A Flexible Floating-Point Wavelet Transform and Wavelet Packet Processor

Andre Guntoro and Manfred Glesner

Department of Electrical Engineering and Information Technology
Institute of Microelectronic Systems
Technische Universität Darmstadt
Email: {guntoro,glesner}@mes.tu-darmstadt.de

***Abstract*—The richness of wavelet transformation is known in many fields. There exist different classes of wavelet filters that can be used depending on the application. In this paper, we propose an IEEE 754 floating-point lifting-based wavelet processor that can perform various forward and inverse Discrete Wavelet Transforms (DWTs) and Discrete Wavelet Packets (DWPs). Our architecture is based on processing elements that can perform either prediction or update on a continuous data stream in every two clock cycles. We also consider the normalization step that takes place at the end of the forward DWT/DWP or at the beginning of the inverse DWT/DWP. To cope with different wavelet filters, we feature a multi-context configuration to select among various DWTs/DWPs. Different memory sizes and multi-level transformations are supported. For the 32-bit implementation, the estimated area of the proposed processor with 2×512 words memory and 8 PEs in a 0.18-μm process is 3.7 mm square and the estimated operating speed is 353 MHz.**

I. Introduction

For the last two decades the wavelet theory has been studied extensively [1], [2] to answer the demand for better and more appropriate functions to represent signals than the ones offered by the Fourier analysis. Along with recent trends and research focuses in applying wavelets in image processing, the application of wavelets is essentially not only limited to this area. Wavelets have been known in many fields such as mathematics, physics, and electrical engineering. In the field of electrical engineering wavelets have been known with the name multi-rate signal processing. Due to numerous interchanging fields, wavelets have been used in many applications such as image compression, feature detection, seismic geology, human vision, etc [3], [4].

Contrary to the Fourier transform, which uses one basis function to transform between domains, there are different classes of wavelets which can be applied on the signal depending on the application. In this paper, we propose a novel architecture to compute forward and inverse transforms of numerous DWTs (Discrete Wavelet Transforms) and also DWPs (Discrete Wavelet Packets) based on their lifting scheme representations. The proposed architecture takes into account that the wavelet coefficients of an arbitrary wavelet filter and the corresponding wavelet transforms cannot be satisfied by using integer computation. The proposed architecture provides a multi-context configuration to choose between various forward and inverse DWTs/DWPs. The memory size, the context size, and the maximum lifting size can be freely chosen to cope with different application demands.

The rest of the paper is organized as follows. Section II describes the second generation of wavelets, DWT, and DWP. The proposed architecture is explained in Section III. Section IV discusses the performance of the proposed architecture and Section V summarizes our conclusions.

II. Backgrounds

A. Lifting Scheme

The second generation of wavelets, more popular under the name lifting scheme, was introduced by Sweldens [5]. The basic principle of the lifting scheme is to factorize the wavelet filter into alternating upper and lower triangular 2×2 matrix. Let $H(z)$ and $G(z)$ be a pair of low-pass and high-pass wavelet filters:

$$H(z) = \sum_{n=k_l}^{k_h} h_n z^{-n} \qquad G(z) = \sum_{n=k_l}^{k_h} g_n z^{-n}$$

where h_n and g_n are the corresponding filter coefficients. $N = |k_h - k_l| + 1$ is the filter length and the corresponding Laurent polynomial degree is given by $h = N-1$. By splitting the filter coefficients into even and odd parts, the filters can be rewritten as:

$$H(z) = H_e(z^2) + z^{-1} H_o(z^2) \qquad G(z) = G_e(z^2) + z^{-1} G_o(z^2)$$

and the corresponding polyphase representation is:

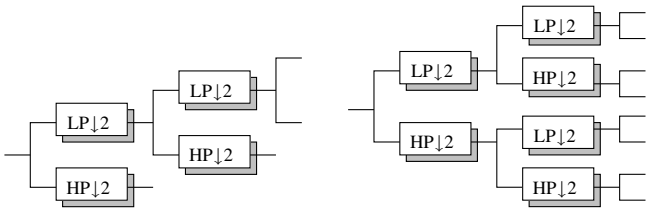
$$P(z) = \begin{bmatrix} H_e(z) & G_e(z) \\ H_o(z) & G_o(z) \end{bmatrix}$$

Daubechies and Sweldens in [5], [6] have shown that the polyphase representation can always be factored into lifting steps by using the Euclidean algorithm to find the greatest common divisors. Thus the polyphase representation becomes:

$$P(z) = \begin{bmatrix} K & 0 \\ 0 & 1/K \end{bmatrix} \prod_{i=n}^{1} \begin{bmatrix} 1 & a_i(z) \\ 0 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ b_i(z) & 1 \end{bmatrix}$$

where K is the normalization factor and $a_i(z)$ and $b_i(z)$ are the Laurent polynomials which correspond to the updaters and the predictors of the lifting steps.

978-3-9810801-5-5/DATE09 © 2009 EDAA



(a) DWT (b) DWP

Fig. 1. Two different transformations.

B. Wavelet Transform and Wavelet Packet

Wavelet transform is a multi-resolution signal analysis. In the traditional wavelet transforms, only the low-pass signal is used on the next transformation level. In wavelet packets, both low-pass and high-pass signals are analyzed, resulting equally spaced frequency bands. Fig. 1 depicts both schemes. LP and HP correspond to low-pass and high-pass filter pair and $\downarrow 2$ corresponds to down-sampling by two. The major issue in DWP is that the resulting HP signals are much smaller than the LP parts in normal circumstances. Thus performing multi-level DWP using integer arithmetics would make these HP signals go to zero, which lead to lower achievable SNR values.

III. Proposed Architecture

As the lifting scheme breaks a wavelet filter into smaller predictions and updates, the resulting predictor and updater can be limited to have two terms on its Laurent polynomial. Without loss of generality, we can formulate the predictor or the updater polynomial as:

$$l(z) = c_1 z^{-p} + c_2 z^{-q}$$

with polynomial constants c_0 and c_1, and $|p - q| = N$. This implies that on each stage (either as a predictor or an updater), two multiplications and two additions are performed.

Taking into account that a predictor and an updater perform a similar computation, we propose a wavelet processor which is based on M processing elements (PEs) to cope with M lifting steps. Due to the nature of a lifting scheme, lifting representations of higher order wavelet filters, which contribute to longer lifting steps, can be broken easily into several lifting steps with a maximum size of M.

A. Architecture of the Processing Element

The PE performs the prediction or the update in every 2 clock cycles. Taking into account that floating-point (FP) multipliers are expensive in term of logic counts, and the PE receives two samples (s and d) at once, we have decided to lower the input rate by half. Thus, the PE requires only one multiplier. From the performance point of view, the processing rate of the PE will be equal to the processor speed and no longer twice as fast. This also implies that the bottleneck issues with the memory will not occur.

Fig. 2 depicts the proposed PE. The PE has two selectors S1 and S2 to choose the prediction or the update samples that correspond to the factors p and q from the Laurent polynomial.

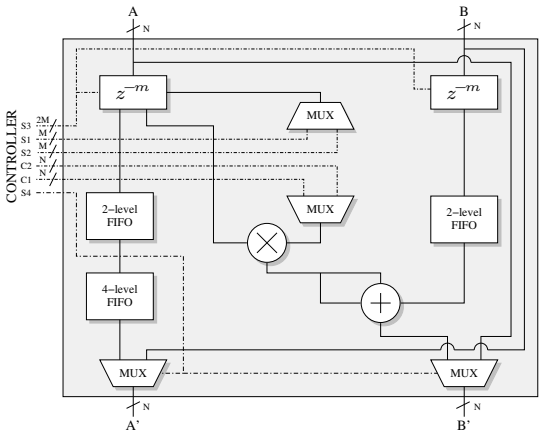


Fig. 2. Block diagram of the PE.

Two constants C1 and C2 that represent the filter coefficients are defined and configured by the controller. By delaying the actual samples, selector S3 controls the prediction or the update that requires future samples. Selector S4 is a bypass selector. The maximum depth of the unit delay z^{-m}, which determines the maximum delay level, can be freely chosen during the design. Two unit delays are implemented on both input ports A and B. In order to reduce the number of registers needed for the unit delay, the unit delay on port A has two input selectors (i.e. S3 and the multiplexer output) and two outputs. The second part performs the FP multiplications on the samples selected by S1 and S2 with the constants C1 and C2. Two 2-level FIFOs on both input samples are implemented to compensate the multiplier delay. The last part performs the addition of three FP values. One 4-level FIFO is implemented to compensate the delay introduced by the adder.

B. Normalization

As normalization can take place at the end of the transformation in case of forward DWT/DWP or at the beginning of the transformation in case of inverse DWT/DWP, two special PEs to handle this function are required. We extend the functionality of the PEs that are located at the top and at the bottom of the proposed wavelet processor. Three additional multiplexers are needed to add the normalization factor unit into the PE. Fig. 3 shows the PE used at the top and at the bottom of the proposed architecture. By enabling S5 and setting S1 and S3 to zero, two inputs of the multiplexer before the multiplier correspond to the actual samples s and d (with the normalization factors K=C1 and $1/K$=C2). The first multiplication product passes through the multiplexer and the 1-level FIFO resulting in $s' = Ks$ (left side). The second multiplication product passes through the multiplexer resulting $d' = d/K$ (right side). The 4-level FIFO is split into 3-level and 1-level FIFOs, with the latter used to make both outputs synchronized.

C. Floating-Point Multiplier

As mentioned earlier, the PE utilizes only one FP multiplier which is time-shared in order to perform two multiplications.

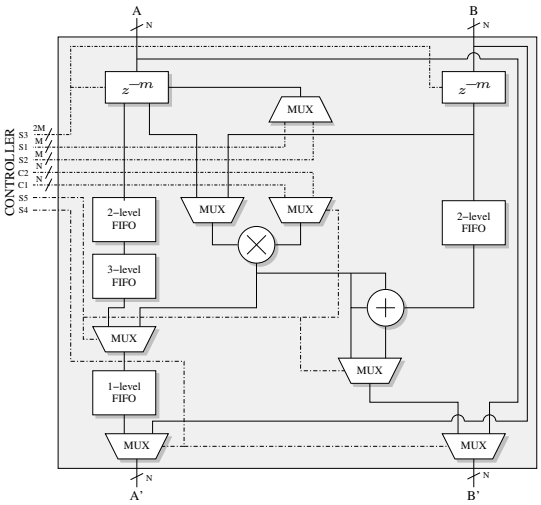
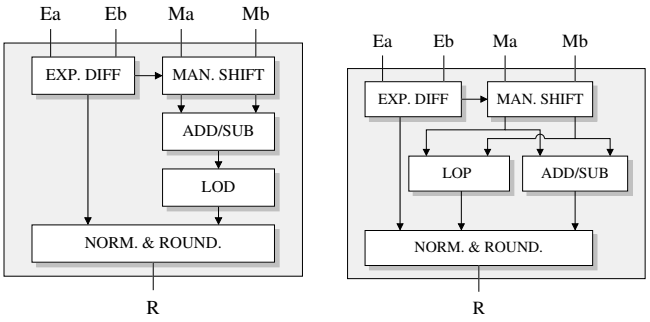


Fig. 3. Block diagram of the PE which is located at the top and at the bottom.



(a) Leading-One Detector. (b) Leading-One Predictor.

Fig. 4. Two-Input Floating-Point Adder.

The first clock cycle performs the first multiplication (i.e. C1 $\times$ M1) and the second cycle performs the second multiplication (i.e. C2 $\times$ M2). M1 and M2 are the time-multiplexed output samples of the unit delay determined by the output of the multiplexer. Our FP multiplier is a 2-level pipeline architecture and can be customized for other FP formats besides the standard IEEE 754 single and double precision formats. Due to the page limitation, the architecture of the FP multiplier is not detailed here.

D. Three-Input Floating-Point Adder

Contrary to the FP multiplier, the FP adder requires more steps due to the algorithm complexity and the data dependency. As depicted in Fig. 4(a), to perform addition between two FP numbers, the following steps are performed:

1) Calculate the exponent difference.
2) Align the mantissa by shifting the mantissa with the lower exponent to the right.
3) Add/subtract both mantissas depending on the sign bits.
4) Perform the Leading-One Detection (LOD) to determine the location of the first logic one.
5) Normalize and round the result.

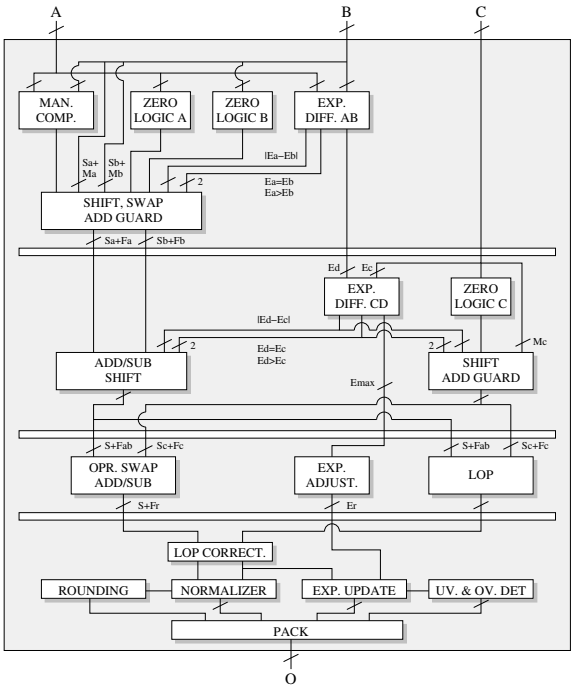


Fig. 5. 3-Input Floating-Point Adder.

In order to decrease critical paths, Leading-One Prediction (LOP) was proposed in [7], [8] as a replacement of LOD, predicting the first occurrence of the logic one directly from the operands. Fig. 4(b) depicts the addition algorithm with LOP. The LOP works in parallel with the adder and it is based on the encoding tree which examines both inputs from left to right. The LOP ignores the possible carry or borrow that might occur on the addition/subtraction result. Thus, it leads to one-bit inaccuracy, which will be corrected during the normalization step. This is why it is more popular with the name *inexact LOP*.

In order to add three FP numbers, two inputs will be added first and the temporary result will then be added to the third input. This introduces a longer pipeline structure with the normal approach (i.e. chaining two FP adders) and consumes more area. To minimize the number of pipeline stages, we have developed a dedicated 4-stage 3-input floating-point adder. Fig. 5 depicts the block diagram of the adder.

1) Stage 1: At the first stage, the two inputs A and B are unpacked. The *mantissa comparator* compares both mantissas M_a and M_b and outputs one decision bit (i.e. $M_a \geq M_b$), which will be used by the *shift, swap, and add guard*. The *zero logic* detects if the corresponding input is zero. The *exponent difference AB* compares both exponents E_a and E_b. This block outputs 4 values. The first 3 values (the shift count $|E_a - E_b|$ and the comparator results $E_a = E_b$ and $E_a > E_b$) will be used by the *shift, swap, and add guard*. The 4th value (the temporary dominant exponent $E_d = max(E_a, E_b)$) will be used by the *exponent difference CD*.

The *shift, swap, and add guard* aligns the mantissa M_a and M_b to have the same exponent degree by shifting the mantissa

with the smaller exponent to the right. The hidden bit and the guard bits are appended to the most significant bit and the least significant bit of both mantissas respectively. The number of bits used as guard bits can be freely chosen. Based on the exponent difference, three different cases are examined here:

- $E_a = E_b$: Depending on the output of the *mantissa comparator*, M_a and M_b will be swapped directly without performing any shifting.
- $E_a > E_b$: M_b will be shifted to the right with the amount determined by the *exponent difference AB* (i.e. $|E_a - E_b|$).
- $E_a < E_b$: M_a will be shifted to the right with the amount determined by the *exponent difference AB*. Both mantissas will be swapped afterwards.

In all of the cases, if a zero number is detected, the corresponding mantissa(s) will be set to zero. The outputs of the *shift, swap, and add guard* are the sorted and extracted fractions F_a and F_b with their corresponding signs.

At this stage, the input C is not processed, thus two values that correspond to the prediction/update sample and the first multiplication result can be added first. The second multiplication result that comes on the next clock cycle will be unpacked at the second stage.

2) Stage 2: The fractions F_a and F_b are added/subtracted depending on the sign difference (i.e. $S_a \oplus S_b$), resulting in the fraction F_{ab}. If $E_c > E_d$, the result will be shifted to the right. These steps are performed by the *add/sub and shift* with the shift parameter determined by the *exponent difference CD*. 3 different cases as at the first stage are also examined here.

The *shift and add guard* prepares the mantissa M_c. If $E_c < E_d$, M_c will be shifted instead. The hidden bit and the guard bits are appended to M_c, resulting in fraction F_c. Finally, if the *zero logic* detects a zero number, F_c will be set to zero.

3) Stage 3: The *operand swap and add/sub* swaps F_{ab} and F_c if necessary. The *LOP* works parallel with the *operand swap and add/sub* to predict the first occurrence of the logic one directly from the operands. Taking into account that one-bit inaccuracy might occur on the prediction, the *LOP* prepares two values at the output to minimize the critical paths on the normalization stage.

Because three addition/subtraction arithmetic operations are involved, the final result might have an increase of exponent by two. The *exponent adjustment* prepares the dominant exponent by simply adding two to the largest exponent (i.e. $E_r = max(E_a, E_b, E_c) + 2$).

4) Stage 4: Because the *LOP* may deliver an error within one-bit degree, the error has to be corrected. The error can easily be detected by looking at the LOP-index bit of the resulting fraction F_r. This step is performed by the *LOP correction*. Additionally, this block also performs the pre-normalization by shifting the resulting fraction F_r to the left with the shift amount determined by the inexact LOP value. We have examined that correcting the LOP result in real time (by adding with one) will increase the critical path on the stage 4. This is why the *LOP* on the stage 3 outputs two values. Therefore, we only need to choose the right value at the end.

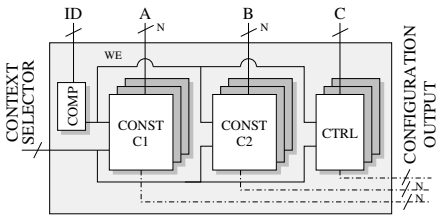


Fig. 6. Configuration Controller for the PE.

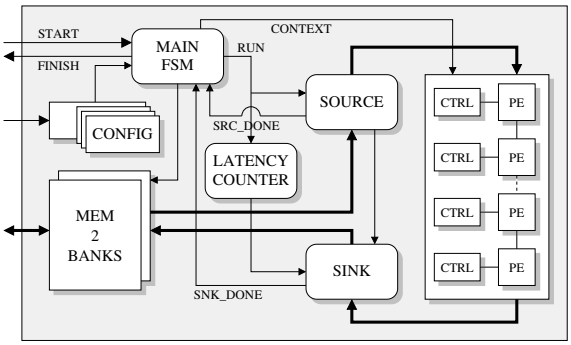


Fig. 7. The Proposed Wavelet Processor.

Should the inexact LOP predicts the leading-one position falsely, the *normalizer* corrects the pre-normalized fraction by shifting it to the left. Here we only need to perform one bit shifting. The rounding logic implements two rounding mechanisms: *rounding to zero* and *rounding to nearest*. Based on the corrected LOP value, the *exponent update* updates the resulting exponent. The *underflow and overflow detector* checks if the resulting exponent lays on the valid FP range. Finally, the sign, the normalized fraction, and the corrected exponent are packed together.

E. Configuration Controller

To cope with various lifting-based forward and inverse DWTs/DWPs, we have separated the configuration-dependent parameters from the PE. Figs. 2 and 3 show how the inputs of the selectors and the multiplier constants are separately drawn on the left side of the figures. The PE is designed to be simple. Thus, no finite state machine is required to control the PE. To support different classes of wavelet filters, which require different types of configurations, we have implemented a multi-context configuration on each PE as depicted in Fig. 6. Each PE is assigned a row index as a unique ID for the configuration. Multiplier constants use the signal data paths to save the wiring cost whereas the selector configuration requires an additional controller path. The context switch is implemented as a memory module. Benefits of a multi-context configuration are: (1) the proposed wavelet processor can be configured to perform the corresponding inverse DWTs/DWPs in a very simple manner; (2) wavelet transforms that use longer wavelet filters can be computed by splitting the lifting steps.

F. Memory Controller

Taking into account that wavelet transform is a multi-resolution signal processing tool, the transformation is performed iteratively on the resulting LP part of the signal, in case of DWT, or on both LP and HP parts, in case of DWP. Fig. 7 depicts the block diagram of the processor along with the PEs and their configuration controller.

1) Main FSM: The main finite state machine controls the wavelet processor. When the transform is initiated, the FSM reads the necessary configurations, such as the transformation level, forward/inverse mode, transform/packet mode, used contexts, etc. from the *config* block. It prepares the source and the sink addresses where the data will be read and stored, and also the length of the data needed to be processed. We exploit the periodicity extension to cope with the boundaries issue in order to compute the transformation on those regions. This implies that source address does not always start on the top of the page. Address masking techniques are applied here to localize the page. The FSM takes care of the possibility of having a longer wavelet transform that has to be split into several lifting steps on the target PEs. The FSM allows multi-level forward/inverse transformation to take place by means of iteration process.

2) Config: It contains the configuration of the transformation. Besides the transformation level and the modes (forward/inverse, DWT/DWP), it also holds the multi-context entry for the longer transformation that cannot be fit into the available PEs. Basically, it informs us which context should be used for the corresponding lifting step.

3) Memory: The memory is organized as 2 banks. The write and read accesses are exclusive, which means that writing to the memory will write to the primary bank and reading from the memory will read from its shadow. This state is switchable automatically, controlled by the *FSM*. When the transformation takes place, the *FSM* grants the memory access to the *source* and *sink* blocks. Writing to or reading from this bank is forbidden and it will generate an error (as an indication of a busy signal).

4) Source and Sink: These blocks generate and automatically increment the read and write addresses. The *source* reads data from the memory and transfers it to the PEs. The *sink* reads data from the PEs and writes it to the memory. A special case is considered when performing transformations that are longer than the available PEs. During the in-between transformation, in case of forward transform, the *sink* will write the data (which corresponds to the intermediate results) to the memory in adjacent manner (resulting L-H-L-H-...). During the final transformation, the *sink* writes the LP and the HP signals into two different pages (resulting L-L-...-H-H-...). The similar handling is also performed by the *source* when performing the inverse transform.

To access the correct page, two address masks are used. The first mask is responsible for the data indexing, and the second mask is responsible for the page indexing.

5) Latency Counter: It delays the *run* signal from the *FSM* to initiate the sink process. The delay amount is different for every lifting steps and it is defined in the *config* block.

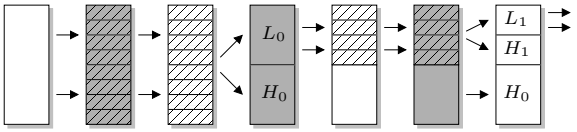


Fig. 8. Forward DWT Process.

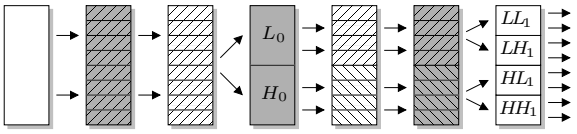


Fig. 9. Forward DWP Process.

Fig. 8 illustrates the N-level and multiple lifting steps DWT. White and grey represent the primary and the shadow banks and diagonal pattern represents the in-between transformation. During the setup, the data is prepared and stored in one bank (this bank is write-only and its shadow is read-only). When the transformation is initiated, this state is reversed, and the source and the sink control the address lines. For each lifting steps, the source reads the written data, and the sink writes the in-between transformation result to the shadow bank. This state is reversed again every time one lifting step is finished, which makes the shadow bank as the primary bank and vice versa. During the last lifting step, the sink stores the LP and the HP results into two different pages. This whole process is performed N times with each iteration decreases the data by half. At each finishing level, a memory copy to transfer the previous HP result to the shadow bank is performed when necessary, e.g. when the lifting steps are odd.

For the DWPs, the HP signal is also transformed, as depicted in Fig. 9. Instead of executing/finalizing the transformation on each signal (LP, and then HP) on each level, the in-between transformations are performed on both signals. With this technique, the banks are not switched during the in-between transformation for both LP and HP signals. Thus, the *FSM* can trigger the *source* to initiate the next data transfer for the next band/page (e.g. HP) without waiting the *sink* to finish from the previous transform. This solution decreases the data preparation time that is caused by exploiting the periodicity extension and the PEs latency. No copy transfer is performed on the DWPs/IDWPs.

IV. Results and Performances

The proposed wavelet processor is based on modular and parametric approaches and is written in VHDL. Wavelet processors with 8 PEs to process eight lifting steps (including the normalization), 8-level unit delays to support higher-order wavelet filters, 16 available contexts to configure the transformations, and 2×512 words memory, are implemented and synthesized. Rounding to nearest with three guard bits is used on all floating-point arithmetics. The design is synthesized using 0.18-μm process. The estimated area and frequency of various data width implementations are reported in Table I.

TABLE II
COMPARISON WITH OTHER LIFTING-BASED ARCHITECTURES.

Architecture	Operating Speed	Area	Filter	Transform	Data Width	Mem. Size	Arithmetic
Andra [9]	200 MHz (0.18-μm)	2.8 mm^2	(5,3) & (9,7)	DWT & IDWT	16-bit	128	Integer
Dillen [10]	110 MHz (FPGA)	–	(5,3) & (9,7)	DWT & IDWT	16-bit	256	Integer
Seo [11]	150 MHz (0.35-μm)	5.6 mm^2	(5,3) & (9,7)	DWT & IDWT	12-bit	512	Integer
Wang [12]	100 MHz (0.18-μm)	1.1 mm^2	Daub-4	DWP	18-bit	8	Integer
Ours	353 MHz (0.18-μm)	3.7 mm^2	Arbitrary	DWT, IDWT DWP, IDWP	32-bit* Configurable	512* Configurable	Floating-Point

TABLE I
ESTIMATED AREA AND FREQUENCY.

Data Width	Est. Area (in mm^2)	% Area for Logics	Est. Freq. (in MHz)
16-bit (10p)	1.841	35.21%	456.62
24-bit (18p)	2.756	37.38%	380.23
32-bit (23p)	3.669	38.56%	353.36

TABLE III
SNR OF DIFFERENT DATA WIDTH (IN DB).

	Daub-6					
	4-level DWT & IDWT			4-level DWP & IDWP		
Source	16-bit	24-bit	32-bit	16-bit	24-bit	32-bit
Sinusoid	46.89	93.17	126.31	46.77	91.51	126.31
Sawtooth	46.87	94.45	125.15	46.70	94.38	125.20
Step	47.71	94.98	133.22	47.54	95.13	132.91
Random	50.86	99.71	129.81	47.24	94.02	125.23
	Symlet-6					
Sinusoid	44.32	91.32	123.06	44.24	90.77	122.89
Sawtooth	43.41	92.39	126.89	43.34	90.91	126.07
Step	40.50	92.50	128.52	39.99	90.99	126.38
Random	40.16	88.68	118.90	37.69	84.87	114.86
	Coiflet-2					
Sinusoid	42.41	86.54	118.15	42.61	87.42	118.19
Sawtooth	42.16	89.37	116.46	41.54	89.38	116.49
Step	39.30	85.23	116.96	39.25	84.53	116.99
Random	42.99	92.39	120.75	39.60	87.73	116.24

The value inside the bracket indicates the number of bits used for the precision. For the 32-bit configuration, the proposed wavelet processor consumes 3.7 mm^2 chip area and has a maximum operating speed of 353 MHz. As a comparison, Andra [9] with 16-bit integer arithmetic can only compute (5,3) and (9,7) filters and requires 2.8 mm^2 area with 200 MHz operating frequency. Table II summarizes the comparison. The other architectures are integer-based with fixed memory size and 12–18-bit data width, and can only perform either DWT/IDWT or DWP.

To measure the level of correctness and to show the flexibility of our design, we perform three different 4-level DWTs and DWPs on some predefined signals. Four different input signals ranged [–1;+1] with 512 samples are used as references. These signals are forward and inverse transformed with Daub-6, Symlet-6, and Coiflet-2 wavelet filters, which do not have integer coefficients. The random signal has a uniform distribution. The SNR values of the different data width implementations are reported in Table III. Depending on the data widths, SNR values vary between 39–133 dB for DWT/IDWT and 38–133 dB for DWP/IDWP, which are sufficient for most applications.

V. CONCLUSIONS

We have proposed a novel architecture that can perform both floating-point forward/inverse DWTs and DWPs. The proposed wavelet processor is configurable and based on M PEs. It can be configured easily to support higher-order lifting polynomials as a result of the factorization of higher-order wavelet filters. Additionally, the proposed architecture takes into account the normalization step that occurs at the end of the forward DWT/DWP or at the beginning of the inverse DWT/DWP. Using 0.18-μm process, the estimated area of the proposed wavelet processor with 32-bit configuration and 2×512 words memory is 3.7 mm^2 and the estimated operating speed is 353 MHz.

REFERENCES

[1] I. Daubechies, "The wavelet transform, time-frequency localization and signal analysis," *IEEE Trans. on Information Theory*, vol. 36, pp. 961–1005, 1990.

[2] S. Mallat, Ed., *A Wavelet Tour of Signal Processing*. Academic Press, Incorporated, 1998.

[3] A. Bradley, "A Wavelet Visible Difference Predictor," *IEEE Trans. Image Processing*, vol. 8, no. 5, pp. 717–730, 1999.

[4] B. Carnero and A. Drygajlo, "Perceptual speech coding and enhancement using frame-synchronized fast wavelet packet transform algorithms," *IEEE Trans. Signal Processing*, vol. 47, pp. 1622–1634, 1999.

[5] W. Sweldens, "The Lifting Scheme: A New Philosophy in Biorthogonal Wavelet Constructions," *Wavelet Applications in Signal and Image Processing*, vol. 3, pp. 68–79, 1995.

[6] I. Daubechies and W. Sweldens, "Factoring Wavelet Transforms into Lifting Steps," *J. Fourier Anal. Appl.*, vol. 4, no. 3, pp. 245–267, 1998.

[7] J. Bruguera and T. Lang, "Leading-one prediction scheme for latency improvement in single datapath floating-point adders," in *Proc. of the Intl. Conference on Computer Design: VLSI in Computers and Processors*, 5-7 Oct. 1998.

[8] H. Suzuki, H. Morinaka, H. Makino, Y. Nakase, K. Mashiko, and T. Sumi, "Leading-zero anticipatory logic for high-speed floating point addition," *IEEE J. Solid-State Circuits*, vol. 31, no. 8, pp. 1157–1164, Aug. 1996.

[9] K. Andra, C. Chakrabarti, and T. Acharya, "A VLSI architecture for lifting-based forward and inverse wavelet transform," *IEEE Trans. Signal Processing*, vol. 50, no. 4, pp. 966–977, 2002.

[10] G. Dillen, B. Georis, J. Legat, and O. Cantineau, "Combined line-based architecture for the 5-3 and 9-7 wavelet transform of JPEG2000," *IEEE Trans. Circuits Syst. Video Technol.*, vol. 13, no. 9, pp. 944–950, Sept. 2003.

[11] Y.-H. Seo and D.-W. Kim, "A New VLSI Architecture of Lifting-Based DWT," *Lecture Notes in Computer Science*, vol. 3985/2006, pp. 146–151, 2006.

[12] C. Wang and W. Gan, "Efficient VLSI Architecture for Lifting-Based Discrete Wavelet Packet Transform," vol. 54, no. 5, pp. 422–426, 2007.