electronics

MDPI

Article

Automated Scalable Address Generation Patterns for 2-Dimensional Folding Schemes in Radix-2 FFT Implementations

Felipe Minotta*, Manuel Jimenez and Domingo Rodriguez

Electrical and Computer Engineering Department, University of Puerto Rico at Mayagüez, Mayagüez 00681-9000, Puerto Rico; manuel.jimenez1@upr.edu (M.J.); domingo.rodriguez@upr.edu (D.R.)
* Correspondence: felipe.minotta@upr.edu

Received: 30 December 2017; Accepted: 1 March 2018; Published: 3 March 2018

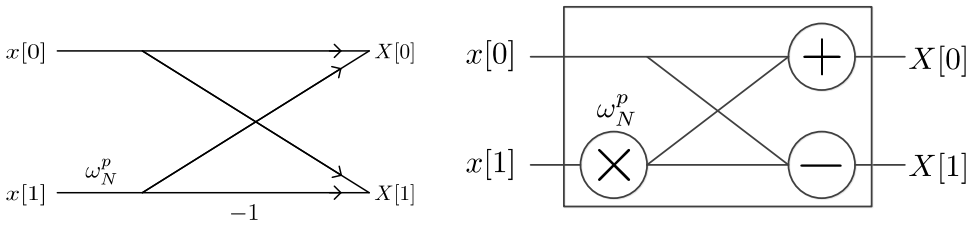
Abstract: Hardware-based implementations of the Fast Fourier Transform (FFT) are highly regarded as they provide improved performance characteristics with respect to software-based sequential solutions. Due to the high number of operations involved in calculations, most hardware-based FFT approaches completely or partially fold their structure to achieve an efficient use of resources. A folding operation requires a permutation block, which is typically implemented using either permutation logic or address generation. Addressing schemes offer resource-efficient advantages when compared to permutation logic. We propose a systematic and scalable procedure for generating permutation-based address patterns for any power-of-2 transform size algorithm and any folding factor in FFT cores. To support this procedure, we develop a mathematical formulation based on Kronecker products algebra for address sequence generation and data flow pattern in FFT core computations, a well-defined procedure for scaling address generation schemes, and an improved approach in the overall automated generation of FFT cores. We have also performed an analysis and comparison of the proposed hardware design performance with respect to a similar strategy reported in the recent literature in terms of clock latency, performance, and hardware resources. Evaluations were carried on a Xilinx Virtex-7 FPGA (Field Programmable Gate Array) used as implementation target.

Keywords: Discrete Fourier Transform; Fast Fourier Transform; Linear Transform; Pease Factorization; scalable address generation; Digital Signal Processing; hardware generation

1. Introduction

The Fast Fourier Transform (FFT) is the main block in many electronic communications, signal processing, and scientific computing operations. It allows for the fast computation of the Discrete Fourier Transform (DFT). The DFT, in turn, is used to obtain the spectrum of any finite discrete signal or numeric sequence. Many FFT factorizations have been proposed since its first formulation, including Cooley-Tucky, Pease, and Hartley, among others, each with different advantages and limitations [1]. As regularity becomes a desirable characteristic when FPGA-based solutions are pursued, this work uses the Pease FFT formulation due to its *constant geometry* structure [2,3]. Our work formulates a set of algorithms for *addressing schemes* or *scheduling operations* in FFT computations. Figure 1a shows the standard representation of a 2-point FFT calculation, also called a *butterfly* operation. In Figure 1b we introduce an alternate butterfly notation which we will use for the butterfly operation. This notation facilitates the formulation of FFT algorithms in terms of Kronecker products algebra, introduced later in this paper. An N-point FFT operation is divided into S stages, where $S = \log_2(N)$ is a positive integer.

Figure 2 shows the structure of an 8-point Pease FFT implementation, illustrating a representative N-point structure where butterflies are presented as *processing elements* (PEs) in the rectangular boxes. Figure 2 also allows for introducing the concepts of horizontal and scalable vertical folding. A horizontal folding consists on reducing by half the number of columns in a FFT structure. A horizontal folding is considered maximum when successively halving the column count, reduces the total number of columns to one, maximizing the usability of hardware resources. The vertical folding consists on reducing the number of butterflies in the single column resulting resulting after a maximum horizontal folding has been applied. A vertical folding is considered scalable when the factor introduced to reduce the number of butterflies in the single column can be conveniently adjusted to fit a particular hardware resource limitation. A scalable vertical folding process affects the overall latency associated with memory operations. It also creates the need for a permutation block which is in charge of controlling the data flow between stages. A *vertical folding factor* ϕ is defined such that $\phi B = N$, where $B = 2\beta$ is defined as the number of memory banks required in the implementation, and β is defined as the number of processing elements in a single column. A permutation block consists of two data switches and B memory banks. For small values of β, the memory latency is high and the amount of consumed hardware resources is low. On the contrary, for large values of β, the latency is low and the amount consumed hardware resources is high. The resulting structure may be classified as a Burst I/O design since the computation must wait for the entire FFT calculation to finish before providing a new input signal to the hardware computational structure (HCS).



(**a**) Standard representation (**b**) Notation used in this work

Figure 1. Representations for a 2-point Fast Fourier Transform (FFT) operation.

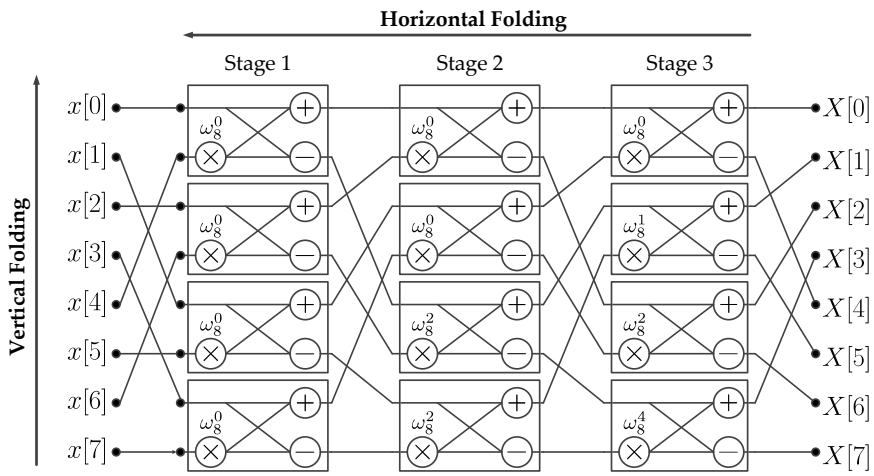


Figure 2. 8-point Pease FFT architecture.

Permutation blocks in folded hardware FFTs can be implemented in one of two ways: either using dedicated permutation logic or using address generation schemes. In the first approach, a complex

dedicated logic circuit controls the data flow, while in the second, data are stored in memory banks and a simple controller generates the addresses depending on the needed permutations.

Several works have been reported using scalable permutation logic with variable number of points and folding factors. A major drawback of these approaches is the large amount of consumed hardware resources [4–6]. Addressing schemes were introduced by Johnson as an alternative able to significantly reduce the amount of consumed hardware resources [7]. Many FFT implementations have been reported to follow the concept of addressing schemes. Some have used a non-scalable approach targeted to specific applications [8–12]. Others provide scalable implementations in terms the number of FFT points [13–21].

Francheti et al. proposed a framework for the hardware generation of linear signal transforms [22]. Although their work produced functional solutions, this framework did not describe in sufficient detail the address generation scheme utilized in its table lookup procedure for twiddle factors or phase factors operations. It is essential to develop FFT computational algorithms with optimal data placement and access strategies, to obtain data accesses with zero conflicts. Data address generation schemes need to be designed to accomplish this task. Our address generation scheme satisfies this design requirement without the need of a reordering procedure, and it is independent of pipeline overlapping techniques, such as those reported in works by Richardson et al. [23].

Despite this diverse set of implementations, there are few approaches addressing the issue of using an HCS to allow for scalable folding processing. The main motivation behind this work is the existing need for a generalized addressing scheme for scalable algorithms based on maximum horizontal folding. To the best of our knowledge, no such generalized addressing scheme has been previously proposed. The key benefits of a our proposed addressing scheme can be summarized as follows:

- a set of methods to generate scalable FFT cores based on an address generation scheme for Field Programmable Gate Array (FPGA) implementation when the vertical folding factor is optimized,
- a mathematical procedure to automatically generate address patterns in FFT computations,
- a set of stride group permutations to guide data flow in hardware architectures, and
- a set of guidelines about space/time tradeoffs for FFT algorithm developers.

The rest of this paper is organized as follows: Section 2 describes the mathematical foundations of the FFT core. Section 3 describes the hardware implementation procedure. Section 4 evaluates the design performance in terms of clock latency and hardware resources when the target HCS is a Xilinx FPGA. Finally, Section 5 provides concluding remarks.

2. Mathematical Preliminaries

In this section, we start by presenting definitions and mathematical concepts used for the formulation of the proposed scalable address generation scheme.

2.1. Definitions

Let us first define two main matrix operations frequently used in this paper: Kronecker products and matrix direct sum. If $A \in \mathbb{R}^{m_1 \times n_1}$ and $B \in \mathbb{R}^{m_2 \times n_2}$, then their Kronecker product is defined as $C \in \mathbb{R}^{m_1 m_2 \times n_1 n_2}$ [24], where

$$C = A \otimes B = [a[k,l] \times B]_{k \in Z_{m_1}, l \in Z_{n_1}} . \tag{1}$$

The matrix direct sum of matrix set $A_1, A_2, ..., A_n$ is defined as

$$\bigoplus_{d=1}^{n} A_d = diag(A_1, A_2, ..., A_n) = \begin{bmatrix} A_1 & & & \\ & A_2 & & \\ & & \ddots & \\ & & & A_n \end{bmatrix} . \tag{2}$$

Now let us define the Pease formulation for an N-point FFT in Kronecker products,

$$F_N = \left\{ \prod_{i=1}^{S} (L_2^N (I_{N/2} \otimes F_2) T_i) \right\} R_N, \tag{3}$$

where L_2^N is a stride by 2 permutation of order N, $I_{N/2}$ is an identity matrix of size $N/2$, R_N is termed the bit reversal permutation of order N, and F_2 is the radix-2 butterfly defined as

$$F_2 = \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix}. \tag{4}$$

T_i is the twiddle or phase factor matrix defined as

$$T_i = \bigoplus_{r=0}^{2^{S-i}-1} (I_{2^{i-1}} \otimes W_2^q), \tag{5}$$

where $q = r \times 2^{i-1}$ and,

$$W_2^q = \begin{bmatrix} 1 & 0 \\ 0 & \omega_N \end{bmatrix}^q, \tag{6}$$

where $\omega_N = \mathrm{e}^{-j\frac{2\pi}{N}}$ [25]. Furthermore, the direct computation shows that

$$R_N = (I_{N/2} \otimes L_2^2)(I_{N/4} \otimes L_2^4) \ldots (I_2 \otimes L_2^{N/2}) L_2^N, \tag{7}$$

where L_2^N is, again, a stride by 2 permutation [26].

2.2. Kronecker Products Formulation of the Xilinx FFT Radix-2 Burst I/O Architecture

Our work is based on an equivalent architecture for the Xilinx FFT Radix-2 Burst I/O core (Figure 3) for implementing DFT computations [27]. Our goal is to present a systematic and scalable procedure for generating permutation-based address patterns for any power-of-2 transform size FFT algorithm and any folding factor. For this purpose, we propose to use the language of Kronecker products algebra as a mathematical tool. Therefore, we start by representing mathematically this architecture using a specific DFT computation example, for $N = 8$.

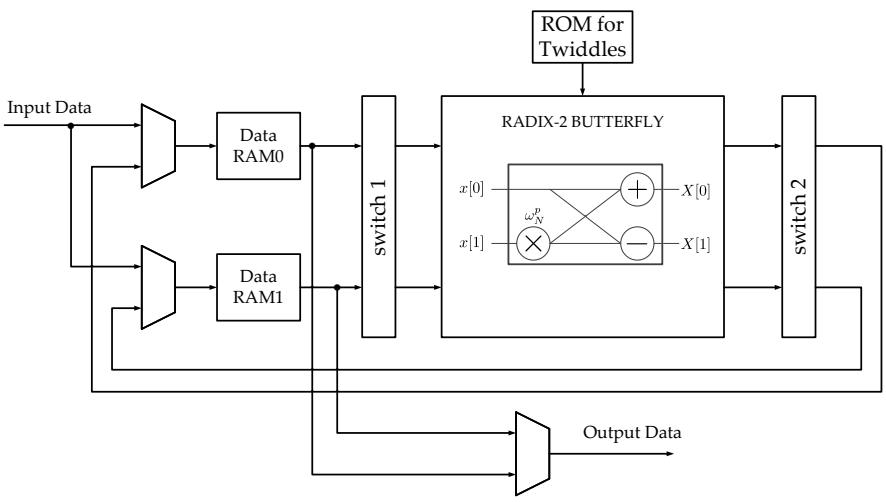


Figure 3. Xilinx Radix-2 Burst I/O architecture. RAM: random access memory; ROM: read only memory

At the beginning of the process, in the Xilinx architecture, data points are stored serially in RAM0 and RAM1 through the input data bus (Figure 3). The first 4 data points are stored in RAM0 and the last 4 are stored in RAM1. Following the Pease FFT formulation, the addressing schemes can be expressed as:

$$RA_0 = R_4 \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}^T, \tag{8}$$

$$RA_{i+1} = L_2^4 RA_i \quad 0 \leq i \leq 2 \ , \tag{9}$$

$$RB_0 = R_4 \begin{bmatrix} 0 & 1 & 2 & 3 \end{bmatrix}^T, \tag{10}$$

$$RB_{i+1} = (J_2 \otimes I_2) L_2^4 RB_i \quad 0 \leq i \leq 2 \ , \tag{11}$$

where RA_i is the addressing scheme for RAM0, RB_i is the addressing scheme for RAM1, i is the *stage identifier*, and J_2 is a data exchange matrix. In the first stage, switch 1 in Figure 3 does not perform any permutation over the input data. In the rest of the stages, switch 1 does not perform any permutation in the first half, and interchanges the input data in the second half. That is,

$$\begin{aligned} SW1_{out} &= SW1_{in}; \quad i = 0, 0 \leq b \leq 3, \\ SW1_{out} &= SW1_{in}; \quad i > 0, 0 \leq b \leq 1, \\ SW1_{out} &= J_2 SW1_{in}; \quad i > 0, 1 \leq b \leq 3, \end{aligned} \tag{12}$$

where $SW1_{out}$ is the output of switch 1, $SW1_{in}$ is the input of switch 1, b is the internal step counter. The single butterfly operation can be expressed as:

$$BF_{out} = \left[\left(\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix} \begin{bmatrix} 1 & 0 \\ 0 & \omega_8 \end{bmatrix}^p \odot \begin{bmatrix} 1 & 1 \\ 1 & -1 \end{bmatrix} \right) BF_{in} \right], \tag{13}$$

where BF_{out} and BF_{in} are the radix-2 butterfly's output and input in Figure 3, respectively. $\omega_8 = \mathrm{e}^{-j\frac{2\pi}{8}}$ and,

$$p = 8 \times 2^{-(i+1)} \left\lfloor \frac{(2^{i+1} b)}{8} \right\rfloor, \tag{14}$$

where $\lfloor a \rfloor$ denotes the floor operation on a. Switch 2 in Figure 3 does not perform any permutation for the even-indexed values of b, the internal step counter. On the other hand, switch 2 interchanges the input data on the odd-values of b. That is,

$$\begin{aligned} SW2_{out} &= SW2_{in}; \quad i \geq 0, b = \{0, 2\}, \\ SW2_{out} &= J_2 SW2_{in}; \quad i \geq 0, b = \{1, 3\}, \end{aligned} \tag{15}$$

where $SW2_{out}$ is the output of the switch 2 and $SW2_{in}$ is the input of the switch 2. This process applies for any value of N. The equations included in this subsection are presented as an example of how a signal transform hardware architecture can be represented mathematically under a unified computational framework.

It is important to notice that this architecture only uses one butterfly. What would happen with the addressing schemes, switches, and phase factor scheduling if two or more butterflies were used? the next subsection addresses this relevant issue.

2.3. Kronecker Products Formulation of the Scalable FFT with Address Generation

Figure 4 shows an FFT architecture designed with the required blocks for our addressing scheme, under the control of a Finite State Machine (FSM). The*Data Address Generator* (DAG) produces the addresses depending on the needed permutations. The *Memory Banks* (MBs) store the input signal and the processed data as well. The *Data Switch Read* (DSR) permutates the data points from the MBs to process them correctly. The *Processing Elements* (PEs) perform the mathematical operations.

Finally, the *Data Switch Write* (DSW) permutates the data to avoid memory conflicts. Every block is mathematically represented as an entity, with specified inputs and outputs. To obtain these mathematical representations, we use again the language of matrix algebra and tensor or Kronecker products. In the next subsection, we define the general aspects of these mathematical representations.

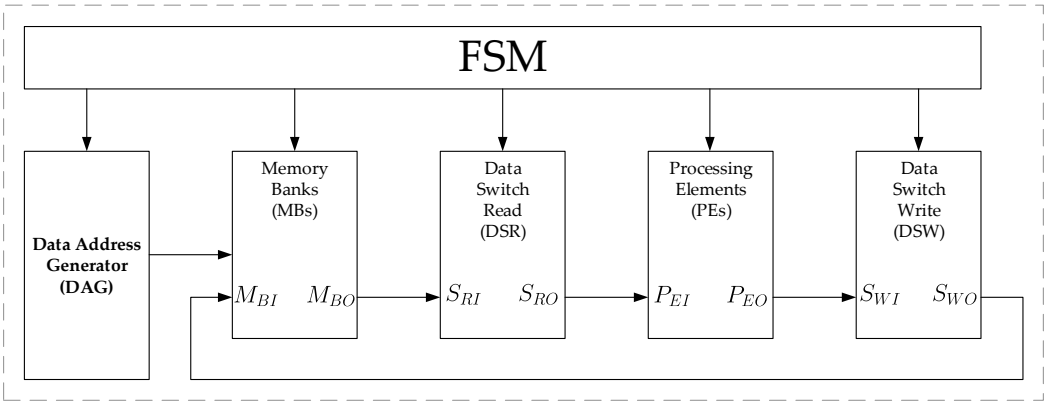


Figure 4. FFT architecture with our proposed addressing scheme. FSM: Finite State Machine.

The DAG must perform two kinds of permutations over the data points, regardless of the folding factor: a bit-reversal permutation in the first stage and stride-2 permutations in the next stages. Furthermore, since our implementation is intended to be used as a "conflict free memory addressing strategy" [7], these permutations have to be performed by rewriting the data points into the same locations as being read. Since each butterfly needs two different values at the same time, 2β different addressing schemes are needed. However, this work demonstrates that for any folding factor, only two array addressing schemes are needed. An *array addressing scheme* is defined as a systematic procedure for simultaneously pointing to B blocks of an ordered set of memory data. For an N-point FFT with vertical folding factor ϕ the array addressing schemes can be expressed as:

$$X_0 = R_\phi \begin{bmatrix} 0 & 1 & 2 & \dots & \phi - 1 \end{bmatrix}^T, \tag{16}$$

$$X_{i+1} = L_2^\phi X_i \quad 0 \leq i \leq S - 1 \ , \tag{17}$$

$$Y_0 = R_\phi \begin{bmatrix} 0 & 1 & 2 & \dots & \phi - 1 \end{bmatrix}^T, \tag{18}$$

$$Y_{i+1} = (J_2 \otimes I_{\phi/2}) L_2^\phi Y_i \quad 0 \leq i \leq S - 1 \ , \tag{19}$$

where X_i is the *even array addressing scheme*, Y_i is the *odd array addressing scheme*, i is the *stage identifier*, J_2 is an exchange matrix, L_2^ϕ and R_ϕ are, again, a stride permutation and a bit-reversal permutation, respectively, and the superscript T denotes matrix transpose operation.

The addressing scheme requires B memory banks. Since each processing element or butterfly accepts two complex numeric values at the same time, and there are β processing elements, the number of memory banks is equal to $B = 2\beta$. Each memory bank consists of ϕ memory locations. The memory banks can be indexed using the following scheme,

$${}_i^b M_k[l]; k \in Z_B, l \in Z_\phi, \tag{20}$$

where ${}^{b}_{i}M_k$ is the k-th memory bank, and $0 \leq b \leq \phi - 1$ is the *internal step counter*. We use row major storage which means that each row contains the complex numeric values of an entire memory bank. That is,

$$ {}^{b}_{i}M = \begin{bmatrix} M_0[0] & M_0[1] & \dots & M_0[l] & \dots & M_0[\phi-1] \\ M_1[0] & M_1[1] & \dots & M_1[l] & \dots & M_1[\phi-1] \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ M_k[0] & M_k[1] & \dots & M_k[l] & \dots & M_k[\phi-1] \\ \vdots & \vdots & \ddots & \vdots & \ddots & \vdots \\ M_{B-1}[0] & M_{B-1}[1] & \dots & M_{B-1}[l] & \dots & M_{B-1}[\phi-1] \end{bmatrix}. \tag{21} $$

To extract the numeric values stored in the memory banks, we perform an inner product stage operation. This operation consists in first creating a *state indexing matrix*

$$ {}^{i}SIM_l[k]; l \in Z_\phi, k \in Z_B, \tag{22} $$

and then performing the*memory bank output* (${}^{i}_{b}M_{BO}$) operation. We form this matrix by interleaving the even and odd array addressing schemes. That is,

$$ {}^{i}SIM = \underbrace{\begin{bmatrix} X_i & Y_i & X_i & \dots & X_i & Y_i \end{bmatrix}}_{\text{B}}, \tag{23} $$

where X_i and Y_i are, again, the even and odd array schemes vectors respectively. Furthermore, we use the notation ${}^{i}SIM_k[l]$ to refer the l-th element in k-th row in the i-th stage. Let E_ϕ be an ordered basis of ϕ column vectors with an entry of value one in the i-th position and zeros elsewhere. Let e_i, termed a *basis vector*, denote the i-th element of order ϕ, of this basis, where i ranges from 0 to $\phi - 1$. That is,

$$ E_\phi = \{e_0, e_1, ..., e_{\phi-1}\}. \tag{24} $$

The result of the inner product stage operation is the memory bank output (${}^{b}_{i}M_{BO}$), and is obtained by making

$$ {}^{b}_{i}S_{RI} = {}^{b}_{i}M_{BO} = \begin{bmatrix} \langle {}^{b}_{i}M_0, e_{{}^{i}SIM_b[0]} \rangle \\ \langle {}^{b}_{i}M_1, e_{{}^{i}SIM_b[1]} \rangle \\ \vdots \\ \langle {}^{b}_{i}M_k, e_{{}^{i}SIM_b[k]} \rangle \\ \vdots \\ \langle {}^{b}_{i}M_{B-1}, e_{{}^{i}SIM_b[B-1]} \rangle \end{bmatrix}, \tag{25} $$

where ${}^{b}_{i}S_{RI}$ is the DSR input bus, $e_{{}^{i}SIM_b[k]} \in E_\phi$ is a basis vector, and

$$ {}^{b}_{i}M_k[{}^{i}SIM_b[k]] = \langle {}^{b}_{i}M_k, e_{{}^{i}SIM_b[k]} \rangle = \sum_{n \in Z_\phi} {}^{b}_{i}M_k[n](e_{{}^{i}SIM_b[k]})[n]. \tag{26} $$

The numeric values fetched throughout these addressing schemes are the ones to be operated by the PEs at each stage, but they are not in the same order according to the butterflies arrangement.

The DSR permutates these numeric values. In the first stage, the DSR performs a bit reversal permutation over the input data. In the rest of the stages, the DSR performs stride-2 permutations in the first half of the stage, and modified stride-2 permutations in the second. That is,

$$ \begin{aligned} {}^{b}_{i}P_{EI} = {}^{b}_{i}S_{RO} = R_B {}^{b}_{i}S_{RI}; & \quad i = 0, 0 \leq b \leq \phi - 1, \\ {}^{b}_{i}P_{EI} = {}^{b}_{i}S_{RO} = L^{B}_{2} {}^{b}_{i}S_{RI}; & \quad i > 0, 0 \leq b \leq \phi/2 - 1, \\ {}^{b}_{i}P_{EI} = {}^{b}_{i}S_{RO} = (J_2 \otimes I_{\phi/2}) L^{B}_{2} {}^{b}_{i}S_{RI}; & \quad i > 0, \phi/2 \leq b \leq \phi - 1, \end{aligned} \tag{27} $$

where S_{RO} is the DSR output bus, and P_{EI} is the PEs input bus. The next action is to process the DSR output through the butterflies. The PEs are an arrangement of radix-2 butterflies represented as F_2 and is expressed as:

$$ {}_i^b S_{WI} = {}_i^b P_{EO} = \begin{bmatrix} (U_2 W_2^p \odot F_2) \begin{bmatrix} {}_i^b P_{EI_0} \\ {}_i^b P_{EI_1} \end{bmatrix} \\ (U_2 W_2^p \odot F_2) \begin{bmatrix} {}_i^b P_{EI_2} \\ {}_i^b P_{EI_3} \end{bmatrix} \\ \vdots \\ (U_2 W_2^p \odot F_2) \begin{bmatrix} {}_i^b P_{EI_k} \\ {}_i^b P_{EI_{k+1}} \end{bmatrix} \\ \vdots \\ (U_2 W_2^p \odot F_2) \begin{bmatrix} {}_i^b P_{EI_{B-2}} \\ {}_i^b P_{EI_{B-1}} \end{bmatrix} \end{bmatrix}, \tag{28} $$

where ${}_i^b S_{WI}$ is the DSW input bus, U_2 is a unit matrix defined as

$$ U_2 = \begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}, \tag{29} $$

and,

$$ W_2^p = \begin{bmatrix} 1 & 0 \\ 0 & \omega_N \end{bmatrix}^p, \tag{30} $$

where $\omega_N = \mathrm{e}^{-j\frac{2\pi}{N}}$ and,

$$ p = 2^{-(i+1)} N \left\lfloor \frac{(2^{i+1}\beta b + k)}{N} \right\rfloor. \tag{31} $$

After the data points are processed by the butterflies, the results have to be stored in the same locations in the memory banks. However, to avoid memory bank conflicts, the results have to be permutated again. The DSW operator permutates these data points. For the even-indexed values of b, the internal step counter, the DSW does not perform any permutation. On the other hand, the DSW performs the next permutation on the odd-values of b. That is,

$$ \begin{aligned} {}_i^b M_{BI} &= {}_i^b S_{WO} = {}_i^b S_{WI}; & i \geq 0, b = 2n, n = \{0, 1, \ldots, \phi/2 - 1\}, \\ {}_i^b M_{BI} &= {}_i^b S_{WO} = (I_{2\beta} \otimes J_2) {}_i^b S_{WI}; & i \geq 0, b = 2n+1, n = \{0, 1, \ldots, \phi/2 - 1\}, \end{aligned} \tag{32} $$

where ${}_i^b S_{WO}$ is the DSW output bus, and ${}_i^b M_{BI}$ is the input bus of the memory banks.

Let G_ϕ be an ordered set of ϕ row vectors with a zero in the i-th position and ones elsewhere. Let g_i, termed a *masking vector*, denote the i-th element of order ϕ, of this set, where i ranges from 0 to $\phi - 1$. That is,

$$ G_\phi = \{g_0, g_1, ..., g_{\phi-1}\}. \tag{33} $$

The next action is to write-back into the memory banks the numeric values contained in ${}_i^b S_{WO}$. This process can be done by making an addition. The operation is expressed as,

$$
{}_{i}^{b+1}M = {}_{i}^{b}H \odot {}_{i}^{b}M + \begin{bmatrix} {}_{i}^{b}M_{BI_0} \left[e_{i_{SIM_b}[0]} \right]^T \\ {}_{i}^{b}M_{BI_1} \left[e_{i_{SIM_b}[1]} \right]^T \\ \vdots \\ {}_{i}^{b}M_{BI_k} \left[e_{i_{SIM_b}[k]} \right]^T \\ \vdots \\ {}_{i}^{b}M_{BI_{B-1}} \left[e_{i_{SIM_b}[B-1]} \right]^T \end{bmatrix}, \tag{34}
$$

where the operator $\odot$ denotes Hadarmard or entrywise product, the superscript T means matrix transpose operation and,

$$
{}_{i}^{b}H = \begin{bmatrix} g_{i_{SIM_b}[0]} \\ g_{i_{SIM_b}[1]} \\ \vdots \\ g_{i_{SIM_b}[k]} \\ \vdots \\ g_{i_{SIM_b}[B-1]} \end{bmatrix}, \tag{35}
$$

where $g_{i_{SIM_b}[k]} \in G_\phi$ is a masking vector. These formulations allow us to devise computational architectures associated with each expression. The next section describes how these expressions, called *functional primitives*, can be implemented as computational hardware architectures.

3. Hardware Implementation

3.1. Data Address Generator

The hardware implementation of this block consists mainly of a standard-order counter and two blocks which modify the counter values. One block generates the addressing scheme for the even-indexed memory banks (Equations (16) and (17)). The other generates the addressing scheme for the odd-indexed memory banks (Equations (18) and (19)). The even-indexed memory banks require stride-2 permutations calculated throughout the stages making a left-circular shift over the previous addressing scheme. An example of how to generate the entire addressing scheme is shown in Figure 5. Similarly, the odd-indexed memory banks require modified stride-2 permutations which are calculated throughout the stages by making a left-circular shift and negating the least significant bit over the previous addressing scheme. Figure 6 shows an example of how to generate the entire addressing schemes Figures 5 and 6 also show the decimal (N column) and decimal bit-reversal (R column) representations, respectively. Since it is easier to calculate the decimal representation, this is the one actually implemented. Nevertheless, the addressing scheme needed is the decimal bit-reversal representation. Therefore, all the address-generator outputs are bit-reversed to produce the correct addresses.

N	Binary Representation			B
0	0	0	0	**0**
1	0	0	1	**4**
2	0	1	0	**2**
3	0	1	1	**6**
4	1	0	0	**1**
5	1	0	1	**5**
6	1	1	0	**3**
7	1	1	1	**7**
	b_2	b_1	b_0	

→

N	Binary Representation			B
0	0	0	0	**0**
2	0	1	0	**2**
4	1	0	0	**1**
6	1	1	0	**3**
1	0	0	1	**4**
3	0	1	1	**6**
5	1	0	1	**5**
7	1	1	1	**7**
	b_1	b_0	b_2	

→

N	Binary Representation			B
0	0	0	0	**0**
4	1	0	0	**1**
1	0	0	1	**4**
5	1	0	1	**5**
2	0	1	0	**2**
6	1	1	0	**3**
3	0	1	1	**6**
7	1	1	1	**7**
	b_0	b_2	b_1	

→

N	Binary Representation			B
0	0	0	0	**0**
1	0	0	1	**4**
2	0	1	0	**2**
3	0	1	1	**6**
4	1	0	0	**1**
5	1	0	1	**5**
6	1	1	0	**3**
7	1	1	1	**7**
	b_2	b_1	b_0	

Figure 5. Addressing scheme example for even-indexed memories ($N = 16$ and $\beta = 1$).

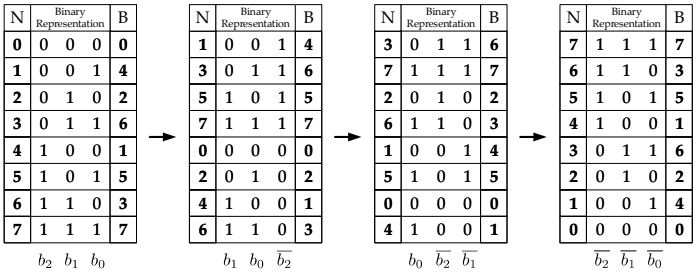


Figure 6. Addressing scheme example for odd-indexed memories ($N = 16$ and $\beta = 1$).

3.2. Phase Factor Scheduling

We took advantage of the periodicity and symmetry properties of the phase factors to reduce memory usage. Only $N/4$ numbers are stored, instead of $N/2$ as an FFT implementation normally requires [28]. The other $N/4$ values needed were obtained using,

$$\omega_N^{n+N/4} = \mathrm{Im}\{\omega_N^n\} - j\mathrm{Re}\{\omega_N^n\} \tag{36}$$

Tables 1 and 2 show the phase factor memory addresses. It is important to remark that the addressing scheme for the phase factors assumes memories of $N/2$ locations. Although we had established that the phase factor memories had $N/4$ locations. This is done due to the fact that the most significant bit of these addresses determines when Equation (36) is used. To establish an algorithm to generate the phase factors, we define three main parameters: the step (s), the frequency (f), and a parameter related with the initial value (n). The step s refers to the value to be added for obtaining the next address value, the frequency f refers to how often this step has to be added, and the initial value n is the first number in the addressing scheme. These main parameters behave as shown in Table 3. The frequency always starts at $N/(2\beta)$, decreases by halving its value in each iteration until the value is one, and remains as such until the calculation is over. The step always starts at $N/2$, and also decreases by halving its value in each iteration until the value reaches the number of butterflies β, and remains as such until the calculation is over. Finally, the parameter n, which represents the repetition and stride, at the same time, of the sequence of initial values, always stars at $N/4$, decreases by making a right shifting operation until the value is zero. Algorithm 1 generates the twiddles address pattern for an implementation of size N and β number of butterflies. In this Algorithm, T is the twiddle address, f is the frequency parameter, s is the step parameter, and n is the initial value.

Variable i represents the stage counter, b is the internal counter, and c is a counter. They are used to initialize the j-sequence per stage and to generate the addresses at each stage, respectively. Finally, R is a temporary variable used to store previous values of T.

Table 1. Phase factor addressing scheme example ($N = 32$ and $\beta = 1$).

Stage	**Twiddle Address**															
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	0	8	8	8	8	8	8	8	8
2	0	0	0	0	4	4	4	4	8	8	8	8	12	12	12	12
3	0	0	2	2	4	4	6	6	8	8	10	10	12	12	14	14
4	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15

Table 2. Phase factor addressing scheme example ($N = 32$ and $\beta = 2$).

Stage	Twiddle Address							
0	0	0	0	0	0	0	0	0
	0	0	0	0	0	0	0	0
1	0	0	0	0	8	8	8	8
	0	0	0	0	8	8	8	8
2	0	0	4	4	8	8	12	12
	0	0	4	4	8	8	12	12
3	0	2	4	6	8	10	12	14
	0	2	4	6	8	10	12	14
4	0	2	4	6	8	10	12	14
	1	3	5	7	9	11	13	15

Table 3. Main parameters behaviour of the phase factor scheduling for $N = 32$.

a $\beta = 1$

Stage	s	f	n
0	16	16	8
1	8	8	4
2	4	4	2
3	2	2	1
4	1	1	0

b $\beta = 2$

Stage	s	f	n
0	16	8	8
1	8	4	4
2	4	2	2
3	2	1	1
4	1	1	0

c $\beta = 4$

Stage	s	f	n
0	16	4	8
1	8	2	4
2	4	1	2
3	4	1	1
4	4	1	0

Algorithm 1: Twiddle Address.

Input: $f = N/(2\beta), s = N/2, n = N/2$
Output: Twiddle Pattern for N points and β butterflies

```
T = 0;
for i = 1 to log_2 N do
    R = 0;
    for c = 0 to β − 1 do
        T = R;
        if c + 1 is divisible by n and T + n < N/2 then
            R = T + n;
        end
        for b = 2 to N/(2β) do
            if b − 1 is divisible by f then
                T = T + s;
            end
        end
    end
    n = n/2;
    if f > 1 then
        f = f/2;
    end
    if s > β then
        s = s/2;
    end
end
```

3.3. Data Switch Read (DSR)

In hardware, the Data Switch Read (DSR) is implemented as an array of multiplexers, as shown in Figure 7. To perform the permutations, every multiplexer must choose the correct input. Equation (27) shows these permutations. The bit-reversal, stride-2, and modified stride-2 permutations are generated with bitwise operations over a standard-ordered sequence, such as circular shifting and simple logical operations. A bit-reversal permutation is obtained by making the *select input bus* (SEL) equal to

$$SEL = R_B \begin{bmatrix} 0 \\ 1 \\ \vdots \\ B-1 \end{bmatrix}, \tag{37}$$

where R_B is, again, a bit-reversal permutation.

A stride-2 permutation is obtained by making the SEL bus equals to

$$SEL = L_B^2 \begin{bmatrix} 0 \\ 1 \\ \vdots \\ B-1 \end{bmatrix}, \tag{38}$$

where L_B^2 is, again, a stride permutation. In hardware, this operation can be implemented by making a left circular shifting. A modified stride-2 permutation is also obtained by making the SEL bus equal to

$$SEL = (J_2 \otimes I_{B/2}) L_B^2 \begin{bmatrix} 0 \\ 1 \\ \vdots \\ B-1 \end{bmatrix}, \tag{39}$$

where J_2 and $I_{B/2}$ are, again, the exchange matrix and the identity matrix, respectively. In hardware, this operation can be implemented by making a left circular shifting only once, but negating the least significant bit.

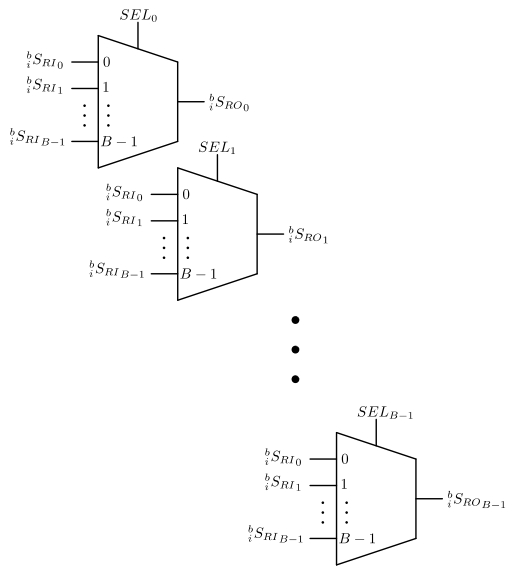


Figure 7. Hardware Implementation of the DSR.

3.4. Data Switch Write (DSW)

In hardware, the Data Switch Read (DSW) is also implemented as an array of multiplexers, as shown in Figure 8.

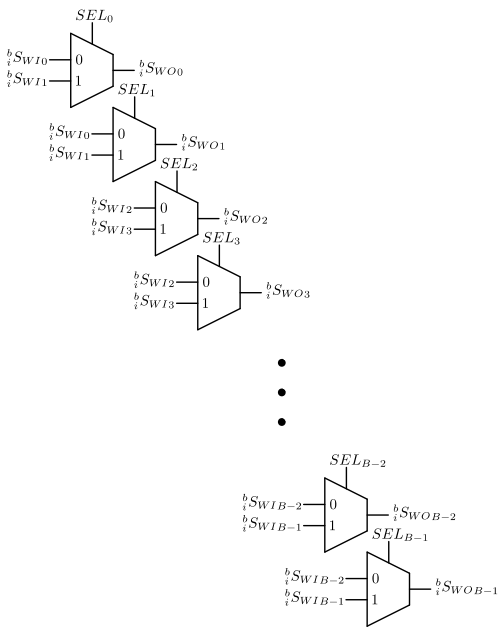


Figure 8. Hardware implementation of the DSW.

Equation (32) shows the required permutations. The first permutation outputs the input data in the same order. That is,

$$SEL = \begin{bmatrix} 0 \\ 1 \\ 0 \\ 1 \\ \vdots \\ 0 \\ 1 \end{bmatrix}. \tag{40}$$

The second permutation interchanges the even indexed input data with the next odd indexed input data. That is,

$$SEL = \begin{bmatrix} 1 \\ 0 \\ 1 \\ 0 \\ \vdots \\ 1 \\ 0 \end{bmatrix}. \tag{41}$$

3.5. Finite State Machine (FSM)

The FSM is a simple two-state Mealy machine. In first state, the machine is in idle state, waiting for an enable signal. The second state performs the address calculation, returning to the first state when $i = S - 1$ and $b = \phi - 1$. The FSM uses the values of i and b to successively generate the signals required to control:

- the output change of the DAG as indicated in Equations (17) and (19),
- the permutation to perform by the DSR and DSW as indicated in Equations (27) and (32),
- the phase factor addresses for the PEs and as indicated in Equation (28),
- the reading and writing process in the MBs as indicated in Equations (25) and (34).

4. FPGA Implementation

This section describes the procedure followed to validate the design. It shows the results obtained after completing the implementation of the proposed address generation scheme. The target hardware platform was a Xilinx VC-707 Evaluation Platform (Xilinx, Inc., San José, CA, United States). This particular board is based on a XC7VX485TFFG1761-2 FPGA (Xilinx, Inc., San José, CA, United States), which is a Virtex-7 family device. Resource consumption results were referenced for this specific chip. In order to provide a reference for comparison, a Xilinx FFT core v9.0 (Xilinx, Inc., San José, CA, United States) were included. Xilinx FFT IP core generator provides four architecture option: Pipelined Streaming I/O, Radix-4 Burst I/O, Radix-2 Burst I/O, and Radix-2 Lite Burst I/O. We concentrated on the Radix-2 architectures since they address the nature of our work. The Radix-2 Lite Burst I/O was not considered since is a simplified version of the Burst I/O version. Furthermore, the Pipelined Streaming I/O was not considered either since the architecture presented in this work optimizes the horizontal folding of any FFT computation.

It is important to emphasize that the Radix-2 Burst I/O architecture was designed to use a single butterfly performing simultaneously vertical and horizontal folding. Our architecture allows for a scalable vertical folding from $N/2$ butterflies (zero folding) to one butterfly (maximum folding). Hence, Radix-2 Burst I/O was chosen because it falls into the same category as the cores designed in this work. Our scalable vertical folding scheme may result in a time speed-up in a factor of β.

4.1. Validation

The implemented FFT with address generation scheme first went through a high level verification process. At this point, a MATLAB® program (MathWorks, Inc., Natick, MA, United States, 2014b) served to validate the correctness of the strategy by implementing the same dataflow that would be applied to the hardware version. After a hardware design was completed in VHSIC (very high speed integrated circuit) hardware description language VHDL,another program in MATLAB® was written with the purpose of generating random test data. These data were generated using the MATLAB® function *rand*, which generates uniformly distributed pseudorandom numbers. The data were then applied to the design by means of a text file. Then, the generated numbers were exported into MATLAB®, where they were converted into a suitable format for comparison with the reference FFT. After completing this procedure, we obtained the Mean Percent Error (MPE) of different implementations through the following formula:

$$MPE = 100 \times \frac{1}{N} \sum_{t=1}^{N} \frac{|a_t - h_t|}{h_t}, \tag{42}$$

where N is number of points, a_t is data generated by MATLAB®, and h_t is data generated by our FFT core in the target FPGA. This procedure was done for power of two transform sizes between 2^4 and 2^{14} points, using one-, two-, and four-butterfly versions, and using single precision floating point format. Figure 9 illustrates MPE of our core compared to the FFT calculated with MATLAB®. Figure 9 also shows that there is a minimum percent error of 1.5×10^{-5} on an FFT core of 16 points and a maximum percent error of 4.5×10^{-5} for an FFT core of 16,384 points. This information reflects an incrementing percent error while the number of points is increased. This is because MATLAB® uses double precision floating point format and rounding errors, which are produced by the large number of operations involved in the calculation. These results confirm that our scalable FFT core implemented using address generation scheme worked as expected.

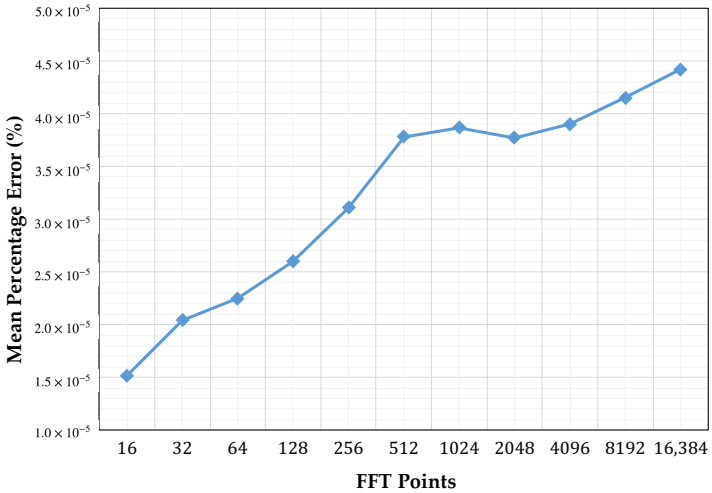


Figure 9. Mean percentage error of our core with respect to MATLAB®'s FFT function.

4.2. Timing Performance

In general, when applying the developed strategy, the number of clock cycles required to complete the operation is a well defined function of the transform size N and the number of butterflies used. This is, for a given transform of size N, with β butterflies, and a butterfly latency T_b, the cycles required to complete the calculation would be:

$$Cycles = \left(\frac{N}{2\beta} + T_b\right) \times (\log_2 N). \tag{43}$$

Figure 10 shows the computation times comparison of the core implemented with the one-, two-, and four-butterfly versions and the Xilinx version running at 100 MHz as a function of the number of points.

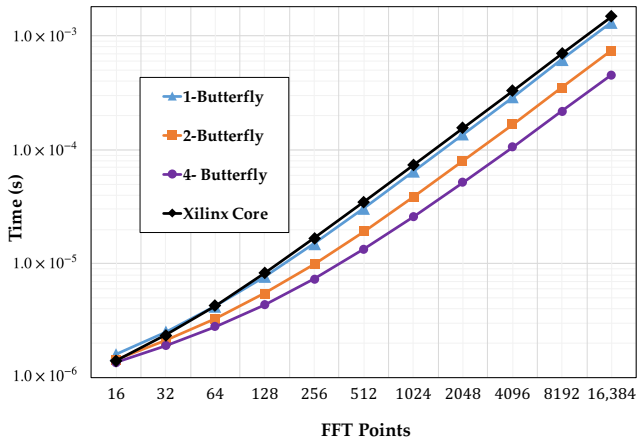


Figure 10. Computation time.

4.3. Resource Consumption

Figure 11 shows the number of slice registers consumed by the different implementations, starting with the single-butterfly to a four-butterfly version.Figure 12 shows the slice look up tables (LUTs) used. Finally, Figure 13 show the memory usage in terms of RAM blocks. The number of slices reflects the amount of logical resources spent. Flip-flops are contained within the slices and give information on how much resource is spent in sequential logic. The number of digital signal processing (DSP) blocks represents the amount of special arithmetic units from the FPGA dedicated to

implement the butterfly. The total memory reflects the resources spent to store both, data points and phase factors. Regarding the DSP blocks consumed, each butterfly in this work uses 24 blocks, 50% more than the used by the Xilinx Core. The XC7VX485TFFG1761-2 has 607,200 slice registers, 303,600 Slice LUTs, and 2800 DSP blocks.

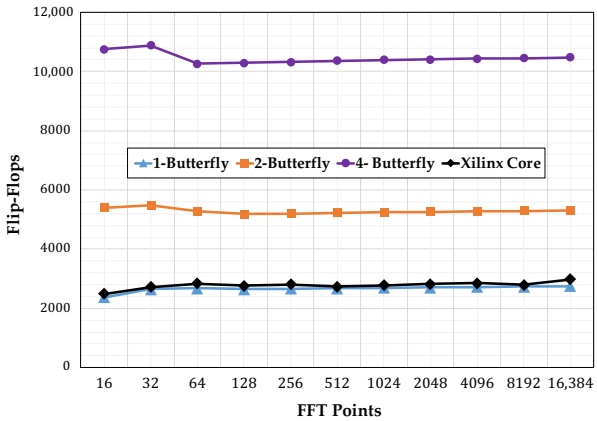


Figure 11. Flip-Flops consumption comparison.

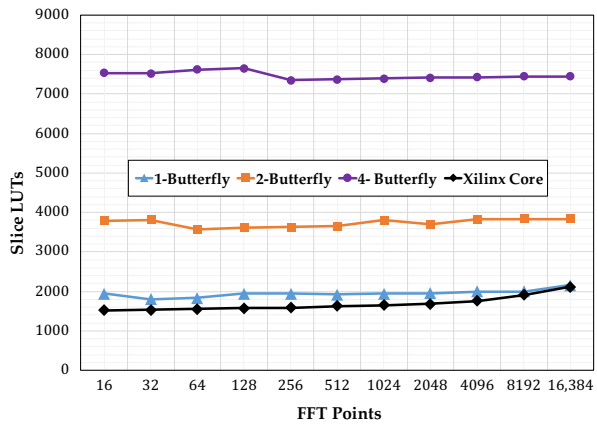


Figure 12. Slice look up tables (LUTs) consumption comparison.

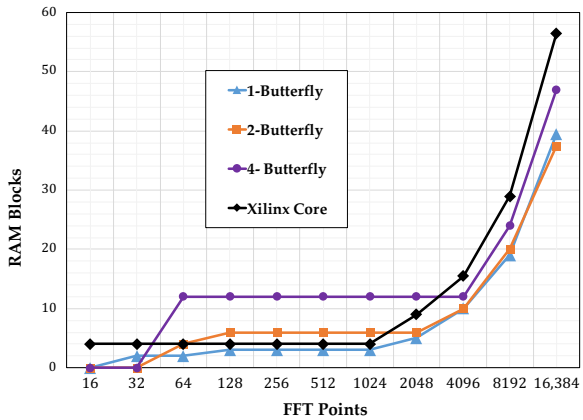


Figure 13. Memory usage comparison.

4.4. Analysis

Figure 13 reveals that the most heavily affected resource was the memory, as its usage increased in proportion to the transform size. However, using our addressing scheme did not affect the consumption

of memory resources. Figure 13 highlights this fact by showing that for increasing levels of parallelism the memory requirements were unaffected. Moreover, for a specific input size and phase factor, the data and address bus width increased proportionally to $\log_2(N)$; therefore, the system blocks managing these buses would not grow aggressively. The one-butterfly version had a smaller slice register consumption and almost the same slice count as the LUTs used by the FFT Xilinx Core. However, the two- and four-butterfly versions had a higher slice register and slice LUTs use. This was the trade-off to obtain a lower latency.

5. Conclusions

This paper proposed a method to generate scalable FFT cores based on an address generation scheme for FPGA implementation when the folding factor is scaled. The proposed method is an effective alternative to achieve a scalable folding of the Pease FFT structure. As a novel contribution, we provide a mathematical formulation to generate the address patterns and to describe data flow in FFT core computations.

Our approach is complete in the sense that it takes into consideration the address sequences required to access data points as well as twiddle or phase factors. This is accomplished by providing an algorithm and hardware to reproduce the twiddle factors for the Pease FFT radix-2 factorization regardless the number of points, folding factor, and numeric format using addressing schemes. Our implementation improves the computation time about 20% with one butterfly, 46% with two butterflies and 58% with four butterflies when compared to a reference design of the FFT Xilinx Core.

Acknowledgments: This work was supported in part by Texas Instruments Inc. through the TI-UPRM Collaborative Program. The views expressed in this manuscript are the opinions of the authors and do not represent any official position of any institution partially supporting this research.

Author Contributions: Felipe Minotta conceived, designed, and performed the experiments; Manuel Jimenez contributed to analyze the data; Domingo Rodriguez contributed with the mathematical formulations. All authors contributed to the writing of the paper.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Cooley, J.W.; Tukey, J. An Algorithm for the Machine Calculation of Complex Fourier Series. *Math. Comput.* **1965**, *19*, 297–301.
2. Pease, M.C. An Adaptation of the Fast Fourier Transform for Parallel Processing. *J. ACM* **1968**, *15*, 252–264.
3. Astola, J.; Akopian, D. Architecture-oriented regular algorithms for discrete sine and cosine transforms. *IEEE Trans. Signal Process.* **1999**, *47*, 1109–1124.
4. Chen, S.; Chen, J.; Wang, K.; Cao, W.; Wang, L. A Permutation Network for Configurable and Scalable FFT Processors. In Proceedings of the IEEE 9th International Conference on ASIC (ASICON), Xiamen, China, 25–28 October 2011; pp. 787–790.
5. Montaño, V.; Jimenez, M. Design and Implementation of a Scalable Floating-point FFT IP Core for Xilinx FPGAs. In Proceedings of the 53rd IEEE International Midwest Symposium on Circuits and Systems (MWSCAS), Seattle, WA, USA, 1–4 August 2010; pp. 533–536.
6. Yang, G.; Jung, Y. Scalable FFT Processor for MIMO-OFDM Based SDR Systems. In Proceedings of the 5th IEEE International Symposium on Wireless Pervasive Computing (ISWPC), Modena, Italy, 5–7 May 2010; pp. 517–521.
7. Johnson, L.G. Conflict Free Memory Addressing for Dedicated FFT Hardware. *IEEE Trans. Circuits Syst. II Analog Digit. Signal Process.* **1992**, *39*, 312–316.
8. Wang, B.; Zhang, Q.; Ao, T.; Huang, M. Design of Pipelined FFT Processor Based on FPGA. In Proceedings of the Second International Conference on Computer Modeling and Simulation, ICCMS '10, Hainan, China, 22–24 January 2010; Volume 4, pp. 432–435.

9. Polychronakis, N.; Reisis, D.; Tsilis, E.; Zokas, I. Conflict free, parallel memory access for radix-2 FFT processors. In Proceedings of the 19th IEEE International Conference on Electronics, Circuits and Systems (ICECS), Seville, Spain, 9–12 December 2012; pp. 973–976.
10. Huang, S.J.; Chen, S.G. A High-Throughput Radix-16 FFT Processor With Parallel and Normal Input/Output Ordering for IEEE 802.15.3c Systems. *IEEE Trans. Circuits Syst. I Regul. Papers* **2012**, *59*, 1752–1765.
11. Chen, J.; Hu, J.; Lee, S.; Sobelman, G.E. Hardware Efficient Mixed Radix-25/16/9 FFT for LTE Systems. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2015**, *23*, 221–229.
12. Garrido, M.; Sanchez, M.A.; Lopez-Vallejo, M.L.; Grajal, J. A 4096-Point Radix-4 Memory-Based FFT Using DSP Slices. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 375–379.
13. Xing, Q.J.; Ma, Z.G.; Xu, Y.K. A Novel Conflict-Free Parallel Memory Access Scheme for FFT Processors. *IEEE Trans. Circuits Syst. II Express Briefs* **2017**, *64*, 1347–1351.
14. Xia, K.F.; Wu, B.; Xiong, T.; Ye, T.C. A Memory-Based FFT Processor Design With Generalized Efficient Conflict-Free Address Schemes. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2017**, *25*, 1919–1929.
15. Gautam, V.; Ray, K.; Haddow, P. Hardware efficient design of Variable Length FFT Processor. In Proceedings of the 2011 IEEE 14th International Symposium on Design and Diagnostics of Electronic Circuits Systems (DDECS), Cottbus, Germany, 13–15 April 2011; pp. 309–312.
16. Tsai, P.Y.; Lin, C.Y. A Generalized Conflict-Free Memory Addressing Scheme for Continuous-Flow Parallel-Processing FFT Processors With Rescheduling. *IEEE Trans. Large Scale Integr. (VLSI) Syst.* **2011**, *19*, 2290–2302.
17. Xiao, X.; Oruklu, E.; Saniie, J. An Efficient FFT Engine With Reduced Addressing Logic. *IEEE Trans. Circuits Syst. II Express Briefs* **2008**, *55*, 1149–1153.
18. Shome, S.; Ahesh, A.; Gupta, D.; Vadali, S. Architectural Design of a Highly Programmable Radix-2 FFT Processor with Efficient Addressing Logic. In Proceedings of the International Conference on Devices, Circuits and Systems (ICDCS), Coimbatore, India, 15–16 March 2012; pp. 516–521.
19. Ayinala, M.; Lao, Y.; Parhi, K. An In-Place FFT Architecture for Real-Valued Signals. *IEEE Trans. Circuits Syst. II Express Briefs* **2013**, *60*, 652–656.
20. Qian, Z.; Margala, M. A Novel Coefficient Address Generation Algorithm for Split-Radix FFT (Abstract Only). In Proceedings of the 2015 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays, Monterey, California, USA, 22–24 February 2015; ACM: New York, NY, USA, 2015; p. 273.
21. Yang, C.; Chen, H.; Liu, S.; Ma, S. A New Memory Address Transformation for Continuous-Flow FFT Processors with SIMD Extension. In Proceedings of CCF National Conference on Compujter Engineering and Technology, Hefei, China, 18–20 October 2015; Xu, W., Xiao, L., Li, J., Zhang, C., Eds.; Springer: Berlin/Heidelberg, Germany, 2016; pp. 51–60.
22. Milder, P.; Franchetti, F.; Hoe, J.C.; Püschel, M. Computer Generation of Hardware for Linear Digital Signal Processing Transforms. *ACM Trans. Des. Autom. Electron. Syst.* **2012**, *17*, 15.
23. Richardson, S.; Marković, D.; Danowitz, A.; Brunhaver, J.; Horowitz, M. Building Conflict-Free FFT Schedules. *IEEE Trans. Circuits Syst. I Regul. Papers* **2015**, *62*, 1146–1155.
24. Loan, C.F.V. The ubiquitous Kronecker product. *J. Comput. Appl. Math.* **2000**, *123*, 85–100. Numerical Analysis 2000. Vol. III: Linear Algebra.
25. Johnson, J.; Johnson, R.; Rodriguez, D.; Tolimieri, R. A Methodology for Designing, Modifying, and Implementing Fourier Transform Algorithms on Various Architectures. *Circuits Syst. Signal Process* **1990**, *9*, 450–500.
26. Rodriguez, D.A. On Tensor Products Formulations of Additive Fast Fourier Transform Algorithms and Their Implementations. Ph.D. Thesis, City University of New York, New York, NY, USA, 1988.
27. Xilinx, Inc. https://www.xilinx.com/support/documentation/ip_documentation/xfft/v9_0/pg109-xfft.pdf (accessed on 20 October 2017).
28. Polo, A.; Jimenez, M.; Marquez, D.; Rodriguez, D. An Address Generator Approach to the Hardware Implementation of a Scalable Pease FFT Core. In Proceedings of the IEEE 55th International Midwest Symposium on Circuits and Systems (MWSCAS), Boise, ID, USA, 5–8 August 2012; pp. 832–835.

Sample Availability: Samples of the compounds ..... are available from the authors.

© 2018 by the authors. Licensee MDPI, Basel, Switzerland. This article is an open access article distributed under the terms and conditions of the Creative Commons Attribution (CC BY) license (http://creativecommons.org/licenses/by/4.0/).