The Iby and Aladar Fleischman
Faculty of Engineering
Tel Aviv University

הפקולטה להנדסה
ע"ש איבי ואלדר פליישמן
אוניברסיטת תל אביב

מודולים ותוכנה עבור פרויקט

מערכי פילטרים על גבי FPGA

פרויקט מס' 22-1-1-2538

מבצעים:

ליעד פנקר 315991109

דורית בס 320471410

מנחים:

מאיר אלון

מקום ביצוע הפרויקט:

אוניברסיטת תל אביב

1.1. <u>תיאור תוכנה</u>

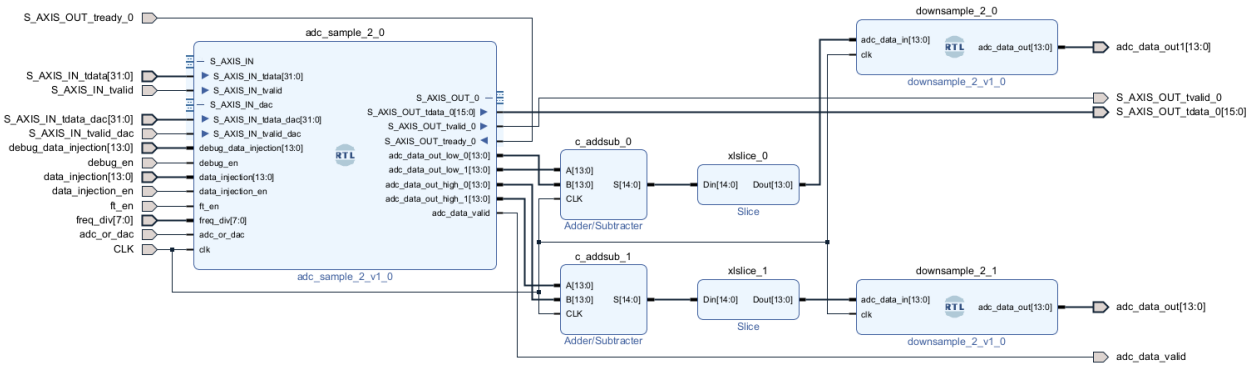
את החלק התוכנתי של הפרויקט ביצענו בעזרת תוכנת ה Vivado וכן Matlab

<u>תיאור מנגנון הdecomposition</u>

מנגנון ה decomposition מומש כפי שתיארנו ברקע התאורטי באיור 3,
כשאר את פילטרים מימשנו בעזרת Xilinx IP's, ומודל adc_sample שכתבנו:

נוסחאות הפילטרים הם, כפי שראינו ברקע התיאורטי:

$$L(x)_k = (l * x)_{2k} = \frac{1}{2}x_{2k} + \frac{1}{2}x_{2k+1}$$
$$H(x)_k = (h * x)_{2k} = \frac{1}{2}x_{2k} - \frac{1}{2}x_{2k+1}$$



איור 1-בלוק decomposition מתוך ה-Vivado

במודול adc_sample_2_0 אנחנו שומרים את הדגימה ה-n ואת הדגימה ה-n+1.
c_addsub_0 מממשים את פעולת החיבור וב-c_addsub_1 את פעולת החיסור, נקבל:

$$L(x)_n = (l * x)_n = x_n + x_{n+1}$$
$$H(x)_n = (h * x)_n = x_n - x_{n+1}$$

xslice_0 מממשים את פעולת החלוקה ב2 של ה-lowpass וב-xslice_1 את החלוקה ב2 של ה-highpass, נקבל:

$$L(x)_n = (l * x)_n = \frac{1}{2}x_n + \frac{1}{2}x_n$$
$$H(x)_n = (h * x)_n = \frac{1}{2}x_n - \frac{1}{2}x_{n+1}$$

כעת מבצעים down sample של שני הפילטרים, זורקים כל דגימה שנייה, downsample_2_0 עבור הLowpass ו-

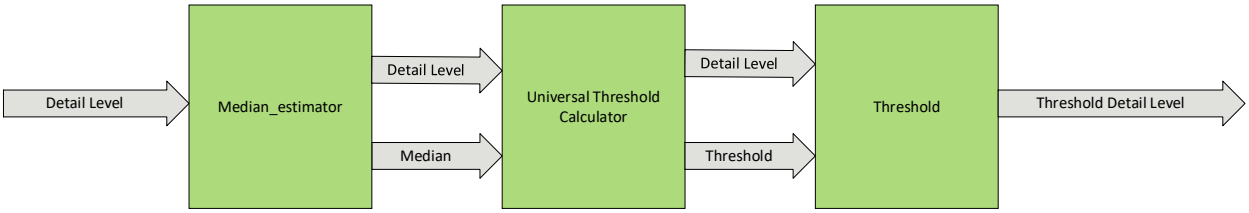
downsample_2_1 עבור ה-high pass, נקבל לבסוף את הפילטרים הרצויים ואת ה-WT level 1 :

$$L(x)_k = (l * x)_{2k} = \frac{1}{2}x_{2k} + \frac{1}{2}x_{2k+1}$$
$$H(x)_k = (h * x)_{2k} = \frac{1}{2}x_{2k} - \frac{1}{2}x_{2k+1}$$

באותו אופן מתבצע גם ה2 wavelet level.

תיאור מנגנון הThresholding

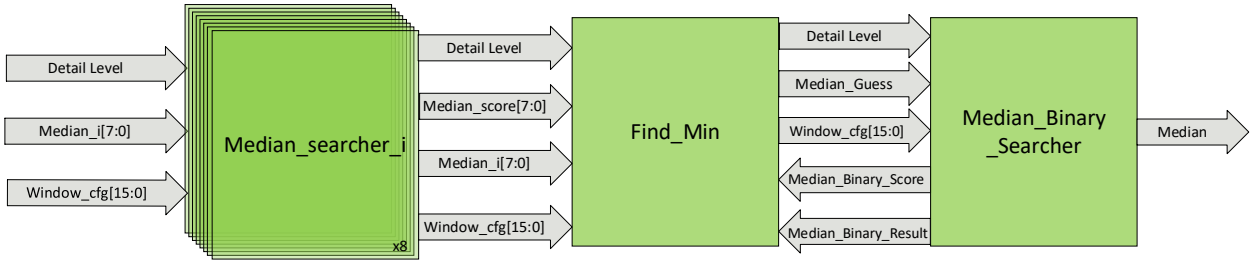
Thresholding module:



איור 2- סכמת בלוקים עבור של מנגנון ה-Thresholding

כפי שתואר ברקע התיאורטי, את הthresholding אנחנו מבצעים על הdetail levels , במקרה שלנו detail level 1 ו- detail level 2, ולכן בלוק הזה ישוכפל פעמים.

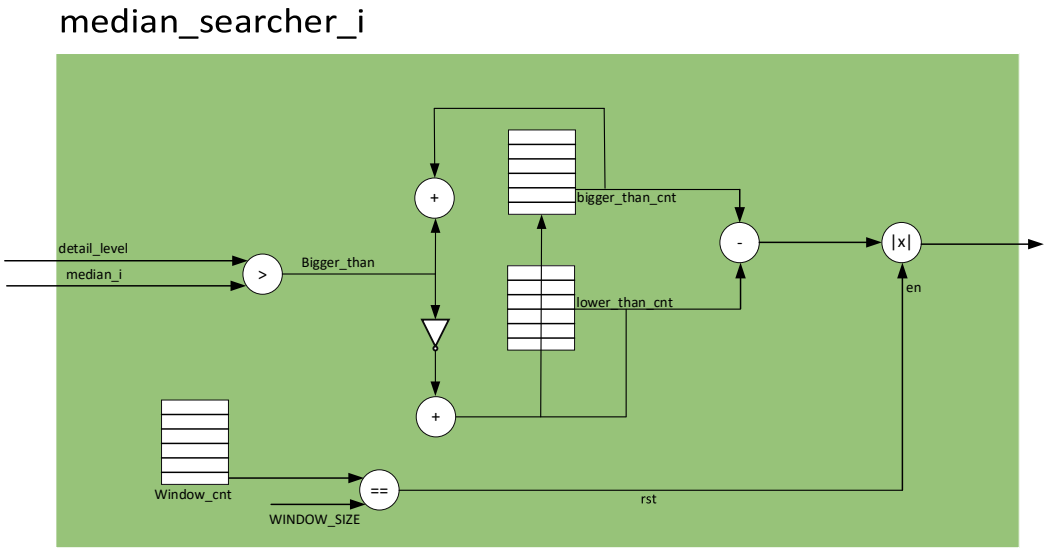
Median_estimator module:



איור 3- סכמת בלוקים המתארת את שכפול מנגנון ה-Thresholding עבור גרות Wavelet השונות

מודול Median_estimator משערך את החציון של הdetail levels, המודול מקבל 8 candidates התחלתיים, חזקות של 2: 8,16,32,64,128,256,512,1024, בעזרת Median_searcher_i מחשב את המרחק של המועמד מהערך של המידיאן האמיתי, לאחר מכן בעזרת מודול Find_min מוצא את המועמד הטוב ביותר, ולאחר מכן באיטרציות מבצע חיפוש בניארי בטווח של החציון כדי למצוא תוצאה אופטימלית יותר.

- Median_searcher_i

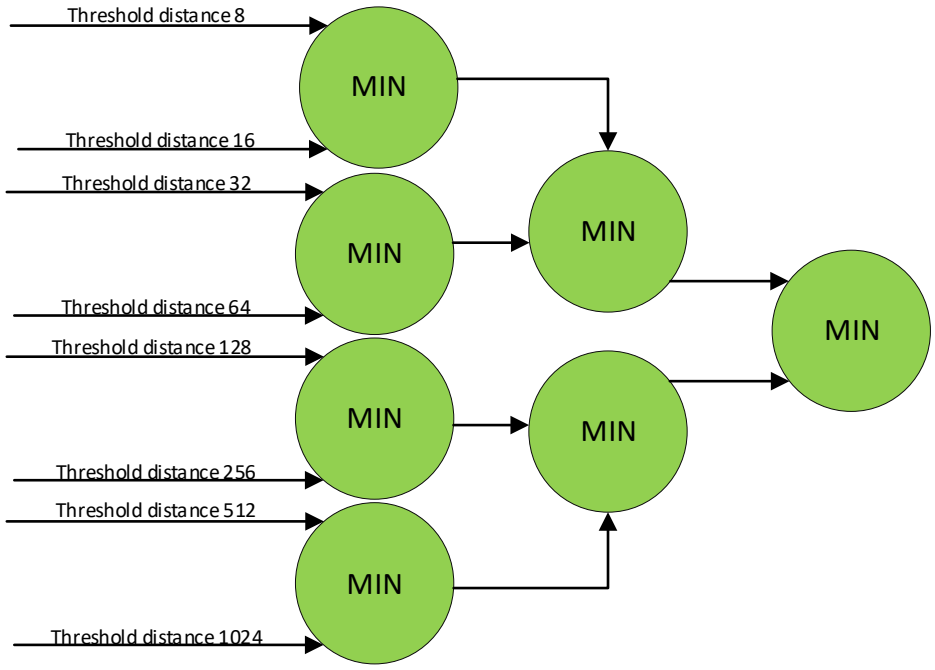


איור 4- תיאור מנגנון חיפוש החציון

מבצע השוואה של מועמד אפשרי לחציון עם הdetail level ובודק כמה ערכים מעל מועמד וכמה ערכים מתחת למועמד,
לאחר מכן מחסיר ביניהם ומחשב ערך מוחלט, ובכך מקבל את המרחק של המעומד מהחציון.
בודקים במקביל 8 מועמדים (יש מועמד 9 בחיפוש הבינארי, נגיע לזה בהמשך)

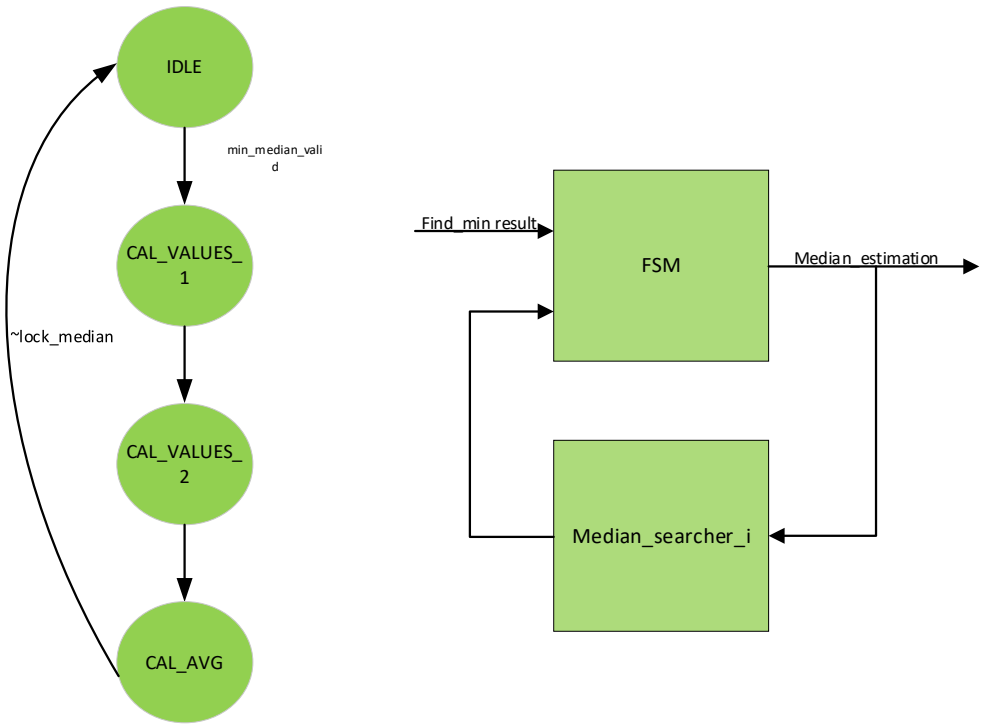
- Find Min

בלוק שמקבל את כל 8 המועמדים ומוציא את המועמד הכי טוב. נעשה על ידי מציאת מינימום מבין המרחקים המינימליים של 8 המועמדים, מחשב זאת בצורת עץ והשוואת 2 מועמדים כל פעם:



איור 5- מציאת המרחק המינימאלי מבין המועמדים

– Binary_searcher



איור 6- מימין דיאגרמת ניחוש הפתרון והשוואה לתוצאות. משמאל, מכונת המצבים של חיפוש החציון

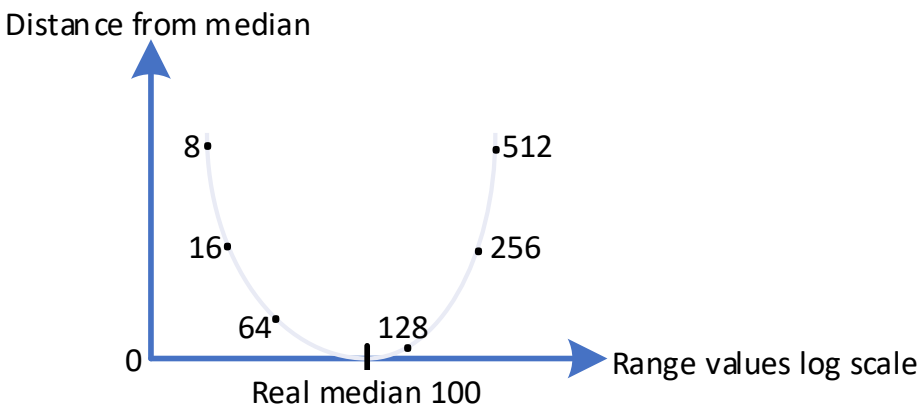
במודול החיפוש הבינארי אנחנו מחפשים בסביבה של המועמד הכי טוב שערוך מדויק יותר של החציון. כל פעם נחשב מועמד נוסף בעזרת חיפוש בינארי ונבדוק אותו בחלון, במידה והמועמד טוב יותר מ8 המועמדים הקודמים ואנחנו מקבלים שיפור בתוצאה, אנחנו שומרים אותו בתור בחציון הזמני וממשיכים בחיפוש עד התכנסות לחציון.
במידה ואחד מ8 המועמדים יותר טוב, מתחילים את החיפוש בינארי מההתחלה.

נפרט כעת יותר כיצד כל התהליך של הmedian estimation מתבצע בעזרת דוגמה:

כזכור חילקנו את הטווח שלנו ל-8 חלקים לפי חזקות של -2 (כל הcandidates ההתחלתיים שלנו)

איור 7- ציר חלוקה עבור חיפוש בינארי

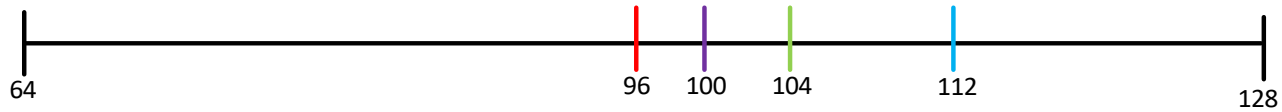
נניח לדוגמא ש-החציון האמיתי שלנו במערכת הוא 100. אנחנו לא יודעים זאת ורוצים למצוא אותו. באיטרציה הראשונה נקבל ממודל Find_Min את הערך 128 שהתקבל עקב היותו המרחק הקצר ביותר מהחציון בחלון. נוכל לראות באיור למטה כי החציון שלנו הוא בין 64 ל128.



איור 8- גרף הדגמה עבור חיפוש החציון האמיתי

נתחיל לבצע חיפוש בינארי בחלון זה באיטרציות, כאשר בכל איטרציה נזכור את הערך החדש ונבצע עליו בדיקה, הבדיקה נעשית בעזרת מודל median_searcher_i שתיארנו למעלה, שמוצא את המרחק מהחציון האמיתי, כאשר כל איטרציה תקרב אותנו לערך החציון האמיתי.

למשל עבור ערך חציון של 100 נצטרך לבצע 6 איטרציות(כל איטרציה בגודל חלון) במקסימום עד שנגיע לערך האמיתי במקרה הזה.
$(\log_2(128-64) = 6)$



איור 9-ציר הדגמה של חיפוש בינארי עבור מציאת החציון

אנחנו יוצאים ממצב idle כאשר מודול Find_min מספק לנו את הcandidate הראשוני(128 בדוגמא), שומרים את הערך האופטימאלי של החציון שקיבלנו(128), ואת אחד מהגבולות הצמודים, עליון או תחתון, תלוי האם החציון נמצא מעל הסף שקיבלנו או מעליו. למשל אם נמשיך עם הדוגמא הקודמת, בה החציון שלנו הוא 100 וקיבלנו בפעול את הערך 128, אנחנו נשמור או את 64 או את 256 תלוי באינדיקטור האם החציון קרוב יותר לגבול התחתון(64) או העליון(256). כיוון שאנחנו יודעים שהוא קרוב ל128 אבל קטן ממנו נשמור את 64, ואת תוצאת הlog של הטווח (6) שמציינת את מספר האיטרציות המקסימלי עד להתכנסות).

נעבור למצב CAL_VALUES_1 , במצב זה אנחנו משווים את התוצאה שקיבלנו מFind_min בחלון הנוכחי עם התוצאה של החיפוש הבינארי ובודקים האם החיפוש הבינארי נתן תוצאה טובה יותר, (שלב זה חשוב כי יכול להיות שהרעש במערכת השתנה וכעת החציון שלנו כבר לא בטווח ששיערנו באיטרציה הקודמת) אם אכן שיפרנו- כלומר התקרבנו לערך החציון האמיתי אנחנו נמשיך לבצע עוד איטרציה שנתכנס, אם זה לא המצב ולא קיבלנו שיפור כנראה שערך החציון שלנו השתנה ונתחיל את החיפוש הבינארי מחדש.

בשלב CAL _VALUES_2 אנחנו בודקים האם החציון נמצא או שהגענו למספר איטרציות מקסימאלי.
בשלב CAL_AVG – נחשב את ערך החציון החדש עבור האלגוריתם ועובר median_seracher_i ונחזור לIDLE.

<u>- Universal_threshold_calculator module</u>

כעת לאחר שמצאנו את החציון אנחנו מחשבים את הuniversal_threshold כפי שתיארנו אותו ברקע התיאורטי:

$$\lambda = \sigma\sqrt{2\ln(N)}$$

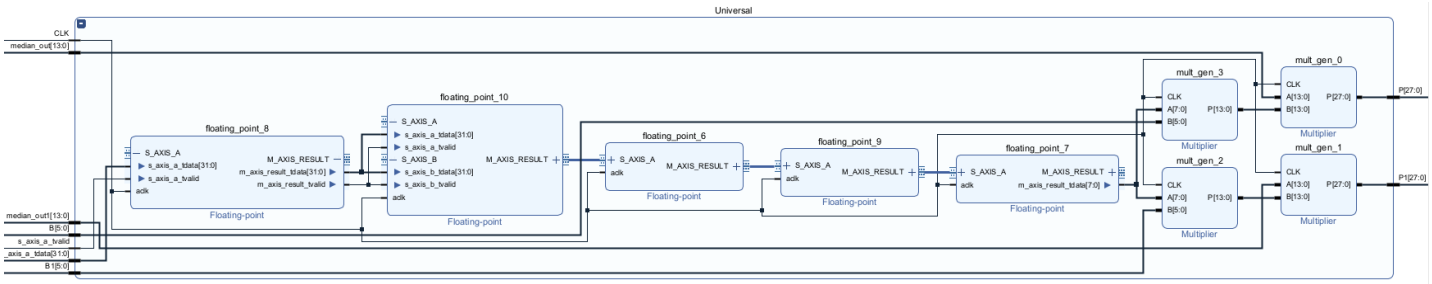
זהו הthreshold שלנו, כאשר σ הוא החציון שמצאנו מוכפל בקבוע:

$$\sigma_i = median_i \cdot C_i \quad , i\epsilon\{1,2\}$$

כאשר C_i– קונפיגורציה fix point 6_2 (מספר בגודל 6 ביט כאשר נקודה העשרונית היא אחרי הספרה השנייה).
לפי הרקע התיאורטי היינו אמורים לחלק ב0.6745 אך בשביל לאפשר גמישות ואופטימיזציה לthreshold הוספנו את האפשרות לכפול בקבוע.
לרוב משתמשים בערך דיפולטי של 1.5, שזה בקירוב 1.48=1/0.6745.

N – זהו אורך הסיגנל שלנו שבו אנחנו מחשבים את החציון, גם כן קונפיגורציה.

<u>דיאגרמה של המודול:</u>



איור 10-דיאגרמת המודל השלם של חיפוש בינארי

Floating_point_8 – ממיר מספר int32 ל- float single precision, ממיר את אורך הסיגנל לייצוג float.
Floating_point_10 – מכפיל את אורך הסיגנל בייצוג float בעצמו – מחשב את N^2.
Floating_point_6 – מחשב ln , כלומר $\ln(N^2) = 2\ln(N)$.
Floating_point_9 – מחשב שורש ריבועי, $\sqrt{2\ln(N)}$.
Floating_point_8 – ממיר מ-float single precision ל-_2fix point 6, סיגנל בגודל 8 ביט שהנקודה העשרונית אחרי 2 ביט (כמו הקבוע C_i).

לאחר מכן יש לנו ארבעה mult_gen modules שמבצעים הכפלה רגילה, כל $median_i$ מכפילים $\sqrt{2\ln(N)}$ ואז מכפילים בC_i
ובגלל שיש רק 2 detail levels ($i \in \{1,2\}$) יש לנו ארבעה מכפלים.
לבסוף לתוצאה עושים shift ימינה 2 כדי להעלים את הנקודה העשרונית.

<u>– Threshold module</u>

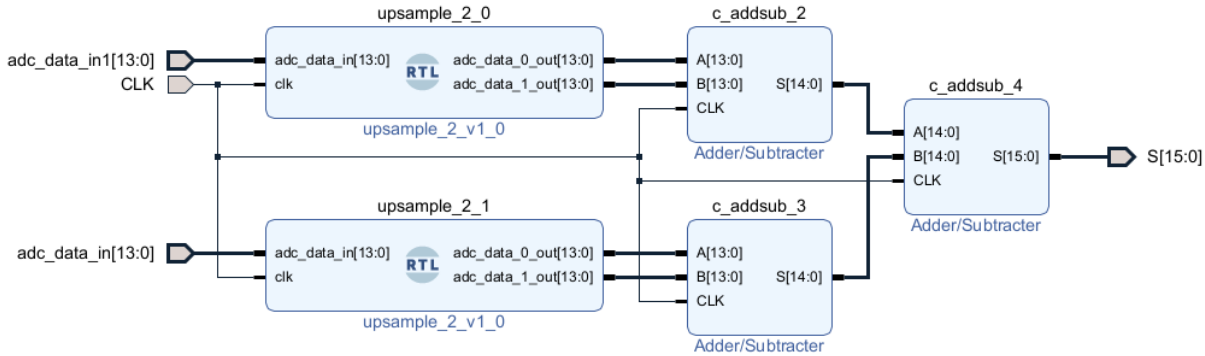
לאחר שחישבנו את הthreshold, אנחנו מבצעים soft threshold או hard threshold כפי שתיארנו ברקע התאורטי.

<u>תיאור מנגנון הreconstruction</u>
לאחר שביצענו את ה-thresholding אנחנו יכולים לבנות את הסיגנל מחדש נטול הרעשים.
הreconstruction כפי שראינו ברקע התיאורטי, איור 5, גם פה הפילטרים מומשו בעזרת Xilinix_ip:
נוסחת הreconstruction כפי שראינו ברקע התיאורטי היא:

$$\left(\tilde{L} * x\right)_l = \begin{cases} x_{2k} & l = 2k \; is \; even \\ x_{2k} & l = 2k+1 \; is \; even \end{cases}$$

$$\left(\tilde{H} * y\right)_l = \begin{cases} y_{2k} & l = 2k \; is \; even \\ -\,y_{2k} & l = 2k+1 \; is \; even \end{cases}$$

$$\left(\tilde{L} * x\right)_l + \left(\tilde{H} * x\right)_l = \begin{cases} x_{2k} + y_{2k} & l = 2k \; is \; even \\ x_{2k} - \; y_{2k} & l = 2k+1 \; is \; even \end{cases}$$



איור 11 - סכמת מנגנון ה-reconstruction

Upsample – מבצע העלאת קצב, מוצא המודול יהיה:

$$(L*x)_{2k} = \begin{cases} x_{2k} & l = 2k \text{ is even} \\ 0 & l = 2k+1 \text{ is even} \end{cases}$$

$$(H*y)_{2k} = \begin{cases} y_{2k} & l = 2k \text{ is even} \\ 0 & l = 2k+1 \text{ is even} \end{cases}$$

c_addsub_2 ממש את פעולת החיבור וב-c_addsub_3 את פעולת החיסור, נקבל:

$$(L*x)_{2k} = \begin{cases} x_{2k} & l = 2k \text{ is even} \\ x_{2k} & l = 2k+1 \text{ is even} \end{cases}$$

$$(H*y)_{2k} = \begin{cases} y_{2k} & l = 2k \text{ is even} \\ -y_{2k} & l = 2k+1 \text{ is even} \end{cases}$$

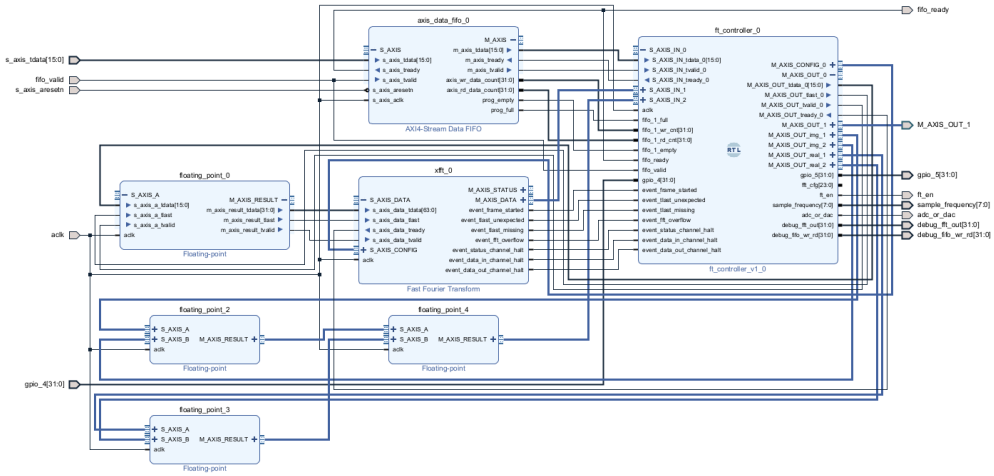
כעת c_addsub_4 ממש פעולה חיבור ונקבל את הreconstructed signal :

$$\left(\tilde{L}*x\right)_l + \left(\tilde{H}*x\right)_l = \begin{cases} x_{2k} + y_{2k} & l = 2k \text{ is even} \\ x_{2k} - y_{2k} & l = 2k+1 \text{ is even} \end{cases}$$

מודול זה משוכפל פעמיים, עבור 2 ה-levels.

FT and STFT

בנוסף להתמרת WT וה-IWT שמימשנו, ישנה אפשרות לחשב את ה-FT וה- STFT במערכת:

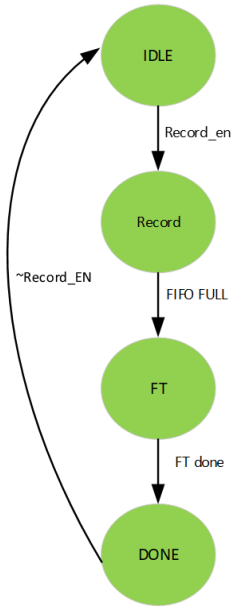


איור 12- סכמת מנגנון ה-FT וה-STFT

- Ft_controller

המודול שמנהל את FFT, מורכב בעיקר ממערכת מצבים וזיכרון BRAM ששומר את התוצאות של הFFT כפי שניתן לראות באיור 23.

יוצאים ממצב idle בעזרת פקודת AXI בקונפיגורציה להפעלת החישוב הFFT, ועוברים ל-record state, לפני שעוברים לrecord mode צריך לעדכן גם כן בעזרת קונפיגורציה את אורך הFFT שרוצים לעשות לפי טבלה 9.



איור 13-מכונת המצבים עבור הFT-controller

FFT length	Window_numbers
4096	1 window
2048	2 windows
1024	4 windows
512	8 windows
256	16 windows
128	32 windows
64	64 windows

טבלה-1 -חלוקת החלונות כתלות האורך הFFT

בשלב הזה אנחנו מקליטים בעזרת מודול FIFO – axi_data_fifo_0, 4096 דגימות של הסיגנל (ניתן לשלוט על תדר הדגימה של הסיגנל בעזרת קונפיגורציה), לאחר שסיימנו להקליט וה-fifo מלא, עובר לשלב ביצוע הFT.

חישוב ה- FT מתבצע במודול xfft_0 של xillinx , לצורך נוחות, ממירים את ה-signal ל-single floating point בעזרת המודול floating_point_0 כדי שלא נצטרך לקנפג את גדלי המוצא בכל level של חישוב הFFT.
זמן חישוב של הtransform משתנה כתלות באורך הFFT:

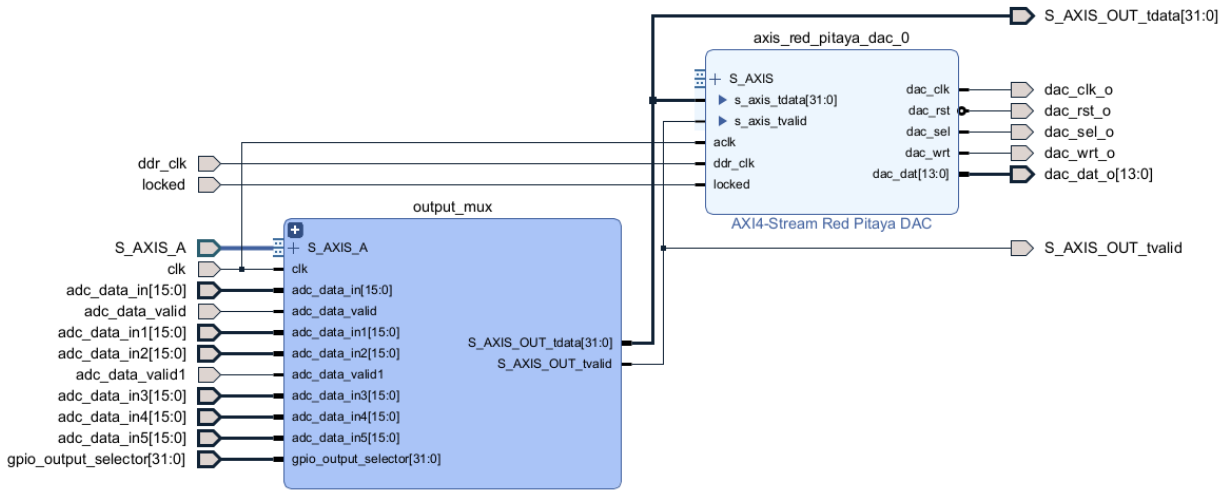
Transform Length	Transform Cycles	Latency(µs)
8	122	0.976
16	154	1.232
32	258	2.064
64	386	3.088
128	657	5.256
256	1169	9.352
512	2208	17.664
1024	4256	34.048
2048	8366	66.928
4096	16558	132.464

טבלה 2-תוצאות זמני חישוב כתלות באורך התמרה

*אין תמיכה באורכים קטנים מ-64.
*כדי לקבל את סך זמן חישוב הSTFT/FT יש להכפיל בגודל החלון.

לאחר שחישוב הסתיים נעבור לstate – DONE וניתן יהיה לקרוא למחשב את תוצאת החישוב.

Output mux and DAC



איור 14-סכמת בלוקים עבור MUX המוצא וה-DAC

המשתמש יכול לבחור בעזרת קונפיגורציה מה לשדר לDAC ולדגום במוצא:

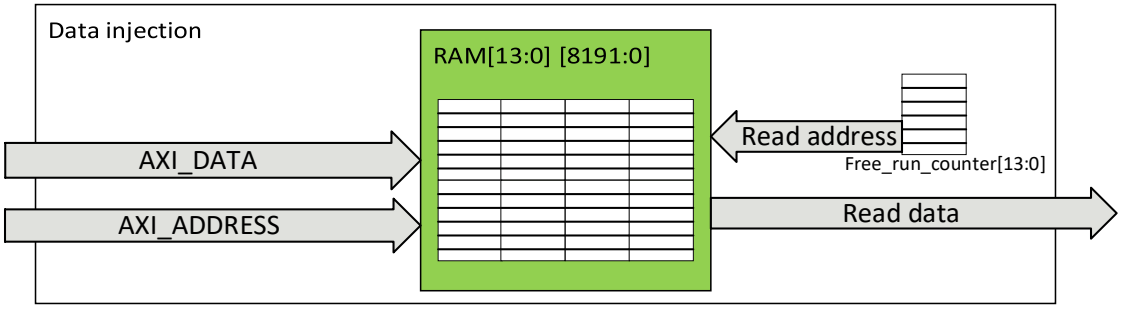
Output_mux – האופציות שניתן לבחור להוציא ב-output_mux ניתנות לשליטה לפי טבלה 11.

Value	DAC output
0x01	Reconstructed signal
0x02	Approximation level
0x04	Detail level 1
0x08	Reconstructed level 2
0x10	Approximation level 2
0x20	Detail level 2
0x40	FT

טבלה 3- שליטה על מוצא המערכת בהתאם לכתובת

Axi_red_pitaya_dac – דרייבר לDAC, משדר לשני הערוצים ומייצר שעון.

Data injection mode



איור 15- תיאור מנגנון הזרקת מידע

המשתמש יכול לעבוד בmode שבו הוא לא קורא את המידע מה-ADC אלא ב-memory פנימי שמשדר באופן חזרתי את המידע שטעון לו בזיכרון.

<u>Design for Test</u>

MUX לדיבוג המודול, המשתמש יכול לקרוא פנימיים מה-FPGA כדי לבדוק שהכול עובד כשורה או כדי לשפר וייעל תהליכים, לדוגמא המשתמש יכול לקרוא את הuniversal threshold שחושב עבור כל שלב, ובכל לשנות את הקבוע C_i בכדי לקבל תוצאות יותר טובות.

Select value	Bits	description
4'b0000	32	Const. value of 0xCAFE_CAFE
4'b0001	32	RTL version
4'b0010	14	The median of level 1
4'b0011	14	The median of level 2
4'b0100	14	The universal threshold of level 1
4'b0101	14	The universal threshold of level 2
4'b0110	14	Debug memory data
4'b0111	9	Start up counter
4'b1000	14	Debug memory write counter
4'b1001	4	Debug memory control signals
4'b1010	14	Debug player value
4'b1011	14	Debug data read/write address
4'b1100	32	FFT data
4'b1101	32	FFT fifo write/read counter
4'b1110	{16,16}	2 values of first data injection
4'b1111	Not used	Not used

טבלה 4 – שליטה על MUX לצורך Debuging.