, we have a cross a specific sequence of steps.
We have two options:

-The first one is the Full step - We stream through one of the coils that cause the anchor to align with the magnet pull, after that, The current flows through the second coil causing the motor to move another step, and in the next phase, just like the first phase, only the current is reversed, and so is the last step. (total of 4 situations, we can see that in the table below State0-State2-State4-State6).

-The second one is the Half step - to the additional states we saw in the full steps, We will add intermediate stages for each step-by-step transition (total of 8 steps).
For example, between the transition from State0 to State2, we will add a new state called State1.
So, that the current flows through a coil and the second coil forming the "half rotation".

Table of the states:

A1	*A2*	*B1*	*B2*	המצב
0	0	0	1	***STATE0***
0	1	0	1	***STATE1***
0	1	0	0	***STATE2***
0	1	1	0	***STATE3***
0	0	1	0	***STATE4***
1	0	1	0	***STATE5***
1	0	0	0	***STATE6***
1	0	0	1	***STATE7***

*N*ote: in the bipolar motor every step performs a 1.8-degree rotation so, to do a complete turn we will need 200 steps for "Full step" (360-degree).

For "Half step", we will need 400 steps (we added the intermediate steps).

Next, if we want to execute a quarter of rotation - 90 degrees, we must do 50 steps for full step ((50 * 1.8 = 90 Degree and for half step 100 steps (adding intermediate steps))

Introduction to our project's design

The point of the project is to control the stepper motor. For this purpose, we able the user to press on Key0, Key1, Key3 and to switch Sw1, Sw2 and Sw3 on the Board. The combination of these inputs will define the progression of a sequence of states in a Final State Machine which will be connected to the stepper motor driver, which will define the level of voltage to apply on the motor itself.

Therefore, we implemented the motor unit, a component which will contain this FSM.

The half step feature led us to implement two different FSM, the first operating with 4 different states which operates when Sw3 in on and the second one with 8 different states operates when Sw3 is off. We could use the same FSM and make two steps in full step mode, but this was quite difficult, we focused on the simplicity. The order to make a step over is received in form of pulses, delivered by the speed_unit. We can pass step over in the FSMs in two different ways: we have a sequence of states; if sw1 is on we move forward in the sequence; if sw1 is off we move backward. The last user input motor_unit gets is Sw2 : it defines if we operate in continuous mode , the motor does not stop to rotate (when Sw2 is on) or if we are in 90deg mode which stop the motor and allow it to make a quarter of rotation at each press on Key1.

Now we know how the steps are passed we can ask how fast are they passed?

For this purpose, we implemented the speed_unit: we give it the clock of the board (27mHz) and there is a counter which let pass a single pulse of this clock after a while. By controlling the value at which the count reaches the trigger, we define the speed of the motor. We must remark that we had to give him also Sw3 as input because we wanted to keep the same speed with full/half step: if we are in half-step mode we have to pass *2 steps so in this case we divided the Max Value of the counter by 2. The user can also change the speed by pressing on key3, it makes one step over in a FSM with values 10-20-30-40-50-60 and go down to 50.... We added a Reset Button to reinitialize the speed to be 10 turns by minute. The first digit of the number which represent the actual speed is then send to the seven_sig_traductor.

The seven_sig_traductor converts the received number in binary in 7 segment and send it to the second digit display on the board. Moreover, it sends zero in 7 segments to all the other displays.

Our last component is quarter rotation: this feature was difficult to implement so we decided to make a self-component for it. The main idea is to produce a long pulse which will stay up only the time the motor_unit must make a quarter of rotation. A quarter of rotation is equals to 90 degrees: in our stepper motor 1 step makes 1.8 degree so a whole rotation is

equal to 200 steps if we are in full mode and 400 if we are in half step. Furthermore, a quarter of rotation will be equal to 50 or 100 steps to do. We will take the speed of steps by the pulses coming from the speed_unit. In a such way, we will count theses pulses until 50 or 100 following half/full mode: A press on Key1 launch the counter and the output will be at 1 until the count will finish.

About the difference between Continuous and 90deg mode we will be able to makes steps over in the FSM only if we are in continuous mode or if we are in 90deg and if the pulse is up.

Original Design schematics and changes during the implementation

This is the design we thought to implement in the beginning of the project:

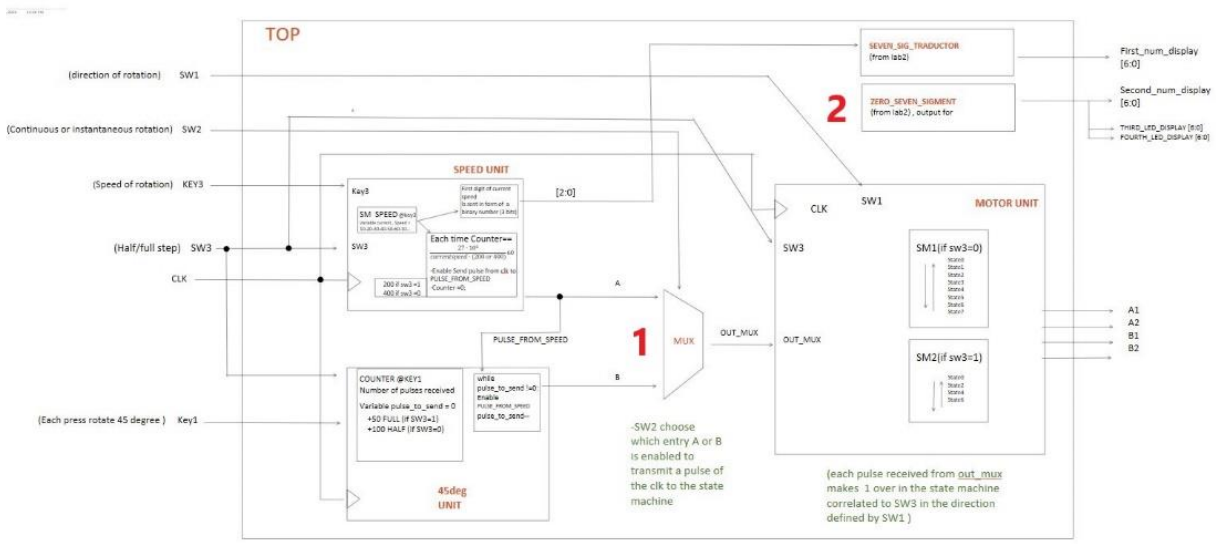


Figure 1 original design of the project

We planned to “copy” the signal of pulses coming from speed unit in the 90deg unit for an amount of time corresponding to execute a 90 deg rotation and then select in a mux controlled by SW2 if we send to the motor unit the pulses that are continuously generated or if we send what have been synthesized in 90deg unit. Then we were detecting the pulses by inserting the system clock (27Mhz) in the motor_unit.

In the tests we observed that working with the system clock was very difficult, the pulses were really shorts it was very hard to detect them and what we planned to “copy” the pulses also didn’t work.

Therefore, we changed the design and made the following changes:

we planned that the motor unit will be directly sensible to the pulses coming from the speed unit and not the system clk. Moreover, we removed the mux and we changed the purpose of 90deg unit ('1' in Figure 1) : We understood that it was easier to synthetize a long pulse corresponding to an enable flag limited in time than copying a signal. Another little thing: if we initially thought about making a separate module to display the zeros in the board we finally implemented this feature in the seven_seg_traductor to minimize the number of components ('2' in Figure 1).

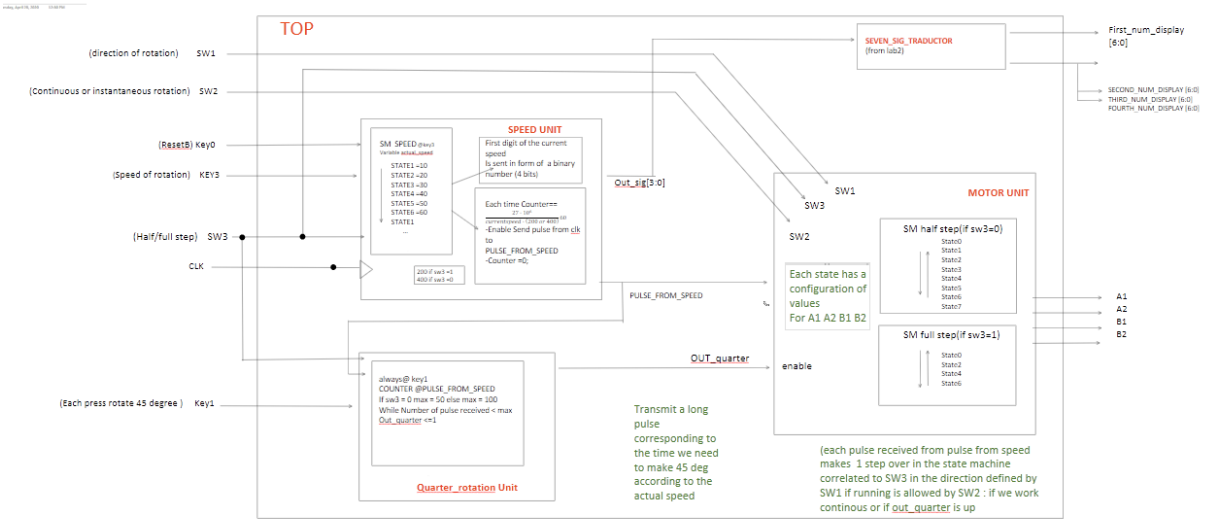


Figure 2 final design of the project

And this is what we finally implemented in Quartus :

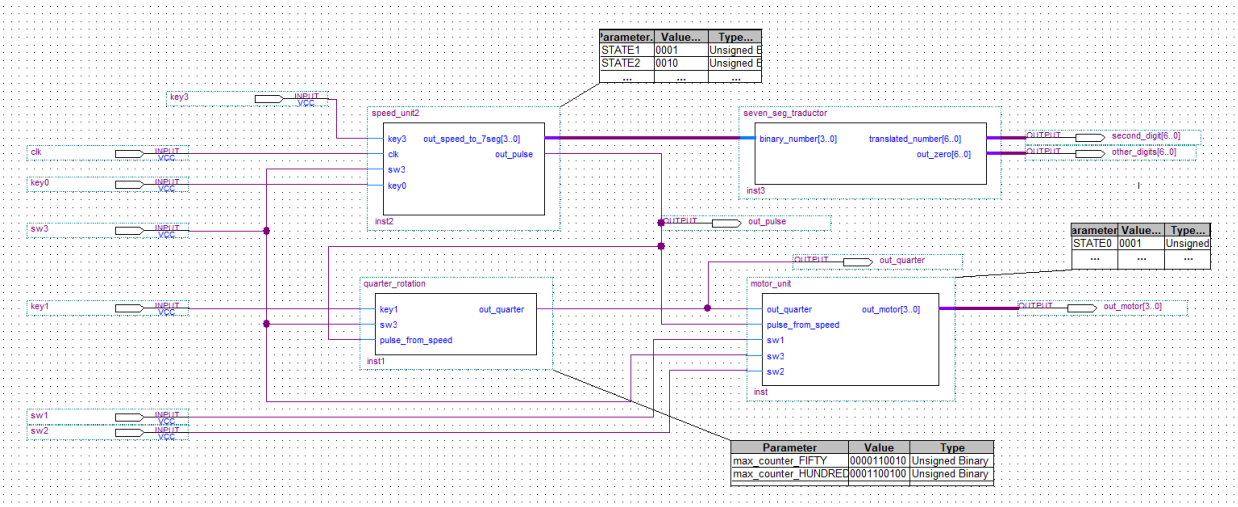


Figure 3 Quartus implementation of the final design

Detail of the components included in the top level

As we detailed in the design part, we have 4 components. We will explain precisely what their I/O signals are and briefly their operation.

Speed_unit2

This component is used to make pulses at the desired speed by the user, by sampling the clock at different intervals.

INPUTS

- Key3: Button input come from the user on the board, 1 all-time the button is not pressed, 0 when the user presses it. This button is pressed by the user to change the speed of the pulses according to the current speed : speed is initialized at 10 rotations by minute and increase by 10 every press on key3 until we reach 60 so the speed decrease by 10 until we reach 10 rot/min and we increase again..

- Clk: Pulses with frequency of 27Mhz all time the system run coming from the board itself.

- Sw3: Switch button controlled by the user on the board, constant 0 or 1, can be assimilated as a mode of operation controller. Mode '1' makes pulses twice lower than mode '0'.

- Key0: Button input, signal like key3. By pressing this button, the user reset the speed to 10 rotations by minute.

OUTPUTS

- Out_speed_to_7_seg[3..0]: 4 bits constant signal updated at each change of speed (ed press on key3) , represent the value of the first digit of the actual speed value (0,1 ,2 ,3 ,4 ,5 ,6).
  We could use 3 bits to send this information, but we took the component we designed for the first project.

- Out_pulse: Pulses at a frequency defined by the following equations:

$$T_{fullstep} = \frac{27 * 10^6}{actualspeed * 200} \cdot 60 \; if \; sw3 = 1 \; or \; T_{halfstep} = \frac{27 * 10^6}{actualspeed * 400} \cdot 60 \; if \; sw3 = 0$$

Remark: we designed the component to be used with a clock frequency of 27 Mhz , we cannot use the 50Mhz signal for example , without changing the internal values.

Quarter_rotation

This component is used to “say” to the motor unit that he has to perform a single 90 degree rotation on the stepper motor : it provides a specific interval of time when the motor_unit could run across its FSM.

INPUTS

- Key1: Button input come from the user on the board, 1 all-time the button is not pressed, 0 when the user presses it. This button is pressed by the user to give the order to realize a quarter rotation.
-
- Pulses_from_speed: Pulses with variable frequency coming from Speed_unit2.

- Sw3: Switch button controlled by the user on the board, constant 0 or 1, can be assimilated as a mode of operation controller. We need it to the output could remain the same time in two modes of operations to sw3 : if we have pulses twice the frequency we need to enable half the time.

OUTPUTS

- Out_to_motor: Long pulse of time defined by the following equations:

  $T = 50 \cdot T_{pulsesfromspeed}$ if sw3=1

  $T = 100 \cdot T_{pulsesfromspeed}$ if sw3=0

  This pulse is a signal that can be understood by the motor_unit as enable its FSM to run.

Seven_seg_traductor

This component “translates” a 4 bits bus corresponding to a binary number into one of the 7 segment display inputs of the board and set the other segments to display zero.

INPUT

- Binary_number [3..0]: bus of 4 bits corresponding to a binary expression of numbers 0,1,2,3,4,5,6

OUTPUTS

- translated_number [6..0]: bus of 7 bits corresponding to a seven segment expression of the input number.
- Out_zero [6..0] : bus of 7 bits corresponding to a seven segment expression of the zero number.

<u>Motor_unit</u>

This component collects the signals given by “speed_unit2” and “quarter_rotation” modules to provide a specific combination of 4 bits changing in the speed defined by the user.

We implemented this kind of combination as a “state” in two different FSM.

INPUTS

- Pulses_from_speed: Pulses with variable frequency coming from Speed_unit2.

- Out_quarter : Long pulse coming from module “Quarter_rotation”

- Sw1 : Switch button controlled by the user on the board, constant 0 or 1, can be assimilated as a mode of operation controller : used to define the order (normal if sw1 is on , inverse if sw1 is off) of the state sequence.

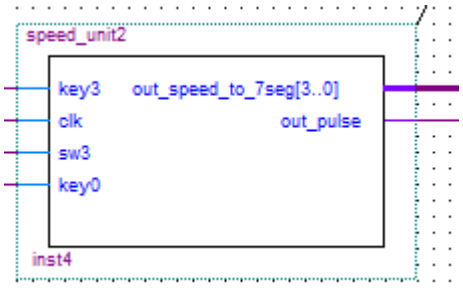
- Sw2: Switch button controlled by the user on the board, constant 0 or 1, can be assimilated as a mode of operation controller : when sw2 is on , the output has to change constantly (continuous mode) and when sw2 Is on , we only allow it to change the time he have to execute a quarter of rotation

- Sw3: Switch button controlled by the user on the board, constant 0 or 1, can be assimilated as a mode of operation controller: used to define how many states we must cross in the FSM to realize a rotation sequence.

OUTPUTS

- Out_motor: The specific 4 bits combination corresponding to the motor driver input which will convert the digital signal into analog corresponding to the voltage polarization of the coils in the motor.

Code and tests



1. **Speed unit**

The following code details the implementation of the unit:

```
1  module speed_unit2(key3,out_speed_to_7seg,out_pulse,clk,sw3,key0);
2
3  input wire key3, clk, sw3,key0;
4
5  output reg [3:0]out_speed_to_7seg;          // value between 1 to 6 in binary sended to 7_seg display
6  output reg out_pulse;                       //pulse sended to the motor_unit
7  reg [19:0] speed_of_pulses;                 // max value (trigger) of the counter
8  reg [19:0] count ;                          // variable of the counter
9  reg [5:0] actual_speed;                     // state defined by the second always to change the speed in the first one
10 reg direction;                              // flag indicates if we are increasing or decreasing "actual_speed"
11
12
13 //=========================================parameters==============================================================//
14
15 parameter
16             STATE1= 4'b0001, STATE2= 4'b0010, STATE3= 4'b0011,
17             STATE4= 4'b0100, STATE5= 4'b0101, STATE6= 4'b0110;
18 parameter
19             SPEED0 = 6'b000000 ,SPEED10= 6'b001010, SPEED20= 6'b010100, SPEED30= 6'b001110,
20             SPEED40= 6'b101000, SPEED50= 6'b110010, SPEED60= 6'b111100;
21
22 //=========================================initialization==========================================================//
23
24 initial
25     begin
26     count = 0;
27     actual_speed = SPEED10;
28     out_speed_to_7seg = STATE1;
29     direction =0;
30     if (sw3) speed_of_pulses      = 20'd810000; //initialization with speed = 10 rot/min
31     else if (~sw3)speed_of_pulses = 20'd400050;
32     end
33
```

```
35 //=========================================First Always============================================================//
36
37 /*contains the changing counter of the clock to manage the pulse frequency according to the SPEED state "actual spped" */
38
39 always@(posedge clk)
40     begin
41
42     if ( count >= speed_of_pulses) // counter
43         begin
44         count <= 0;
45         out_pulse<=1;
46         end
47     else
48         begin
49         count <= count +1 ;
50         out_pulse <=0;
51         end
52
53     case(actual_speed)
54         /* state machine that define the value at which the counter is triggered and send pulse
55             the values were computed by the following eq : (Fclk*60)/(actual_speed*(200 or 400))
56             when // sw3 on : full step (200) , sw3 off : half step (400)*/
57
58                 SPEED10: speed_of_pulses <= sw3 ? 20'd810000 : 20'd405000;
59                 SPEED20: speed_of_pulses <= sw3 ? 20'd405000 : 20'd202500;
60                 SPEED30: speed_of_pulses <= sw3 ? 20'd270000 : 20'd135000;
61                 SPEED40: speed_of_pulses <= sw3 ? 20'd202500 : 20'd101250;
62                 SPEED50: speed_of_pulses <= sw3 ? 20'd162000 : 20'd81000;
63                 SPEED60: speed_of_pulses <= sw3 ? 20'd135000 : 20'd67500;
64                 default: speed_of_pulses <= sw3 ? 20'd810000 : 20'd405000;
65
66         endcase
67
68
69     end
70
```

Comments on the code: 1-in line 42 we had to use the >= operator which is a heavy implementation in the FPGA because we have not success to find a way only using '==’. 2-We think that there is a way to use out_speed_to_7_seg as the index of the states to simplify the code but we have not found it.

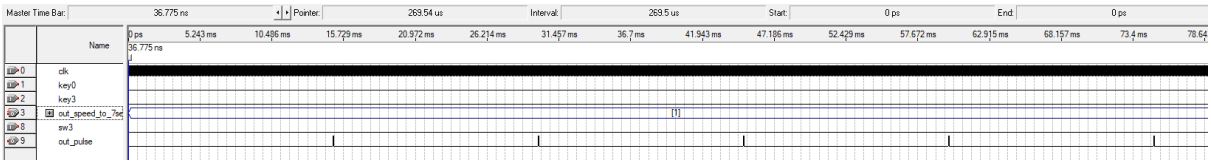
```
//=========================================Second Always==========================================================//

/*change the speed_state and the value sended to the display according to user input key0 and key3.*/

always@(negedge key3 or negedge key0)

        begin
        if(~key0)
            begin
            actual_speed = SPEED10;
            out_speed_to_7seg = STATE1;
            direction =0;
            end
        else if (~key3)
                begin
                case(actual_speed) // state machine to control the speed and the display
                SPEED10:    begin
                            actual_speed = SPEED20;
                            out_speed_to_7seg = STATE2;
                            direction =0;
                            end
                SPEED20:    begin
                            if (~direction)
                                begin
                                actual_speed = SPEED30;
                                out_speed_to_7seg = STATE3;
                                end
                            else if (direction)
                                begin
                                actual_speed = SPEED10;
                                out_speed_to_7seg = STATE1;
                                end
                            end
                SPEED30:    begin
                            if (~direction)
                                begin
                                actual_speed = SPEED40;
                                out_speed_to_7seg = STATE4;
                                end
                            else if (direction)
                                begin
                                actual_speed = SPEED20;
                                out_speed_to_7seg = STATE2;
                                end
                            end
                SPEED40:    begin
                            if (~direction)
                                begin
                                actual_speed = SPEED50;
                                out_speed_to_7seg = STATE5;
                                end
                            else if (direction)
                                begin
                                actual_speed = SPEED30;
                                out_speed_to_7seg = STATE3;
                                end
                            end
                SPEED50:    begin
                            if (~direction)
                                begin
                                actual_speed = SPEED60;
                                STATE5: out_speed_to_7seg = STATE6;
                                end
                            else if (direction)
                                begin
                                actual_speed = SPEED40;
                                STATE5: out_speed_to_7seg = STATE4;
                                end
                            end
                SPEED60:    begin
                            if (~direction)
                                begin
                                actual_speed = SPEED50;
                                out_speed_to_7seg = STATE5;
                                direction =1; //direction 0 is: forward , direction 1: is backwared
                                end
                            end
                default:    begin
                            actual_speed = SPEED10;
                            out_speed_to_7seg = STATE1;
                            direction =0;
                            end
                endcase
                end
        end
endmodule
```

As we said before the speed unit must sample the clock of 27 Mhz at constant intervals. We first decided to run a simulation with the clock of 27Mhz to check if we have no problems of timing. Assuming sw3 if off when the speed is initialized to be 10 rot/min the counter has

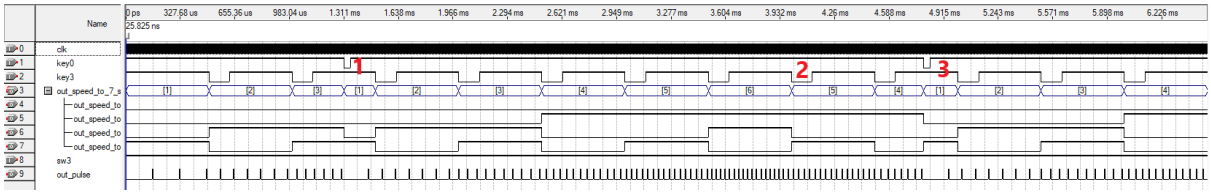
405000 pulses to count. So if we multiply it by the time T clock we get the time between two pulses:

$$T_{pulse} = \frac{1}{27 * 10^6} \cdot 405000 = 15ms$$

Indeed, we can observe on the following test result that there is 15ms between each pulse.



We changed the Max values of the counter and the clock frequency to get a reasonable pattern:



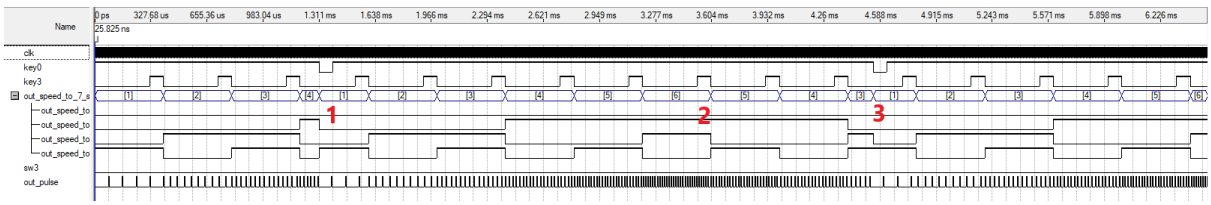
We will explain how this test shows that the desired features are accomplished.

We explained in the design part that the main idea of controlling the speed was to send pulses at different intervals to the motor unit. On the test this corresponds to “out_pulse”: the intervals become straighter each time we press key3. Moreover, this increase the number which will be displayed, in the test this corresponds to out_speed_to_7_segment.

As we already explained, pressing on Key0 reinitialize the interval to the default value corresponding to 10 rot/min. Another feature is that each press on key3 leads to increase the speed to 60 and then decrease. In the test we can see on it in the “2”: out_speed_to_7_segment goes 5 after being 6.

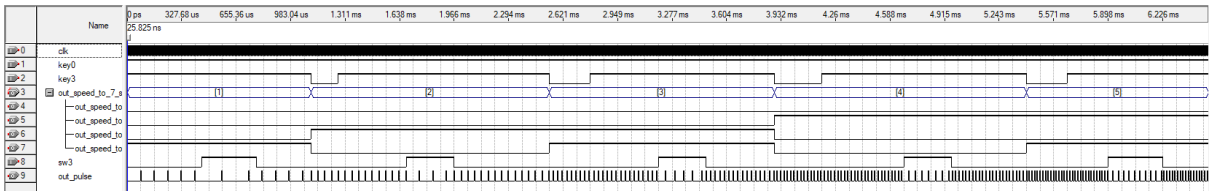
For the press on key0, it works both when we increase or when we decrease as we marked in “1” and “3”: out_speed_to_7_segment goes 1.

We asked what the output look like if the user press on key3 most of the time. We saw that it works, and we decided to bring here the results with another scale which shows more explicitly the variable interval between the pulses:



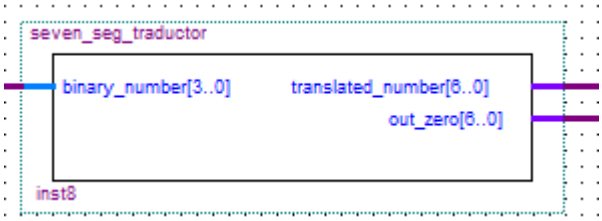
Another Feature is changing the direction of rotation we decided to make it a proper test to not miss any undesirable behavior.

The design says that with sw3 = 1 we must 2 times the trigger value than with sw3 =0. Indeed, it corresponds to an interval 2 times larger:



We can see on the test that the interval of “out_pulse” is 2 times larger only when sw3 on.

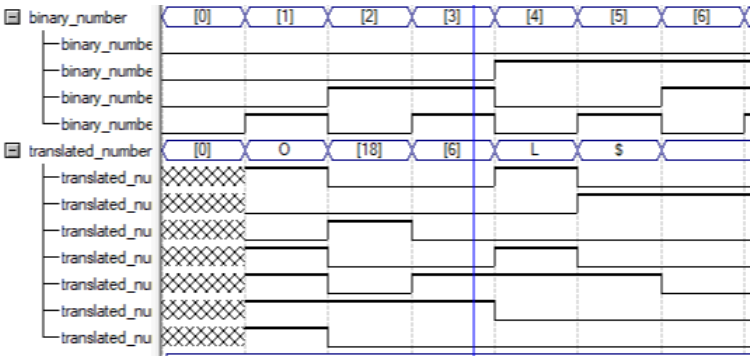
2. Seven_sig_traductor



This component is a traductor between binary input of a number and the segments that have to be tuned on so the user could read it : it's printed as decimal number on the board display.

There is no FSM , only direct assignment so we just checked that for each binary number the output corresponds to the segment that have to be turned on according to the following table:

Decimal Digit	Individual Segments Illuminated						
	a	b	c	d	e	f	g
0	×	×	×	×	×	×	
1		×	×				
2	×	×		×	×		×
3	×	×	×	×			×
4		×	×			×	×
5	×		×	×		×	×
6	×		×	×	×	×	×
7	×	×	×				
8	×	×	×	×	×	×	×
9	×	×	×			×	×

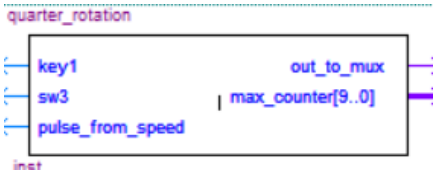


We can see in the test that when a bit of the output is down corresponds to a cross in the table.

```
module seven_seg_traductor (binary_number , translated_number, out_zero);

    input wire [3:0] binary_number;
    output reg [6:0]translated_number ;
    output reg [6:0]out_zero ;


    always@ (*) begin
    out_zero =7'b0000001;
    if (binary_number == 4'b0001)
    //one
        translated_number = 7'b1001111;
     else if (binary_number == 4'b0010)
    //two
        translated_number = 7'b0010010;
    else if (binary_number == 4'b0011)
    //tree
        translated_number = 7'b0000110;
     else if (binary_number == 4'b0100)
    //four
        translated_number = 7'b1001100;
    else if (binary_number == 4'b0101)
    //five
        translated_number = 7'b0100100;
    else if (binary_number == 4'b0110)
    //six
        translated_number = 7'b0100000;
    end

endmodule //seven_seg_traductor
```

3. **Quarter_rotation**:

We will present the code for the module:



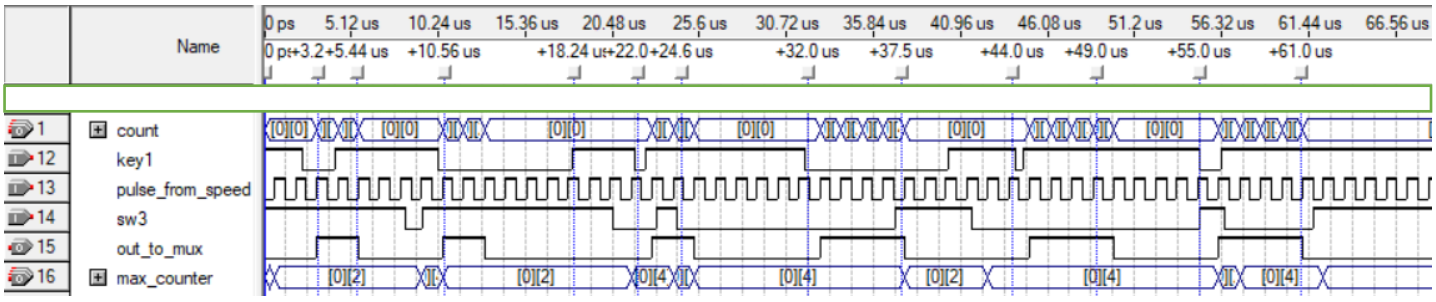

```
module quarter_rotation(key1,sw3,pulse_from_speed,out_to_mux,max_counter);


// -------------------------------------Definition---------------------------------//

input wire key1,sw3,pulse_from_speed;
reg enable_count,end_count;
reg [9:0] count;
output reg out_to_mux;
output reg[9:0] max_counter;

// -------------------------------------Parameters---------------------------------//
parameter max_counter_FIFTY = 10'd50;
parameter max_counter_HUNDRED= 10'd100;

// -------------------------------------Initial------------------------------------//
initial
    enable_count=0;
initial
    count = 0;
initial
    end_count=0;
initial
    max_counter <=10'd0;

// -------------------------------------Logic--------------------------------------//
```

```
// -------------------------------------Logic--------------------------------------//


//enable_count is a reg that says when we able to count or not
//              when enable_count =1 we can start to count
//              when enable_count =0 stop to count trigger
// max_counter -> the max number we can count
always@(negedge key1 or posedge end_count)
    begin
    if (end_count)
        begin
            enable_count <=0;
        end
    //the max counter get 50/100 steps depeant on full/half step
    else if (~key1)
        begin
            enable_count <=1;
        end

    end


//In the first condition, we check if we should count
//The second condition, when we finish counting we reset the registers
always @(posedge pulse_from_speed)
        begin
            max_counter <= 4'd4; //max_counter_HUNDRED

            if (sw3)
            begin
            max_counter <= 4'd2; //max_counter_FIFTY
```

```
//In the first condition, we check if we should count
//The second condition, when we finish counting we reset the registers
always @(posedge pulse_from_speed)
        begin
            max_counter <= 4'd4; //max_counter_HUNDRED

            if (sw3)
            begin
            max_counter <= 4'd2; //max_counter_FIFTY
            end
            if (count < max_counter & enable_count == 1)
                begin
                    out_to_mux <= 1;
                    count <= count +1;
                end
            else
                begin
                out_to_mux <=0;
                end_count <= (count == max_counter);
                if (count == max_counter) count <=0;
                end
        end

endmodule
```

Remark: We forgot to change the name of the output in the code and the tests of this component : out_mux was for our old design, the real name in the top level is out quarter.

Simulations:



Explanation of the simulation:

To make sure that the rotation is correctly done, we note that the number of system inputs is relatively small. Therefore, we will cover the times listed by constructing the following table when the times in [us] unit so that we can verify that the system is correct.

	Sw3(1)	Sw3(0)	Sw3(from 0 to 1)	Sw3(from 1 to 0)
Key1(short)	3.2-5.44	44-49	22-24.6	55-61
Key1(long)	10.56-18.34	32-37.5		
Key1 (1)	62-66	57-61		

<u>We will explain the table parameters:</u>

-Key1(short) is a single click

-Key1 (long) is a push button

Sw3 (1) is a full-step mode-

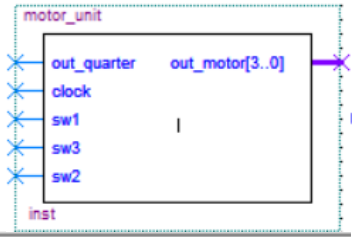
-Sw3 (0) is a half-step state

In the simulation we have added auxiliary parameter max_counter that specifies the number of pulses we have to count, depending on half/full step.

So that we can show the results in the simulation, we substituted the original value of the max_counter parameter as seen in the simulation and code, for full step we set the count to 2, and for half a step we set the count to 4.

Here is a sample run from the table: At times between 10.56 [us] -18.34 [us] there is a long press when Sw3 = 1 meaning we are at full step and expect a maximum count to 2. As seen in the simulation counter, 2 pulses were counted, and the count does not end the out_to_mux parameter stay on 1. We note that the long click performs the counting operation once as required and like a single click.

4. **Motor unit**



We will present the code for the module:

```
module motor_unit(out_quarter,out_motor,pulse_from_speed,sw1,sw3,sw2);

input wire pulse_from_speed , sw1, sw3,sw2,out_quarter;
output reg [3:0]out_motor;

initial out_motor = 0;

parameter   STATE0= 4'b0001,
            STATE1= 4'b0101,
            STATE2= 4'b0100,
            STATE3= 4'b0110,
            STATE4= 4'b0010,
            STATE5= 4'b1010,
            STATE6= 4'b1000,
            STATE7= 4'b1001;
```

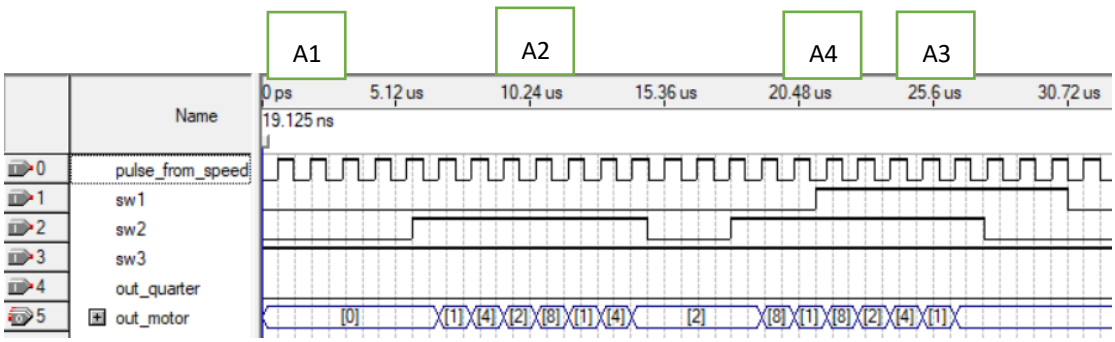
```
always@(posedge pulse_from_speed)
    begin
        if(sw2==1 | (sw2==0 & out_quarter==1))
            begin
            if (~sw3)//halfstep
                begin
                case (out_motor)
                    STATE0: out_motor <=(~sw1) ? STATE1:STATE7;
                    STATE1: out_motor <=(~sw1) ? STATE2:STATE0;
                    STATE2: out_motor <=(~sw1) ? STATE3:STATE1;
                    STATE3: out_motor <=(~sw1) ? STATE4:STATE2;
                    STATE4: out_motor <=(~sw1) ? STATE5:STATE3;
                    STATE5: out_motor <=(~sw1) ? STATE6:STATE4;
                    STATE6: out_motor <=(~sw1) ? STATE7:STATE5;
                    STATE7: out_motor <=(~sw1) ? STATE0:STATE6;
                    default:out_motor = STATE0;
                endcase
                end

            else if(sw3)//fullstep
                begin
                case (out_motor)
                    STATE0: out_motor <=(~sw1) ? STATE2:STATE6;
                    STATE2: out_motor <=(~sw1) ? STATE4:STATE0;
                    STATE4: out_motor <=(~sw1) ? STATE6:STATE2;
                    STATE6: out_motor <=(~sw1) ? STATE0:STATE4;
                    default:out_motor = STATE0;
                endcase
                end
            end
    end
```

Simulation num 1

To explain better what we did we will present each time a section. We noted that there are many transitions and so what we chose to do is to show that each state works individually which includes the transitions between them.
Then we show the transitions that we think are critical in the system (At the end we will show the table in full).

Time aprox[us]	Sw1	Sw2	Sw3	Quarter	Full step	Half step
	0	0	0	0		C1
	0	0	0	1		D1
0-5.9	0	0	1	0	A1	
	0	0	1	1	B1	
	0	1	0	0		C2
	0	1	0	1		D2
5.9-14.8	0	1	1	0	A2	
	0	1	1	1	B2	
	1	0	0	0		C3
	1	0	0	1		D3
27.74-29.46	1	0	1	0	A3	
	1	0	1	1	B3	
	1	1	0	0		C4
	1	1	0	1		D4
21.15-23.74	1	1	1	0	A4	
	1	1	1	1	B4	

Each time, we look at the area marked by zone, which represents the following simulation

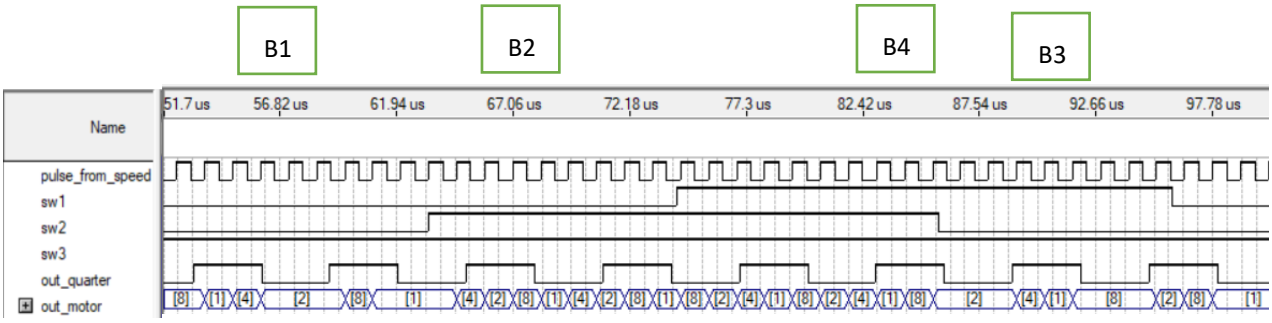


Explanation of purpose Shortly: beginning we wanted to analyze the Full Step mode. In this mode, there are 4 transitions in the state machine: 1-4-2-8. When sw2 is gets 1, it allows basic counting operation and so it can be seen that the mode machine progresses every time step 1 is operated (Example A2).

Also, we checked the progress of the mode machine in the opposite direction where Sw1 = 1 as seen in A4
That means the mode machine is moving in the opposite direction: 1-8-2-4
Besides, you can see when sw2 = 0 the system is not progressing as we expected (it is waiting for the trigger from out_quarter and since it is zero the machine is not progressing as seen in A1, A3.
Finally, we can see that we also checked the basic transitions between these situations.

<u>Simulation num 2</u>
The table:

Time[us]	Sw1	Sw2	Sw3	Quarter	Full step	Half step
52-54	0	0	1	1	B1	
65-67	0	1	1	1	B2	
89-91	1	0	1	1	B3	
83-85	1	1	1	1	B4	



Explanation of purpose briefly: We will continue to check the Full Step mode when we added the Quarter this time it should be noted that it does not have to be cyclical but here we chose it.

The main test here is to check when we get a pulse from out_quarter and when sw2 = 0, We want to see that the state machine is progressing as long as out_quarter is in position 1, as seen in B1 and B3.

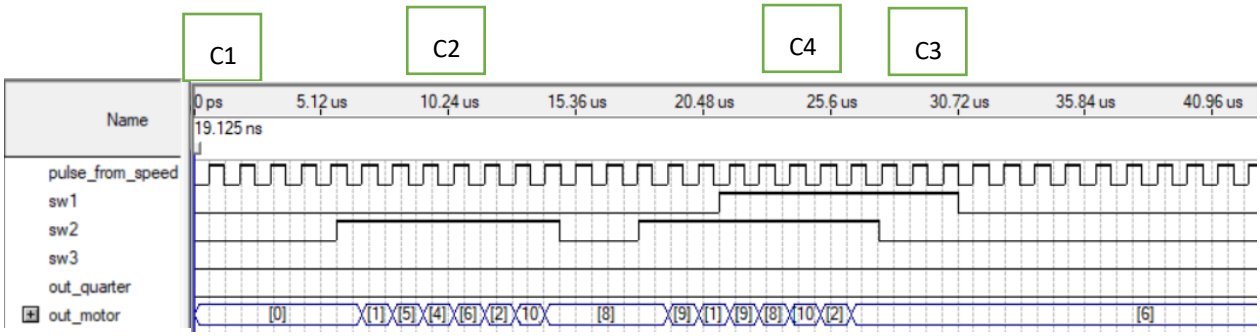
You can also see when sw2 = 1 is independent of the out_quarter variable and the state machine is progressing as usual. (B2, B4)

Also, we checked the transitions between modes and the correctness of progress of this machine in the opposite direction when Sw1 = 1

Simulation num 3

Time[us]	Sw1	Sw2	Sw3	Quarter	Full step	Half step
0-5.12	0	0	0	0		C1
5.5-14	0	1	0	0		C2
27.5-30.72	1	0	0	0		C3
21-27.5	1	1	0	0		C4

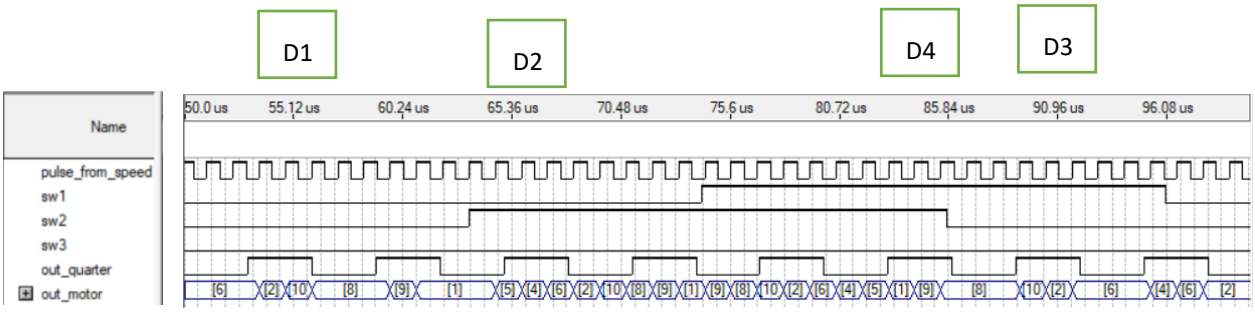
Each time, we look at the area marked by zone, which represents the following simulation



Explanation of purpose brief: Similar to the purpose of simulation number 1, only here we examine the Half step mode
We note that the clockwise mode machine (sw1 = 0) is 1-5-4-6-2-10-8-9
And the machine advances in the opposite direction (sw1 = 1) is 1-9-8-10-2-6-5-4 (c4)

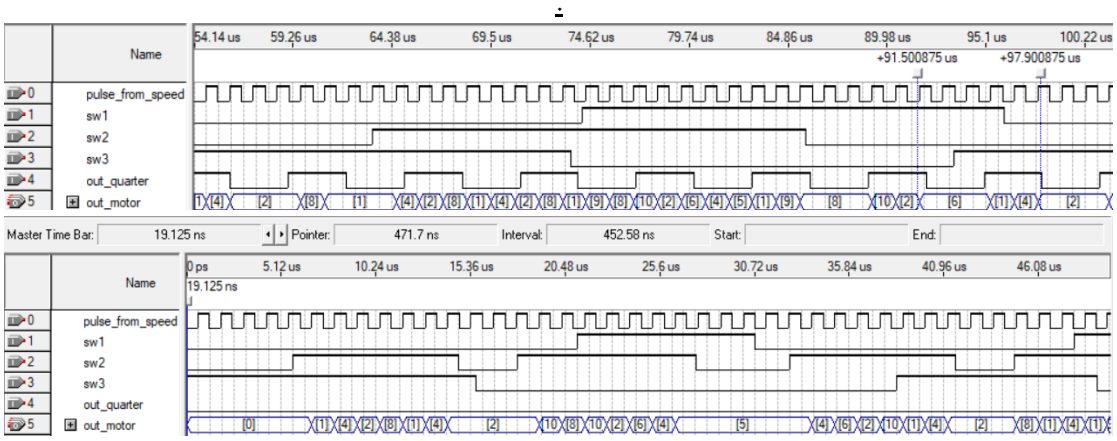
Simulation num 4

Time[us]	Sw1	Sw2	Sw3	Quarter	Full step	Half step
53-55	0	0	0	1		D1
65-67	0	1	0	1		D2
89-91	1	0	0	1		D3
83-85	1	1	0	1		D4



Explanation of purpose brief: Same goal as simulation number 2, when we work with Half step.

<u>We also examined the transitions between Full Step and Half Step in both directions</u>



<u>Purpose of the simulation</u>: After examining the various modes half and separately, we wanted to examine the transitions between these steps.Recall the mode maker 1-5-4-6-2-10-10-8-9 (green modes are suitable for half a step while full step contains all transitions).When we switch between a particular mode after you can see that the mode machine is functioning properly. When the simulation below is without out_quarter and the upper one with. The transitions can be seen when the sw3 changes, as the state machine depending on the type of step you can see.

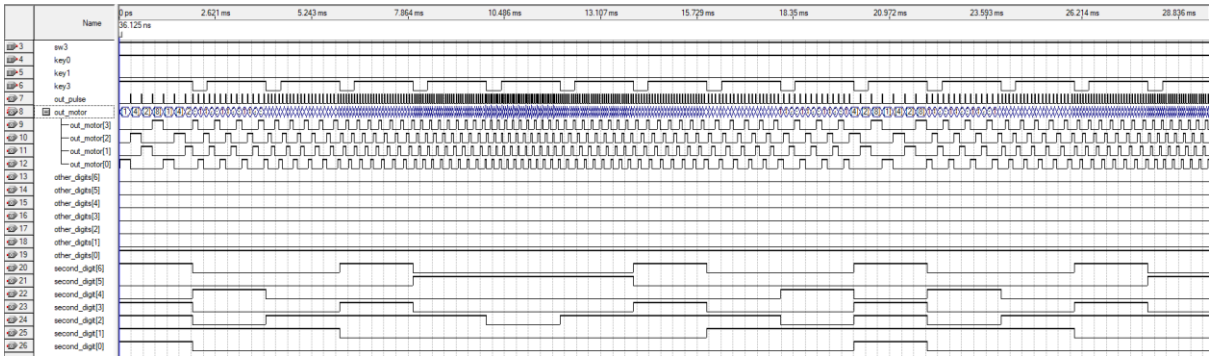
Example: In a time of 15.36 [us] switch from full step to half step and at about 37 [us] switch from half step to full step.

Note: There can be a situation where we move from the full step parade machine to our directness is in the half step mode, such a condition the value shared receives is the value in the boot that we set in the software which is the first half step condition.

Test of the top level

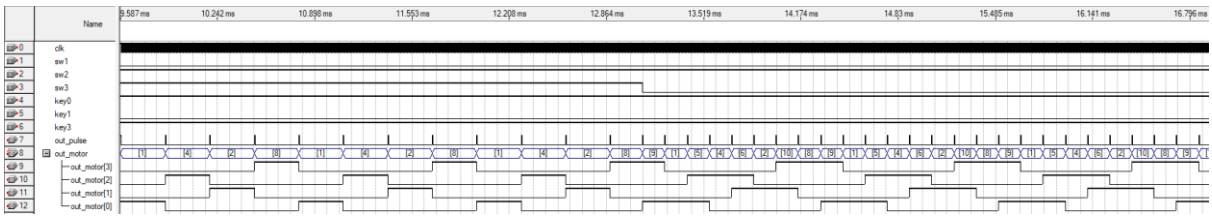
We reduced the trigger values of the variable counter of the speed_unit the same way as its test to allow our tests to be shorter.

In first place we launched the test with presses on key3 : we can see on the following test that the interval between the pulses (out_pulse) vanishes according to our purpose and that “out_motor” which is the final output of our system is driven by this signal. Setting 30ms with 2ms between each press permitted us to see the speed rising 60 and decreasing before increasing again. Moreover, we can see that the display on the board works properly, changing at each press on key3.



For now, we will test the half step / full step

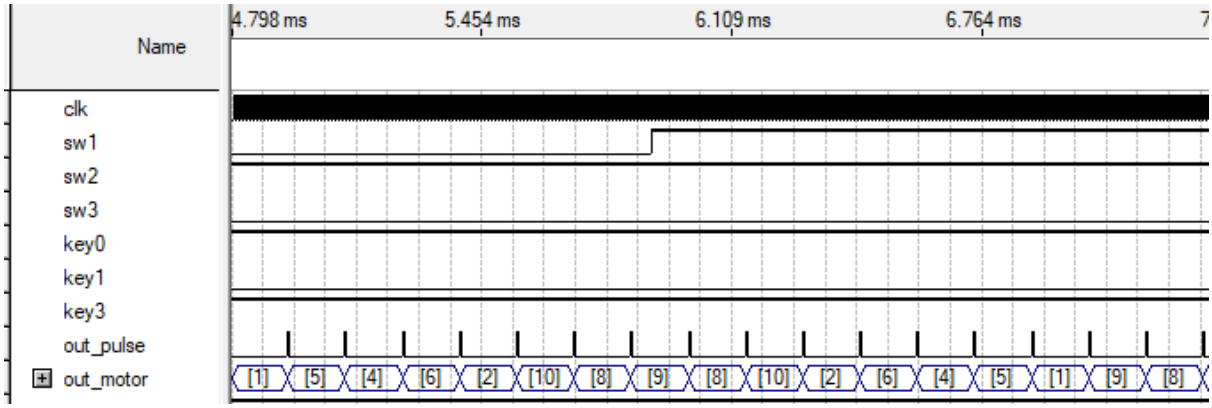
Setting full step mode for the half of the test (until ~13ms):



We can see that the sequence of out_motor is 1-4-2-8 but after passing to full step we have 1-5-4-6-2-10-8-9. These values match with the voltage configuration presented in the introduction

Moreover, we can see that if the interval of out_pulse is twice with sw3 on than with sw3 off, the cycle of out_motor is the half of the second part, so the speed remains constant.

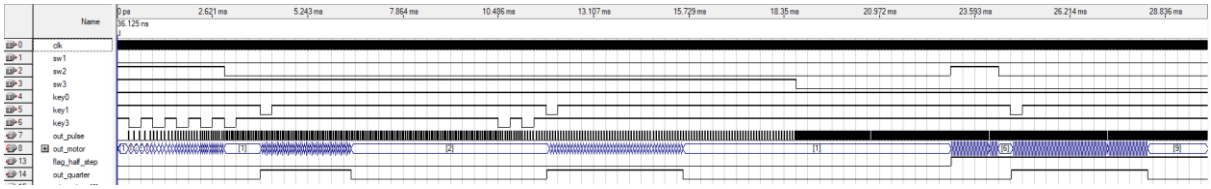
Then we tested changing the direction of rotation



As we can see we go from the sequence 1-5-4-6-2-10-8-9 to 9-8-10-2-6-4-5-1

For now, we want to check if the quarter rotation works properly.

First, we are interested to check the transition between continuous and quarter mode. Moreover, the long pulse as to be proportional to the speed.



Until 2.6 ms the motor run, we increased the speed to facilitate the simulation. Then sw2 is down the motor stops. A little later we press on key 1 and the motor run for the time out_quarter is up. After this we decrease the speed of 20 and the next long pulse will be longer than the first one because we decrease the speed: we need more time to finish the quarter of rotation. We also tested in half step mode the pulse out_quarter is the same length but the steps run twice the speed: our system works properly.