

LAB 1

ADVANCED CPU ARCHITECTURE AND HARDWARE ACCELERATORS LAB

IDO RON : 322384330

BENYAMIN OUMANSKY: 322699946

מבוא:

מטרת מעבדה 1 היא ללמוד לדעת איך להשתמש בvhdl , במעבדה ניצור מערכת top שמורכבת מתת מערכות , כך שכל תת מערכת מבצעת פעולות בנפרד לפי בחירתנו.

איך המערכת מורכבת:

: TOP

כניסות: - אות X, אות Y וקו בקרה ALUFN[4:3] שמטרתו לברור הפונקציה שאנחנו רוצים להשתמש כך:

addersub 01

shifter 10

logic 11

לכל פונקציה יש תת אופציות שאותן נבחר על ידי ALUFN[2:0].

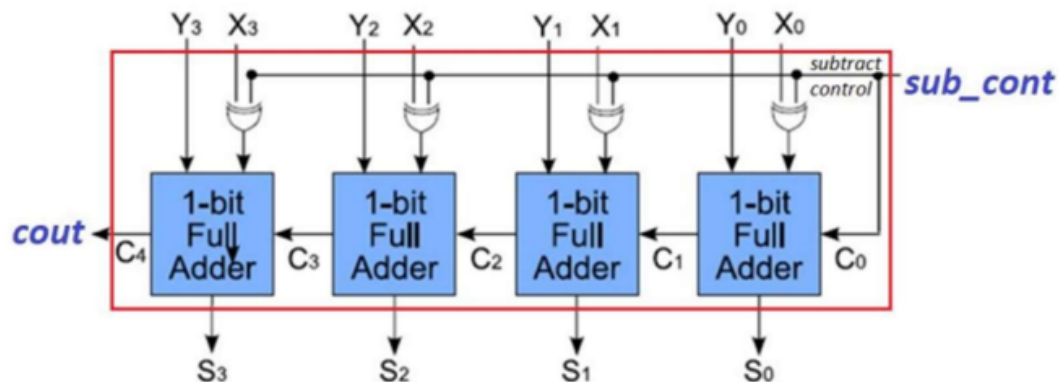
מוצא :

1. מוצא של אחת מהפונקציות שנבחר לפי [3:4] ALU

2. דגלים: בהתאם למוצא יצאו הדגלים המתאימים. הדגלים הם: Z,C,N
NEGATIVE,CARRY,ZERO

הרכיבים :

ADDERSUB



משתמשים בפונקציה כאשר $ALUFN[4:3] = "01"$, ויש לה 3 אפשרויות לביצוע: חיבור, חיסור, neg.

אנחנו השתמשנו במבנה של FA ו ripple adder שהיו נתונים לנו בדוגמאות. והוספנו והתאמנו את ה ripple adder כדי שיוכל לבצע גם חיסור ו neg.

בררנו לפי $ALUFN[2:0]$ כדי לבחור איזה תת פעולה להשתמש.

האות X נכנס ל rippleadder שמורכב מ-n רכיבי FA, בהתאם לאיור תיארונו את החומרה החדשה - לכל ביט של הוקטור x שנכנס ל-FA הוסיפו שער xor כך שכאשר sub_cont שווה ל0 יתבצע חיבור וכאשר הוא שווה ל1 יתבצע חיסור. (החיסור מתבצע על ידי שיטת המשלים ל2 כי כל הסיביות של X מתחלפות ואנחנו סוכמים $c0=1$).

כדי לממש את neg השתמשנו גם בפעולת החיסור ובכך שהגדרנו

ש $y=0$.

איך מימשנו את זה בקוד ?

הגדרנו סיגנלים מקומיים לקלטים, כדי שנוכל להתאים אותם בהתאם לתת הפעולה שנשתמש בה, ולאחר מכן השתמשנו בfa עבור כל ביט כך שנעבור על כל וקטור הקלטים המקומיים החדשים כדי לבצע את הפעולה הנדרשת.

סימולציה:

יצרנו test עבור הפעולה הזו עם כמה קלטים כדי לבדוק האם היא עובדת.

/addersub_tb/x	11111111	00010100	00001100	00101100	00001100	00000000	11111111
/addersub_tb/y	00000001	00000011	00000010	00010010	01010010	00000000	00000001
/addersub_tb/sub...	000	000	001	011	100	000	
/addersub_tb/cout	1						
/addersub_tb/s	00000000	00010111	111110110	00010011	01010001	00000000	00000000

ניתן לראות חלק מהדוגמאות הבאות (משמאל לימין):

1. חיברנו בין 20 ל 3 ואכן קיבלנו 23 עם cout=0

2. חיסרנו בין 2 ל 12 וקיבלנו בפלט 10-.

3. פעולת neg ל 48 עם cout =0

6. חיבור בין 2 מספרים עם cout=1.

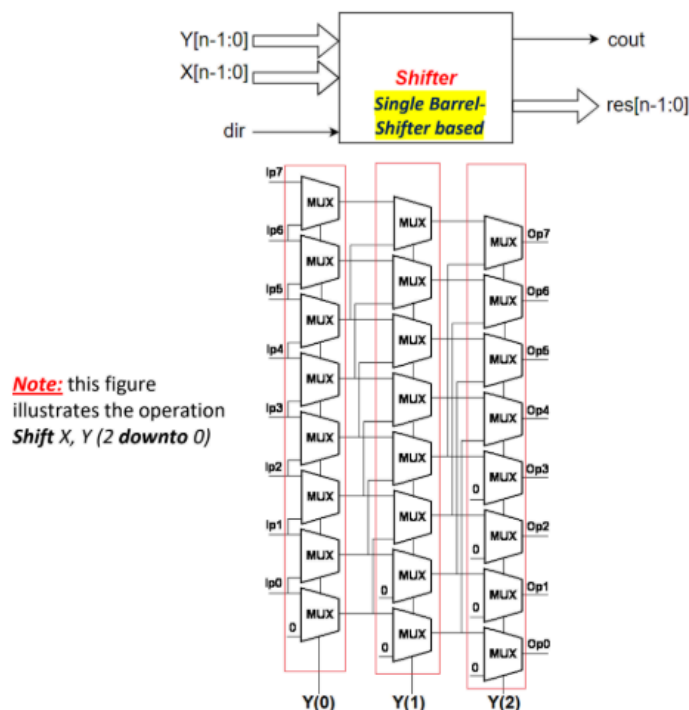
:Shifter

הרכיב יכול לבצע הזזה ימינה או שמאלה של הקלט Y .

שלושת הביטים הראשונים של ALUFN ישמרו בסיגנל direction וזה מה שיגדיר את הכיוון. נגדיר מטריצה שכל שורה בה שומרת את הערך המוזז של Y . שורה 0 שומרת את הערך המקורי של Y כלומר ללא הזזה אם ההזזה הינה שמאלה, אחרת נשמור את Y הפוך (כי מבצעים הזזה ימינה, לאחר מכן נהפוך חזרה). כך נבצע לולאה שבכל שלב תשמור את הערך המוזז של Y בהזזות שהם בחזקת השלב, כלומר בשלב הראשון נשמור בשורה 1 של המטריצה את Y מוזז פעם אחת או 0, בשורה השנייה הזזות של 2, בשורה 3 הזזות של 4 וכך הלאה. בנוסף בכל שלב נוסיף אפסים משמאל אם צריך בהתאמה לשלב. הקלט X הוא זה שמגדיר את מספר הזזות שנעשה בכל שלב כלומר ביט ה LSB קובע הזזה של 1, הביט השני הזזות של 2 וכך הלאה.

בסוף הלולאה בשורה האחרונה של המטריצה נקבל את הערך הסופי המוזז של Y אך אם אנחנו מבצעים הזזה ימינה אנו הופכים את הוקטור ושומרים אותו בפלט של הרכיב.

בנוסף מבחינת הCOUT בכל שלב נשמור את הספרה האחרונה ש"יוצאת" בתוך סיגנל (לדוגמא בשלב השני זו הספרה השנייה ש"יוצאת", בשלב השלישי הספרה הרביעית ש"יוצאת"), כך בלולאה עוברים שלב שלב על כל שורה של המטריצה ואם התבצעה הזזה באותו השלב אז נעדכן את הCOUT אחרת ללא שינוי. בסוף נוציא אותו כפלט.



סימולציה:

/shifter_tb/x	00000010	00000001	00000010	00000100	00000010	00000010
/shifter_tb/y	00100001	00001000	00010010	00011100	00100001	00100001
/shifter_tb/direction	001	000	001	000	001	001
/shifter_tb/cout	0					
/shifter_tb/res	00001000	00010000	00000100	11000000	00001000	00001000

בדוגמא השמאלית הראשונה ניתן לראות שלקחנו את הוקטור 00001000 והזזנו אותו במקום אחד שמאלה.

בדוגמא האחרונה מימין הזהנו את 2 ביטים ימינה.

: Logic

נבצע פעולות לוגיות על X,Y :

Not y , y or x , y and x , y xor x , y nor x , y nand x,y xnor x

חלק מהפעולות הלוגיות לא קיימות בחבילה , לכן יצרנו אותן על ידי NOT,AND ,XOR

סימולציה : לקחנו רווח של 40 ms לכל דוגמא

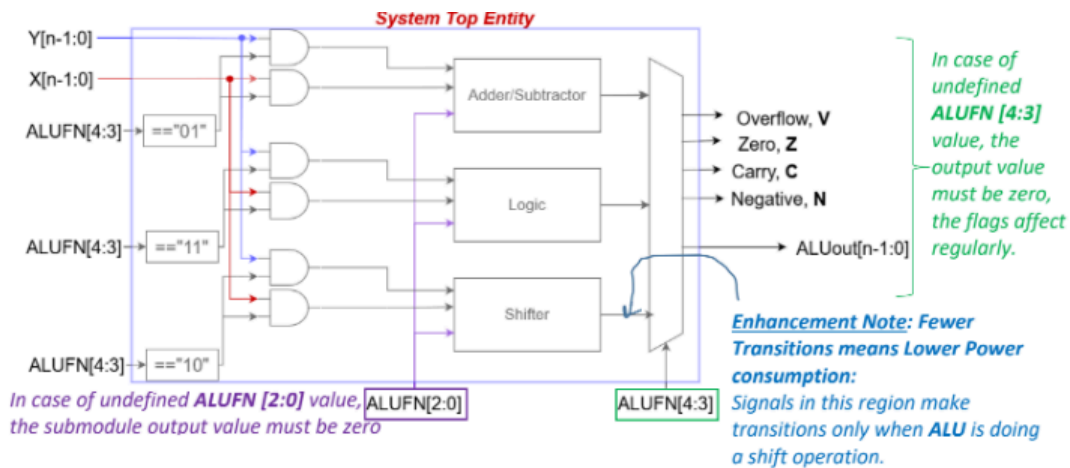
/logic_tb/x	00000000	00000100	00000010	00100100	00000011	11000100	00010100	11111111	00000000
/logic_tb/y	00000000	00000010	00001100	00001010	00001111	01000010	00000011	00000000	00000000
/logic_tb/res_logic	11111111	11111110	00001110	00000000	00001100	00111001	11111111	00000000	11111111
/logic_tb/alufn_logic	000	000	001	010	011	100	101	111	000

בדוגמא הרביעית משמאל אנחנו מבצעים XOR בין X=00000011 וY=00001111 ולפי טבלת האמת של xor אכן קיבלנו במוצא 00001100

TOP

זו המערכת שעוטפת את כל הפעולות שהגדרנו, היא מקבלת 2 קלטים X, Y , קלט שאחראי לברור איזה תת מודול להשתמש. מוציאה בפלט דגלים ואת המוצא. נגדיר שרק פעולה אחת תעבוד בהתאם ל $alufn$ בזמן שהשאר לא.

כל תת מודול מחזיר תוצאה, ו $cout$, ה Top בוחר מביניהן את התוצאה הסופית בהתאם ל $ALUOUT$ ובנוסף ה Top מחשב ארבעה דגלים (Zero, Carry, Negative Overflow) – על סמך התוצאה הסופית, $cout$ של תת המודול שבחרנו והתנאים המתאימים לכל דגל.



סימולציה:

נריץ את הטסט tb_ref1 :

	Msgs	
/tb/Y	11011000	11101100 11101011 11101010 11101001 11101000 11100111 11100110 11100101 11100100
/tb/X	01111001	01000001 00110111 00101101 00100011 00011001 00001111 00000101 11111011 11110001
/tb/ALUFN	00100	10001 10010 10000 10001 10010
/tb/ALUout	00000000	01110110 00000000 01001000 11010000 00000001 00000111 00000000
/tb/Nflag	0	
/tb/Cflag	0	
/tb/Zflag	1	
/tb/Vflag	0	

ואכן קיבלנו את התוצאות שציפנו להן.

ונסתכל גם הlist , ונשווה אותו לlist שבמשימה ונשים לב שהם זהים מלבד הדלתאות (שהן ניתנות להזנחה). **ימין** - מהמשימה **שמאל**- פלט שלנו

ps delta	/tb/Y	/tb/X	/tb/ALUout	/tb/ALUFN	/tb/Nflag	/tb/Cflag	/tb/Zflag	/tb/Vflag
0 +7	11111111	11111111	01000	11111110	1	1	0	0
50000 +8	11111110	11110101	01000	11110011	1	1	0	0
100000 +7	11111101	11101011	01001	00010010	0	1	0	0
150000 +8	11111100	11100001	01001	00011011	0	1	0	0
200000 +8	11111011	11010111	01010	00101001	0	0	0	0
250000 +6	11111010	11001101	01010	00110011	0	0	0	0
300000 +7	11111001	11000011	01011	11111010	1	0	0	0
350000 +7	11111000	10111001	01011	11111001	1	0	0	0
400000 +8	11110111	10101111	01100	11111010	1	1	0	0
450000 +7	11110110	10100101	01100	11111010	1	1	0	0
500000 +10	11110101	10011011	01000	10010000	1	1	0	0
550000 +8	11110100	10010001	01000	10001010	1	1	0	0
600000 +8	11110011	10000111	01001	01101100	0	1	0	0
650000 +9	11110010	01111010	01001	01110101	0	1	0	1
700000 +7	11110001	01110011	01111	00000000	0	1	0	0
750000 +1	11110000	01101001	01111	00000000	0	1	0	0
800000 +9	11101111	01011111	10000	10000000	1	1	0	0
850000 +9	11101110	01010101	10000	11000000	1	0	0	0
900000 +9	11101101	01001011	10001	00011101	0	1	0	0
950000 +8	11101100	01000001	10001	01110110	0	0	0	0
1000000 +5	11101011	00110111	10010	00000000	0	1	0	0
1050000 +1	11101010	00101011	10010	00000000	0	1	0	0
1100000 +9	11101001	00100011	10000	01001000	0	1	0	0
1150000 +9	11101000	00011001	10000	11010000	1	1	0	0
1200000 +7	11100111	00001111	10001	00000000	0	1	0	0
1250000 +9	11100110	00000101	10001	00000000	0	1	0	0
1300000 +5	11100101	11111011	10010	00000000	0	1	0	0
1350000 +1	11100100	11110011	10010	00000000	0	1	0	0
1400000 +5	11100011	11101111	10010	11111011	1	0	0	0
1450000 +5	11100010	11101101	10010	11111011	1	0	0	0
1500000 +6	11100001	11010011	10010	11000001	1	0	0	0
1550000 +6	11100000	11010001	10010	11000000	1	0	0	0
1600000 +6	11011111	10111111	11101	01100000	0	0	0	0
1650000 +6	11011110	10110101	11101	01101011	0	0	0	0
1700000 +6	11011101	10101011	11111	10001001	1	0	0	0
1750000 +5	11011100	10100001	11111	10000010	1	0	0	0
1800000 +5	11011011	10010111	11011	01001100	0	0	0	0
1850000 +5	11011010	10001011	11011	01010111	0	0	0	0
1900000 +3	11011001	10000011	01000	00000000	0	0	1	0
1950000 +1	11011000	01111001	01000	00000000	0	0	1	0
2000000 +1	11011011	01101111	01000	00000000	0	0	1	0


```
1950000 +1 |11011000 01111001 00100 00000000 0 0 1 0  
2000000 +1 |11010111 01101111 00100 00000000 0 0 1 0
```