Parallel-Pipeline Fast Walsh-Hadamard Transform Implementation Using HLS

A. Manjarres Garcia[1], C. Osorio Quero[1], J. Rangel-Magdaleno[1], J. Martinez-Carranza[2] and D. Durini Romero[1]
[1] *Digital Systems Group - Electronics Department*
[2] *Computer Science Department*
Instituto Nacional de Astrofísica, Óptica y Electrónica. (INAOE)
Luis Enrique Erro 1, 72840 Tonantzintla, Puebla, Mexico
email:{manjarres, caoq, jrangel, carranza, ddurini}@inaoep.mx

***Abstract*—Walsh Hadamard Transform (WHT) is an orthogonal, symmetric, involutional, and linear operation used in data encryption, data compression, and quantum computing. The WHT belongs to a generalized class of Fourier transforms, which allows that many algorithms developed for the fast Fourier transform (FFT) work for fast WHT implementations (FWHT). This paper employs this property and uses a parallel-pipeline FFT well-known strategy for VLSI implementation to build parallel-pipeline architectures for FWHT. We apply the FFT parallel-pipeline approach on a Fast WHT and use the High-Level Synthesis (HLS) tool from Xilinx Vitis to generate an FPGA solution. We also provide an open-source code with the basic blocks to build any model with any parallelization level. The parallel-pipeline proposed solutions achieve a latency reduction of up to 3.57% compared to a pipeline approach on a 256-long signal using 32 bit floating-point numbers.**

I. Introduction

The Walsh-Hadamard Transform (WHT) decomposes any signal into a series of basis functions called Walsh functions, which are rectangular or square with values of $+1$ and -1. The WHT of a signal with 2^n samples can be computed by the matrix multiplication with the $(2^n \times 2^n)$ Hadamard matrix. This Hadamard matrix can be defined recursively by

$$H_k = \frac{1}{\sqrt{2}} \begin{bmatrix} H_{k-1} & H_{k-1} \\ H_{k-1} & -H_{k-1} \end{bmatrix} \tag{1}$$

with $H_0 = [1]$, or using the Kronecker product by:

$$H_k = H_1 \otimes H_{k-1}. \tag{2}$$

Since the values of the Hadamard matrix are $+1$ and -1, the WHT computation requires only additions and subtractions. Also, the definition allows the use of FFT-based algorithms to reduce the computational complexity of the transform. These algorithms are based on the Cooley–Tukey algorithm and reduce the computational complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$. These new algorithms receive the name of Fast Walsh-Hadamard Transform (FWHT). Figure 1 illustrates a FWHT in a vector of 8 samples.

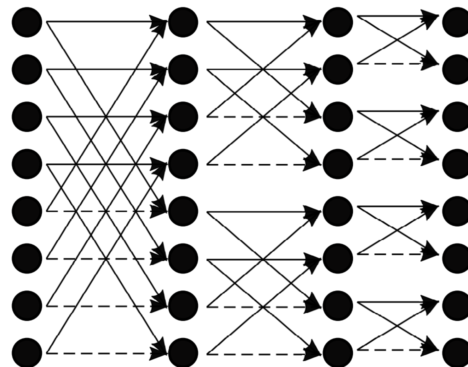


Fig. 1. Fast Walsh-Hadamard Transform on a vector of 8 samples. The black dots perform the sum between the two input arrows. The dashed lines invert the data sign, and the solid lines keep the data sign.

FWHT has attracted a lot of attention, thanks to its application in areas such as quantum information theory [1], optics [2], image processing [3] and error correction [4]. This attention brings with it the need for faster implementations. To address this, several works have been proposed that use specific hardware to execute the transform [5], [6], [7], [8]. Although these implementations provide good latencies, they are mostly pipelined architectures.

In this paper, we propose a flexible parallel-pipeline architecture to compute the FWHT using high-level synthesis (HLS). The code was developed in C++, allowing to build new layouts with different numbers of parallel inputs. The architectures were tested using the Vitis synthesis tools on a Virtex 9 FPGA and achieve a latency improvement of up to 3.57% compared to a pipeline version.

We start by a brief introduction of the parallel-pipeline algorithm in section II, then in the next section we explain how we used HLS to build the architectures. The section IV and V present the results of two experiments and the conclusions, respectively.

II. Fast Walsh-Hadamard Transform

The main objective of the Fast Walsh-Hadamard Transform (FWHT) is to reduce the number of operations from N^2 to $N \log_2 N$, based on generalizations of the Cooley-Tukey FFT algorithm. One of these FFT solutions was presented by Wold and Despain in [9], which uses parallel-pipeline architectures to achieve high throughputs and low latencies. In this section, we will summarize the Wold and Despain parallel-pipeline approach.

A. Parallel-Pipeline Architecture

First, we present the operators that constitute the parallel-pipeline architecture. To define these operators, it is necessary to introduce the index notation. In this notation each element on the data stream is represented by a vector $[x, y]$ which indicates the position of that element into the layout. These indices are written in binary format as $x = [x_u \cdots x_1]$ and $y = [y_u \cdots y_1]$, for example the four element of a row data stream have a index of $[0011, 0000]$.

With this notation we define the operators as:

$$\begin{aligned}
\delta_{(l)}[x,y] &= [[x_u \cdots x_{l+1}], [y_u \cdots y_1 x_l \cdots x_1]] \\
\mu_{(j,l)}[x,y] &= [[x_u \cdots x_{l+1} x_{l-j} \cdots x_1], [y_u \cdots y_1 x_l \cdots x_{m-j+1}]] \\
\mu_{(l)}[x,y] &= \mu_{(j,l)}[x,y] \quad \forall j = 1 \\
\mu^{-1}_{(j,l)}[x,y] &= [[x_u \cdots x_{l-j+1} y_j \cdots y_1 x_{l-j} \cdots x_1], [y_u \cdots y_{j+1}]] \\
\mu^{-1}_{(l)}[x,y] &= \mu^{-1}_{(j,l)}[x,y] \quad \forall j = 1 \\
R_{(l)}[x,y] &= [x, [y_u \cdots y_2 (y_1 \oplus y_{l+1})]] \\
\phi_{(l,j)}[x,y] &= [x, [y_u \cdots y_{l+j+1} (y_{l+j} \oplus x_l) y_{l+j-1} \cdots y_1]] \\
\phi_{(l)}[x,y] &= \phi_{(l,j)}[x,y] \quad \forall j = 0 \\
T_{(l,j)}[x,y] &= [[x_u \cdots x_{j+1} y_l \cdots y_1], [y_u \cdots y_{l+1} x_j \cdots x_1]]
\end{aligned}$$

where, $\oplus$ is the logical operator XOR.

Now, we define the three types of networks that can be created, according to the parallelization level. This parallelization level is indicated with the letter k, so the network implements 2^k parallel rows.

1) Case 1 ($n - k = k$):

$$\begin{aligned}
&\delta_{(k)}\mu_{(k)} B R_{(k)} \mu^{-1}_{(k)} \cdots \mu_{(1)} B R_{(1)} \mu^{-1}_{(1)} \phi_{(k)} \cdots \phi_{(1)} \\
&\qquad \mu_{(k)} R_{(k)} B \mu^{-1}_{(k)} \cdots \mu_{(1)} R_{(1)} B \mu^{-1}_{(1)} \mu^{-1}_{(k,n)}
\end{aligned} \tag{3}$$

2) Case 2 ($n - k > k$):

$$\begin{aligned}
&\delta_{(k)}\mu_{(n-k)} B \mu^{-1}_{(n-k)} \cdots \mu_{(k+1)} B \mu^{-1}_{(k+1)} \\
&\quad \mu_{(k)} B R_{(k)} \mu^{-1}_{(k)} \cdots \mu_{(1)} B R_{(1)} \mu^{-1}_{(1)} \phi_{(k)} \cdots \phi_{(1)} \\
&\qquad \mu_{(k)} R_{(k)} B \mu^{-1}_{(k)} \cdots \mu_{(1)} R_{(1)} B \mu^{-1}_{(1)} \mu^{-1}_{(k,2k)}
\end{aligned} \tag{4}$$

3) Case 3 ($n - k < k$):

$$\begin{aligned}
&\delta_{(k)}\mu_{(n-k)} B R_{(k)} \mu^{-1}_{(n-k)} \cdots \mu_{(1)} B R_{(2k-n+1)} \mu^{-1}_{(1)} \phi_{(n-k,2k-n)} \\
&\cdots \phi_{(1,2k-n)} \mu_{(n-k)} R_{(k)} B \mu^{-1}_{(n-k)} \cdots \mu_{(1)} R_{(2k-n+1)} B \mu^{-1}_{(1)} \\
&T_{(2k-n,n-k)} \mu_{(2k-n)} B \mu^{-1}_{(2k-n)} \cdots \mu_{(1)} B \mu^{-1}_{(1)} \mu^{-1}_{(n-k,k)} \mu^{-1}_{(n-k,n)}
\end{aligned} \tag{5}$$

The $T_{(2k-n,n-k)}$ operator in the expression requires the implementation of a buffer with 2^n elements, breaking the pipeline characteristic of the architecture. To replace this operator we use the following equations, considering that $j = 2k - n$ and $l = (n - k) - j$

$$T_{(j,l-j)} = T_{(j,j)} \mu^{-1}_{(j,2j)} \mu_{(l-j,n)} \tag{6}$$

$$\begin{aligned}
T_{(j,j)} = &\mu_{(j)} R_{(j)} \mu^{-1}_{(j)} \cdots \mu_{(1)} R_{(1)} \mu^{-1}_{(1)} \phi_{(j)} \cdots \phi_{(1)} \\
&\qquad \mu_{(j)} R_{(j)} \mu^{-1}_{(j)} \cdots \mu_{(1)} R_{(1)} \mu^{-1}_{(1)}
\end{aligned} \tag{7}$$

In the above expressions B represents the *butterfly* module, a module that performs the additions and subtractions of the FWHT. Also the $\cdots$ represent that the sub index of the operator decrease. For example: $\mu_{(4)} B R_{(6)} \mu^{-1}_{(4)} \cdots \mu_{(1)} B R_{(3)} \mu^{-1}_{(1)}$, it is equal to:

$$\mu_{(4)} B R_{(6)} \mu^{-1}_{(4)} \mu_{(3)} B R_{(5)} \mu^{-1}_{(3)} \mu_{(2)} B R_{(4)} \mu^{-1}_{(2)} \mu_{(1)} B R_{(3)} \mu^{-1}_{(1)}$$

III. Hardware Implementation

Regardless of the parallelization level, any architecture is built using *Processing Elements* (PEs) that perform the additions and subtractions and *Connection Elements* (CEs) that rearrange the data. In this section, we present the PEs and CEs, along with a latency analysis of the architectures. All modules are written in Vivado HLS in a way that is easy to build any parallel-pipeline architecture. The code with some examples is available on GitHub[1].

A. Processing Elements

The PEs in the architecture are four and represent the expressions: $\mu B \mu^{-1}$, $\mu R B \mu^{-1}$, $\mu B R \mu^{-1}$ and $\mu R B R \mu^{-1}$. Although the latter is not in the former expressions, it appears when the operator T is expanded in (5).

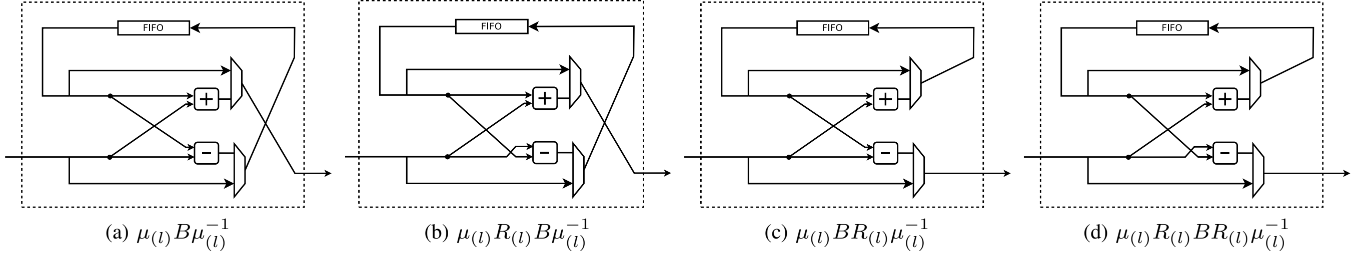
We implement the $\mu_{(l)} B \mu^{-1}_{(l)}$ element using an adder, a subtractor, two multiplexers, and a memory element (FIFO), as illustrated in Figure 2a. We divided the operation of this block into three stages: (*i*) we write the input data to the FIFO until it is full. (*ii*) We read the FIFO values and compute them with new incoming data, the subtraction results go to FIFO and the sum to the output. (*iii*) Once the FIFO is filled with the subtraction results, the output takes values from the FIFO read port, and the new data are written to the FIFO as in the first stage. The final two stages are repeated until there is no more input data.

The rest of the PEs (Figure 2b, Figure 2c and, Figure 2d) are the equivalent, but with inputs, outputs, or inputs and outputs reversed. These implementations follow a classical way to implement the butterfly modules, but in FPGA when we have limited resources, the use of an adder and subtractor that operate in the same data seems inefficient. Then we propose a second PEs version (Figure 3) that uses only one adder and implement the subtractor by inverting the sign of one input. The sign inversion represented in the figure by a multiplier is implemented using the XOR gate. This new version reduces the number of adders but adds another memory element of the same size as the existing one.

B. Connection Elements

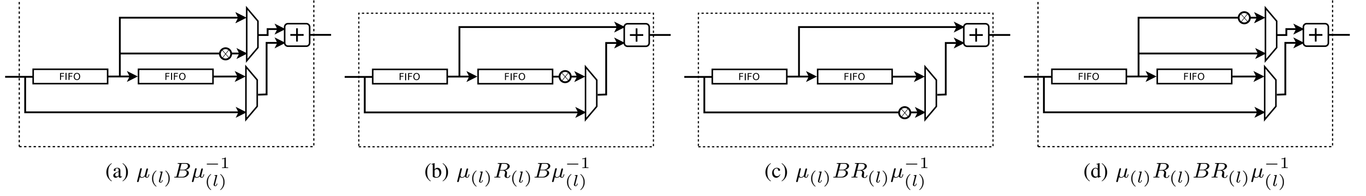
The connections elements define how the data is rearranged. The first CE block is the $\delta_{(l)}$, which converts the serial data into 2^l rows at the beginning of the architecture. The opposite of $\delta_{(l)}$ are the $\mu^{-1}_{(l,j)}$ blocks, at the end of the architecture. The last CE is the $\phi_{(l,j)}$ block which is a permutation within each column. We use multiplexers and demultiplexers to implement these blocks.

[1] https://github.com/andres091096/fwht_hls



(a) $\mu_{(l)}B\mu^{-1}_{(l)}$ (b) $\mu_{(l)}R_{(l)}B\mu^{-1}_{(l)}$ (c) $\mu_{(l)}BR_{(l)}\mu^{-1}_{(l)}$ (d) $\mu_{(l)}R_{(l)}BR_{(l)}\mu^{-1}_{(l)}$

Fig. 2. First version of the processing elements implemented on the architectures.



(a) $\mu_{(l)}B\mu^{-1}_{(l)}$ (b) $\mu_{(l)}R_{(l)}B\mu^{-1}_{(l)}$ (c) $\mu_{(l)}BR_{(l)}\mu^{-1}_{(l)}$ (d) $\mu_{(l)}R_{(l)}BR_{(l)}\mu^{-1}_{(l)}$

Fig. 3. Second version of the processing elements implemented on the architectures.

C. Latency analysis

The architecture uses the HLS pragma dataflow to achieve task-level parallelism. Thanks to this, the latency of the architecture is almost the latency of the largest block within the pipeline. The latency of the blocks is given by the equation $L = D + II * N$, where D is known as the depth of the module, i.e. the number of cycles to process one sample (typically no more than ten cycles in this architecture), N is the number of samples to be processed (2^n), and II is the initialization interval ($II = 1$ in all the blocks except $\phi_{(l,j)}$).

The CEs, $\delta_{(l)}$ and $\mu^{-1}_{(l,j)}$, have a latency of almost 2^n, and $\phi_{(l,j)}$ a latency of 2^{n-k}. The two versions of the PEs have a latency of $\frac{3}{2}2^{n-k}$. Then, when $k = 0$ the PE blocks have a latency greater than 2^n making them the largest blocks in the pipeline. But, when $k > 0$ the PEs have a latency less than 2^n making the $\delta_{(l)}$ and the $\mu^{-1}_{(l,j)}$ the largest blocks.

IV. Evaluation

A. Environment Setup

We evaluate the implementations using Vitis v2019.2 over the XCVU9P-FLGB2104-2-I board. All the results presented in this paper were extracted from Vitis reports after the synthesis stage targeting a clock frequency of 100 MHz, and using 32 bit floating-point arithmetic.

B. Experiments

1) Parallelization levels experiment: We test six architectures with parallelization levels (k) from 0 to 5 on a signal with 256 samples ($n = 8$). $k = 0$ is a pipeline architecture built using only a row of PEs. The $k = 1$, 2, and 3 models belong to Case 2 (Expression 4). Replacing the k and n values into the Case 2 expression, we obtain:

for $k = 1$

$$\begin{gathered}\delta_{(1)}\mu_{(7)}B\mu^{-1}_{(7)}\cdots\mu_{(2)}B\mu^{-1}_{(2)}\mu_{(1)}BR_{(1)}\mu^{-1}_{(1)} \\ \phi_{(1)}\mu_{(1)}R_{(1)}B\mu^{-1}_{(1)}\mu^{-1}_{(1,2)}\end{gathered} \tag{8}$$

for $k = 2$

$$\begin{aligned}&\delta_{(2)}\mu_{(6)}B\mu^{-1}_{(6)}\cdots\mu_{(3)}B\mu^{-1}_{(3)}\mu_{(2)}BR_{(2)}\mu^{-1}_{(2)} \\ &\quad\mu_{(1)}BR_{(1)}\mu^{-1}_{(1)}\phi_{(2)}\phi_{(1)}\mu_{(2)}R_{(2)}B\mu^{-1}_{(2)} \\ &\qquad\mu_{(1)}R_{(1)}B\mu^{-1}_{(1)}\mu^{-1}_{(2,4)}\end{aligned} \tag{9}$$

and for $k = 3$

$$\begin{aligned}&\delta_{(3)}\mu_{(5)}B\mu^{-1}_{(5)}\mu_{(4)}B\mu^{-1}_{(4)}\mu_{(3)}BR_{(3)}\mu^{-1}_{(3)}\cdots \\ &\quad\mu_{(1)}BR_{(1)}\mu^{-1}_{(1)}\phi_{(3)}\phi_{(2)}\phi_{(1)}\mu_{(3)}R_{(3)}B\mu^{-1}_{(3)}\cdots \\ &\qquad\mu_{(1)}R_{(1)}B\mu^{-1}_{(1)}\mu^{-1}_{(3,6)}\end{aligned} \tag{10}$$

The $k = 4$ architecture corresponds to Case 1 (Expression 3), so the expression for this level is:

$$\begin{gathered}\delta_{(4)}\mu_{(4)}BR_{(4)}\mu^{-1}_{(4)}\cdots\mu_{(1)}BR_{(1)}\mu^{-1}_{(1)}\phi_{(4)}\cdots\phi_{(1)} \\ \mu_{(4)}R_{(4)}B\mu^{-1}_{(4)}\cdots\mu_{(1)}R_{(1)}B\mu^{-1}_{(1)}\mu^{-1}_{(4,8)}\end{gathered} \tag{11}$$

Finally, $k = 5$ correspond to Case 3 (Expression 5), where the architecture expression is:

$$\begin{aligned}&\delta_{(5)}\mu_{(3)}BR_{(5)}\mu^{-1}_{(3)}\cdots\mu_{(1)}BR_{(3)}\mu^{-1}_{(1)}\phi_{(3,2)}\phi_{(2,2)}\phi_{(1,2)} \\ &\quad\mu_{(3)}R_{(5)}B\mu^{-1}_{(3)}\cdots\mu_{(1)}R_{(3)}B\mu^{-1}_{(1)}T_{(2,3)}\mu_{(2)}B\mu^{-1}_{(2)} \\ &\qquad\mu_{(1)}B\mu^{-1}_{(1)}\mu^{-1}_{(3,5)}\mu^{-1}_{(3,8)}\end{aligned} \tag{12}$$

To break the $T_{(2,3)}$ operator, we applied the equations 6 and 7, to obtain the final expression:

TABLE I

LATENCY AND RESOURCE CONSUMPTION RESULTS FOR SIX LEVELS OF PARALLELIZATION OF AN FWHT IMPLEMENTATION USING THE PES VERSION 1.

Samples (2^n)	k Level	Area				Latency (Cycles)
		LUTs	FFs	DSPs	BRAMs	
256	0	9505	8550	32	24	400
256	1	17840	15874	64	26	330
256	2	34877	31843	128	52	335
256	3	67326	61784	256	107	338
256	4	133233	122738	512	194	348
256	5	258166	239600	1024	324	366

TABLE II

LATENCY AND RESOURCE CONSUMPTION RESULTS FOR SIX LEVELS OF PARALLELIZATION OF AN FWHT IMPLEMENTATION USING THE PES VERSION 2.

Samples (2^n)	k Level	Area				Latency (Cycles)
		LUTs	FFs	DSPs	BRAMs	
256	0	7558	5664	16	27	400
256	1	14142	10664	32	30	330
256	2	26826	20733	64	54	334
256	3	52676	41938	128	98	338
256	4	103473	85010	256	194	348
256	5	202230	171152	512	324	366

$$\delta_{(5)}\mu_{(3)}BR_{(5)}\mu_{(3)}^{-1}\cdots\mu_{(1)}BR_{(3)}\mu_{(1)}^{-1}\phi_{(3,2)}\phi_{(2,2)}\phi_{(1,2)}$$
$$\mu_{(3)}R_{(5)}B\mu_{(3)}^{-1}\mu_{(1)}R_{(4)}BR_{(2)}\mu_{(1)}^{-1}\mu_{(1)}R_{(3)}BR_{(1)}\mu_{(1)}^{-1}$$
$$\phi_{(2)}\phi_{(1)}\mu_{(2)}B\mu_{(2)}^{-1}\mu_{(1)}B\mu_{(1)}^{-1}\mu_{(2,4)}^{-1}\mu_{(3,8)}^{-1} \quad (13)$$

We implement these architectures using the two versions of PEs so, Table I presents the latency measured in clock cycles and the resource consumption for the six levels of parallelization using the first version of the PEs and, Table II shows the results for the second version.

The results show that in both versions of the PEs, the latency of the pipeline architecture $k = 0$ is the highest and, the $k = 1$ is the lowest. The results also show that the latency increases with parallelization. This increase is justified by the previous analysis of the latency. If we have $k > 0$, the latency will approach the ideal value $2^n + D$. But, with more parallelization, more $\phi_{(l,j)}$ stages have the architectures, increasing the depth of the pipeline.

In addition, these results show that the second version of the PE consumes fewer resources than the first version. Since the second version uses only one adder, the DSP usage is half that of the first version. Also, the memory of the second version is twice that of the first version, but in the BRAM results, both versions consume almost the same, this is because part of the new memory is implemented using the FFs and LUTs, which have a significant reduction in the second version.

2) Data length experiment: The second experiment evaluates architectures with $k = 1$, changing the number of samples. All the architectures use the second version of the PEs.

Table III presents the resource consumption and latency results of the architectures. The latency in the last five designs is almost the ideal 2^n latency, but the three previous designs are far from this optimal point because the clock frequency. The BRAMs show a significant increase in the last two input sizes because the input size also shows a notable increase, and part of the memory is no longer implemented as LUTs.

TABLE III

CONSUMED RESOURCES AND LATENCY FOR EIGHTH ARCHITECTURES, WITH DIFFERENT NUMBER OF SAMPLES AND THE SAME PARALLELIZATION LEVEL ($k = 1$). THE ARCHITECTURES USE THE PES VERSION 2.

Samples (2^n)	k Level	Area				Latency (Cycles)
		LUTs	FFs	DSPs	BRAMs	
16	1	7625	6194	16	10	62
64	1	10894	8586	24	18	124
256	1	14142	10664	32	30	330
1024	1	17536	12940	40	42	1112
4096	1	20981	15340	48	58	4198
16384	1	24532	17762	56	106	16500
65536	1	28392	20254	64	282	65666
262144	1	32240	22818	72	978	262288

V. CONCLUSIONS

The parallel-pipelined architectures presented in this work show an improvement compared to pipelined solutions. To improve performance and use the resources better, we suggest working with architectures with a parallelization level $k = 1$. The second version of PEs presented in this paper consumed fewer resources than the classical version, especially in LUTs, Flip-Flops, and DSP blocks.

The HLS blocks to build the architecture are easily configurable and allow building new architectures with different parallelization levels for any number of samples.

REFERENCES

[1] R. F. Werner, “All teleportation and dense coding schemes,” *Journal of Physics A: Mathematical and General*, vol.34, no.35, pp.7081-7094, 2001.

[2] M. Harwit and N. Sloane, *Hadamard transform optics*, Academic Press, 1979.

[3] A. T. Ho, Jun Shen, A. K. Chow and J. Woon, “Robust digital image-in-image watermarking algorithm using the fast Hadamard transform,” *Proceedings of the 2003 International Symposium on Circuits and Systems*, ISCAS ’03., 2003, pp. III-III, doi: 10.1109/ISCAS.2003.1205147.

[4] Horadam, K. J. *Hadamard Matrices and Their Applications*, Princeton University Press, 2007.

[5] A. Amira and S. Chandrasekaran, “Power Modeling and Efficient FPGA Implementation of FHT for Signal Processing,” in *IEEE Trans. on Very Large Scale Integration (VLSI) Systems*, vol. 15, no. 3, pp. 286-295, March 2007, doi: 10.1109/TVLSI.2007.893606.

[6] G. Bi, A. Aung and B. P. Ng, “Pipelined Hardware Structure for Sequency-Ordered Complex Hadamard Transform,” in *IEEE Signal Processing Letters*, vol. 15, pp. 401-404, 2008, doi: 10.1109/LSP.2008.922515.

[7] P. K. Meher and J. C. Patra, “Fully-pipelined efficient architectures for FPGA realization of discrete Hadamard transform,” presented at the *Int. Conf. Application-Specific Systems, Architectures and Processors*, Leuven, Belgium, Jul. 2-4, 2008.

[8] J. Liu, Q. Xing, X. Yin, X. Mao and F. Yu, “Pipelined Architecture for a Radix-2 Fast Walsh–Hadamard–Fourier Transform Algorithm,” in *IEEE Trans. on Circuits and Systems II: Express Briefs*, vol. 62, no. 11, pp. 1083-1087, Nov. 2015, doi: 10.1109/TCSII.2015.2456371.

[9] E. Wold and A. Despain, “Pipeline and Parallel-Pipeline FFT Processors for VLSI Implementations,” in *IEEE Trans. on Computers*, vol. C-33, no. 5, pp. 414-426, May 1984, doi: 10.1109/TC.1984.1676458.