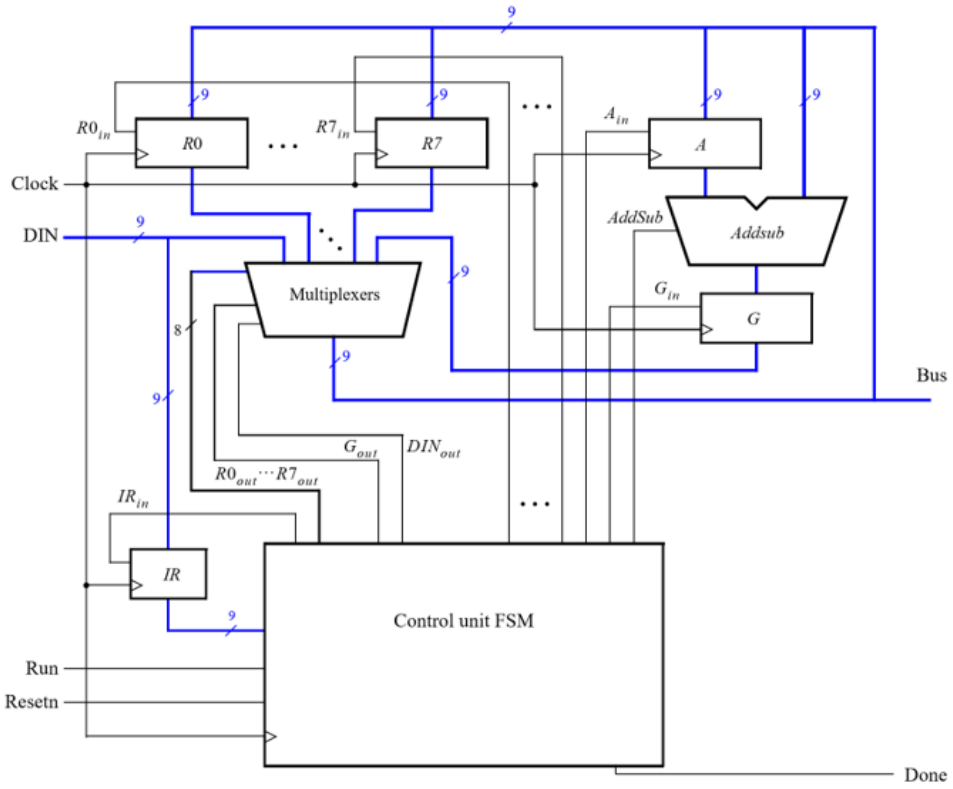
דו"ח מעבדה

מעבדה 3 – מעבד multi-cycle פשוט



במעבדה הזאת ממשנו מעבד ה-multi-cycle אשר תומך בפעולות: move, move immediate, add, subtract, specialMult. בפרויקט שלנו קראנו לspecialMult - Smult .

מבנה המעבד:

המערכת מורכבת מ-4 חלקים עיקריים:

1 - זיכרון:

במעבד קיימים 11 רגיסטרים של 9 ביטים (R0-R7,A,G,IR). הרכיבים הללו סינכרוניים ופועלים בעליית שעון.

רגיסטרים R0-R7: רגיסטרים המכילים את "המידע ממש" של המעבד.

כניסות:

- clk -אות שעון
- Resetn – ריסט לאיפוס
- BusWires – המידע שנכנס לרגיסטר ה-i
- Rin[i] – ביט enable שמגיע מהמולטיפלקסר.

יציאה:

- R[i] – הרגיסטר עצמו – מחובר למולטיפלקסר כדי להעביר את הערך השמור הלאה.

רגיסטרים A,G: מיועדים לביצוע פעולות אריתמטיות. ערכיהם נכנסים בפעולות החיבור חיסור והמכפלה ב3.5 לALU

כניסות:

- clk -אות שעון
- Resetn – ריסט לאיפוס
- BusWires – המידע שנכנס לרגיסטר
- Ain/Gin – ביט enable שמגיע מהמולטיפלקסר.

יציאה:

- Aout_bus/Gout_bus – הרגיסטר עצמו. היציאה של A מחוברת לALU לצורך ביצוע הפעולה האריתמטית. היציאה של G מחוברת למולטיפלקסר לצורך העברת הערך הנדרש לרגיסטרים R0-R7.

רגיסטר IR: שומר את הפקודה הנוכחית שיש לבצע. קבלת הפקודה היא מהכניסה DIN.

כניסות:

- clk -אות שעון
- Resetn – ריסט לאיפוס
- DIN– המידע שנכנס לרגיסטר
- IRin– ביט enable שמגיע מהמולטיפלקסר

יציאה:

- IR– מחובר ליחידת הבקרה לצורך פיענוח ומימוש הפקודה

להלן מופיע מודול הרגיסטר שניתן לנו, בו השתמשנו לטובת מימוש 11 הרגיסטרים:

```
module regn(R, Rin, Clock, Resetn, Q); //with this
//module we have implemented the 11 registers

parameter n = 9;
input wire [n-1:0] R; //Wires input
input wire Rin, //enables the reg
           Clock,
           Resetn;
output reg [n-1:0] Q; //the reg itself

always @(posedge Clock or negedge Resetn)
   if (~Resetn)
      Q <= 0;
   else if (Rin)
      Q <= R;
endmodule
```

2 – מולטיפלקסר :

יחידה א-סינכרונית שאחראית על העברת המידע בין החלקים השונים במעבד לפי אותות הבקרה המתקבלים.
סה"כ בוררת בין 10 ערוצים.

כניסות:

- רגיסטרים R0-R7
- הפקודה DIN
- מוצא רגיסטר G – Gout_bus
- Rout – אות בקרה – ערך של איזה רגיסטר עלינו להעלות על הBusWires.
- Gout - אות בקרה – להעלאת הערך שברגיסטר G על ה BusWires לטובת הכנסה לאחד הרגיסטרים.
- DINout - אות בקרה – להעלאת הערך שבכניסה DIN לאחד הרגיסטרים.

יציאה:

- mux_reg - הערך שעולה בפועל מכל הערוצים הBusWires.

להלן המימוש שלנו ליחידת המולטיפלקסר :

```
module mux10to1 (
    //registers:
    input wire [8:0] R0,
    input wire [8:0] R1,
    input wire [8:0] R2,
    input wire [8:0] R3,
    input wire [8:0] R4,
    input wire [8:0] R5,
    input wire [8:0] R6,
    input wire [8:0] R7,

    input wire [8:0] DIN, //input command
    input wire [8:0] Gout_bus, //output of G reg
    //control signals
    input wire [7:0] Rout,
    input wire Gout,
    input wire DINout,

    output reg [8:0] mux_reg //what goes out form the mux on the BusWires
    );

    reg [9:0] mux_sel;
    // assign mux_sel={Rout,Gout,DINout};
```

```
    always @(*)
    begin
    //here are all the possibilities to choose depented on the control signals
        if (Gout) begin
            mux_reg <= Gout_bus;
        end else if (DINout) begin
            mux_reg <= DIN;
        end else if (Rout == 8'b00000001) begin
            mux_reg <= R0;
        end else if (Rout == 8'b00000010) begin
            mux_reg <= R1;
        end else if (Rout == 8'b00000100) begin
            mux_reg <= R2;
        end else if (Rout == 8'b00001000) begin
            mux_reg <= R3;
        end else if (Rout == 8'b00010000) begin
            mux_reg <= R4;
        end else if (Rout == 8'b00100000) begin
            mux_reg <= R5;
        end else if (Rout == 8'b01000000) begin
            mux_reg <= R6;
        end else if (Rout == 8'b10000000) begin
            mux_reg <= R7;
        end
    end


endmodule
```

3 - ALU:

מימשנו במקום יחידת הaddsub. זוהי יחידה אסינכרונית שאחראית על ביצוע הפעולות האריתמטיות.

הפעולות הן חיבור וחיסור בין רגיסטרים והן מכפלה של רגיסטר ב3.5 (תחת הנחה שהרגיסטר בעל כפולה שלמה של 10)

כניסות:

- ALUin1 - הכניסה – הערך שמגיע מרגיסטר A
- ALUin2 - הכניסה – הערך שמגיע מרגיסטר G
- sel – סלקטור שמקבל איזה פקודה צריך לבצע.

יציאה:

- ALUout - המוצא של הALU.

להלן המימוש :

```
module ALU(
    input wire [8:0] ALUin1,    // reg A
    input wire [8:0] ALUin2,    // reg G
    input wire [1:0] sel,       // selector what operation to do
    output reg [8:0] ALUout);   // output
    //operations:
    parameter ADD = 2'b01;
    parameter SUB = 2'b00;
    parameter MULT = 2'b10;

    //implemented with blocking assignment because of combinatoric logic
    always @(*)
       case (sel)
          ADD:
          begin
             ALUout = ALUin1 + ALUin2;
          end

          SUB:
          begin
             ALUout = ALUin1 - ALUin2;
          end
          MULT:
          begin
             ALUout = (ALUin2 << 1) + ALUin2 + (ALUin2 >> 1);
          end
       endcase
endmodule
```

4- יחידת בקרה (FSM):

מדובר בלב של המעבד. ממומש כמכונת מצבים שמחליפה בין מצבים בעליית שעון.

בכל מחזור שעון מתבצע חלק מסוים של פקודה (מספר מחזורי השעון משתנה בין פקודות, בין השאר מכיוון שיש BusWires אחד) ונשלחים אותות בקרה הרלוונטים למחזור שעון העכשווי.

חלקי הFSM מומשו בקובץ proc.v – להלן חלקי מכונת המצבים:

הגדרת רגיסטרים:

```
// parameter regs
reg [1:0] Tstep_Q;    // cs
reg [1:0] Tstep_D;    // ns
```

עדכון הצמב:

```
// FSM - Resetn controling
always @(posedge Clock, negedge Resetn)
   begin
      if (!Resetn)
         Tstep_Q <= T0;
      else
         Tstep_Q <= Tstep_D;
   end
```

מעבר בין מחזורי שעון:

```
// Control FSM- cycles depented on commands
always @(*)
   begin
      case (Tstep_Q)
         T0: begin
            if (!Run)
               Tstep_D <= T0;// If not given Run prompt
            else
               Tstep_D <= T1;
         end
         T1: begin
            if (Done)
               Tstep_D <= T0; // return if command is 2 cycles long
            else
               Tstep_D <= T2;
         end
         T2: begin
            if (Done)
               Tstep_D <= T0;  //happens if Smult happens
            else
            Tstep_D <= T3;
         end
         T3: begin
            Tstep_D <= T0;
         end
         default: Tstep_D <= T0;
      endcase
   end
```

שליטה בלוגיקה:

```
// Control FSM
always @(*)
   begin

      // instance for all the regs
      IRin <= 1'b0;
      Gin <= 1'b0;
      Ain <= 1'b0;
      Done <= 1'b0;
      Rin <= 8'b0;
      Gout <= 1'b0;
      DINout <= 1'b0;
      Rout <= 8'b0;

      case (Tstep_Q)    // CURRENT STATE CASE:
         T0: begin
               IRin <= 1'b1;  // enable reg IR
            end
```

```
        T1:   // first cycle
           case(I)
                 MV:
                 begin
                    Rin <= Xreg;   // read reg X
                    Rout <= Yreg;  // write reg Y
                    Done <= 1'b1;  // enable Done
                 end
                 MVI:
                 begin
                    Rin <= Xreg;   // write reg X
                    DINout <= 1'b1;// enable DINout
                    Done <= 1'b1;  // enable Done
                 end
                 ADD:
                 begin
                    Rout <= Xreg;  // read reg X
                    Ain <= 1'b1;   // enable reg A
                 end
                 SUB:
                 begin
                    Rout <= Xreg;  // read reg X
                    Ain <= 1'b1;   // enable reg A
                 end
                SMULT:
                begin
                   Rout <= Xreg;  // read reg X
                   Gin <= 1'b1;   // enable reg G
                   AddSub <= 2'b10; //enable smult operation in alu
                end
                endcase
        T2:   // second cycle
           case(I)
                 ADD:
                 begin
                    Rout <= Yreg;  // read from reg Y
                    Gin <= 1'b1;   // enable reg G
                    AddSub <= 2'b01;  // selector for add
                 end
                 SUB:
                 begin
                    Rout <= Yreg;  // read from reg Y
                    Gin <= 1'b1;   // enable reg G in
                    AddSub <= 2'b00;  // selector for subtract
                 end
                 SMULT:
                 begin
                    Gout <= 1'b1; // enable reg G out
                    Rin <= Yreg;  // // write to reg Y
                    Done <= 1'b1;
                 end
           endcase
        T3:
           case(I)
                 ADD:
                 begin
                    Gout <= 1'b1;  // enable reg G out
                    Rin <= Xreg;   // write to reg X
                    Done <= 1'b1;  // enable Done
                 end
                 SUB:
                 begin
                    Gout <= 1'b1;
                    Rin <= Xreg;
                    Done <= 1'b1;
                 end
           endcase
     endcase
  end
```

כאשר I בcase שבקוד לעיל הוא השלושת הביטים הMSB שמציינים איזו פקודה מבוצעת (יוסבר בהמשך),

Xreg ו Yreg הם אותות הבקרה שמציינים לאיזה רגיסטר עלינו לפנות (יוסבר בהמשך)

הקלט למעבד נכנס ב DIN כקלט של 9 ביטים כIIIXXXYYY. כאשר שלושת הביטים הMSB מייצגים את הפקודה. האמצעיים מייצגים את RX מתוך 8 הרגיסטרים, הLSB מייצגים את RY מתוך 8 הרגיסטרים.
בלחיצה על RUN ובעליית שעון מתבצעת פקודה חדשה (במעבדה השתמשנו בKEY, כך שדגימת השעון קוראת בnegedge כי כך עובדים הKEYS). ביט Done נשלח במחזור שעון האחרון שלכל פקודה.

במחזור שעון הראשון של כל פקודה , הפקודה עצמה נשמרת ברגיסטר IR. ואנחנו מפרקים אותה לפי הפירוק שלעיל : נראה את החלק הרלוונטי בקוד :

```
// decode input of IR
assign I = IR[8:6];
dec3to8 decX (IR[5:3], 1'b1, Xreg);
dec3to8 decY (IR[2:0], 1'b1, Yreg);
```

וכך מקבלים את Xreg, Yreg, I במכונת מצבים שלעיל.

קבלת Xreg ו Yreg קוראת ע"י שימוש בדיקודר 3 ל8 , שמעביר ייצוג בינארי לייצוג של 8 ספרות :

```
module dec3to8(W, En, Y);
input wire [2:0] W;
input wire En;
output reg [7:0] Y;

always @(*)
   begin
      if (En == 1)
         case (W)
            3'b111: Y = 8'b10000000;
            3'b110: Y = 8'b01000000;
            3'b101: Y = 8'b00100000;
            3'b100: Y = 8'b00010000;
            3'b011: Y = 8'b00001000;
            3'b010: Y = 8'b00000100;
            3'b001: Y = 8'b00000010;
            3'b000: Y = 8'b00000001;
         endcase
      else
         Y = 8'b00000000;

   end
endmodule
```

בקובץ proc.v המצורף לקובץ זה מופיעים בנוסף כל הכניסות והיציאות של המעבד,
מימוש הFSM (כפי שצורף כאן) , האינסטנציאציות לALU, לרגיסטרים ולמולטיפלקסר.

טבלת תיאור הפקודות:

בתאים של T0 -T3 מתוארים אותות הבקרה שנדלקים במחזורי השעון הללו.

Operation	Instruction (MSB)	What happens	T0	T1	T2	T3
mv	000	RX ← [RY]	IRin	Ryout, Rxin **Done**		
mvi	001	RX ← D	IRin	DINout, Rxin, **Done**		
add	010	RX ← [RX]+[RY]	IRin	Rxout, Ain	Ryout, Gin, addsub=01	Gout, Rxin, **Done**
sub	011	RX ← [RX]-[RY]	IRin	Rxout, Ain	Ryout, Gin, addsub=00	Gout, Rxin, **Done**
smult	100	RY ← 3.5*RX	IRin	Rxout, Gin, addsub=10	Gout, Ryin, **Done**	

הסבר לפקודות:

נסביר את מה שקורה בכל פקודה, כל מחזור שעון , ועל כן אותות הבקרה שבטבלה הם הללו שצריכים לדלוק (חשוב היה לדאוג שהם כבויים כל זמן שהם אינם נדרשים להיות דלוקים)

MV - ראשית שומרים את הפקודה בIRin ב T0. במחזור שעון T1 מוציאים את הערך שיש ברגיסטר Y ומכניסים אותו לרגיסטר X , מוציאים ביט Done.

MVI - ראשית שומרים את הפקודה בIRin ב T0. במחזור שעון T1 מכניסים בDIN את הערך אותו רוצים לכתוב לרגיסטר X ומכניסים אותו לרגיסטר X ומוציאים ביט Done.

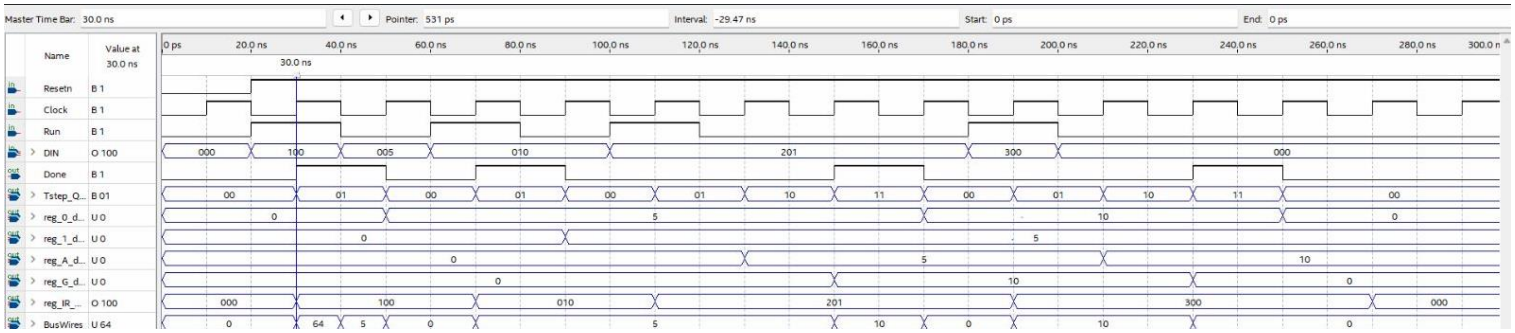
ADD – ראשית שומרים את הפקודה בIRin ב T0. במחזור שעון T1 מכניסים את הערך של רגיסטר X לרגיסטר A. במחזור שעון T2 מכניסים את הערך שברגיסטר Y לALU ביחד עם הערך שברגיסטר A ומתבצעת פעולת החיבור והערך מוכנס לרגיסטר G . בT3 הערך יוצא מרגיסטר G ונכנס לרגיסטר X ומוציאים ביט Done.

SUB - זהה לADD עד כדי חיסור בין הרגיסטרים.

Smult – ראשית שומרים את הפקודה בIRin ב T0. בT1 אנחנו מוציאים את הערך של רגיסטר X ,מבצעים את פעולת ההכפלה עליו ושומרים את התוצאה ברגיסטר G. בT2 מוציאים את הערך שיש ברגיסטר G, מכניסים אותו לרגיסטר Y ומוציאים ביט Done.

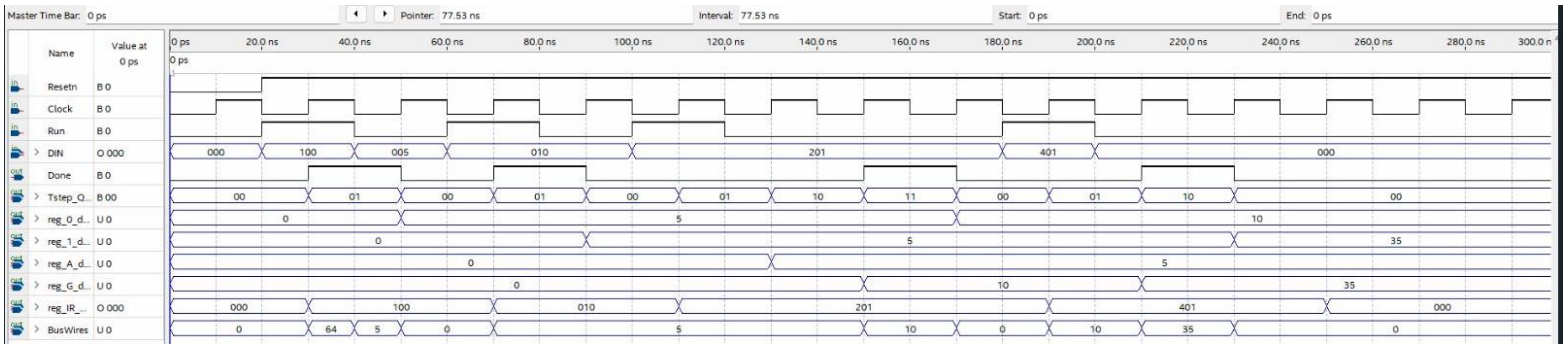
סימולציות-

נראה כעת סימולציות המראות את נכונות מימוש המעבד :



תוצאת הסימולציה זהה לזו שניתנה לנו בתרגיל

נראה סימולציה נוספת עם מימוש של הפקודה הנוספת (התחלה בזמן 180 ns) ניתן לראות שרגיסטר 1 מקבל את הערך של רגיסטר 0 מוכפל ב 3.5 (ברגיסטר 0 היה 10 – על כן נכנס לרגיסטר 1 הערך 35). כמו כן ניתן לראות שהפקודה שמימשנו לוקחת 3 מחזורי שעון, כפי שרצינו לממש.



לפי מה שביקשו בחלק א' עשינו מודול top למעבד, והצריבה ל FPGA קרתה דרכו.

נראה את המודול כאן :

```
module proc_top(DIN, Resetn, Clock, Run, Done, BusWires);

input wire [8:0] DIN; //data into the processor
input wire Resetn, Clock, Run;
output wire Done; // sent in the last cycle of each command
output wire [8:0] BusWires; //entry to regs & output of mux

//instance to the processor:S
proc proc_inst(.DIN(DIN), .Resetn(Resetn), .Clock(Clock),
               .Run(Run), .Done(Done), .BusWires(BusWires));

endmodule
```

מציאת תדר מקסימלי - מצאנו את הנתונים מהפלט של הקומפליציה :

Slow 1100mV 0C Model Fmax Summary

<<Filter>>

	Fmax	Restricted Fmax	Clock Name
1	52.04 MHz	52.04 MHz	proc:proc_inst|Tstep_Q.T1
2	79.18 MHz	79.18 MHz	Clock
3	137.06 MHz	137.06 MHz	proc:proc_inst|AddSub[0]

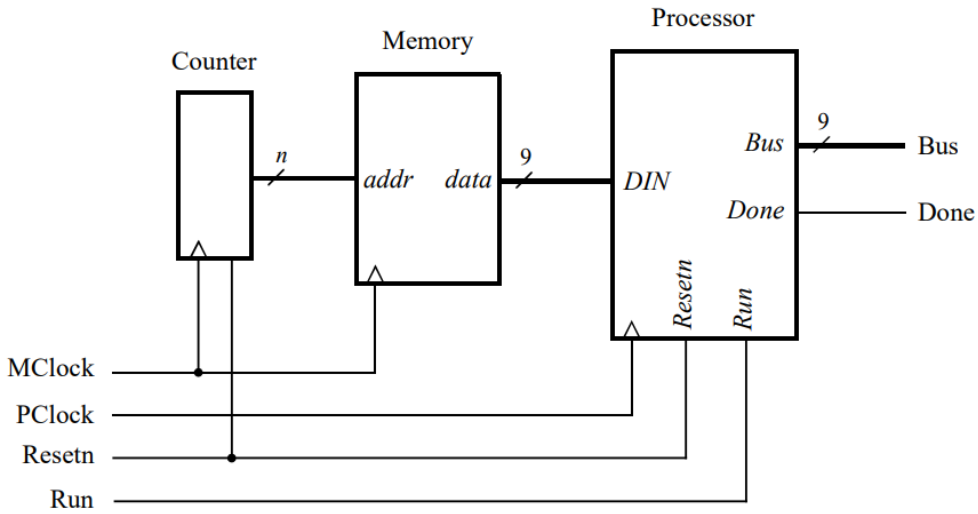
כלומר בהינתן הנתונים בשורה הכחולה השעון (Clock) המקסימלי שהמערכת תעבוד בו יהיה 79.18 MHz .

מצורף כאן קישור לסרטון להראות את נכונות המימוש לאחר הצריבה:

https://drive.google.com/file/d/1VgN2lJRZ8xGvc64mTOYu7wc-Lt9yze-t/view?usp=sharing

חלק 2 של המעבדה

בחלק השני של המעבדה נדרשנו לממש את המערכת הבאה:



כאשר מדובר במעבד אותו יצרנו כבר בחלק הקודם (עם כל הכניסות היציאות) – כעת הDIN מגיע מרכיב הזיכרון (SROM – synchronous read only memory – על כן זהו זיכרון ממנו ניתן רק לקרוא). את הפקודות בזיכרון ניתן לקדם ע״י שימוש בcounter ובכך להביא בעצם פקודות חדשות לפי מחזור שעון. נשים לב כי כעת יש שעון נפרד למעבד (Pclock) ושעון נפרד לSROM ולcounter (Mclock). הריסט משותף ל3 הרכיבים.

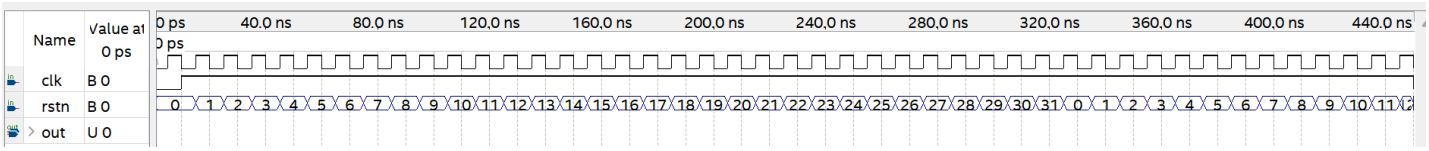
הסבר איך המערכת עובדת: הcounter סופר מ0 עד 31 (יש סה״כ 32 פקודות- יוסבר בהמשך) – ומוציא כל מחזור שעון של Mclock את הספירה הנוכחית (בציור מדובר ב n). n שבציור נכנס לזיכרון – הכניסה הזאת מתורגמת לאיזה שורה (כל שורה היא פקודה של 9 ביטים) צריך להוציא כoutput, שייכנס לDIN ומשם למעבד.

מימוש של הcounter:

```
module counter ( //counter 0 to 31 included
    input wire clk,
    input wire rstn,
    output wire [4:0] out //this enters the memory
    );
    parameter COUNT_TO = 5'b11111; //31 in binary
    reg [4:0] count; // saves the current counting

    always @(posedge clk or negedge rstn)
    begin
       if (~rstn)
          begin
             count <= 0; //no need to count
          end
       else if (count > COUNT_TO) //need to initialize
          begin
             count <= 5'b0;
          end
       else
          begin
             count <= count + 5'b1; //else: need to add 1
          end
    end
assign out = count; //sends as output current counting
endmodule
```

נראה דיאגרמת גלים לנכונות של המימוש :



ניתן לראות שהרכיב סופר עד הנדרש, ומתאפס אח״כ ומתחיל ספירה מחדש.

מימוש רכיב הזיכרון:

לצורך המימוש עקבנו אחר ההסברים שניתנו. השתמשו בIP Catalog ליצירת זיכרון של 32 מילים, כאשר כל מילה של 9 ביטים. יצרנו זיכרון ROM-1PORT , צרבנו לתוכו את הערכים של קובץ inst_mem.mif שיצרנו (נראה תכף) ושמרנו את הזיכרון כקובץ בשם inst_mem.v

קובץ הmif (memory initialization file) שיצרנו בעל 32 שורות, כל שורה של 9 ביט. כך שכל שורה מתאימה לכניסה לDIN (וכמות השורות מתאימה למה שדרשו מאיתנו). הכנסנו לקובץ mif את הפקודות הנדרשות.
יצרנו את הזיכרון פעמיים -פעם אחת עבור סימולציה שתהיה תואמת לסימולציה שהראו לנו בכיתה, ופעם נוספת עבור סימולציה שמחליפה את הפקודה 300 שהופיעה (פעולת חיסור – בין רגיסטר 0 לעצמו) לפקודה 401 שהיא הפקודה שמדגימה את נכונות הפקודה smult שקיבלנו לממש (פעולת smult - הכנסת ערך של רגיסטר 0 כפול 3.5 לרגיסטר 1)

להלן קבצי הmif :

- ללא פקודה נוספת :

```
DEPTH = 32;
WIDTH = 9;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00: 001000000;
01: 000000101;
02: 000001000;
03: 010000001;
04: 011000000;
05: 000000000;
06: 000000000;
07: 000000000;
08: 000000000;
09: 000000000;
0A: 000000000;
0B: 000000000;
0C: 000000000;
0D: 000000000;
0E: 000000000;
0F: 000000000;
10: 000000000;
11: 000000000;
12: 000000000;
13: 000000000;
14: 000000000;
15: 000000000;
16: 000000000;
17: 000000000;
18: 000000000;
19: 000000000;
1A: 000000000;
1B: 000000000;
1C: 000000000;
1D: 000000000;
1E: 000000000;
1F: 000000000;

END;
```

- בתוספת פקודת smult :

```
DEPTH = 32;
WIDTH = 9;
ADDRESS_RADIX = HEX;
DATA_RADIX = BIN;
CONTENT
BEGIN

00: 001000000;
01: 000000101;
02: 000001000;
03: 010000001;
04: 100000001;
05: 000000000;
06: 000000000;
07: 000000000;
08: 000000000;
09: 000000000;
0A: 000000000;
0B: 000000000;
0C: 000000000;
0D: 000000000;
0E: 000000000;
0F: 000000000;
10: 000000000;
11: 000000000;
12: 000000000;
13: 000000000;
14: 000000000;
15: 000000000;
16: 000000000;
17: 000000000;
18: 000000000;
19: 000000000;
1A: 000000000;
1B: 000000000;
1C: 000000000;
1D: 000000000;
1E: 000000000;
1F: 000000000;

END;
```

מודול ה mem_inst.v נוצר באופן אוטומטי, נראה את החתימה שלו (ניתן לצפות בקובץ כולו שמצורף) :

```
module inst_mem (
    address,
    clock,
    q);

    input [4:0]  address;
    input   clock;
    output   [8:0]  q;
```

כעת נראה את קובץ ה top שמאחד את המעבד עם הזיכרון ועם הcounter:

```
module part2_top(Resetn, MClock, PClock, Run, Done, BusWires);

input wire Resetn, Run;
input wire MClock, PClock; //clocks for memory and processor
output wire Done; // sent in the last cycle of command in Pclock
output wire [8:0] BusWires;

wire [8:0] q;  //output of the Srom
wire [4:0] count_out; //output of counter

// counter instance
counter counter_inst(.clk(MClock), .rstn(Resetn), .out(count_out));

// memory instance
inst_mem inst_mem(.address(count_out), .clock(MClock), .q(q));

// processor instance
proc proc_inst(.DIN(q), .Resetn(Resetn), .Clock(PClock),
               .Run(Run), .Done(Done), .BusWires(BusWires));

endmodule
```

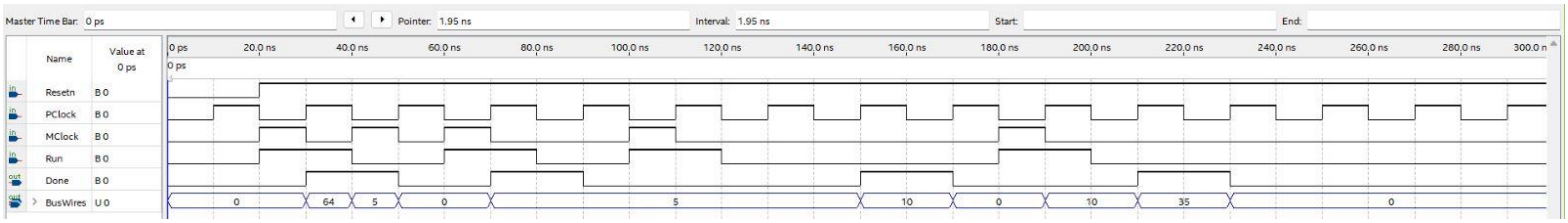
נראה כעת סימולציות לנכונות הdesign. נציין כי היה צורך לשים את הclocks בהיסט אחד מהשני, בנוסף השעונים הללו בהכרח לא חופפים, מכיוון שיש פקודות שצריכות פנייה אחת לזיכרון (לשליפת הפקודה) , ושיש את פקודת movi שצריכה פנייה כפולה לזיכרון.

סימולציות להראת נכונות הdesign:

ב2 הסימולציות ניתן לראות שהBusWires והDone זהים למה שנראה לפני חיבור המעבד לזיכרון ולcounter, והם משתנים בזמנים הנכונים

- ללא פקודה נוספת:

- עם פקודה נוספת:



צרבנו את הdesign לכרטיס לפי ההנחיות.

מצורף קישור סרטון בהינתן הפקודה הנוספת:

https://drive.google.com/file/d/1WIm2kdUMirYlhQVq6mn-fNzLsLylDfCv/view?usp=sharing