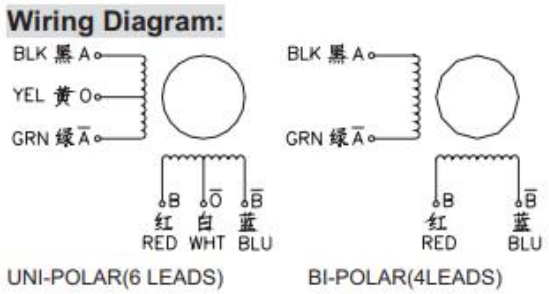
דו״ח מעבדה

מעבדה 2 – בקר מנוע צעד

מגישים: 342851284 איליה פורטוס

206698433 ארד שפירא

1. סקירה על מנוע צעד



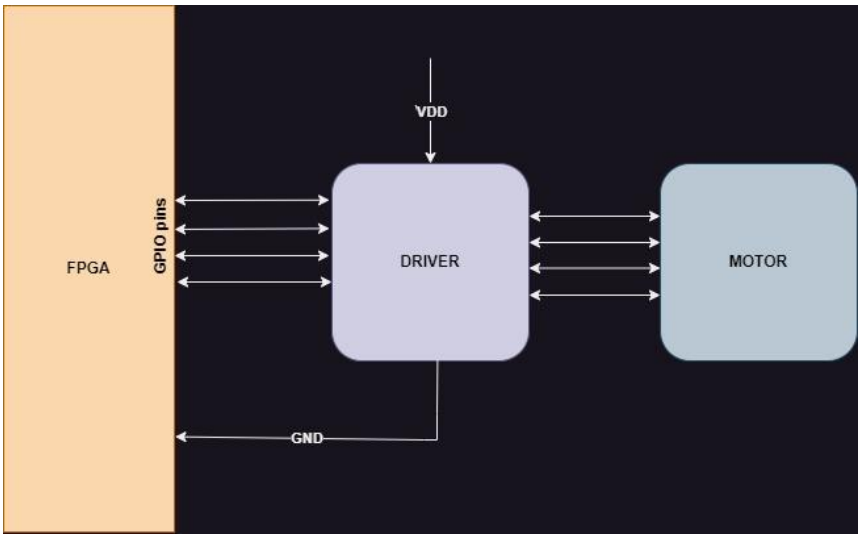
מנוע צעד הוא סוג של מנוע המופעל על ידי מתח DC. ההזזה של המנוע מתקיימת על ידי הפעלת מתח על הסלילים אשר מזיזים את המנוע בצעדים קבועים. הסלילים מקבלים מתח וכתוצאה מכך נוצר שדה מגנטי אשר מזיז את ציר המנוע בצעד. על ידי הפעלת מתח על הסלילים השונים בסדר מסוים ניתן להזיז את המנוע בסיבובים שלמים בהתאם לפרמטרים (מהירות המנוע, כיוון התנועה, גודל הצעד וכדומה).
אנחנו עובדים עם מנוע bipolar, שמורכב משני סלילים הפועלים יחד כדי לסובב את ציר המנוע. קיים גם מנוע unipolar שמורכב מ4 סלילים.

<u>הפעלת המנוע:</u> המנוע זז ב-$1.8°$ בצעד מלא וב-$0.9°$ בחצי צעד. לכן להשלמת סיבוב מלא נדרשים בהתאם 200 ו-400 צעדים. הסלילים מופעלים על ידי הספקת סדרות הבאות:

- צעד מלא עם כיוון השעון: 1000, 0010, 0100, 0001
- צעד מלא נגד כיוון השעון: 0001, 0100, 0010, 1000
- חצי צעד עם כיוון השעון: 1000, 1010, 0010, 0110, 0100, 0101, 0001, 1001
- חצי צעד נגד כיוון השעון: 1001, 0001, 0101, 0100, 0110, 0010, 1010, 1000

בצעד מלא הסלילים מופעלים אחד-אחד, לעומת זאת עבור צעד חלקי מפעילים סליל ראשון, אז מפעילים את השני בלי לכבות את הראשון ולבסוף מפסיקים הספקת מתח על הראשון ומפעילים רק על השני. אופן פעולה של חצי צעד מאפשר תנועה יותר חלקה.

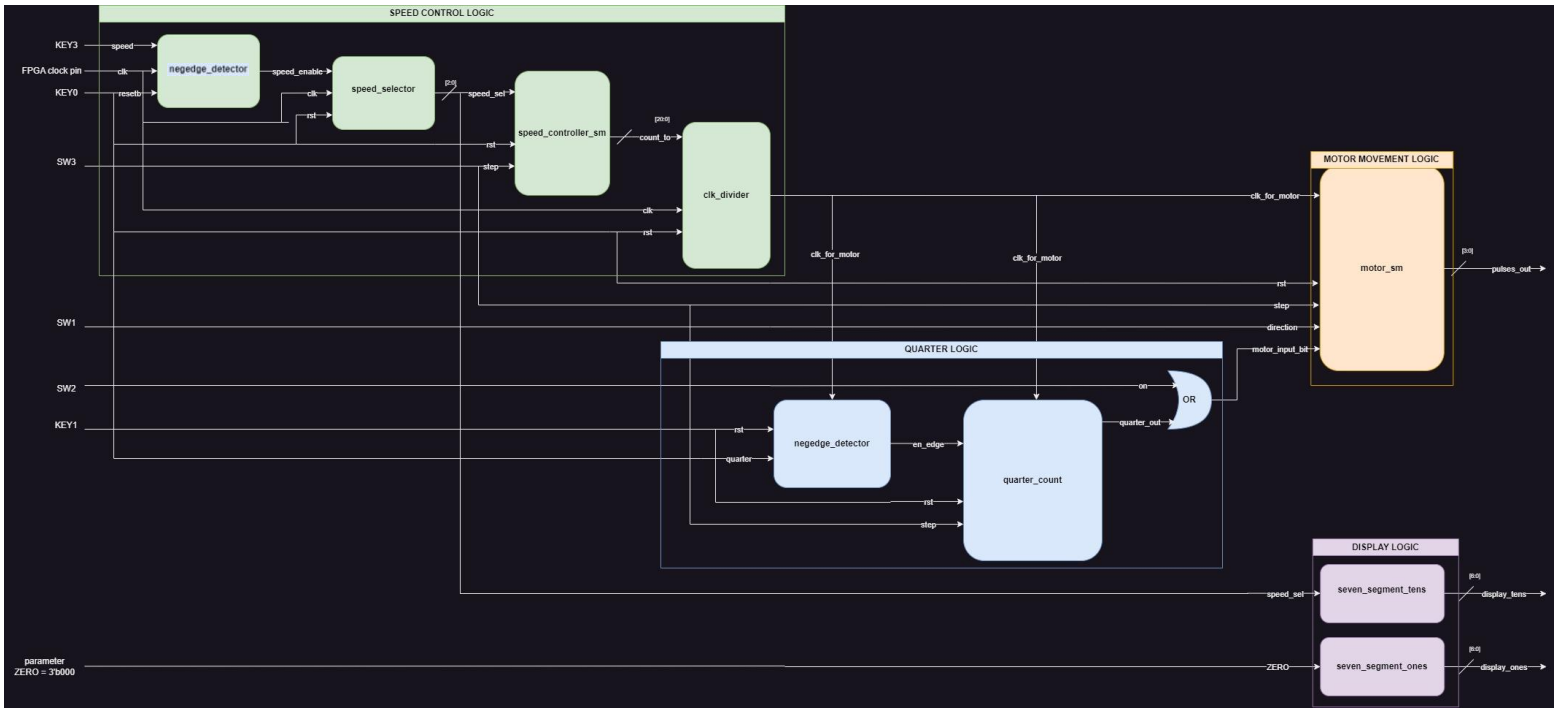
חיבור המנוע לכרטיס : בחיבור ל-FPGA השתמשנו בדרייבר חיצוני על מנת לספק את המתח הדרוש. להלן מוצגת דיאגרמת החיבור :



2. דיאגרמת הבלוקים של המעגל

להלן מוצגת דיאגרמת הבלוקים של המעגל. בה ניתן לראות את החלוקה לאזורים לוגיים :

1. לוגיקה של שינוי מהירויות
2. לוגיקה של רבע סיבוב
3. הצגת מהירות למסך 7-segment
4. מכונת מצבים האחראית על הזזת המנוע



המימוש שבחרנו מאפשר מודולריות גבוהה של המערכת, כלומר ניתן בקלות להוריד/להוסיף פעולה כלשהי למנוע בלי לפגוע בשאר המערכת.

3. הסבר של המימוש + סימולציות

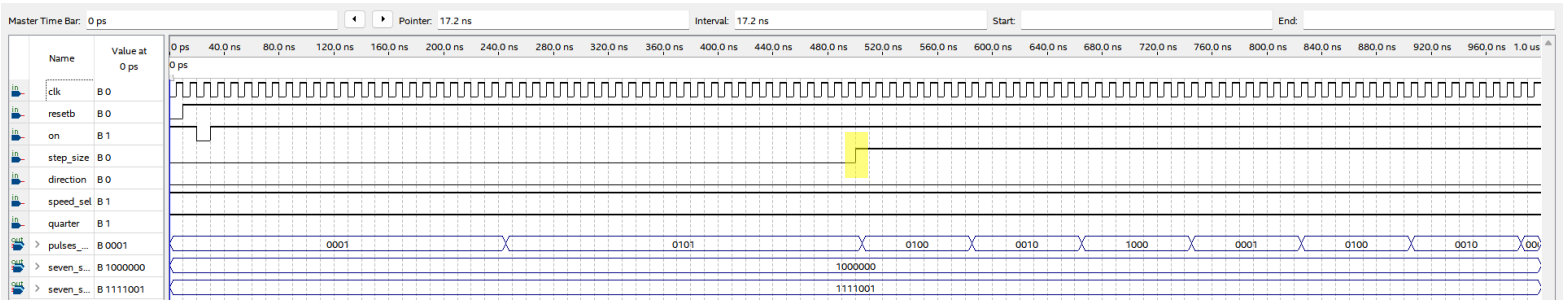
כעת נעבור על כל המודולים ונראה מהי הפעולה של כל אחד מהם.

- *step_motor_top*

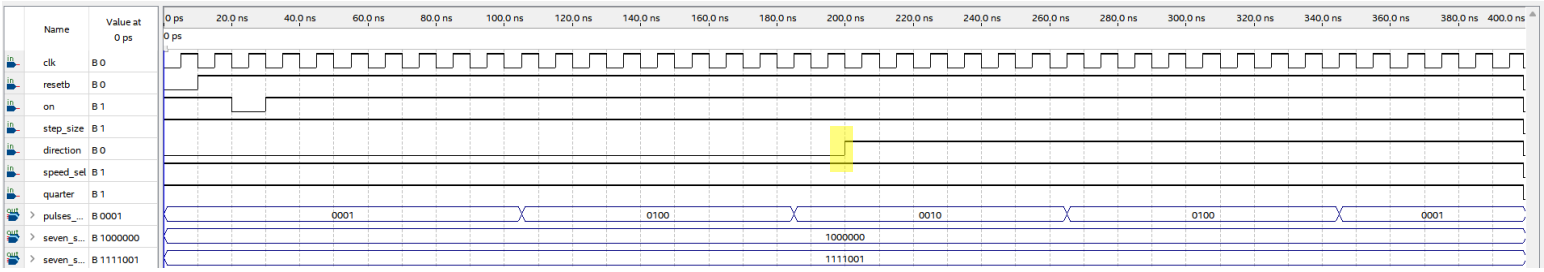
```
module step_motor_top(clk, resetb, direction, speed_sel, on, quarter, step_size, seven_seg_o, seven_seg_t, pulses_out);

   // IO declaration
   input wire clk;
   input wire resetb;
   input wire direction;
   input wire step_size;
   input wire speed_sel;
   input wire quarter;
   input wire on;
   output wire [6:0] seven_seg_o;   // for 7 seg
   output wire [6:0] seven_seg_t;   // for 7 seg
   output wire [3:0] pulses_out;    // signal for the driver


   // inner regs
   wire clk_for_motor;
   wire [2:0] speed;
   wire [20:0] count_to;
   wire speed_enable;
   wire en_edge;
   wire quarter_out;
   wire motor_input_bit;

   // parameters
   parameter ZERO = 3'b000;


   //posedge detector for velocity changes
   negedge_detector negedge_detector_inst1(.clk(clk), .rst(resetb), .sig(speed_sel), .out(speed_enable));

   //speed selector
   speed_selector speed_selector_inst(.clk(clk), .rst(resetb), .en(speed_enable), .speed_out(speed));

   //speed controller state machine
   speed_controller_sm speed_controller_sm_inst(.rst(resetb), .step(step_size), .speed_sel(speed), .count_to(count_to));

   //clock divider
   clk_divider clk_divider_inst(.clk(clk), .rst(resetb), .clk_out(clk_for_motor), .count_to(count_to));

   // negedge detector for quarter
   negedge_detector negedge_detector_inst2(.clk(clk_for_motor), .rst(resetb), .sig(quarter), .out(en_edge));

   //quarter counter
   quarter_count quarter_count_inst(.clk(clk_for_motor), .rst(resetb), .step(step_size), .en_edge(en_edge), .quarter_out(quarter_out));

   // OR gate to define whether the motor is in quarter state or in ON state
   assign motor_input_bit = (on || quarter_out);

   //motor controller
   motor_sm motor_sm_inst(.clk(clk_for_motor), .nrst(resetb), .input_bit(motor_input_bit), .direction(direction), .step(step_size), .out(pulses_out));

   //seven segment display output for ones
   seven_segment seven_segment_inst_ones(.number_in(ZERO), .hex_out(seven_seg_o));

   //seven segment display output for tens
   seven_segment seven_segment_inst_tens(.number_in(speed), .hex_out(seven_seg_t));

endmodule
```

זהו מודול ה-top בו מתקיימות הקריאות למודולים אחרים עם הכניסות המתאימות. הלוגיקה היחידה שנמצאת במודול (שורה 48) היא בחירה בין סיגנל מהמתג on ובין quarter_out – ביט המוצא של מודול של רבע סיבוב. על ידי שער OR אנו קובעים איזה סיגנל יהיה ה-enable של מכונת המצבים של המנוע. להלן סימולצית גלים של המודול:

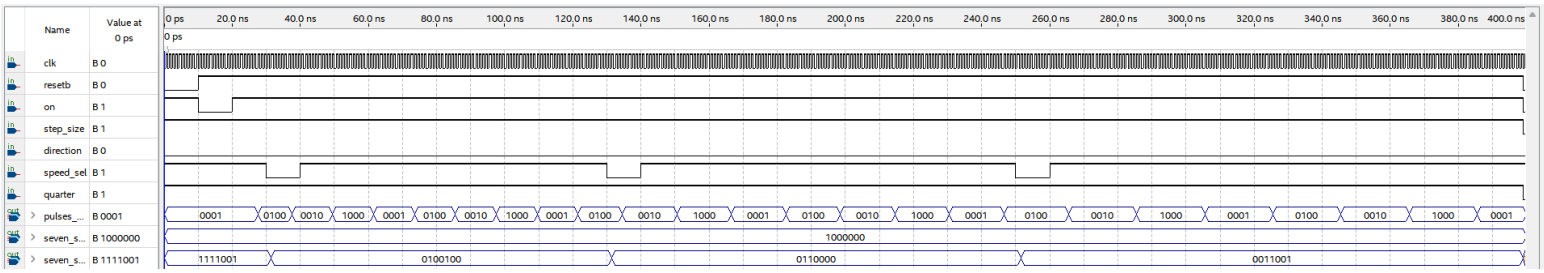
1. שינוי גודל הצעד – התדירות של שינוי המצבים גדלה פי 2 – כפי שציפינו מ-half step



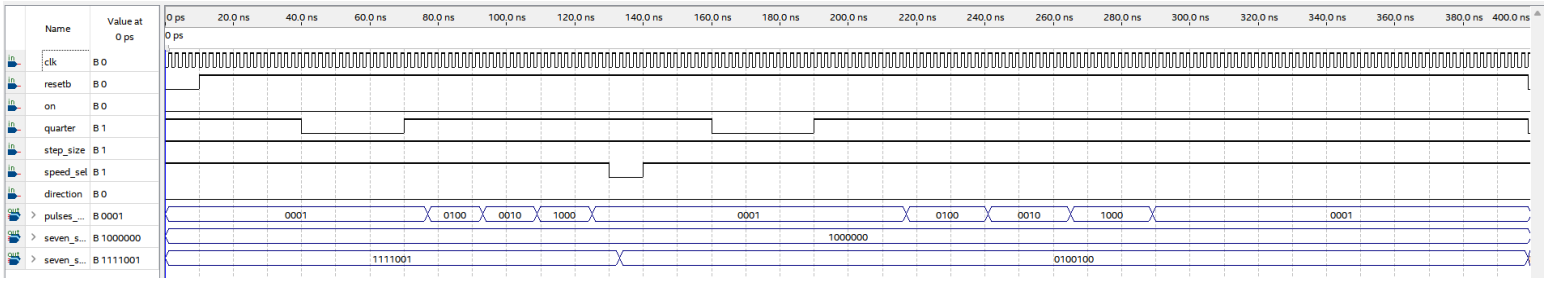
2. שינוי כיוון התנועה – בשינוי כיוון התנועה המנוע נשאר במצב הנוכחי הנדרש ובמצב הבא עובר למצב של הכיוון ההפוך.



3. שינוי המהירות – ניתן לראות שינוי בהחלפת המהירות



4. רבע סיבוב – ניתן לראות שסופרים עד 4 מצבים (הערך לבדיקה – בפועל סופרים ל-100/200) ואז נשארים במצב האחרון



בלוק speed control logic

הסבר אופן הפעולה : ברגע של לחיצה על כפתור המהירות (KEY3), negedge_detector מוציא ביט באורך מחזור שעון אחד אשר פועל בתור enable ל-speed selector. הוא מוציא את המהירות המתאימה למצב. המהירות המתאימה נכנסת ל-speed_controller_sm שמוציא את כמות מחזורי שעון שצריך לספור בהתאם למהירות. ה-clock_divider מקבל את המספר הזה ומוציא את השעון החדש המחולק בהתאם לכמה שצריך לספור.

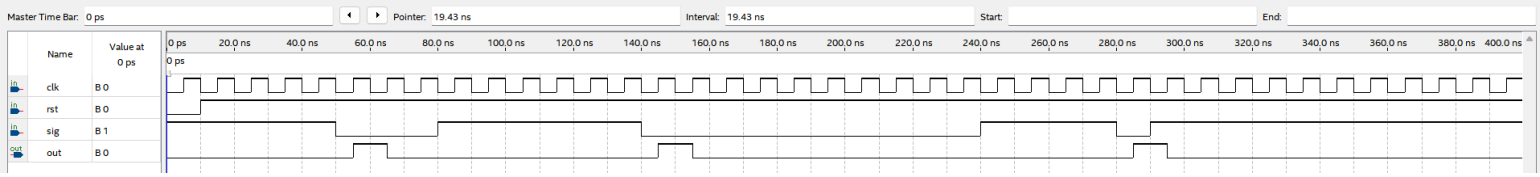
נסתכל כעת על כל אחד מהמודולים ונראה סימולציות.

- *negedge_detector*

```
module negedge_detector(
    // Io declaration
    input wire clk,
    input wire rst,
    input wire sig,
    output wire out);

    // inner reg
    reg [1:0] sig_del;

    // logic as in posedge detector
    always @(posedge clk or negedge rst)
    begin
        if (~rst) begin
            sig_del <= 2'b11;
        end else begin
            sig_del <= {sig_del[0], sig};
        end
    end

assign out = ((~sig_del[0]) & (sig_del[1]));

endmodule
```

המודול אחראי על זיהוי ה-negedge של לחיצה על כפתור שינוי המהירות. מכיוון שהלחיצה היא הרבה יותר ארוכה ממחזור השעון של הFPGA, רצינו לזהות את תחילת הלחיצה ולאחר מכן לשלוח פולס לאורך מחזור שעון אחד אשר ייקבע לדרגה הבאה שהתרחשה הלחיצה.

הסימולציה : ניתן לראות שבירידת סיגנל מתקבל פולס של מחזור שעון אחד

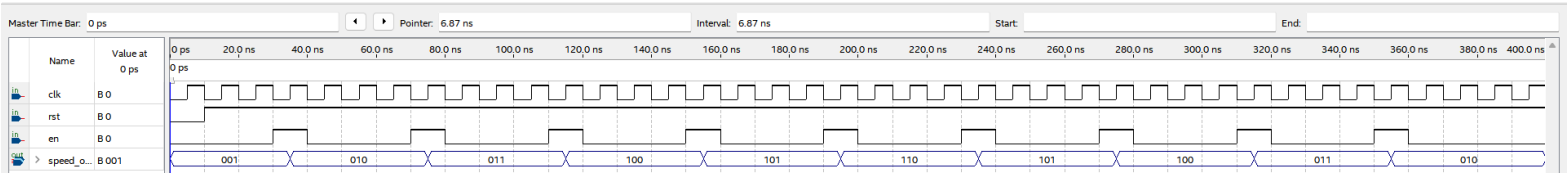


- *speed_selector*

```
module speed_selector(
    // IO declaration
    input wire clk,
    input wire rst,
    input wire en,
    output wire [2:0] speed_out);

 // inner regs
 reg [2:0] cs;
 reg [2:0] ns;
 reg flag;

 // parameters: states that are passed to motor
 parameter state10 = 3'b001;
 parameter state20 = 3'b010;
 parameter state30 = 3'b011;
 parameter state40 = 3'b100;
 parameter state50 = 3'b101;
 parameter state60 = 3'b110;


 // update current state to the next state logic
 always @(posedge clk or negedge rst)
 begin
    if (~rst)
       begin
          cs <= state10;
          flag <= 1'b0;
       end
    else if (en)
       begin
          cs <= ns;
          if ((cs == state10) | (cs == state60))
             flag <= (~flag);  // change the flag when at 10/60 rpm to go to the opposite side
       end
 end

 // state machine
 always @(*)
    begin
       case (cs)
          state10: ns = state20;
          state20: ns = (flag) ? state30 : state10;
          state30: ns = (flag) ? state40 : state20;
          state40: ns = (flag) ? state50 : state30;
          state50: ns = (flag) ? state60 : state40;
          state60: ns = state50;
          default: ns = state10;
       endcase
    end

 assign speed_out = cs;

 endmodule
```

המודול כולל את מכונת המצבים האחראית על החלפת מהירויות. הוא מקבל פולס של מחזור שעון אחד מהמודול הקודם, ובהתאם מחליף את המצב שלו. ישנם 6 מצבים בהתאם למהירויות. כשאר מגיעים למצב של 10/60 rpm, מחליפים את משתנה ה-flag וחוזרים חזרה בהתאם לתנאי בכל מצב (flag=1 – קדימה, אחרת אחורה).

הסימולציה: ניתן לראות שמגיעים ממהירות 10 ל-60 וחוזרים אחורה



speed_controller_sm -

```
module speed_controller_sm(
    input wire rst,
    input wire step,
    input wire [2:0] speed_sel,
    output reg [20:0] count_to
);


 // all the values must be *4 but in reality we got the right speed values by dividing by 4


 // REAL VALUES
 //parameter full_10rpm =21'd375000; // 1500000
 //parameter full_20rpm =21'd187500; // 750000
 //parameter full_30rpm =21'd125000; // 500000
 //parameter full_40rpm =21'd93750;  // 375000
 //parameter full_50rpm =21'd75000;  // 300000
 //parameter full_60rpm =21'd62500;  // 250000
 //parameter half_10rpm =21'd187500; // 750000
 //parameter half_20rpm =21'd93750;  // 375000
 //parameter half_30rpm =21'd62500;  // 250000
 //parameter half_40rpm =21'd46750;  // 187500
 //parameter half_50rpm =21'd37500;  // 150000
 //parameter half_60rpm =21'd31250;  // 125000


 // VALUES FOR WAVEFORM TEST
 parameter full_10rpm =21'd1;
 parameter full_20rpm =21'd2;
 parameter full_30rpm =21'd3;
 parameter full_40rpm =21'd4;
 parameter full_50rpm =21'd5;
 parameter full_60rpm =21'd6;
 parameter half_10rpm =21'd7;
 parameter half_20rpm =21'd8;
 parameter half_30rpm =21'd9;
 parameter half_40rpm =21'd10;
 parameter half_50rpm =21'd11;
 parameter half_60rpm =21'd12;

 // logic
 always @(*)
 begin
         if (~rst)
            count_to <= full_10rpm;
         if ((speed_sel == 3'b001) && (step))
            count_to <= full_10rpm;

         else if ((speed_sel == 3'b001) && (~step))
            count_to <= half_10rpm;

         else if ((speed_sel == 3'b010) && (step))
            count_to <= full_20rpm;

         else if ((speed_sel == 3'b010) && (~step))
            count_to <= half_20rpm;

         else if ((speed_sel == 3'b011) && (step))
            count_to <= full_30rpm;

         else if ((speed_sel == 3'b011) && (~step))
            count_to <= half_30rpm;

         else if ((speed_sel == 3'b100) && (step))
            count_to <= full_40rpm;

         else if ((speed_sel == 3'b100) && (~step))
            count_to <= half_40rpm;

         else if ((speed_sel == 3'b101) && (step))
            count_to <= full_50rpm;

         else if ((speed_sel == 3'b101) && (~step))
            count_to <= half_50rpm;

         else if ((speed_sel == 3'b110) && (step))
            count_to <= full_60rpm;

         else if ((speed_sel == 3'b110) && (~step))
            count_to <= half_60rpm;
         else
            count_to <= full_10rpm;
 end


 endmodule
```

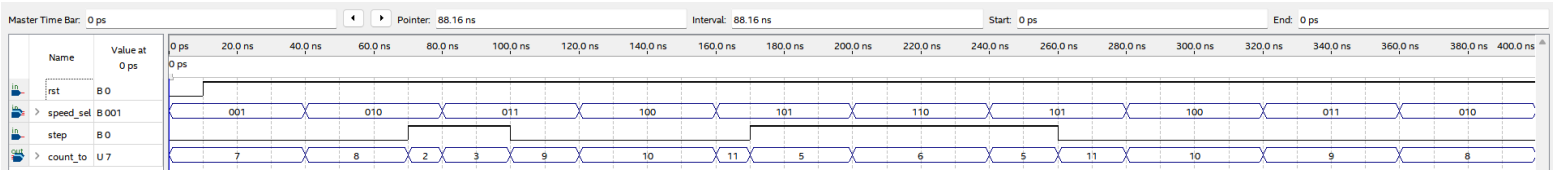
המודול מקבל את ערך המהירות מהמודול הקודם (3 ביט) ובהתאם לגודל הזה וגודל הצעד (input step) מוציא את מספר עד עליו נצטרך לספור. המספר הזה מתקבל אח״כ במחלק השעון (ראה עמוד הבא).

<u>הערה</u>: חישבנו את המספרים האמיתיים לפי פרמטרי המנוע, אך בהצבה של המספרים האלה לא קיבלנו את המהירות הרצויה (מדדנו בפועל את כמות הסיבובים לדקה שהמנוע עושה). לכן חילקנו את כל אחד מהמספרים ב-4, שזה נותן בדיוק את המהירויות הרצויות.

<u>הערה</u>: חישבנו את הערך שעליו צריך לספור לפי הנוסחה הבאה:

$$count = \frac{clock\ of\ FPGA}{\frac{turns}{sec} * (\frac{360}{1.8}\ for\ full\ step\ OR\ \frac{360}{0.9}\ for\ half\ step)}$$

הסימולציה : ניתן לראות ש-count_to משתנה בהתאם לפרמטרי הכניסה כנדרש



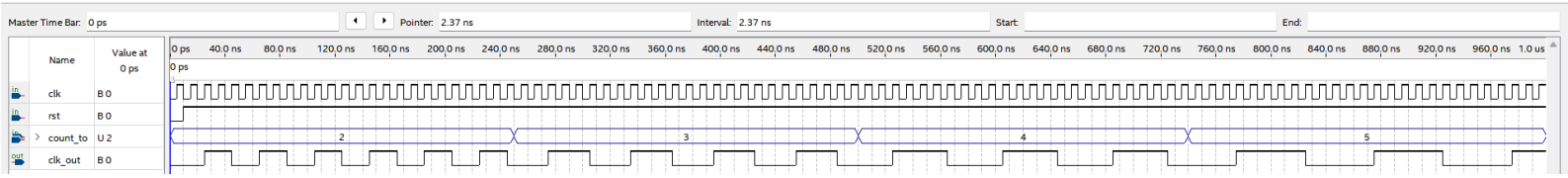
<u>הערה</u> : על מנת להבחין בשינויים שינינו את הערכים (בקוד זה מופיע בתור values for test)

- *clock_divider*

```
1   module clk_divider(clk, rst, count_to, clk_out);
2
3   // IO declaration
4   input wire clk;
5   input wire rst;
6   input wire [20:0] count_to;
7   output reg clk_out;
8
9   // inner reg
10  reg [20:0] count;
11
12  // clock divider logic - create clock with half period of count_to
13  always @(posedge clk or negedge rst)
14     begin
15        if (~rst) begin
16              count <= 21'b0;
17              clk_out = 0;
18        end else if (count >= count_to - 1'b1) begin
19           count <= 21'b0;
20           clk_out = (~clk_out); // change the clock
21        end else begin
22           count <= count + 21'b1;
23        end
24     end
25
26  endmodule
```

זהו המודול של מחלק שעון. הוא מקבל את המספר עד עליו צריך לספור (count_to) ובהתאם למהירות שהתקבלה מייצר את השעון המחולק אשר ישמש במכונת המצבים של המנוע. המודול בנוי כמו counter רגיל שראינו בקורס, רק אם הבדל אחד של החלפת ערך במקום הוצאת אותו חוצה (שורה 20). השינוי הזה גורם לכך שהמוצא של מודול לא יראה כמו רצף פולסים אלא שיהיה שעון עם מחזור שעון ארוך מהזה של FPGA.

הסימולציה: ניתן לראות כי בהתאם לכניסה מחזור השעון המחולק משתנה כנדרש



בלוק quarter logic

הסבר אופן הפעולה : ברגע של לחיצה על כפתור רבע סיבוב (KEY1), negedge_detector מוציא ביט באורך מחזור שעון אחד אשר פועל בתור enable ל-quarter_count. הוא מוציא 1 לוגי בהתאם לכמות מחזורי שעון שצריך לספור (50 לצעד מלא, 100 לחצי צעד) ואז מוציא 0 לוגי.

נסתכל כעת על כל אחד מהמודולים ונראה סימולציות.

- *negedge_detector*

```
module negedge_detector(
    // Io declaration
    input wire clk,
    input wire rst,
    input wire sig,
    output wire out);

    // inner reg
    reg [1:0] sig_del;

    // logic as in posedge detector
    always @(posedge clk or negedge rst)
    begin
       if (~rst) begin
          sig_del <= 2'b11;
       end else begin
          sig_del <= {sig_del[0], sig};
       end
    end

assign out = ((~sig_del[0]) & (sig_del[1]));

endmodule
```

הראנו מקודם את אופן הפעולה של המודול ואת הסימולציה

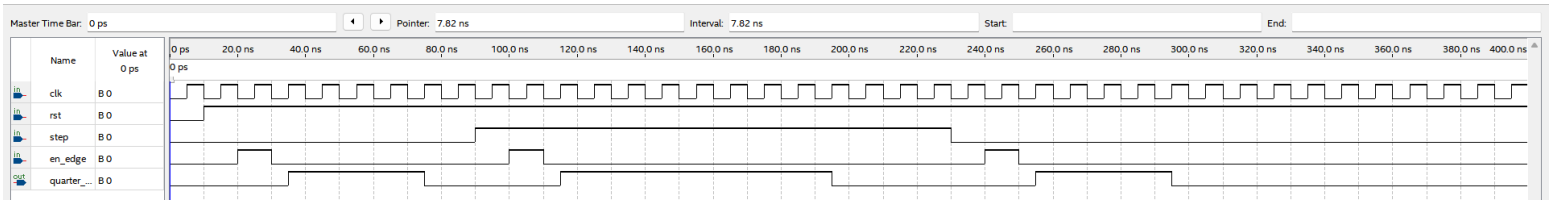
- *quarter_count*

```
module quarter_count(
    // IO declaration
    input wire clk,   //clk_for_motor
    input wire rst,
    input wire step,
    input wire en_edge,
    output reg quarter_out);

    // parameters: according to step size
    parameter HS_COUNT = 8'd200;
    parameter FS_COUNT = 8'd100;

    // inner regs
    reg [7:0] count;
    reg start_count;

    // counter logic
    always @(posedge clk or negedge rst)
    begin
       if (~rst)
          begin
             count <= 8'd0;
             quarter_out <= 1'b0;
          end
       else if (en_edge) // if enabled
          begin
             start_count <= en_edge;
          end
       else if (start_count)
          begin
             if (step == 1'b1) // if in full step
                begin
                   if (count >= FS_COUNT)
                      begin
                         quarter_out <= 1'b0;
                         start_count <= 1'b0;
                         count <= 8'b0;
                      end
                   else
                      begin
                         count <= count + 8'b1;
                         quarter_out <= 1'b1;
                      end
                end
             else if (step == 1'b0)  // the same logic but if in half step
                begin
                   if (count >= HS_COUNT)
                      begin
                         quarter_out <= 1'b0;
                         start_count <= 1'b0;
                         count <= 8'b0;
                      end
                   else
                      begin
                         count <= count + 8'b1;
                         quarter_out <= 1'b1;
                      end
                end
          end
    end

endmodule
```

המודול שולט על פעולה של רבע סיבוב. המודול מקבל את השעון המחולק בהתאם למהירות הנוכחית, את הפולס מהמודול הקודם שקובע את הלחיצה על הכפתור ואת גודל הצעד. בתוך המודול יש counter עם enable, כלומר אם זיהינו את הלחיצה, המודול מופעל, לפי גודל הצעד נקבע עד כמה לספור וה-counter מאותחל. המודול מוציא החוצה ביט באורך של כמות המחזורים (בהתאם לגודל הצעד) ואז 0.

<u>הערה</u>: בגלל שחילקנו את כל המספרים במודול של speed_controller_sm פי 4, בהתאם נצטרך לספור לא עד 100 (חצי צעד) ו-50 מחזורים (צעד מלא), אלא ל-200 ו-100 בהתאם. בבדיקה המעשית ראינו שעם המספרים האלה המנוע אכן עושה רבע סיבוב כנדרש.

הסימולציה: ניתן לראות שהמוצא הוא כנדרש



<u>הערה</u>: על מנת לבחון בשינויים בסימולציה, הקטנו את הפרמטרים מ-100 ו-200 ל-4 ו-8 בהתאם.

בלוק motor movement logic

הסבר אופן הפעולה : הבלוק כולל את מודול של מכונת מצבים האחראית על שליחת ביטים להזזת המנוע.

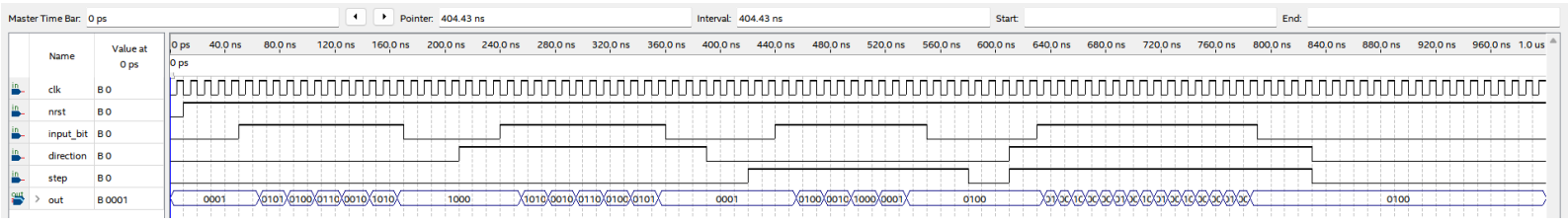
- *motor_sm*

```
module motor_sm (clk,
                 nrst,
                 input_bit,
                 direction,
                 step,
                 out);

    // IO definition
    input wire clk;
    input wire nrst;
    input wire input_bit;
    input wire direction;
    input wire step;  // full step=1, half step=0
    output wire [3:0] out;

    // inner variables
    reg [3:0] cs;
    reg [3:0] ns;

    // parameters
    parameter A = 4'b1000,
              AB = 4'b1010,
              B = 4'b0010,
              BC = 4'b0110,
              C = 4'b0100,
              CD = 4'b0101,
              D = 4'b0001,
              DA = 4'b1001;

    // state machine
    always @(posedge clk or negedge nrst)
    begin
       if (~nrst)
          begin
          cs <= cs;
          end
       else if (input_bit == 1'b1)
          begin
             cs <= ns;

             case (cs)
                A:
                   if (direction == 1'b1 && step == 1'b1)
                      ns = B;
                   else if (direction == 1'b0 && step == 1'b1)
                      ns = D;
                   else if (direction == 1'b1 && step == 1'b0)
                      ns = AB;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = DA;
                AB:
                   if (direction == 1'b1 && step == 1'b0)
                      ns = B;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = A;

                B:
                   if (direction == 1'b1 && step == 1'b1)
                      ns = C;
                   else if (direction == 1'b0 && step == 1'b1)
                      ns = A;
                   else if (direction == 1'b1 && step == 1'b0)
                      ns = BC;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = AB;

                BC:
                   if (direction == 1'b1 && step == 1'b0)
                      ns = C;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = B;

                C:
                   if (direction == 1'b1 && step == 1'b1)
                      ns = D;
                   else if (direction == 1'b0 && step == 1'b1)
                      ns = B;
                   else if (direction == 1'b1 && step == 1'b0)
                      ns = CD;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = BC;
```

```
                CD:
                   if (direction == 1'b1 && step == 1'b0)
                      ns = D;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = C;

                D:
                   if (direction == 1'b1 && step == 1'b1)
                      ns = A;
                   else if (direction == 1'b0 && step == 1'b1)
                      ns = C;
                   else if (direction == 1'b1 && step == 1'b0)
                      ns = DA;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = CD;

                DA:
                   if (direction == 1'b1 && step == 1'b0)
                      ns = A;
                   else if (direction == 1'b0 && step == 1'b0)
                      ns = D;
                default:
                   ns = A;
             endcase
          end
       end


assign out = cs;

endmodule
```

זהו המודול אשר קובע את המצב הנוכחי של המנוע ושולח לו את 4 ביט. המודול מכיל את מכונת המצבים אשר לפי גודל הצעד וכיוון התנועה (עם/נגד כיוון השעון) מחליף בין הצמבים. יש סה״כ 8 מצבים אפשריים – 4 ראשיים לצעד מלא ו-4 מצבי ביניים לחצי צעד. המנוע מקבל את השעון המחולק בהתאם למהירות מה-clock_divider.

הסימולציה : ניתן לראות שהמצבים משתנים בהתאם לכניסות כנדרש



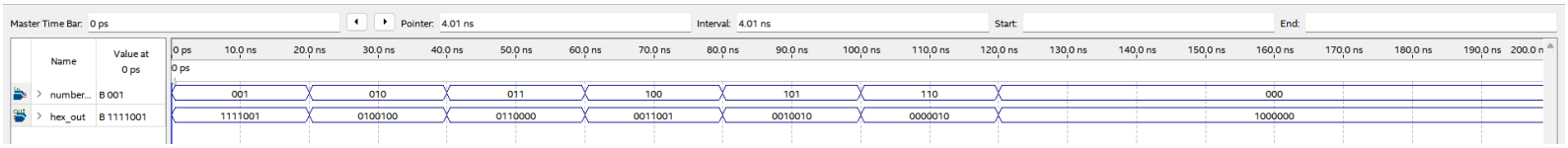
בלוק display logic

- *seven_segment*

```
1   module seven_segment(number_in, hex_out);
2
3      // in/out declaration
4      input wire [2:0] number_in;
5      output wire [6:0] hex_out;
6
7      // inner reg declaration
8      reg [6:0] hex;
9
10     // logic
11     always @(*)
12        begin
13           case(number_in)
14                 3'b000: hex = 7'b1000000;  // number = 0
15                 3'b001: hex = 7'b1111001;  // number = 1
16                 3'b010: hex = 7'b0100100;  // number = 2
17                 3'b011: hex = 7'b0110000;  // number = 3
18                 3'b100: hex = 7'b0011001;  // number = 4
19                 3'b101: hex = 7'b0010010;  // number = 5
20                 3'b110: hex = 7'b0000010;  // number = 6
21
22           endcase
23        end
24
25     assign hex_out = hex;
26
27  endmodule
```

זהו המודול אשר מעביר את הערך של המהירות למספר אשר נשלח ל-7-segment display. יש שתי קריאות למודול – אחת לעשרות ואחת לאחדות. הקריאה לאחדות מקבלת פרמטר 0 תמיד (הרי המהירויות שלנו הם כפולות של 10) והקריאה לעשרות מקבלת את הערך מהמודול speed_controller (אשר מוציא את הערכים 1 עד 6 בהתאם למהירות הנוכחית).

הסימולציה:



4. סרטון של המנוע

להלן מצורף קישור לסרטון המוכיח את פעולת המנוע בכל המצבים הנדרשים שלו:

https://drive.google.com/drive/folders/1NjUFtS9h3SJfQyZAFKnY2IZ3iQ2w-Z4H?usp=sharing