דו"ח מעבדה דיגיטלית 1

יואב אשד 305384869

גלעד בינו 302252101

1. מימוש $FLIP\ FLOP$ מסוג SR.

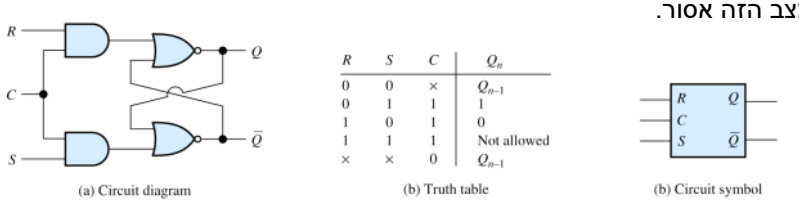
-רקע:

בתרגיל נתנה לנו טבלת האמת של $FLIP\ FLOP$ מסוג SR, ומימוש שלו באמצעות שערים לוגיים עליו התבססנו לטובת יצירת המעגל ב$quartus$.

$FLIP\ FLOP$ מסוג SR הוא אחד מהמעגלים הלוגים הבסיסיים ביותר. FF כזה מקבל ערך שני קלטים, S וR , כאשר $S = set$ עבורו המוצא יהיה "1", ו $R = reset$ עבורו המוצא יהיה "0".

תיאור הFF נקרא $SR \rightarrow SET\ RESET$. זאת מאחר והקלט של ה$reset$ מאפס את הFF למצב המקורי שלו עם מוצא שיהיה בערך לוגי "1" או ערך לוגי "0" , תלוי במצב $set/reset$.

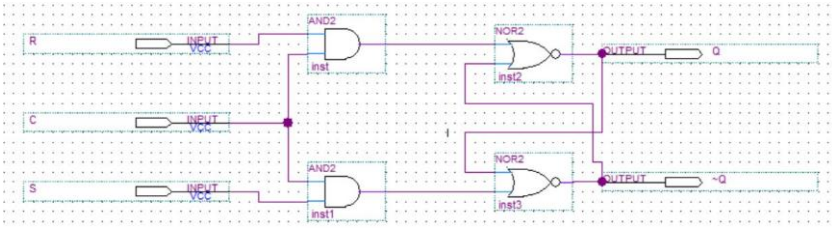
ערך הFF ביציאה משתנה עבור עליית שעון, עבור "0" בשני הכניסות המצב לא ישתנה, עבור ערכים שונים בכניסות נקבל שינוי מצב של שתי כניסות "1" נקבל FF לא יציב ולכן המצב הזה אסור.



R	S	C	Q_n
0	0	×	Q_{n-1}
0	1	1	1
1	0	1	0
1	1	1	Not allowed
×	×	0	Q_{n-1}

Figure 7.41 A clocked SR flip-flop.

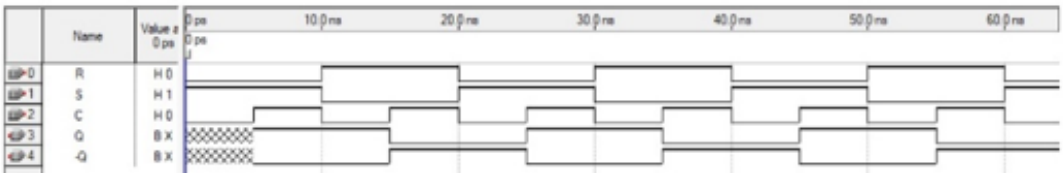
-תכנון המעגל ב$quartus$:



- סימולציית functional

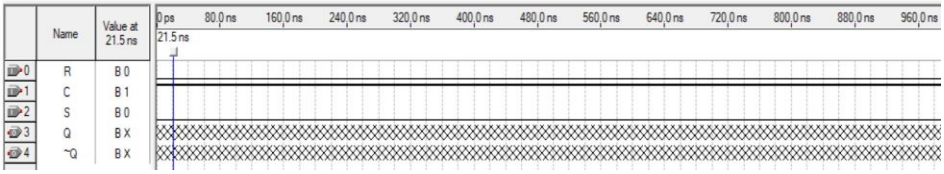
מהסימולציות שבוצעו ניתן לשים לב כי התנהגות הFF תלויה בשעון, והמוצא משתנה עבור עליית השעון , כלומר הדגימה מתרחשת בעליית השעון , ונקבעת לפי ערך הכניסה. לכן הרכיב הינו רכיב סינכרוני.

עבור כניסות $R = 0; S = 1;$:



ניתן לראות כי בכל דגימת שעון , המוצא משתנה בהתאם לכניסות ומקיים את דרישות טבלת האמת, כאשר $R = 0, S = 1$ המוצא מראה 1, ועבור $R = 1, S = 0$.

עבור כניסות $R = 0; S = 0;$:



ניתן לראות כי המערכת לא עובדת במצופה

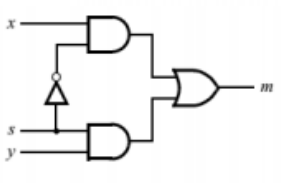
2. <u>סלקטור 8 ל1 באמצעות סלקטור 2 ל1.</u>

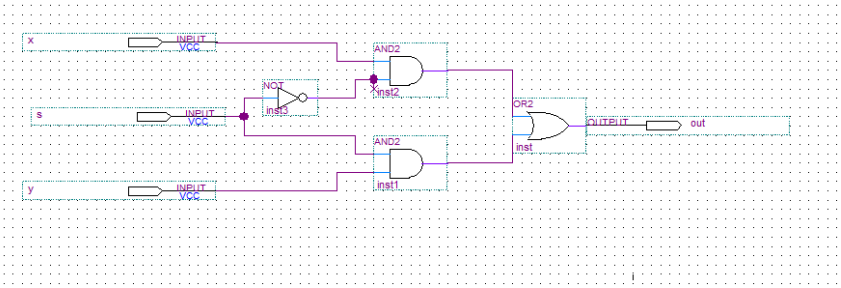
-רקע: mux או $muliplexer$ הינו רכיב עם קלטים מרובים ומוצא יחיד. לmux כניסות select הקובעים אילו מכניסות המידע מחוברות למוצא וכן את כמות המידע שיכולה לעבור בפרק זמן מסוים. muxים יכולים לשמש ביישומים אנלוגיים או דיגיטליים , כאשר ביישומים אנלוגיים muxים מורכבים מטרנזיסטורים ושנאים, וביישומים דיגיטליים muxים מורכבים משערים דיגיטליים.

1.

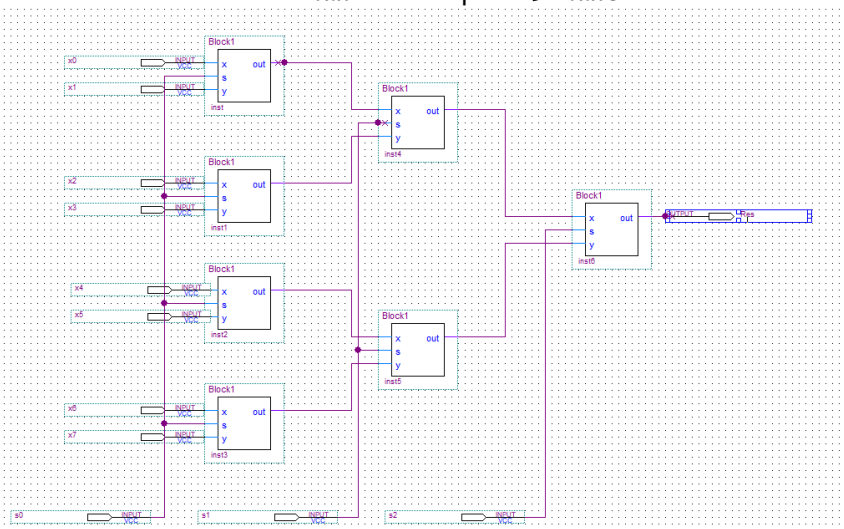
א. מימוש $mux2:1$ באמצעות דיאגרמת בלוקים, כאשר השתמשנו בשרטוט שסופק בתרגיל.

שרטוט | שרטוט בquartus



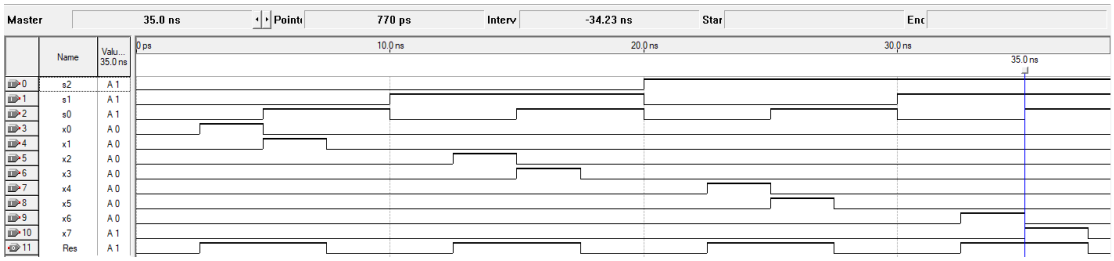


ב. מימוש $mux8:1$ ע"י בלוקים של $mux2:1$



בחרנו לחבר את הmux 8:1 באופן זה מאחר וכך אנו יכולים להשתמש בבלוקים של $mux\ 2:1$ אותו יצרנו.

סימולציית תקינות ל$mux\ 8:1$



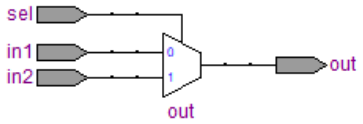
ניתן לראות כי שלושת הביטים של כניסת ה $select$ שולטים איזו כניסה מראה את היציאה. בדקנו את כל המקרים האפשריים עבור שלוש כניסות וניתן לראות שיש אות ביציאה רק כאשר יש אות באחת משמונה הכניסות וזה אכן תואם את טבלת האמת של $mux\ 8:1$ כנדרש.

.2

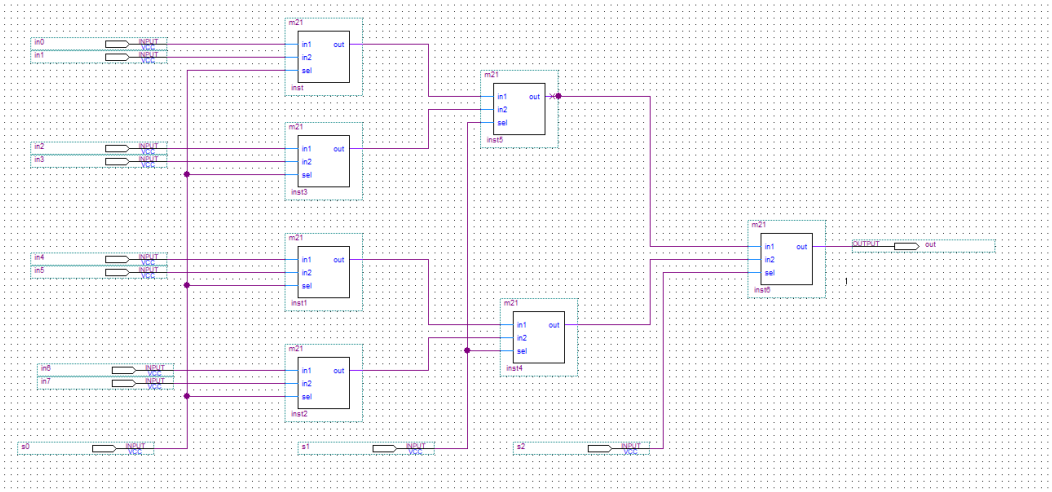
א. מימוש בלוק של mux2:1 בקוד ורילוג

```
//mux 2:1
module m21(in1, in2, sel, out);
//inputs
input wire in1;
input wire in2;
input wire sel;
//outputs
output out;
//mux  function
assign out = (sel)?in2:in1;
endmodule
```

שרטוט RTL



ב. מימוש $mux8:1$ באמצעות בלוקים של הmux2:1 שתוכנן בקוד בסעיף הקודם , התכנון נובע מאותו שיקול של הסעיף הקודם.



.3 מימוש $mux8:1$ באמצעות מופעים של המודול מסעיף 2א'

כל מופע מייצג mux 2:1 וכמו בסעיפים הקודמים, 4 mux מובילים ל2 וה2 מובילים ל1 שממנו מקבלים את אות היציאה.

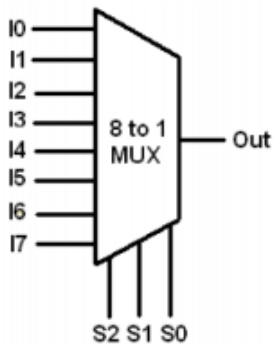
```
// mux 8:1 with 2.A code instances
module mux81_23(in0,in1,in2,in3,in4,in5,in6,in7,s0,s1,s2,out);
// inputs
    input wire  s0;
    input wire  s1;
    input wire  s2;
    input wire  in0;
    input wire  in1;
    input wire  in2;
    input wire  in3;
    input wire  in4;
    input wire  in5;
    input wire  in6;
    input wire  in7;
//output
    output wire out;
//inner wires (from first level to second level(8->4) and from second level to third level(4-2)
    wire    wire_0;
    wire    wire_1;
    wire    wire_2;
    wire    wire_3;
    wire    wire_4;
    wire    wire_5;
//first level mux, entries 0 and 1, out wire_0
    m21 m21_inst0
    (
        .in1(in0) , // input  in1_sig
        .in2(in1) , // input  in2_sig
        .sel(s0) ,  // input  sel_sig
        .out(wire_0)     // output  out_sig
    );
//first level mux, entries 2 and 3, out wire_1
    m21 m21_inst1
    (

//first level mux, entries 4 and 5 out wire_2
    m21 m21_inst2
    (

//first level mux, entries 6 and 5 out wire_3
    m21 m21_inst3
    (

//second level mux, entries wire_0 and wire_1 out wire_4
    m21 m21_inst4
    (

//second level mux, entries wire_0 and wire_1 out wire_5
    m21 m21_inst5
    (

//third level mux, entries wire_4 and wire_5 out mux total output
    m21 m21_inst6
    (
endmodule
```

4. $mux8:1$ במימוש על ידי קוד.

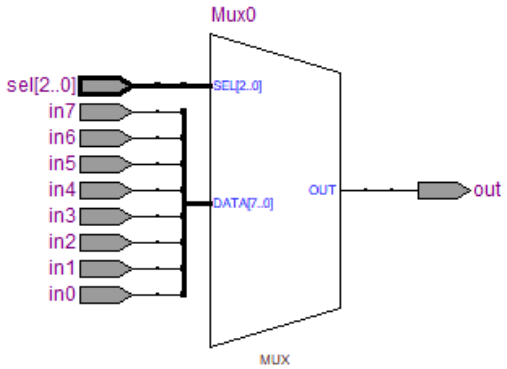
על מנת לממש בצורה הפשוטה ביותר בחרנו להשתמש בצורה של מקרים עבור כל צירוף אותות בקרה אפשריים.

```
//mux 8:1
module mux81_24(in0,in1,in2,in3,in4,in5,in6,in7,out,sel);
//inputs
    input wire in0;
    input wire in1;
    input wire in2;
    input wire in3;
    input wire in4;
    input wire in5;
    input wire in6;
    input wire in7;
//select inputs
    input wire[2:0]sel;
//output wire
    output reg out;
//output selection process
    always @(*)
    begin
        case(sel)
        3'b000:
            out = in0;
        3'b001:
            out = in1;
        3'b010:
            out = in2;
        3'b011:
            out = in3;
        3'b100:
            out = in4;
        3'b101:
            out = in5;
        3'b110:
            out = in6;
        3'b111:
            out = in7;
        endcase
    end
endmodule
```

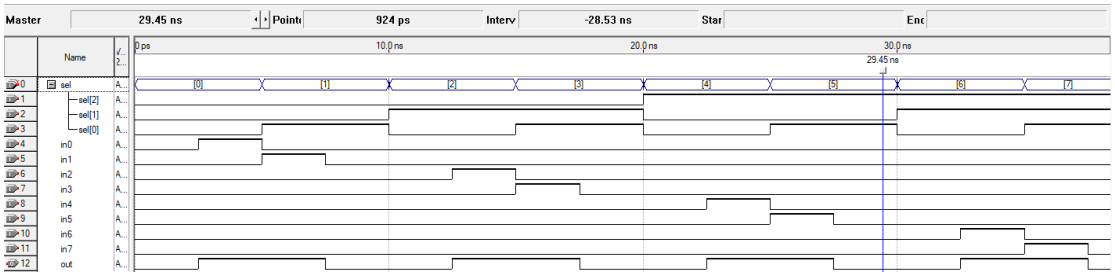
<u>שרטוט שסופק בתרגיל</u>



<u>שרטוט RTL</u>



סימולציית פעולה:



ניתן לראות כי התוצאה של הסימולציה דומה עד זהה, גם כאן שלושת הביטים של כניסת ה $select$ שולטים איזו כניסה מראה את היציאה, וגם כאן היציאה עולה רק כאשר מוכנס אות באחת הכניסות וכי זה מקיים את טבלת האמת של הmux, ולכן ה mux מתפקד כנדרש.

-יתרונות לשימוש בשפת VHDL:

- מורכבות: כאשר יש לממש מעגל מורכב , אם מבחינת פונקציונאליות ואם מבחינת מספר הכניסות והיציאות של הרכיב, אז קל יותר לעשות זאת באמצעות כתיבת קוד $verilog$ מאשר באמצעות שרטוט. כך מתקבל סיכוי נמוך יותר לחיבורים שגויים או יצירת סכמה מסורבלת ולא קריאה.

- גמישות: על מנת לממש מעגל בסכמת בלוקים, נרצה להיעזר ברכיבים קיימים(כפי שעשינו בתרגיל הנ"ל). בדיאגרמת בלוקים אנו מוגבלים לפונקציות הקיימות בספרייה הקיימת ואם נרצה להוסיף נצטרך להוסיף את החסר באמצעות קוד. על ידי שימוש בקוד איננו נתונים למגבלות הסביבה.

-יתרונות בשימוש בסכמת בלוקים:

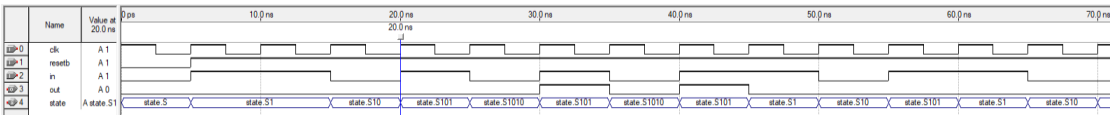
- נוחות השימוש: מימוש מעגל באמצעות דיאגרמת בלוקים הוא אינטואיטיבי ומהיר יחסית עבור מעגלים פשוטים.
- מבט על המערכת – יותר קל להבין דיאגרמת בלוקים מאשר להתחיל לעבור על קוד על מנת להבין כיצד המכונה עובדת.

3. מכונת מצבים FSM

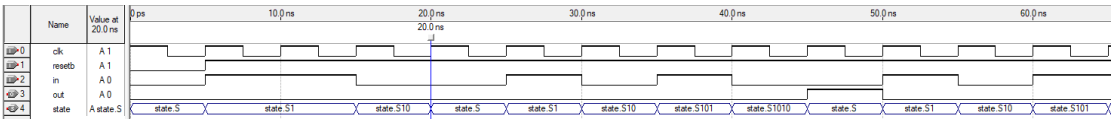
מימוש מכונת מצבים המזהה את הרצף 1010. המכונה ממומשת בקובץ יחיד (מצורף) , כאשר בתור קלט היא מקבלת אות כניסה , אות ריסט ואות שעון, ומוציאה אות מוצא שמשתנה מ0 ל1 רק כאשר הרצף 1010 נדגם.

את המצבים בחרנו לממש בצורה של מקרים (case) כאשר כל $case$ מייצג מצב במכונת המצבים והם מתחלפים בהתאם לשינוי באות הכניסה.

תוצאות עבור אות הכניסה אותו התבקשנו להכניס:



כמו כן ניסינו רצף מספרים נוסף על מנת לוודא את תקינות .

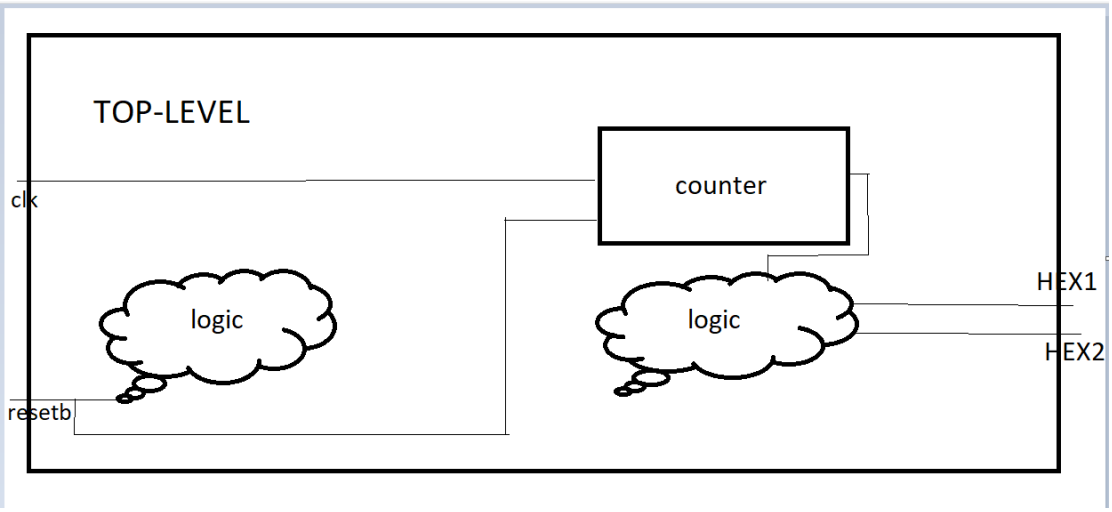


ניתן לראות כי מכונת המצבים עוברת ממצב למצב כנדרש ולפי ההגדרות אשר ניתנו בתרגיל.

4. מונה 99 שניות

נרצה להציג את המימוש שלנו למונה 99 שניות בתדר של 27Mhz. המימוש כולל TOP LEVEL ועוד תת תוכנית המכילה קאונטר.

סכמת הבלוקים של המימוש שלנו:



נציין כי בבדיקות שלנו לא ספרנו 27 מיליון עליות שעון כי זהו מספר גדול מאוד. לכן בחרנו לספור 14 עליות שעון. מאחר והמימוש עבד עבור מספר עליות שעון זה, כפי שנראה בהמשך, נדע כי הוא יעבוד גם עבור 27 מיליון עליות שעון.

תיאור המימוש:

מטרתת תת התוכנית (COUNTER) הינה לספור שניות כך שהשעון הפנימי סופר 27 מיליון עליות שעון. כאשר השעון הגיע למספר זה הגדרנו אינדיקטור שיוחלף ל-1 לוגי ואז מתחיל לספור מחדש. ניתן לראות זאת בקוד המצורף:

```
module counter(clk, resetb, indicator);
    input wire clk;
    input wire resetb;
    output reg indicator;
    reg [24:0] count;
    //we decided to use the 27Mhz system clock which requiers 25 bits
    always @(posedge clk or negedge resetb)
        begin
            if(~resetb)
                count <= 25'b0000000000000000000000000;
            else if (count == 25'b1100110111111110011000000)
            //if the theres 27M cycles, the indicator turns to
            //logic 1 and the count sums 1 each time. when it happens we know
            //that 1 second passed.
                begin
                    indicator <= 1;
                    count <= 25'b0000000000000000000000000;
                end
            else
                begin
                    count <= count + 25'b0000000000000000000000001;
                    indicator <= 0;
                end
        end
endmodule
```

כעת נסתכל על הקוד של ה- TOP LEVEL של התוכנית. שכאמור, היא מוגדרת לספור עד 99 ולאפס את המונה כאשר הוא מגיע למספר זה. בקוד שלנו הגדרנו אינדיקטור שמעלה את עצמו ב-1 עבור כל '1' לוגי שמתקבל מתת התוכנית COUNTER. בנוסף הגדרנו פרמטר לספרת האחדאות ופרמטר למספר העשרות (digit0, digit1) שמשנים את ערכם לפי האינדיקטור. פרמטרים אלה בסופו של דבר קובעים את ערכם של היציאות HEX.

```
module seconds_timer(clk, resetb, hex1, hex2);
    input wire clk;
    input wire resetb;
    //the seconds keeper lets us know what digit we need to set
    //at every moment the hex0 and hex1
    reg [6:0] seconds_indicator;
    //digit0 refers to the ones digits and digit1 refers to the tens digits
    reg [3:0] digit0;
    reg [3:0] digit1;
    output reg [6:0] hex1;
    output reg [6:0] hex2;
    //the indicator we defined in the counter module
    wire indicator;

    counter counter_inst(.clk(clk), .resetb(resetb), .indicator(indicator));
    //we get the value of the seconds indicator each second.
    always @(posedge clk or negedge resetb)
        begin
            if(~resetb)
                seconds_indicator <= 7'b0000000;
            else if (indicator)
                begin
                seconds_indicator <= seconds_indicator + 7'b0000001;
                end
            //if the seconds indicator passes the value 99, we reset it to 0
            else if (seconds_indicator > 7'b1100011)
            begin
                seconds_indicator <= 7'b0000000;
            end
```

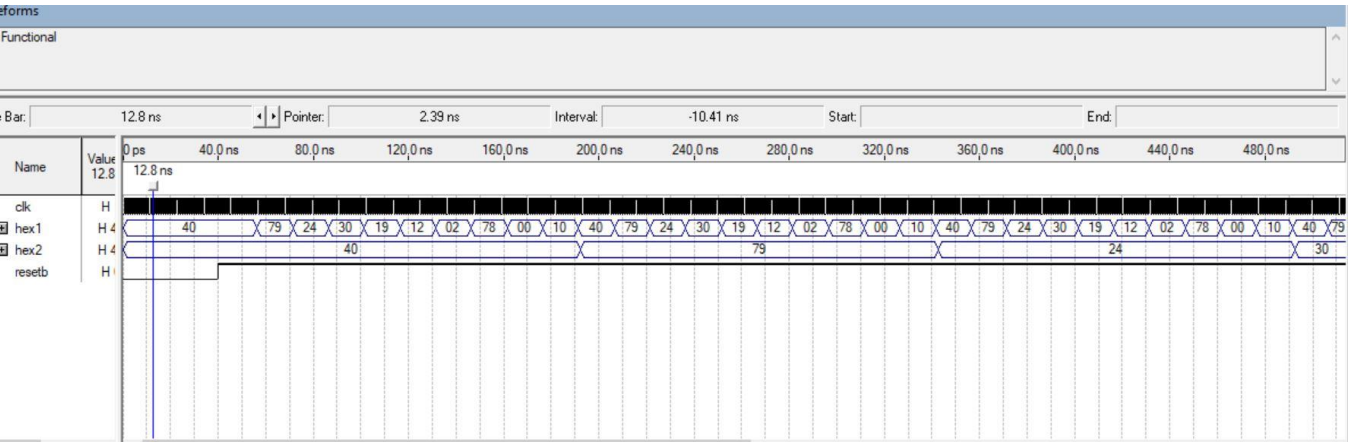
```
    always @(posedge clk or negedge resetb)
        begin

            //each second we define the digits that we will see on screen
            if (~resetb)
                begin
                digit0 <= 4'b0000;
                digit1 <= 4'b0000;
                end
            else
                begin
                digit0 <= seconds_indicator%10;
                digit1 <= (seconds_indicator-digit0)/10;
                end
            //we define hex1 and hex2 compared with the values of the
            //parameters digit0 and digit1
            case (digit0)
                4'b0000: hex1 <= 7'b1000000;
                4'b0001: hex1 <= 7'b1111001;
                4'b0010: hex1 <= 7'b0100100;
                4'b0011: hex1 <= 7'b0110000;
                4'b0100: hex1 <= 7'b0011001;
                4'b0101: hex1 <= 7'b0010010;
                4'b0110: hex1 <= 7'b0000010;
                4'b0111: hex1 <= 7'b1111000;
                4'b1000: hex1 <= 7'b0000000;
                4'b1001: hex1 <= 7'b0010000;
            endcase
```

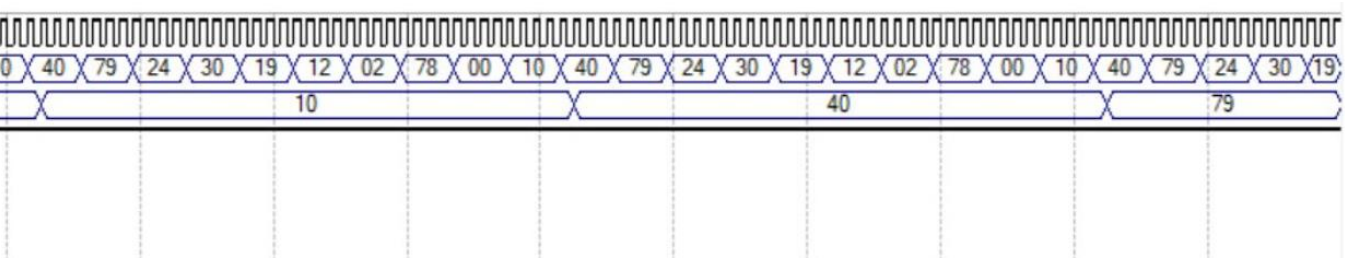
```
            case (digit1)
                4'b0000: hex2 <= 7'b1000000;
                4'b0001: hex2 <= 7'b1111001;
                4'b0010: hex2 <= 7'b0100100;
                4'b0011: hex2 <= 7'b0110000;
                4'b0100: hex2 <= 7'b0011001;
                4'b0101: hex2 <= 7'b0010010;
                4'b0110: hex2 <= 7'b0000010;
                4'b0111: hex2 <= 7'b1111000;
                4'b1000: hex2 <= 7'b0000000;
                4'b1001: hex2 <= 7'b0010000;
            endcase
        end
endmodule
```

כעת נעבור לסימולציות של התוכנית.

הגדרנו תחילה את RESETB להיות 0 בשביל להראות שאכן כלום לא משתנה. בשלב זה שתי הספרות של המונה הן 0, כלומר 40 עבור HEX.

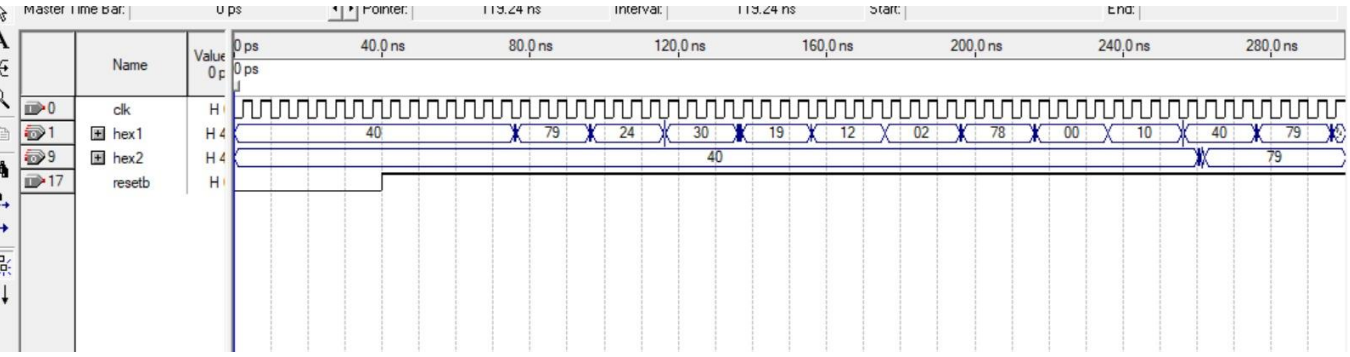
לאחר מכן RESETB משתנה ל-1 וספרת האחדות HEX1 מתחילה לגדול עד שמגיעה ל-9 ואז מתאפסת. כאשר היא מתאפסת ספרת העשרות HEX2 מתחילה היא לגדול וכן הלאה. בצילומים שנצרף נראה את התחלת המונה ואת רגע איפוס המונה כאשר הוא מגיע ל-99.





כפי שניתן לראות, המונה אכן מתאפס כאשר מגיע ל-99 וזאת כאשר שתי היציאות HEX שוות שתיהן ל-10 ואז מקבלות את הערך 40 שזה כאמור 0.

כעת נסתכל על בדיקת TIMING.



קיבלנו תוצאות עם מקטעי מספרים בין המעברים ביציאות. אנו מסיקים כי היו שינויים הקרובים מדי לעליית שעון או מנגד התרחשה קפיצה של אחד הרכיבים שגרמה לשינוי וזה עשוי לנבוע מכך שבדיקת ה- TIMING מסתכלת על דברים שלא בהכרח אידאליים.

נרצה למצוא גם את תדר המקסימלי של המונה. זאת עשינו בעזרת

Time Quest Timing Analyzer

ושם בחרנו שהתוכנה תחזיר לנו את התדבר המקסימלי. קיבלנו:

3	76.7 MHz	76.7 MHz	clk

מהו ובמה תלוי התדר המקסימלי בו המונה שלכם יכול לעבוד?

הסקתינו היא שהתדבר המקסימלי נקבע על פי פעולת המערכת, כלומר מימוש הקוד והשערים הלוגים שמתקבלים מכך. לכן, בעקבות השימוש ב-CASE בקוד שלנו המעגל שלנו דורש המון השוואות שעלולות לצרוך חומרה רבה ובכך להוריד את התדבר המקסימלי של המונה.