Design of an Application-specific Instruction Set Processor for High-throughput and Scalable FFT

Xuan Guan, Hai Lin and Yunsi Fei
Dept. of Electrical & Computer Engineering
University of Connecticut, Storrs, CT, USA
E-mail: {xug06002,hal06002,yfei}@engr.uconn.edu

Abstract—**Various Orthogonal Frequency Division Multiplexing (OFDM)-based wireless communication standards have raised more stringent requirements on throughput and flexibility of Fast Fourier Transformation (FFT), a kernel data transformation task in communication systems. Application-specific instruction set processor (ASIP) has emerged as a promising solution to meet these requirements. In this paper, we propose a novel ASIP design tailored for FFT computation. We reconstruct the FFT computation flow into a scalable array structure based on an 8-point butterfly unit (BU). Any-point FFT computation can be carried out in the array structure which can easily expand along both the horizontal and vertical dimensions. We incorporate custom register files to reduce memory access. The data address for custom registers in each FFT stage is changed accordingly, and we derive a regular address changing (AC) rule. With the microarchitecture modifications, we extend the instruction set with three custom instructions correspondingly. Our FFT ASIP implementation achieves great performance improvement over the standard FFT software implementation, one TI DSP processor, and one commercial Xtensa ASIP, with the data throughput improvement as 866.5X, 5.9X, 2.3X, respectively. Meanwhile, the area and power consumption overhead of the custom hardware is negligible.**

I. Introduction

Orthogonal frequency-division multiplexing (OFDM) [1], [2], a multi-carrier modulation technique for high data rate digital communications, has seen its increasingly broader usage in various communication standards. The 802.15.3 Multi-band Ultra Wide Band (MB-UWB) standard has the highest data rates ranging from 200 to 480 Mbps [3]. Fast Fourier Transformation (FFT) and inverse FFT (IFFT) are the most time-consuming block in a MB-UWB receiver, which requires a throughput rate of more than 409.6 M sample points per second. Current commercial DSPs, such as Sandbridge's Sandblaster and SODA [4], [5], have to apply multi-cores to meet this real-time processing requirement. Other DSPs like TI's TMS320c6X processor achieve good performance in embedded and real-time applications [6], while it uses 256-bit long instructions, which is not energy-efficient for domain-specific applications.

In addition to the high throughput requirement, another key feature of the current 3G and 4G wireless systems is that they should be easily reprogrammed or reconfigured to support various standards and operating modes. This imposes a requirement of high flexibility on FFT processing. For example, in WiMAX/IEEE 802.16 standard, the system has to adjust the FFT size from 128 to 2048 points to scale the channel bandwidth from 1.25 to 20 MHz for different applications. Therefore, the size of FFT is desired to be changeable under different operation environments. The existing application-specific integrated circuits (ASICs), although can provide high throughput, cannot offer the required flexibility and programmability. On the other hand, the programmable DSP can not meet the system throughput requirement with an efficient energy cost. Application-specific instruction set processor (ASIP) has emerged in recent decades as an intermediate design option between ASIC and general purpose DSP, which extends some base processor core with application-specific custom hardware for computation acceleration [7], [8]. Hence, it can offer both good flexibility with base core software control and high throughput with hardware acceleration, providing a feasible design choice for high-throughput and scalable FFT algorithms.

A. Related Work

Recently, a variety of ASIP implementations have been presented for FFT algorithms, falling into two categories. For the classic Cooley-Tukey FFT (CT-FFT), different ASIP implementations have been presented in [9], [10], [11], which utilize parallelism in the datapath for performance improvement. Specifically in [9], an instruction capable of calculating a whole butterfly operation is implemented, and the computation resources are distributed into three execution stages to reduce the clock cycle. However, long pipelines consume more energy, and the speed-up from one integrated butterfly computation is still not enough to meet the high throughput requirement of the demanding IEEE 802.15.3a UWB communication standard. A vectorized Ultra-Long Instruction Word (ULIW) approach is introduced in [10], which performs eight radix-2 butterfly operations in parallel, at the cost of high gate count and power dissipation for the wide instruction length of 619 bits. The Xtensa ASIP design for FFT adds a set of special instructions which parallelize the data load/store and computation operations [11], thus realizing software pipelining and increasing the overall throughput. However, since each unit FFT computation requires input data loaded from the memory and output results stored to the memory, the number of memory access may be significant and thus would incur great performance degradation and energy consumption.

Acknowledgments: This work was supported by National Science Foundation under grant CPA-0541102.

Different from the instruction and data-level parallelization techniques, the other category of FFT ASIP implementations addresses the memory bottleneck. Baas et al. presented a cache-memory architecture to reduce the communication between the FFT processor and memory [12]. They divide the FFT computation into two epochs of equal length, where each epoch spans several stages, the data points are processed in independent groups of a fixed size, and the intermediate results are temporally stored in the cache. Only at the beginning and end of each epoch, the cache exchanges data with the main memory by load and store operations, thus reducing the memory access. A recent ASIP design has utilized a modified cached FFT (MCFFT) algorithm which can support variable-length epochs [13], to reduce the total number of cache loading and dumping. Further improvement is presented in [14], which interleaves group executions in different epochs to exploit temporal parallelism for higher throughput.

B. Paper Overview

Our work falls into the second category of reducing the memory access by adding custom data registers, with more efficient and scalable architecture based on our proposed array structured FFT. The array FFT ASIP is designed to have both high computation parallelism and low memory access, so that it can meet the throughput and flexibility requirements. We adopt the *epoch* idea in our design, splitting the N-point FFT into two smaller FFT loops. We then reconstruct the inner-loop FFT data flow into an array structure with a basic computation module composed of 4 parallel butterfly units. We extend the basic processor core with the computation module as a functional unit and also add custom register files to store all the intermediate results of the inner loop FFT computation. We propose efficient addressing methods for both data and coefficients. According to the microarchitecture modifications, we customize the instruction set with an additional data manipulation instruction and two data transfer instruction. With the support of an array structure and efficient data address logics, the FFT algorithm can be easily scaled for any size of computation.

The rest of the paper is organized as follows. Section II-A describes our new array structured FFT, and explains the corresponding data address changing algorithm and coefficient addressing method. Section III introduces the architecture of the array FFT-based ASIP, including the custom hardware, the enhanced instruction set, and custom program. Section IV gives simulation results and comparison among several different FFT implementations. Finally, Section V draws conclusions.

II. Scalable Array FFT Algorithm

The Cooley-Tukey algorithm is the most common FFT algorithm [15], which can be applied in two domains, decimation in time (DIT) and decimation in frequency (DIF). In a standard CT-FFT implementation, there are load and store operations for each stage. An N-point FFT (with $log_2 N$ stages) has a total of $N * log_2 N$ loads and stores for the whole dataflow, which may degrade the performance of the FFT algorithm greatly with certain cache-memory architecture.

A. Hierarchical FFT Architecture

To address this problem, we employ the cached FFT (CFFT) structure proposed by Baas et al. [12], where an N-point FFT computation is split into two loops of $\sqrt{N}$-point FFT (assume N is a square number, $N = 2^{2m}$, more general cases will be discussed later), and thus the FFT architecture is decomposed into two epochs consisting of several stages. The processor-memory data communication is reduced to one-time between the two epochs, and the intermediate results inside each epoch are stored in an on-chip storage of size $\sqrt{N}$.

Based on the concept of epoch, we adjust the FFT data flow to obtain a hierarchical and fine-grained FFT processing unit. Taking the 64-point FFT as example, we reconstruct it into a regular array structure, as shown in Fig. 1. The FFT is first split into two epochs, with data shuffling between them. In each epoch, there are eight independent groups of 8-point (8=$\sqrt{64}$) FFT transformations. Each 8-point FFT consists of three stages ($log_2 8$), where each stage is computed by a regular functional module. Fig. 2 shows a modular structure for an 8-point FFT. The functional module contains four parallel butterfly processing units for eight data points. The overall FFT structure in Fig. 1 becomes an array, with 6 columns (corresponding to 2 epochs and 3 stages in each epoch) and 8 rows (corresponding to group computations within an epoch). At the beginning of each epoch, data points are loaded from the memory to on-chip register file. For each stage within an epoch, the intermediate results are stored in the register file, and used for the next stage. At the end of an epoch, content of the register file is dumped into the memory. The advantage of the array structure is its simple FFT processing in each stage, and better scalability to meet the requirement of communication systems for arbitrary size of FFT computations.

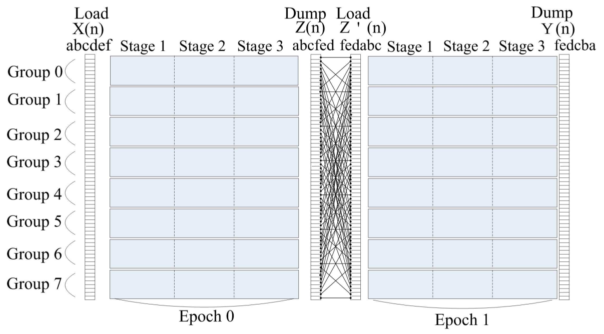

Fig. 1. A regular array architecture for 64-point FFT data flow

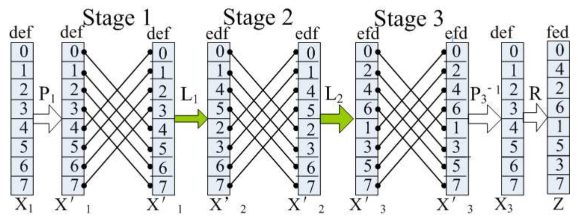

Fig. 2. A modular structure for an 8-point FFT

More generally, we can split N as $N = P * Q$ in case N is not a square number. When $N = 2^{2m-1} = P * Q$, the number of stages within the two epochs would be $p = log_2 P = m, q = log_2 Q = m - 1$, respectively, the size of each FFT computation group within an epoch would be $P = 2^m$, and the number of data-independent FFT groups in an epoch is $Q = 2^{m-1}$.

B. Data Address Change Algorithm

With the FFT computation structure changed, the sequence of the data points for a group in each stage should be changed accordingly to provide the right inputs for the butterfly operations. There are two kinds of data shuffling in the proposed FFT structure. One is the global data shuffle between the two epochs through the off-chip memory, as shown in Fig. 1. The other is for the local data shuffle between two stages within an epoch, as shown in Fig. 2.

The data memory containing all the N data points is only accessed at the beginning and end of each epoch. We use AI_i to represent the address for the loaded input data of epoch i, and AO_i for the output address. For an N-point FFT ($N = 2^n$), AI_0 is $[a_{n-1}a_{n-2}...a_p][a_{p-1}...a_0]$ = $[A_H][A_L]$,where p, q are the total number of stages in epoch 0 and 1, respectively, satisfying $p + q = n$, and $0 \leq p - q \leq 1$.

The other addresses are represented as:

$AO_0 : [a_{n-1}...a_p][a_0a_1...a_{p-1}]$ = $[A_H][\overline{A_L}]$
$AI_1 : [a_0a_1...a_{p-1}][a_{n-1}...a_p]$ = $[\overline{A_L}][\overline{A_H}]$
$AO_1 : [a_0a_1...a_{p-1}][a_p...a_{n-1}]$ = $[\overline{A_L}][\overline{A_H}]$

Since for each P-point FFT, the data outputs should be in an reversed order of inputs, AO_0 is changed from AI_0 by reversing the order of the lower p bits (i.e., $\overline{A_L}$). The same rule applies to AI_1 and AO_1. The data shuffling between the two epochs is to change the data points in memory with an address distance of 1 to a distance of 2^p, i.e., AI_1 is obtained by swapping the higher q bits with the lower p bits of AO_0. An example is shown in Fig. 1 with data sets of X, Z, Z', Y using addresses of AI_0, AO_0, AI_1, AO_1, respectively.

Within an epoch, there are 2^q independent groups of 2^p-point FFT transformations, and all the intermediate results are stored in a register file instead of the memory. Take the 8-point FFT in Fig. 2 as example, where $p = 3$, each stage of the FFT computation has two columns of local data sandwiching the modular butterfly operations, the right column for computation output data stored in the register file and the left column for input data. There are $2^p = 2^3 = 8$ entries in each storage column, and thus the data address for the register file is represented by a 3-bit value, e.g., input address for Stage 1 is def, which is the lower 3 bits of the memory address $abcdef$ for the epoch input. There is data shuffling between outputs of the previous stage and inputs of the current stage. For example, in Stage 3, the data address for the input column is efd, which is obtained by switching the 2nd (to the leftmost) and 3rd bit of edf, the data address for the output column of the previous stage, as shown by the address changing step of L_2 in Fig. 2. The final address fed for this group after the bit-reverse transformation R is the rightmost 3 bits of the memory address for the output data of epoch 0, Z.

In general, the *local address changing* is between two adjacent stages for a group of data points, as represented by green arrows (L_1, L_2) in Fig. 2. In stage j, the input address is obtained by switching the j^{th} and $(j-1)^{th}$ bit (from the leftmost bit) of the previous stage output address.

In addition to the inter-stage local address changing rule as described above, we derive a global address changing rule between the original FFT and our new FFT structure within an epoch. In the original FFT structure, the input addresses for all the stages are the same. For DIF-FFT, the input data address for Stage j is represented as $A_j = a_{p-1}...a_1a_0$. The corresponding new address in the modular FFT, $A_j^{'}$, is obtained by putting the $(p-2)^{th}$ bit of A_j in the j^{th} bit, and other bits are still kept in their original order.

This data changing rule can be proved for any point FFT ($P = 2^p$), by formulating the address changing process into a matrix computation form, as shown below in Fig. 3.

$$\begin{tikzcd}
X_1 \arrow[r, "B_1"] \arrow[d, "P1"] & X_2 \arrow[r, "B_2"] \arrow[d, "P2"] & X_3 \arrow[r, "..."] \arrow[d, "P3"] & X_n \arrow[r, "B_n"] \arrow[d, "Pn"] & X_{n+1} \arrow[d, "Pn+1"] \\
X_1' \arrow[r, "L_1A"] & X_2' \arrow[r, "L_2A"] & X_3' \arrow[r, "..."] & X_n' \arrow[r, "L_nA"] & X_{n+1}'
\end{tikzcd}$$

Fig. 3. Matrix representation of the transformations between data addresses for the two FFT structures

The terms in Fig. 3 include (for each stage j):

1) X_j, a $P \times 1$ vector, representing the original data sequence in stage j. For example, $X_1 = [0, 1, 2, 3, 4, 5, 6, 7]^T$, as shown in the leftmost column in Fig. 2.
2) $X_j^{'}$, a $P \times 1$ vector, representing the new sequence for the input data after the address changing in stage j. For example, $X_1^{'} = [0, 1, 2, 3, 4, 5, 6, 7]^T$, as shown in the second data column in Fig. 2.
3) B_j, a $P \times P$ matrix, representing the original butterfly operations in stage j to generate X_{j+1} from X_j.
4) A, a $P \times P$ matrix, represents the modular FFT operation in one stage in the array FFT structure, which is the same for all stages.
5) L_j, a $P \times P$ matrix, representing the inter-stage local address changing operation (AC), as shown by steps L_1 and L_2 in Fig. 2.
6) P_j, a $P \times P$ matrix, representing the global address changing between X_j and $X_j^{'}$.

For a given number of points P and stage j, the matrixes B_j and A are already known from the FFT algorithm and structure. L_j and P_j are the row-switching matrixes that have performed on $A \times X_j^{'}$ and X_j, respectively, which can be generated by our local and global address changing rule. For example, the sparse matrix P_3 would be

$$P_3 = \left| \begin{array}{cccccccc} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ & & 1 & & & & & \\ & & & & 1 & & & \\ & & & & & & 1 & \\ & 1 & & & & & & \\ & & & 1 & & & & \\ & & & & & 1 & & \\ & & & & & & & 1 \end{array} \right| .$$

With all the matrix and vector expressions ready, we can prove an equation for each stage j: $P_{j+1} \times B_j = L_j \times A \times P_j$. Starting with j=1, multiplying both sides of the equation with X_1 we can get $P_2 \times X_2 = X_2^{'}$. Continue this iterative process, and finally we can prove:

$X_{n+1}^{'} = P_{n+1} \times X_{n+1}$.

It means that by changing the original FFT data computation B_j into a regular computation module A, and applying an address changing step L_i on the intermediate data before they go to module A, the final FFT outputs are the same as in the original FFT structure, i.e., our new FFT structure functions correctly. More detailed proof can be found in the technical report [16].

C. Coefficient Address Algorithm

Besides data points, there are coefficients for butterfly operations that also need to be stored. Because an N-point FFT is split into two P-point FFT loops, there are two kinds of coefficients. One is used for the P-point FFT within an epoch, and the other is the pre-rotation coefficients applied between the two epochs.

For the intra-epoch P-point FFT computations, only number of $P/2$ coefficients are needed, where $P = \sqrt{N}$ when $log_2 N$ is an even number, and $P = \sqrt{2N}$ otherwise. As shown in Fig. 1, Z is the output of epoch 0. Let $s, m \in [0, P-1], l \in [0, Q-1]$. The FFT computations in epoch 0 are:

$$Z(s + Pl) = \sum_{m=0}^{P-1} X(l + Qm) W_P^{sm}$$

The coefficients are evenly distributed between $[W_P^0, W_P^{P/2-1}]$, with $W_P = e^{-\frac{2j\pi}{P}}$. Since this kind of coefficients is frequently accessed by different groups in different stages, they can be stored in an on-chip ROM. The ROM address will range from 0 to $P/2 - 1$. In each stage j, there are number of $P/8$ 8-point butterfly operations (BU), and each BU needs 4 coefficients for computation. The addresses for the 4 coefficients depend on the stage number j within an epoch, and the module number i within the stage. Take a 32-point FFT for example, there are 5 stages and 4 BU modules. In Stage 2, the 16 coefficient addresses for module 1 through module 4 are (0,0,0,0), (0,0,0,0), (8,8,8,8), (8,8,8,8), which increase with a stride of 8 for every 8 steps. In general, the address in Stage j starts from 0 and increases with a stride of $P/2^j$ for every $P/2^j$ steps. The 4 coefficient addresses for the i^{th} BU module in Stage j are expressed as:

$$\begin{cases} p_1 = \left\lfloor \dfrac{4i-4}{\frac{P}{2^j}} \right\rfloor \cdot \dfrac{P}{2^j} \\ p_2 = \left\lfloor \dfrac{4i-3}{\frac{P}{2^j}} \right\rfloor \cdot \dfrac{P}{2^j}, \quad i = 1 \sim \dfrac{P}{8} \\ p_3 = \left\lfloor \dfrac{4i-2}{\frac{P}{2^j}} \right\rfloor \cdot \dfrac{P}{2^j}, \quad j = 1 \sim \log_2 P \\ p_4 = \left\lfloor \dfrac{4i-1}{\frac{P}{2^j}} \right\rfloor \cdot \dfrac{P}{2^j} \end{cases}$$

The second kind of coefficients is the pre-rotation weights applied to the intermediate output results at the end of the first epoch, Z. The input data for the second epoch, Z', is got by:

$$Z'(s + Pl) = W_N^{sl} Z(s + Pl)$$

The coefficients between the two epochs are W_N^{sl}. There are $N/2$ different inter-epoch coefficients evenly distributed between $[W_N^0, W_N^{N/2-1}]$, and can be stored in the off-chip memory. Because of the circular symmetry in a complex plane, only the first $\frac{N}{8} + 1$ coefficients need to be stored, and others can be easily generated by conjugation or swapping the real and imaginary parts of a complex coefficient. The real address needed to locate the coefficient in memory is $(sl) mod(N/8)$ when $\left\lfloor \frac{sl}{N/8} \right\rfloor$ is an even number, and $N/8 - (sl) mod(N/8)$ when $\left\lfloor \frac{sl}{N/8} \right\rfloor$ is an odd number, which will first locate the complex data $[a, b]$ in memory, and then generate the corresponding real coefficient, among $[b, a]$, $[-b, a]$ and $[-a, b]$.

III. Design of an ASIP with Array Structured FFT

In this section, we will explain the implementation of our proposed FFT processor, based on the description of the array FFT algorithm and the addressing logic for both data and coefficients.

A. Custom Hardware Extension

As the ASIP is specificly tailored for FFT computations, a highly customized data path is optimized to speed up the computation, and the instruction decoder may need to be modified to control the data flow. As shown in Fig. 4, in the custom hardware, we incorporate a Basic Unit (BU) composed by four parallel butterfly units, a separate custom register file (CRF) to store all the intermediate data of the FFT computations in each epoch, and an on-chip ROM for the intra-epoch coefficients. An address changing logic (AC) is added in the decoder to give the right data address and coefficient address.

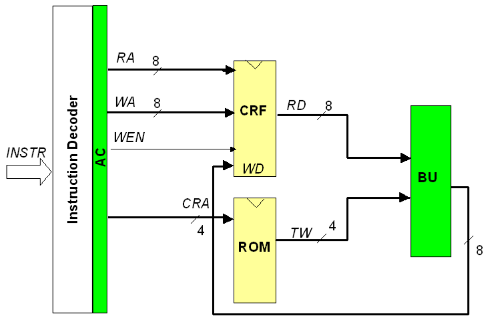

Fig. 4. Structure of the custom hardware extension

In each operation cycle, the BU gets 8 data points from the CRF, with the generated register file address (RA). The computed outputs are stored back to the CRF with the same address (WA), replacing the previous input data. The BU operations are performed on independent groups, stage by stage. For example, as shown in Fig. 1, the BU operations are applied in a horizontal order first (from Stage 1 to Stage 2, and so on for the first group of data points), and then the vertical order (from the top group to the bottom group).

One advantage of this architecture is that it removes all the address calculation instructions from the assembly code of FFT program, and moves this computation to additional hardware in the instruction decoder. Thus, the execution cycles for the FFT computation may be reduced.

Another advantage comes from the use of custom register files. All the intermediate results of FFT computations within an epoch are stored in the custom register file, and then used for the next stage, thus reducing access to memory. The only load and store operations between memory and custom register files are at the beginning and the end of those two epochs.

B. Application-specific Instructions

As shown in Fig. 4, in our array FFT-based ASIP architecture, there is a special vectorized FFT functional unit, BU, which calculates 4 radix-2 butterfly operations in parallel. Although the execution of BU needs eight input addresses and eight output addresses, with the array FFT architecture, the corresponding new instruction $BUT4$ only needs the current module number, which is between $[1, \frac{P}{8}]$, and the current stage number, for data address calculation performed in the decoder (AC).

Before entering the FFT BU computation iterations along the vertical groups and horizontal stages, a new instruction $LDIN$ is used to load input data points from the main memory to custom registers. This instruction has two operands, one is the source address for the main memory, and the other is the destination address for the custom register file (CRF). With a 64-bit data bus, each $LDIN$ instruction loads two data points. Hence, for an N-point FFT, this instruction needs to be repeated for N times in total before the epoch computation. The similar operation is performed at the end of BU iterations to dump FFT computation results from the custom registers to the main memory by instantiating another new instruction, $STOUT$.

The processor pipeline is then extended correspondingly to accommodate the three application-specific instructions, $BUT4$, $LDIN$, and $STOUT$, with modifications in the decoding (ID) and executions (EX) stages.

C. Custom FFT Program running on the Array FFT-based ASIP

With all the custom hardware extension and new instructions in place, the FFT algorithm is reprogrammed to incorporate the new instructions. Algorithm 1 illustrates the pseudo-code for an N-point FFT computation, where there are 2 epochs, which have number of p and q computation stages respectively, and each stage has $\frac{N}{P}$ independent FFT groups with $\frac{P}{8}$ module computations in each group. The inner-loop computation is just one $BUT4$ instruction. All the intermediate data within an epoch is hold in a P-entry CRF.

Algorithm 1 FFT Algorithm running on the array FFT-based ASIP

```
Input: N: total number of FFT points;
    p (q): total number of stages in epoch 0 (1);
    e: index of the current epoch;
    d: index of the data groups loaded from the CRF;
    j: index of the current stage;
    i: index of the BU operation in each stage;
1:  for $e = 0$ to $1$ do
2:    if e=0 then
3:      s=p
4:    else
5:      s=q
6:    end if
7:    for $d = 1$ to $2^q$ do
8:      Repeat LDIN for $2^p$ times;
9:      for $j = 1$ to $s$ do
10:       for $i = 1$ to $2^p/8$ do
11:         BUT4 (i, j) ;
12:       end for
13:     end for
14:     if e=0 then
15:       Repeat coefficient pre-rotation for $2^p$ times;
16:     end if
17:     Repeat STOUT for $2^p$ times;
18:   end for
19: end for
```

IV. Simulations and Evaluations

In this section, we present experimental results on evaluating both the hardware cost and software performance, and compare them with results from other approaches.

The extra hardware modules for the array FFT ASIP include the Butterfly Unit (BU), the Address Changing Logic (AC), custom register files ($CRFs$), and coefficient ROM, as described in previous sections. We designed all the modules in VHDL and synthesized them by Synopsys's Design compiler [17], using the standard TSMC13GFSG library under CMOS 0.18 μm technology. The critical path of the AC module is negligible, and that of the BU module is 3.2 ns, and hence the processor can work at a clock speed of up to 300 MHz. The total gate count of the BU and AC modules estimated by Design compiler is 17324, and the gate count of CRF and coefficient ROM is 15764. The total gate count for the extra hardware is 33K, which is acceptable as an accelerator compared to the basic PISA core (106K gates including a 32KB cache). The power consumption of BU and AC is 17.68 mW at 300 MHz.

For a 1024-point FFT computation implemented on our ASIP, the data throughput can reach to 440.6 Mbps which attains UWB-OFDM specifications. In addition, the array FFT ASIP is designed to have ease of scalability as a primary objective. The FFT algorithm is reprogrammed and recompiled for different FFT sizes. We modified the Simplescalar software suite for our FFT ASIP implementation with the base processor core modeled on PISA [18]. Table I presents the simulation results for data throughput for different FFT sizes with our array-based ASIP. It shows that the throughput is decreasing slightly as the number of points increases. This is because with the FFT size grows, more cycles will be spent on the software control of reusing the modular BU, e.g., more loop iterations, and thus the hardware acceleration will be offset a little by the software overhead.

TABLE I
SIMULATION RESULTS OF DATA THROUGHPUT FOR DIFFERENT FFT SIZES

FFT Points	Total Cycle Number	Data Throughput (Mbps)
64	197	584.7
128	402	572.2
256	851	540.9
512	1828	502.2
1024	4168	440.6

We also simulate a 1024-point FFT computation for four different implementations: the standard pure software implementation on the base PISA core, the optimized software implementation on TI's TMS320C6713 DSP [6], the custom algorithm on the FFT ASIP provided by Xtensa [11], and our array FFT-based implementation. The TI DSP uses 256-bit long instructions which can issue 8 operations per cycle (2 LD/ST, 2 MULT, 2 ADD/SUB, and

TABLE II
COMPARISON AMONG DIFFERENT FFT IMPLEMENTATIONS FOR 1024-POINT COMPUTATION

	Imple 1:	**Imple 2:**	**Imple 3:**	**Imple 4:**			
	Standard SW FFT	**TI's DSP SW FFT**	**Xtensa FFT ASIP**	**Proposed Array FFT ASIP**	**Improvement over Imple 1 (X)**	**Improvement over Imple 2 (X)**	**Improvement over Imple 3 (X)**
# of Cycles	3611551	24976	9705	4168	866.5	5.9	2.3
# of Loads	91675	-	5494	1059	86.6	-	5.2
# of Stores	91677	-	5301	1192	76.9	-	4.4
# of Data Cache Misses	114575	9944	284	106	1080.6	93.8	2.6

2 BR). With a 128-bit data bus, 4 complex numbers can be loaded or stored in parallel. Since there is no special function unit for FFT, the average processing time for a butterfly operation is about 4 cycles after software pipelining. In Xtensa FFT ASIP, they parallelize the butterfly computation with the load and store operations by using their new TIE instructions. This way, they hide the butterfly computations by load and store operations for the next set of data points.

We modified Simplescalar for the two custom ASIP implementations, respectively. The simulation results for a 1024-point FFT under different implementations are given in Table II. In the table, we present the results for the total number of execution cycles, number of data loads and store operations (not available for the TI DSP though), and the data cache miss counts. We also give the performance improvement of our implementation over the baseline software implementation, the TI's DSP implementation and the Xtensa ASIP in the last three columns. The data throughput improvement is 866.5X, 5.9X and 2.3X, respectively. The overall performance improvement of our FFT ASIP over Xtensa's is contributed by the reduction in load and store instructions (5.2X and 4.4X), number of data cache misses (2.6X), and parallel computation in butterfly operations. We notice that compared to the Xtensa implementation, our number of loads and stores is about 1/4-1/5, however, the throughput is not 4 times higher. This is because we do not parallelize the load and store operations with the butterfly computations, hence, the cycle count saving from the reduction of loads and stores is offset by the addition of computation cycles. For Xtensa's FFT ASIP, even if they employ a butterfly unit with four parallel computations as we have used, their throughput will not change, because the bottleneck of their FFT algorithm is the load and store operations, and hence the speedup in computation does not affect the performance at all.

V. CONCLUSIONS

This paper proposes a highly efficient and flexible array FFT ASIP design, which can meet both the high throughput and flexibility requirements of various contemporary OFDM standards. We derive an address changing rule, which manages the intermediate register data and memory data storage, and works together with the restructured FFT to guarantee right data flow. With this regular FFT structure, we propose our ASIP design for FFT computation. New application-specific instructions are incorporated to accelerate the butterfly computations in groups, and move data between the main memory and custom hardware. It has been demonstrated that our array FFT-based ASIP implementation can improve the overall performance greatly by reducing the number of main memory access and utilizing computation parallelism. The array structure makes the hardware/software implementation of FFT easily scalable to any points. The cost in both custom hardware area and power consumption is acceptable.

REFERENCES

[1] G. J. Byung, S. S. Byung, H. S. Myung, and Y. S. Kim. A high-speed FFT processor for OFDM systems. In *Proc. Int. Symp. on Circuits & Systems*, pages 281–284, May 2002.

[2] N. Weste and D. J. Skellern. VLSI for OFDM. In *IEEE Commun. Mag.*, pages 127–131, Oct. 1998.

[3] C. Now, L. Lampe, and R. Schober. Performance analysis of multiband OFDM for UWB communication. In *Int. Conf. on Commun.*, pages 2573 – 2578, May 2005.

[4] M. Schulte, J. Glossner, S. Jinturkar, M. Moudgill, S. Mamidi, and S. Vassiliadis. A low-power multithreaded processor for software defined radio. *J. VLSI Signal Process. Syst.*, 43:143–159, 2006.

[5] Y. Lin, H. Lee, M. Woh, Y. Harel, S. A. Mahlke, T. N. Mudge, C. Chakrabarti, and K. Flautner. Soda: A low-power architecture for software radio. In *Proc. Int. Symp. on Computer Architecture*, pages 89–101, June 2006.

[6] *TMS320C6713 Floating-Point Digital Signal Processor*. TI datasheet, 2005.

[7] D. Goodwin and D. Petkov. Automatic generation of application specific processors. In *Int. Conf. Compilers, Architecture & Synthesis for Embedded Systems*, pages 137–147, Oct. 2003.

[8] K. L. Heo, S. M. Cho, J. H. Lee, and M. H. Sunwoo. Application-specific DSP architecture for fast Fourier transform. In *Int. Conf. on Application-Specific Systems, Architectures, & Processors*, pages 369–377, June 2003.

[9] M. Nicola, G. Masera, M. Zamboni, H. Ishebabi, D. Kammler, G. Ascheid, and H. Meyr. FFT processor: A case study in ASIP development. In *IST Mobile Summit*, 2005.

[10] R. Chidambaram, R. V. Leuken, M. Quax, I. Held, and J. Huisken. A multistandard FFT processor for wireless system-on-chip implementations. In *Proc. Int. Symp. on Circuits & Systems*, pages 4–7, May 2006.

[11] Implementing the Fast Fourier Transform for the Xtensa Processor. [`http://www1.tensilica.com/pdf/FFT_apnote.pdf/`].

[12] B. M. Baas. A low-power, high-performance, 1024-point FFT processor. *IEEE Journal of Solid-State Circuits*, pages 380–387, Mar. 1999.

[13] O. Atak, A. Atalar, E. Arikan, H. Ishebabi, D. Kammler, G. Ascheid, H. Meyr, and M. Nicolaand G. Masera. Design of application specific processors for the cached FFT algorithm. In *Proc. Int. Conf. Acoustics Speech & Signal Processing*, pages 14–19, May 2006.

[14] H. Ishebabi, G. Ascheid, H. Meyr, O. Atak, A. Atalar, and E. Arikan. An efficient parallelization technique for high throughput FFT-ASIPs. In *Proc. Int. Symp. on Circuits & Systems*, pages 21–24, May 2006.

[15] W. Cooley and J. W. Tukey. An algorithm for the machine calculation of a complex Fourier series. *Math. Computat*, 19:297–301, 1965.

[16] X. Guan. Addressing logic for array-structure FFT. Technical report, Department of Electrical and Computer Engineering, University of Connecticut. [`http://laurel.engr.uconn.edu/∼ggxuan/FFTrp.pdf/`].

[17] Sysnopsys. [`http://www.synopsys.com/products/logic/design_compiler.html`].

[18] C. Vieri. Pendulum: A reversible computer architecture. Master's thesis, MIT Artificial Intelligence Laboratory, 1995.