Design, Optimization, and Implementation of a Universal FFT Processor

A Thesis

Submitted to the Faculty

of

Drexel University

by

Pinit Kumhom

in partial fulfillment of the

requirement for the degree

of

Doctor of Philosophy

March 2001

Dedications

To my father...

Acknowlegements

This work was partially supported by DARPA through research grant DABT63-98-1-0004 administered by the Army Directorate of Contracting. Additional support was provided by MathStar, Inc. I would like to thank the Royal Thai Government for the scholarship I received.

I would like to thank my advisors, Dr. Prawat Nagvajara and Dr. Jeremy Johnson. Not only have they provided me with knowledge and advice, but also with patience and support. Without them, I cannot imagine myself finishing this thesis. I greatly appreciate everything that they have done for me.

I also would like to thank my friends and colleagues in our research laboratory, Mike Balog, Ryan Buchert and Sompan Chansil to name a few. I had my share of hard times during my studies here, and I would like to thank my friends from Thailand who helped me through my surgery. Without them, I don't know if I could have gotten back to work so quickly. Ken Towshend and his family provided me a second home during my stay in the U.S., and I appreciate their friendships and generosity.

Finally, I would like to thank my family in Thailand for their patience. They were in need of me for a very hard time at home, but allowed me to finish my studies.

Table of Contents

LIST OF TABLES

LIST OF FIGURES

Abstract

Design, Optimization, and Implementation of a Universal FFT Processor
Pinit Kumhom
Prawat Nagvajara Supervisor
Jeremy Johnson Co-supervisor

There exist Fast Fourier transform (FFT) algorithms, called dimensionless FFTs [1], that work independent of dimension. These algorithms can be configured to compute different dimensional discrete Fourier transforms (DFTs) simply by relabeling the input data and by changing the values of the twiddle factors occurring in the butterfly operations. This observation allows the design of a universal FFT processor, which with minor reconfiguring, can compute one, two, and three dimensional DFTs. In this thesis a family of FFT processors, parameterized by the number of points, the dimension, the number of processors, and the internal dataflow is designed. Mathematical properties of the FFT are used systematically to simplify and optimize the processor design, and to explore different algorithms and design choices. Different dimensionless FFTs have different dataflows and consequently lead to different performance characteristics. A performance model is used to evaluate the different algorithmic choices and their resulting dataflow. Using the performance model, a search was conducted to find the optimal algorithm for the family of processors considered. The resulting algorithm and corresponding hardware design was implemented using FPGA.

[1] L. Auslander, J. Johnson and R. Johnson, Dimensionless Fast Fourier Transform Method and Apparatus, Patent #US6003056, issued Dec. 14, 1999.

1.0 INTRODUCTION

There are many computation devices available that can execute Digital Signal Processing (DSP) algorithms [1]. The spectrum of such devices ranges from a general-purpose processor to special-purpose hardware designed to compute a specific algorithm. These two choices represent the extremes in tradeoff between performance and design cost. Special-purpose hardware provides higher performance, but lacks the flexibility provided by a general purpose processor. The more flexible solution, provided by a general purpose processor, can be amortized over multiple applications whereas less flexible special-purpose hardware generally must be redesigned for each application. There are a variety of compromises that attempt to balance performance and flexibility and hence design cost. From the general-purpose processor end, there exists a class of processors designed to boost the performance of DSP algorithm while providing the flexibility of a general purpose computer. From the special-purpose hardware end, flexibility is provided using parameterized designs known as Intellectual Property (IP) cores [2]. These cores provide designers with a range of design choices during the design process. Designers can customize their designs in term of parameters such as data word length, size of problem and choices of arithmetic units. However, once the parameters are set, they can not be changed during run-time. For many DSP algorithms such as the fast Fourier transform (FFT), some run-time flexibility can be provided through the use of run-time parameters. This thesis demonstrates a design methodology for special-purpose hardware that extends the flexibility not only during design process but during run-time as well. The basic idea is to restrict the design space to a limited class of well-structured algorithms and then to use properties of the family of algorithms to systematically explore design choices and to provide a rich set of run-time parameters that can be used to provide maximal coverage during execution. The proposed methodology is illustrated and carried out with the design of a family of special-purpose processors for computing multi-dimensional discrete Fourier transforms (DFTs).

Many DSP applications make heave use of the FFT, which can be a computationally intensive task due to the large amounts of data that must be processed. In some applications, FFTs of different size and dimension are required. To this end, engineers and scientists rely on approaches such as highly-tuned code for uniprocessors [3], DSP processors [4–11], ASIC [12], IP cores [13], and reconfigurable architecture [14], to meet the performance requirements with respect to other design constraints such as physical space and power limitations.

The study in this thesis, which is part of the SPIRAL project [15], focuses on using mathematical properties of the FFT in the design and implementation of a parameterized high-performance FFT processor. We design and implement a universal FFT engine that is parameterized by the number of points and dimension of the transform during run-time, and by the choice of algorithm during the design process. The proposed design and design process uses a class of FFT algorithm called "dimensionless FFTs" [1] [16] and a family of distributed memory parallel architectures.

The architecture is comprised of multiple processing units and memory units connected via an interconnection network. The data is distributed over the memory modules while the tasks are deterministically distributed to processing units. The deterministic distribution, which is done during the design process, allows us to have localized control for each processing unit. The use of localized control provides scalability in terms of the number of processing and memory units. This deterministic,

[1]L. Auslander, J. Johnson and R. Johnson, Dimensionless Fast Fourier Transform Method and Apparatus, Patent #US6003056, issued Dec. 14, 1999.

decentralized control unit is obtained using mathematical properties of the dimensionless FFT algorithms under consideration.

A dimensionless FFT is an algorithm which, with very minor changes, can compute any multi-dimensional DFT on a fixed number of points independent of dimension. These algorithms can be configured to compute different dimensional DFTs simply by relabeling the input data and by changing the values of the twiddle factors occurring in the butterfly operations[2]. This observation is what allows the design of a universal FFT processor, that can be parameterized to compute one, two, and three dimensional DFTs.

A dimensionless FFT algorithm can be derived as a matrix factorization, which can be concisely described by a mathematical formula consisting of structured matrices built using the tensor product (Kronecker product) of a two-point DFT matrices, permutations and diagonal matrices called twiddle factor matrices. The permutation matrices in the formula precisely define the dataflow of the algorithm. The twiddle factor matrices provide the correct coefficients needed to compute different dimensional DFTs. An algorithm described by such a formula can be mapped to our processor. This mapping is done during the design process. For each algorithm, permutation matrices produce the sequence of addresses needed for butterfly operations while the twiddle factor matrices produce the corresponding twiddle factors used in those butterfly operations. The butterfly operations are distributed (mapped) to the processors which include an “address generator (AG)” and a “twiddle factor generator (TFG)” and a control. The permutations in the matrix formula serve as parameters for the design of the address generator and the twiddle factor matrices

[2]A bufferfly operation is the basic computation in the FFT and consists of a complex multiplication of one of the inputs with a constant, called a twiddle factor, and a complex addition and subtraction.

serve as parameters used in the design of the twiddle factor generator. Since the assignment of butterfly operations is done at design time through the construction of the address and twiddle generators, the complete execution schedule is known in advance and localized control at each individual memory and processor is possible. Furthermore, it is possible to optimize memory usage at design time by considering different algorithms with their corresponding schedules at design time.

For a fixed size FFT, there exist many dimensionless FFT algorithms. Each algorithm has different performance characteristics. Because algorithms are characterized mathematically, it is possible to optimize the design systematically through the use of mathematical transformations and classification results rather than using more traditional ad-hoc approaches. Optimization is established as a well-defined search problem over the space of mathematical formulas that can represent dimensionless FFT algorithms. The search is performed using a performance model based on approaches presented in [17–20]. The technique of finding optimal algorithms through searching is similar to the techniques used by FFTW [3], ATLAS [21] and the SPIRAL project [15]. However, we apply the technique to hardware design and use a high-level performance model to evaluate the cost of different algorithms rather than execution time.

Since the performance cost of an FFT algorithm on a distributed memory architecture is dominated by the cost of memory-access [22], the performance model emphasizes the cost of memory access and contention of the interconnect. The performance model simulates the memory access patterns and the behavior of the interconnect in our processor while introducing parameterized delays for the necessary computation. The performance model is implemented in VHDL using the ADEPT tool [19]. In order to find an optimal algorithm, we used the performance model to simulate the execution of many different dimensionless FFT algorithms parameterized by the permutations occurring in the corresponding matrix factorizations. A search was performed, using the cost returned by the simulation, for the algorithm with minimal cost. A simulation was performed for different input sizes and using different numbers of processing elements in our processor family. The algorithms found by the search exhibited some interesting patterns and properties. A generalization of the formulas discovered was used as the basis of our implementation. We conjecture that this class of algorithms is optimal in general for our processor model. It is interesting to note that while the optimal algorithm found maximizes locality of memory access, as would be expected, it also incorporates a schedule of tasks that seems to minimize contention for the interconnect. Algorithms with the same memory locality can be differentiated using our performance model which captures, in addition to the different memory access times between global and local access, memory contention.

It is also worth pointing out that similar locality could have been obtained by using a higher radix FFT algorithm; however, we obtain the same locality properties using a radix two approach. This is important since it simplifies the twiddle factor computations and easily supports the dimensionless FFT. Standard higher radix approaches would have to be modified to obtain the additional properties that reduce contention.

After selecting what is believed to be an optimal design, the final step in our methodology is to implement the selected design using the proposed architecture. This involves completing the detailed design necessary to compute the appropriate addresses and twiddle factors for the scheduled butterfly operations. In addition, control must be added for the processing elements, computation units, memory units, and interconnect. Here too, the mathematics of the FFT can be used to speed up and simplify the design process. In fact, it should be possible, using the ideas in this thesis, to automate this process.

The implementation process consists of the following steps. First, the necessary arithmetic units such as floating point adders and multipliers are designed. These units can be varied and optimized. For example, one may choose to implement a fixed-point arithmetic instead of floating point. The advantages/disadvantages of these choices are beyond the scope of this thesis. For our prototype, single precision floating point arithmetic conforming to the ANSI/IEEE 754 standard was used. The floating point units, an adder and multiplier, were implemented using a pipelined design. This part of the process follows standard design techniques and can utilize existing IP cores. Second, using the arithmetic units, we assemble a computation unit that performs butterfly operations and computes the necessary twiddle factors. The resulting computation unit was designed to be independent of the FFT algorithm selected. Algorithmic specific information is passed as control information to the computation unit. Specifically, for each butterfly operation, it receives three inputs namely a fraction called "twiddle fraction" that is used to generate a twiddle factor and two inputs data used for computing the butterfly operation. These inputs are provided to it by the twiddle factor generator and the address generator respectively. The address generator and the twiddle fraction generators are the two units that depend on the algorithm.

Hence, the final step of the implementation process is the design of the twiddle factor and address generator. It is this part of the design process that is aided by our mathematical description of the FFT. Two implementation techniques, based on MUXs and adders respectively, are investigated. Both implementations are derived directly from the permutation and twiddle factor matrices in the description of the dimensionless FFT algorithm. In addition to general techniques that apply to every dimensionless FFT, special properties of the optimal algorithm are incorporated into the design. These additional properties further simply the resulting implementation. Specifically the logic for the address and twiddle generators using either implementation approach is greatly reduced.

Not only does the optimal algorithm provides a simple implementation of address and twiddle factor generators, it also provides a simple communication pattern. The simplified communication pattern allows the interconnection network to be optimized. For the optimal algorithm in a 2^m-processor system, there are only m stages that need the interconnection. Moreover, the interconnection is used in a pair wise fashion with the pair of communicating processors fixed for each stage. Therefore, only m network configurations are needed and each processor needs to be able to connect to $m-1$ other processors as opposed to 2^m processors in case of general interconnection network. Since this information is known at compile time the interconnection network does not have to support general communication patterns and consequently can be greatly simplified and can provide higher performance.

For proof of concept, the design was implemented on the Wildforce^{TM} [23] reconfigurable (FPGA) board. The design was implemented using synthesizable VHDL, and was verified in two stages. First a VHDL simulation was used to test the logic of the design and then the actual configured board was validated. The validation process completely tested the implementation by applying the FFT computation for a set of fixed sizes to a basis and comparing the results to those specified by the DFT matrix. After validating the implementation we benchmarked the processor to study the efficiency and scalability of the implementation. A future implementation may use ASIC technology for the floating-point (complex numbers) arithmetical cores and the FPGA technology for the parameterized flow control units. Figure 1.1 summarizes the purposed design methodology.

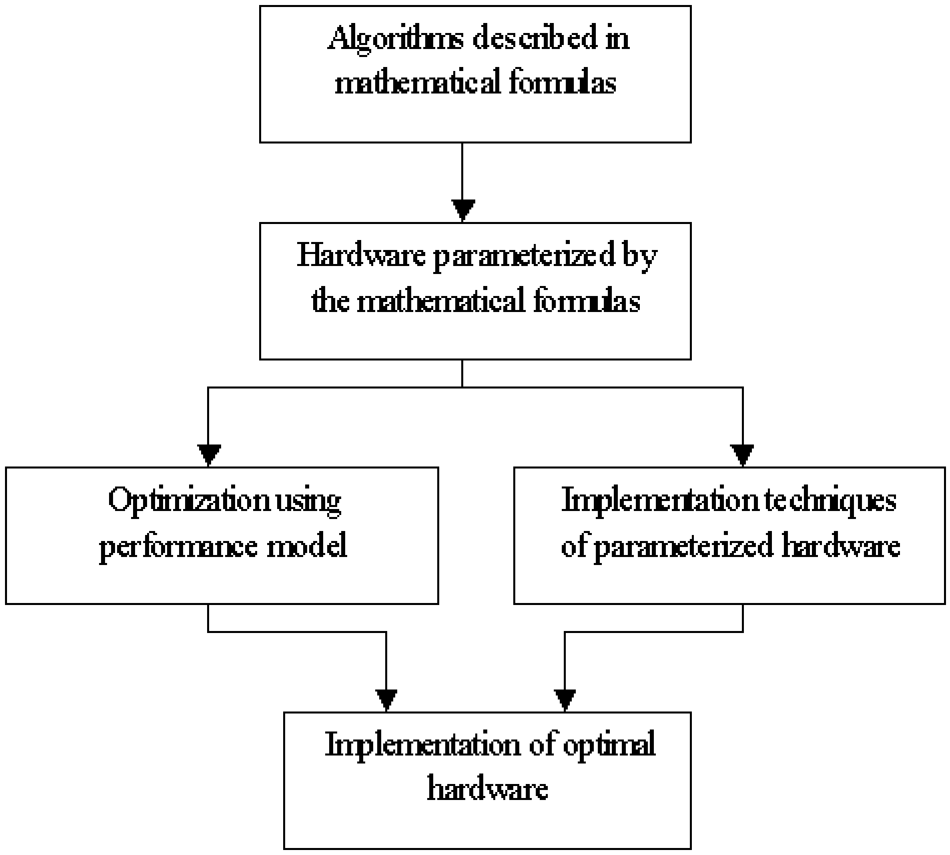

Algorithms described in mathematical formulas
Hardware parameterized by the mathematical formulas
Optimization using performance model
Implementation techniques of parameterized hardware
Implementation of optimal hardware

Figure 1.1 The proposed design methodology

The resulting design and implementation has some similarities to existing FFT processors [24, 25]; however, we believe that the following properties make it unique: (1) it implements the dimensionless FFT, (2) it uses completely localized control for address and twiddle factor generation, (3) it utilizes a scalable distributed memory architecture optimized to the FFT with an interconnect and schedule designed to minimize contention. Since we have optimized the design to the FFT, the resulting processor is quite different than other more general distributed memory architectures. More important than the particular design is the process in which it was obtained.

In conclusion, the novelty of this thesis is threefold. First the FFT processor is based on the dimensionless FFT [16] which allows a single hardware design to compute one, two, and three dimensional DFTS. Second, a framework for systematically mapping alternative FFT algorithms onto parameterized scalable hardware is provided. This is obtained by mapping a mathematical description of the FFT, based on matrix factorizations [26], to hardware that implements flow control and generation of the necessary roots of unity (twiddle factors). By incorporating information about the algorithm in the design stage many simplifications in the resulting hardware were obtained; particularly the support needed for task allocation and communication. Third, the optimization of the design was performed using a systematic search. There are many different FFT algorithms, each with different dataflow, and consequently different performance characteristics. Thus our hardware design becomes an optimization problem over the space of possible FFT dataflows [22].

The remainder of the thesis is arranged as follows. Chapter 2 provides the necessary mathematical background. In particular the dimensionless FFT is reviewed and the space of possible dimensionless FFT algorithms is described. Chapter 3 introduces the architectural framework and and the methodology for mapping FFT algorithms to the architecture. Chapter 4 discusses the performance model and shows how the performance model was used to optimize the design of the FFT processor. Chapter 5 presents an implementation of the optimal design using the Wildforce^{TM} FPGA board. A detailed discussion of the implementation and design of the address and twiddle factor generators is presented. Chapter 6 discusses the verifcation and the performance of the resulting implementation. Finally, conclusions and suggestions for future research are available in Chapter 7. To make concrete all of the ideas in this thesis we provide a detailed example, in the appendix A, using a 64-point FFT computation with four processors.

2.0 MATHEMATICAL FORMULATION OF THE FFT

One of the main themes of this thesis is that domain-specific knowledge should be utilized in the design and implementation of special-purpose hardware. In the design of an FFT processor the extensive knowledge available about the FFT should be incorporated. This knowledge is most conveniently expressed using a mathematical description of the FFT. In this chapter we review the mathematics needed to analyze and fully understand the FFT. Special emphasis is provided to material utilized in the design and optimization of our processor.

The discrete Fourier transform (DFT) is conveniently expressed as a matrix-vector product, and fast algorithms for computing the DFT are obtained from factorizations of the DFT matrix into a product of structured sparse matrices. The matrix formulation of the FFT has been presented and promoted in the books by Tolimier et al. [27] and Van Loan [26]. Some of the material and notation in our presentation is derived from the paper [28] which not only emphasizes the role of matrix factorizations in deriving and describing FFT algorithms, but shows how the mathematics can be used in implementing FFT algorithms. In this chapter we review the relevent material on the matrix formulation of the FFT, present the dimensionless approach [16] to multidimensional FFTs, and discuss the space of FFT algorithms that are considered for the design of our processor.

The chapter is organized as follows. Section 2.1 presents the basic mathematical properties of the tensor product, permutation matrices, and a family of diagonal matrices called twiddle factor matrices. Section 2.2 presents a class of FFT algorithms using matrix factorizations, and interprets different factorizations in terms of dataflow. Section 2.3 describes the multidimensional Discrete Fourier Transform, and the dimensionless FFT algorithm is presented in Section 2.4. Finally the concept of an FFT dataflow is introduced in Section 2.5. The analysis of the different dataflows that can occur in an FFT algorithm is crucial to the optimization of our processor.

2.1 Mathematical Background

As discussed in the introduction to this chapter, the FFT can be described as a factorization of the DFT matrix into a product of structured sparse matrices. In this section we review the necessary concepts from matrix algebra and introduce the matrices that occur in the FFT. This includes a discussion of the tensor (or Kronecker) product, the direct sum, permutation matrices, and twiddle factor matrices. We also relate the tensor product to the indexing operations occuring in FFT algorithms.

2.1.1 Tensor Product

The tensor product provides a tool for describing and manipulating a class of block structured matrices.

Definition 1 (Tensor Product). *Let A and B be $p \times q$ and $r \times s$ matrices respectively. Then, the tensor product of A and B, denoted by $A \otimes B$, is the $pr \times qs$ matrix*

$$(A \otimes B) \quad = \quad \begin{pmatrix} a_{0,0}B & a_{1,1}B & \dots & a_{0,q-1}B \\ a_{1,0}B & a_{1,1}B & \dots & a_{2,q-1}B \\ \vdots & \vdots & \ddots & \vdots \\ a_{p-1,0}B & a_{p-1,1}B & \dots & a_{p-1,q-1}B \end{pmatrix}, \tag{2-1}$$

where $a_{i,j}$, $0 \leq i < p$ and $0 \leq j < q$, is the element at the $(i+1)^{st}$ row and the $(j+1)^{st}$ column of matrix A.

The tensor product satisfies many basic properties used for factoring block structured matrices like the DFT. The following list is provided as a reference.

Property 1 (Tensor Product Properties). *The tensor product satisfies the following basic properties, where I_p is the $p \times p$ identity matrix, indicated inverses exist, and matrix dimensions are such that all products make sense.*

1.1 $(\alpha A) \otimes B = A \otimes (\alpha B) = \alpha(A \otimes B)$

1.2 $(A + B) \otimes C = (A \otimes C) + (B \otimes C)$

1.3 $A \otimes (B + C) = (A \otimes B) + (A \otimes C)$

1.4 $1 \otimes A = A \otimes 1 = A$

1.5 $A \otimes (B \otimes C) = (A \otimes B) \otimes C$

1.6 $(A \otimes B)^t = A^t \otimes B^t$

1.7 $(AB \otimes CD) = (A \otimes C)(B \otimes D)$

1.8 $(A \otimes B) = (I_p \otimes B)(A \otimes I_q) = (A \otimes I_q)(I_p \otimes B)$

1.9 $(A_1 \otimes \cdots A_t)(B_1 \otimes \cdots \otimes B_t) = (A_1B_1 \otimes \cdots \otimes A_tB_t)$

1.10 $(A_1 \otimes B_1) \cdots (A_t \otimes B_t) = A_1 \cdots A_t \otimes B_1 \cdots B_t$

1.11 $(I_p \otimes B_1 \cdots B_t) = (I_p \otimes B_1) \cdots (I_p \otimes B_t)$

1.12 $(A \otimes B)^{-1} = A^{-1} \otimes B^{-1}$

1.13 $I_p \otimes I_q = I_{pq}$

All of these identities follow from the definition or simple applications of preceding properties (see [29]).

Another matrix operation used in this thesis is the direct sum.

Definition 2 (Direct Sum). *Let A and B be two matrices of arbitrary sizes, the direct sum of A and B, denoted by $A \oplus B$, is*

$$A \oplus B = \left(\begin{array}{c|c} A & \mathbf{0} \\ \hline \mathbf{0} & B \end{array}\right). \tag{2-2}$$

A tensor product of the form $I_p \otimes A$ can be written as a direct sum of p copies of A. This can be interpreted as a parallel operation when $I_p \otimes A$ is applied to a vector.

Property 2 (Parallel Form of the Tensor Product).

$$I_p \otimes A = \left(\begin{array}{c|c|c} A & & \\ \hline & \ddots & \\ \hline & & A \end{array}\right) = \bigoplus_{i=1}^{p} A \tag{2-3}$$

The computation of $(I_p \otimes A)x$ can be obtained by dividing the input, x into p segments and applying A independently to each segment.

In general the tensor product is not commutative (i.e. $A \otimes B \neq B \otimes A$). In particular, $A \otimes I$ is naturally interpreted as a vector operation rather than a parallel operation.

Property 3 (Vector Form of the Tensor Product). *Let A be an $p \times p$ matrix. Then*

$$A \otimes I_q = \begin{pmatrix} a_{0,0}I_q & \dots & a_{0,p-1}I_q \\ \vdots & \ddots & \vdots \\ a_{p-1,0}I_q & \dots & a_{p-1,p-1}I_q \end{pmatrix} \tag{2-4}$$

The computation of $y = (A \otimes I_q)x$ can be performed using vector operations. Let x_i be the i-th segment of the input vector x and y_j be the j-segment of the output vector y. Then

$$y_j = \sum_{j=0}^{p-1} a_{ji}x_i,$$

where the sum consists of scalar-vector products and vector additions.

Using Property 1.8 an arbitrary tensor product can be factored into a parallel and vector operation. Computing $y = (A \otimes B)x$ as $y = (A \otimes I_q)(I_p \otimes B)x$ corresponds to the row-column algorithm commonly used for computing two-dimensional DFTs. To see this, we remark that if A is a $p \times p$ matrix, B is a $q \times q$ matrix, and X and Y are the $p \times q$ matrices obtained by placing consecutive elements of the vectors x and y in the rows of X and Y respectively, then

$$Y = AXB^t. \tag{2-5}$$

Corresponding to the factorization $y = (A \otimes I_q)(I_p \otimes B)x$, Y is computed by first appling B to the rows of X and then applying A to the columns of the intermediate result.

2.1.2 Indexing and Basis Vectors

A key component of an FFT program or an FFT processor is the calculation of addresses of data elements. Address computation can be expressed using mixed-radix numbers. Mixed-radix numbers are related to the tensor product and consequently naturally arise in the FFT. In this section we review mixed-radix numbers and relate them to the tensor product.

Definition 3 (Mixed-Radix Number System). *Let* $N = N_{t-1} \times \cdots \times N_0$ *and* $0 \leq i_j < N_j$, $0 \leq j < t$. *Then, the number whose mixed-radix representation is* $(i_{t-1}, \ldots, i_0)$ *is equal to*

$$(N_{t-2} \cdots N_0)i_{t-1} + (N_{t-3} \cdots N_0)i_{t-2} + \cdots + N_0 i_1 + i_0.$$

Arranging the mixed-radix number $(i_{t-1}, \ldots, i_0)$ in lexicographical order is equivalent to counting from 0 to $N - 1$.

Example: Let $N = 3 \times 2 \times 3 = 18$ and $0 \leq i_1 < 3$, $0 \leq i_2 < 2$,$0 \leq i_3 < 3$. Then, (i_2, i_1, i_0) denotes the mixed-radix representation of the system, where

$$i = (i_2, i_1, i_0) = (3 \cdot 2)i_2 + 3i_1 + i_0.$$

For instance, $(1, 0, 1) = (2 \cdot 3) \cdot 1 + 3 \cdot 0 + 1 = 7$.

Counting (i_2, i_1, i_0) from $(0, 0, 0)$ to $(2, 1, 2)$ in lexicographical order results that i is counted from 0 to 17; that is

$$\begin{array}{ccccccc} i & : & 0 & 1 & 2 & 3 & \cdots & 17 \\ (i_2, i_1, i_0) & : & (0,0,0) & (0,0,1) & (0,0,2) & (0,1,0) & \cdots & (2,1,2) \end{array}$$

□

When $N = r^t$, mixed-radix numbers become radix-r numbers. For convenience, we will use the following notation for the radix-r number system.

Let b_j be the j^{th} digit of t-digit radix-r number, denoted by $(b_{t-1}, \cdots, b_0)_r$. Then,

$$(b_{t-1}, \cdots, b_0)_r = \sum_{j=0}^{t-1} b_j r^j; \quad b_j \in \{0, 1, \cdots, r-1\}$$

For the binary number system ($r = 2$), where $b_j \in \{0, 1\}$, we drop the commas and the notation becomes $(b_{t-1} \cdots b_0)_2$

When dealing with matrix factorizations, indexing operations are obtained using standard basis elements. The set of inputs to an FFT algorithm of size N are the set of N-tuples of complex numbers (other domains are possible, but in this thesis we will restrict the FFT to complex data), which forms a vector space denoted by $\mathbb{C}^N$ (for a review of vector spaces and linear algebra see [30]). The elements of $\mathbb{C}^N$ can be uniquely written as a linear combination of N basis vectors. For our purposes it is convenient to use the standard basis.

Definition 4 (Standard Basis). *The standard basis for the vector space of N-tuple of the complex number, $\mathbb{C}^N$, is*

$$\left\{ \mathbf{e}_i^N \mid 0 \leq i < N \right\}, \tag{2-6}$$

where $\mathbf{e}_i^N$ is the vector with 1 in the i^{th} component and zeros elsewhere.

Let $\mathbf{x} \in \mathbb{C}^N$. Then, we can write $\mathbf{x}$ uniquely as the linear combination of the standard basis as follows.

$$\mathbf{x} = (x_0, \ldots, x_{N-1})^t = \sum_{i=0}^{N-1} x_i \mathbf{e}_i^N \tag{2-7}$$

Using the standard basis, it is easy to go from the abstract notion of a linear operator to the concrete notion of a matrix.

Property 4 (Linear Operator). *A linear operator A on $\mathbb{C}^N$ is a mapping from from $\mathbb{C}^N$ to $\mathbb{C}^N$ which satisfies the following properties.*

4.1. $A(\alpha \mathbf{x}) = \alpha(A\mathbf{x})$

4.2. $A(\alpha \mathbf{x} + \beta \mathbf{y}) = A(\alpha \mathbf{x}) + B(\beta \mathbf{y}) = \alpha(A\mathbf{x}) + \beta(A\mathbf{y})$

It is easy to see that any matrix is linear, in particular the DFT is linear. The computation of a linear operator applied to an arbitrary vector is known once it is known what it does to a basis.

$$A\mathbf{x} = A(\sum_{i=0}^{N-1} x_i \mathbf{e}_i^N) = \sum_{i=0}^{N-1} x_i A e_i^N \tag{2-8}$$

Given a linear operator, A, on $\mathbb{C}^N$ it is easy to obtain the corresponding matrix.

Property 5 (Matrices and the Standard Basis). *Let A be an $N \times N$ matrix. Then $A\mathbf{e}_i^N$ is the i^{th} column of A.*

If A is given abstractly as a linear operator (think of a black box program which when given an input $\mathbf{x}$ returns the output vector $\mathbf{y}$ and satisfies the linearity properties) then the matrix corresponding to A is obtained by appling the operator to each of the elements in the standard basis. Each computation returns a column of the matrix.

When a matrix is equal to a tensor product of matrices it is easier to compute with tensor product of basis elements. The following property describes the tensor product of basis vectors. This property relates the tensor product to mixed-radix numbers.

Property 6 (Tensor Product of Standard Basis Elements).

$$\mathbf{e}_i^p \otimes \mathbf{e}_j^q = \mathbf{e}_{qi+j}^{pq}; \quad 0 \leq i < p, 0 \leq j < q \tag{2-9}$$

More generally,

$$\mathbf{e}_{i_{t-1}}^{N_{t-1}} \otimes \cdots \otimes \mathbf{e}_{i_0}^{N_0} = \mathbf{e}_{(i_{t-1},\ldots,i_0)}^{N}, \tag{2-10}$$

where $(i_{t-1}, \ldots, i_0)$ is the mixed-radix number defined in Definition 3.

This property illustrates that the collection of vectors $\mathbf{e}_{i_{t-1}}^{N_{t-1}} \otimes \cdots \otimes \mathbf{e}_{i_0}^{N_0}$ with $0 \leq i_j < N_j$ for $j = 0, \ldots, t-1$ forms the standard basis for $\mathbb{C}^N$.

Example: Let $\mathbf{x} = (x_0, x_1)^t$ and $\mathbf{y} = (y_0, y_1, y_2)^t$. Then,

$$\begin{aligned}
\mathbf{x} &= \sum_{i=0}^{1} x_i \mathbf{e}_i^2, \quad \mathbf{y} = \sum_{j=0}^{2} y_j \mathbf{e}_j^3 \\
\mathbf{x} \otimes \mathbf{y} &= \sum_{i=0}^{1} x_i \mathbf{e}_i^2 \otimes \sum_{j=0}^{2} y_j \mathbf{e}_j^3 \\
&= \sum_{i=0}^{1} \sum_{j=0}^{2} x_i y_i \mathbf{e}_i^2 \otimes \mathbf{e}_j^3 \\
&= \sum_{i=0}^{1} \sum_{j=0}^{2} x_i y_i \mathbf{e}_{3i+j}^6 \\
&= (x_0 y_0, x_0 y_1, x_0 y_2, x_1 y_0, x_1 y_1, x_1 y_2)^t
\end{aligned}$$

□

To see why it is convenient to use the tensor product of basis vectors with computing with tensor products, observe, using Property 1.7, that $(A \otimes B)(e_i^p \otimes e_j^q) = Ae_i^p \otimes Be_j^q$, the tensor product of the i-th column of A and the j-th column of B.

2.1.3 Permutation Matrices

An essential component of FFT factorizations are permutation matrices. These matrices provide addressing and dataflow information. In this section, we collect the definitions and properties involving the permutation matrices.

First, we introduce a class of functions called "permutation" defined as follows.

Definition 5 (Permutation). *A permutation, σ, of degree N is a one-to-one mapping from $\{0, \cdots, N-1\}$ to $\{0, \cdots, N-1\}$. We denote σ by*

$$\sigma = \begin{pmatrix} 0 & 1 & \dots & N-1 \\ \sigma(0) & \sigma(1) & \dots & \sigma(N-1) \end{pmatrix} \tag{2-11}$$

which mean the function maps $i \to \sigma(i)$, $0 \le i < N$. For convenience, we often drop the first row and use the following notation.

$$\sigma = (\sigma(0), \sigma(1), \ldots, \sigma(N-1)) \tag{2-12}$$

Example: The permutation

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \end{pmatrix} = (1, 2, 3, 0)$$

is the mapping $0 \to 1$, $1 \to 2$, $2 \to 3$ and $3 \to 0$. $\square$

Two permutations of the same degree can be combined, by composition, to form another permutation; i.e. if σ and τ are two arbitary permutations of the same degree, then

$$\sigma\tau(i) = \sigma(\tau(i))$$

The inverse of the permutation σ is the permutation which when composed with σ is the identity permutation (i.e. the permutation that maps i to i for all i). It is easy to see that the inverse of σ is the permutation that maps $\sigma(i)$ to i.

Example: The inverse of the permutation

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 1 & 2 & 3 & 0 \end{pmatrix}$$

is the permutation

$$\sigma = \begin{pmatrix} 0 & 1 & 2 & 3 \\ 3 & 0 & 1 & 2 \end{pmatrix}$$

Composing these permutations we see that $0 \to 0$, $1 \to 1$, $2 \to 2$, and $3 \to 3$. $\square$

Associated with permutations is a class of matrices called permutation matrices, whose product, is equivalent to the composition of permutations.

Definition 6 (Permutation Matrices). *Let σ be a permutation of degree N. Then, the N-by-N permutation matrix P_σ can be defined by*

$$P_\sigma \mathbf{e}_i^N = \mathbf{e}_{\sigma(i)}^N \tag{2-13}$$

Since $P_\sigma \mathbf{e}_i^N$ is the i^th column of P_σ, we can construct P_σ from the definition.

Example: Let $\sigma = (1, 2, 3, 0)$. Then,

$$P_\sigma \mathbf{e}_0^4 = \mathbf{e}_1^4, \quad P_\sigma \mathbf{e}_1^4 = \mathbf{e}_2^4, \quad P_\sigma \mathbf{e}_2^4 = \mathbf{e}_3^4, \quad P_\sigma \mathbf{e}_3^4 = \mathbf{e}_0^4$$

and

$$\begin{aligned} P_\sigma &= \begin{pmatrix} \mathbf{e}_1^N & \mathbf{e}_2^N & \mathbf{e}_3^N & \mathbf{e}_0^N \end{pmatrix} \\ &= \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \end{aligned}$$

□

Multiplying a permutation matrix with a vector, $P_\sigma \mathbf{x}$, permutes the elements of the vector corresponding to the inverse of σ. The result of $P_\sigma \mathbf{x}$ can be viewed in two perspectives described by the "duality" property.

Property 7 (Duality). *Let x be a vector of size N and P_σ be a permutation matrix of size $N \times N$ specified by permutation function σ. Then, the permuted vector $P_\sigma x$ can be viewed in two perspectives. From the perspective of permuting the basis, we have*

$$\begin{aligned} \mathbf{x} &= \sum_{i=0}^{N-1} x_i \mathbf{e}_i^N \\ P_\sigma \mathbf{x} &= \sum_{i=0}^{N-1} x_i P_\sigma \mathbf{e}_i^N = \sum_{i=0}^{N-1} x_i \mathbf{e}_{\sigma(i)}^N \end{aligned}$$

This means that the basis $\mathbf{e}^N_{\sigma(i)}$ is moved from the $\sigma(i)$-th position to the i-th position.

Let $P_\sigma \mathbf{e}^N_i = \mathbf{e}^N_j$. Then,

$$\begin{aligned} \mathbf{e}^N_i &= P_\sigma^{-1}\mathbf{e}^N_j = P_{\sigma^{-1}}\mathbf{e}^N_j = \mathbf{e}^N_{\sigma^{-1}(j)} \\ i &= \sigma^{-1}(j) \end{aligned}$$

Therefore,

$$P_\sigma \mathbf{x} = \sum_{i=0}^{N-1} x_i P_\sigma \mathbf{e}^N_i = \sum_{i=0}^{N-1} x_i \mathbf{e}^N_j = \sum_{j=0}^{N-1} x_{\sigma^{-1}(j)}\mathbf{e}^N_j$$

This means that the element $x_{\sigma^{-1}(j)}$ is moved from the $\sigma^{-1}(j)$-th position to the j-th position.

Example: Let $\mathbf{x} = (x_0, x_1, x_2, x_3)^t$, $\sigma = (1, 2, 3, 0)$ and $\sigma^{-1} = (3, 0, 1, 2)$. Then,

$$\begin{aligned} P_\sigma \mathbf{x} &= \sum_{i=0}^{3} x_i P_\sigma \mathbf{e}^4_i = \sum_{i=0}^{3} x_i \mathbf{e}^4_{\sigma(i)} \\ &= x_0\mathbf{e}^4_1 + x_1\mathbf{e}^4_2 + x_2\mathbf{e}^4_3 + x_3\mathbf{e}^4_0 \\ &= x_3\mathbf{e}^4_0 + x_0\mathbf{e}^4_1 + x_1\mathbf{e}^4_2 + x_2\mathbf{e}^4_3 \\ &= \sum_{j=0}^{3} x_{\sigma^{-1}(j)}\mathbf{e}^4_j = (x_3, x_0, x_1, x_2)^t \end{aligned}$$

□

The following list collects the basic properties of permutation matrices.

Property 8 (Properties of Permutation Matrices).

8.1. **[Product]** $P_\sigma P_\tau = P_{\sigma\tau}$

8.2. **[Identity]** $P_{id} = I_N$, *where* $id = (0, \ldots, N-1)$

8.3. **[Inverse]** $P_{\sigma^{-1}} = (P_\sigma)^{-1} = (P_\sigma)^t$

8.4. **[Permuting Rows]** *$P_{\sigma^{-1}}A$ permutes rows of A by σ.*

8.5. **[Permuting Columns]** *AP_σ permutes columns of A by σ.*

8.6. **[Conjugation]** *$A^\sigma = P_\sigma^{-1}AP_\sigma$, where $(A^\sigma)_{i,j} = (A)_{\sigma(i),\sigma(j)}$.*

8.7. **[Direct Sum]** *$P_\sigma \oplus P_\tau$ is a permutation matrix denoted by $P_{\sigma\oplus\tau}$*

8.8. **[Tensor Product]** *Assume that σ is a permutation of degree m and τ is a permutation of degree n. $P_\sigma \otimes P_\tau$ is the permutation matrix that maps $\mathbf{e}_i^m \otimes \mathbf{e}_j^n$ to $\mathbf{e}_{\sigma(i)}^m \otimes \mathbf{e}_{\tau(j)}^n$ and is denoted by $P_{\sigma\otimes\tau}$.*

All of these identities follow from the definition or simple applications of preceding properties (see [26, 27, 29].)

The following example illustrates the product property.

Example: Let $\sigma = (1,2,3,0)$ and $\tau = (1,0,3,2)$. Then, computing $P_\sigma P_\tau$ by definition, we have

$$P_\sigma P_\tau = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 \end{pmatrix} = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

Applying the product property, we construct the permutation $P_{\sigma\tau}$ as follows.

$$\begin{aligned} P_\sigma P_\tau \mathbf{e}_0^4 &= P_{\sigma\tau}\mathbf{e}_0^4 = \mathbf{e}_{\sigma\tau(0)}^4 = \mathbf{e}_2^4, \;\; P_\sigma P_\tau \mathbf{e}_1^4 = P_{\sigma\tau}\mathbf{e}_1^4 = \mathbf{e}_{\sigma\tau(1)}^4 = \mathbf{e}_1^4 \\ P_\sigma P_\tau \mathbf{e}_2^4 &= P_{\sigma\tau}\mathbf{e}_2^4 = \mathbf{e}_{\sigma\tau(2)}^4 = \mathbf{e}_0^4, \;\; P_\sigma P_\tau \mathbf{e}_3^4 = P_{\sigma\tau}\mathbf{e}_3^4 = \mathbf{e}_{\sigma\tau(3)}^4 = \mathbf{e}_3^4 \end{aligned}$$

$$P_\sigma P_\tau = \begin{pmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}$$

□

The following example shows the inverse property.

Example: Let $\sigma = (1,2,3,0)$ and $\sigma^{-1} = (3,0,1,2)$. Then,

$$P_\sigma = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}, \quad P_{\sigma^{-1}} = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = P_\sigma^t$$

$$P_\sigma P_{\sigma^{-1}} = \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix} \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} = I_4$$

□

Using the same permutation, the following example illustrates the conjugation property.

Example: $\sigma = (1,2,3,0)$

$$A^\sigma = P_\sigma^{-1} A P_\sigma = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_{0,0} & a_{0,1} & a_{0,2} & a_{0,3} \\ a_{1,0} & a_{1,1} & a_{1,2} & a_{1,3} \\ a_{2,0} & a_{2,1} & a_{2,2} & a_{2,3} \\ a_{3,0} & a_{3,1} & a_{3,2} & a_{3,3} \end{pmatrix} \begin{pmatrix} 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \end{pmatrix}$$

$$= \begin{pmatrix} 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} a_{0,1} & a_{0,2} & a_{0,3} & a_{0,0} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,0} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \end{pmatrix} = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,0} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \\ a_{0,1} & a_{0,2} & a_{0,3} & a_{0,0} \end{pmatrix}$$

First, permuting the columns of A with σ, we have

$$A^\sigma = P_\sigma^{-1} A P_\sigma = P_{\sigma^{-1}} \begin{pmatrix} a_{0,1} & a_{0,2} & a_{0,3} & a_{0,0} \\ a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,0} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \end{pmatrix}$$

Then, permuting the rows of the resulting matrix, we have

$$A^\sigma = P_\sigma^{-1} A P_\sigma = \begin{pmatrix} a_{1,1} & a_{1,2} & a_{1,3} & a_{1,0} \\ a_{2,1} & a_{2,2} & a_{2,3} & a_{2,0} \\ a_{3,1} & a_{3,2} & a_{3,3} & a_{3,0} \\ a_{0,1} & a_{0,2} & a_{0,3} & a_{0,0} \end{pmatrix}$$

□

2.1.4 Tensor Permutations

The following definition introduces an important class of permutations related to mixed-radix numbers. These permutations are defined by permuting the digits in the mixed-radix number system. Because of the relationship of mixed-radix numbers to the tensor product and the definition below, we call these permutations **tensor permutations**.

Definition 7 (Tensor Permutation Matrix). *Let* $N = N_{t-1} \times \cdots \times N_0$ *and* σ *be a permutation function of degree* t*. Then, the* $N \times N$ *tensor permutation matrix,* $T_\sigma^{(N_{t-1}\times\cdots\times N_0)}$*, is defined by*

$$T_\sigma^{(N_{t-1}\times\cdots\times N_0)}\mathbf{e}_{i_{t-1}}^{N_{t-1}} \otimes \cdots \otimes \mathbf{e}_{i_0}^{N_0} \quad = \quad \mathbf{e}_{\sigma(i_{t-1})}^{N_{t-1}} \otimes \cdots \otimes \mathbf{e}_{\sigma(i_0)}^{N_0} \tag{2-14}$$

In the case that $N_0 = N_1 = \cdots = N_{t-1} = 2$, we replace the superscipt by the product of the radices and the tensor permutation becomes

$$T_\sigma^{2^t}\mathbf{e}_{b_{t-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2 \quad = \quad \mathbf{e}_{\sigma(b_{t-1})}^2 \otimes \cdots \otimes \mathbf{e}_{\sigma(b_0)}^2 \tag{2-15}$$

Example: Let $\sigma = (1, 2, 3, 0)$. Then

$$T_\sigma^{2^4}(e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) = e_{b_0}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2.$$

The corresponding permutation matrix is obtained by setting each bit, b_j, to 0 and 1. This is shown using the binary numbers $(b_3b_2b_1b_0)_2 \rightarrow (b_0b_3b_2b_1)_2$.

$$\begin{array}{ccc}
(0000)_2 & \longrightarrow & (0000)_2 \\
(0001)_2 & \longrightarrow & (1000)_2 \\
(0010)_2 & \longrightarrow & (0001)_2 \\
(0011)_2 & \longrightarrow & (1001)_2 \\
(0100)_2 & \longrightarrow & (0010)_2 \\
(0101)_2 & \longrightarrow & (1010)_2 \\
(0110)_2 & \longrightarrow & (0011)_2 \\
(0111)_2 & \longrightarrow & (1011)_2 \\
(1000)_2 & \longrightarrow & (0100)_2 \\
(1001)_2 & \longrightarrow & (1100)_2 \\
(1010)_2 & \longrightarrow & (0101)_2 \\
(1011)_2 & \longrightarrow & (1101)_2 \\
(1100)_2 & \longrightarrow & (0110)_2 \\
(1101)_2 & \longrightarrow & (1110)_2 \\
(1110)_2 & \longrightarrow & (0111)_2 \\
(1111)_2 & \longrightarrow & (1111)_2
\end{array}$$

□

When all of the radices are equal, the product of two tensor permutations is a tensor permutation and generally the tensor product of tensor permutations is a tensor permutation.

Property 9. *Let σ_1 and σ_2 be two permutations of degree t specifying tensor permutation matrices of size $2^t \times 2^t$, denoted by $T^{2^t}_{\sigma_1}$ and $T^{2^t}_{\sigma_2}$ respectively. Then,*

9.1. $T^{2^t}_{\sigma_1} T^{2^t}_{\sigma_2} = T^{2^t}_{\sigma_1 \sigma_2}$

9.2. $T^{2^t}_{\sigma_1} \otimes T^{2^t}_{\sigma_2} = T^{2^t}_{\sigma_1 \oplus \sigma_2}$

There are two subclasses of the tensor permutation called "stride" and "bit-reversal" permutations that arise in common DFT factorizations. Stride permutations permute the indices of a vector by gathering them at a given stride. They can be defined as tensor permutations.

Definition 8 (Stride Permutation Matrix). *The stride permutation, L^{rs}_s, is the tensor permutation matrix of size $rs \times rs$ defined as follows.*

$$L^{rs}_s \mathbf{e}^r_i \otimes \mathbf{e}^s_j = \mathbf{e}^s_j \otimes \mathbf{e}^r_i \qquad (2\text{-}16)$$

Since $\mathbf{e}^s_j \otimes \mathbf{e}^r_i = e^{rs}_{jr+i}$, we see that the basis elements are permuted at stride r. Using the duality property (7) we see that when a stride permutation is applied to a vector the elements of the vector are gathered at stride s.

Example: Let $N = 4 \times 2$. Then,

$$L^8_2 \mathbf{e}^8_i = L^8_2 \mathbf{e}^4_{i_1} \otimes \mathbf{e}^2_{i_0} = \mathbf{e}^2_{i_0} \otimes \mathbf{e}^4_{i_1} = \mathbf{e}^8_{4i_0+i_1} = \mathbf{e}^8_j$$

Then, counting (i_1, i_0), $0 \leq i_1 < 4$, $0 \leq i_0 < 2$, lexicographically generates the corresponding $j = 4i_0 + i_1$. That is the mapping from $i \to j$ is

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 1 & 5 & 2 & 6 & 3 & 7 \end{pmatrix}$$

Therefore,

$$\begin{aligned} L^8_2 &= \begin{pmatrix} \mathbf{e}^8_0 & \mathbf{e}^8_4 & \mathbf{e}^8_1 & \mathbf{e}^8_5 & \mathbf{e}^8_2 & \mathbf{e}^8_6 & \mathbf{e}^8_3 & \mathbf{e}^8_7 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

The following list provides the basic properties of the stride permutation matrices.

Property 10 (Stride Permutation Properties). *The stride permutation satisfies the following basic properties.*

10.1. $L^{rst}_s L^{rst}_t = L^{rst}_{st}$

10.2. $(L^{rs}_s)^{-1} = L^{rs}_r$

10.3. $L^{rs}_{rs} = L^{rs}_1 = I_{rs}$

Previously it was mentioned that the tensor product is not commutative. However, $(A \otimes B)$ and $(B \otimes A)$ only differ by a permutation. More specifically they are obtained from each other by conjugating by a stride permutation.

Theorem 1 (Commutation Theorem). *Let A be an $r \times r$ matrix and B be an $s \times s$ matrix. Then,*

$$A \otimes B = L^{rs}_r (B \otimes A) L^{rs}_s.$$

The second special case of tensor permutations is called the bit-reversal permutation. In the case when all of the radices are equal to two this permutation is obtained by reversing the bits of the binary representation of the indices of a vector. More generally we give the following definition.

Definition 9 (Bit-Reversal Permutation Matrix). *Let $N = N_{t-1} \times \cdots \times N_0$. Then, the bit-reversal permutation, $R^{(N_{t-1}\times\cdots\times N_0)}$, is defined as following.*

$$R^{(N_{t-1}\times\cdots\times N_0)}\mathbf{e}_{i_{t-1}}^{N_{t-1}} \otimes \cdots \otimes \mathbf{e}_{i_0}^{N_0} \;=\; \mathbf{e}_{i_0}^{N_0} \otimes \cdots \otimes \mathbf{e}_{i_{t-1}}^{N_{t-1}} \qquad (2\text{-}17)$$

For $N = 2 \times \cdots \times 2 = 2^t$, we will drop the superscript and denote the bit-reversal with R_N. That is

$$R_N\mathbf{e}_{b_{t-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2 \;=\; \mathbf{e}_{b_0}^2 \otimes \cdots \otimes \mathbf{e}_{b_{t-1}}^2; \quad 0 \leq b_j < 2 \qquad (2\text{-}18)$$

The following list gathers the basic properties of the bit-reversal permutation matrices.

Property 11. *The bit-reversal R_{2^t} satisfies the following properties.*

11.1. $(R_{2^n})^t = (R_{2^n})^{-1} = R_{2^n}$

11.2. $R_{2^n} = (R_{2^i} \otimes R_{2^{n-i}})L_{2^i}^{2^n}$

11.3. $R_{2^n} = (I_2 \otimes R_{2^{n-1}})L_2^{2^n}$

11.4. $R_{2^n} = \prod_{i=2}^{n}(I_{2^{n-i}} \otimes L_2^{2^i})$

Note that $R_2 = I_2$ and $R_4 = L_2^4$.

Let $N = 2 \times 2 \times 2$. Then,

$$R_8\mathbf{e}_i^8 = R_8\mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2 = \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_2}^2 = \mathbf{e}_{4b_0+2b_1+b_3}^8 = \mathbf{e}_j^8$$

Then, counting $i = (b_2b_1b_0)_2$ lexicographically generates the corresponding $j = (b_0b_1b_2)_2$. That is the mapping from $i \rightarrow j$ is

$$\begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 4 & 2 & 6 & 1 & 5 & 3 & 7 \end{pmatrix}$$

Therefore,

$$\begin{aligned} R_8 &= \begin{pmatrix} \mathbf{e}_0^8 & \mathbf{e}_4^8 & \mathbf{e}_2^8 & \mathbf{e}_6^8 & \mathbf{e}_1^8 & \mathbf{e}_5^8 & \mathbf{e}_3^8 & \mathbf{e}_7^8 \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned}$$

The following examples illustrate Property 11.

1. $R_8 = (I_2 \otimes L_2^4)L_2^8$. Applying both sides to an arbitrary basis vector, we have

$$\begin{aligned} (I_2 \otimes L_2^4)L_2^8 \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2 &= (I_2 \otimes L_2^4)\mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \\ &= \mathbf{e}_{b_0}^2 \otimes L_2^4 \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \\ &= \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_2}^2 \end{aligned}$$

which is the same as $R_8(\mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2)$.

2. $R_{16} = (R_4 \otimes R_4)L_4^{16}$. Applying both sides to an arbitrary basis vector, we have

$$\begin{aligned} (R_4 \otimes R_4)\mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2 &= (R_4 \otimes R_4)\mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \\ &= R_4 \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_0}^2 \otimes R_4 \mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \\ &= \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_3}^2 \end{aligned}$$

which is the same as $R_{16}(\mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2)$. □

By the definition of a tensor permutation, there is one-to-one mapping from permutations of degree t to the permutations of degree N. Let $N = N_{t-1} \times \cdots \times N_0$ and σ be permutation of degree t. Then, the tensor permutation, T_σ, can be written as

$$T_\sigma \mathbf{e}_i^N = \mathbf{e}_{\gamma(i)}^N,$$

where γ is the permutation of degree N (see previous examples),

$$\begin{aligned} i &= (i_{t-1}, \ldots, i_0) = (N_{t-2} \cdots N_0) i_{t-1} + (N_{t-3} \cdots N_0) i_{t-2} + \ldots + N_0 i_1 + i_0, \text{ and} \\ \gamma(i) &= (i_{\sigma(t-1)}, \cdots, i_{\sigma(0)}) \\ &= (N_{\sigma(t-2)} \cdots N_{\sigma(0)}) i_{\sigma(t-1)} + (N_{\sigma(t-3)} \cdots N_{\sigma(0)}) i_{\sigma(t-2)} + \ldots + N_{\sigma(0)} i_{\sigma(1)} + i_{\sigma(0)} \end{aligned}$$

The permutation γ of degree N can be generated by using a counter with adaptive carry propagation specified by $N_{\sigma(t-1)} \times \cdots \times N_{\sigma(0)}$. By the duality property, T_σ, permutes a vector $\mathbf{x}$ by γ^{-1} which can be generated by using σ^{-1} instead of σ;

$$\begin{aligned} \gamma^{-1}(i) &= (i_{\sigma^{-1}(t-1)}, \cdots, i_{\sigma^{-1}(0)}) \\ &= (N_{\sigma^{-1}(t-2)} \cdots N_{\sigma(0)}) i_{\sigma^{-1}(t-1)} + \ldots + N_{\sigma^{-1}(0)} i_{\sigma^{-1}(1)} + i_{\sigma^{-1}(0)}. \end{aligned}$$

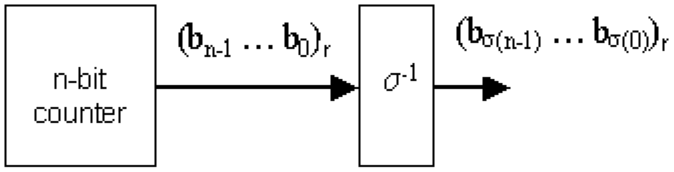


Figure 2.1 Tensor counter

For $N = r^t$, where a radix-r number is used, the permutation γ can be generated by the counting $(i_{t-1}, \cdots, i_0)_r$, $0 \leq i_j < r-1$ and permuting the radix-r digits with the permutation σ as shown in Figure 2.1. for $N = 2^t$, the counter is simply an n-bit binary counter and the permutation σ can be implemented using multiplexers (MUXs). In Chapter 5, two implementation techniques for generating permuted numbers is presented.

Example: Let N $= 2 \times 2 \times 2$ and $\sigma = (1, 2, 0)$ and $\sigma^{-1} = (2, 0, 1)$. Let $(b_2 b_1 b_0)_2$, $0 \leq b_j < 2$ and $0 \leq j < 2$, be a binary number. Then, permuting $(b_2 b_1 b_0)_2$ with σ

result in $(b_{\sigma^{-1}(2)}b_{\sigma^{-1}(1)}b_{\sigma^{-1}(0)})_2 = (b_1b_0b_2)_2$. Counting $i = (b_2b_1b_0)_2 = b_2 \cdot 4 + b_1 \cdot 2 + b_0$ lexicographically generates the corresponding number $j = (b_1b_0b_2)_2 = b_1 \cdot 4 + b_0 \cdot 2 + b_2$. That is

$$\begin{array}{lcllllllll} (b_2b_1b_0)_2 & : & (000)_2 & (001)_2 & (010)_2 & (011)_2 & (100)_2 & (101)_2 & (110)_2 & (111)_2 \\ (b_1b_0b_2)_2 & : & (000)_2 & (010)_3 & (100)_3 & (110)_2 & (001)_2 & (011)_2 & (101)_2 & (111)_2 \end{array}$$

or

$$\gamma = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 \\ 0 & 2 & 4 & 6 & 1 & 3 & 5 & 7 \end{pmatrix}$$

□

2.1.5 Twiddle Factor and Diagonal Matrices

Another important class of matrices arising in FFT factorizations are diagonal matrices containing roots of unity. These diagonal matrices are called twiddle factor matrices. This section collects useful properties of diagonal matrices especially those containing twiddle factors.

Definition 10 (Diagonal Matrix). *Let D be a diagonal matrix of size $N \times N$. Then, $(D)_{i,j} = 0$ if $i \neq j$. Let d_i be the $(i,i)^{th}$ element of the diagonal matrix. Then, we write the diagonal elements of the diagonal matrix as follows.*

$$D \quad = \quad \mathbf{diag}(d_0, d_1, \cdots, d_{N-1}) = \bigoplus_{i=0}^{N-1} d_i \tag{2-19}$$

Conjugating a diagonal matrix with a permutation result in a new diagonal matrix whose diagonal elements are permuted.

Property 12 (Conjugated Diagonal Matrix). *Let D be a diagonal matrix of size $N \times N$. Conjugating D with permutation P_σ results in a new diagonal matrix whose diagonal elements are permuted by σ.*

$$P_\sigma^{-1} D P_\sigma = \mathbf{diag}(d_{\sigma(0)}, d_{\sigma(1)}, \ldots, d_{\sigma(N-1)}) = \bigoplus_{i=0}^{N-1} d_{\sigma(i)} \tag{2-20}$$

Example: Let $\sigma = (1,2,3,0)$ and $D = \mathbf{diag}(d_0, d_1, d_2, d_3)$. Then,

$$\begin{aligned} P_\sigma^{-1} D P_\sigma &= \mathbf{diag}(d_{\sigma(0)}, d_{\sigma(1)}, d_{\sigma(2)}, d_{\sigma(3)}) \\ &= \mathbf{diag}(d_1, d_2, d_3, d_0) \end{aligned}$$

$\square$

A tensor product of diagonal matrices is another diagonal matrix defined as follows.

Definition 11 (Tensor of Diagonal Matrices). *Let D_1 and D_2 be two diagonal matrices of size $r \times r$ and $s \times s$ respectively. Then,*

$$\begin{aligned} D_1 &= \mathbf{diag}(d_0, d_1, \ldots, d_{r-1}) \\ D_2 &= \mathbf{diag}(e_0, e_1, \ldots, e_{s-1}) \\ D_1 \otimes D_2 &= \bigoplus_{i=0}^{r-1} \bigoplus_{j=0}^{s-1} d_i e_j \end{aligned} \tag{2-21}$$

Example: Let $D_1 = \mathbf{diag}(d_0, d_1)$ and $D_2 = \mathbf{diag}(e_0, e_1, e_2)$. Then,

$$\begin{aligned} D_1 &= \begin{pmatrix} d_0 & 0 \\ 0 & d_1 \end{pmatrix}, \quad D_2 = \begin{pmatrix} e_0 & 0 & 0 \\ 0 & e_1 & 0 \\ 0 & 0 & e_2 \end{pmatrix} \\ D_1 \otimes D_2 &= \left(\begin{array}{c|c} d_0 D_2 & \mathbf{0} \\ \hline \mathbf{0} & d_1 D_2 \end{array}\right) = \left(\begin{array}{ccc|ccc} d_0e_0 & 0 & 0 & 0 & 0 & 0 \\ 0 & d_0e_1 & 0 & 0 & 0 & 0 \\ 0 & 0 & d_0e_2 & 0 & 0 & 0 \\ \hline 0 & 0 & 0 & d_1e_0 & 0 & 0 \\ 0 & 0 & 0 & 0 & d_1e_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & d_1e_2 \end{array}\right) \\ &= \bigoplus_{i=0}^{1} \bigoplus_{j=0}^{2} d_i e_j \\ &= \mathbf{diag}(d_0e_0, d_0e_1, d_0e_2, d_1e_0, d_1e_1, d_1e_2) \end{aligned}$$

$\square$

The diagonal matrices occuring in FFT factorizations consist of roots of unity. An N-th root of unity is a number ω such that $\omega^N = 1$. The N-th root of unity, ω, is a primitive root of unity if N is the smallest positive exponent k such that $\omega^k = 1$. In the complex domain there are exactly N N-th roots of unity, namely the complex solutions of the equation $x^N - 1$. The N solutions of $x^N - 1$ are equally spaced on the unit circle and are equal to $\exp(2\pi ij/N)$ for $j = 0, 1, \ldots, N-1$. All of these solutions are obtained as powers of the primitive N-th root of unity equal to $\omega_N = \cos(2\pi/N) + i\sin(2\pi/N)$. Since the N-th roots of unity all have absolute value equal to 1, their inverses are equal to their complex conjugates. In particular, $\omega_N^{-1} = \overline{\omega_N} = \cos(2\pi/N) - i\sin(2\pi/N)$. Observe that ω_N^{-1} is also primitive as are ω_N^k for any k that is relatively prime to N. Finally, we remark, that if $N = rs$, then ω_N^r is a primitive s-th root of unity.

A diagonal matrix containing coefficients for computing the DFT is called a twiddle factor matrix. The common form of twiddle factor matrices found in an FFT algorithm is $P^{-1}(I_q \otimes T_s^{rs})P$. The following properties and definitions show how to compute with twiddle factors in this form.

Definition 12 (T_s^{rs}). *The twiddle factor matrix, denoted by T_s^{rs}, is a diagonal matrix defined by*

$$T_s^{rs}(\mathbf{e}_i^r \otimes \mathbf{e}_j^s) = \omega_{rs}^{ij}(\mathbf{e}_i^r \otimes \mathbf{e}_j^s), \quad 0 \le i < r, \quad 0 \le j < s \tag{2-22}$$

$$T_s^{rs} = \bigoplus_{i=0}^{r-1}\bigoplus_{j=0}^{s-1}\omega_{rs}^{ij} = \bigoplus_{i=0}^{r-1}(W_s(\omega_{rs}))^i \tag{2-23}$$

where

$$W_n(\alpha) = \mathbf{diag}(1, \alpha, \cdots, \alpha^{n-1}). \tag{2-24}$$

Example:

$$\begin{aligned}
T_4^8 \mathbf{e}_i^2 \otimes \mathbf{e}_j^4 &= \omega_8^{ij}(\mathbf{e}_i^2 \otimes \mathbf{e}_j^4) \\
T_4^8 &= \bigoplus_{i=0}^{1} \bigoplus_{j=0}^{3} \omega_8^{ij} \\
&= \mathbf{diag}(\omega_8^{0\cdot 0}, \omega_8^{0\cdot 1}, \omega_8^{0\cdot 2}, \omega_8^{0\cdot 3}, \omega_8^{1\cdot 0}, \omega_8^{1\cdot 1}, \omega_8^{1\cdot 2}, \omega_8^{1\cdot 3}) \\
&= \mathbf{diag}(1, 1, 1, 1, 1, \omega_8, \omega_8^2, \omega_8^3)
\end{aligned}$$

□

Twiddle factor matrices in alternative FFT algorithms are obtained by taking tensor products of identity matrices and the basic twiddle factor matrix T_r^{rs}, and by conjugating by tensor permutations.

Property 13 (Conjugating Twiddle Factor). *Conjugating T_s^{rs} with L_r^{rs} result in T_r^{rs}; i.e.*

$$L_s^{rs} T_s^{rs} L_r^{rs} = T_r^{rs} \tag{2-25}$$

Example: Conjugating T_4^8 with L_2^8, we have

$$\begin{aligned}
L_4^8 T_4^8 L_2^8 \mathbf{e}_i^4 \otimes \mathbf{e}_j^2 &= L_4^8 T_4^8 \mathbf{e}_i^4 \otimes \mathbf{e}_j^2 \\
&= \omega_8^{ji} L_4^8 \mathbf{e}_j^2 \otimes \mathbf{e}_i^4 == \omega_8^{ji} \mathbf{e}_i^4 \otimes \mathbf{e}_j^2 \\
L_4^8 T_4^8 L_2^8 &= \bigoplus_{i=0}^{4} \bigoplus_{j=0}^{1} \omega_8^{ij} \\
&= \mathbf{diag}(\omega_8^{0\cdot 0}, \omega_8^{0\cdot 1}, \omega_8^{1\cdot 0}, \omega_8^{1\cdot 1}, \omega_8^{2\cdot 0}, \omega_8^{2\cdot 1}, \omega_8^{3\cdot 0}, \omega_8^{3\cdot 1}) \\
&= \mathbf{diag}(1, 1, 1, \omega_8, 1, 1, \omega_8^2, 1, \omega_8^3)
\end{aligned}$$

By Definition 12,

$$\begin{aligned}
T_2^8 \mathbf{e}_i^4 \otimes \mathbf{e}_j^2 &= \omega_8^{ij}(\mathbf{e}_i^4 \otimes \mathbf{e}_j^2) \\
T_2^8 &= \bigoplus_{i=0}^{4} \bigoplus_{j=0}^{1} \omega_8^{ij}
\end{aligned}$$

which is the same as $L^8_4 T^8_4 L^8_2$. □

Property 14 (Twiddle Factor $I_q \otimes T^{rs}_s$).

$$\begin{aligned}(I_q \otimes T^{rs}_s)\mathbf{e}^q_i \otimes \mathbf{e}^r_j \otimes \mathbf{e}^s_k &= \omega^{jk}_{rs}(\mathbf{e}^q_i \otimes \mathbf{e}^r_j \otimes \mathbf{e}^s_k) &\quad (2\text{-}26)\\ I_q \otimes T^{rs}_s &= \bigoplus_{i=0}^{q-1}\bigoplus_{j=0}^{r-1}\bigoplus_{k=0}^{s-1} \omega^{jk}_{rs} &\quad (2\text{-}27)\\ &= \bigoplus_{i=0}^{q-1}\bigoplus_{j=0}^{r-1} (W_s(\omega_{rs}))^j &\quad (2\text{-}28)\end{aligned}$$

Example: Consider $L^{32}_4(I_4 \otimes T^8_4)L^{32}_8$.

$$L^{32}_4(I_4 \otimes T^8_4)L^{32}_8 \mathbf{e}^4_i \otimes \mathbf{e}^4_j \otimes \mathbf{e}^2_k = L^{32}_4(I_4 \otimes T^8_4)\mathbf{e}^4_j \otimes \mathbf{e}^2_k \otimes \mathbf{e}^4_i = \omega^{ki}_8 \mathbf{e}^4_i \otimes \mathbf{e}^4_j \otimes \mathbf{e}^2_k$$

$$\begin{aligned}L^{32}_4(I_4 \otimes T^8_4)L^{32}_8 &= \bigoplus_{i=0}^{3}\bigoplus_{j=0}^{3}\bigoplus_{k=0}^{1} \omega^{ik}_8 = \bigoplus_{i=0}^{3}\bigoplus_{j=0}^{3}(W_2(\omega_8))^j\\ W_2(\omega_8) &= \mathbf{diag}(1, \omega_8) = \begin{pmatrix} 1 & 0 \\ 0 & \omega_8 \end{pmatrix}\end{aligned}$$

$$\begin{aligned}L^{32}_4(I_4 \otimes T^8_4)L^{32}_8 = \mathbf{diag}(&1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3,\ 1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3,\\ &1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3,\ 1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^3)\end{aligned}$$

□

2.2 Fast Fourier Transform (FFT)

In this section, the class of FFT algorithms is described as factorizations of the DFT matrix.

Definition 13 (Fourier Matrix). *The Discrete Fourier Transform (DFT) of a vector x of size N is the matrix-vector product $F_N x$, where*

$$F_N = (\omega^{pq}_N), \quad \textit{where } \omega_N = \exp\left(\tfrac{2\pi i}{N}\right),\ 0 \le p, q < N \quad \textit{and } i = \sqrt{-1}, \qquad (2\text{-}29)$$

is the $N \times N$ discrete Fourier matrix [26].

Example:

$$F_8 = \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8 & \omega_8^2 & \omega_8^3 & -1 & -\omega_8 & -\omega_8^2 & -\omega_8^3 \\ 1 & \omega_8^2 & -1 & -\omega_8^2 & 1 & \omega_8^2 & -1 & -\omega_8^2 \\ 1 & \omega_8^3 & -\omega_8^2 & \omega_8 & -1 & -\omega_8^3 & \omega_8^2 & -\omega_8 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega_8 & \omega_8^2 & -\omega_8^3 & -1 & \omega_8 & \omega_8^2 & \omega_8^3 \\ 1 & -\omega_8^2 & -1 & \omega_8^2 & 1 & \omega_8^2 & -1 & \omega_8^2 \\ 1 & -\omega_8^3 & -\omega_8^2 & -\omega_8 & -1 & \omega_8^3 & \omega_8^2 & \omega_8 \end{pmatrix}$$

Note that we apply the properties that $\omega_8^8 = 1$ and $\omega_8^4 = -1$. □

The class of algorithms called Fast Fourier Transform (FFT) can compute the DFT of vector $\mathbf{x}$ in $O(N \log(N))$ operations instead of the $O(N^2)$ operations required when computing the DFT by definition. Many authors [26–28, 31] have shown that different factorizations of the DFT matrix correspond to different FFT algorithms. In the following subsections, we describe the two well-known FFT algorithms, the Cooley-Tukey and Pease algorithms, using matrix factorizations.

2.2.1 Cooley-Tukey Algorithm

The FFT algorithm was popularly introduced by James W. Cooley and J.W. Tukey [32] in 1965. The following theorem describes the Cooley-Tukey algorithm with matrix factorization.

Theorem 2 (Cooley-Tukey Theorem). *Let $N = rs$. Then, the FFT algorithm introduced by Cooley-Tukey can be derived using the following factorization.*

$$F_{rs} = (F_r \otimes I_s) T_s^{rs} (I_r \otimes F_s) L_s^{rs} \tag{2-30}$$

Proof: *The proof of the theorem can be found in [28].* □

Example: The Cooley-Tukey algorithm for 8-point DFT can be written as the following factorization of F_8.

$$F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_4^8$$

Applying necessary properties described in previous sections, we show that the factorization is valid.

$$\begin{aligned}
(F_2 \otimes I_4)T_4^8(I_2 \otimes F_4) &= \begin{pmatrix} I_4 & I_4 \\ I_4 & -I_4 \end{pmatrix} \begin{pmatrix} I_4 & \mathbf{0} \\ \mathbf{0} & W_4(\omega_8) \end{pmatrix} \begin{pmatrix} F_4 & \mathbf{0} \\ \mathbf{0} & F_4 \end{pmatrix} \\
&= \begin{pmatrix} I_4 & W_4(\omega_8) \\ I_4 & -W_4(\omega_8) \end{pmatrix} \begin{pmatrix} F_4 & \mathbf{0} \\ \mathbf{0} & F_4 \end{pmatrix} \\
&= \begin{pmatrix} F_4 & W_4(\omega_8)F_4 \\ F_4 & -W_4(\omega_8)F_4 \end{pmatrix}
\end{aligned}$$

Since

$$\begin{aligned}
F_4 &= \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & \omega_4 & -1 & -\omega_4 \\ 1 & -1 & 1 & -1 \\ 1 & -\omega_4 & -1 & \omega_4 \end{pmatrix}, W_4(\omega_8) = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & \omega_8 & 0 & 0 \\ 0 & 0 & \omega_8^2 & 0 \\ 0 & 0 & 0 & \omega_8^3 \end{pmatrix}, \quad \text{and} \\
W_4(\omega_8)F_4 &= \begin{pmatrix} 1 & 1 & 1 & 1 \\ \omega_8 & \omega_8^3 & -\omega_8 & -\omega_8^3 \\ \omega_8^2 & -\omega_8^2 & \omega_8^2 & -\omega_8^2 \\ \omega_8^3 & \omega_8 & -\omega_8^3 & -\omega_8 \end{pmatrix},
\end{aligned}$$

we have

$$(F_2 \otimes I_4)T_4^8(I_2 \otimes F_4) = \begin{pmatrix}
1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
1 & \omega_8^2 & -1 & -\omega_8^2 & \omega_8 & \omega_8^3 & -\omega_8 & -\omega_8^3 \\
1 & -1 & 1 & -1 & \omega_8^2 & -\omega_8^2 & \omega_8^2 & -\omega_8^2 \\
1 & -\omega_8 & -1 & \omega_8 & \omega_8^3 & \omega_8 & -\omega_8^3 & -\omega_8 \\
1 & 1 & 1 & 1 & -1 & -1 & -1 & -1 \\
1 & \omega_8^2 & -1 & -\omega_8^2 & -\omega_8 & -\omega_8^3 & \omega_8 & \omega_8^3 \\
1 & -1 & 1 & -1 & -\omega_8^2 & \omega_8^2 & -\omega_8^2 & \omega_8^2 \\
1 & -\omega_8 & -1 & \omega_8 & -\omega_8^3 & -\omega_8 & \omega_8^3 & \omega_8
\end{pmatrix}$$

Permuting the columns of the resulting matrix with stride 4, we obtain

$$(F_2 \otimes I_4)T^8_4(I_2 \otimes F_4)L^8_4 \;=\; \begin{pmatrix} 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\ 1 & \omega_8 & \omega_8^2 & \omega_8^3 & -1 & -\omega_8 & -\omega_8^2 & -\omega_8^3 \\ 1 & \omega_8^2 & -1 & -\omega_8^2 & 1 & \omega_8^2 & -1 & -\omega_8^2 \\ 1 & \omega_8^3 & -\omega_8^2 & \omega_8 & -1 & -\omega_8^3 & \omega_8^2 & -\omega_8 \\ 1 & -1 & 1 & -1 & 1 & -1 & 1 & -1 \\ 1 & -\omega_8 & \omega_8^2 & -\omega_8^3 & -1 & \omega_8 & -\omega_8^2 & \omega_8^3 \\ 1 & -\omega_8^2 & -1 & \omega_8^2 & 1 & -\omega_8^2 & -1 & \omega_8^2 \\ 1 & -\omega_8^3 & -\omega_8^2 & -\omega_8 & -1 & \omega_8^3 & \omega_8^2 & \omega_8 \end{pmatrix} = F_8$$

□

The factorization in Equation 2-30 can be applied recursively and naturally leads to a recursive FFT algorithm. An iterative version of the Cooley-Tukey algorithm is described by the following theorem. The interpretation of DFT factorizations as programs can be found in [28].

Theorem 3 (Iterative Cooley-Tukey Algorithm). *Let $N = 2^n$. Then, the following matrix factorization describes the iterative Cooley-Tukey FFT algorithm.*

$$F_{2^n} \;=\; \left\{ \prod_{i=0}^{n-1} (I_{2^i} \otimes F_2 \otimes I_{2^{n-i-1}})(I_{2^i} \otimes T^{2^{n-i}}_{2^{n-i-1}}) \right\} R_{2^n} \qquad (2\text{-}31)$$

Proof: *This follows by induction using Theorem 2 and properties of the tensor product in Property 1 (see [28]).* □

The following example shows the derivation of the 8-point iterative Cooley-Tukey algorithm.

Example: The iterative Cooley-Tukey algorithm for the 8-point DFT, following Theorem 3, is

$$F_8 = (F_2 \otimes I_4)T^8_4(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T^4_2)(I_4 \otimes F_2)R_8 \qquad (2\text{-}32)$$

$$\begin{array}{rcll}
F_8 & = & (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_4^8 & \text{Theorem 2} \\
 & = & (F_2 \otimes I_4)T_4^8(I_2 \otimes (F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4)L_4^8 & \text{Theorem 2} \\
 & = & (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_2 \otimes I_2 \otimes F_2)(I_2 \otimes L_2^4)L_4^8 & \text{Property 1.11} \\
 & = & (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2)R_8 & \text{Property 1.13} \\
 & = & (F_2 \otimes I_4)T_4^8(I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4)(I_4 \otimes F_2)R_8 & \text{Property 11}
\end{array}$$

□

Note that the theorem can be extended to a multi-radix factorization, where N is factored into $N_{t-1} \times \cdots \times N_0$ (see [28]).

An FFT factorization can be interpreted as a dataflow diagram (sometimes called a signal flow graph) which depicts the computation as a graph showing the arithmetic operations and their dependencies. Given an FFT factorization, we can construct the corresponding FFT dataflow.

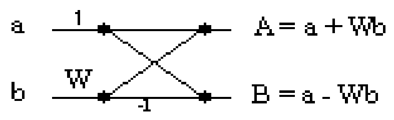


Figure 2.2 Basic butterfly operation

An FFT factorization typically consists of n stages of the form $(I_p \otimes F_2 \otimes I_q)T$, where T is a twiddle factor matrix. The dataflow diagram can be drawn where each column corresponds to a stage in the factorization. Figure 2.3 shows the dataflow diagram for the 8-point Cooley-Tukey algorithm described by by Equation 2-32 (see [1]). Note that the dataflow diagram is read from left to right, whereas the FFT factorization is computed from right to left.

Since the twiddle factor matrix is a diagonal matrix, it is translated into the vector product of the input vector to each stage and the vector storing the diagonal elements of the matrix. In the dataflow, the operation is represented by placing the twiddle

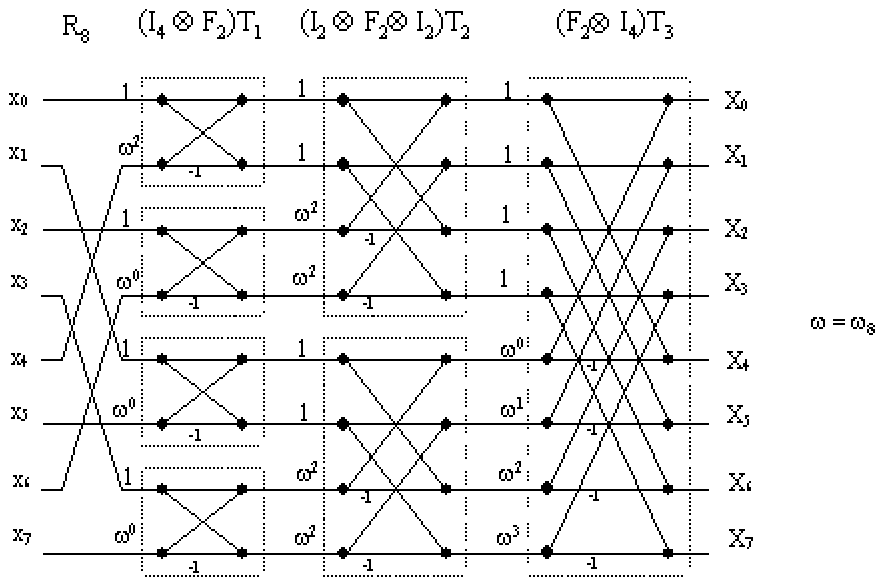


Figure 2.3 Dataflow diagram of the 8-point Cooley-Tukey algorithm

factors over an edge in the graph. When data traverses the edge it is multiplied by the corresponding twiddle factor.

The results from a column of twiddle factor multiplications become the inputs to the operation $(I_p \otimes F_2 \otimes I_q)$, which consists of pq copies of F_2, and is translated into pq bufferfly operations which add and subtract the inputs (addition is depicted by a node in the graph). A butterfly operation, F_2 combined with twiddle factor multiplication, is shown in Figure 2.2.

If after each stage the temporary results are stored in memory, then the inputs to a bufferfly operation can be indicated by a pair of addresses. The corresponding addresses for the four butterfly operations corresponding to the factor $(I_2 \otimes F_2 \otimes I_2)$ can be computed as follows.

$$(I_2 \otimes F_2 \otimes I_2)\mathbf{e}_i^2 \otimes \mathbf{e}_j^2 \otimes \mathbf{e}_k^2 \;=\; \mathbf{e}_i^2 \otimes F_2\mathbf{e}_j^2 \otimes \mathbf{e}_k^2 = (I_2 \otimes F_2 \otimes I_2)\mathbf{e}_{4i+2j+k}^8$$

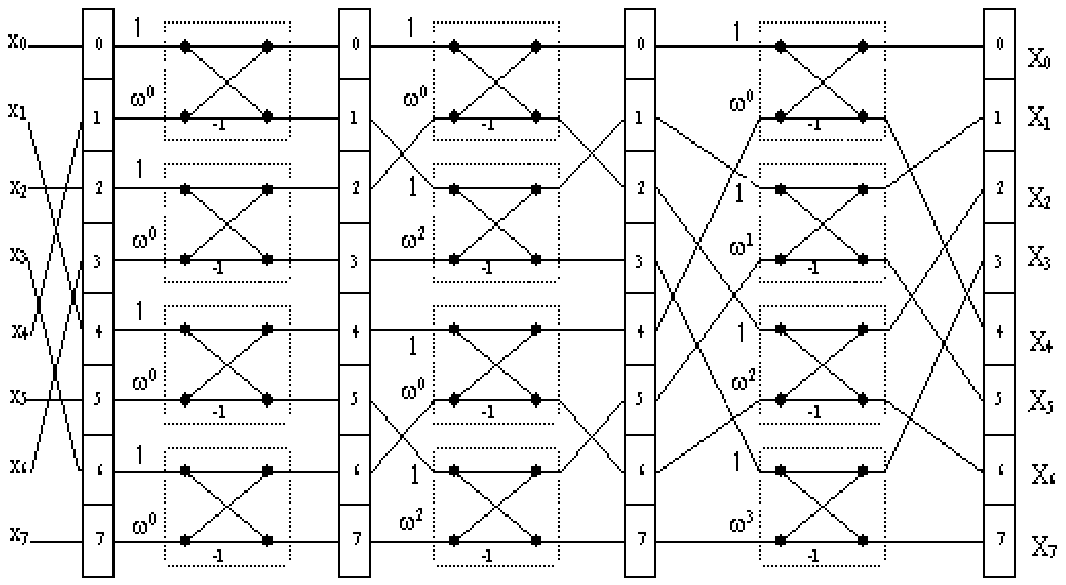


Figure 2.4 Dataflow diagram of the 8-point Cooley-Tukey algorithm in parallel form

The two addresses of the inputs to a butterfly operation are obtained by setting $j = 0$ and $j = 1$ respectively; i.e. the two addresses are equal to $4i + k$ and $4i + k + 2$. Counting (i, j, k) lexicographically while computing $4i + 2k + j$, we generate the sequence of addresses needed by the butterfly operations.

$$\begin{array}{rcccccccc} (i,j,k): & (0,0,0) & (0,0,1) & (0,1,0) & (0,1,1) & (1,0,0) & (1,0,1) & (1,1,0) & (1,1,1) \\ a: & 0 & 2 & 1 & 3 & 4 & 6 & 5 & 7 \end{array}$$

The dataflow in Figure 2.3 groups the butterfly operations corresponding to $(F_2 \otimes I_p)$ so that they can be interpreted as a butterfly operation on a vector of size p. Using the Commutation Theorem (Theorem 1), $(F_2 \otimes I_p) = L_p^{2p}(I_p \otimes F_2)L_2^{2p}$, and the computation can be reorganized into a parallel form with a sequence of butterfly operations whose inputs are accessed at stride. The permutation L_2^{2p} specifies the butterfly addresses. Figure 2.4 shows the dataflow of the 8-point Cooley-Tukey algorithm when grouping all operations in the parallel form. The columns between stages represent

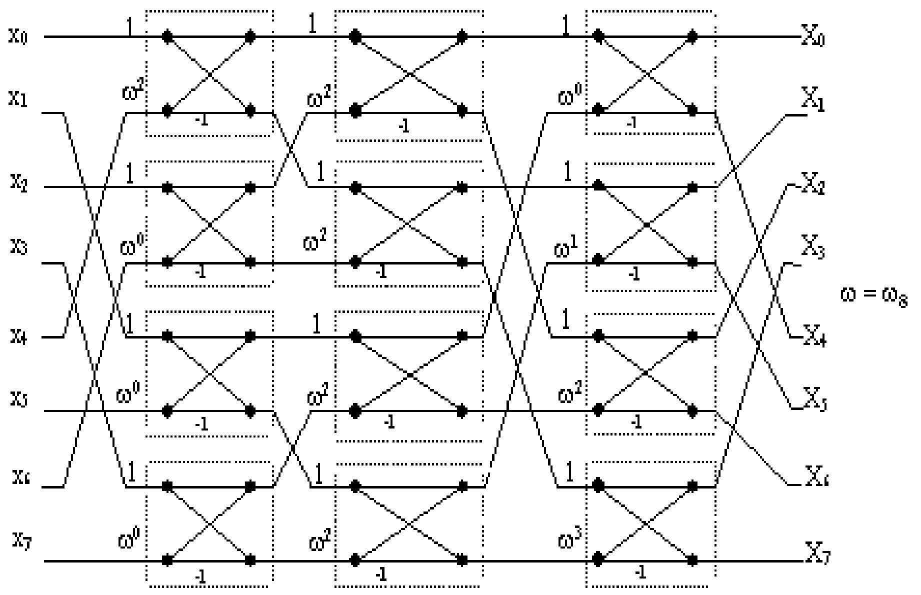


Figure 2.5 Dataflow diagram of the 8-point Cooley-Tukey algorithm in parallel form and combining permutations between stages

memory where the intermediate results are stored, and each butterfly operation explicitly refers to a pair of addresses in memory. Combining the permutations between stages, we obtain another variation of the dataflow of the Cooley-Tukey algorithm as shown in Figure 2.5.

In the remainder of this thesis we consider FFT algorithms in the parallel form with explicit addressing between stages.

Definition 14 (Parallel In-Place FFT).

$$\left\{\prod_{i=0}^{n-1} P_i^{-1}(I_{2^{n-1}} \otimes F_2)P_i\right\} R$$

where P_i, for $i = 0, \ldots, n-1$ and R are tensor permutations. Since each computation $(I_{2^{n-1}} \otimes F_2)$ is conjugated by a permutation, it can be done in-place. This means that the input and output addresses for each butterfly operation are the same.

2.2.2 Pease Algorithm

In this section we review an FFT algorithm developed by M. C. Pease [33] that is in the parallel form (Definition 14). Pease originally presented his algorithm as one well suited for parallel processing. Moreover, he combined the permutations between the stages and observed that each permutation as identical. Conseqently his algorithm is sometimes called a constant geometry algorithm. Since we want an in-place algorithm we present Pease's original algorithm and a modified version.

Theorem 4 (Pease Algorithm). *Let $N = 2^n$. Then,*

$$F_{2^n} = \left\{ \prod_{i=0}^{n-1} L_2^{2^n} (I_{2^{n-1}} \otimes F_2) T_{n-i} \right\} R_{2^n}, \tag{2-33}$$

$$F_{2^n} = \left\{ \prod_{i=0}^{n-1} L_{2^{n-i-1}}^{2^n} (I_{2^{n-1}} \otimes F_2) T_{n-i} L_{2^{i+1}}^{2^n} \right\} R_{2^n}, \tag{2-34}$$

$$T_{n-i} = L_{2^{n-i-1}}^{2^n} (I_{2^i} \otimes T_{2^{n-i-1}}^{2^{n-i}}) L_{2^{i+1}}^{2^n} \tag{2-35}$$

Proof: *Both factorizations in this theorem can be derived from the iterative Cooley-Tukey factorization (Theorem 3) using the Commutation theorem (Theorem 1) and properties of stride permutations. (see [28].)* □

Both factorizations (Equation 2-33 and 2-34) have the same twiddle factors described by Equation 2-35. Using twiddle factor properties (Section 2.1.5) and tensor product product properties (Section 2.1.1), we can write the twiddle factors T_{n-i} as a direct sum.

Let $r = 2^{n-i-1}$, $s = 2^i$, and $N = 2rs = 2^n$. Then,

$$T_{n-i} = L_{2^{n-i-1}}^{2^n} (I_{2^i} \otimes T_{2^{n-i-1}}^{2^{n-i}}) L_{2^{i+1}}^{2^n} = L_r^{2rs} (I_s \otimes T_r^{2r}) L_{2s}^{2rs},$$

and

$$\begin{aligned}
T_{n-i}\mathbf{e}_a^r \otimes \mathbf{e}_b^s \otimes \mathbf{e}_c^2 &= L_r^{2rs}(I_s \otimes T_r^{2r})L_{2s}^{2rs}\mathbf{e}_a^r \otimes \mathbf{e}_b^s \otimes \mathbf{e}_c^2 \\
&= L_r^{2rs}(I_s \otimes T_r^{2r})\mathbf{e}_b^s \otimes \mathbf{e}_c^2 \otimes \mathbf{e}_a^r \\
&= L_r^{2rs}(\mathbf{e}_b^s \otimes T_r^{2r} \otimes \mathbf{e}_c^2 \otimes \mathbf{e}_a^r) \\
&= \omega_{2r}^{a\cdot c}(L_r^{2rs}\mathbf{e}_b^s \otimes \mathbf{e}_c^2 \otimes \mathbf{e}_a^r) \\
&= \omega_{2r}^{a\cdot c}(\mathbf{e}_a^r \otimes \mathbf{e}_b^s \otimes \mathbf{e}_c^2) \\
T_{n-i} &= \bigoplus_{a=0}^{r-1}\bigoplus_{b=0}^{s-1}\bigoplus_{c=0}^{1}\omega_{2r}^{a\cdot c}, \quad \text{where } r = 2^{n-i-1},\ s = 2^i \qquad \text{(2-36)}
\end{aligned}$$

The following example illustrates the derivation of both forms of the Pease algorithm. shown in the following example.

Example: By the Theorem 4, the Pease algorithm of 16-point DFT is described by

$$\begin{aligned}
F_{16} &= L_2^{16}(I_8 \otimes F_2)T_4L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3L_4^{16} \cdot \\
&\quad L_8^{16}(I_8 \otimes F_2)T_2L_2^{16} \cdot (I_8 \otimes F_2)T_1R_{16} \qquad \text{(2-37)} \\
F_{16} &= L_2^{16}(I_8 \otimes F_2)T_4L_4^{8} \cdot L_2^{16}(I_8 \otimes F_2)T_3L_2^{8} \cdot \\
&\quad L_2^{16}(I_8 \otimes F_2)T_2L_2^{8} \cdot L_2^{16}(I_8 \otimes F_2)T_1R_{16} \qquad \text{(2-38)}
\end{aligned}$$

where

$$\begin{aligned}
T_1 &= I_8 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1) \\
T_2 &= L_2^{16}(I_4 \otimes T_2^4)L_8^{16} = \bigoplus_{a=0}^{1}\bigoplus_{b=0}^{3}\bigoplus_{c=0}^{1}\omega_4^{ac} \\
&= \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_4,1,\omega_4,1,\omega_4,1,\omega_4)
\end{aligned}$$

$$\begin{aligned}
T_3 &= L_4^{16}(I_2 \otimes T_4^8)L_4^{16} = \bigoplus_{a=0}^{3}\bigoplus_{b=0}^{1}\bigoplus_{c=0}^{1}\omega_8^{ac} \\
&= \mathbf{diag}(1,1,1,1,1,\omega_8,1,\omega_8,1,\omega_8^2,1,\omega_8^2,1,\omega_8^3,1,\omega_8^3) \\
T_4 &= L_8^{16}T_8^{16}L_8^{16} = \bigoplus_{a=0}^{7}\bigoplus_{c=0}^{1}\omega_8^{ac} \\
&= \mathbf{diag}(1,1,1,\omega_{16},1,\omega_{16}^2,1,\omega_{16}^3,1,\omega_{16}^4,1,\omega_{16}^5,1,\omega_{16}^6,1,\omega_{16}^7)
\end{aligned}$$

Both Equation 2-37 and 2-38 can be derived from the iterative Cooley-Tukey as follows.

$$\begin{aligned}
F_{16} &= (F_2 \otimes I_8)T_8^{16} \cdot (I_2F_2 \otimes I_4)(I_2 \otimes T_4^8)\cdot \\
&\quad (I_2 \otimes F_2 \otimes I_2)(I_2 \otimes T_2^4) \cdot (I_4 \otimes F_2) \cdot R_8 && \text{(Theorem 3)} \\
&= L_2^{16}(I_8 \otimes F_2)L_8^{16}T_8^{16} \cdot L_4^{16}(I_4 \otimes I_2 \otimes F_2)L_4^{16}(I_2 \otimes T_4^8)\cdot \\
&\quad L_8^{16}(I_2 \otimes I_4 \otimes F_2)L_2^{16}(I_4 \otimes T_2^4) \cdot (I_8 \otimes F_2) \cdot R_{16} && \text{(Property 10.4)} \\
&= L_2^{16}(I_8 \otimes F_2)L_8^{16}T_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)L_4^{16}(I_2 \otimes T_4^8)\cdot \\
&\quad L_8^{16}(I_8 \otimes F_2)L_2^{16}(I_4 \otimes T_2^4) \cdot (I_8 \otimes F_2) \cdot R_{16} && \text{(Property 1.13)} \\
&= L_2^{16}(I_8 \otimes F_2)T_4L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3L_4^{16}\cdot \\
&\quad L_8^{16}(I_8 \otimes F_2)T_2L_2^{16} \cdot (I_8 \otimes F_2)T_1 \cdot R_{16}
\end{aligned}$$

where

$$\begin{aligned}
T_1 &= I_8 \\
T_2L_2^8 &= L_2^{16}(I_4 \otimes T_2^4) &&\Leftrightarrow T_2 = L_2^{16}(I_4 \otimes T_2^4)L_8^{16} \\
T_3L_4^{16} &= L_4^{16}(I_2 \otimes T_4^8) &&\Leftrightarrow T_3 = L_4^{16}(I_2 \otimes T_4^8)L_4^{16} \\
T_4L_8^{16} &= L_8^{16}T_8^{16} &&\Leftrightarrow T_4 = L_8^{16}T_8^{16}L_2^{16}
\end{aligned}$$

This is the same as Equation 2-37. The dataflow for the conjugate form of the Pease algorithm is shown in Figure 2.6.

Combining the permutation between stages, we obtain Equation 2-38.

$$\begin{aligned}
F_{16} &= L_2^{16}(I_8 \otimes F_2)T_4 \cdot L_8^{16}L_4^{16}(I_8 \otimes F_2)T_3 \cdot L_4^{16}L_8^{16}(I_8 \otimes F_2)T_2 \cdot L_2^{16}(I_8 \otimes F_2)T_1 \cdot R_{16} \\
&= L_2^{16}(I_8 \otimes F_2)T_4 \cdot L_2^{16}(I_8 \otimes F_2)T_3 \cdot L_2^{16}(I_8 \otimes F_2)T_2 \cdot L_2^{16}(I_8 \otimes F_2) \cdot R_{16}
\end{aligned}$$

Figure 2.7 shows the dataflow diagram of the Pease algorithm for 16-point DFT when the permutation between stages are combined. □

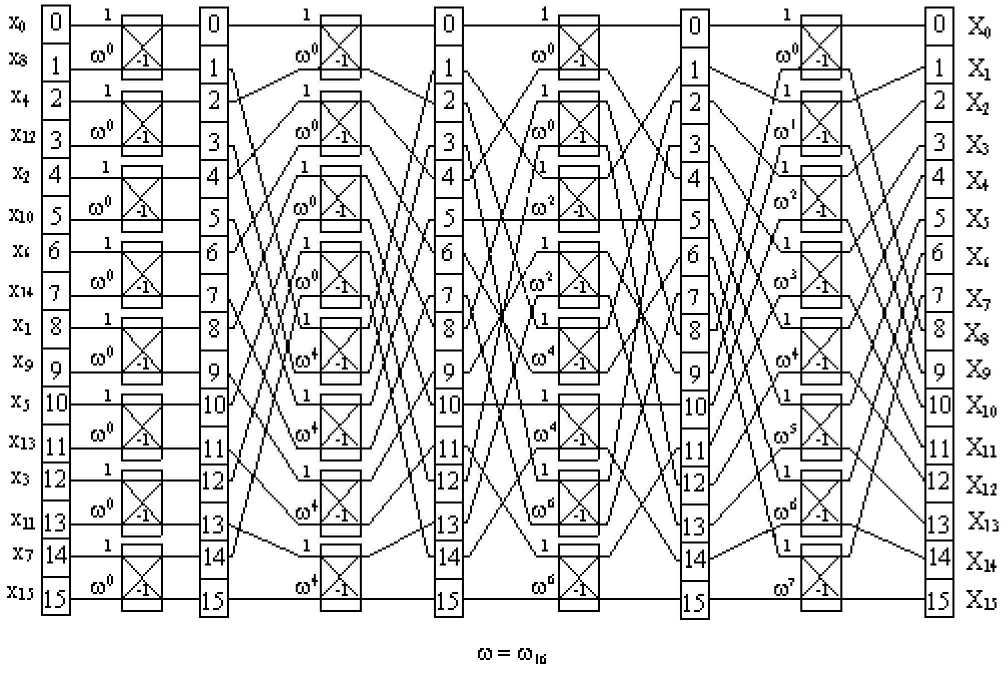


Figure 2.6 Dataflow diagram of the 16-point conjugate Pease algorithm

2.3 Multidimensional DFT

In this section we define the multidimensional DFT and review common algorithms for computing the multidimensional DFT. We also show by an example how to relate the computation of the multidimensional DFT to the computation of the one-dimensional DFT. This leads into our discussion of the dimensionless FFT in the next section.

Definition 15 (Multidimensional DFT). *Let $X(a_1, \ldots, a_t)$ be a function of t variables, where $0 \leq a_i < N_i$. The t-dimensional $(N_1 \times \cdots \times N_t)$-point DFT of X is*

$$\hat{X}(b_1, \ldots, b_t) = \sum_{0 \leq a_i < n_i} e^{\frac{2\pi i}{n_1} a_1 b_1} \cdots e^{\frac{2\pi i}{n_t} a_t b_t} X(a_1, \ldots, a_t)$$

The multidimensional DFT can be interpreted as a matrix-vector product. Let $\mathbf{x}$ and $\hat{\mathbf{x}}$ be the vectors of size N obtained by ordering the elements of X and $\hat{X}$

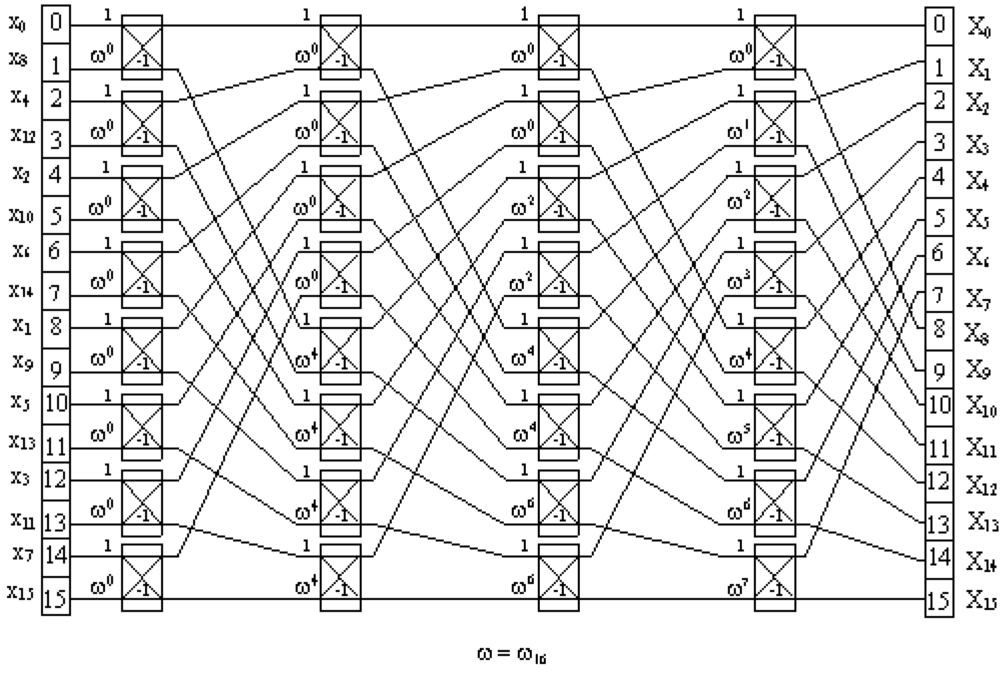


Figure 2.7 Dataflow diagram of the 16-point Pease algorithm

lexicographically. Then, $\hat{\mathbf{x}} = (F_{N_1} \otimes \cdots \otimes F_{N_t})x$, where F_{N_i} is the N_i-point discrete Fourier matrix defined in Definition 13.

Example: Let $X(a_1, a_2)$ be a function of 2 variables denoted by a_1 and a_2, where $0 \le a_1 < 4$ and $0 \le a_2 < 4$. Then, the two-dimensional DFT of X is

$$\hat{X}(b_1, b_2) = \sum_{a_1=0}^{4} \sum_{a_2=0}^{4} e^{\frac{2\pi i}{4} a_1 b_1} \cdot e^{\frac{2\pi i}{4} a_2 b_2} X(a_1, a_2)$$

Let X and $\hat{X}$ be two 4×4 matrices storing elements of X and $\hat{X}$ respectively. Then,

$$X = \begin{pmatrix} X(0,0) & X(0,1) & X(0,2) & X(0,3) \\ X(1,0) & X(1,1) & X(1,2) & X(1,3) \\ X(2,0) & X(2,1) & X(2,2) & X(2,3) \\ X(3,0) & X(3,1) & X(3,2) & X(3,3) \end{pmatrix}$$

$$\hat{X} = \begin{pmatrix} \hat{X}(0,0) & \hat{X}(0,1) & \hat{X}(0,2) & \hat{X}(0,3) \\ \hat{X}(1,0) & \hat{X}(1,1) & \hat{X}(1,2) & \hat{X}(1,3) \\ \hat{X}(2,0) & \hat{X}(2,1) & \hat{X}(2,2) & \hat{X}(2,3) \\ \hat{X}(3,0) & \hat{X}(3,1) & \hat{X}(3,2) & \hat{X}(3,3) \end{pmatrix}$$

Let $\mathbf{x}$ and $\hat{\mathbf{x}}$ be the vectors of size 16 obtained by ordering the elements of X and $\hat{X}$ lexicographically; that is

$$\mathbf{x} = \begin{pmatrix} X(0,0) \\ X(0,1) \\ X(0,2) \\ X(0,3) \\ X(1,0) \\ X(1,1) \\ X(1,2) \\ X(1,3) \\ X(2,0) \\ X(2,1) \\ X(2,2) \\ X(2,3) \\ X(3,0) \\ X(3,1) \\ X(3,2) \\ X(0,3) \end{pmatrix}, \quad \hat{\mathbf{x}} = \begin{pmatrix} \hat{X}(0,0) \\ \hat{X}(0,1) \\ \hat{X}(0,2) \\ \hat{X}(0,3) \\ \hat{X}(1,0) \\ \hat{X}(1,1) \\ \hat{X}(1,2) \\ \hat{X}(1,3) \\ \hat{X}(2,0) \\ \hat{X}(2,1) \\ \hat{X}(2,2) \\ \hat{X}(2,3) \\ \hat{X}(3,0) \\ \hat{X}(3,1) \\ \hat{X}(3,2) \\ \hat{X}(3,3) \end{pmatrix}$$

Then, $\hat{\mathbf{x}} = (F_4 \otimes F_4)\mathbf{x}$, where F_4 is 4-point discrete Fourier matrix.

□

We can compute the 2-dimensional $(N_1 \times N_2)$-point DFT in many different ways. The common algorithm is called the "row-column" algorithm which can be described by the following factorization.

$$\begin{aligned} \hat{X} &= F_{N_1} X F_{N_0} \\ \hat{\mathbf{x}} &= (F_{N_1} \otimes F_{N_0})\mathbf{x} = (F_{N_1} \otimes I_{N_0})(I_{N_1} \otimes F_{N_0})\mathbf{x} \end{aligned}$$

The "row-column" algorithm applies the N_2-point, F_{N_2}, to the rows of X resulting in $\bar{X} = XF_{N_2}$. Then, it performs N_1-point FFT on the column of $\bar{X}$. The factorization

$(F_{N_1} \otimes I_{N_2}\mathbf{x})(I_{N_1} \otimes F_{N_2})$ can also be interpreted as the row-column algorithm. Using the Commutation Theorem (Theorem 1),

$$\hat{\mathbf{x}} = L_{N_1}^{N_1 N_0}(I_{N_0} \otimes F_{N_1}) L_{N_0}^{N_1 N_0}(I_{N_1} \otimes F_{N_0})\mathbf{x} \tag{2-39}$$

This factorization can be interpreted as the row-column algorithm with an explicit transposition (corner turning), when the stride permutations are performed as run-time permutations of the data.

The following example illustrate the "row-column" algorithm for (4×4)-point DFT.

Example: Consider again (4×4)-point FFT. Let

$$\mathbf{x} = \begin{pmatrix} \mathbf{x_0} \\ \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_3} \end{pmatrix}, \quad \text{where } \mathbf{x_i} = \begin{pmatrix} X(i,0) \\ X(i,1) \\ X(i,2) \\ X(i,3) \end{pmatrix}, \ 0 \le i < 4,$$

is the i^{th} row of X. Then, $(I_4 \otimes F_4)\mathbf{x}$ is the computation F_4 on the rows of X; i.e.

$$\begin{aligned} (F_4 \otimes F_4)\mathbf{x} &= (F_4 \otimes I_4)(I_4 \otimes F_4)\mathbf{x} \\ &= (F_4 \otimes I_4) \begin{pmatrix} F_4 & \mathbf{0} & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & F_4 & \mathbf{0} & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & F_4 & \mathbf{0} \\ \mathbf{0} & \mathbf{0} & \mathbf{0} & F_4 \end{pmatrix} \begin{pmatrix} \mathbf{x_0} \\ \mathbf{x_1} \\ \mathbf{x_2} \\ \mathbf{x_3} \end{pmatrix} = (F_4 \otimes I_4) \begin{pmatrix} F_4\mathbf{x_0} \\ F_4\mathbf{x_1} \\ F_4\mathbf{x_2} \\ F_4\mathbf{x_3} \end{pmatrix} \\ &= (F_4 \otimes I_4)\bar{\mathbf{x}}. \end{aligned}$$

The vector operator $(F_4 \otimes I_4)\bar{\mathbf{x}}$ is the computation of F_4 on the columns of $\bar{X}$. Following the commutation theorem (Property 10), we have

$$(F_4 \otimes I_4)\bar{\mathbf{x}} = L_4^{16}(I_4 \otimes F_4)L_4^{16}\bar{\mathbf{x}}$$

That is L_4^{16} permutes $\bar{\mathbf{x}}$ in stride 4 resulting that

$$L_4^{16}\bar{\mathbf{x}} = \begin{pmatrix} \bar{\mathbf{x}}_0 \\ \bar{\mathbf{x}}_1 \\ \bar{\mathbf{x}}_2 \\ \bar{\mathbf{x}}_3 \end{pmatrix}, \quad \text{where } \bar{\mathbf{x}}_i = \begin{pmatrix} \bar{X}(0,i) \\ \bar{X}(1,i) \\ \bar{X}(2,i) \\ \bar{X}(3,i) \end{pmatrix}, \ 0 \le i < 4,$$

That is $\bar{\mathbf{x}}_i$ stores the i^{th} column of $\bar{X}$. Therefore,

$$L_4^{16}(I_4 \otimes F_4) \quad = \quad \begin{pmatrix} \bar{\mathbf{x}}_0 \\ \bar{\mathbf{x}}_1 \\ \bar{\mathbf{x}}_2 \\ \bar{\mathbf{x}}_3 \end{pmatrix} = L_4^{16} \begin{pmatrix} F_4\bar{\mathbf{x}}_0 \\ F_4\bar{\mathbf{x}}_1 \\ F_4\bar{\mathbf{x}}_2 \\ F_4\bar{\mathbf{x}}_3 \end{pmatrix}.$$

Note that the permutation L_4^{16} permutes the result back to the original order. □

The row-column algorithm illustrated for a two-dimensional DFT can easily be extended to an arbitrary multidimensional DFT.

Theorem 5 (Multidimensional FFT).

$$\begin{aligned} F_{N_1} \otimes \cdots \otimes F_{N_t} &= \prod_{i=1}^{t} (I_{N(i-1)} \otimes F_{N_i} \otimes I_{N/N(i)}) \\ &= \prod_{i-1}^{t} L_{N_i}^{N}(I_{N/N_i} \otimes F_{N_i}) \end{aligned}$$

Proof: *This follows by induction using properties of the tensor product in Property 1 (see [28]).* □

If instead we replace F_{N_i} in $F_{N_1} \otimes \cdots \otimes F_{N_t}$ by the iterative Cooley-Tukey factorization and use properties of the tensor product to manipulate the resulting equation, it is easy to derive a factorization similar to the iterative Cooley-Tukey factorization for multidimensional DFTs. The following example illustrates this factorization for $F_4 \otimes F_4$.

Example: Applying the iterative Cooley-Tukey factorization in Equation 2-30 to F_4 in $F_4 \times F_4$, we have

$$\begin{aligned} F_4 \otimes F_4 &= (F_4 \otimes I_4)(I_4 \otimes F_4) \\ &= ([(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4] \otimes I_4) \\ &\quad (I_4 \otimes [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4] \end{aligned}$$

Applying tensor product properties (Property 1), we obtain

$$\begin{aligned} F_4 \otimes F_4 &= (F_2 \otimes I_8)(T_2^4 \otimes I_4)(I_2 \otimes F_2 \otimes I_4)(R_4 \otimes I_4) \\ &\quad (I_8 \otimes F_2)(I_4 \otimes T_2^4)(I_4 \otimes F_2 \otimes I_2)(I_4 \otimes R_4) \end{aligned}$$

The permutation $R_4 \otimes I_4$ occuring in the middle of this factorization can be moved to the front since $(A \otimes I)(I \otimes B) = (I \otimes B)(A \otimes I)$. The resulting factorization

$$\begin{aligned} &(F_2 \otimes I_4)(T_2^4 \otimes I_4) \cdot (I_2 \otimes F_2 \otimes I_4) \cdot \\ &(I_8 \otimes F_2)(I_4 \otimes T_2^4) \cdot (I_4 \otimes F_2 \otimes I_2) \cdot (R_4 \otimes R_4) \end{aligned}$$

resembles the factorization in the 16-point iterative Cooley-Tukey algorithm exept that the twiddle factor matrices are different.

Similar to how the 1-D Pease algorithm was derived from the iterative Cooley-Tukey factorization, we obtain a Pease algorithm for the 2-D DFT $F_4 \otimes F_4$.

$$\begin{aligned} F_4 \otimes F_4 &= L_2^{16}(I_8 \otimes F_2)T_4 L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3 L_4^{16} \cdot \\ &\quad L_8^{16}(I_8 \otimes F_2)T_2 L_2^{16} \cdot (I_8 \otimes F_2)T_1 \cdot (R_4 \otimes R_4) \end{aligned}$$

where

$$\begin{aligned} T_4 &= L_8^{16}(T_2^4 \otimes I_4)L_2^{16} \\ T_3 &= I_{16} \\ T_2 &= L_2^{16}(I_4 \otimes T_2^4)L_8^{16} \\ T_1 &= I_{16} \end{aligned}$$

Except for the twiddle factor matrices and the initial permutation $(R_4 \otimes R_4)$ the factorization is exactly the same as the Pease algorithm for one-dimensional DFT shown in Theorem 4. $\square$

The same process can be extended to the t-dimensional DFT, $F_{N_1} \otimes \cdots \otimes F_{N_t}$. This leads us to a class of algorithms called "dimensionless FFT" by which any multidimensional DFT of a fixed size can be computed. The following section describes the dimensionless FFT.

2.4 Dimensionless FFT

There exist a class of FFT algorithms called "dimensionless" FFTs which can be used to compute an arbitrary multidimensional DFT of a fixed size [16]. The only change in the algorithm that is necessary when changing dimension is a relabeling of the input and output points and a modification of the values of the twiddle factors.

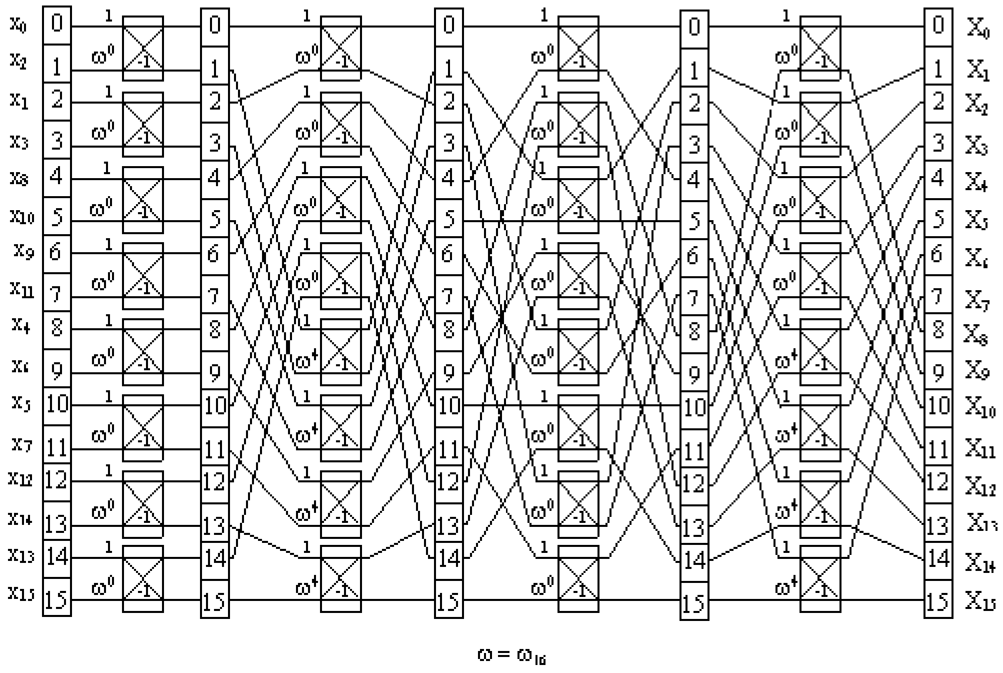


Figure 2.8 Dataflow diagram of the Pease algorithm for computing 2-D (4×4)-point DFT

To illustrate the idea of a dimensionless FFT, let us consider the two-dimensional (4×4)-point DFT described by $F_4 \otimes F_4$. In Section 2.2.2, we introduce the Pease algorithm for one-dimensional 2^n-point DFT. Figure 2.6 shows the dataflow of the one-dimensional 16-point DFT (F_{16}) using the Pease algorithm. The two-dimensional (4×4)-point DFT can be computed using the same dataflow with the exception that (1) the input data is provided in a different order and (2) the twiddle factors are modified. Figure 2.8 shows the dataflow of the Pease alogorithm when the order of input and the twiddle factors are modified for computing the (4×4)-point DFT.

The one-dimensional 16-point Pease algorithm is described by the matrix factorization

$$\begin{aligned} F_{16} &= L_2^{16}(I_8 \otimes F_2)T_4 L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3 L_4^{16} \cdot \\ &\quad L_2^{16}(I_8 \otimes F_2)T_2 L_8^{16} \cdot (I_8 \otimes F_2)T_1 \cdot R_{16} \end{aligned} \tag{2-40}$$

where

$$\begin{aligned} T_1 &= I_{16} = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1), \\ T_2 &= L_2^{16}(I_4 \otimes T_2^4)L_8^{16} = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4), \\ T_3 &= L_4^{16}(I_2 \otimes T_4^8)L_2^{16} = \mathbf{diag}(1,1,1,1,1,\omega_{16}^2,1,\omega_{16}^2,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^6,1,\omega_{16}^6), \\ T_4 &= L_8^{16}T_8^{16}L_2^{16} = \mathbf{diag}(1,1,1,\omega_{16},1,\omega_{16}^2,1,\omega_{16}^3,1,\omega_{16}^4,1,\omega_{16}^5,1,\omega_{16}^6,1,\omega_{16}^7). \end{aligned} \tag{2-41}$$

The two-dimensional (4×4)-point Pease algorithm can be described by the same factorization except that the permutation R_{16} and the twiddle factors T_1, T_2, T_3 and T_4 are different. The following example illustrate the derviation of the two-dimensional (4×4)-point Pease algorithm.

Example:

$$\begin{array}{rcll}
F_4 \otimes F_4 & = & [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4] \otimes [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4] & \text{(Theorem 3)} \\
& = & ([(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)] \otimes I_4) & \text{(Prop. 1.7} \\
& & (I_4 \otimes [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)]) \cdot (R_4 \otimes R_4) & \text{ and 1.8)} \\
& = & (F_2 \otimes I_2 \otimes I_4)(T_2^4 \otimes I_4) \cdot (I_2 \otimes F_2 \otimes I_4) \cdot & \\
& & (I_4 \otimes F_2 \otimes I_2)(I_4 \otimes T_2^4) \cdot (I_4 \otimes I_2 \otimes F_2) \cdot (R_4 \otimes R_4) & \text{(Prop. 1.11)} \\
& = & (F_2 \otimes I_8)(T_2^4 \otimes I_4) \cdot (I_2 \otimes F_2 \otimes I_4) \cdot & \\
& & (I_4 \otimes F_2 \otimes I_2)(I_4 \otimes T_2^4) \cdot (I_8 \otimes F_2) \cdot (R_4 \otimes R_4) & \text{(Prop. 1.13)} \\
& = & L_2^{16}(I_8 \otimes F_2)L_8^{16}(T_2^4 \otimes I_4) \cdot L_4^{16}(I_4 \otimes I_2 \otimes F_2)L_4^{16} \cdot & \\
& & L_8^{16}(I_2 \otimes I_4 \otimes F_2)L_2^{16}(I_4 \otimes T_2^4) \cdot (I_8 \otimes F_2) \cdot (R_4 \otimes R_4) & \text{(Prop. 10.4)} \\
& = & L_2^{16}(I_8 \otimes F_2)T_4 L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3 L_4^{16} \cdot & \\
& & L_8^{16}(I_8 \otimes F_2)T_2 L_2^{16} \cdot (I_8 \otimes F_2)T_1 \cdot (R_4 \otimes R_4) & \text{(Prop. 1.13)}
\end{array}$$

where

$$\begin{array}{rcl}
T_1 I_{16} = I_{16} \Rightarrow T_1 & = & I_{16} = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1), \\
T_2 L_2^{16} = L_2^{16}(I_4 \otimes T_2^4) \Rightarrow T_2 & = & L_2^{16}(I_4 \otimes T_2^4)L_8^{16} \\
& = & \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4), \\
T_3 L_4^{16} = L_4^{16} \Rightarrow T_1 & = & I_{16} = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1), \\
T_4 L_8^{16} = L_8^{16}(T_2^4 \otimes I_4) \Rightarrow T_2 & = & L_8^{16}(T_2^4 \otimes I_2)L_2^{16} = L_2^{16}(I_4 \otimes T_2^4)L_8^{16} \\
& = & \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4),
\end{array} \tag{2-42}$$

Notice that we achieve the same form of matrix factorization as those for F_{16}. The only differences are that R_{16} is replaced by $R_4 \otimes R_4$ and the twiddle factor matrices in Equation 2-41 are replaced by the above twiddle factor matrices. □

The dimensionless FFT generalizes this computation to an arbitrary multidimensional DFT.

Theorem 6 (Dimensionless FFT Algorithm). *Let $\mathcal{F}_N$ denote an arbitrary multidimensional DFT of size $N = 2^n$. There exist a fixed set of permutations P_i,*

$i = 1, \ldots, n$, such that for any factorization $N = N_1 \times \ldots \times N_t$, there exist an initial permutation P_0, and diagonal matrices T_i, $i = 1, \ldots, n$ such that

$$\begin{aligned}\mathcal{F}_N &= F_{N_1} \otimes \cdots \otimes F_{N_t} \\ &= \left\{ \prod_{i=0}^{n-1} P_{n-i}^{-1} (I_{2^{n-1}} \otimes F_2) T_{t-i} P_{n-i} \right\} P_0. \qquad (2\text{-}43)\end{aligned}$$

The "internal permutation" $P_i, i = 1, \ldots, n$ remain fixed independent of the dimension, and consequently define a dimensionless FFT. *Only the initial permutation P_0 and the twiddle factors depend on the dimension t.*

Proof: *The existence of a dimensionless FFT follows from a calculation similar to the one in the beginning of this section. A complete description, including proofs, of these algorithms can be found in [16].* □

There are many possible dimensionless FFT algorithms. The one-dimensional Cooley-Tukey algorithm can be extended so that it becomes dimensionless.

Theorem 7 (Dimensionless Iterative Cooley-Tukey Algorithm). *Let $N = N_1 \times \cdots \times N_t = 2^n$, where $N_k = 2^{n_k}$, $1 \leq k \leq t$, and $n = \sum_{k=1}^{t} n_k$. Then, the following factorization describes the dimensionless Cooley-Tukey algorithm.*

$$\begin{aligned}\mathcal{F}_N &= F_{N_1} \otimes \cdots \otimes F_{N_t} \\ &= \left\{ \prod_{i=0}^{n-1} (I_{2^i} \otimes F_2 \otimes I_{2^{n-i-1}}) T_{n-i} \right\} P_0, \qquad (2\text{-}44)\end{aligned}$$

where P_0 and T_{n-i}, $0 \leq i < n$, depend on the dimension specified by $n_1, \cdots, n_t$.

Let k be an index counting from 1 to t and for every value of k, let j be an index counting from 0 to $n_k - 1$ and let d be a function defined as

$$d(k) = n_1 + \cdots + n_{k-1} = \sum_{j=1}^{k-1} n_j. \qquad (2\text{-}45)$$

Then, the index i in Equation 2-44 can be written as

$$i \;=\; d(k) - n_k + j,$$

and the twiddle factors T_{n-i} and the intial permutation P_0 in Equation 2-44 are defined by

$$T_{n-i} \;=\; I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{n-d(k)}} \tag{2-46}$$

$$P_0 \;=\; R_{2^{n_{t-1}}} \otimes \cdots \otimes R_{2^{n_0}} \tag{2-47}$$

Proof: *This can be proved by substituting the iterative Cooley-Tukey factorization (Theorem 3) into the factors of the multidimensional row-column factorization (Theorem 5) and then applying some simple tensor product manipulations. These steps are outlined below.*

1. *Factor $F_{N_1} \otimes \cdots \otimes F_{N_t}$ to the general case of the row-column factorization.*

$$F_{N_1} \otimes \cdots \otimes F_{N_t} = \prod_{k=1}^{t} (I_{2^{d(k)-n_k}} \otimes F_{2^{n_k}} \otimes I_{2^{n-d(k)}})$$

For example, for $t = 3$, we have

$$\begin{aligned} F_{N_1} \otimes F_{N_2} \otimes F_{N_3} &= (F_{2^{n_1}} \otimes I_{2^{n_2}} \otimes I_{2^{n_3}})(I_{2^{n_1}} \otimes F_{2^{n_2}} \otimes I_{2^{n_3}})(I_{2^{n_1}} \otimes I_{2^{n_2}} \otimes F_{2^{n_3}}) \\ &= (F_{2^{n_1}} \otimes I_{2^{n_2+n_3}})(I_{2^{n_1}} \otimes F_{2^{n_2}} \otimes I_{2^{n_3}})(I_{2^{n_1+n_2}} \otimes F_{2^{n_3}}) \\ &= (I_{2^{d(1)-n_1}} \otimes F_{2^{n_1}} \otimes I_{2^{n-d(1)}})(I_{2^{d(2)-n_2}} \otimes F_{2^{n_2}} \otimes I_{2^{n-d(2)}}) \\ &\quad (I_{2^{d(3)-n_3}} \otimes F_{2^{n_3}} \otimes I_{2^{n-d(3)}}) \end{aligned}$$

2. *Replace $F_{2^{n_k}}$ with the iterative Cooley-Tukey factorization. This results in the following factorization.*

$$\mathcal{F}_N = \prod_{k=1}^{t} (I_{2^{d(k)-n_k}} \otimes \left\{ \prod_{j=0}^{n_k-1} (I_{2^j} \otimes F_2 \otimes I_{2^{n_k-j-1}})(I_{2^j} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}}) \right\} R_{N_k} \otimes I_{2^{n-d(k)}})$$

3. Apply necessary tensor product properties to obtain Equation 2-44.

□

Following the derivation of the Pease algorithm from the iterative Cooley-Tukey algorithm, we obtain the dimensionless Pease algorithm from the dimensionless Cooley-Tukey.

Theorem 8 (Dimensionless Pease Algorithm). *Let $N = N_1 \times \cdots \times N_t = 2^n$, where $N_k = 2^{n_k}$, $0 \leq k \leq t$, and $n = \sum_{k=1}^{t} n_k$. Then, the dimensionless Pease algorithm is described by*

$$\mathcal{F}_N = \left\{ \prod_{i=0}^{n-1} L^{2^n}_{2^{i+1}} (I_{2^{n-1}} \otimes F_2) T_{n-i} L^{2^n}_{2^{n-i-1}} \right\} P_0, \tag{2-48}$$

where P_0 is defined in Equation 2-47, and T_{n-i}, $0 \leq i < n$, depending on the dimension specification $(n_1, \cdots, n_t)$ is defined as the following.

$$T_{n-i} = L^{2^n}_{2^{n-i-1}} (I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{n-d(k)}}) L^{2^n}_{2^{i+1}}, \tag{2-49}$$

where k is an index counting from 1 to t, j is an index counting from 0 to $n_k - 1$, $d(k)$ is the function defined in Equation 2-45 and $i = d(k) - n_k + j$.

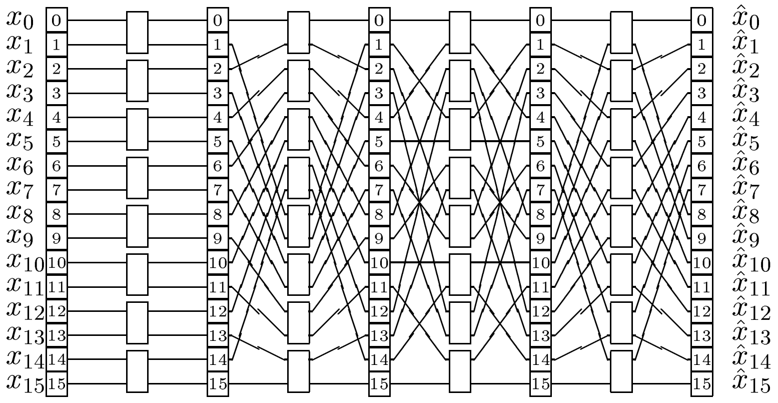
Proof: *This follows from Theorem 7 in exactly the same way that Theorem 4 was proved.* □

Using definitions and properties in Section 2.1.5, we obtain the twiddle factor matrices of the Pease algorithm in a simple form paramertized by the dimension specfication.

Let $p = 2^{n_k-j-1}$, $q = 2^{n-d(k)}$, $r = 2^i = 2^{d(k)-n_k+j}$ and $2^n = 2pqr$. Then, applying basis vector to both sides of Equation 2-49, we have

$$\begin{aligned}
&L^{2^n}_{2^{n-i-1}}(I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{n-d(k)}})L^{2^n}_{2^{i+1}}(\mathbf{e}^p_a \otimes \mathbf{e}^q_b \otimes \mathbf{e}^r_c \otimes \mathbf{e}^2_d)\\
&= L^{2^n}_{2^{n-i-1}}(I_r \otimes T^{2p}_p \otimes I_q)(\mathbf{e}^r_c \otimes \mathbf{e}^2_d \otimes \mathbf{e}^p_a \otimes \mathbf{e}^q_b)\\
&= L^{2^n}_{2^{n-i-1}}(\mathbf{e}^r_c \otimes T^{2p}_p(\mathbf{e}^2_d \otimes \mathbf{e}^p_a) \otimes \mathbf{e}^q_b)\\
&= \omega^{a\cdot d}_{2p} L^{2^n}_{2^{n-i-1}}(\mathbf{e}^r_c \otimes \mathbf{e}^2_d \otimes \mathbf{e}^p_a \otimes \mathbf{e}^q_b)\\
&= \omega^{a\cdot d}_{2p}(\mathbf{e}^p_a \otimes \mathbf{e}^q_b \otimes \mathbf{e}^r_c \otimes \mathbf{e}^2_d)
\end{aligned}$$

$$T_{n-i} = \bigoplus_{a=0}^{p-1}\bigoplus_{b=0}^{q-1}\bigoplus_{c=0}^{r-1}\bigoplus_{d=0}^{1} \omega^{a\cdot d}_{2p};\ p = 2^{n_k-j-1}, q = 2^{n-d(k)},\ \text{and } r = 2^i \tag{2-50}$$



x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
$\hat{x}_0$ $\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$ $\hat{x}_7$ $\hat{x}_8$ $\hat{x}_9$ $\hat{x}_{10}$ $\hat{x}_{11}$ $\hat{x}_{12}$ $\hat{x}_{13}$ $\hat{x}_{14}$ $\hat{x}_{15}$

Figure 2.9 Dataflow diagram of the dimensionless Pease algorithm of size 16 points

Example: To illustrate the dimensionless FFT, let us consider $N = 16 = 2^4$. A Fourier transform on 16 points can have dimensions equal to 1 (F_{16}), 2 ($F_2 \otimes F_8$, $F_4 \otimes F_4$, and $F_8 \otimes F_2$), 3 ($F_2 \otimes F_2 \otimes F_4$, $F_2 \otimes F_4 \otimes F_2$, and $F_4 \otimes F_2 \otimes F_4$), or 4 ($F_2 \otimes F_2 \otimes F_2 \otimes F_2$). According to the dimensionless FFT theorem, all of these transforms can be computed using the same algorithm with adaptive twiddle factors and initial permutation. Let us choose the dimensionless Pease algorithm (Theorem 8). Then, the following factorization describes the dimensionless Pease algorithm of size 16.

$$L_2^{16}(I_8 \otimes F_2)T_4L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3L_4^{16} \cdot L_2^{16}(I_8 \otimes F_2)T_2L_8^{16} \cdot (I_8 \otimes F_2)T_1 \cdot P_0$$

In order to compute different dimensions of 16-point Fourier transform using the Pease algorithm, we need to change only the initial permutation, P_0, and the twiddle factors, T_1, T_2, T_3 and T_4.

In Section 2.2 and the previous example, we have shown that there exist the twiddle factors for computing F_{16} and $F_4 \otimes F_4$ (see Equation 2-41 and 2-42). Here we demonstrate how the twiddle factors of a specific dimension can be computed from Theorem 8.

For three-dimensional $(2 \times 4 \times 2)$-point DFT, where

$$\begin{aligned} n_1 &= 1, n_2 = 2, n_3 = 1, \\ d(1) &= n_1 = 1, d(2) = n_1 + n_2 = 3, \text{ and } d(3) = n_3 = n_1 + n_2 + n_3 = 4, \end{aligned}$$

we have

$$\begin{aligned} k = 1, j = 0, i = 0, \Rightarrow T_4 &= L_8^{16}(I_1 \otimes T_1^2)L_2^{16} = I_{16} \\ &= \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1) \\ k = 2, j = 0, i = 1, \Rightarrow T_3 &= L_4^{16}(I_2 \otimes T_2^4 \otimes I_2)L_4^{16} = \bigoplus_{a=0}^{1}\bigoplus_{b=0}^{1}\bigoplus_{c=0}^{1}\bigoplus_{d=0}^{1} \omega_4^{a\cdot d} \\ &= \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_4,1,\omega_4,1,\omega_4,1,\omega_4) \\ k = 2, j = 1, i = 2, \Rightarrow T_2 &= L_8^{16}(I_4 \otimes T_1^2 \otimes I_2)L_2^{16} = I_{16} \\ &= \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1) \\ k = 3, j = 0, i = 3, \Rightarrow T_1 &= L_{16}^{16}(I_8 \otimes T_1^2)L_1^{16} = I_{16} \\ &= \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1) \end{aligned}$$

Table 2.1 Initial permutation and twiddle factors for computing all possible dimensional DFT of size 16 points using the dimensionless Pease algorithm

1-D	$P_0 = R_{16}$
F_{16}	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,\omega_{16}^2,1,\omega_{16}^2,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^6,1,\omega_{16}^6)$
	$T_4 = \mathbf{diag}(1,1,1,\omega_{16},1,\omega_{16}^2,1,\omega_{16}^3,1,\omega_{16}^4,1,\omega_{16}^5,1,\omega_{16}^6,1,\omega_{16}^7)$
2-D	$P_0 = R_2 \otimes R_8$
$F_2 \otimes F_8$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,\omega_{16}^2,1,\omega_{16}^2,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^6,1,\omega_{16}^6)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
2-D	$P_0 = R_4 \otimes R_4$
$F_4 \otimes F_4$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
2-D	$P_0 = R_8 \otimes R_2$
$F_8 \otimes F_2$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,\omega_{16}^2,1,\omega_{16}^2,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^6,1,\omega_{16}^6)$
3-D	$P_0 = R_2 \otimes R_2 \otimes R_4$
$F_2 \otimes F_2 \otimes F_4$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
3-D	$P_0 = R_2 \otimes R_4 \otimes R_2$
$F_2 \otimes F_4 \otimes F_2$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
3-D	$P_0 = R_4 \otimes R_2 \otimes R_2$
$F_4 \otimes F_2 \otimes F_2$	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
4-D	$P_0 = R_2 \otimes R_2 \otimes R_2 \otimes R_2 = I_{16}$
$F_2 \otimes F_2 \otimes F_2 \otimes F_2$	$T_i = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$ for $i = 1,2,3,4$

Applying the theorem to all combinations of $2^4 = N_1 \times \cdots \times N_t$, we obtain the twiddle factors for computing the corresponding multidimensional DFT using the dimensionless Pease algorithm. Table 2.1 summarizes all configurations of the initial permutation , P_0, and the twiddle factors, T_1, T_2, T_3 and T_4, for computing all possible multidimensional DFT of size 16 points.

Figure 2.9 shows the dataflow of the 16-point dimensionless Pease FFT, where the input vector and the twiddle factors are configured according to the specified dimension. $\square$

2.5 FFT Dataflow

This section introduces the space of dimensionless FFT algorithms that we considered for implementation on our processor. We remark that it can be shown [22] that the space of algorithms that we consider are the only possible algorithms compatible with our framework.

The dimensionless FFT computes any multidimensional DFT of a fixed size using the same dataflow. The only modifications when changing the dimension are the order of the input data and the twiddle factors. Following Theorem 6, a dimensionless FFT algorithm is specified by three sets of matrices: the internal permutations P_i, $1 \leq i \leq n$, the initial permutation P_0 and the twiddle factors T_i, $1 \leq i \leq n$. The internal permutations are independent of the dimension while the initial permutation P_0 and the twiddle factor T_i depend on the dimension specification. Assume that the twiddle factors are generated or pre-stored and that the initial permutation is performed when the data is loaded. Then, the cost of an FFT computation is essentially independent of dimension. In particular, the dataflow is defined solely by the internal permutations, which are independent of dimension. The internal permutations define what we call an *FFT dataflow*, and the space of dimensionless FFT algorithms is classified by determining the set of possible dataflows.

Definition 16 (FFT Dataflow). *A dimensionless FFT has a fixed set of internal permutations, P_i, $i = 1, \ldots, n$. A set of permutations is called an FFT dataflow [22] if it is possible to find an initial permutation, P_0, and a set of twiddle factor matrices, T_i, $i = 1, \cdots, n$, such that the resulting factorization in Equation 2-43 computes all of the possible multidimensional DFTs for a given number of points.*

Example: The set of internal permutations $P_i = L^{16}_{2^i}$, $1 \leq i \leq 4$ defines the dataflow of the dimensionless Pease algorithms of size 16 points. This is because following Theorem 8, we can configure the initial permutation P_0 and the twiddle factors T_i, $1 \leq i \leq 4$, such that the factorization in Equation 2-43 is true for any multidimensional DFT of size 16 points. Table 2.1 gathers such configurations.

In general, the permutation $P_i = L^{2^n}_{2^i}$, $i = 1, \cdots, n$, defines the FFT dataflow of the Pease algorithm for computing all multidimensional DFT of size 2^n points. $\square$

In this thesis, we consider a class of dimensionless FFT algorithms obtained from the Pease dimensionless FFT by permuting the butterfly operations (the computation boxes in a FFT dataflow) in the stages of the Pease factorization. The following example illustrates the mathematical computations that carry out this process.

Example: Let us consider a dimensionless FFT algorithm derived from the dimensionless Pease algorithm of size 16 points as following.

We permute the 8 copies of F_2 in each stage of the Pease algorithm with the stride permutation $L^8_{2^{s(i)}}$ by conjugating the $(I_8 \otimes F_2)$ with a permutation of the form $(L^8_{2^{s(i)}} \otimes I_2)$, $0 \leq s(i) < 3$; that is the term $L^{16}_{2^{i+1}}(I_8 \otimes F_2)T_{n-i}L^{16}_{2^{4-i-1}}$ in each stage becomes

$$\begin{aligned}
&L^{16}_{2^{i+1}}(I_8 \otimes F_2)T_{4-i}L^{16}_{2^{4-i-1}}\\
&\quad = L^{16}_{2^{i+1}}(L^8_{2^{3-s(i)}} \otimes I_2)(I_8 \otimes F_2)(L^8_{2^{s(i)}} \otimes I_2)T_{4-i}L^{16}_{2^{4-i-1}}\\
&\quad = L^{16}_{2^{i+1}}(L^8_{2^{3-s(i)}} \otimes I_2)(I_8 \otimes F_2)T'_{4-i}(L^8_{2^{s(i)}} \otimes I_2)L^{16}_{2^{4-i-1}}\\
&\quad = P^{-1}_{4-i}(I_8 \otimes F_2)T'_{4-i}P_{4-i}
\end{aligned}$$

where T_{4-i} is the twiddle factor of the dimensionless Pease algorithm of size 16 point at stage $4-i$,

$$\begin{aligned}
P_{4-i} &= (L^8_{2^{s(i)}} \otimes I_2)L^{16}_{2^{4-i-1}}, \quad P^{-1}_{4-i} = L^{16}_{2^{i+1}}(L^8_{2^{3-s(i)}} \otimes I_2)\\
T'_{4-i}(L^8_{2^{s(i)}} \otimes I_2) &= (L^8_{2^{s(i)}} \otimes I_2)T_{4-i} \Rightarrow T'_{4-i} = (L^8_{2^{s(i)}} \otimes I_2)T_{4-i}(L^8_{2^{3-s(i)}} \otimes I_2).
\end{aligned}$$

Replacing T_{4-i} from Equation 2-49, we have

$$\begin{aligned}
T'_{4-i} &= (L^8_{2^{s(i)}} \otimes I_2)L^{16}_{2^{n-i-1}}(I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{4-d(k)}})L^{16}_{2^{i+1}}(L^8_{2^{3-s(i)}} \otimes I_2)\\
&= P_{4-i}(I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{4-d(k)}})P^{-1}_{4-i}
\end{aligned}$$

Setting $s(0) = s(1) = 2$, $s(2) = 0$ and $s(3) = 1$, we obtains the dimensionless FFT

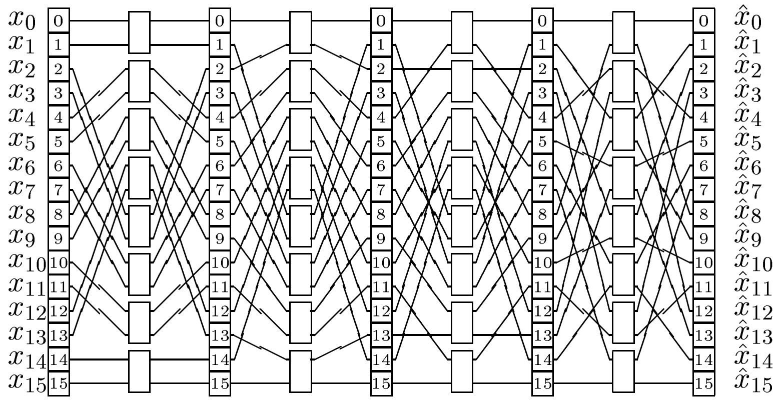

x_0 x_1 x_2 x_3 x_4 x_5 x_6 x_7 x_8 x_9 x_{10} x_{11} x_{12} x_{13} x_{14} x_{15}
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15
$\hat{x}_0$ $\hat{x}_1$ $\hat{x}_2$ $\hat{x}_3$ $\hat{x}_4$ $\hat{x}_5$ $\hat{x}_6$ $\hat{x}_7$ $\hat{x}_8$ $\hat{x}_9$ $\hat{x}_{10}$ $\hat{x}_{11}$ $\hat{x}_{12}$ $\hat{x}_{13}$ $\hat{x}_{14}$ $\hat{x}_{15}$

Figure 2.10 Dataflow diagram of an alternative FFT algorithm described by internal permutations in Equation 2-51

Table 2.2 Initial permutation, P_0, and twiddle factors for computing 1-D 16-point DFT (F_{16}), 2-D (4×4)-point DFT ($F_4 \otimes F_4$) and 3-D (2×2×4)-point DFT ($F_2 \otimes F_2 \otimes F_4$) using an alternative algorithm specified by permutations in Equation 2-51.

1D F_{16}	$P_0 = R_{16}$
	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_4,1,\omega_4,1,\omega_4,1,\omega_4)$
	$T_3 = \mathbf{diag}(1,1,1,\omega_8,1,1,1,\omega_8,1,\omega_8^2,1,\omega_8^2,1,\omega_8^3,1,\omega_8^3)$
	$T_4 = \mathbf{diag}(1,1,1,\omega_4,1,\omega_{16},1,\omega_{16}^5,1,\omega_{16}^2,1,\omega_{16}^6,1,\omega_{16}^3,1,\omega_{16}^7)$
2D $F_4 \otimes F_4$	$P_0 = R_4 \otimes R_4$
	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
3D $F_2 \otimes F_2 \otimes F_4$	$P_0 = R_2 \otimes R_2 \otimes R_4$
	$T_1 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_2 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4,1,\omega_{16}^4)$
	$T_3 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$
	$T_4 = \mathbf{diag}(1,1,1,1,1,1,1,1,1,1,1,1,1,1,1,1)$

algorithm described by

$$\mathcal{F}_{16} = P_4^{-1}(I_8 \otimes F_2)T_4'P_4P_3^{-1}(I_8 \otimes F_2)T_3'P_3P_2^{-1}(I_8 \otimes F_2)T_2'P_2P_1^{-1}(I_8 \otimes F_2)T_1' \cdot P_0 \tag{2-51}$$

where

$$\begin{array}{ll} P_1 = (L_2^8 \otimes I_2) & P_2 = L_2^{16} \\ P_3 = (L_4^8 \otimes I_2)L_4^{16} & P_4 = (L_4^8 \otimes I_2)L_8^{16} \end{array}$$

Figure 2.10 shows the dataflow of the algorithm. Note that at stage 1, 2, 3, and 4 the algorithm permutes the computation boxes of the Pease algorithm by L_2^8, I_8, L_4^8 and L_4^8 respectively. The initial permutations and the twiddle factors for computing 1-D F_{16}, 2-D $F_4 \otimes F_4$ and 3-D $F_2 \otimes F_2 \otimes F_4$ using dimensionless FFT described by Equation 2-51 are shown in Table 2.2. □

The following theorem presents a class of dimensionless FFT algorithms that are derived from the Pease algorithm simply by relabeling the butterfly operations in each stage.

Theorem 9 (Dimensionless Pease Based Algorithms). *Let $N = N_t \times \cdots \times N_1 = 2^n$, where $N_k = 2^{n_k}$, $0 \leq k \leq t$, and $n = \sum_{k=1}^{n_k}$. Then, the following factorization describes a dimensionless FFT algorithm.*

$$\begin{aligned} \mathcal{F}_N &= F_{2^{n_1}} \otimes \ldots \otimes F_{2^{n_t}} \\ &= \left\{ \prod_{i=1}^{t} P_{n-i}^{-1}(I_{2^{n-1}} \otimes F_2) T_{n-i} P_{n-i} \right\} P_0 \end{aligned} \tag{2-52}$$

where the initial permutation P_0 is defined in Equation 2-47.

$$P_{n-i} = (Q_{n-i} \otimes I_2) L^{2^n}_{2^{n-i-1}}, \tag{2-53}$$

$$P_{n-i}^{-1} = L^{2^n}_{2^{i+1}}(Q_{n-i}^{-1} \otimes I_2), \tag{2-54}$$

where Q_{n-i} is an arbitrary permutation of size $2^{n-1} \times 2^{n-1}$.

$$T_{n-i} = P_{n-i}(I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{n-d(k)}}) P_{n-i}^{-1}, \tag{2-55}$$

The number k is an index counting from 1 to t, j is an index counting from 0 to $n_k - 1$, $d(k) = n_1 + \cdots + n_k$ and $i = d(k) - n_k + j$.

Proof: *Substitute $(Q_{n-1}^{-1} \otimes I_2)(I_{2^{n-1}} \otimes F_2)(Q_{n-i} \otimes I_2)$ into the dimensionless Pease factorization (8). Let T'_{n-i} denote the twiddle factor occuring in the Pease algorithm. Then, since that $(Q_{n-i} \otimes I_2)T'_{n-i} = T_{n-i}(Q_{n-i} \otimes I_2)$, the theorem is proved.* $\square$

We remark [22] that if we assume that the permutations in an FFT dataflow are tensor permutations, then the only FFT dataflows are those in Theorem 9. In general the number of FFT dataflows defined in Theorem 9 of size $N = 2^n$ is the number of sequences of permutations $P_1, \ldots, P_n$. Since there are $(N/2)!$ choices for Q_i, there are

$(N/2)!^n$ possible dataflows corresponding to $(N/2)!^n$ dimensionless FFT algorithms. For our design we only consider FFT dataflows built using tensor permutations. Since there are $(n-1)!$ of tensor permutation matrices of size $2^{n-1} \times 2^{n-1}$, the number of such dataflows is equal to $[(n-1)!]^n$. For example, with the restriction that the permutations are tensor permutation the number of dimensionless FFT algorithms for $n = 4$ is equal to $(3!)^4 = 1296$.

The goal of this thesis is to find the algorithm from this space of dimensionless FFTs that has the optimal performance. In the following chapter we define the architecture model we will consider and provide a mechanism to map any of the dimensionless FFT algorithms discussed in this chapter to the architecture. We then provide a performance model for our architecture, and finally, using the mapping procedure and performance model, we systematically search for the optimal design by searching for the algorithm that has the best performance.

3.0 FFT PROCESSOR AND MAPPING METHODOLOGY

In the previous chapter, we described the class of dimensionless FFT algorithms. Each algorithm in this class has a fixed dataflow which is independent of the dimension of the FFT that is computed. However, each algorithm in the space of dimensionless FFTs has a different dataflow and consequently may lead to different performance. In addition we presented the mathematical tools needed to manipulate and implement these algorithms.

In this chapter we present an architecture model and derive a mapping from the space of algorithms in the previous chapter to this architecture. Using the mapping we can systematically explore design tradeoffs and can search for the optimal implementation of the FFT for the given architecture.

The main point of this thesis is to explore techniques for systematically implementing and optimizing special-purpose hardware designed to accelerate a collection of algorithms. The chosen architectural model is not so important. What is important is that the model be flexible enough that different mathematical expressions for the FFT can be utilized to explore performance tradeoffs. Details of the hardware implementation should be delayed so that algorithmic information can be included in the design process and so that information obtained from high-level performance modeling can influence the final design.

For this study we have chosen a distributed memory processor. This choice highlights the importance of dataflow and benefits from the mathematical exploration of the FFT presented in the previous chapter. The process of mapping an FFT formula to the architecture amounts to allocating a sequence of tasks (butterfly operations) to the processing elements. The permutations in the formula control the flow of data through the processor. Different sequences of permutations correspond to different memory access patterns and consequently lead to different performance. Studying the dataflow patterns is enough to assist in the choice of the optimal algorithm (see the following chapter); however, a detailed implementation requires the development of hardware to compute addresses needed to access data and to compute twiddle factors needed for the butterfly operations. In this chapter we describe, in detail, how to compute addresses and twiddle factors from a given formula. The derivation of the actual hardware is presented in Chapter 5.

This chapter is organized as follows. The architecture model is introduced in Section 3.1. The architecture consists of a collection of processors and memories and an interconnection network. Section 3.2 describes the mapping from an FFT formula onto the chosen architecture. The mapping process consists of loading the data using the labeling corresponding to the initial permutation in the formula and then mapping the sequence of butterfly operations contained in the remaining stages of the formula Section 3.2.1 describes the loading process. Section 3.2.2 defines an FFT task and explains how an FFT formula is translated into a sequence of tasks. The FFT tasks are deterministically scheduled to processor elements. Section 3.2.3 describes the scheduling process. This schedule is incorporated into an address generating unit which is described in Section 3.2.4. Finally, Section 3.2.5 shows how to obtain the sequence of twiddle factors needed to correctly compute the scheduled butterfly operations. Twiddle factor computation is performed in a twiddle factor unit which is incorporated into the computation unit where butterfly operations are performed. The twiddle factor computation must take into account the dimension, which is determined from runtime parameters.

3.1 Distributed Memory Architecture

Although there are many possible architectures, we will choose only one architecture to demonstrate or design methodology. It is important to note that we do not claim that the chosen architecture will produce the best FFT processor. We focus on finding the design that will produce the best performance on the chosen architecture.

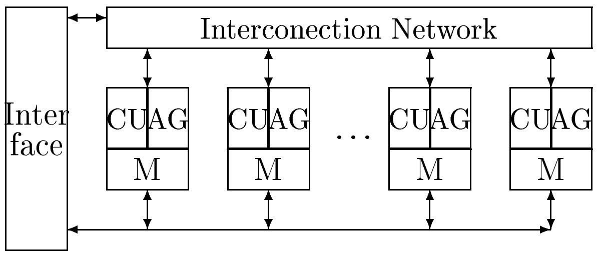


Figure 3.1 The architecture

The proposed architecture shown in Figure 3.1 is a simple distributed-memory architecture containing 3 main units: interface, interconnection network, and processor elements (PEs.)

The interface unit is used as a means to transfer run-time parameters and data to/from the system. The reconfigurable interconnection network provides the communication between the PEs. Each PE contains three main units: memory (M), a computation unit (CU), and an address generator (AG). They are deterministically coupled; i.e. each PE has its own “schedule” that is deterministically mapped from the algorithms.

At this level of the design process, we want an architecture framework on which the algorithm can be executed and their performance evaluated, yet we want to leave open the ability to investigate different designs. Moreover, we want to be able to parameterize the designs by the choice of algorithm. The following subsections describes each part of architecture.

3.1.1 Interconnection Network

Abstractly, the interconnection networks job is to transfer data between PEs. Although there are many possible implementation of the interconnection network, we do not need to specify it in this level of design. However, the interconnection network must be able to (1) distribute the runtime parameters to PEs and input data to memory module, and (2) transfer data between two PEs.

The parameters and the input data are sent to PEs from the interface unit. Therefore, the interconnection network must be able to broadcast these data to the PEs. At this point, the transferring of data between any PEs is assumed to be possible. However, only necessary network configurations will be implemented. Later we will show that the network configurations for the optimal algorithm are simple. The only protocol at this point is that the sender is the one that requests for transferring the data. This requires that the target of the data are specified.

3.1.2 Processor Element and Memory

Abstractly, a processor element's job is to compute a sequence of pre-scheduled tasks. This requires (1) a hardwired scheduler, (2) a computation unit that can compute the necessary tasks and (3) a controller for handling data transfer. We divide a processor element into two units called the address generator (AG) and the computation unit (CU).

A computation unit computes an operation by assuming that its input data are scheduled. For computing the FFT, we choose the two-point butterfly operation and the twiddle factor computation as the primitive operation of the computation unit. The inputs of the operation are two input data and a fraction for computing the twiddle factor. The twiddle fractions depend on the FFT algorithm and the dimension of the FFT, and can be computed using hardwired control based on the algorithm and runtime parameters specifying the dimension.

Although the input data change, their addresses are fixed given a dataflow. The address generator job is to (1) generate the addresses, (2) get the input data, and (3) store back the result. The generation of addresses are local and deterministically coupled with other PEs; i.e. although the addresses generated in each PE follows the same dataflow, they are generated independently. The retrieving and storing of data involve either local or remote memory accesses.

Following the abstract, we propose a PE architecture shown in Figure 3.2. As explained earlier, the CU contains a butterfly operation and the twiddle fraction generator which generate the sequence of twiddle fractions parameterized during runtime by dimension and during compile time by the choice of FFT algorithm. The AG includes buffers, a data control unit, and the address generation unit. The address generation unit generates a sequence of tasks described in term of local addresses, memory identification number (MID), and processor identification number (PID). The data control unit uses these parameters for scheduling the data to/from the memory and to/from the CU.

In this architecture, schedules are viewed by two perspectives. From the perspective of a PE, its schedule is the sequence of data from any memory modules provided in the input buffers. We assume that the data from memory module $\mathbf{M}_i$ that are scheduled to $\mathbf{PE}_j$ are stored in the input buffers in the right order. Therefore, from the perspective of a PE, the schedule is a sequence of “source” MID. The data control unit use the source MID for sequencing the input data to the butterfly operations.

From the perspective of a memory module, a schedule is a sequence of data going out (the input data denoted by ‘X’) and coming into (the result denoted by Y) the memory. The sequence of data going out from the memory is specified by the sequence of local addresses and “target” PID. The data control unit uses the addresses to read the data from the memory and uses the target PID to send the data to the PE that will operate on the data. If the data is local, it is stored in the “local X” FIFO; otherwise, it is sent to its target PE via the interconnection network. The same sequence of local addresses and target PID is used again for writing back the result to the memory.

Since both schedules for PE and for memory follows the same dataflow, we implicitly synchronize the two schedules. The advantage is that we do not have to send the addresses when accessing remote memory.

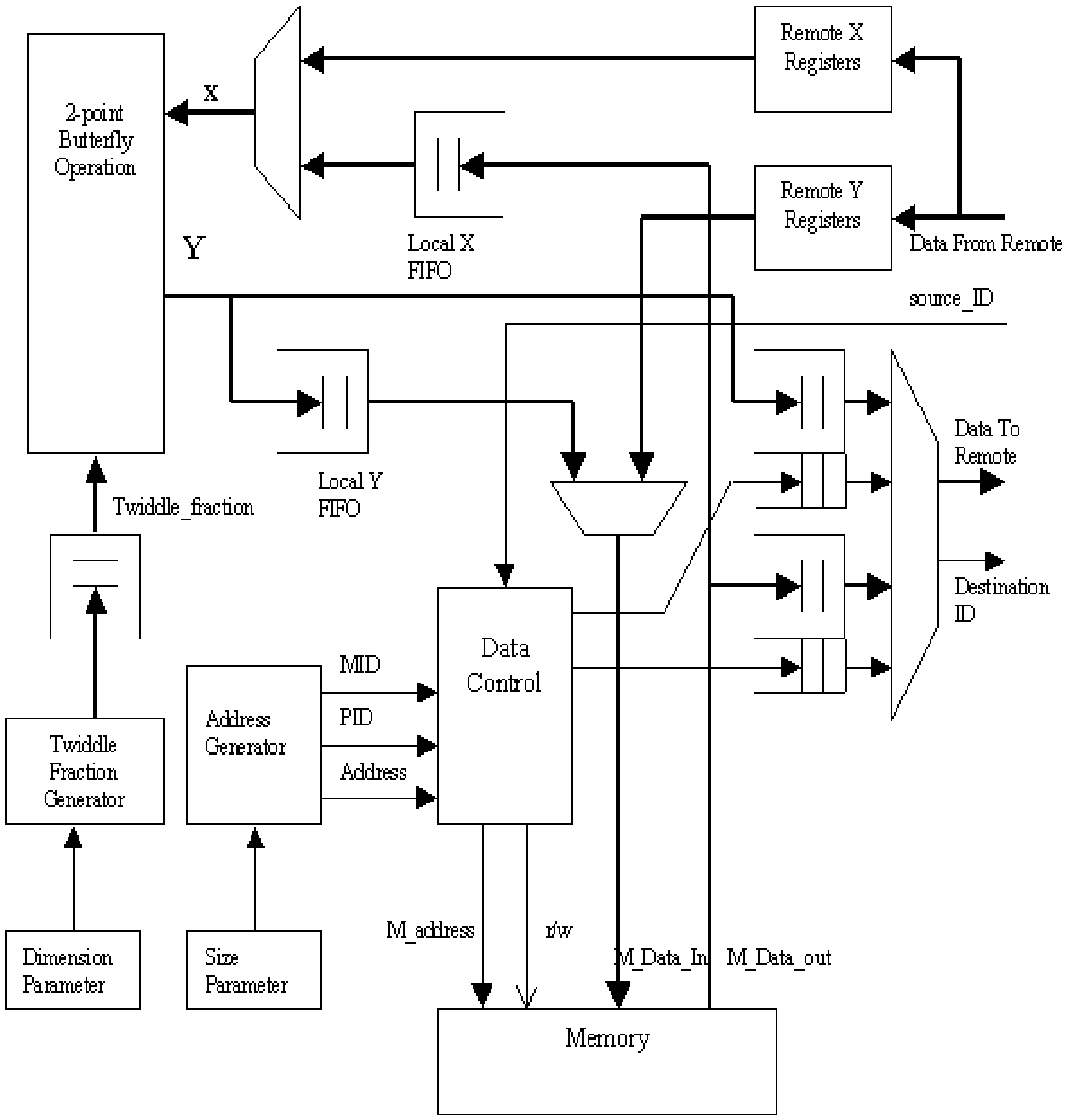


Figure 3.2 Processor element architecture

3.2 Mapping FFT Formulas to the FFT Processor

In the previous chapter, FFT algorithms were described by mathematical formulas. Specifically, an FFT algorithm is specified by three sets of matrices: the initial permutation P_0, the internal permutations P_i, $1 \leq i \leq n$ and the twiddle factor matrices, T_i, $1 \leq i \leq n$. In this section, we explain how an FFT formula is mapped to the proposed architecture.

The initial permutation is performed during the loading of the input data and consequently is mapped to the I/O interface unit. Since the architecture has distributed memory components, the interface unit must distribute the data to the appropriate memory unit. The initial permutation is incorporated into this distribution phase. After the data has been distributed, the remaining stages in the formula are converted to a sequence of butterfly tasks that are scheduled to the PEs. Since the schedule is known at design time, the generation of tasks can be hardwired. Moreover, the memory units can be configured to forward the data elements needed for the tasks to the appropriate PEs without having to explicitly receive address tags from the PEs. The data needed for a particular task may be stored locally or may reside in a remote memory. If the data is remote, it must be forwarded by the appropriate memory unit over the interconnection network.

In the following sections we discuss how data is allocated to memories and how to interpret addresses in terms of a memory identifier and a local offset. Once the addressing convention is fixed, different permutations can be interpreted as address sequences with different memory access patterns. We then discuss how the initial permutation is performed and how the sequence of task addresses are determined by the internal permutations. In addition to address information a butterfly task must contain the twiddle factor needed to actually perform the butterfly operation. These twiddle factors are determined from the twiddle factor matrices contained in the FFT formula. The initial distribution of data and the determination of the addresses and twiddle factors for the tasks specified in an FFT formula constitutes the mapping from the formula to the processor.

3.2.1 Input Loading

In the proposed architecture, the data is distributed equally amongst the processor memories, and the initial permutation P_0 will be performed during the loading of input data to memory. Let $M = 2^m$ be number of processor elements and $N = 2^n$ be number of points. Then, each memory module stores $\frac{N}{M} = 2^{n-m}$ data points. Let data points be addressed from 0 to $N-1$ represented by the binary number $(b_{n-1}b_{n-2}\cdots b_0)_2$.

Let σ_0 be a permutation of degree n specifying P_0. Then, the sequence of input data permuted by P_0 can be generated by permuting $(b_{n-1}b_{n-2}\cdots b_0)_2$ with σ_0 while counting $(b_{n-1}b_{n-2}\cdots b_0)_2$.

$$(b_{n-1}b_{n-2}\cdots b_0)_2 \overset{\sigma_0}{\rightarrow} (b_{\sigma_0^{-1}(n-1)}b_{\sigma_0^{-1}(n-2)}\cdots b_{\sigma_0^{-1}(0)})_2$$

The most significant bits of the permuted address bits, $(b_{\sigma_0(n-1)}\cdots b_{\sigma_0(n-m)})_2$, specifies the memory identification number (MID) to which the data are loaded. The remaining n-m bits, $(b_{\sigma_0(n-m-1)}\cdots b_{\sigma_0(0)})_2$, specifies the off-set or local address of the data. The initial permutation P_0 depends on the dimension and the number of points, n_i, $1 \leq i \leq t$, in each dimension. For the class of algorithms under consideration, the initial permutation is $R_{2^{n_1}} \otimes \cdots \otimes R_{2^{n_t}}$.

Example: To illustrate the loading process, let us consider 1-D 16-point, and 2-D (4×4)-point DFT and assume that there are 4 processor elements. For a 1-D 16-point DFT, $P_0 = R_{16}$ which can be specified by the permutation $\sigma_0 = (3, 2, 1, 0)$.

Permuting $(b_3b_2b_1b_0)_2$ with the $\sigma_0 = (3, 2, 1, 0)$ results in $(b_0b_1b_2b_3)_2$. For a 4-processor system, the 2-most significant bits of the permuted bits, $(b_0b_1)_2$, identify the target memory ID and the remaining bits, (b_2b_3), specify the local offset. The permuted addresses are equal $(b_0b_1b_2b_3)_2$ while counting $(b_3b_2b_1b_0)_2$.

$(b_3b_2b_1b_0)_2$	:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$(b_0b_1b_2b_3)_2$	:	0	8	4	12	2	10	6	14	1	9	5	13	3	11	7	15
target MID $(b_0b_1)_2$	:	0	2	1	3	0	2	1	3	0	2	1	3	0	2	1	3
local offset $(b_2b_3)_2$	:	0	0	0	0	2	2	2	2	1	1	1	1	3	3	3	3

For a 2-D (4×4)-point DFT, $P_0 = R_4 \otimes R_4$, which can be specified by the permutation $\sigma_0 = (2, 3, 1, 0)$. Then, permuting $(b_3b_2b_1b_0)_2$ with $\sigma_0 = (2, 3, 1, 0)$ results in $(b_2b_3b_0b_1)_2$. For a 4-processor system, the target memory ID is specified by $(b_2b_3)_2$ and the local offset is specified by $(b_0b_1)_2$.

$(b_3b_2b_1b_0)_2$	:	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
$(b_2b_3b_0b_1)_2$	:	0	2	1	3	8	10	9	11	4	6	5	7	12	14	13	15
target MID $(b_2b_3)_2$	:	0	0	0	0	2	2	2	2	1	1	1	1	3	3	3	3
local offset $(b_0b_1)_2$	:	0	2	1	3	0	2	1	3	0	2	1	3	0	2	1	3

$\square$

Once the input data are loaded, the computation starts following a schedule specified by the internal permutations P_i and twiddle factor matrices T_i. The twiddle factors T_i which are needed for the butterfly computations are mapped to the computation units (CU) while the internal permutations P_i are mapped to the address generators (AU).

3.2.2 FFT Tasks

An N-point FFT, where $N = 2^n$, contains n stages of $\frac{N}{2}$ butterfly operations. Let $t_{i,j}$ be the j^{th} butterfly operation in the i^{th} stage, where $1 \le i \le n$ and $0 \le j < \frac{N}{2}$. Then,

$$\mathbf{y_j} = \begin{pmatrix} y_{2j} \\ y_{2j+1} \end{pmatrix} = F_2 W_2(\omega_N^{r_j})\mathbf{x_j} = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \omega_N^{r_j} \end{pmatrix} \begin{pmatrix} x_{2j} \\ x_{2j+1} \end{pmatrix} \tag{3-1}$$

where x_{2j} and x_{2j+1} are input data, y_{2j} and y_{2j+1} are output data, and $w_N^{r_j}$ is the twiddle factor of the butterfly operation. Since $w_N^{r_j} = e^{2\pi i \frac{r_j}{N}}$, it can be represented by the fraction $\frac{r_j}{N}$. When the operations are done in-place, the addresses of both input and output data are the same. Therefore, a butterfly task is defined by a pair of addresses and a twiddle factor.

Definition 17 (FFT Task). *A task $t_{i,j}$ is the j^{th} butterfly operation of stage i defined in Equation 3-1. We denote $t_{i,j}$ by the following tuple.*

$$t_{i,j} \;\; := \;\; (A_{2j}, A_{2j+1}, \frac{r_j}{N}),$$

where

A_{2j} is the address of x_{2j} and y_{2j},

A_{2j+1} is the address of x_{2j+1} and y_{2j+1}

$\frac{r_j}{N}$ represents the twiddle factor, $w_N^{r_j}$.

Although the address part and the twiddle part of the task are related, we generate them independently. A task is divided into two parts: the butterfly addresses (A_{2j}, A_{2j+1}) and the twiddle fraction $\frac{r_j}{N}$.

Different FFT algorithms generate different sequence of tasks. Describing an FFT algorithm by a matrix formula allows us to parameterize a sequence of FFT tasks by a set of permutations P_i and a set of twiddle factors T_i, $1 \leq i \leq n$. Let σ_i be the permutation defining P_i. Then the internal permutations can be generated by σ_i. Similarly the twiddle factors can be generated by σ_i; however, in addition to σ_i the dimension parameters $n_1, \ldots, n_t$ are required. The following examples illustrate how the sequence of FFT tasks is generated.

Example: The dimensionless Pease algorithm of size 16 points is described by the following formula.

$$\begin{aligned}\mathcal{F}_{16} &= L_2^{16}(I_8 \otimes F_2)T_4L_8^{16} \cdot L_4^{16}(I_8 \otimes F_2)T_3L_4^{16} \cdot \\ &\quad L_2^{16}(I_8 \otimes F_2)T_2L_8^{16} \cdot (I_8 \otimes F_2)T_1 \cdot P_0\end{aligned}$$

As explained above, the initial permutation P_0 is performed during the loading process. The task $t_{i,j} = (A_{2j}, A_{2j+1}, \frac{r_j}{N})$, where $1 \leq i \leq 4$ and $0 \leq j < 8$ is parameterized by the permutation P_i and twiddle matrix T_i.

Let $j = (b_3b_2b_1)_2$. Then, $2j = (b_3b_2b_10)_2$, $2j+1 = (b_3b_2b_11)_2$ and $(b_3b_2b_1b_0)_2 = 2j + b_0$. Let σ_i is the permutation of degree 4 specifying P_i. Then, the address pair (A_{2j}, A_{2j+1}) is computed by permuting $(b_3b_2b_1b_0)_2$ with σ_i while counting $(b_3b_2b_1b_0)_2$. Table 3.1 shows the permuted bits at each stage of the Pease algorithm, and Table 3.2 shows the sequence of butterfly addresses. Note that the butterfly addresses are

Table 3.1 Permuted bits for generating butterfly addresses following the Pease algorithm of size 16 points

Stage	P_i	σ_i	Permuted Bits	(A_{2j}, A_{2j+1})
1	I_{16}	$(0,1,2,3)$	$(b_3b_2b_1b_0)$	$((b_3b_2b_10)_2, (b_3b_2b_11)_2)$
2	L_2^{16}	$(1,2,3,0)$	$(b_2b_1b_0b_3)$	$((b_2b_10b_3)_2, (b_2b_11b_3)_2)$
3	L_4^{16}	$(2,3,0,1)$	$(b_1b_0b_3b_2)$	$((b_10b_3b_2)_2, (b_11b_3b_2)_2)$
4	L_8^{16}	$(3,0,1,2)$	$(b_0b_3b_2b_1)$	$((0b_3b_2b_1)_2, (1b_3b_2b_1)_2)$

Table 3.2 Sequence of butterfly addresses following the Pease algorithm of size 16 points

Stage	(A_{2j}, A_{2j+1})							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	(0,1)	(2,3)	(4,5)	(6,7)	(8,9)	(10,11)	(12,13)	(14,15)
2	(0,2)	(4,6)	(8,10)	(12,14)	(1,3)	(5,7)	(9,11)	(13,15)
3	(0,4)	(8,12)	(1,5)	(9,13)	(2,6)	(10,14)	(3,7)	(11,15)
4	(0,8)	(1,9)	(2,10)	(3,11)	(4,12)	(5,13)	(6,14)	(7,15)

independent of the dimension.

The twiddle factor T_i depends on two parameters: the algorithm specified by permutation σ_i and the dimension specification specified by t and $n_1, \ldots n_t$. For the Pease algorithm of size 16 points, the twiddle factors follow Equation 2-49. Let us consider the case of 1-D 16-point and 2-D (4 × 4)-point DFTs using the Pease algorithm. As shown in Section 2.4, the twiddle factors for the Pease algorithm are described by the following equation.

$$T_{n-i} \;=\; \bigoplus_{a=0}^{p-1}\bigoplus_{b=0}^{q-1}\bigoplus_{c=0}^{r-1}\bigoplus_{d=0}^{1} \omega_{2p}^{a \cdot d};\; p = 2^{n_k - l - 1}, q = 2^{n-d(k)},\text{ and } r = 2^i$$

where k is an index counting from 1 to t, l is an index counting from 0 to $n_k - 1$, $d(k) = \sum_{i=1}^{k} n_i$, and $i = d(k) - n_k + l$. Note that the index d counts from 0 to 1. The twiddle factor for the task $t_{i,j}$ is equal to ω_{2p}^{a} and is represented by $\frac{a}{2p}$.

The index j can be written in terms of the indices a, b, c. In particular, $j = qra + rb + c = (a, b, c)$, where (a, b, c) is the mixed-radix representation of number system specified by the factorization $p \times q \times r$. Since p, q and r are powers of two, we can represent (a, b, c) with a binary number. Let $(a, b, c) = (b_3 b_2 b_1)_2$, in which case $b_3 = a$, $b_2 = b$, and $b_1 = c$. Table 3.3 shows the sequence of twiddle fractions, $\frac{a}{2p} = \frac{b_3}{2p}$, generated in each stage for 1-D 16-point DFT using the Pease algorithm. The twiddle fractions for the 2-D (4 × 4)-point DFT are shown in Table 3.4. □

Table 3.3 Sequence of twiddle factors of a 1-D 16-point DFT using the Pease algorithm

Stage	$\frac{r_j}{N}$							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
2	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
3	$\frac{0}{8}$	$\frac{0}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{2}{8}$	$\frac{3}{8}$	$\frac{3}{8}$
4	$\frac{0}{16}$	$\frac{1}{16}$	$\frac{2}{16}$	$\frac{3}{16}$	$\frac{4}{16}$	$\frac{5}{16}$	$\frac{6}{16}$	$\frac{7}{16}$

Table 3.4 Sequence of twiddle factors of a 2-D (4 × 4)-point DFT using the Pease algorithm

Stage	$\frac{r_j}{N}$							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
2	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
3	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
4	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$

Different FFT algorithms produces different sequence of tasks $t_{i,j}$. The following example presents a different algorithm.

Example: Let us consider an alternative FFT algorithm specified by the following set of internal permutations P_i:

$$\begin{array}{lll}
P_1 = (L_2^8 \otimes I_2)L_2^{16}, & \sigma_1 = (0,2,3,1), & \sigma_1^{-1} = (0,3,1,2) \\
P_2 = L_2^{16}, & \sigma_2 = (2,1,0,3), & \sigma_2^{-1} = (3,0,1,2) \\
P_3 = (L_4^8 \otimes I_2)L_4^{16}, & \sigma_3 = (2,1,3,0), & \sigma_3^{-1} = (3,1,0,2) \\
P_4 = (L_4^8 \otimes I_2)L_8^{16}, & \sigma_4 = (3,2,0,1), & \sigma_4^{-1} = (2,3,1,0)
\end{array}$$

Table 3.5 shows the permuted bits at each stage using the alternate algorithm. The

Table 3.5 Permuted bits for generating butterfly addresses of the alternative algorithm

Stage	σ_i	σ_i^{-1}	Permuted Bits	(A_{2j}, A_{2j+1})
1	$(0,2,3,1)$	$(0,3,1,2)$	$(b_2b_1b_3b_0)_2$	$((b_2b_1b_30)_2, (b_2b_1b_31)_2)$
2	$(2,1,0,3)$	$(3,0,1,2)$	$(b_2b_1b_0b_3)_2$	$((b_2b_10b_3)_2, (b_2b_11b_3)_2)$
3	$(0,2,3,1)$	$(3,1,0,2)$	$(b_2b_0b_1b_3)_2$	$((b_20b_1b_3)_2, (b_21b_1b_3)_2)$
4	$(0,2,3,1)$	$(2,3,1,0)$	$(b_0b_1b_3b_2)_2$	$((0b_1b_3b_2)_2, (1b_1b_3b_2)_2)$

sequence of butterfly addresses (A_{2j}, A_{2j+1}) at each stage can be generated by counting $(b_3b_2b_1b_0)_2$ while computing the permuted bits. The butterfly addresses A_{2j} and A_{2j+1} are equal to the pair of addresses when $b_0 = 0$ and $b_0 = 1$ respectively. Table 3.6 shows the sequence of butterfly addresses in each stage of the alternative algorithm. Note that the sequence of addresses is different from those of the Pease algorithm

Table 3.6 Addresses sequence for the alternative algorithm

Stage	(A_{2j}, A_{2j+1})							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	(0,1)	(4,5)	(8,9)	(12,13)	(2,3)	(6,7)	(10,11)	(14,15)
2	(0,2)	(4,6)	(8,10)	(12,14)	(1,3)	(5,7)	(9,11)	(13,15)
3	(0,4)	(2,6)	(8,12)	(10,14)	(1,5)	(3,7)	(9,13)	(11,15)
4	(0,8)	(4,12)	(1,9)	(5,13)	(2,10)	(6,14)	(3,11)	(7,15)

shown in Table 3.2.

The twiddle factors for the alternative algorithms are obtained from Equation 2-55.

$$T_{n-i} \quad = \quad P_{n-i}(I_{2^i} \otimes T^{2^{n_k-l}}_{2^{n_k-l-1}} \otimes I_{2^{n-d(k)}})P^{-1}_{n-i}$$

where k is an index counting from 1 to t, l is an index counting 0 to n_k, $d(k) = \sum_{i=1}^{k} n_i$, and $i = d(k) - n_k + l$.

Let consider a 1-D 16-point DFT ($t = 1$ and $n_1 = 4$) and a 2-D (4×4)-point DFT ($t = 2$, $n_1 = 2$, and $n_2 = 2$). Then,

$$\begin{aligned}
&T_{4-i}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0}) \\
&= P_{4-i}(I_{2^i} \otimes T^{2^{n_k-l}}_{2^{n_k-l-1}} \otimes I_{2^{n-d(k)}})P^{-1}_{4-i}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0}) \\
&= P_{4-i}(I_{2^i} \otimes T^{2^{n_k-l}}_{2^{n_k-l-1}} \otimes I_{2^{n-d(k)}})(\mathbf{e}^2_{b_{\sigma^{-1}_{4-i}(3)}} \otimes \mathbf{e}^2_{b_{\sigma^{-1}_{4-i}(2)}} \otimes \mathbf{e}^2_{b_{\sigma^{-1}_{4-i}(1)}} \otimes \mathbf{e}^2_{b_{\sigma^{-1}_{4-i}(0)}})
\end{aligned}$$

This shows that the twiddle factors depend on σ_i, $d(k)$, n_k, and l. The permutation σ_i specifies the FFT dataflow. The other parameters depend on the dimension specification.

<u>**$\mathbf{1-D\ 16-point\ DFT}$**</u>

Stage $4 : \sigma_4^{-1} = (2, 3, 1, 0),\ k = 1,\ d(k) = 4, l = 4,\ i = 0$

$$\begin{aligned}
&T_4(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0}) \\
&\quad = P_4(T^{16}_8)(\mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2}) = \omega_{16}^{(b_1 b_3 b_2)_2 \cdot b_0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})
\end{aligned}$$

Stage 3 : $\sigma_3^{-1} = (3, 1, 0, 2),\ k = 1,\ d(k) = 4, l = 1,\ i = 1$

$$T_3(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_3(I_2 \otimes T^8_4)(\mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_3}) = \omega_8^{(b_1 b_3)_2 \cdot b_0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

Stage 2 : $\sigma_2^{-1} = (3, 0, 1, 2),\ k = 1,\ d(k) = 4, l = 2,\ i = 2$

$$T_2(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_2(I_4 \otimes T^4_2)(\mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_3}) = \omega_4^{b_3 \cdot b_0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

Stage 1 : $\sigma_1^{-1} = (0, 3, 1, 2),\ k = 1,\ d(k) = 4, l = 3,\ i = 3$

$$T_1(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_1(I_8 \otimes T^2_1)(\mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_0}) = \omega_2^{0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

$\underline{\mathbf{2-D\ (4 \times 4)-point\ DFT}}$

Stage 4 : $\sigma_4^{-1} = (2, 3, 1, 0),\ k = 1,\ d(k) = 2, l = 0,\ i = 0$

$$T_4(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_4(T^4_2 \otimes I_4)(\mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2}) = \omega_4^{b_1 \cdot b_0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

Stage 3 : $\sigma_3^{-1} = (3, 1, 0, 2),\ k = 1,\ d(k) = 2, l = 1,\ i = 1$

$$T_3(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_3(I_2 \otimes T^2_1 \otimes I_4)(\mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_3}) = \omega_2^{0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

Stage 2 : $\sigma_2^{-1} = (3, 0, 1, 2),\ k = 2,\ d(k) = 4, l = 0,\ i = 2$

$$T_2(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$
$$= P_2(I_4 \otimes T^4_2)(\mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0} \otimes \mathbf{e}^2_{b_3}) = \omega_4^{b_3 \cdot b_0}(\mathbf{e}^2_{b_3} \otimes \mathbf{e}^2_{b_2} \otimes \mathbf{e}^2_{b_1} \otimes \mathbf{e}^2_{b_0})$$

$$\text{Stage } 1 : \sigma_1^{-1} = (0, 3, 1, 2),\ k = 2,\ d(k) = 4, l = 1,\ i = 3$$

$$T_1(\mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2)$$

$$= P_1(I_8 \otimes T_1^2)(\mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_0}^2) = \omega_2^0(\mathbf{e}_{b_3}^2 \otimes \mathbf{e}_{b_2}^2 \otimes \mathbf{e}_{b_1}^2 \otimes \mathbf{e}_{b_0}^2)$$

Table 3.7 and 3.8 show the twiddle fractions for computing 1-D 16-point and 2-D

Table 3.7 Sequence of twiddle fractions for a 1-D 16-point using the alternative algorithm

Stage	$\frac{r_j}{N}$							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
2	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
3	$\frac{0}{8}$	$\frac{2}{8}$	$\frac{0}{8}$	$\frac{2}{8}$	$\frac{1}{8}$	$\frac{1}{8}$	$\frac{3}{8}$	$\frac{3}{8}$
4	$\frac{0}{16}$	$\frac{4}{16}$	$\frac{1}{16}$	$\frac{5}{16}$	$\frac{2}{16}$	$\frac{6}{16}$	$\frac{5}{16}$	$\frac{7}{16}$

Table 3.8 Sequence of twiddle fractions for a 2-D (4×4)-point using the alternative algorithm

Stage	$\frac{r_j}{N}$							
i	j=0	j=1	j=2	j=3	j=4	j=5	j=6	j=7
1	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
2	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$	$\frac{1}{4}$
3	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$	$\frac{0}{2}$
4	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{0}{4}$	$\frac{1}{4}$	$\frac{0}{4}$	$\frac{1}{4}$

(4×4)-point DFTs using the alternative algorithm respectively. $\square$

3.2.3 Round-robin Scheduling

In the previous section we showed how to determine the sequence of tasks corresponding to an FFT formula. In this section we map the tasks to PEs. The particular mapping that is used is not important, since any mapping of a given formula is equivalent to some other formula using any other mapping simply by using appropriate permutations. Thus we choose to use a simple round-robin schedule.

Let 2^n and 2^m be the number of points and number of processor elements respectively. Let $j = (b_{n-1} \cdots b_1)_2$ be the task number in each stage. Then, with the round-robin scheduling, the task $t_{i,j}$ is mapped to processor $\mathbf{PE}_r$ where $r = j \bmod 2^m$. Since $j \bmod 2^m$ is equal to $(b_m \cdots b_1)_2$, all the tasks $t_{i,j}$, that have the same $(b_m \cdots b_1)_2$ are mapped to the same processor elements. The following example demonstrates round-robin scheduling.

Example: We map the sequence of tasks corresponding to the Pease and alternative algorithms presented in the previous section to a processor with four processing elements. Since $m = 2$ and $n = 4$, all tasks that have the same $(b_2b_1)_2$ are mapped to the same processor elements. Specifically, all tasks $t_{i,j}$ where $j = (b_300)_2$, $j = (b_301)_2$, $j = (b_310)_2$, and $j = (b_311)_2$ are mapped to $\mathbf{PE}_0$, $\mathbf{PE}_1$, $\mathbf{PE}_2$, and $\mathbf{PE}_3$ respectively. Tables 3.9 and Table 3.10 show the round-robin schedules for the Pease and alternative algorithms respectively. □

Table 3.9 Address schedule corresponding to the Pease algorithm of size 16 points

Stage	$\mathbf{PE}_0$, $j = (b_300)_2$		$\mathbf{PE}_1$, $j = (b_301)_2$		$\mathbf{PE}_2$, $j = (b_310)_2$		$\mathbf{PE}_3$, $j = (b_311)_2$	
i	j=0	j=4	j=1	j=5	j=2	j=6	j=3	j=7
1	(0,1)	(8,9)	(2,3)	(10,11)	(4,5)	(12,14)	(6,7)	(14,15)
2	(0,2)	(1,3)	(4,6)	(5,7)	(8,10)	(9,11)	(12,14)	(13,15)
3	(0,4)	(2,6)	(8,12)	(10,14)	(1,5)	(3,7)	(9,13)	(11,15)
4	(0,8)	(4,12)	(1,9)	(5,13)	(2,10)	(6,14)	(3,11)	(7,15)

Table 3.10 Address schedule corresponding to the alternative algorithm of size 16 points

Stage	$\mathbf{PE}_0$, $j = (b_300)_2$		$\mathbf{PE}_1$, $j = (b_301)_2$		$\mathbf{PE}_2$, $j = (b_310)_2$		$\mathbf{PE}_3$, $j = (b_311)_2$	
i	j=0	j=4	j=1	j=5	j=2	j=6	j=3	j=7
1	(0,1)	(2,3)	(4,5)	(6,7)	(8,9)	(10,11)	(12,13)	(14,15)
2	(0,2)	(1,3)	(4,6)	(5,7)	(8,10)	(9,11)	(12,14)	(13,15)
3	(0,4)	(1,5)	(2,6)	(3,7)	(8,12)	(9,13)	(10,14)	(11,15)
4	(0,8)	(2,10)	(4,12)	(6,14)	(1,9)	(3,11)	(5,13)	(7,15)

It is important to note that since this mapping will be done during the design process (compile-time), each PE can generate its own schedule independently. This allows us to have a scalable design in terms of the number of processor elements.

3.2.4 Address Generation

Corresponding to an FFT dataflow are a sequence of tasks. This section describes how to generate the addresses for the tasks given an FFT dataflow. The sequence of addresses can be generated independently for each PE. An address mapped to $\mathbf{PE}_r$ may belong to any memory module. If an address does not belong to local memory, the data must be accessed remotely via the interconnection network.

It is advantageous to generate addresses from both the perspective of the PE and the memory (the two perspectives are inverses of each other). In our processor design (see Figure 3.2) the address generator generates a sequence of addresses; however, rather than generating the addresses needed by the corresponding PE it generates the sequence of addresses of the elements of its local memory that are used. In addition to these local addresses, the address generator generates the PID of the processor where the element will be used. If this PID is remote, then the local memory element is sent to the interconnection network. If it is local then the memory element is placed in a FIFO. Finally, the MID of the addresses of the data elements that are used by the PE are also generated. There is no need to know the local offset of the data elements used locally as these will be computed by the memory where they come from. However, the MID must be known so that the PE knows where to expect the data to come from.

The local addresses, the target PID and the source MID are generated based on the same FFT dataflow specified by the internal permutations. Regardless of the implementation techniques described later in Chapter 5, these three values can be generated by permutation of the bits that defines the tensor permutations in the FFT dataflow. The following procedure describes a method for generating the local address, the target PID, and the source MID, from a permutation σ of the address bits.

Mapping Procedure

(1) Permute the global counter bits denoted by $(b_{n-1}\cdots b_0)_2$ by σ_i. Then, the m-most significant bits are set to the MID denoted by $(p_{m-1}\cdots p_0)_2$;

$$\begin{aligned} b_{n-1}\cdots b_0 &\overset{\sigma_i}{\rightarrow} b_{\sigma^{-1}(n-1)}\cdots b_{\sigma^{-1}(0)} \\ &\rightarrow \underline{p_{m-1}\cdot p_0} b_{\sigma^{-1}(n-m-1)}\cdots b_{\sigma^{-1}(0)} \end{aligned}$$

In other words, we relabel $(b_{\sigma^{-1}(n-1)}\cdots b_{\sigma^{-1}(n-m)})$ with $(p_{m-1}\cdots p_0)$. For example, if $P_i = L^{16}_8$ and $m = 2$,

$$\begin{aligned} \sigma_i &= (3,0,1,2), \quad \sigma_i^{-1} = (1,2,3,0) \\ b_3b_2b_1b_0 &\overset{\sigma_i}{\rightarrow} b_{\sigma_i^{-1}(3)}b_{\sigma_i^{-1}(2)}b_{\sigma_i^{-1}(1)}b_{\sigma_i^{-1}(1)} \rightarrow b_0b_3b_2b_1 \rightarrow \underline{p_1p_0}\ b_2b_1 \end{aligned}$$

(2) Then permute the new label back to the original order with the inverse permutation σ_i^{-1}. However, now there are m bits that are fixed to the MID. The remaining (n-m) bits become the output bits of the local counter. We relabel these bits as $a_{n-m-1},\ldots,a_0$. Since $(b_m\cdots b_1)_2$ specifies the PID to which the address are mapped, the bits that replaces them specifies the "target PID". For example, if $P_i = L^{16}_8$ and $m = 2$.

$$\begin{aligned} p_1p_0b_2b_1 &\overset{\sigma_i^{-1}}{\rightarrow} p_0\underline{b_2b_1}p_1 = p_0\underline{a_1a_0}p_1 \\ \texttt{target PID} &= a_1a_0 \end{aligned}$$

(3) Permute the new label (local counter bits combined with the MID) by σ_i to get the permuted bit pattern for generating the local addresses. For the above example,

$$\begin{aligned} p_1 a_1 a_0 p_0 &\overset{\sigma_i}{\rightarrow} p_1 p_0 a_1 a_0 \\ \texttt{Local Address} &= a_1 a_0 \end{aligned}$$

(4) The permuted bit pattern for generating the source MID can be found by (1) replacing the original counter bits $(b_m \cdots b_1)$ with $(p_{m-1} \cdots p_0)$ (2) relabeling the remaining (n-m) bits, $(b_{n-1} \cdots b_{n-m} b_0)$, with the local counter bits $a_{n-m-1} \cdots a_0$ (3) permuting the new bits with σ_i and (4) taking the m most significant bits after the permutation as the source MID. For the above example, we have

$$\begin{aligned} a_1 p_1 p_0 a_0 &\overset{\sigma_i}{\rightarrow} \underline{a_0 a_1} p_1 p_0 \\ \texttt{source MID} &= a_0 a_1 \end{aligned}$$

Example: Table 3.11 shows the local addresses, target PID and source MID

Table 3.11 Source MID, target PID and local addresses in a 4-processor system at stage i when $P_i = L_8^{16}$ or $\sigma_i = (3, 0, 1, 2)$

Local Counter	Source MID	Target PID	Local Address
a_1a_0	a_0a_1	a_1a_0	a_1a_0
00	00	00	00
01	10	01	01
10	01	10	10
11	11	11	11

generated in each PE at stage i when $P_i = L_8^{16}$ and there are 4 processors $(m = 2)$.

From the perspective of the memory module, all memory modules send their first data (local address = 00) to $\mathbf{PE}_0$. Therefore, $\mathbf{PE}_0$ will receive data addressing

$(0000)_2$, $(0100)_2$, $(1000)_2$ and $(1100)_2$ from $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$ respectively. These data are stored in the corresponding FIFO. The data control unit, then, uses the "source MID" to sequence the data from the FIFOs to the computation unit. Note that L_8^{16} is the permutation at stage 4 of the Pease algorithm of size 16 points. Since the source MID is equal to 0, 2, 1, 3 respectively, the sequence of butterfly addresses mapped to $\mathbf{PE}_0$ is $(0, 8)$ and $(4, 12)$ respectively. This is the same as the sequence of butterfly addresses shown Table 3.9 of Section 3.2.3.

Similarly, the second data of each memory module are sent to $\mathbf{PE}_1$. Therefore, $\mathbf{PE}_1$ will receive data with addresses $(0001)_2$, $(0101)_2$, $(1001)_2$ and $(1101)_2$ from $\mathbf{M}_0$, $\mathbf{M}_1$, $\mathbf{M}_2$ and $\mathbf{M}_3$ respectively. According to the source MID, the data are sequenced to the computation in order of $(1, 9)$ and $(5, 13)$ which is the same as those in Table 3.9.

Similarly, we can see that the address sequence mapped to $\mathbf{PE}_2$ is $(2, 10)$, $(6, 14)$, and addresses sequence mapped to $\mathbf{PE}_3$ is $(3, 11)$, $(7, 15)$. $\square$

We apply the mapping process to the Pease algorithm of size 2^n points and number of processors equal to 2^m.

Example: For the Pease algorithm, the permutation P_i, $1 \leq P_i \leq n$, is equal to $L_{2^{i-1}}^{2^n}$ which can be defined by a permutation function σ_i as follows.

For $i = 1$,

$$\sigma_1 = \sigma_1^{-1} = (0, 1, \ldots, n-2, n-1)$$

For $i \geq 2$,

$$\begin{aligned} \sigma_i &= (i-1, i, \ldots, n-1, 0, \ldots, i-3, i-2) \\ \sigma_i^{-1} &= (n-i, \ldots, n-1, 0, \ldots, n-i-2, n-i-1) \end{aligned}$$

Step 1: Permute $(b_{n-1}\cdots b_0)_2$ with σ_i. Then, set the m most significant bits to $p_{m-1}\cdots p_0$.

$$b_{n-1}b_{n-2}\cdots b_0 \overset{\sigma_i}{\rightarrow} b_{n-i}b_{n-i-2}\cdots b_1b_0b_{n-1}\cdots b_{n-i+2}b_{n-i+1}$$

For $i=1$,

$$b_{n-1}b_{n-2}\cdots b_1b_0 \rightarrow \underline{p_{m-1}\cdots p_0}\ b_{n-1}\cdots b_1b_0$$

For $2 \le i < n-m$,

$$b_{n-i}b_{n-i-2}\cdots b_1b_0b_{n-1}\cdots b_{n-i+2}b_{n-i+1} \rightarrow \underline{p_{m-1}\cdots p_0}\ b_{n-m-i}\cdots b_0b_{n-1}\cdots b_{n-i+1}$$

For $i=n-m$,

$$b_m\cdots b_1b_0b_{n-1}\cdots b_{m+2}b_{m+1} \rightarrow \underline{p_{m-1}\cdots p_0}\ b_0b_{n-1}\cdots b_{m+2}b_{m+1}$$

For $n-m<i\le n$, $i=n-m+j$ and $1\le j\le m$

$$b_{m-j}\cdots b_1b_0b_{n-1}\cdots b_{m-j+1} \rightarrow \underline{p_{m-1}\cdots p_0}\ b_{n-j}\cdots b_{m-j+1}$$

Step 2: Permute the result from the first step with σ_i^{-1} and relabel the remaining bits to $a_{n-m-1},\cdots,a_0$. The bits that replace $b_m\cdots b_1$ becomes the "target PID".

Step 3: Permute the new label with σ_i. The $(n-m)$ least significant bits specifies the "local address".

For $i=1$,

$$\begin{aligned} p_{m-1} \quad \cdots \quad & p_0\ b_{n-m-1}\cdots b_1b_0 \\ \overset{\sigma_i^{-1}}{\rightarrow}\quad & p_{m-1}\cdots p_0b_{n-m-1}\cdots b_1a_0 \\ \rightarrow\quad & p_{m-1}\cdots p_0a_{n-m-1}\cdots a_{m+1}\underline{a_m\cdots a_1}a_0 \\ \overset{\sigma_i}{\rightarrow}\quad & p_{m-1}\cdots p_0\ \underline{a_{n-m-1}\cdots a_1a_0} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (a_m \cdots a_1)_2 \\ \texttt{local address} &= (a_{n-m-1} \cdots a_0)_2 \end{aligned}$$

For $2 \le i < n-2m$,

$$\begin{aligned} p_{m-1} \cdots\ & p_0\ b_{n-m-i} \cdots b_0 b_{n-1} \cdots b_{n-i+1} \\ \overset{\sigma_i^{-1}}{\to}\ & b_{n-1} \cdots b_{n-i+1}\ p_{m-1} \cdots p_0\ b_{n-m-i} \cdots b_1 b_0 \\ \to\ & a_{n-m-1} \cdots a_{n-m-i+1}\ p_{m-1} \cdots p_0\ a_{n-m-i} \cdots a_{m+1}\underline{a_m \cdots a_1}a_0 \\ \overset{\sigma_i}{\to}\ & p_{m-1} \cdots p_0\ \underline{a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1}} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (a_m \cdots a_1)_2 \\ \texttt{local address} &= (a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1})_2 \end{aligned}$$

For $i = n-2m$,

$$\begin{aligned} p_{m-1} \cdots\ & p_0\ b_{n-m-i} \cdots b_0 b_{n-1} \cdots b_{n-i+1} \\ \overset{\sigma_i^{-1}}{\to}\ & b_{n-1} \cdots b_{n-i+1}\ p_{m-1} \cdots p_0\ b_{n-m-i} \cdots b_1 b_0 \\ \to\ & a_{n-m-1} \cdots a_{n-m-i+1}\ p_{m-1} \cdots p_0\ \underline{a_m \cdots a_1}a_0 \\ \overset{\sigma_i}{\to}\ & p_{m-1} \cdots p_0\ \underline{a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1}} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (a_m \cdots a_1)_2 \\ \texttt{local address} &= (a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1})_2 \end{aligned}$$

For $n-2m < i < n-m$, $i = n-2m+j$, and $1 \le j < m$

$$\begin{aligned} p_{m-1} \cdots\ & p_0\ b_{n-m-i} \cdots b_0 b_{n-1} \cdots b_{n-i+1} \\ \overset{\sigma_i^{-1}}{\to}\ & b_{n-1} \cdots b_{n-i+1}\ p_{m-1} \cdots p_0\ b_{n-m-i} \cdots b_1 b_0 \\ \to\ & a_{n-m-1} \cdots a_{n-m-i+1}\ p_{m-1} \cdots p_j\underline{p_{j-1} \cdots p_0\ a_{m-j} \cdots a_1}a_0 \\ \overset{\sigma_i}{\to}\ & p_{m-1} \cdots p_0\ \underline{a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1}} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (p_{j-1}\cdots p_0 a_{m-j}\cdots a_1)_2 \\ \texttt{local address} &= (a_{n-m-i}\cdots a_0 a_{n-m-1}\cdots a_{n-m-i+1})_2 \end{aligned}$$

For $i = n - m$,

$$\begin{aligned} p_{m-1} \quad \cdots \quad & p_0\ b_0 b_{n-1}\cdots b_{m+2} b_{m+1} \\ \overset{\sigma_i^{-1}}{\to} \quad & b_{n-1}\cdots b_{m+2} b_{m+1}\ p_{m-1}\cdots p_0 b_0 \\ \to \quad & \ a_{n-m-1}\cdots a_2 a_1\ \underline{p_{m-1}\cdots p_0} a_0 \\ \overset{\sigma_i}{\to} \quad & p_{m-1}\cdots p_0\ \underline{a_0 a_{n-m-1}\cdots a_2 a_1} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (p_{m-1}\cdots p_0)_2 \\ \texttt{local address} &= (a_0 a_{n-m-1}\cdots a_1)_2 \end{aligned}$$

For $i = n - m + 1$,

$$\begin{aligned} p_{m-1} \quad \cdots \quad & p_0\ b_{n-1}\cdots b_m \\ \overset{\sigma_i^{-1}}{\to} \quad & b_{n-1}\cdots b_m\ p_{m-1}\cdots p_0 \\ \to \quad & a_{n-m-1}\cdots a_1 \underline{a_0 p_{m-1}\cdots p_1} p_0 \\ \overset{\sigma_i}{\to} \quad & p_{m-1}\cdots p_0\ \underline{a_{n-m-1}\cdots a_0} \end{aligned}$$

$$\begin{aligned} \texttt{target PID} &= (a_0 p_{m-1}\cdots p_1)_2 \\ \texttt{local address} &= (a_{n-m-1}\cdots a_1 a_0)_2 \end{aligned}$$

For $n - m + 1 < i \leq n$, $i = n - m + j$ and $2 \leq j < m$,

$$\begin{aligned} p_{m-1} \quad \cdots \quad & p_0\ b_{n-j}\cdots b_{m-j+1} \\ \overset{\sigma_i^{-1}}{\to} \quad & p_{j-2}\cdots p_0\ b_{n-j}\cdots b_{m-j+1}\ p_{m-1}\cdots p_{j-1} \\ \to \quad & p_{j-2}\cdots p_0\ a_{n-m-1}\cdots a_j \underline{a_{j-1}\cdots a_0 p_{m-1}\cdots p_j} p_{j-1} \\ \overset{\sigma_i}{\to} \quad & p_{m-1}\cdots p_0\ \underline{a_{n-m-1}\cdots a_0} \end{aligned}$$

$$\begin{aligned}\texttt{target PID} &= (a_{j-1}\cdots a_0 p_{m-1}\cdots p_j)_2\\ \texttt{local address} &= (a_{n-m-1}\cdots a_1 a_0)_2\end{aligned}$$

For $i = n$,

$$\begin{aligned}p_{m-1} &\cdots\ p_0\ b_{n-m}\cdots b_1\\ &\xrightarrow{\sigma_i^{-1}}\ p_{m-2}\cdots p_0\ b_{n-m}\cdots b_1\ p_{m-1}\\ &\rightarrow\ p_{m-2}\cdots p_0\ a_{n-m-1}\cdots a_m\underline{a_{m-1}\cdots a_0}p_{m-1}\\ &\xrightarrow{\sigma_i}\ p_{m-1}\cdots p_0\ \underline{a_{n-m-1}\cdots a_0}\end{aligned}$$

$$\begin{aligned}\texttt{target PID} &= (a_{m-1}\cdots a_0)_2\\ \texttt{local address} &= (a_{n-m-1}\cdots a_1 a_0)_2\end{aligned}$$

Step 4: Find permuted bits for the "source MID". First, $(b_m\cdots b_1)$ is replaced by $(p_{m-1}\cdots p_0)$ and the remaining $n-m$ bits are relabeled to $(a_{n-m-1}\cdots a_0)$. Then, permute the new bits with σ_i. The m most significant bits of the permuted bits specifies the sequence of "source MID".

For $1 \le i \le n-2m$,

$$\begin{aligned}a_{n-m-1} &\cdots\ a_1 p_{m-1}\cdots p_0 a_0\\ &\xrightarrow{\sigma_i}\ \underline{a_{n-m-i}\cdots a_{n-2m-i+1}}\cdots a_1 p_{m-1}\cdots p_0 a_0 a_{n-m-1}\cdots a_{n-m-i+1}\\ \texttt{source MID} &= (a_{n-m-i}\cdots a_{n-2m-i+1})_2.\end{aligned}$$

For $n-2m < i < n-m$, $i = n-2m+j$ and $1 \le j < m$,

$$\begin{aligned}a_{n-m-1} &\cdots\ a_1 p_{m-1}\cdots p_0 a_0\\ &\xrightarrow{\sigma_i}\ \underline{a_{m-j}\cdots a_1 p_{m-1}\cdots p_j}\ p_{j-1}\cdots p_0 a_0 a_{n-m-1}\cdots a_{m-j-1}\\ \texttt{source MID} &= (a_{m-j}\cdots a_1 p_{m-1}\cdots p_j)_2\end{aligned}$$

For $i = n - m$,

$$\begin{aligned} a_{n-m-1} \quad \cdots \quad a_1 p_{m-1} \cdots p_0 a_0 &\overset{\sigma_i}{\to} \underline{p_{m-1} \cdots p_0}\ a_0 a_{n-m-1} \cdots a_1 \\ \texttt{source MID} \quad &= \quad (p_{m-1} \cdots p_0)_2 \end{aligned}$$

For $i = n - m + 1$,

$$\begin{aligned} a_{n-m-1} \quad \cdots \quad & a_1 p_{m-1} \cdots p_0 a_0 \\ \overset{\sigma_i}{\to} \quad & \underline{p_{m-2} \cdots p_0 a_0}\ a_{n-m-1} \cdots a_1 p_{m-1} \\ \texttt{source MID} \quad = \quad & (p_{m-2} \cdots p_0 a_0)_2 \end{aligned}$$

For stage i, $n - m < i < n - 1$, $i = n - m + j$, and $2 \leq j < m$

$$\begin{aligned} a_{n-m-1} \quad \cdots \quad & a_1 p_{m-1} \cdots p_0 a_0 \\ \overset{\sigma_i}{\to} \quad & \underline{p_{m-j-1} \cdots p_0 a_0 a_{n-m-1} \cdots a_{n-m-j+1}}\ a_{n-m-j} \cdots a_1 p_{m-1} \cdots p_{m-j} \\ \texttt{source MID} \quad = \quad & (p_{m-j-1} \cdots p_0 a_0 a_{n-m-1} \cdots a_{n-m-j+1})_2 \end{aligned}$$

For $i = n$,

$$\begin{aligned} a_{n-m-1} \quad \cdots \quad & a_1 p_{m-1} \cdots p_0 a_0 \\ \overset{\sigma_i}{\to} \quad & \underline{a_0 a_{n-m-1} \cdots a_{n-2m+1}} a_{n-2m} \cdots a_1 p_{m-1} \cdots p_0 \\ \texttt{source MID} \quad = \quad & (a_0 a_{n-m-1} \cdots a_{n-2m+1})_2 \end{aligned}$$

Table 3.12, 3.13 and 3.14 summarize the permuted bits pattern for generating the “local address”, “target PID’, and “source MID” for the Pease algorithm of size 2^n. □

Concrete examples of the mapping procedure are provided in the appendix for the Pease algorithm and an algorithm determined to be optimal on our architecture both of size 64.

Table 3.12 Bit patterns for generating local addresses in PE number $(p_{m-1} \cdots p_0)$ using the Pease algorithm

Stage i	P_i	Local Address (n-m bits)
1	I_{2^n}	$(a_{n-m-1} \cdots a_1 a_0)_2$
$2 \leq i \leq n-m$	$L^{2^n}_{2^{i-1}}$	$(a_{n-m-i} \cdots a_0 a_{n-m-1} \cdots a_{n-m-i+1})_2$
$n-m < i \leq n$	$L^{2^n}_{2^{i-1}}$	$(a_{n-m-1} \cdots a_1 a_0)_2$

Table 3.13 Bit patterns for generating "target PID" in PE number $(p_{m-1} \cdots p_0)_2$ using the Pease algorithm

Stage i	P_i	Target ID (m bits)
1	I_{2^n}	$(a_m \cdots a_1)_2$
$2 \leq i \leq 2n-m$	$L^{2^n}_{2^{i-1}}$	$(a_m \cdots a_1)_2$
n-2m+1	$L^{2^n}_{2^{n-2m}}$	$(p_0 a_{m-1} \cdots a_1)_2$
$n-2m+1 \leq i < n-m$ $j = i-n+2m$	$L^{2^n}_{2^{n-2m+j}}$	$(p_{j-1} \cdots p_0 a_{m-j-1} \cdots a_1)_2$
n-m	$L^{2^n}_{2^{n-m-1}}$	$(p_{m-1} \cdots p_0)_2$
n-m+1	$L^{2^n}_{2^{n-m}}$	$(a_0 p_{m-1} \cdots p_0)_2$
$n-2m+1 \leq i < n-m$ $j = i-n+2m$	$L^{2^n}_{2^{n-m+j-1}}$	$(a_{j-1} \cdots a_0 p_{m-1} \cdots p_j)_2$
n-1	$L^{2^n}_{2^{n-1}}$	$(a_{m-1} \cdots a_0)_2$

3.2.5 Twiddle Factor Generation

In addition to the two input/output addresses an FFT task includes the twiddle factor needed for the corresponding butterfly operation. Recall that we represent the twiddle factor by a fraction called the twiddle fraction. The task of the twiddle factor generator is to generate the sequence of twiddle fractions for all of the tasks scheduled to a PE. It is important to stress that the twiddle fractions can be generated independently for each PE.

The twiddle fractions are determined from the twiddle matrices occurring in the

Table 3.14 Bit patterns for generating "source MID" in PE number $(p_{m-1} \cdots p_0)_2$ using the Pease algorithm

Stage i	P_i	Source ID (m bits)
$1 \le i \le 2n - m$	$L^{2^n}_{2^{i-1}}$	$(a_{n-m-i} \cdots a_{n-2m-i+1})_2$
$n - 2m < i < n - m$ $j = i - n + 2m$	$L^{2^n}_{2^{n-2m+j}}$	$(a_{m-j} \cdots a_1 p_{m-1} \cdots p_j)_2$
n-m	$L^{2^n}_{2^{n-m-1}}$	$(p_{m-1} \cdots p_0)_2$
n-m+1	$L^{2^n}_{2^{n-m}}$	$(p_{m-1} \cdots p_0 a_0)_2$
$n - m + 1 < i < n - m$ $j = i - n + m$	$L^{2^n}_{2^{n-m+j-1}}$	$(p_{m-j-1} \cdots p_j a_0 a_{n-m-1} \cdots a_{n-m-j+1})_2$
n-1	$L^{2^n}_{2^{n-1}}$	$(a_0 a_{n-m-1} \cdots a_{n-2m+1})_2$

FFT formula, and the twiddle matrices are completely determined by the internal permutations in the corresponding FFT dataflow and the number of points in each dimension of the desired FFT. We will show that the twiddle fractions can be determined in a two step process. The first step is independent of dimension. It computes the twiddle fraction for the one-dimensional FFT of the given size. The second step uses the dimension specifications to compute the desired twiddle fraction from the corresponding one-dimensional twiddle fraction. This step simply masks out some of the bits in the one-dimensional twiddle fraction. A consequence of this two step procedure is that the major part of twiddle computation depends solely on the dataflow, i.e. the sequence of internal permutations. In fact, the first part of the computation of the twiddle fraction is obtained using the same bit permutation that is used for address calculation.

For the class of algorithms under consideration, Equation 2-55 describes the twiddle factors in term of the conjugating permutations and the dimensions. For convenience they are reproduced here.

$$\begin{aligned}
T_{n-i} &= P_{n-i}(I_{2^i} \otimes T^{2^{n_k-j}}_{2^{n_k-j-1}} \otimes I_{2^{n-d(k)}})P^{-1}_{n-i} \\
k &= 1, 2, \cdots, t \\
j &= 0, 1, \ldots, n_1 - 1, 0, 1, \ldots, n_2 - 1, \ldots, 0, 1, \ldots, n_t - 1 \\
d(k) &= \sum_{l=1}^{k} n_l \\
i &= d(k) - n_k + j
\end{aligned}$$

Converting the indices so that the index $n - i$ counts from 1 to n, we have

$$\begin{aligned}
T_i &= P_i(I_{2^{n-i}} \otimes T^{2^{j+1}}_{2^j} \otimes I_{2^{n-d_k}})P^{-1}_i \\
k &= t, t-1, \cdots, 1 \\
j &= 0, \ldots, n_t - 1, \ldots, 0, \ldots, n_2 - 1, \ldots, 0, \ldots, n_1 - 1 \\
d(k) &= \sum_{l=1}^{k} n_l \\
i &= n - d(k) + j + 1
\end{aligned}$$

The following procedure computes T_i provided the dimension specification $n_1, \cdots, n_t$ and the internal permutations P_i.

```
$T \leftarrow$ **Twiddle**$(n, n_1, \ldots, n_t, P_1, \ldots, P_n)$
$d_k = n$
for $k = t, t-1, \ldots, 1$ loop
  $q = n - d_k$
  for $j = 0, \ldots, n_k - 1$ loop
    $i = q + j + 1$
    $T_i = P_i(I_{2^{n-i}} \otimes T^{2^{j+1}}_{2^j} \otimes I_{2^q})P^{-1}_i$
  end loop
  $d_k = d_k - n_k$
end loop
```

Applying the twiddle properties (Section 2.1.5), we have

$$\begin{aligned}
T_i \mathbf{e}_j^{2^n} &= P_i(I_{2^{n-i}} \otimes T_{2^j}^{2^{j+1}} \otimes I_{2^{n-d_k}})P_i^{-1}(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2) \\
&= P_i(I_{2^{n-i}} \otimes T_{2^j}^{2^{j+1}} \otimes I_{2^{n-d_k}})(\mathbf{e}_{r_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{r_0}^2) \\
&= P_i[\mathbf{e}_{r_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{r_i}^2 \otimes \\
&\qquad T_{2^j}^{2^{j+1}}(\mathbf{e}_{r_{i-1}}^2 \otimes \mathbf{e}_{r_{i-2}}^2 \otimes \cdots \otimes \mathbf{e}_{r_q}^2) \otimes \mathbf{e}_{r_{q-1}}^2 \otimes \cdots \otimes \mathbf{e}_{r_0}^2] \\
&= \omega_{2^{j+1}}^{(r_{i-2}\cdots r_q)_2 \cdot r_{i-1}} P_i(\mathbf{e}_{r_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{r_0}^2) \\
&= \omega_{2^{j+1}}^{(r_{i-2}\cdots r_q)_2 \cdot r_{i-1}} (\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2)
\end{aligned}$$

where

$$\begin{aligned}
\mathbf{e}_{r_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{r_0}^2 &= P_i^{-1}(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2) \\
(b_{n-1}\cdots b_0)_2 &\overset{\sigma_i}{\rightarrow} (r_{n-1}\cdots r_0)_2 = (b_{\sigma_i^{-1}(n-1)} \cdots b_{\sigma_i^{-1}(0)})_2 \\
r_{i-1} &= b_{\sigma_i^{-1}(i-1)} \\
(r_{i-2}\cdots r_q)_2 &= (b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(q)})_2, \quad q = n - d(k) = i - j - 1
\end{aligned}$$

Since $P_i^{-1} = L_{2^{n-i+1}}^{2^n}(Q_i^{-1} \otimes I_2)$, we have

$$\begin{aligned}
P_i^{-1}(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2) &= L_{2^{n-i+1}}^{2^n}(Q_i^{-1} \otimes I_2)(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_0}^2) \\
&= L_{2^{n-i+1}}^{2^n} Q_i(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_1}^2) \otimes \mathbf{e}_{b_0}^2 \\
&= L_{2^{n-i+1}}^{2^n}(\mathbf{e}_{c_{n-2}}^2 \otimes \cdots \otimes \mathbf{e}_{c_0}^2) \otimes \mathbf{e}_{b_0}^2 \\
&= \mathbf{e}_{c_{n-i-1}}^2 \otimes \cdots \otimes \mathbf{e}_{c_0}^2 \otimes \mathbf{e}_{b_0}^2 \otimes \mathbf{e}_{c_{n-2}}^2 \otimes \cdots \otimes \mathbf{e}_{c_{n-i-2}}^2 \\
&= \mathbf{e}_{b_{\sigma_i^{-1}(n-1)}}^2 \otimes \cdots \otimes \mathbf{e}_{b_{\sigma_i^{-1}(i-1)}}^2 \otimes \cdots \otimes \mathbf{e}_{b_{\sigma_i^{-1}(0)}}^2
\end{aligned}$$

where

$$\begin{aligned}
Q_i^{-1}(\mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_1}^2) &= \mathbf{e}_{c_{n-2}}^2 \otimes \cdots \otimes \mathbf{e}_{c_0}^2, \quad \text{and} \\
Q_i(\mathbf{e}_{c_{n-2}}^2 \otimes \cdots \otimes \mathbf{e}_{c_0}^2) &= \mathbf{e}_{b_{n-1}}^2 \otimes \cdots \otimes \mathbf{e}_{b_1}^2.
\end{aligned}$$

Therefore, $b_{\sigma^{-1}(i-1)} = b_0$ and

$$\begin{aligned} T_i &= \bigoplus_{b_{n-1},\ldots,b_0} \omega_{2^{j+1}}^{(b_0)_2 \cdot (b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(q)})_2}, \quad q = i-j-1 \\ &= \bigoplus_{l=0}^{2^{n-1}-1} \bigoplus_{b_0=0}^{1} \omega_{2^{j+1}}^{b_0 \cdot r_l}, \quad r_l = (b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(i-j-1)})_2, \; l = (b_{n-1} \cdots b_1)_2 \end{aligned} \tag{3-2}$$

The two consecutive twiddle factors that are mapped to the same processor are those with $b_0 = 0$ and $b_0 = 1$ respectively. The first twiddle factor ($b_0 = 0$) is always equal to one. Hence, we can represent the twiddle factor of the l^{th} butterfly operation by the twiddle fraction $\frac{r_l}{2^{j+1}}$, where $r_l = (b_{\sigma^{-1}(i-2)} \cdots b_{\sigma^{-1}(i-j-1)})_2$, and $l = (b_{n-1} \cdots b_1)_2$.

The fraction $\frac{r_l}{2^{j+1}}$ can be exactly represented by a binary number of $j+1$ bits or less. Let us assume that $2^{n_{max}}$ is the maximum number of points that the universal FFT will compute. Then, we will need a binary number of $(n_{max} - 1)$ bits or less to represent a twiddle fraction.

Definition 18 (Fraction Representation). *Let a twiddle fraction be represented by n_{max}-bit binary number denoted by $(0.0r_{n_{max}-1} \cdots r_0)_2$. Then,*

$$(0.0r_{n_{max}-1} \cdots r_0)_2 = \sum_{i=0}^{n_{max}-1} \frac{r_i}{2^{n_{max}-i+1}} = \frac{r_{n_{max}}}{4} + \frac{r_{n_{max}-1}}{8} + \cdots + \frac{r_0}{2^{n_{max}+1}} \tag{3-3}$$

Let $\mathtt{TF}_i(l) = \frac{r_l}{2^{j+1}}$ be the twiddle fraction of the l^{th} butterfly operation at stage i. Then,

$$\mathtt{TF}_i(l) = \frac{r_l}{2^{j+1}} = (0.0b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(i-j-1)} 0 \cdots 0)_2 \tag{3-4}$$

Note that for different dimension specification, the index j counts differently. However, since $i = n - d_k + j + 1$, and $n \geq d_k$, j is always less than i. This leads to the following property.

Property 15 (Twiddle Fraction).

$$\begin{aligned}
\mathtt{TF}_i(l) &= \frac{r_l}{2^{j+1}} = \mathtt{TF1}_i(l) \ \mathbf{and}\ \mathtt{mask(j)}, && \text{(3-5)}\\
\mathtt{mask(j)} &= (0.01\cdots10\cdots0)_2 = \frac{2^j-1}{2^{j+1}} && \text{(3-6)}\\
\mathtt{TF1}_i(l) &= (0.0b_{\sigma_i^{-1}(i-2)}\cdots b_{\sigma_i^{-1}(0)}0\cdots0)_2 && \text{(3-7)}
\end{aligned}$$

where **and** *is the bit-wise-and operation,* $\mathtt{TF1}_i(l)$ *is the twiddle fraction for the one-dimensional DFT in which case* $i = j+1$*, and* $\mathtt{mask(j)}$ *is the mask depending on the dimension specification* $(t, n_1, \ldots, n_t)$*.*

Proof: *This follows the fact that* $\mathtt{mask(j)}$ *contains* j *bits of* 1*. Therefore,*

$$\mathtt{TF1}_i(l)\ \mathbf{and}\ \mathtt{mask(j)} = (0.0b_{\sigma_i^{-1}(i-2)}\cdots b_{\sigma_i^{-1}(i-j-1)}0\cdots0)_2$$

which is the same as Equation 3-4. □

This property allows us to separate the generation of twiddle fractions into two parts. The first part, TF1, is the twiddle fraction of the one-dimensional DFT which depends only on the FFT dataflow specified by σ_i^{-1}. The second part, TF1, depends only on the dimension specification $(t, n_1, \ldots, n_t)$.

The following procedure generates all twiddle fractions parameterized by n, t,

$n_1, \ldots, n_t$, and σ_i, $i = 1, \cdots, n$.

```
$TF \leftarrow$ **TwiddleFraction**$(n, t, n_1, \ldots, n_t, \sigma_1, \ldots, \sigma_n)$
$d_k = n$
for $k = t, t-1, \ldots, 1$ loop
  for $j = 0, \ldots, n_k - 1$ loop
    $i = n - d_k + j + 1$
    mask(j) $= (0.01 \cdots 10 \cdots 0)_2 = \frac{2^j - 1}{2^{j+1}}$
    for $l = 0, 1, \ldots, \frac{N}{2} - 1$ loop
      $l = (b_{n-1} \cdots b_1)_2$
      TF1$_i(l) = (0.0 b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(0)} 0 \cdots 0)_2$
      TF$_i(l) =$ TF1$_i(l)$ **and** mask(j)
    end loop
  end loop
  $d_k = d_k - n_k$
end loop
```

With round-robin mapping, the twiddle fraction $\mathtt{TF}_i(l)$, $l = (b_{n-1} \cdots b_1)_2$ are mapped to PE number $(b_m \cdots b_1)_2$. The following procedure generates all twiddle fractions mapped to $\mathbf{PE}_P$, where $\mathtt{P} = (b_m \cdots b_1)_2$.

```
$TF \leftarrow$ **PETwiddleFraction**$(n, m, n_1, \ldots, n_t, \sigma_1^{-1}, \ldots, \sigma_n^{-1}, \mathtt{P})$
$M = 2^m$
$d_k = n$
for $k = t, t-1, \ldots, 1$ loop
  for $j = 0, \ldots, n_k - 1$ loop
    $i = n - d_k + j + 1$
    mask(j) $= (0.01 \cdots 10 \cdots 0)_2 = \frac{2^j - 1}{2^{j+1}}$
    for $l = 0, 1, \ldots, \frac{N}{2M} - 1$ loop
      $l = (b_{n-1} \cdots b_{n-m})_2$
      $\mathtt{P} = (b_m \cdots b_1)_2$
      $t = l \cdot 2^{n-m} + \mathtt{P} = (b_{n-1} \cdots b_1)_2$
      TF1$_i(t) = (b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(0)})_2$
      TF$_i(l) =$ TF1$_i(t)$ **and** mask(j)
    end loop
  end loop
  $d_k = d_k - n_k$
end loop
```

Example: Let us consider the Pease algorithm of size 2^n, where $P_i^{-1} = L^{2^n}_{2^{n-i+1}}$ and

$$\sigma_i^{-1} = (n-i+1, \cdots n-1, 0, \cdots, n-i-1).$$

Note that $\sigma_i^{-1}(0) = n-i+1$ and $\sigma_i^{-1}(i-2) = n-1$. Then,

$$\begin{aligned}\texttt{TF1}_i(l) &= (0.0b_{\sigma_i^{-1}(i-2)} \cdots b_{\sigma_i^{-1}(0)}0 \cdots 0)_2 \\ &= (0.0b_{n-1} \cdots b_{n-i+1}0 \cdots 0)_2\end{aligned}$$

For two-dimensional $(2^{n_1} \times 2^{n_2})$-point DFT, the twiddle fractions are

$$\begin{aligned}\texttt{TF}_i(l) &= \begin{cases} (0.0b_{n-1} \cdots b_{n-i+1}0 \cdots 0)_2, & \text{for } 1 \le i \le n_2 \\ (0.0b_{n-1} \cdots b_{n-j}0 \cdots 0)_2, & \text{for } n_2+1 \le i \le n, \quad j = i-n_2-1 \end{cases} \\ &= \texttt{TF1}_i(l) \textbf{ and } \texttt{mask(j)}, \\ \text{where } \texttt{mask(j)} &= (0.01 \cdots 10 \cdots 0)_2 = \frac{2^j}{2^{j+1}}, \\ j &= \begin{cases} i-1 & \text{for } 1 \le i \le n_2 \\ i-n_2-1 & \text{for } n_2+1 \le i \le n \end{cases}\end{aligned}$$

Notice that for the Pease algorithm, the twiddle fractions of the one-dimensional DFT are almost the same for all stages. In fact,

$$\texttt{TF1}_i(l) = (0.0b_{n-1} \cdots b_1 0 \cdots 0)_2 \textbf{ and } (\frac{2^{i-1}-1}{2^i}) = (0.0b_{n-1} \cdots b_{n-i+1} \cdots 0)_2$$

Since $\frac{2^{i-1}-1}{2^i}$ is equal to the mask for the one-dimensional DFT, we can generate all twiddle fractions of the Pease algorithm using the following Equation.

$$\texttt{TF}_i(l) = (0.0b_{n-1} \cdots b_1 0 \cdots 0)_2 \textbf{ and } \texttt{mask(j)} \tag{3-8}$$

where $\texttt{mask(j)}$ is defined in Equation 3-6 and $l = (b_{n-1} \cdots b_1)_2$.

Let $P = (p_{m-1} \cdots p_0)_2$ be the processor number to which $\texttt{TF}_i(l)$ is assigned. Therefore, $(b_m \cdots b_1)_2 = (p_{m-1} \cdots p_0)_2$. Relabeling $(b_{n-1} \cdots b_{m-1})$ with $(a_{n-m-1} \cdots b_1)$, we obtain the following Equation for generating all twiddle fractions assigned to $\mathbf{PE}_P$.

$$\texttt{TF}_i(l) = (0.0a_{n-m-1} \cdots a_1 p_{m-1} \cdots p_0 0 \cdots 0)_2 \textbf{ and } \texttt{mask(j)}, \tag{3-9}$$

where `mask`(`j`) is defined in Equation 3-6 and $l = (a_{n-m-1} \cdots a_1)_2$. $\square$

In the appendix, concrete examples of twiddle fraction generation are provided for the Pease algorithm and the optimal algorithm of size 64.

4.0 PERFORMANCE MODEL AND OPTIMIZATION

In the previous two chapters, we show that the design choices of distributed-memory FFT processors can be parameterized by FFT algorithms described by matrix factorizations. In this chapter, we present a systematic search for the optimal algorithm.

This process is similar to the search process used in several recent software packages, FFTW [3], SPIRAL project [15] and ATLAS [21], which adapt to the underlying hardware on which they are executing. These systems use empirically measured execution times to search for a configuration or algorithm with smallest runtime. In this thesis, we apply the same technique to find the best hardware implementation of a distributed memory FFT. Since the search is performed at design time, it can not use execution time as a cost function, but instead must use a performance model. Rather than using an analytic model of performance we created a model using ADEPT that simulates the dataflow of the algorithm in the proposed architecture. This approach provides an easier mechanism for capturing the complexity of a real system and is amenable to experimenting with a variety of hardware parameters. Furthermore, by using a high-level model rather than a detailed simulation, we were able to search over a much larger space of design choices. While a high-level performance model does not accurately predict absolute runtimes it can effectively choose between alternative designs (see [17–19] for a similar use of a performance model).

It has been shown that high-level performance models are an important tool in hardware design [20, 34, 35], since they allow various design choices to be evaluated early in the design process. Moreover, design tools such as ADEPT (Advanced Design Environment Prototyping Tool) [19], which is implemented in VHDL, allow a smooth transition from the performance model to a functional model due to the unified environment. Although the application of performance models are typically used to help designers choose appropriate design partitions and components [17,18], they can be applied to other design considerations. We present an application where a performance model is used to adapt the underlying algorithm in order to maximize performance. A large collection of algorithmic choices are systematically generated and the performance model is used to select the algorithm that is predicted to lead to the design of an application specific processor with the best performance. In particular, a distributed memory fast Fourier transform (FFT) processor is optimized by searching in the space of FFT algorithms for the algorithm that best utilizes the underlying memory architecture.

4.1 Search Problem Statement

The systematic mapping of an FFT algorithm described in mathematical formula to the architecture described in Chapter 3 provides us a well-defined optimization problem.

The FFT processor we designed is built using the distributed-memory architecture shown in Figure 3.1. An dimensionless FFT algorithm of size 2^n points described by Equation 2-52 is mapped to 2^m processors using the method shown in Chapter 3. Since the mapping procedure is fixed, different FFT algorithms result in different schedules. In return, different FFT algorithms lead to implementations with different performance. This leads to an optimization problem which is the focus of this chapter.

When mapped to the proposed architecture, an dimensionless FFT algorithm of size 2^n points comprises 2^{n-1} tasks, where a task is 2-point butterfly operation with twiddle factor described by Equation 3-1. Each task consists of three variables namely two addresses and a twiddle factor. However, the twiddle factors are generated inside each processor. Therefore, they have no effect over the performance. In other words, only a sequence of addresses effects the performance. Since the FFT dataflow described by P_i, $1 \leq i \leq n$, determines the sequence of addresses, the optimization problem becomes "which FFT dataflow yields the best performance".

With the restriction that only tensor permutations are allowed, the space of possible FFT dataflows are those with $P_i = (Q_i \otimes I_2)L^N_{2^{i-1}}$, where Q_i is an arbitrary tensor permutation of size $N/2$. Since there are n stages and each stage has $(n-1)!$ possible permutations, the size of the space of FFT algorithm is equal to $(n-1)!^n$. Then, the search problem is to find an FFT dataflow specified by a set of $P_i = (Q_i \otimes I_2)L^N_{2^{i-1}}$, $1 \leq i \leq n$, that produces the best performance. In the next section, we describe how performance model is used to estimate the performance cost corresponding to an FFT dataflow.

4.2 Performance Model

Performance of an algorithm from Equation 2-52 is determined by its memory access pattern. Different access patterns can lead to differing amounts locality and contention of the interconnection network. A performance model was constructed to simulate the flow of data through the architecture and to capture the memory access delays due to non-local access and contention.

4.2.1 Token Protocol

In the performance model, data are represented by tokens, and the flow of the tokens emulates the dataflow. The model was implemented using the Advanced Design

Environment Prototype Tool's (ADEPT) developed by University of Virginia [19]. ADEPT is based on Petri-Nets [36] and is implemented in VHDL.

Tokens in ADEPT have two fields; a status field and a color field. Status provides a protocol for passing tokens through the system and color provides additional information for the particular model. The values of status are "removed", "presented", "acked" and "released". A token is passed between 2 models by changing the status field of a signal connecting two models. A new token is allowed to be sent only if the previous token has been removed. A sender sends a token by setting the status field to "presented". When the receiver is ready to receive the token, it sets the status field to "acked". Immediately, the sender releases the token by setting the status field to "released". Then, the receiver removes the token by setting the status field to "removed" allowing the next token to be sent. In our model, the color field is used to store information such as the pair of addresses required for a butterfly operation.

4.2.2 Performance Model for a Distributed-Memory FFT Processor

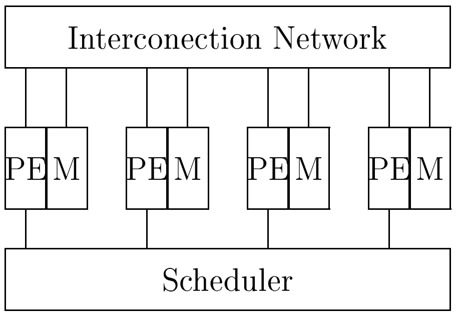


Figure 4.1 Top-level performance model

Figure 4.1 shows the top-level performance model of our architecture with 4 processors. The scheduler creates a sequence of tasks and distributes them in a round-robin fashion to the different processing elements. Each task consists of a pair of addresses needed for a two-point butterfly operation. In order to compute the correct result a task would have to include twiddle factor information; however, this is not needed for simulating performance. The sequence of tasks for a specific FFT are determined by the permutations in Equation 2-52. For an $N = 2^n$ point FFT the tasks are generated using an n-bit counter. For the i-th stage of the FFT, the permutation P_i is determined by a permutation, σ_i, of the n bits $b_{n-1} \cdots b_0$, generated by the counter. The counter generates all possible bit sequences which are permuted by σ to obtain the addresses. Each task contains the two addresses $b_{\sigma(n-1)} \cdots b_{\sigma(0)}$ with b_0 equal to 0 and 1.

For example, tasks in the 5-th stage of the 64-point Pease algorithm are determined by $\sigma_5 = (0 \rightarrow 1, 1 \rightarrow 0, 2 \rightarrow 5, 3 \rightarrow 4, 4 \rightarrow 3, 5 \rightarrow 2)$. The sequence of addresses corresponding to σ_5 is 0, 16, 32, 48, 1, 17, 33, 49, …, 15, 31, 47, and 63, and the corresponding tasks are (0,16), (32,48), (1,17), (33,49), …, (15,31) and (47,63). These tokens are distributed to the PEs in round-robin. For our example, PE0 will get 8 tokens with (0,16), (2,18), (4,20), (6,22), (8,24), (10,26), (12,28), (14, 30) while PE1 gets tokens with (32,48), (34,50), (36,52), (38,54), (40,56), (42,58), (44,60), (46,62). At the same time, PE2 gets (1,17), (3,19), (5,21), (7,23), (9,25), (11,27), (13,29), (15, 31) and PE3 gets (33,49), (35,51), (37,53), (39,55), (41,57), (43,59), (45,61), (47,63).

After a task is generated by the scheduler, the token containing the task is sent to the PE model. The addresses stored in the task are used to generate additional tokens that simulate memory reads and and writes and the computation of the butterfly operation. Figure 4.2 shows the performance model of each PE connected to its local memory and the interconnection network.

The input sequencer extracts the addresses of the task and forwards them to the memory sequencer to initiate the corresponding memory reads. Additional memory

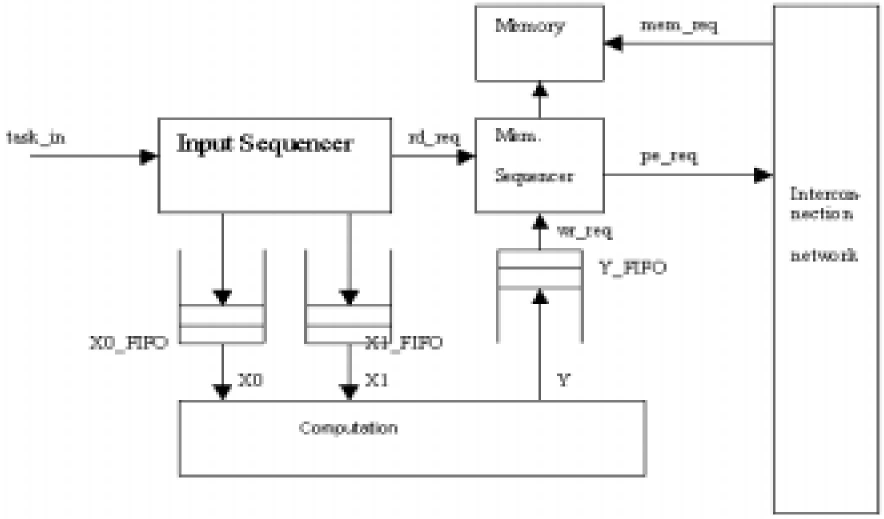


Figure 4.2 PE performance model

requests are presented to the memory sequencer from the “Y FIFO” which contains write requests generated by the computation unit. Memory addresses are distributed to the separate PE memories in a block cyclic fashion. Assume the size of the input vector is 2^n and the number of processors is 2^m. Using this scheme the high-order m bits of the n-bit address is the identifier of the memory (MID) containing the address and the low order $n - m$ bits is an offset into the processor memory. The memory sequencer extracts the MID and offset and determines if the address is local or remote. A local address request is sent directly to memory and a remote access is sent to the corresponding memory unit via the interconnection network.

The interconnection network is modeled as a bus. Only one request can be processed at a time. Access to the bus is granted by arrival order and if there are simultaneous requests a round-robin priority scheme is used. If PE number i currently has the highest priority the priority is changed to PE number $(i + 1) \bmod P$ where P is the number of processors, after the next request from PE number i is granted.

After a read request has completed the corresponding data is placed in a FIFO (either X0 or X1 depending on whether it is the first or second address in the task). When both elements arrive in the corresponding FIFOs the task is sent to the computation unit. Computation is modeled by a delay and after the delay two write requests are placed in the "Y FIFO" to be sent to the memory sequencer.

In order to measure performance we model memory access, bus access, and computation by the delays D_m, D_b, and D_c respectively. The performance cost is simulation time which includes these delays plus any time spent waiting for a request to be granted. The computation is organized into stages and the total simulation time is the sum of the time spent on each stage.

To illustrate how the performance model distinguishes different FFT dataflows, we compare two different dataflows for a 64-point FFT. The first is the Pease dataflow shown in Chapter 2.1 and the second is the optimal dataflow discovered by the search in the following section. Table 4.1 shows number of local and remote memory accesses at each stage of both dataflows. Table 4.2 shows the performance cost from the simulation with 3 different delay parameters. Notice that at stage 5, both dataflow gives the same number of local (64) and remote (64) memory accesses. However, the optimal dataflow gives a better performance (384 v.s. 258). The improved performance is due to less memory contention and is captured in the model by the reduction in time spent waiting for memory requests to be granted.

Table 4.1 Local vs. remote memory accesses for Pease and optimal dataflow

Stage	Pease algorithm		Optimal algorithm	
	Local	Remote	Local	Remote
1	32	96	128	0
2	32	96	128	0
3	64	64	128	0
4	128	0	128	0
5	64	64	64	64
6	32	96	64	64
Total	352	416	640	128

Table 4.2 Performance of 64-point Pease and optimal dataflow

Unit = clock cycle

Stage i	D_c = 20 D_b = 2 D_m = 2		D_c = 1 D_b = 2 D_m = 2		D_c = 1 D_b = 0 D_m = 2	
	Pease	Opt.	Pease	Opt.	Pease	Opt.
1	388	168	396	64	238	64
2	386	168	386	64	216	64
3	384	168	390	64	210	64
4	168	168	64	64	64	64
5	384	274	384	282	204	184
6	386	258	392	258	230	174
Total	2096	1204	2012	790	1154	610

4.3 Search and Results Analysis

Using the performance model from the previous section we can systematically search for the optimal FFT dataflow given by Equation 2-52. This section summarizes the results from this search. We show that there is a wide range in performance and that the optimal algorithm is considerably better than the Pease algorithm or the expected performance from the search space. Since the size of the search space is $(n-1)!^n$ an exhaustive search is out of the question for all be the smallest values of n. However, since each stage is independent it is only necessary to search for the optimal dataflow in each stage, which reduces the search space to $n(n-1)! = n!$. Using this reduced search space, we empirically found the optimal algorithm up to size $N = 2^{10}$. These results led us to consider an algorithm that is conjectured to be optimal in general.

Figure 4.3 shows the distribution of performance for all permutations occurring in stage 5 of a 64-point FFT with 4, 8, and 16 processors. Using similar results from the other stages we constructed an optimal algorithm. Similar constructions were carried out for $n = 7, 8, 9$ and 10. Table 4.3 compares the performance of these optimal

Table 4.3 Performance of 64-point Pease algorithm and optimal dataflow with 4 processors and different delay parameters

m	Pease/Optimal				
	n=6	n = 7	n = 8	n = 9	n = 10
2	1.74	1.86	1.95	2.01	2.07
3	2.23	2.48	2.68	2.87	3.04
4	2.14	2.50	2.80	3.10	3.37

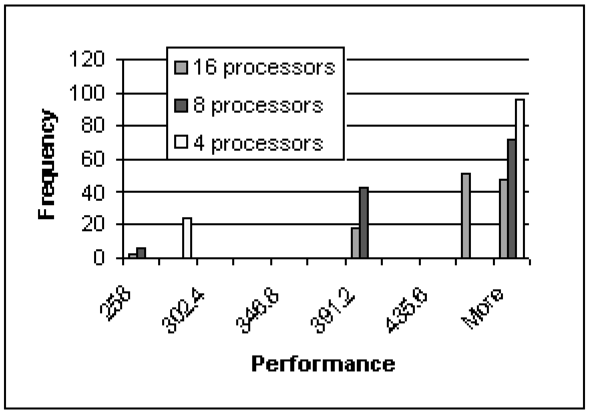


Figure 4.3 Performance histogram for all stage 5 permutations

algorithms to the Pease algorithm. Figure 4.4 shows the distribution of performance

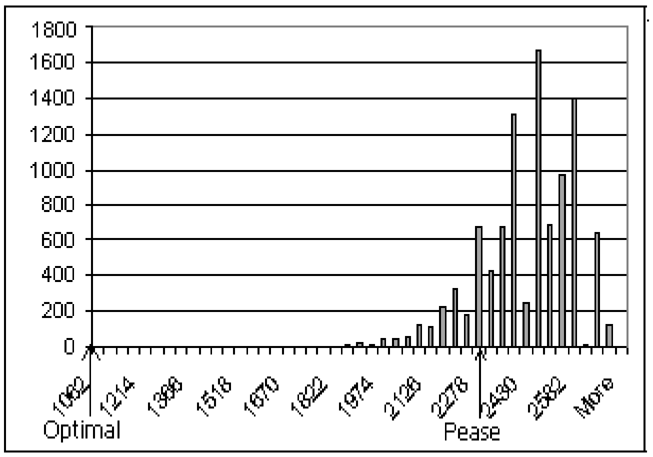


Figure 4.4 Performance distribution 10,000 random dataflows

for 10,000 random algorithms of size 64 using 8 processors. The location of the optimal and Pease algorithms are indicated. This shows that an optimal algorithm is very rare. The resulting optimal algorithms and their generalization are described in the next section.

4.4 The Optimal Algorithm

After exhaustively searching over all possible permutations in each stage for FFT of size 2^6 to 2^{10} points, we use the pattern of the optimal permutation to conjecture the optimal algorithm of size 2^n on 2^m processors. Table 4.4 shows the conjectured optimal algorithm described by permutations of degree n. Table 4.5 shows the permuted bits for generating source local addresses, target PID and source MID for optimal algorithm obtained by using the mapping method describe in Chapter 3.

Following the twiddle factor mapping explained in Chapter 3, the bit patterns for generating twiddle fractions for one-dimensional DFT using the optimal is shown in

Table 4.4 Permutations σ_i and σ_i^{-1} specifying the optimal algorithm of size 2^n points on 2^m processors

Stage i	σ_i and σ_i^{-1}	
1	σ_1	$= (0, n-m-1, \cdots, n-1, 1, \cdots, n-m-2)$
	σ_1^{-1}	$= (0, m+1, \cdots, n-1, 1, \cdots, m)$
2	σ_2	$= (1, n-m, \cdots, n-1, 0, 2, \cdots, n-m-1)$
	σ_2^{-1}	$= (m+1, 0, m+2, \cdots, n-1, 1, \cdots, m)$
$\vdots$	$\vdots$	
i	σ_i	$= (i-1, n-m, \cdots, n-1, 0, \cdots, i-2, i, \cdots, n-m-1)$
	σ_i^{-1}	$= (m+1, \cdots, m+i-1, 0, m+i, \cdots, n-1, 1, \cdots, m)$
$\vdots$	$\vdots$	
n-m	σ_{n-m}	$= (n-m-1, \cdots, n-1, 0, \cdots, n-m-2)$
	σ_{n-m}^{-1}	$= (m+1, \cdots, n-1, 0, 1, \cdots, m)$
n-m+1	σ_{n-m+1}	$= (n-m-1, n-m-2, n-m, \cdots, n-1, 0, \cdots, n-m-3)$
	σ_{n-m+1}^{-1}	$= (m+1, \cdots, n-1, 1, 0, 2, \cdots, m)$
$\vdots$	$\vdots$	
n-m+j	σ_{n-m+j}	$= (i-1, n-m-1, \cdots, i-2, n-m-2, i, \cdots, n-1, 0, \cdots, n-m-3)$
	σ_{n-m+j}^{-1}	$= (m+1, \cdots, n-1, j, 1, \cdots, j-1, 0, j+1, \cdots, m)$
$\vdots$	$\vdots$	
n	σ_{n-1}	$= (n-1, n-m-1, \cdots, n-2, n-m-2, 0, \cdots, n-m-3)$
	σ_{n-1}^{-1}	$= (m+1, \cdots, n-1, m, 1, \cdots, m-1, 0)$

Table 4.6. Note that the twiddle fractions for t-dimensional DFT is generated by masking off the twiddle fractions for the one-dimensional case (Section 3.2.5.)

Note that the target PID and the source MID is equal to the PE number during stage 1 to stage $n-m$. This mean that all memory accesses are local. However, for stage i, $n-m \leq i \leq n$, there is only one bit difference between the target PID or the source MID and the processor number. This means that half of data assigned to a processor comes from itself and the other half comes from another processor throughout the whole stage.

Table 4.5 Permutation bits the optimal algorithm of size 2^n points on 2^m-processor system

`Stage i`	`Source MID = Target PID`	`Local Address`
1	$p_{m-1} \cdots p_0$	$a_{n-m-1} \cdots a_1 a_0$
2	$p_{m-1} \cdots p_0$	$a_{n-m-1} \cdots a_2 a_0 a_1$
$\vdots$	$\vdots$	$\vdots$
i	$p_{m-1} \cdots p_0$	$a_{n-m-1} \cdots a_i a_0 a_{i-1} \cdots a_1$
$\vdots$	$\vdots$	$\vdots$
$n-m$	$p_{m-1} \cdots p_0$	$a_0 a_{n-m-1} \cdots a_2 a_1$
$n-m+1$	$p_{m-1} \cdots a_0$	$a_0 a_{n-m-1} \cdots a_2 a_1$
$\vdots$	$\vdots$	$\vdots$
$n-m+j$	$p_{m-1} \cdots p_{j+1} a_0 p_{j-1} \cdots p_0$	$a_0 a_{n-m-1} \cdots a_2 a_1$
$\vdots$	$\vdots$	$\vdots$
n	$a_0 p_{m-2} \cdots p_0$	$a_0 a_{n-m-1} \cdots a_2 a_1$

In addition to good memory access pattern, the optimal algorithm has other interesting properties that simplify its implementation. In the next chapter, we will explain how the optimal algorithm simplifies the implementation.

Table 4.7 shows the permutation σ_i, the bit patterns for generating local addresses, target PID, source MID, and twiddle fractions using the optimal algorithms of size 2^6 points on 2^3 processors.

Table 4.6 Twiddle fractions for one-dimensional DFT of size 2^n using the optimal algorithm

Stage	One-dimensional Twiddle Fraction
1	$0.00\cdots0$
2	$0.0b_{m+1}0\cdots0$
$\vdots$	$\vdots$
i	$0.0b_{m+i-1}\cdots b_{m+1}0\cdots0$
$\vdots$	$\vdots$
n-m	$0.0b_{n-1}\cdots b_{m+1}0\cdots0$
n-m+1	$0.0b_1b_{n-1}\cdots b_{m+1}0\cdots0$
$\vdots$	$\vdots$
n-m+t	$0.0b_{t-1}\cdots b_1b_tb_{n-1}\cdots b_{m+1}0\cdots0$
$\vdots$	$\vdots$
n	$0.0b_{m-1}\cdots b_1b_mb_{n-1}\cdots b_{m+1}0\cdots0$

Table 4.7 Bit patterns for generating local addresses, target PID, source MID, and twiddle fractions using the optimal algorithms of size 2^6 points on 2^3 processors

Stage i	σ_i and σ_i^{-1}	Source MID = Target PID	Local Address	Twiddle fractions ($\mathbf{TF1}_i$)
1	$\sigma_1 = (0,2,3,4,5,1)$	$(p_2p_1p_0)_2$	$(a_2a_1a_0)_2$	$0.0\cdots0$
	$\sigma_1^{-1} = (0,4,5,1,2,3)$			
2	$\sigma_2 = (1,3,4,5,0,2)$	$(p_2p_1p_0)_2$	$(a_2a_0a_1)_2$	$0.0a_10\cdots0$
	$\sigma_2^{-1} = (4,0,5,1,2,3)$			
3	$\sigma_3 = (2,3,4,5,0,1)$	$(p_2p_1p_0)_2$	$(a_0a_2a_1)_2$	$0.0a_2a_10\cdots0$
	$\sigma_3^{-1} = (4,5,0,1,2,3)$			
4	$\sigma_4 = (3,2,4,5,0,1)$	$(p_2p_1a_0)_2$	$(a_0a_2a_1)_2$	$0.0p_0a_2a_10\cdots0$
	$\sigma_4^{-1} = (4,5,1,0,2,3)$			
5	$\sigma_5 = (4,3,2,5,0,1)$	$(p_2a_0p_0)_2$	$(a_0a_2a_1)_2$	$0.0p_0p_1a_2a_10\cdots0$
	$\sigma_5^{-1} = (4,5,2,1,0,3)$			
6	$\sigma_6 = (5,3,4,2,0,1)$	$(a_0p_1p_0)_2$	$(a_0a_2a_1)_2$	$0.0p_0p_1p_2a_2a_10\cdots0$
	$\sigma_6^{-1} = (4,5,3,1,2,0)$			

5.0 IMPLEMENTATION OF THE FFT PROCESSOR

As shown in Chapter 3, an FFT algorithm can be systematically mapped to the architecture shown in Figure 5.1. Based on the mapping, each algorithm produces a corresponding hardware. In Chapter 4, we search for the algorithm that will give the best performance. In this chapter, we describe the implementation of the universal FFT provided a choice of algorithm.

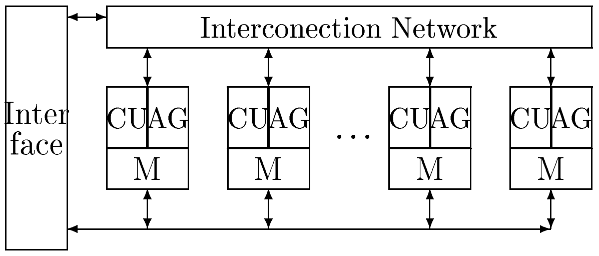


Figure 5.1 The FFT Engine architecture

Based on the architecture and mapping methodology in Chapter 3, all 3 parts of the architecture (the I/O interface, the interconnection network and the processor elements) are parameterized by 3 sets of matrices describing an FFT algorithm. In particular, the I/O interface unit who is responsible for loading the input data and uploading the result is parameterized by the initial permutation P_0; the interconnection network depends on the internal permutations P_i, $1 \leq i \leq n$, and the processor elements are parameterized by both the internal permutations and the twiddle factor matrices T_i, $1 \leq i \leq n$. We will discuss how these units are implemented.

All units that are parameterized by the matrices depends on the generation of numbers permuted by a tensor permutation, we introduce two implementation techniques of such generator in Section 5.1. In Section 5.2, we describe the universal FFT engine specification. The implementation of the I/O interface is described in

Section 5.3. We discuss the implementation of the interconnection network in Section 5.4. Section 5.5.1 and 5.5.2 describes the implementation the computation unit (CU) and the address generator (AG) of the processor elements respectively. We conclude the chapter by discussing the implementation of design in Wildforce^{TM} FPGA board [23] for proof of concept.

5.1 Implementation of Tensor Permutations

As shown in Section 2.1, a sequence of numbers permuted by a tensor permutation matrix of size $2^n \times 2^n$ can be generated using n-bit counter whose output bits are permuted by a permutation of degree n. In essence, we can specify how the permuted numbers are generated by a permuted bit pattern. For instance, $(b_0 \cdots b_{n-1})_2$ specifies the generation of bit-reversal permutation matrix of size $2^n \times 2^n$. In Chapter 3, we show that the relabeling of input during the loading, the address generation, and the twiddle fraction generation are derived from a tensor permutation matrix and can be specified by a permuted bits pattern. We propose two implementation techniques for generating the permuted numbers. The first technique uses an array of MUXs to permute the counter's output bits. The second method uses an adder (accumulator) with adaptable increment numbers. The following subsections describes these two techniques.

5.1.1 Implementation using MUXs

Given a permuted bit, a straightforward implementation of the tensor permutation matrix is by using multiplexers (MUXs) to realize the permutation of the counter's output bits. Figure 5.2 shows the diagram of the implementation.

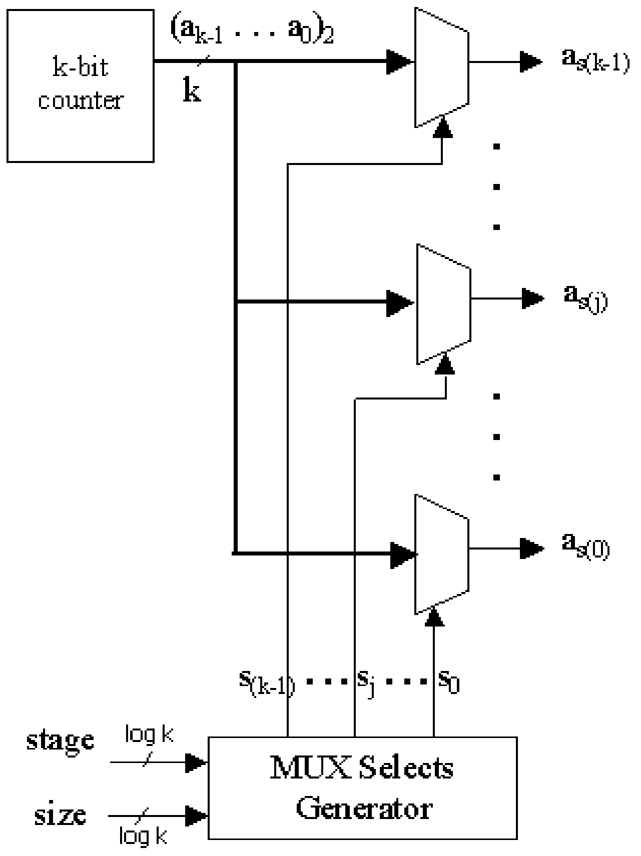


Figure 5.2 Implementation of tensor permutation using MUXs

Note that if the generator is built for only one permuted bit pattern of a fixed size, we can hardwire the permuted bits. However, that is not the case of the generators in the universal FFT processor. Firstly, there are multiple stages with different permuted bit patterns. Secondly, a generator must be able to handle different sizes of FFT. Since the permuted bit pattern for different stages depend on the set of permutations specifying the algorithm, they are compile-time parameters. However, the size is a run-time parameter.

Let $2^{n_{max}}$ and 2^m be the maximum number of points and number of processors respectively. Then, we need a $(n_{max} - m)$-bit counter in each processor. The size of the MUXs can be varied depending on the algorithm.

Let $n_{max} - m = k$ and $(a_{k-1} \cdots a_0)_2$ be the output bits of the counter. In general, an array of k k-to-1 MUXs can be used to implemented any possible permuted bit pattern.

An k-to-1 MUX selects one of k inputs as its output. Let the input signal that controls the MUX's output be called MUX select. Let a_j, $0 \leq j < n_m$, be the j^{th} input of a k-to-1 MUX. Let $\mathbf{M}_j$ be the j^{th} MUX of an array of k MUXs and let $s(j)$, $0 \leq s(j) < k$, be the MUX's select of M_j. Then, $a_{s(j)}$ is the output of M_j and $(a_{s(k-1)} \ldots a_{s(0)})_2$ is the output of the array of the MUXs. In other words, the MUXs permute $(a_{k-1} \ldots a_0)_2$ to $(a_{s(k-1)} \ldots a_{s(0)})_2$.

The MUXs selects are generated accordingly to a set of permuted bit patterns which is derived from a set of permutations. For example, for $n_{max} - m$ and $m = 2$, there are $n_{max} - m + 1 = 9$ possible sizes. Since there are n set of MUX selects for size n, $2 \leq n \leq 10$, there are totally $\sum_{n=m}^{n_{max}} n$ sets of MUX selects. However, since the MUX selects are based on the same set of permutations, they are related. Hence, they can be generated based on the relationship. In Section 5.5.4 we show how MUX selects of the Pease and the optimal algorithms can be generated.

5.1.2 Implementation using Adders

In many cases where the permuted bit pattern contains only a few portions of consecutive bits, using adders to generate the permuted numbers has some advantages over using MUXs. The most important advantage is when n is large. This is because with a large n, using MUXs can take a lot of space and long delay compared to adders.

Figure 5.3 shows a basic form of an adder for generating permuted numbers. The base address (INIT), and increment (INC) depends on the permuted bit pattern.

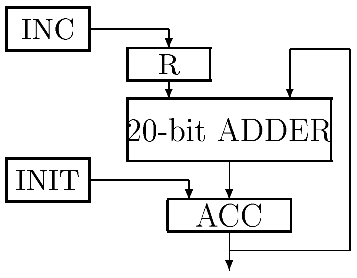


Figure 5.3 Adder used for generating sequence of permuted numbers

Specifically, they depends on the number of portions of consecutive bits. For example, the permuted bit pattern $(a_3a_2a_1a_0a_5a_4)_2$ contains two portions of consecutive bits: $(a_3a_2a_1a_0)_2$ and $(a_5a_4)_2$. Therefore, two increment numbers are needed.

There are two properties when incrementing a binary number $(a_{k-1}\cdots a_0)_2$ by 1. Firstly, the value of a_0 is flip-flopped between 0 and 1. Secondly, the remaining of the bits changes accordingly to the carry propagation. When the bits are permuted, we need to keep these two properties intact by providing the right increment number instead of 1.

We can keep the flip-flop property of a_0 by aligning '1' with a_0. For example, let the permuted bit pattern be $(a_3a_2a_1a_0a_5a_4)_2$, then aligning 1 with a_0 results in the increment number equal to $(000100)_2$. Adding $(000100)_2$ to the current value changes the value of a_0 between 0 and 1. The carry propagation, however, are more difficult to maintain. The following illustrates the technique.

Example: Let n_{max} be the maximum size of the generator using adder. This means the size of the adder is n_{max} bits. Let us consider the permuted bit pattern $(a_3a_2a_1a_0a_6a_5a_4)_2$. Notice that there are 2 portions of consecutive bits. The first portion is $(a_3\cdots a_0)$ containing 4 bits. The second portion is $(a_6a_5a_4)$ containing 3 bits. The flip-flop property dictates that '1' should be aligned with a_0. Therefore, the n_{max}-bit $(0\cdots01000)_2$ is the increment number when the carry propagation is not broken. This is true whenever there is no carry from a_3 to a_4. The only time that this carry occurs is when the current value of the portion $(a_3 \cdots a_0)$ is equal to (1111). Since we start from address $(0 \cdots 0)_2$, this will occur in every 2^4 addresses. In other words, it happens in stride 2^4, where 4 is the number of bits in the first portion of consecutive bits. We will refer to this number as a "stride number".

The stride number signifies that the increment number must be changed in order to keep the integrity of the carry propagation from a_3 to a_4. Since it occurs only when $(a_3 \cdots a_0) = (1111)_2$, the current addresses are equal to $(0 \cdots 01111a_6a_5a_4)_2$. The next address obtained by adding '1' to $(0 \cdots 0a_6a_5a_4a_3a_2a_1a_0)$ is equal to $(0 \cdots 00000a_6a_5a_4)_2 + 1$. Then, the different between the next value and the current value is the increment number; i.e.

$$\begin{aligned}
\mathtt{INC} &= ((0 \cdots 00000a_6a_5a_4)_2 + (0 \cdots 01)_2 - (0 \cdots 01111a_6a_5a_4)_2 \\
&= ((0 \cdots 00000a_6a_5a_4)_2 - (0 \cdots 01111a_6a_5a_4)_2) + (0 \cdots 01)_2 \\
&= (0 \cdots 01)_2 - (0 \cdots 01111000)_2 \\
&= (1 \cdots 10001001)_2 = -2^7 + 2^3 + 1
\end{aligned}$$

In conclusion, to generate the permuted numbers parameterized by the permuted bit pattern $(a_3a_2a_1a_0a_6a_5a_4)_2$, we need 2 increment numbers: $(0 \cdots 01000)_2 = 2^4$ and $(1 \cdots 10001001)_2 = -2^7 + 2^3 + 1$. If $(a_3a_2a_1) = (1111)_2$, the increment is $(1 \cdots 10001001)_2$; otherwise the increment is $(0 \cdots 01000)_2$. $\square$

Based on this implementation, changing increment numbers is done in stride 2^s. The adder shown in Figure 5.3 can also be used as the stride 2^s counter, where there is the carry out in every 2^s numbers. Let n_{max} be number of bits of the adder, then setting the increment number to $2^{n_{max}-s}$ generates the carry out in every 2^s numbers (or in stride 2^s.)

Example: Let $n_{max} = 10$ and $s = 3$. Then, setting the increment number to $2^{10-3} = (0010000000)_2$ generates a carry out in every 8 numbers. □

Note that the base address (INIT) is usually equal to 0.

5.2 Universal FFT Engine

In order to understand the implementation of each unit, let us first look at the big picture of the FFT engine. Let us assume that there exists a protocol of I/O interface between a user denoted as "Host" in Figure 5.1 and the FFT engine. Such protocol provides a way to transfer data between the user and the FFT engine. One example of such protocol is the PCI bus by which a FPGA board communicates with a PC acting as a host. Then, the host starts using the FFT engine by sending run-time parameters followed by the sequence of input data. Once the host finishes the loading, it waits for a "done" signal from the FFT engine. Once it receives the signal, it uploads the result from the FFT engine. The run-time parameters include number of points (2^n), n, number of dimensions (t), and size of each dimension ($n_1, \cdots, n_t$).

The FFT engine is either in ready or computing states. If it is in ready state, the first data sent to it by the host is considered the first run-time parameter. Then, it will start retrieving the next parameters and distributes them to all PEs. Once the run-time parameters are set, it starts loading the input data and computing the FFT. Once it is done, it sends the "done" signal to host and goes into the ready state waiting for the next parameters.

The I/O interface unit is responsible for this interaction between the user and the FFT engine. The following section describes the I/O interface unit.

5.3 I/O Interface Unit

The I/O interface unit functions are (1) receive parameters, (2) convert the parameters to the values used by each unit, (3) retrieving and distributing the input data to memory modules, (4) wait for all PEs to finish their jobs and (5) send the "done" signal to host when all computations are done. Figure 5.4 shows the diagram of the I/O interface unit. The parameters and input data are put do the input FIFO

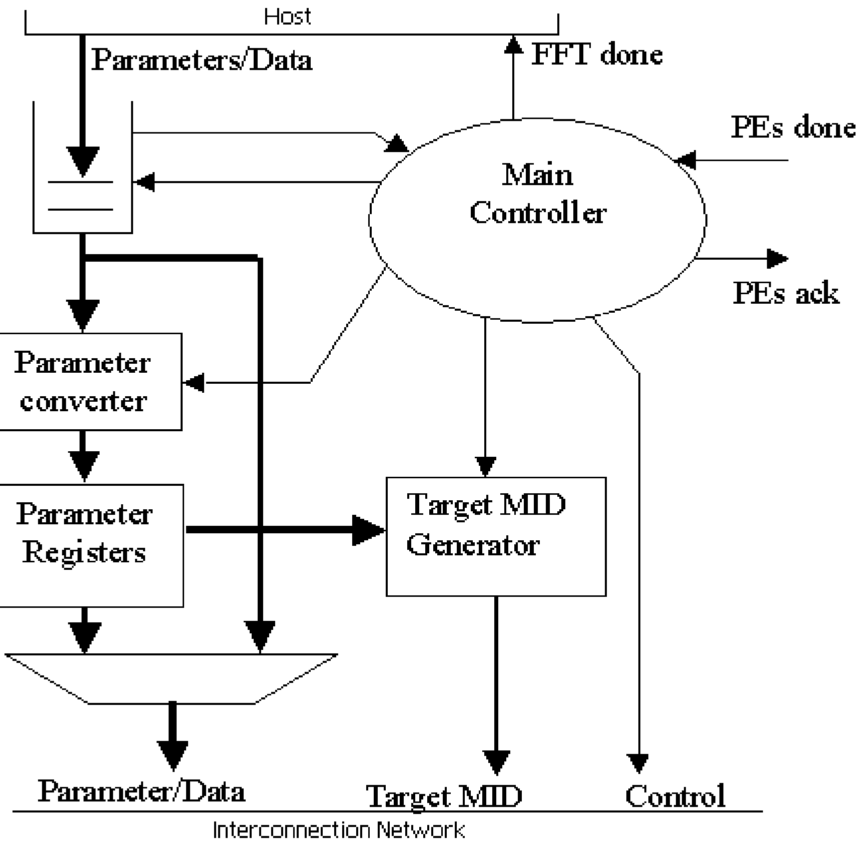


Figure 5.4 I/O interface Unit

by the host. If the input is a parameter, it is converted by the "parameter converter" unit and kept in a register. If it is the input data, it is sent directly to the interconnection network. The MUX is used to select either the parameters or input data.

The unit called "Target MID Generator" generates the MID to which the data are sent. All these jobs are controlled by the main controller.

The only unit that is parameterized by P_0 is "Target MID Generator." For the class of algorithm under consideration, P_0 depend only on the dimension specification; i.e. it depends on the number of dimensions (t) and $n_1, \cdots, n_t$. In Section 3.2.1 we shown that the target PID is the m most significant bits of the permuted bits permuted by σ_0. This is equal to $(b_{\sigma_0^{-1}(n-1)} \cdots b_{\sigma_0^{-1}(n-m)})_2$, where σ_0^{-1} is the permutation of degree n specifying $R_{2^{n_1}} \otimes \cdots \otimes R_{2^{n_t}}$. Since $R_{2^{n_i}}$ is a bit reversal permutation of size n_i bits,

$$\sigma_0^{-1} = (n_t - 1, \cdots, 0, n_t - n_{t-1} - 1, \cdots, n_t, \cdots, n-1, \cdots n - n_1)$$

Example: To illustrate σ_0^{-1}, consider the case where $n = 10$, $t = 3$, $n_1 = 4$, $n_2 = 3$ and $n_3 = 3$. Then, $\sigma_0^{-1} = (2, 1, 0, 5, 4, 3, 9, 8, 7, 6)$. Therefore, for 4-processor system ($m = 2$), the target MID is $(b_6 b_7)_2$. □

We can implement the "Target MID Generator" using adders as shown in Section 5.1.2. The following example illustrates the "Target MID Generator".

Example: Assume that the FFT engine can computed up to 2^{20} points and maximum number of dimension is 3. Then, we can generate the target MID for distributing input data using 3 accumulators (adders) as shown in Figure 5.5. The "adder1" has 3 possible increment numbers: $\texttt{INC1} = 2^{n_1+n_2}$, $\texttt{INC2} = 2^{n_1+n_2} + 2^{n_1}$ and $\texttt{INC3} = 2^{n_1} + 1$.

Since the permutation $R_{2^{n_1}} \otimes R_{2^{n_2}} \otimes R_{2^{n_3}}$ contains 3 sections of bit-reversal permutations. The first section permutes $(b_{n_3-1} \cdots 0)_2$ to $(b_0 \cdots b_{n_3-1})_2$. The second section permutes $(b_{n_2+n_3-1} \cdots b_{n_3})_2$ to $(b_{n_3} \cdots b_{n_2+n_3-1})_2$ and the last section permute

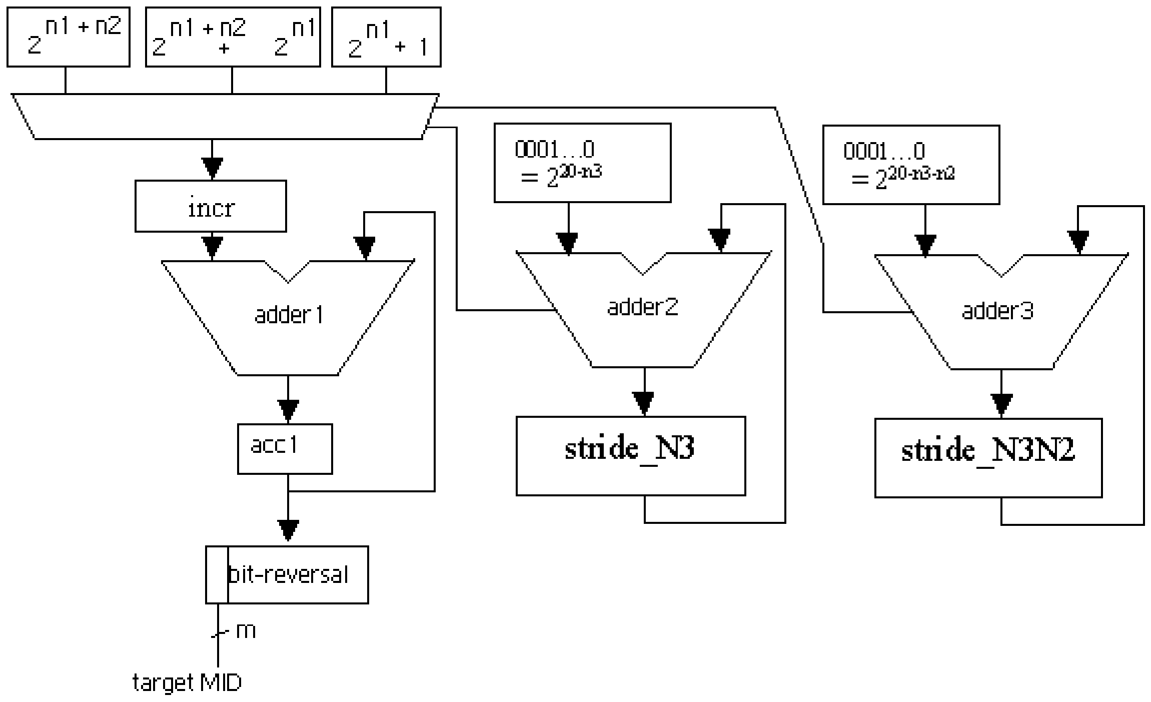


Figure 5.5 Target MID Generator for FFT Engine capable of computing 1-D, 2-D or 3-D DFT

$(b_{n-1} \cdots b_{n_2+n3})_2$ to $(b_{n_2+n_3} \cdots b_{n-1})_2$. Combining the three sections, we have the permuted bits equal to

$$(b_{n_2+n_3} \cdots b_{n-1})(b_{n_3} \cdots b_{n_2+n_3-1})(b_0 \cdots b_{n_3-1})$$

The m-most significant bits of the above permuted bits are the target MID. Notice that the above permuted bits is equal to reversing the following bit orders.

$$(b_{n_3-1} \cdots b_0)(b_{n_2+n_3-1} \cdots b_{n_3})(b_{n-1} \cdots b_{n_2+3}).$$

A number corresponding to the above permuted bit pattern can be generated by adding either `INC1`, `INC2` or `INC3` to the previous value starting from 0. This follows the technique described in Section 5.1.2.

The selection of the increment number works as following. The `INC3` has the highest priority. It is selected in stride $2^{n_2+n_3}$. The "adder3" produces the stride signal for selecting `INC3`. The `INC2` has second priority. It is selected in stride 2^{n_3} except when `INC3` is selected. The "adder2" generates the stride signal for selecting `INC2`. If both strides are not active, the `INC1` is selected.

Using the technique describes in Section 5.1.2, we find that

$$\begin{aligned} \texttt{INC1} &= (0\cdots010\cdots0)_2 = 2^{n_1+n_2}, \\ \texttt{INC2} &= (0\cdots010\cdots010\cdots0)_2 = 2^{n_1+n_2} + 2^{n_1}, \\ \texttt{INC3} &= (0\cdots010\cdots01)_2 = 2^{n_1} + 1. \end{aligned}$$

□

5.4 Interconnection Network

Regardless of the choice of algorithm, the interconnection network must be able to broadcast the parameters or input data to all PEs. Therefore, the first configuration of the network is that the I/O interface unit is the sender and all PEs are the receivers. All PEs take parameters, but receive only the data that are belong them. A data belong to $\mathbf{PE}_r$ if the target MID sent along with it is equal to r.

The other network configurations depend on the choice of algorithm. This is where the optimal algorithm shows an advantage in addition to giving the best performance.

Let 2^m be number of processors. Then, for computing FFT of size 2^n using the optimal algorithm, in addition to broadcast by the I/O interface unit, it requires only m network configurations, where each configuration has two PEs communicate in pair. Moreover, a network configuration stays the same through out the stage; hence, there are only m change of configurations.

Note that for other choice of algorithm, the communication pattern is also deterministic but not necessary as simple.

5.5 Processor Element

There are two parts of a processor element, the computation unit (CU) and the address generator (AG). Figure 5.6 shows the interface between the two units. The

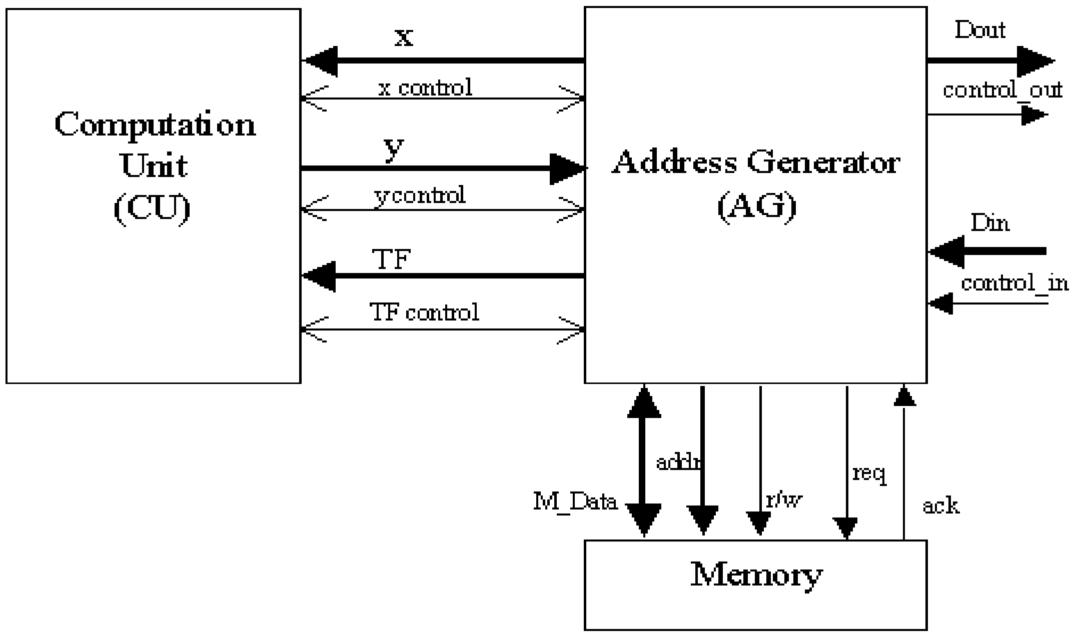


Figure 5.6 Processor Element

computation unit is responsible for generating twiddle factors and computing butterfly operation. The address generator is responsible for scheduling the input data and twiddle fractions to the computation unit and storing the result. Let us first consider the computation unit.

5.5.1 Computation Unit (CU)

As described in previous chapters, we base our design on radix-2 dimensionless FFT algorithms. A primitive computation in such algorithm is a 2-point butterfly operation described by Equation 3-1.

Although the FFT algorithm is defined over various fields that contain a N^{th} root of unit, we consider only FFT algorithm of vector space of the complex field. In this case, we need at least a complex adder and a complex multiplier. There are also choices of representing complex data, floating point or fixed point. In fixed-point system, implementation of arithmetic units (adder and multiplier) are usually simpler, smaller and faster. However, it works within a smaller range of data. Contrarily, in floating-point representation, the implementation of arithmetic units are more complicated, bigger and slower, but it covers a much larger data range.

To cover the larger data range, we choose to implement the floating point system. An floating point adder and floating point multiplier are designed following the IEEE standard (IEEE Std. 754-1985) for single-precision floating point number [37, 38]. Pipeline technique is employed in both floating point adder and floating multiplier. Specifically, the floating point adder contains 4 pipeline stages and the floating point multiplier contains 5 pipeline stages. Both designs are completely tested by test vectors following the IEEE standard.

The pipeline adder and multiplier allows us to design a pipeline butterfly operation and twiddle factor generator. The unit contains 24 pipeline stages with $\frac{1}{2}$ output rate. In other words, the output data of 2-point butterfly operations are available in every 4 clock cycles provided that the input data are fed continuously. The optimal algorithm takes full advantage of the pipeline.

Provided a twiddle fraction $\frac{r_j}{N}$, the twiddle factor can be generated in different ways ranging from using the complete table lookup to generating the next value from the previous values. The trade-off is the propagation errors. In the complete table lookup, there is no propagation errors while the generation from the previous value accumulates the errors. We reduce the errors by pre-storing some twiddle factors. Doing so, we can reset the error when it hits those pre-stored twiddle factor. The details of the implementation of the twiddle factor and the butterfly operation can be found in [39].

5.5.2 Address Generator (AG)

The address generator contains 3 parts: the data control unit (DCU), the address generation unit (AGU) and the twiddle fraction generation unit (TFG). Figure 5.7 shows the interface of the 3 units. The following subsections describe the 3 units.

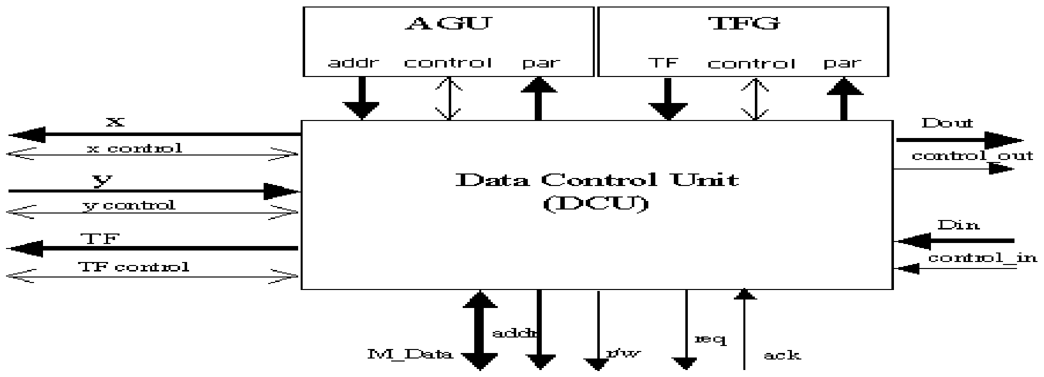


Figure 5.7 Address generator

5.5.3 Data Control Unit (DCU)

The data control unit provides the transfer of data to/from the computation unit, the memory and the interconnection network. Figure 5.8 shows the datapath and the controller inside the DCU.

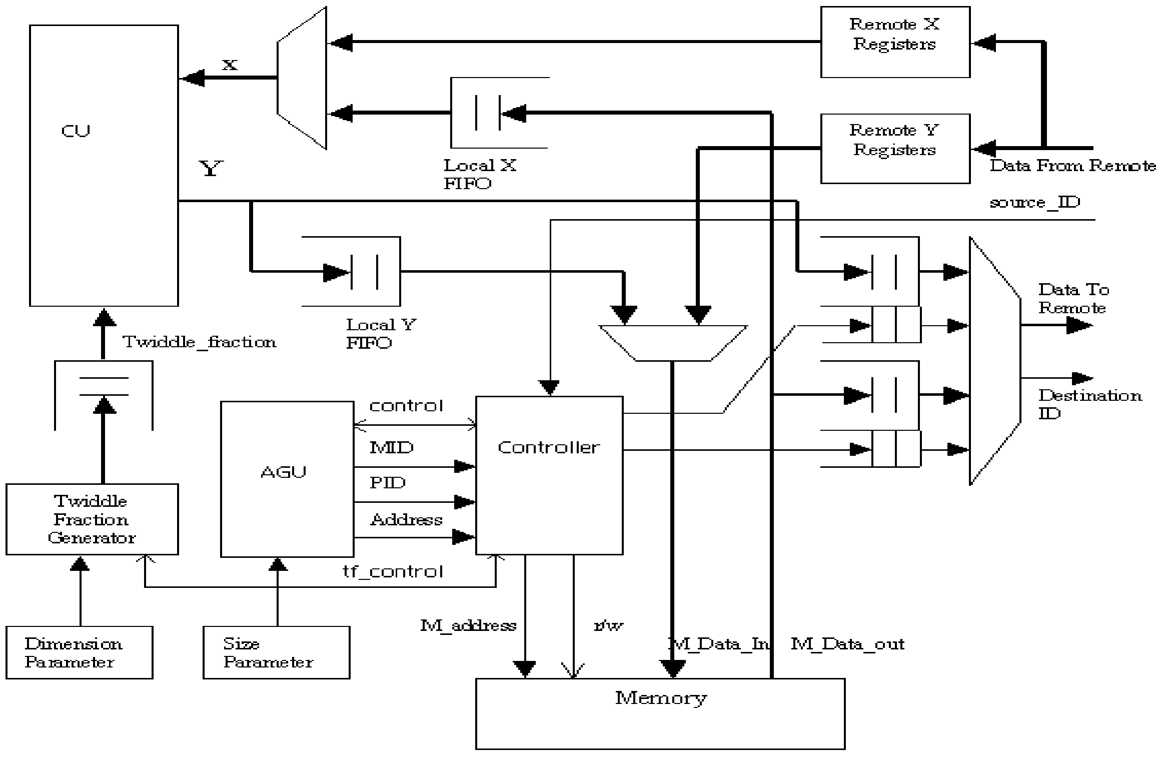


Figure 5.8 Data Control Unit (DCU)

The controller uses the information (address, target PID and source MID) from the AGU to control the transfer of data. The data to/from the CU, the memory and the interconnection network are buffered in their corresponding FIFOs.

Note that functionally the DCU does not depend on the algorithm. However, the optimal algorithm can simplify the controller and reduce the size of buffers.

5.5.4 Address Generation Unit (AGU)

Figure 5.9 shows the interface entity of an address generation unit. An address generator is parameterized during the runtime by the "size_parameter" and during the compile-time by the FFT dataflow and the maximum size denoted by n_{max}.

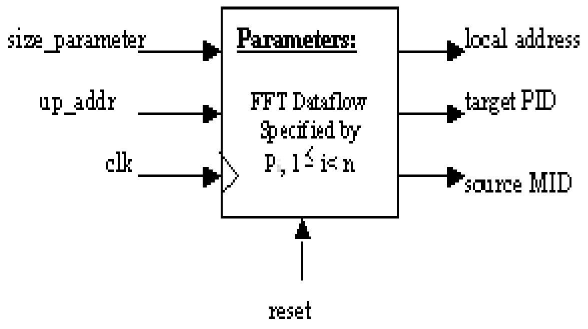


Figure 5.9 Interface entity of an address generation unit

As shown in Chapter 3, the AGU in each PE generates the sequence of local addresses, their target PIDs, and the source MIDs derived from the permutations specifying the FFT dataflow. Specifically, at stage i, all three parts are derived from the permutation function σ_i specifying the internal permutation P_i. In addition to the local address, the target ID and the source ID, an AGU also produces flags for the controller.

As shown in Chapter 3, the local addresses can be represented by permuted bit pattern. The permuted bits are derived from permutation function σ_i. Similarly, the target PIDs and source MIDs can be represented by permuted bit patterns. However, the target ID may includes the bits of the PID. Let the $(p_{m-1} \cdots p_0)_2$ is binary representation of the PID. Then, permuted bits representing target ID or source ID are derived from permuting the combination of the counter bits and the PID bits. For example, Table 3.12-3.14 shows the bit patterns for generating local addresses and target ID, and source ID using the Pease algorithm of size 2^n points FFT and 2^m processors respectively.

Based on tensor permutation, an address generator can be implemented using MUXs or adders as described in Section 5.1. We will use the Pease algorithm and the optimal algorithm of size 2^n points on 2^m processor elements as the examples for illustrating the implementation. We also assume that the maximum size is $2^{n_{max}}$ points.

5.5.4.1 Implementation using MUXs. Figure 5.10 shows the block diagram of address generator using an counter and MUXs.

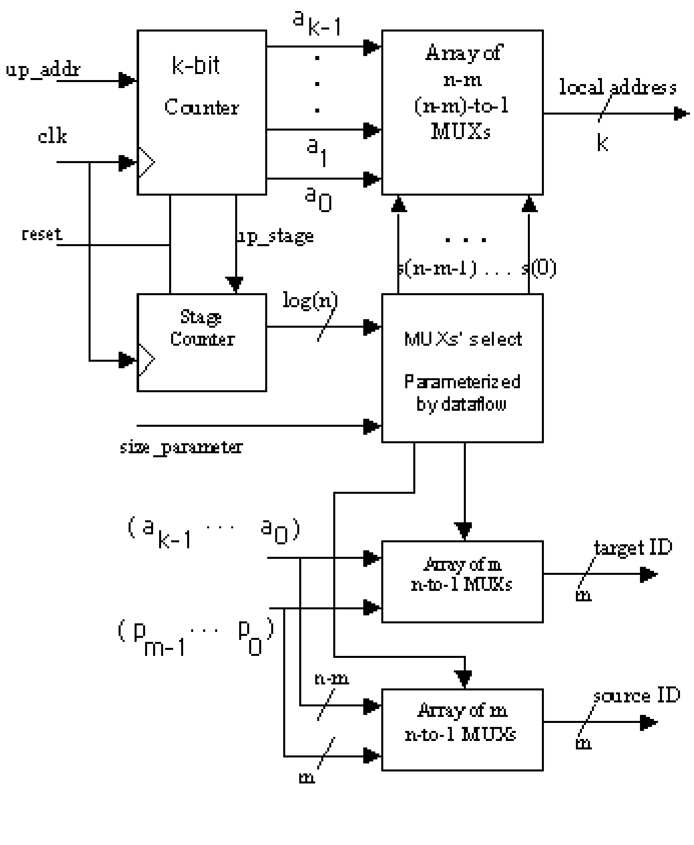


Figure 5.10 Address generator for FFT of size 2^n-points on 2^m-processor system using MUXs

Let $k = n_{max} - m$ and a_j, $0 \leq j < k$, be the j^{th} input of a k-to-1 MUX. Let $\mathbf{M}_j$ be the j^{th} MUX of an array of k MUXs. Let $s(j)$, $0 \leq s(j) < k$, be the MUX's select of M_j. Then, $a_{s(j)}$ is the output of M_j and $(a_{s(k-1)} \ldots a_{s(0)})_2$ is the output of the array of the MUXs. In other words, the MUXs permute $(a_{k-1} \ldots a_0)_2$ to $(a_{s(k-1)} \ldots a_{s(0)})_2$.

Similarly, target ID and source ID can be generated using array of m MUXs. However, in addition to the counter output, $a_{n-m-1}, \ldots, a_0$, the target PID and the source MID may include the PID bits labeled as $p_{m-1}, \ldots, p_0$. Therefore, m n-to-1 MUXs are required for generation of the target ID. Let a_j, $0 \leq j < n - m$, be the j^{th} input of an n-to-1 MUX and p_k, $0 \leq k < m$, be the $(n - m + k)^{th}$ input of the same n-to-1 MUX. Then, an array of m n-to-1 MUXs can be used to generate the target ID by setting the MUX's selects so that the target ID is realized. we also needs m n-to-1 MUXs for the generation of the source ID,

Table 5.1 MUXs selects for the generation of local address following the Pease algorithm

Stage	Local Address
	$s(n-m-1), \ldots, s(0)$
1	$n-m-1, \ldots, 0$
2	$n-m-2, \ldots, 0, n-m-1$
$\vdots$	$\vdots$
i	$n-m-i, \ldots 0, n-m-1, \ldots, n-m-i+1$
$\vdots$	$\vdots$
$n-m$	$0, n-m-1, \ldots, 1$
$n-m+1$	$n-m-1, \ldots, 0$
$\vdots$	$\vdots$
n	$n-m-1, \ldots, 0$

Example: The Pease algorithm

Tables 5.1-5.3 show the MUXs selects for the Pease algorithm of size 2^n and 2^m processors. These MUXs selects follow the permuted bits shown in Table3.12-3.14

Table 5.2 MUXs selects for the generation of target ID following the Pease algorithm

Stage	Target ID
	$st(m-1),\ldots,st(0)$
1	$m,\ldots,1$
2	$m,\ldots,1$
$\vdots$	$\vdots$
$n-2m+1$	$n-m,m-1,\ldots,1$
$\vdots$	$\vdots$
$n-2m+j$	$n-m+j-1,\ldots,n-m,m-j-1,\ldots,1$
$\vdots$	$\vdots$
$n-m$	$n-1,\ldots,n-m$
$\vdots$	$\vdots$
$n-m+j$	$j-1,\ldots,0,n-1,\ldots,n-m+j$
$\vdots$	$\vdots$
n	$m-1,\ldots,0$

Table 5.3 MUXs selects for the generation of source ID following the Pease algorithm

Stage	Source ID
	$ss(m-1),\ldots,ss(0)$
1	$n-m-1,\ldots,n-2m$
2	$n-m-2,\ldots,n-2m-1$
$\vdots$	$\vdots$
i	$n-m-i,\ldots 0,n-2m-i-1$
$\vdots$	$\vdots$
$n-2m+1$	$n-m-2,\ldots,n-2m-1,n-m$
$\vdots$	$\vdots$
$n-2m+j$	$n-m-j-1,\ldots,n-2m-1,n-m+j-1,\ldots,n-m$
$\vdots$	$\vdots$
$n-m$	$n-1,\ldots,n-m$
$\vdots$	$\vdots$
$n-m+j$	$n-j-1,\ldots,n-m,0,n-m-1,\ldots,n-m+j-1$
$\vdots$	$\vdots$
n	$0,n-m-1,\ldots,n-2m-1$

which are systematically derived from the permutation functions σ_i, $1 \leq i \leq n$, specifying the Pease algorithm. In the table, st_j and ss_j, $0 \leq j < m$, denote the MUXs selects for the j^{th} bit of the target ID and the source ID respectively. The concrete example of the MUXs selects are shown in the appendix.

MUX Selects

We needs n sets of $n + m$ MUX's selects for one size. If the system is to handle a fixed-size FFT, then the size parameter becomes a compile-time parameter. Once the size, n, is specified, the MUXs selects for the size are fixed for the particular size.

However, our goal is to implement universal FFT processors whose schedules are hardwired during the compile-time. In other words, we want the size to be varied during the run-time. Let the size of FFT varies from n_{min} to n_{max}. Then, we need $n_{max} - m$ $(n_{max} - m)$-to-1 MUXs and $2m$ n_{max}-to-1 MUXs for address generator. If n is a size in the range ($n_{min} \leq n \leq n_{max}$, we needs n sets of the MUXs selects for the n stages. Since there are $n_{max} - n_{min} + 1$ possible sizes in the range, there are possible $\sum_{n=n_{min}}^{n_{max}} n$ sets of the MUXs selects.

1. **Pre-stored MUX Selects**

   All possible MUXs selects can be pre-stored in a read-only memory (ROM). Each MUX's select is a $\lceil \log n_{max} \rceil$-bit binary number. Therefore, the ROM size is $\lceil \log n_{max} \rceil \times (n_{max} + m) \times \sum_{n=n_{min}}^{n_{max}} n$ bits.

   The other option of run-time size parameter is to use random-access memory (RAM) as the storage of MUX selects. The MUX selects for size n are loaded to the RAM as part of parameters. It increases the size of run-time parameters but reduces the memory size for storing MUX's selects to $\lceil \log n_{max} \rceil \times (n_{max} + m) \times n_{max}$.

2. **Generated MUX Selects**

   While pre-stored MUX's selects works for any FFT dataflow, it does not take advantage of nice properties provided by some algorithm. Since MUX's selects depends on permutation functions σ at each stage. If there exists a relationship between σ_i and σ_{i-1}, it can be used to generate the MUX's selects of stage i from MUX selects of previous stages.

   For Pease algorithm, the relationship of σ_i and σ_{i-1} is shifting. For the Pease algorithm, the initial value of the MUX select for local address denoted s_j, $0 \leq j < n-m$ is equal to j. Then, when changing stage and the current stage is less than $n-m$, s_{j-1}, $0 < j < n-m$ is shifted to become s_j. The MUX select s_0 depends on the size n which is the run-time parameter and m which is a compile-time parameter. It is initialized to 0. The next value of s_0 is $n-m-i$, where $i = 1, \ldots, n-m$. This can be generated by a decrement or a countdown counter initialized to $n-m-1$. If the current stage is greater or equal to $n-m$, the MUXs selects is reset to the initial values. Similarly, the MUX selects for target ID denoted st_j, $0 \leq j < m$, can be generated using m registers and a mod-m counter initialized to $n-m$, and the MUX selects for the source ID denoted ss_j, $0 \leq j < m$, can be generated using m registers and a countdown counter. Figure 5.11 shows a circuit used for generating MUX's selects of address generator of Pease algorithm of size 5 to 10 and 4 processors.

$\square$

Not only does the optimal dataflow produce the best performance, it tremendously simplify the implementation of the address generator when using MUXs. The following example illustrate the implementation of the optimal algorithm.

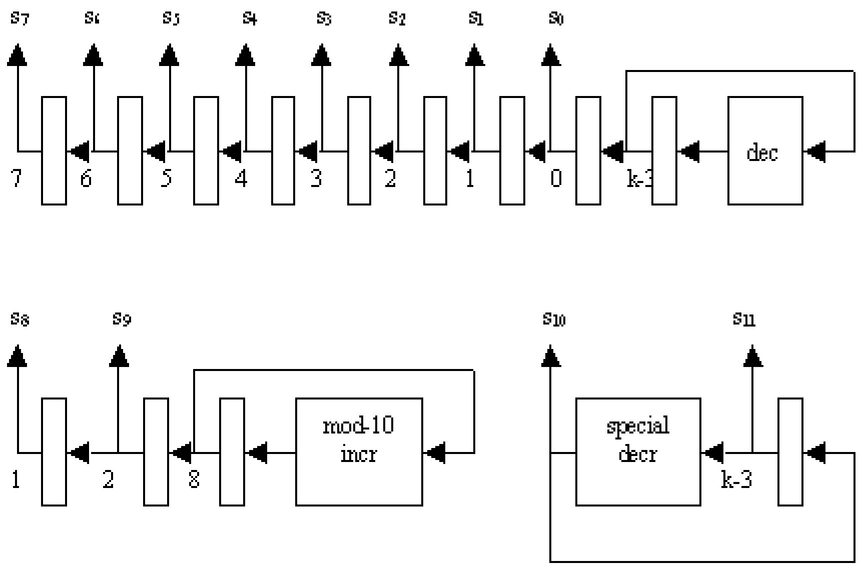


Figure 5.11 Generation of MUX's selects for address generator of Pease algorithm of size 2^m to 2^{10}

Example: The Optimal Algorithm

Table 4.5 in Chapter 4 shows the permuted bits for generating local address following the optimal algorithm. Notice that throughout all stages there are at most 3 bits that are permuted to an output bit. Specifically, let $(c_{n-m-1} \cdots c_0)$ be the local address generated by an address generator. Then, c_j, $0 \leq j < n - m$, is equal to either a_0, a_j, or a_{j+1}. This allows us to use 3-to-1 MUXs instead of (n-m)-to-1 MUXs.

Let s_j be the MUX select for selecting c_j. Then,

$$c_j \;=\; \begin{cases} a_0 & \texttt{if } s_j = 0 \\ a_j & \texttt{if } s_j = 1 \\ a_{j+1} & \texttt{if } s_j = 2 \end{cases} \;,\quad 0 \leq j < n \texttt{ and } a_{n-m} = a_{n-m-1}$$

Figure 5.12 shows the implementation of address generator for optimal algorithm using MUXs provided that the n_{max} is equal to 10. As shown in Figure 5.12, the MUX selects s_j can be generated using the shifting function. At the first stage, we initialize s_j, $1 \leq j < n - m$, to 1 and s_0 to 0. Then, when changing the stage and the

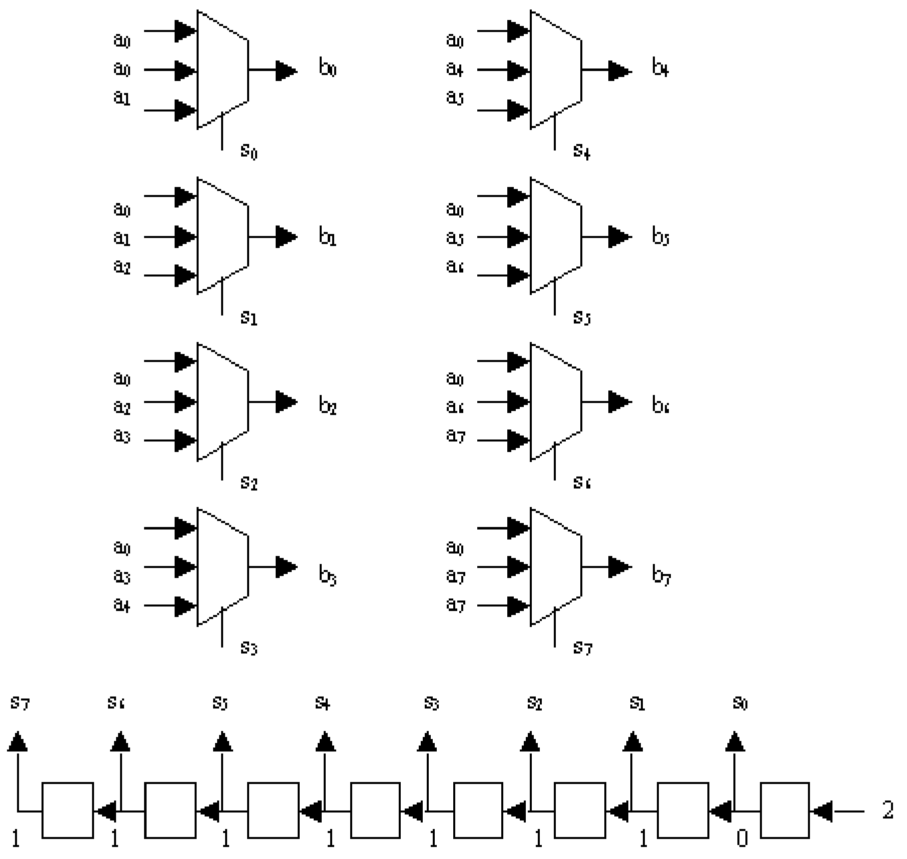


Figure 5.12 Address generator for optimal FFT algorithm of size 32 to 1024 points using 3-to-1 MUXs

current stage is less than $n-m$, s_0 is loaded with 2 and the current s_{j-1} is shifted to become the next s_j, $1 \leq j < n-m$. For stage greater or equal to $n-m$, the MUXs selects are that $s_{n-m-1} = 0$ and $s_j = 2$, $0 \leq j < n-m-1$.

Moreover, the target PID and the source MID have at most 1 bit different from the PE number. The communication between processors is simplified and there is no need to generate the whole target and source MID. Instead, we need to generate only a flag specifying whether the target PID and the source MID are local or remote. During stage 1 to $n-m$, both the source MID and the target PID are equal to the PE number, $(p_{m-1} \cdots p_0)$. Therefore, during these stages the flag is set to local. During stage $n-m+1$ to n, both the target PID and the source MID has only 1 bit different from the PE number. That bit is replaced by a_0. Therefore, during these last m stages, the flag is equal to a_0. $\square$

5.5.4.2 Implementation using Adders. In case such as the Pease algorithm, the generation of addresses using adders as described in Section 5.1.2 has advantages over the MUXs.

Example: The Pease algorithm

For the Pease algorithm of size 2^n points, the permuted bit pattern at stage i is described in Table3.12 which is shown again as follows.

$$\texttt{local address} = \begin{cases} (a_{n-m-1}\cdots a_0)_2, & i=1 \\ (a_{n-m-i}\cdots a_0 a_{n-m-1}\cdots a_{n-m-i+1})_2, & 2\le i\le n-m \\ (a_{n-m-1}\cdots a_0)_2, & n-m+1\le i\le n \end{cases}$$

This means that we needs only 2 increment numbers for generating using the adders. The first increment number (`INC1`) is equal to $2^i = (0\cdots 010\cdots 0)_2$. Notice that at stage 1 and stage i, $n-m+1\le i\le n$, the increment is always equal to 1. During stage 2 to $n-m$, the second increment number (`INC2`) is selected in stride $2^{n-m-i+1}$ or whenever $(a_{n-m-i}\cdots a_0) = (1\cdots 1)_2$; that is

$$\begin{aligned} \texttt{INC2} &= (0\cdots 0a_{n-m-1}\cdots a_{n-m-i+1})_2 + 1 - (0\cdots 01\cdots 1a_{n-m-1}\cdots a_{n-m-i+1})_2 \\ &= 1-(0\cdots 01\cdots 10\cdots 0)_2 \\ &= 1-(2^{n-m-i+1}-1)2^{i-1} = 1+2^{i-1}-2^{n-m} \\ &= (1\cdots 10\cdots 010\cdots 01)_2 \end{aligned}$$

Notice that only 2^{i-1} changes according to the changing of stage. Therefore, the `INC2` can be generated by a shift register and **or** gates. The number $1-2^{n-m}$ is initialized as a "size_parameter". The number 2^{i-1} is initialized to $(0\cdots 1)_2$ at stage 1. Then, it is shifted to the left by 1 bit when changing the stage. The `INC2` is bit-wise **or** of the two numbers. □

Since the implementation of the optimal algorithm's AGU using MUXs is simple, there is no clear advantage using adders. However, we show the implementation of its address generation unit as following.

Example: The Optimal algorithm

For the Optimal algorithm of size 2^n points, the permuted bit pattern at stage i is described in Table4.5 which is shown again as follows.

$$\texttt{local address} = \begin{cases} (a_{n-m-1} \cdots a_0)_2, & i = 1 \\ (a_{n-m-1} \cdots a_{i-1} a_0 a_{i-2} \cdots a_1)_2, & 2 \le i \le n-m-1 \\ (a_0 a_{n-m-1} \cdots a_1)_2, & n-m \le i \le n \end{cases}$$

There are 3 portions of consecutive bits; hence we needs 3 increment numbers. The first increment number (`INC1`) is equal to 2^{i-1}. It can be generated using a shift-left shift register initialized to $(0 \cdots 01)_2$ at stage 1. The second increment number (`INC2`) is for keeping the carry propagation from a_0 to a_1; i.e.

$$\begin{aligned} \texttt{INC2} &= (a_{n-m-1} \cdots a_{i-1} 0 a_{i-2} \cdots a_1)_2 + 1 - (a_{n-m-1} \cdots a_{i-1} 1 a_{i-2} \cdots a_1)_2 \\ &= 1 - (0 \cdots 010 \cdots 0)_2 = 1 - 2^{i-1} \\ &= (1 \cdots 10 \cdots 01)_2 \end{aligned}$$

The third increment number (`INC3`) is for keeping the carry propagation from a_{i-2} to a_{i-1}; i.e.

$$\begin{aligned} \texttt{INC3} &= (a_{n-m-1} \cdots a_{i-1} 0 \cdots 0)_2 + 1 - (a_{n-m-1} \cdots a_{i-1} 1 \cdots 1)_2 \\ &= 1 - (0 \cdots 01 \cdots 1)_2 = 1 - 2^i - 1 = -2^i \\ &= (1 \cdots 10 \cdots 0)_2 \end{aligned}$$

The INC3 can be generated by using the a shift-left shift register. Since INC2 is equal to the previous value of INC3 plus 1, we can generate INC2 by setting bit 0 to 1 and loading the remaining bits from the previous INC3. From stage $n-m$ to stage n, the increment numbers are the same as of stage $n-m-1$.

The INC3 has the highest priority. It is selected in stride 2^{i-1}. Then if INC3 is not selected and $a_0 = 1$, the INC2 is selected; otherwise, INC1 is selected. Since a_0 is flip-flopped between 0 and 1, we can generated it using a toggle flip-flop. Therefore, only one stride counter which counts in stride 2^{i-1} is required. □

5.5.5 Twiddle Fraction Generation (TFG)

Figure 5.13 shows interface entity of the twiddle fraction generator. Like the address generation unit, the compile-time parameter of twiddle fraction generator is the FFT dataflow. However, in addition to the size parameter, the dimension specification is also run-time parameter for the TFG.

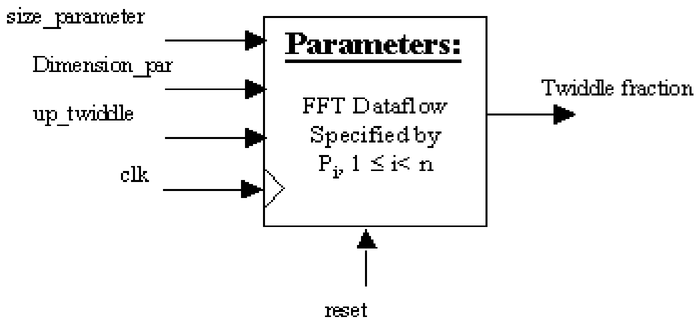


Figure 5.13 Twiddle fraction generators for FFT of size 5 to 10 using MUXs

As shown in Chapter 3, the twiddle fractions for any dimensions can be generated from twiddle fractions for one-dimensional FFT. Figure 5.14 shows the two steps used for implement twiddle fractions. First, the twiddle fractions of one-dimensional

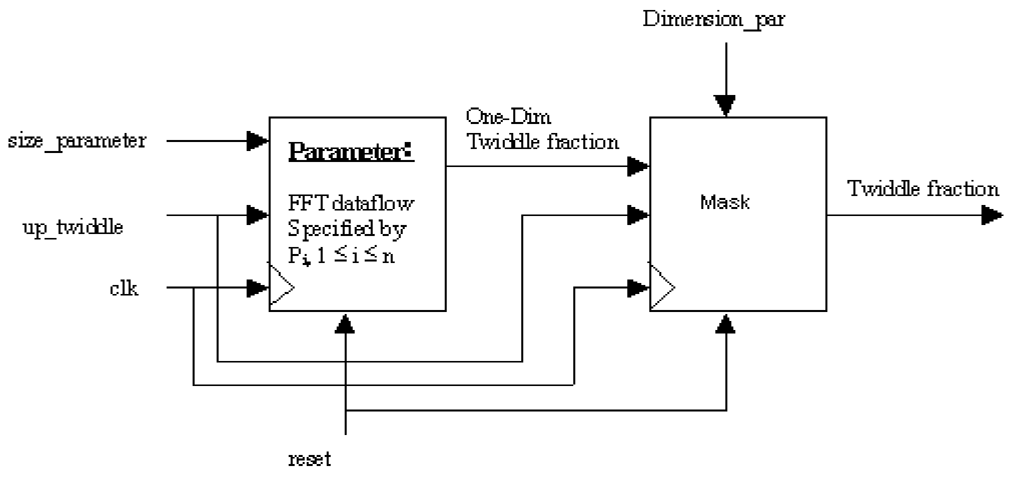


Figure 5.14 Two steps for generating twiddle fractions

FFT are generated. Let $(r_{n_{max}-1} \cdots r_0)_2$ be the n_{max}-bit binary representing one-dimensional 2^n twiddle fraction generated at stage i. Then, for general case of t-dimensional $(2^{n_1} \times \ldots \times 2^t)$-points FFT, where there are 2^{n_k} points, $1 \leq k < t$, in the k^{th} dimension, its twiddle fractions can be computed by masking off unwanted bits of $(r_{n_{max}-1} \cdots r_0)_2$. For example, Table 5.4 shows the masks for generation of twiddle fractions of algorithms for computing DFT up to 3 dimensions.

Figure 5.15 shows the implementation of mask for the Pease and the optimal algorithms. The key component of the mask generation is the n_{max}-bit shift right register whose MSB is loaded with '1'. The shifting occurs when changing stage and is reset when changing the stage and changing the dimension. The $\lceil \log n_{max} \rceil$-bit counter is used to count the number of stage in each dimension labeled as j. Its output is compared with n_k which is the number of stage in the k^{th} dimension, where k = 3, 2, and 1. Whenever the $j = n_k$, k is reduced by 1 and j is reset to 0. The 2-bit counter is used to count down k and k is used to select n_k. The counter is initialized to the number of dimensions labeled as "dim_no" and is counted down when $j = n_k$.

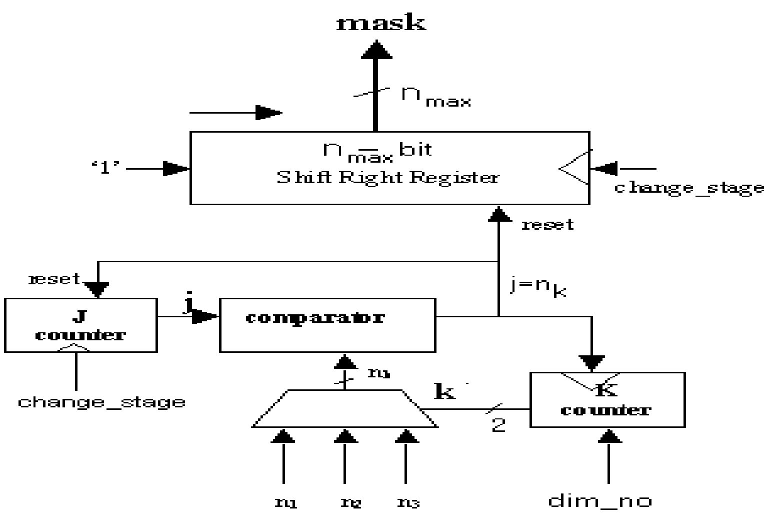


Figure 5.15 Generation of mask for computing twiddle fractions

The dimension parameters labeled as "Dimension_par" in Figure 5.13 and 5.14 are specified by n_1, n_2, n_3 and "dim_no."

5.5.5.1 Implementation using MUXs.

Similar to the generation of addresses, the one-dimensional twiddle fraction can be generated using MUXs as shown in Figure 5.16. In this figure, n_{max} is equal to 9. As shown in Chapter 3, the twiddle fractions for FFT of size 2^n can be represented by $(n-1)$-bit number. Therefore, if we represent the twiddle fractions by 9-bit binary number, the system can compute up to 2^{10} points.

The MUX selects $s_8, \ldots, s_0$ depend on the bit patterns of twiddle fractions derived from the one-dimensional twiddle factor matrices T_i, $1 \leq i \leq n$. For example, Table 5.5 shows the one-dimensional bit patterns of the one-dimensional Pease algorithm of size 2^n, where $m \leq n \leq 10$, 2^m is number of processors. The binary number $(a_{10-m-1} \cdots a_0)_2$ is the output of 10-bit counter and $(p_{m-1} \cdots p_0)_2$ is the PID.

Table 5.4 Mask for generating twiddle fractions of the Pease and optimal algorithms

k	j	$2^{n_{max}}$-bit Mask	Stage
3	0	$(0\cdots0)_2 = 0$	1
	$\vdots$	$\vdots$	$\vdots$
	j	$(1\cdots10\cdots0)_2 = (2^j-1)2^{n_{max}-j}$	j
	$\vdots$	$\vdots$	$\vdots$
	n_3-1	$(1\cdots10\cdots0)_2 = (2^{n_3-1}-1)2^{n_{max}-n_3-j}$	n_3
2	0	$(0\cdots0)_2 = 0$	n_3+1
	$\vdots$	$\vdots$	$\vdots$
	j	$(1\cdots10\cdots0)_2 = (2^j-1)2^{n_{max}-j}$	n_3+j+1
	$\vdots$	$\vdots$	$\vdots$
	n_2-1	$(1\cdots10\cdots0)_2 = (2^{n_2-1}-1)2^{n_{max}-n_2-1}$	n_3+n_2
1	0	$(0\cdots0)_2 = 0$	n_3+n_2+1
	$\vdots$	$\vdots$	$\vdots$
	j	$(1\cdots10\cdots0)_2 = (2^j-1)2^{n_{max}-j}$	n_3+n_2+j+1
	$\vdots$	$\vdots$	$\vdots$
	n_1-1	$(1\cdots10\cdots0)_2 = (2^{n_2-1}-1)2^{n_{max}-n_1-1}$	$n_3+n_2+n_1=n$

The MUX's selects $s_8, s_7, \ldots, s_0$ are either stored in memory or generated. Since $\lceil \log 9 \rceil$ is equal to 4, each MUX select is a 4 bit number. Hence, we will need a 4×9 bits of ROM for one set of the MUX selects. In general, the total number of bits needed for storing all possible sizes is $\sum_{n=2}^{10} 4 \times \times 9 \times n$ bits because there are 9 possible sizes and each size has n stages. However, for Pease algorithm the MUX selects of the last stage can be used as of all stages when used with the mask. Therefore, we need only $4 \times \times 9 \times 9$ bits. Table 5.6 shows the MUX's selects for Pease algorithms of size 2^n, $2 \leq n \leq 9$.

5.5.5.2 Implementation using Adders. The adder shown in Figure 5.3 can be used for generating the one-dimensional twiddle fractions. The increment (INC) depends on the bit patterns of twiddle fractions derived from the one-dimensional twid-

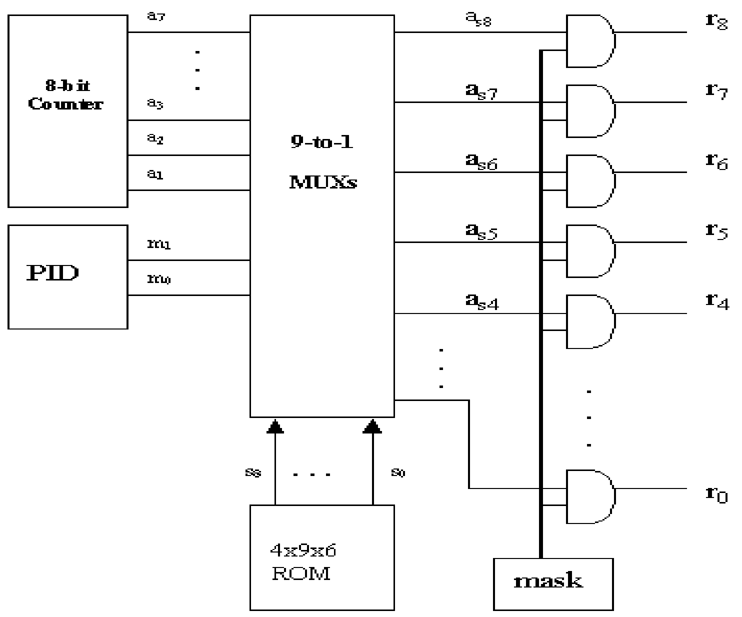


Figure 5.16 Twiddle fraction generators for FFT of size 2^m to 2^10 using MUXs

dle factor matrices T_i, $1 \leq i \leq n$. As described in Section 5.5.25.5.4.2 of this chapter, number of increment numbers in each stage depends on the number of portions with consecutive bits.

Example: Pease Algorithm

At the last stage of the Pease algorithm, the whole pattern is a portion with consecutive bits, which is $(a_{n-m-1}\cdots a_1 p_{m-1}\cdots p_0 0\cdots 0)_2$. This pattern contains two parts: the counter bits $(a_{n-m-1}\cdots a_1)_2$ and the PID's bits $(p_{m-1}\cdots p_0)_2$. Since the PID portion is a constant, only the portion $(a_{n-m-1}\cdots a_1)$ is incrementing. This can be accomplished by setting the increment number to $(0\cdots 010\cdots 0) = 2^{n_{max}-n+m+1}$ which can be implemented by shifting 1 to the left by $n_{max}-n+m+1$ bits. The PID portion can be included in the initial value (INIT). Specifically, the initial value can be set to $(0\cdots 0p_1p_0 0\cdots 0)_2 = (p_1p_0)_2 2^{n_{max}-n+1}$ which can be implemented by shifting $(p_1p_0)_2$ to the left by $n_{max}-n+1$ bits. For example, if $n_{max}=9$, $m=2$ and $n=5$,

Table 5.5 Twiddle fractions for Pease algorithms of size 2^n where $m \leq n \leq 10$

Stage	9-bit twiddle fractions
0	$(0 \cdots 0)_2$
1	$(a_{n-m-1}0 \cdots 0)_2$
$\vdots$	$\vdots$
i	$(a_{n-m-1} \cdots a_{n-m-i}0 \cdots 0)_2$
$\vdots$	$\vdots$
n-m-1	$(a_{n-m-1} \cdots a_1 0 \cdots 0)_2$
n-m	$(a_{n-m-1} \cdots a_1 p_{m-1} 0 \cdots 0)_2$
$\vdots$	$\vdots$
n-m+j	$(a_{n-m-1} \cdots a_1 p_{m-1} \cdots p_{m-j-1} 0 \cdots 0)_2$
$\vdots$	$\vdots$
n-1	$(a_{n-m-1} \cdots a_1 p_{m-1} \cdots p_0 0 \cdots 0)_2$

the increment number is $(010000000)_2 = 2^7$ and the initial value is $(00p_1p_000000)_2 = (p_1p_0)_2 2^{9-5+1}$. If $n = 8$, the increment number becomes $(000010000)_2 = 2^5$ and the initial value becomes $(00000p_1p_000)_2 = (p_1p_0)_2 2^{9-8+1}$.

The same INC and INIT numbers can be used for all the stages because the unwanted bits are masked off by the mask shown previously. Therefore, both INC and INIT numbers are changed only when the size (n) is changed. □

Example: The Optimal Algorithm

For the optimal algorithm, the bit patterns for generating twiddle fractions of the 1-D 2^n-point FFT following the optimal algorithm is shown in Table 4.6. This can be generated using a adder with increment number equal to $2^{n_{max}-i+1}$. However, during stage $n - m + 1$ to stage n the initial value ($INIT$) is set to $(p_{j-1} \cdots p_0 p_j)2^{n_{max}-j}$, where $j = i - n + m$. □

Table 5.6 MUXs' selects for generating twiddle fractions of Pease algorithm for FFT of size 2^n, $2 \le n \le 10$

Size	ROM	MUXs′selects								
	Address	s_8	s_7	s_6	s_5	s_4	s_3	s_2	s_1	s_0
2	0	8	0	0	0	0	0	0	0	0
3	0	9	8	0	0	0	0	0	0	0
4	0	1	9	8	0	0	0	0	0	0
5	0	2	1	9	8	0	0	0	0	0
6	1	3	2	1	9	8	0	0	0	0
7	2	4	3	2	1	9	8	0	0	0
8	3	5	4	3	2	1	9	8	0	0
9	4	6	5	4	3	2	1	9	8	0
10	5	7	6	5	4	3	2	1	9	8

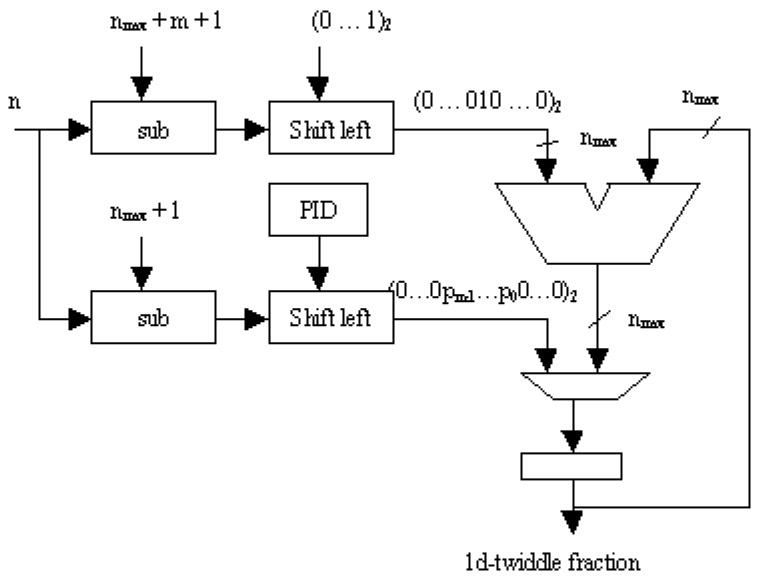


Figure 5.17 Generation of one-dimensional twiddle fractions of the Pease algorithm using adders

5.6 Implementation of the Optimal Algorithm on the Wildforce^{TM} Board

In this section, we describe the implementation of the optimal algorithm on the Wildforce^{TM} FPGA board shown in Figure 5.18. The board consists of 5 Xilinx FPGA chips [40] (XC4085XLA) connected via a configurable crossbar interconnect. Each processor has its own memory. The board communicates with a host computer using a PCI bus. The host can read/write data from/to either the FIFOs connected to CPE0, PE1 and PE4 respectively or from/to the memories directly.

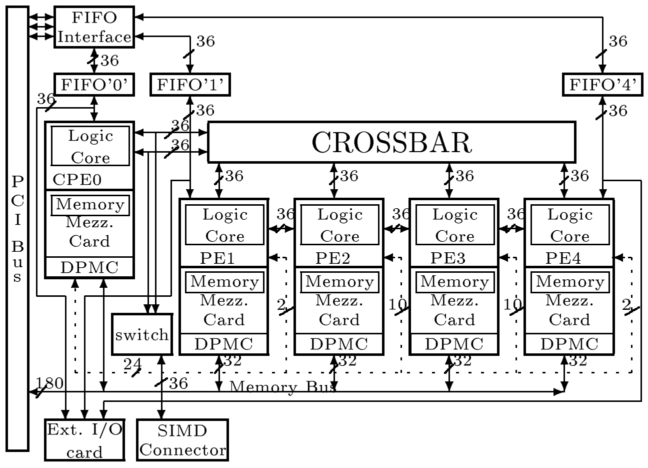


Figure 5.18 Wildforce^{TM} Architecture

The architecture in Section 3, is mapped to the board as follows. The processor CPE0, along with the FIFO '0' is used as interface unit. The remaining 4 PEs are used to implement the 4 processors , each with a computation unit and address generator. The crossbar provided by the board is used as the interconnection network. Figure 5.19 shows the datapath of a PE. Notice that we need only 2 FIFOs for receiving data from the interconnection network and 2 FIFOs for sending. This is because the network configurations is fixed throughout a stage; therefore there is no

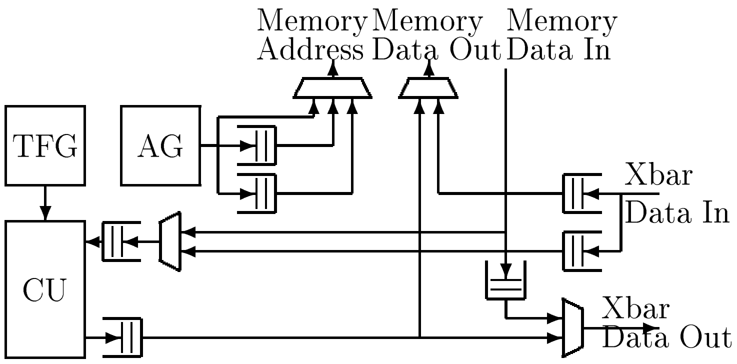


Figure 5.19 Processor Element's datapath

need to specify the target PID or source MID. Only a flag identifying whether a data is local or remote is enough.

The design of all units are done in synthesizable VHDL. We use a commercial synthesis tool to synthesize the net list. Then, the place and route is done using the tool provided by Xilinx.

Since the design of a PE is barely fit in one chip, we can only run up to 10 MHz. In the next chapter, we discuss the verification and the performance of the implementation.

6.0 VERIFICATION AND PERFORMANCE

In this chapter, we discuss the testing and the benchmarking we performed to verify the correctness and to analyze the potential performance of our design and implementation. Section 6.1 presents the techniques we used to verify the design and prototype implementation. These techniques rely on the fact that the FFT is a linear computation and hence can be completely tested by using a set of inputs equal to a basis. Section 6.2 reports on the performance of our prototype implementation.

6.1 Verification

Since the DFT is linear we can completely test correctness by verifying linearity and testing on a basis. Let A be a matrix with N columns and let $\mathbf{x}$ be a vector of size N. Then, since

$$\begin{aligned}\mathbf{x} &= \sum_{i=0}^{N-1} x_i e_i^N \\ A(\mathbf{x}) &= A(\sum_{i=0}^{N-1} x_i e_i^N) = \sum_{i=0}^{N-1} x_i A(e_i^N)\end{aligned}$$

we can test any program or hardware designed to compute the linear computation $y = A\mathbf{x}$ by testing A on the standard basis. Assuming that the computation is indeed linear (this can be verified for random inputs), this provides an exhaustive test. Since $A\mathbf{e}_i^N$ is equal to the i^{th} column of A, we compare the output of calling A with input $\mathbf{e}_i^N$ to the i^{th} column of A.

Since the correctness of our FFT engine depends on the correctness of the floating point units (we need to assume that arithmetic is correct) these were tested using a standard test suite. Then, we applied the above basis test and compared the result with the i^{th} column of the DFT matrix which can be computed easily by definition.

This testing technique is quite different from what is usually done where typically a test is performed on an incomplete set of inputs such as a sinusoidal function. An additional benefit of our test, is that it allows us to easily trace through the computation stages (we know the output for each stage) to help track down errors.

6.2 Performance

The performance of the universal FFT Engine implemented on Wildforce^{TM} FPGA board is measured by counting the number of clocks. This allows us to easily scale performance data with a more realistic clock rate and it allows us to compare the actual performance to the ideal performance bounded by memory requirements.

When running an FFT algorithm of size 2^n on the proposed architecture, there are $4 \cdot 2^n$ memory accesses in each stage ($2 \cdot 2^n$ for reading and $2 \cdot 2^n$ writing). If we can feed all input data to the computation unit continuously, we can accomplish the best case performance in which one data point, which is equal to 4 floating point numbers, is ready every 8 clock cycles.

Assume that we use one PE to compute a 2^n-point FFT. For each butterfly operation, we need 4 clock cycles for reading 2 points of data; i.e. for each point, we need 1 clock cycle for real part and 1 clock cycle for imaginary part. Writing back the result will need another 4 clock cycles for each butterfly operation. Assume that we can not read and write at the same time. We will need at least 8 clock cycles for computing each butterfly operation, in which case a result from each butterfly operation is ready in every 8 clock cycles. There are $\frac{2^n}{2}$ butterfly operations in each stage. Therefore, we need $8(\frac{2^n}{2}) = 4(2^n)$ clock cycles to compute butterfly operations in each stage. Since there are n stages, the best case performance for the single processor system is equal to $4n2^n$ clock cycles. When the computations are distributed to 4 processors, the best case performance is equal to $\frac{4n2^n}{4}$. Table 6.1 shows number of clock cycles

Table 6.1 Performance measured in number of clock cycles

Size	Performance measured in number of clocks			
	Total	FFT Engine	Best Case	
(n)	$(= 4n(2^n))$	(4 PEs)	$(= \frac{\text{Total}}{4})$	$\frac{\text{FFT Engine}}{\text{Best Case}}$
5	640	445	160	2.78
6	1536	717	384	1.87
7	3584	1400	896	1.56
8	8192	2848	2048	1.39
9	18432	5960	4608	1.29
10	40960	12656	10240	1.24
11	90112	27032	22528	1.20
12	196608	57792	49152	1.18
13	425984	123368	106496	1.16
14	917504	262672	229376	1.15
15	1966080	557624	491520	1.13

needed by the universal FFT engine to perform an FFT of size 2^k, where $5 \leq k \leq 15$. It also compares the result with the best case performance. Note that for an FFT of size 2^{15}, the performance time is 13 % more than the best case.

Because of limitations in space on the Xilinx chips we used, our implementation could only achieve a clock rate of 10 MHz. However, because of the scalability of the FFT engine, we can linearly project the performance at 100 MHz.

In addition to showing the scalability of our design (i.e. how close to the ideal case we are when we increase the number of processors), we compare our performance to running a highly tuned software implementation on a standard processor. We compared our design with FFTW running on a Pentium II 450 MHz. Table 6.2

Table 6.2 Performance measured in micro seconds

Size	FFT Engine		FFTW
n	10 MHz	100 MHz	450 MHz
5	44.5	4.45	2.53
6	71.7	7.17	6.24
7	140.0	14.00	16.64
8	284.8	28.48	37.86
9	1843.2	184.32	86.26
10	1265.6	126.56	206.32
11	2703.2	270.32	513.44
12	5779.2	577.92	1220.72
13	12336.8	1233.68	2671.15
14	26267.2	2626.72	6044.92
15	55762.4	5576.24	15460.45

compares the real performance of FFT engine at 10 MHz and the scale performance at 100 MHz with the performance of FFTW. Our data shows upto a factor of 3 in performance gain.

Since most FFT processors report performance for size equal to 1024 points, we compare our design with other FFT processors collected by Bevan M. Bass [41]. We found that our FFT processor running at 100 MHz (126.56 μsec) is comparable to commercial FFT processors such as DaSP/PaC/RaS (131 μsec) by Array Microsystem [42] and synthesizable FFT processor core (90 μsec) Inventa by Mentor Graphic [43].

Note that our design is also scalable in term of number of processors. For example, for the 8-processor system, the performance time is almost twice that of the 4-processor system.

We would like to remark that while these figures suggest that the performance we obtained is good, it was not our purpose to obtain a high-performance implementation. We were mainly interested in demonstrating our methodology.

7.0 CONCLUSIONS

In this thesis we designed, optimized, and implemented a universal FFT processor. The universal FFT processor is based on a class of algorithms called dimensionless FFTs developed by Auslander, Johnson, and Johnson [16]. The use of the dimensionless FFT allows our processor to use the same design for computing 1, 2, and 3 dimensional FFTs. This increases the flexibility of an FFT processor, while retaining the hardwired control, that leads to improved performance over general-purpose processors.

The main theme of this thesis is not the end product but the methodology we used to optimize the design. Using mathematical properties of the FFT we were able to perform a systematic search for the optimal design. There is a large family of FFT algorithms which can be classified by their dataflow properties (i.e. memory access patterns). Starting with an abstract architectural model for a distributed memory processor and a mapping from the space of FFT algorithms to the architecture we were able to automatically optimize the processor design. We introduced a high-level performance model that captures key properties of the memory system, and using this model as a cost function we searched for the algorithm that led to a processor with the best performance. The search automatically found an interesting algorithm with good locality and communication patterns that dramatically reduces memory contention as compared to standard algorithms.

An implementation of the processor which is parameterized, at compile time, by the choice of algorithm was presented. The parameterization allows us to instantiate many different designs. We ultimately chose a design based on the optimal algorithm found by the search. Once this particular design was chosen, we were able to further simplify the implementation by taking advantage of properties of the particular algorithm that was chosen.

Using mathematical properties of the FFT and the fact that the communication of a given algorithm is known a priori, we designed a distributed memory processor with completely local control. The use of a design with local control and simple communication patterns leads to a scalable design. The resulting design was prototyped using a board with 5 Xilinx FPGA processors. This implementation was systematically verified for correctness and a preliminary performance study was performed to investigate scalability and its performance potential. The performance of the implementation prototype is closed to the best performance of the architecture under consideration. Because of the limitation of the space and the technology of the FPGA chip (XC4085XLA), the actual implementation can run up to 10 MHz. However, the prototype can be scaled up both in terms of clock speed and number of processors. At 100 MHz, the performance of the implementation prototype is comparable to available FFT processors and outperforms a general purpose processor running a highly tuned FFT package (we compared using the FFTW package).

Based on our preliminary results we would like to build a high-performance version of our design using a large number of processors. Additional optimizations will need to be incorporated, including the use of an hierarchical system where there is a board containing many smaller FFT processors, each containing many processing elements like the ones in this thesis, in order to fully utilize a large number of processors. In addition to the extensions necessary to build a large scale FFT processor, another possible direction for future work is the extension of our design to other signal processing computations, in particular other classes of fast transforms such as fast trigonometric transforms and wavelets. More generally one could investigate the applicability of our design methodology to other problem areas.

BIBLIOGRAPHY

[1] John G. Ackenhusen, *Real-Time Signal Processing: Design and Implementation of Signal Processing Systems*, Prentice Hall, 1999.

[2] J. Nurmi, J; Takala, "A new generation of parameterized and extensible DSP cores," in *Signal Processing Systems 1997, SIPS 97-Design and Implementation, 1997 IEEE Workshop on*, 1997.

[3] M. Frigo and S.G. Johnson, "FFTW: An Adaptive Software Architecture for the FFT," in *ICASSP Conference Proceedings*, 1998, vol. 3, p. 1381.

[4] Arun Chhabra and Ramesh Iyer, "A Block Floating Point Implementation on the TMS300C54xDSP," Tech. Rep. SPRA610, Texas Instruments, Inc., December 1999.

[5] Mike Hannah and Aaron Kofi Aboagye, "Implementation of Double-Precision Complex FFT for the TMS300C54xDSP," Tech. Rep. SPRA554B, Texas Instrument, Inc., August 1999.

[6] Jeff Alexlrod, "TMS300C200 C-Callable FFT Package," Tech. Rep. SPRA354, Texas Instrument, Inc., May 1998.

[7] Charles Wu, "Implementing the Radix-4 Decimation in Frequency (DIF) Fast Fourier Transform (FFT) Algorithm Using a TMS320C80DSP," Tech. Rep. SPRA152, Texas Instrument, Inc., January 1998.

[8] Rebert Matusiak, "Implementing Fast Fourier Transform Algorithms of Real-Valued Sequence with the TMS320 DSP Family," Tech. Rep. SPRA291, Texas Instrument, Inc., December 1997.

[9] Rose Marie Piedra, “Parallel 2-D FFT Implementation with TMS320C4X DSP,” Tech. Rep. SPRA027A, Texas Instrument, Inc., Febuary 1994.

[10] Rose Marie Piedra, “Parallel 1-D FFT Implementation with TMS320C4X DSP,” Tech. Rep. SPRA108, Texas Instrument, Inc., Febuary 1994.

[11] “Parallel Processing with the TMS320C4x,” Tech. Rep. SPRA031, Texas Instrument, Inc., Febuary 1994.

[12] J. Isoaho L. Jia, Y. Gao and H. Tenhunen, “A new vlsi-oriented FFT algorithm and implementation,” in *Proc. IEEE Int’l ASIC Conf.*, 1998, pp. 337–341.

[13] IComm. Technologies Inc., *FFT-1024 Complex 1024-points FFT/iFFT Processor*, November 1999, Product Design Specification.

[14] “1-K 32-bit floating point complex FFT,” `http://www.annapmicro.com`.

[15] J. M. F. Moura, J. Johnson, R. Johnson, D. Padua, V. Prasanna, and M. M. Veloso, “SPIRAL: Portable Library of Optimized SP Algorithms,” 1998, `http://www.ece.cmu.edu/~spiral/`.

[16] L. Auslander, J. R. Johnson and R. W. Johnson, “Dimensionless fast Fourier transforms,” Tech. Rep. DU-MCS-97-01, Drexel University, 1997, `http://www.mcs.drexel.edu`.

[17] C. Hein J. Ammon, “Vhdl-based performance modeling: An application of the pmw tool suite to an image classification system,” in *VIUF-VHDL International Users Forum*, October 1997.

[18] W. Schaming, “Hardware/software co-design in the rapid prototyping of application-specific signal processors methodology,” in *VIUF-VHDL International Users Forum*, October 1997.

[19] "Unified modeling (UM) reference manual (ADEPT Version A.1)," Tech. Rep. 960620.0, CSIS, University of Virginia, 1996.

[20] "RASSP Application Notes," `http://www.atl.external.lmco.com`.

[21] Jack J. Dongarra R. Clint Whaley, Antoine Petitet, "Automated Empirical Optimization of Software and the ATLAS Project," 2000, `http://www.netlib.org/atlas/`.

[22] J. R. Johnson and R. W. Johnson, "Distributed memory FFT algorithms and dataflow," in *Proc. of 1999 High Performance Embed Computing Conference (HPEC99)*, MIT Lincoln Lab, Cambridge, MA, September 1999.

[23] Annapolis Micro Systems, Inc., *WILDFORCE Reference Manual Revision 3.4*, 1999, `http://www.annapmicro.com`.

[24] "BOPS, Inc.," `http://www.bops.com/`.

[25] "DSP Arhitecture, Inc.," `http://www.dsparchitectures.com/`.

[26] C. Van Loan, *Computational Framework for the Fast Fourier Transform*, SIAM, Philadelphia, PA, 1992.

[27] M. An R. Tolimieri and C. Lu., *Algorithms for Discrete Fourier Transform and Convolution*, Springer-Verlag, New York, 1989.

[28] J.R. Johnson, R.W. Johnson, D. Rodriguez, and R. Tolimieri, "A methodology for designing, modifying, and implementing fourier transform algorithms on various architecture," *Circuit, Systems, and Signal Processing*, vol. 9, no. 4, pp. 249–500, 1990.

[29] R. A. Horn and C. R. Johnson, *Topics in Matrix Analysis*, Cambridge University Press, Cambridge, MA, 1991.

[30] Thomas W. Hungerford, *Algebra*, Springer-Verlag, New York, 1989.

[31] N. P. Pitsianis, *The Kronecker Product in Optimization and Fast Transform Generation*, Ph.D. thesis, Cornell University, 1997.

[32] James W. Cooley and J.W. Tukey, "An algorithm for the machine calculation of complex series," *Math. Comput.*, 1965.

[33] M. C. Pease, "An adaptation of the fast fourier transform for parallel processing," *ACM*, vol. 15, no. 2, pp. 252–264, April 1968.

[34] R. J. Auletta, *An Uninterpreted Model for Hardware Description Languages*, Ph.D. thesis, University of Virginia, 1987.

[35] B.W. Johnson J.H. Aylor, R. Waxman and R.D. Williams, *The Integration of Performance and Functional Modeling in VHDL*, chapter 2, pp. 22–145, Prentice Hall, Englewood Cliffs, NJ, 1992.

[36] K. Jensen, *Coloured Petri Nets: A High Level Language for System Design and Analysis*, High-level Petri Nets: Theory and Applications. Springer-Verlag, Berlin, Germany, 1991.

[37] John L. Hennessy and David A. Patterson, *Computer Architecture: A Quantitative Approach*, Morgan Kaufmann Publishers, Inc., 1990.

[38] ANSI/IEEE Std. 754-1985, *IEEE Standard for Binary Floating-point Arithmetic.*

[39] Ryan Buchert, "A design of twiddle factor generator for universal fft processor," M.S. thesis, Drexel University, 2001.

[40] Xilinx, Inc., *The Programmable Logic Data Book*, `http://www.xilinx.com`.

[41] Bevan M. Bass, “FFT Info. Page,” `http://nova.standford.edu/~ bbass/`.

[42] “Array Microsystem, Inc.,” `http://www.array.com/`.

[43] “Mentor Graphics, Inc.,” `http://www.mentorg.com/inventra/`.

[44] J. R. Johnson, J. Dwyer and R. W. Johnson, “FFT machine description and documentation,” preprint, October 1997.

[45] J. R. Johnson and R. W. Johnson, “A universal FFT machine,” preprint, October 1997.

[46] P. Kumhom, J. R. Johnson and P. Nagvajara, “Design, optimization, and implementation of a universal FFT engine,” Tech. Rep. DU-MCS-00-01, Drexel University, 2000, `http://www.mcs.drexel.edu`.

[47] L. Auslander, J. R. Johnson and R. W. Johnson, “Multidimensional cooley-tukey algorithms revisited,” *Adv. Appl. Math.*, vol. 19, no. 9, pp. 297–301, April 1996, `http://www.mcs.drexel.edu`.

[48] S.Y. Kun, *VLSI Array Processor*, Prentice Hall, Englewood Cliffs, NJ, 1987.

[49] J.L. Peterson, *Petri Net Theory and the Modeling of Systems*, Prentice Hall, Englewood Cliffs, NJ, 1981.

[50] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, “Xputer-an open family of non von neuman architecture,” Tech. Rep. 195/89, University of Kaiserslautern, 1989.

[51] Michael Herz Reiner W. Hartenstein, Juergen Becker and Ulrich Nageldinger, "A novel universal sequencer hardware," in *Proceeding of Fachtagung Architekturen von Rechensystemen ARCS'97*, Rostock, Germany, September 8-11 1997.

[52] R.W. Hartenstein, A.G. Hirschbiel, and M. Weber, "A pseudo parallel architecture for systolic algorithms," in *Proc. of the Int'l Conference on VLSI and CAD*, 1989.

[53] Texas Memory Systems, Inc., *TM-66 swiFFT Chip User Guide.*

[54] M.A. Wendl Pan; Shams, A.; Bayoumi, "Neda: a new distributed arithmetic architecture and its application to one dimensional discrete cosine transform," in *Signal Processing Systems, 1999. SiPS 99. 1999 IEEE Workshop on*, 1999, pp. 159 –168.

[55] Sandeep K.S. Gupta Zhiyong Li, John H. Reif, "Synthesizing efficient out-of-core programs for block recursive algorithms using block-cyclic data distributions," *IEEE Transactions on Parallel and Distributed Systems*, vol. 10, no. 3, pp. 297–315, March 1999.

APPENDIX A

APPENDIX A EXAMPLES ON 64-POINT FFT

We use the 64-point DFT to illustrate every step of the methodology. The Pease algorithm and the optimal algorithm are used to demonstrate the effects of different algorithms. First, we step through all design steps (except the optimization step) using the Pease algorithm of size 64 points and 4 processors as the example. Then, the same approach is applied to the optimal algorithm of size 64 points and 4 processors. For convenience, we summarize the design steps as follows.

Step 1: Mathematical formula describing an algorithm

Step 2: Mapping formula to FFT processors

Step 3: Optimization

Step 4: Implementation of the chosen algorithm

A.1 Example 1: The Pease Algorithm

Even though the proposed design methodology is independent of the chosen algorithm, the Pease algorithm is a good choice for demonstrating the methodology. This is because (1) the FFT algorithms under consideration are based on it and (2) it a well-known FFT algorithm that can serves as a benchmark for other algorithms.

A.1.1 Formula Representing the Pease Algorithm

As shown in Section 2.4, FFT algorithms under consideration are described by Equation 2-52. For the 64-point FFT, the formula becomes

$$\begin{aligned} F_{64} &= P_6^{-1}(I_{32} \otimes F_2)T_6P_6 \cdot P_5^{-1}(I_{32} \otimes F_2)T_5P_5 \cdot \\ &\quad P_4^{-1}(I_{32} \otimes F_2)T_4P_4 \cdot P_3^{-1}(I_{32} \otimes F_2)T_3P_3 \cdot \\ &\quad P_2^{-1}(I_{32} \otimes F_2)T_2P_2 \cdot P_1^{-1}(I_{32} \otimes F_2)T_1P_1 \cdot P_0 \end{aligned} \tag{A-1}$$

Such FFT algorithm can be parameterized by 3 sets of matrices: the initial permutation P_0, the internal permutations P_i and the twiddle fraction matrices T_i, where $1 \leq i \leq n$ and 2^n is the number of points.

To demonstrate that a new FFT algorithm can be generated by manipulating a known algorithm, we show how the conjugating Pease algorithm is derived from the iterative Cooley-Tukey algorithm following Theorem 3.

The iterative Cooley-Tukey algorithm of size $2^6 = 64$ points can be described by the following formula.

$$\begin{aligned} F_{2^6} &= \left\{ \prod_{i=1}^{6} (I_{2^{i-1}} \otimes F_2 \otimes I_{2^{6-i}})(I_{2^i} \otimes T_{2^{6-i-1}}^{2^{6-i}}) \right\} R_{2^6} \\ F_{64} &= (F_2 \otimes I_{32})T_{32}^{64} \cdot (I_2 \otimes F_2 \otimes I_{16})(I_2 \otimes T_{16}^{32}) \cdot \\ &\quad (I_4 \otimes F_2 \otimes I_8)(I_4 \otimes T_8^{16}) \cdot (I_8 \otimes F_2 \otimes I_4)(I_8 \otimes T_4^8) \cdot \\ &\quad (I_{16} \otimes F_2 \otimes I_2)(I_{16} \otimes T_2^4) \cdot (I_{32} \otimes F_2) \cdot R_{64} \end{aligned}$$

Applying the tensor product properities list in Property 1 to each stage of the formula, we have

$$\begin{aligned} F_{64} &= L_2^{64}(I_{32} \otimes F_2)L_{32}^{64}T_{32}^{64} \cdot L_4^{64}(I_{32} \otimes F_2)L_{16}^{64}(I_2 \otimes T_{16}^{32}) \cdot \\ &\quad L_8^{64}(I_{32} \otimes F_2)L_8^{64}(I_4 \otimes T_8^{16}) \cdot L_{16}^{64}(I_{32} \otimes F_2)L_4^{64}(I_8 \otimes T_4^8) \cdot \\ &\quad L_{32}^{64}(I_{32} \otimes F_2)L_2^{64}(I_{16} \otimes T_2^4) \cdot (I_{32} \otimes F_2) \cdot R_{64} \end{aligned}$$

$$\begin{aligned} F_{64} &= L_2^{64}(I_{32} \otimes F_2)T_6 L_{32}^{64} \cdot L_4^{64}(I_{32} \otimes F_2)T_5 L_{16}^{64} \cdot \\ &\quad L_8^{64}(I_{32} \otimes F_2)T_4 L_8^{64} \cdot L_{16}^{64}(I_{32} \otimes F_2)T_3 L_4^{64} \cdot \\ &\quad L_{32}^{64}(I_{32} \otimes F_2)T_2 L_2^{64}(I_{32} \otimes F_2)T_1 \cdot R_{64} \end{aligned}$$

where

$$\begin{aligned} T_6 L_{32}^{64} = L_{32}^{64} T_{32}^{64} \quad &\rightarrow \quad T_6 = L_{32}^{64} T_{32}^{64} L_2^{64} \\ T_5 L_{16}^{64} = L_{16}^{64}(I_2 \otimes T_{16}^{32}) \quad &\rightarrow \quad T_5 = L_{16}^{64}(I_2 \otimes T_{16}^{32}) L_4^{64} \\ T_4 L_8^{64} = L_8^{64}(I_4 \otimes T_8^{16}) \quad &\rightarrow \quad T_4 = L_8^{64}(I_4 \otimes T_8^{16}) L_8^{64} \\ T_3 L_4^{64} = L_4^{64}(I_8 \otimes T_4^{8}) \quad &\rightarrow \quad T_3 = L_4^{64}(I_8 \otimes T_4^{8}) L_{16}^{64} \\ T_2 L_2^{64} = L_2^{64}(I_{16} \otimes T_2^{4}) \quad &\rightarrow \quad T_2 = L_2^{64}(I_{16} \otimes T_2^{4}) L_{32}^{64} \\ &\quad\;\;\;\; T_1 = I_{64} \end{aligned}$$

This is the one-dimensional Pease algorithm of size 64 points. When written in term of Equation A-1, the 3 parameters of the one-dimensional 64-point Pease algorithm are

$$\begin{aligned} P_0 \quad &= \quad R_{64} \\ \texttt{Stage 1:} \quad & \quad P_1 = I_{64}, \quad T_1 = I_{64} \\ \texttt{Stage 2:} \quad & \quad P_2 = L_2^{64}, \quad T_2 = L_2^{64}(I_{16} \otimes T_2^4) L_{32}^{64} \\ \texttt{Stage 3:} \quad & \quad P_3 = L_4^{64}, \quad T_3 = L_4^{64}(I_8 \otimes T_4^8) L_{16}^{64} \\ \texttt{Stage 4:} \quad & \quad P_4 = L_8^{64}, \quad T_4 = L_8^{64}(I_4 \otimes T_8^{16}) L_8^{64} \\ \texttt{Stage 5:} \quad & \quad P_5 = L_{16}^{64}, \quad T_5 = L_{16}^{64}(I_2 \otimes T_4^{32}) L_4^{64} \\ \texttt{Stage 6:} \quad & \quad P_6 = L_{32}^{64}, \quad T_6 = L_{32}^{64} T_{32}^{64} L_2^{64} \end{aligned}$$

Following the dimensionless Pease algorithm (Theorem 8), only the twiddle factor matrices T_i, $1 \leq i \leq 6$, are changed when the Pease algorithm is used to compute different dimensions. To illustrate this point, we consider the two-dimensional (16×4)-point DFT using Pease algorithm. The two-dimensional (16×4)-point DFT can be described by $F_{16} \otimes F_4$. The following steps show derivation of the Pease algorithm from the Cooley-Tukey algorithm (Theorem 3).

$$\begin{aligned}
F_{16} \otimes F_4 &= F'_{16} R_{16} \otimes F'_4 R_4 \\
&= (F'_{16} \otimes F'_4)(R_{16} \otimes R_4) \\
&= (F'_{16} \otimes I_4)(I_{16} \otimes F'_4)(R_{16} \otimes R_4) \\
&= L^{64}_{16}(I_4 \otimes F'_{16}) L^{64}_4 (I_{16} \otimes F'_4)(R_{16} \otimes R_4)
\end{aligned}$$

where

$$\begin{aligned}
F'_{16} &= (F_2 \otimes I_8) T^{16}_8 (I_2 \otimes F_2 \otimes I_4)(I_2 \otimes T^8_4)(I_4 \otimes F_2 \otimes I_2)(I_4 \otimes T^4_2)(I_8 \otimes F_2) \\
F'_4 &= (F_2 \otimes I_2) T^4_2 (I_2 \otimes F_2)
\end{aligned}$$

Replacing F'_{16} and F'_4, we have

$$\begin{aligned}
F_{16} \otimes F_4 &= L^{64}_{16}\{I_4 \otimes (F_2 \otimes I_8) T^{16}_8 (I_2 \otimes F_2 \otimes I_4)(I_2 \otimes T^8_4) \\
&\quad (I_4 \otimes F_2 \otimes I_2)(I_4 \otimes T^4_2)(I_8 \otimes F_2)\} \\
&\quad L^{64}_4 \{I_{16} \otimes (F_2 \otimes I_2) T^4_2 (I_2 \otimes F_2)\}(R_{16} \otimes R_4) \\
&= L^{64}_{16}(I_4 \otimes F_2 \otimes I_8)(I_4 \otimes T^{16}_8)(I_4 \otimes I_2 \otimes F_2 \otimes I_4)(I_4 \otimes I_2 \otimes T^8_4) \\
&\quad (I_4 \otimes I_4 \otimes F_2 \otimes I_2)(I_4 \otimes I_4 \otimes T^4_2)(I_4 \otimes I_8 \otimes F_2) \\
&\quad L^{64}_4 (I_{16} \otimes F_2 \otimes I_2)(I_{16} \otimes T^4_2)(I_{16} \otimes I_2 \otimes F_2)(R_{16} \otimes R_4) \\
&= L^{64}_{16} L^{64}_8 (I_8 \otimes I_4 \otimes F_2) L^{64}_8 (I_4 \otimes T^{16}_8) \cdot L^{64}_{16}(I_4 \otimes I_8 \otimes F_2) L^{64}_4 (I_8 \otimes T^8_4) \cdot \\
&\quad L^{64}_{32}(I_2 \otimes I_{16} \otimes F_2) L^{64}_2 (I_4 \otimes I_4 \otimes T^4_2) \cdot (I_4 \otimes I_8 \otimes F_2) \cdot \\
&\quad L^{64}_4 L^{64}_{32}(I_2 \otimes I_{16} \otimes F_2) L^{64}_2 (I_{16} \otimes T^4_2) \cdot (I_{32} \otimes F_2) \cdot (R_{16} \otimes R_4)
\end{aligned}$$

$$\begin{aligned} F_{16} \otimes F_4 &= L_2^{64}(I_{32} \otimes F_2) L_8^{64}(I_4 \otimes T_8^{16}) L_4^{64} \cdot L_4^{64}(I_{32} \otimes F_2) L_4^{64}(I_8 \otimes T_4^8) L_4^{64} \cdot \\ &\quad L_8^{64}(I_{32} \otimes F_2) L_2^{64}(I_{16} \otimes T_2^4) L_4^{64} \cdot L_{16}^{64}(I_{32} \otimes F_2) L_4^{64} \cdot \\ &\quad L_{32}^{64}(I_{32} \otimes F_2) L_{32}^{64}(I_{16} \otimes T_2^4) \cdot (I_{32} \otimes F_2) \cdot (R_{16} \otimes R_4) \end{aligned}$$

$$\begin{aligned} F_{16} \otimes F_4 &= L_2^{64}(I_{32} \otimes F_2) T_6 L_{32}^{64} \cdot L_4^{64}(I_{32} \otimes F_2) T_5 L_{16}^{64} \cdot \\ &\quad L_8^{64}(I_{32} \otimes F_2) T_4 L_8^{64} \cdot L_{16}^{64}(I_{32} \otimes F_2) T_3 L_4^{64} \cdot \\ &\quad L_{32}^{64}(I_{32} \otimes F_2) T_2 L_2^{64} \cdot (I_{32} \otimes F_2) T_1 \cdot (R_{16} \otimes R_4) \end{aligned}$$

where

$$\begin{aligned} T_6 L_{32}^{64} = L_8^{64}(I_4 \otimes T_8^{16}) L_4^{64} &\rightarrow T_6 = L_{32}^{64}\{L_{16}^{64}(I_4 \otimes T_8^{16}) L_4^{64}\} L_2^{64} \\ T_5 L_{16}^{64} = L_4^{64}(I_8 \otimes T_4^8) L_4^{64} &\rightarrow T_5 = L_{16}^{64}\{L_{16}^{64}(I_8 \otimes T_4^8) L_4^{64}\} L_4^{64} \\ T_4 L_8^{64} = L_2^{64}(I_{16} \otimes T_2^4) L_4^{64} &\rightarrow T_4 = L_8^{64}\{L_{16}^{64}(I_{16} \otimes T_2^4) L_2^{64}\} L_8^{64} \\ T_3 L_4^{64} = L_4^{64} &\rightarrow T_3 = I_{64} \\ T_2 L_2^{64} = L_2^{64}(I_{16} \otimes T_2^4) &\rightarrow T_2 = L_2^{64}(I_{16} \otimes T_2^4) L_{32}^{64} \\ &\quad\;\; T_1 = I_{64} \end{aligned}$$

Written in term of Equation A-1, the 3 parameters of the two-dimensional (16×4)-point Pease algorithm are

$$\begin{aligned} P_0 &= R_{16} \otimes R_4 \\ \texttt{Stage 1:} &\quad P_1 = I_{64}, \quad T_1 = I_{64} \\ \texttt{Stage 2:} &\quad P_2 = L_2^{64}, \quad T_2 = L_2^{64}(I_2 \otimes T_2^4) L_{32}^{64} \\ \texttt{Stage 3:} &\quad P_3 = L_4^{64}, \quad T_3 = I_{64} \\ \texttt{Stage 4:} &\quad P_4 = L_8^{64}, \quad T_4 = L_8^{64}\{L_{16}^{64}(I_{16} \otimes T_2^4) L_4^{64}\} L_8^{64} \\ \texttt{Stage 5:} &\quad P_5 = L_{16}^{64}, \quad T_5 = L_{16}^{64}\{L_{16}^{64}(I_8 \otimes T_4^8) L_4^{64}\} L_4^{64} \\ \texttt{Stage 6:} &\quad P_6 = L_{32}^{64}, \quad T_6 = L_{32}^{64}\{L_{16}^{64}(I_4 \otimes T_8^{16}) L_4^{64}\} L_2^{64} \end{aligned}$$

Notice that the parameters follow Theorem 8 in Section 2.4.

The load-stride permutation $L^{64}_{2^r}$ can be described by a permutation function σ. Therefore, the internal permutation P_i, $1 \leq i \leq 6$, of the Pease algorithm can be specified a permutation function σ_i as following.

$$\texttt{Stage 1}: \quad \sigma_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} = (0,1,2,3,4,5) \tag{A-2}$$

$$\sigma_1^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 2 & 3 & 4 & 5 \end{pmatrix} = (0,1,2,3,4,5)$$

$$\texttt{Stage 2}: \quad \sigma_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 0 \end{pmatrix} = (1,2,3,4,5,0) \tag{A-3}$$

$$\sigma_2^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 0 & 1 & 2 & 3 & 4 \end{pmatrix} = (5,0,1,2,3,4)$$

$$\texttt{Stage 3}: \quad \sigma_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 0 & 1 \end{pmatrix} = (2,3,4,5,0,1) \tag{A-4}$$

$$\sigma_3^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 0 & 1 & 2 & 3 \end{pmatrix} = (4,5,0,1,2,3)$$

$$\texttt{Stage 4}: \quad \sigma_4 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 1 & 2 \end{pmatrix} = (3,4,5,0,1,2) \tag{A-5}$$

$$\sigma_4^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 1 & 2 \end{pmatrix} = (3,4,5,0,1,2) \tag{A-6}$$

$$\texttt{Stage 5}: \quad \sigma_5 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 4 & 5 & 0 & 1 & 2 & 3 \end{pmatrix} = (4,5,0,1,2,3) \tag{A-7}$$

$$\sigma_5^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 3 & 4 & 5 & 0 & 1 \end{pmatrix} = (2,3,4,5,0,1)$$

$$\texttt{Stage 6}: \quad \sigma_6 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 0 & 1 & 2 & 3 & 4 \end{pmatrix} = (5,0,1,2,3,4) \tag{A-8}$$

$$\sigma_6^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 2 & 3 & 4 & 5 & 0 \end{pmatrix} = (1,2,3,4,5,0)$$

A.1.2 Mapping the Pease Algorithm

Using the mapping methodology in Chapter 3, the 64-point Pease algorithm is mapped to 4 processors labeled as PE0, PE1, PE2 and PE3 as following.

A.1.2.1 Address Mapping.

The global addresses generated in stage i can be completely defined by the permutation function σ_i. Let $(b_5b_4b_3b_2b_1b_0)_2$ be binary number representing the order of addresses. Then, the butterfly addresses in stage i can be generated by permuting $(b_5b_4b_3b_2b_1b_0)_2$ with σ_i defined in Equation A-2-A-8 while counting $(b_5b_4b_3b_2b_1b_0)_2$.

$$\texttt{Stage 1} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_1}{\rightarrow} b_5b_4b_3b_2b_1b_0$$

$$\texttt{Stage 2} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_2}{\rightarrow} b_4b_3b_2b_1b_0b_5$$

$$\texttt{Stage 3} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_3}{\rightarrow} b_3b_2b_1b_0b_5b_4$$

$$\texttt{Stage 4} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_4}{\rightarrow} b_2b_1b_0b_5b_4b_3$$

$$\texttt{Stage 5} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_5}{\rightarrow} b_2b_0b_1b_5b_4b_3$$

$$\texttt{Stage 6} : b_5b_4b_3b_2b_1b_0 \overset{\sigma_6}{\rightarrow} b_0b_1b_2b_5b_4b_3$$

Table A-21-A-24 shows the sequence of addresses generated in stage 1, 2, 3, 4, 5, and 6 following the Pease algorithm.

The addresses are mapped to PE0, PE1, PE2 and PE3 in round-robin fashion. The result is that all addresses containing $(b_2b_1)_2 = (00)_2$ are mapped to PE0, that all addresses containing $(b_2b_1)_2 = (01)_2$ are mapped to PE1, that all addresses containing $(b_2b_1)_2 = (10)_2$ are mapped to PE2, and that all addresses containing $(b_2b_1)_2 = (10)_2$ are mapped to PE3. Table A-25 to A-28 in the appendix shows the sequence of addresses that are generated in stage 1, 2, 3, 4, 5, and 6 following the Pease algorithm, and mapped to PE0, PE1, PE2 and PE3.

The address generator in each processor generates local addresses, target PID, and source MID. The local addresses are used for accessing all local memory. The target PID specifies the PID that operates on the data. The source MID specifies the MID from which the input data are coming and to which the results are sent. The bit patterns of the local addresses, target PID, and source MID are derived from σ_i. Following steps demonstrate the steps used for obtaining these bit patterns.

(1) Permute the global counter bits by σ_i. Then, set the 2 most significant bits to the MID (p_1p_0).

$$\texttt{Stage 1}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_1}{\rightarrow} b_5b_4b_3b_2b_1b_0 \rightarrow \underline{p_1p_0}b_3b_2b_1b_0$$

$$\texttt{Stage 2}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_2}{\rightarrow} b_4b_3b_2b_1b_0b_5 \rightarrow \underline{p_1p_0}b_2b_1b_0b_5$$

$$\texttt{Stage 3}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_3}{\rightarrow} b_3b_2b_1b_0b_5b_4 \rightarrow \underline{p_1p_0}b_1b_0b_5b_4$$

$$\texttt{Stage 4}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_4}{\rightarrow} b_2b_1b_0b_5b_4b_3 \rightarrow \underline{p_1p_0}b_0b_5b_4b_3$$

$$\texttt{Stage 5}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_5}{\rightarrow} b_1b_0b_5b_4b_3b_2 \rightarrow \underline{p_1p_0}b_1b_5b_4b_3$$

$$\texttt{Stage 6}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_6}{\rightarrow} b_0b_5b_4b_3b_2b_1 \rightarrow \underline{p_1p_0}b_2b_5b_4b_3$$

(2) Permute the permuted bits back to their original order by the inverse permutation σ_i^{-1}. However, now there are 2 bits that are fixed to the MID (p_1p_0). The remaining bits are the 4-bit counter that will count the local memory addresses. To make it clear, we relabel these bits as $a_3,\ a_2,\ a_1,\ a_0$ while keeping the order of the bits. This procedure maps one-to-one the label $(b_5, b_4, b_3, b_2, b_1, b_0)$ to the new label $(a_3, a_2, a_1, a_0, p_1, p_0)$. The order of the new label depends on the permutation function σ_i. Since $(b_2b_1)_2$ in the old label specifies the target PID, whatever replaces it becomes bit patterns of the target PID.

$$\texttt{Stage 1}: \quad p_1p_0b_3b_2b_1b_0 \overset{\sigma_1^{-1}}{\rightarrow} p_1p_0b_3b_2b_1b_0 \rightarrow p_1p_0a_3\underline{a_2a_1}a_0$$

$$\texttt{target PID} := a_2a_1$$

$$\texttt{Stage 2 :} \quad p_1p_0b_2b_1b_0b_5 \overset{\sigma_2^{-1}}{\rightarrow} b_5p_1p_0b_2b_1b_0 \rightarrow a_3p_1p_0\underline{a_2a_1}a_0$$

$$\texttt{target PID} := a_2a_1$$

$$\texttt{Stage 3 :} \quad p_1p_0b_1b_0b_5b_4 \overset{\sigma_3^{-1}}{\rightarrow} b_5b_4p_1p_0b_1b_0 \rightarrow a_3a_2p_1\underline{p_0a_1}a_0$$

$$\texttt{target PID} := p_0a_1$$

$$\texttt{Stage 4 :} \quad p_1p_0b_0b_5b_4b_3 \overset{\sigma_4^{-1}}{\rightarrow} b_5b_4b_3p_1p_0b_0 \rightarrow a_3a_2a_1\underline{p_1p_0}a_0$$

$$\texttt{target PID} := p_1p_0$$

$$\texttt{Stage 5 :} \quad p_1p_0b_5b_4b_3b_2 \overset{\sigma_5^{-1}}{\rightarrow} b_5b_4b_3b_2p1p_0 \rightarrow a_3a_2a_1\underline{a_0p_1}p_0$$

$$\texttt{target PID} := a_0p_1$$

$$\texttt{Stage 6 :} \quad p_1p_0b_4b_3b_2b_1 \overset{\sigma_6^{-1}}{\rightarrow} p_0b_4b_3b_2b_1p_1 \rightarrow p_0a_3a_2\underline{a_1a_0}p_1$$

$$\texttt{target PID} := a_1a_0$$

(3) Permute the new label by σ_i to get the bit patterns of the local addresses.

$$\begin{aligned}
&\texttt{Stage 1 :} && p_1p_0a_3a_2a_1a_0 \overset{\sigma_1}{\rightarrow} p_1p_0\ \underline{a_3a_2a_1a_0}\\
&\texttt{Stage 2 :} && a_3p_1p_0a_2a_1a_0 \overset{\sigma_2}{\rightarrow} p_1p_0\ \underline{a_2a_1a_0a_3}\\
&\texttt{Stage 3 :} && a_3a_2p_1p_0a_1a_0 \overset{\sigma_3}{\rightarrow} p_1p_0\ \underline{a_1a_0a_3a_2}\\
&\texttt{Stage 4 :} && a_3a_2a_1p_1p_0a_0 \overset{\sigma_4}{\rightarrow} p_1p_0\ \underline{a_0a_3a_2a_1}\\
&\texttt{Stage 5 :} && a_3a_2a_1a_0p_1p_0 \overset{\sigma_5}{\rightarrow} p_1p_0\ \underline{a_3a_2a_1a_0}\\
&\texttt{Stage 6 :} && p_0a_3a_2a_1a_0p_1 \overset{\sigma_6}{\rightarrow} p_1p_0\ \underline{a_3a_2a_1a_0}
\end{aligned}$$

(4) The source data operated by PE number $(p_1p_0)_2$ can be generated by relabeling $(b_5b_4b_3b_1b_0)_2$ such that the $(b_2b_1)_2$ is replaced by $(p_1p_0)_2$. The remaining of the

bits are replaced by the local counter in the same order; i.e. $b_5b_4b_3b_0$ is relabeled to $a_3a_2a_1a_0$. The 2 most significant bits after the permutation σ_i specify the bit pattern of the source MID.

$$\begin{aligned}
&\texttt{Stage 1}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_1}{\rightarrow} \underline{a_3a_2}\ a_1p_1p_0a_0 \\
& && \texttt{source MID} := a_3a_2 \\
&\texttt{Stage 2}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_2}{\rightarrow} \underline{a_2a_1}\ p_1p_0a_0a_3 \\
& && \texttt{source MID} := a_2a_1 \\
&\texttt{Stage 3}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_3}{\rightarrow} \underline{a_1p_1}\ p_0a_0a_3a_2 \\
& && \texttt{source MID} := a_1p_1 \\
&\texttt{Stage 4}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_4}{\rightarrow} \underline{p_1p_0}\ a_0a_3a_2a_1 \\
& && \texttt{source MID} := p_1p_0 \\
&\texttt{Stage 5}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_5}{\rightarrow} \underline{p_0a_0}\ a_3a_2a_1p_1 \\
& && \texttt{source MID} := p_0a_0 \\
&\texttt{Stage 6}: && a_3a_2a_1p_1p_0a_0 \overset{\sigma_6}{\rightarrow} \underline{a_0a_3}\ a_2a_1p_1p_0 \\
& && \texttt{source MID} := a_0a_3
\end{aligned}$$

Table A-1 concludes the bit patterns of local address, target PID and source MID following the optimal algorithm. Table A-29-A-34 in the appendix shows the generation of these numbers in each PE and each stage.

Table A-1 Bit patterns for generating local address, target PID and source MID using the Pease algorithm

Stage	source MID	target PID	Local Address
1	a_3a_2	a_2a_1	$a_3a_2a_1a_0$
2	a_2a_1	a_2a_1	$a_1a_2a_0a_3$
3	a_1p_1	p_0a_1	$a_1a_0a_3a_2$
4	p_1p_0	p_1p_0	$a_0a_3a_2a_1$
5	p_0a_0	a_0p_1	$a_3a_2a_1a_0$
6	a_0a_3	a_1a_0	$a_3a_2a_1a_0$

A.1.2.2 Twiddle Factor Mapping.

Similar to the address mapping, the twiddle factor matrices T_i, $1 \leq i \leq 6$, are mapped to the 4 processors. We will show 2 sets of the twiddle factors for computing 2 different dimensional DFT using the Pease algorithm. These twiddle factors matrices are based on the twiddle factor matrix T_r^{rs} defined in Section 2.1.5. Applying the definition to the twiddle factors of one-dimensional 64-point DFT and two-dimensional (64×4)-point DFT, we have

$$\textbf{One} - \textbf{dimensional 64} - \textbf{point FFT}$$

`Stage 1 :`

$$T_1 = I_{64} = I_{64}$$
$$T_1 = \textbf{diag}(1, \ldots, 1)$$

`Stage 2 :`

$$T_2 = L_2^{64}(I_{16} \otimes T_2^4)L_{32}^{64}$$

$$\begin{aligned}
L_2^{64}(I_{16} &\otimes T_2^4)L_{32}^{64}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= L_2^{64}(I_{16} \otimes T_2^4)(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
&= L_2^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_2^4(e_{b_0}^2 \otimes e_{b_5}^2)) \\
&= \omega_4^{(b_0)_2 \cdot (b_5)_2} L_2^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
&= \omega_4^{(b_0)_2 \cdot (b_5)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_2 &= \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}
\end{aligned}$$

Stage 3 :

$$\begin{aligned}
T_3 &= L_4^{64}(I_8 \otimes T_4^8)L_{16}^{64} \\
& L_4^{64}(I_8 \otimes T_4^8)L_{16}^{64}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= L_4^{64}(I_8 \otimes T_4^8)(e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2) \\
&= L_4^{64}(e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_4^8(e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2)) \\
&= \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2} L_4^{64}(e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2) \\
&= \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_3 &= \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2}
\end{aligned}$$

Stage 4 :

$$\begin{aligned}
T_4 &= L_8^{64}(I_4 \otimes T_8^{16})L_8^{64} \\
& L_8^{64}(I_4 \otimes T_8^{16})L_8^{64}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= L_8^{64}(I_4 \otimes T_8^{16})(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= L_8^{64}(e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_8^{16}(e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2)) \\
&= \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2} L_8^{64}(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_4 &= \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}
\end{aligned}$$

`Stage 5 :`

$$\begin{aligned}
T_5 &= L^{64}_{16}(I_2 \otimes T^{32}_{16})L^{64}_{4} \\
& L^{64}_{16}(I_2 \otimes T^{32}_{16})L^{64}_{4}(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
&= L^{64}_{16}(I_2 \otimes T^{32}_{16})(e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2}) \\
&= L^{64}_{16}(e^2_{b_1} \otimes T^{32}_{16}(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2})) \\
&= \omega_{32}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2)_2} L^{64}_{16}(e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2})) \\
&= \omega_{32}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2)_2} (e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
T_5 &= \bigoplus_{b_5,\ldots,b_0} \omega_{32}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2)_2}
\end{aligned}$$

`Stage 6 :`

$$\begin{aligned}
T_6 &= L^{64}_{32}T^{64}_{32}L^{64}_{2} \\
& L^{64}_{32}T^{64}_{32}L^{64}_{2}(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
&= L^{64}_{32}T^{64}_{32}(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1}) \\
&= \omega_{64}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2 b_1)_2} L^{64}_{32}(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1}) \\
&= \omega_{64}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_3 b_1)_2} (e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
T_6 &= \bigoplus_{b_5,\ldots,b_0} \omega_{64}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2 b_1)_2}
\end{aligned}$$

Two – dimensional (16 × 4) – point FFT

`Stage 1 :` $T_1 = I_{64} = I_{64}$

$$T_1 = \mathbf{diag}(1, \ldots, 1)$$

`Stage 2 :` $T_2 = L_2^{64}(I_{16} \otimes T_2^4)L_{32}^{64}$

$$\begin{aligned}
& L_2^{64}(I_{16} \otimes T_2^4)L_{32}^{64}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
= \; & L_2^{64}(I_{16} \otimes T_2^4)(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
= \; & L_2^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_2^4(e_{b_0}^2 \otimes e_{b_5}^2)) \\
= \; & \omega_4^{(b_0)_2 \cdot (b_5)_2} L_2^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
= \; & \omega_4^{(b_0)_2 \cdot (b_5)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2)
\end{aligned}$$

$$T_2 = \bigoplus_{b_5, \ldots, b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$$

`Stage 3 :` $T_3 = I_{64} = I_{64}$

$$T_3 = \mathbf{diag}(1, \ldots, 1)$$

`Stage 4 :` $T_4 = L_8^{64} L_{16}^{64}(I_{16} \otimes T_2^4)L_4^{64} L_8^{64}$

$$\begin{aligned}
& L_8^{64} L_{16}^{64}(I_{16} \otimes T_2^4)L_4^{64} L_8^{64}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
= \; & L_8^{64} L_{16}^{64}(I_{16} \otimes T_2^4)L_4^{64}(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
= \; & L_8^{64} L_{16}^{64}(I_{16} \otimes T_2^4)(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
= \; & L_8^{64} L_{16}^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_2^4(e_{b_0}^2 \otimes e_{b_5}^2)) \\
= \; & \omega_4^{(b_0)_2 \cdot (b_5)_2} L_8^{64} L_{16}^{64}(e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2) \\
= \; & \omega_4^{(b_0)_2 \cdot (b_5)_2} L_8^{64}(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
= \; & \omega_4^{(b_0)_2 \cdot (b_5)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2)
\end{aligned}$$

$$T_4 = \bigoplus_{b_5, \ldots, b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$$

$$
\begin{aligned}
\texttt{Stage 5:}\quad & T_5 = L^{64}_{16}L^{64}_{16}(I_8 \otimes T^8_4)L^{64}_4 L^{64}_4 \\
& L^{64}_{16}L^{64}_{16}(I_8 \otimes T^8_4)L^{64}_4 L^{64}_4(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
= \ & L^{64}_{16}L^{64}_{16}(I_8 \otimes T^8_4)L^{64}_4(e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2}) \\
= \ & L^{64}_4(I_8 \otimes T^8_4)(e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4}) \\
= \ & L^{64}_4(e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes T^4_2(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4})) \\
= \ & \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2} L^{64}_{16}L^{64}_{16}(e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4}) \\
= \ & \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2}(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
T_5 = \ & \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5 b_4)_2}
\end{aligned}
$$

$$
\begin{aligned}
\texttt{Stage 6:}\quad & T_6 = L^{64}_{32}L^{64}_{16}(I_4 \otimes T^{16}_8)L^{64}_4 L^{64}_2 \\
& L^{64}_{32}L^{64}_{16}(I_4 \otimes T^{16}_8)L^{64}_4 L^{64}_2(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
= \ & L^{64}_{32}L^{64}_{16}(I_4 \otimes T^{16}_8)L^{64}_4(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1}) \\
= \ & L^{64}_8(I_4 \otimes T^{16}_8)(e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3}) \\
= \ & L^{64}_8(e^2_{b_2} \otimes e^2_{b_1} \otimes T^{16}_8(e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3})) \\
= \ & \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2} L^{64}_{32}(e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3}) \\
= \ & \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
T_6 = \ & \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}
\end{aligned}
$$

Table A-2 summarizes the twiddle factors for one-dimensional 64-point DFT and two-dimensional (16×4)-point DFT using the Pease algorithm.

Since all twiddle factors are written in term of $\omega_N^r = e^{2\pi \frac{r}{N} i}$, they can be represented by the fraction $\frac{r}{N}$. A n_{max}-bit binary number, $2^{n_{max}} \geq N$ may be used to represent a fraction $\frac{r}{N}$. For the sake of illustration, let n_{max} be equal to 8. Then, the 8-binary number $(0.r_7 \cdots r_0)_2$ represents a fraction $\frac{r}{2^8}$, where $r = (r_7 \cdots r_0)_2$. Therefore, if we

Table A-2 Twiddle factors of the Pease algorithm for one-dimensional 64-point DFT and two-dimensional (16×4)-point DFT

Stage	T_i	
i	1-D 64-point	2-D (16×4)-point
1	$T_1 = \mathbf{diag}(1, \ldots, 1)$	$T_1 = \mathbf{diag}(1, \ldots, 1)$
2	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$
3	$T_3 = \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2}$	$T_3 = \mathbf{diag}(1, \ldots, 1)$
4	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}$	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$
5	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{32}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2)_2}$	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_5 b_4)_2}$
6	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{64}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2 b_1)_2}$	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}$

want to represent twiddle fractions $\frac{r}{N}$ with 8-bit binary number, all twiddle factors should be converted to $\omega_{2^8}^r$, where $\omega_{2^8} = e^{2\pi \frac{r}{2^8} i}$ is the 256-th root of unity. For example, since $\omega_8 = \omega_{2^8}^{2^5}$, the twiddle factor of one-dimensional 64-point FFT at stage 3 becomes

$$T_3 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4)_2 \cdot 2^5}$$

which can be represented by binary number $(b_5 b_4)_2 \cdot 2^5 = (0 b_5 b_4 000000)_2$.

Converting the twiddle factors in Table A-2 to $\omega_{2^8}^r$ result in the twiddle factors in Table A-3. The value of b_0 identifies whether the address is the first or second

Table A-3 Twiddle factors of the Pease algorithm written in term of ω_{256}

Stage	T_i	
i	1-D 64-point	2-D (16×4)-point
1	$T_1 = \mathbf{diag}(1, \ldots, 1)$	$T_1 = \mathbf{diag}(1, \ldots, 1)$
2	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5)_2 \cdot 2^6}$	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5)_2 \cdot 2^6}$
3	$T_3 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4)_2 \cdot 2^5}$	$T_3 = \mathbf{diag}(1, \ldots, 1)$
4	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2 \cdot 2^4}$	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5)_2 \cdot 2^6}$
5	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2)_2 \cdot 2^3}$	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4)_2 \cdot 2^5}$
6	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4 b_3 b_2 b_1)_2 \cdot 2^2}$	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2 \cdot 2^4}$

Table A-4 Twiddle fractions of Pease algorithm for one-dimensional 64-point DFT and two-dimensional (16 × 4)-point DFT

Stage	Twiddle Fraction = $(r_7 \cdots r_0)$	
i	1-D 64-point	2-D (16 × 4)-point
1	$(00000000)_2$	$(00000000)_2$
2	$(0a_3000000)_2$	$(0a_3000000)_2$
3	$(0a_3a_200000)_2$	$(00000000)_2$
4	$(0a_3a_2a_10000)_2$	$(0a_3000000)_2$
5	$(0a_3a_2a_1p_1000)_2$	$(0a_3a_200000)_2$
6	$(0a_3a_2a_1p_1p_000)_2$	$(0a_3a_2a_10000)_2$

address of the butterfly operation. When $b_0 = 0$, it is the first address and the twiddle factor for this datum is always equal to 1. When $b_1 = 1$ signifies that the address is for the second datum which will be multiplied by twiddle factor ω_{256}^r. Therefore, the twiddle factors in Table A-3 can be written as the twiddle fractions in Table A-4.

The twiddle factors are mapped to PE0, PE1, PE2 and PE3 in round-robin fashion. Therefore, the $(b_2 b_1)$ is replaced by PID $(p_1 p_0)$ and $(b_5 b_4 b_3 b_0)$ is replaced by the output of the local counter $(a_3 a_2 a_1 a_0)$. Table A-35-A-38 show the twiddle fractions of the Pease algorithms generated in each stage. Table A-39-A-50 show the twiddle fractions of the Pease algorithms generated in each stage inside each PE.

A.1.3 Implementation

The address and twiddle fraction generators can be implemented using MUXs or adders. In both cases, we can systematically translate the bit patterns into parameters needed for the implementation.

First, we need to specify the compile-time parameters. In addition to the algorithm specified by P_i and T_i and the number of processors specified by m, where 2^m is the number of processors, the other compile-time parameter is the maximum size specified by n_{max}. This means that the universal FFT engine can compute FFT of size 2^m to $2^{n_{max}}$ points. Let n_{max} be equal to 8. This means that the local addresses generated inside each processor is represented by a $n_{max} - m = 8 - 2 = 6$ bit number.

A.1.3.1 Address Generation.

Implementation Using MUXs

The bit pattern for generating the local addresses is the permutation of the output of 6-bit counter, $(a_5a_4a_3a_2a_1a_0)_2$. Let $(c_5c_4c_3c_2c_1c_0)_2$ be the local address at a particular stage. Let $a_0, \ldots a_5$ be the inputs of a 6-to-1 MUX whose MUX select is s_j and whose output is c_j, where $c_j = a_{s_j}$, $0 \leq j < 8$, and s_j can be derived directly from the bit pattern of the local address shown in Table A-1.

Similarly, we can generate target PID and source MID using MUXs. However, since the bit patterns for generating target PID or source MID includes the PID $((p_1p_0)_2)$, 10-to-1 MUXs are required. We would need 2 10-to-1 MUXs for generating target PID and 2 10-to-1 MUXs for generating source MID. Let $(d_1d_0)_2$ and $(e_1e_0)_2$ be the target PID and the source MID at a particular stage. Let st_j and ss_j be the MUX selects for generating d_j and e_j, $0 \leq j < 2$ respectively. Then,

$$d_j = \begin{cases} p_0 & \texttt{if } st_j = 0 \\ p_1 & \texttt{if } st_j = 1 \\ a_0 & \texttt{if } st_j = 2 \\ a_1 & \texttt{if } st_j = 3 \\ a_2 & \texttt{if } st_j = 4 \\ a_3 & \texttt{if } st_j = 5 \\ a_4 & \texttt{if } st_j = 6 \\ a_5 & \texttt{if } st_j = 7 \end{cases}, \quad 0 \leq j < 2$$

$$e_j = \begin{cases} a_0 & \texttt{if } ss_j = 0 \\ p_0 & \texttt{if } ss_j = 1 \\ p_1 & \texttt{if } ss_j = 2 \\ a_1 & \texttt{if } ss_j = 3 \\ a_2 & \texttt{if } ss_j = 4 \\ a_3 & \texttt{if } ss_j = 5 \\ a_4 & \texttt{if } st_j = 6 \\ a_5 & \texttt{if } st_j = 7 \end{cases}, \quad 0 \le j < 2$$

Table A-5 MUX selects for generating local address, target PID and source MID of the 64-point Pease algorithm

Stage	Local Address						target PID		source MID	
	s_5	s_4	s_3	s_2	s_1	s_0	st_1	st_0	ss_1	ss_0
1	5	4	3	2	1	0	4	3	5	4
2	5	4	2	1	0	3	4	3	4	3
3	5	4	1	0	3	2	0	3	3	2
4	5	4	0	3	2	1	1	0	2	1
5	5	4	3	2	1	0	2	1	1	0
6	5	4	3	2	1	0	3	2	0	5

Table A-5 shows the MUXs selects for generating local address, target PID and source MID of the Pease algorithm of size 64 points. These MUX selects may be pre-stored in a ROM or generated. Figure A-1 shows the address generator for the Pease algorithm of size 64 points using MUXs.

The generation of the MUX selects depends on the runtime parameter which the size of FFT specified by n, where 2^n is the number of points. Therefore, for our example, $n = 6$. The MUX select s_j, $0 \le j < n - m$, can be generated using parallel shifting. At the first stage, $s_j = j$, $0 \le j < n - m$. Then, when changing the stage and the current stage is less than or equal to $n - m = 4$, the s_0 is loaded with output from a countdown counter whose output is initialize to $n - m - 1 = 3$ and s_j, $1 \le j < n - m - 1$ is loaded with s_{j-1} resulting in the MUXs selects of the next

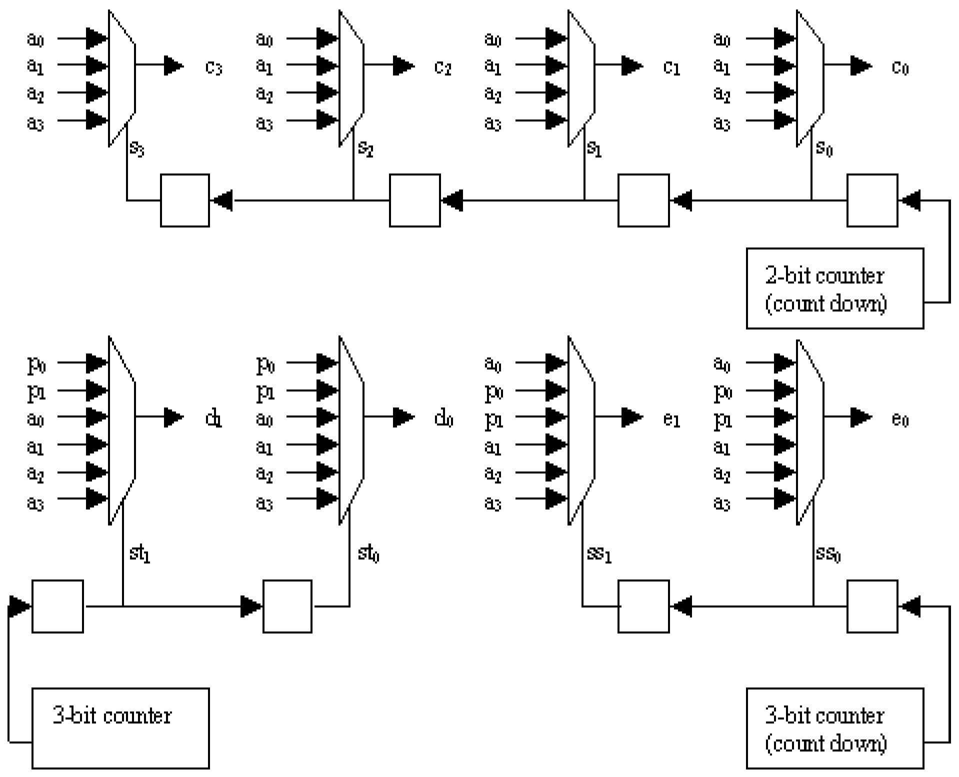


Figure A-1 Address generator for Pease FFT algorithm of size 64 points using MUXs

stage. For the remaining m stage (stage 5 and 6), the MUX selects are reset back to their initial values.

The MUX select st_j and ss_j, $0 \leq j < 2$, can also be generated using parallel shifting. At the first stage, st_j is initialized to $n-m-j = 6-2+j-1 = 3+j$. Then, when changing of stage and current stage is greater than or equal to $n-2m = 6-4 = 2$, $st_{m-1} = s_1$ is loaded with output from the 3-bit counter whose output is initialized to 0. At the same time, the counter counts up and st_j, $0 \leq j < m-1$, is loaded with st_{j+1}.

At the first stage, ss_j, $0 \leq j < 2$, is initialized to $n-1+j = 5+j$. Then, when changing stage, ss_0 is loaded with output from the 3-bit countdown counter whose output is initialized to $n - m - 1 = 6 - 2 - 1 = 3$. At the same time, ss_j, $0 < j < m$, is loaded with ss_{j-1} and the counter counts down. If the current count value is 0, the next count value is set to $n - 1$.

Implementation Using Adders

The address generator can also be generated using adder as shown in Figure 5.3. We will first focus on how the local addresses are generated using adders.

Notice that the bit patterns for generating local addresses have at most 2 portions of consecutive bits. Therefore, we need 2 increment numbers at each stage.

At the first stage, the pattern is $(00a_3a_2a_1a_0)_2$ which can be generated by setting the initial number (INIT) to $(000000)_2$ and the increment number to $(000001)_2$. Since we always start from address 0, the INIT value at the beginning of stage is always equal to $(000000)_2$. For $2 \leq i \leq 4$ the pattern becomes $(00a_{i-2} \cdots a_0a_3 \cdots a_{i-1})$. The portions of consecutive bits are $(00a_{i-2} \cdots a_0)$ and $(00a_3 \cdots a_{i-1})$, $1 \leq i \leq 4$. For stage $5 \leq i \leq 6$, the bit pattern is reset back to $(00a_3a_2a_1a_0)$ which can be

Let $\texttt{INC}_1(i)$ be the increment number at stage i when there is no carry, $\texttt{INC}_2(i)$ be the increment number at stage i when there is carry from a_{i-2} to a_{i-1}. To accommo-

Table A-6 Increment and initial numbers for generating address of Pease algorithm using adders

Stage i	INIT(i)	Increment		
		$\texttt{INC}_1(i)$	$\texttt{INC}_2(i)$	$\texttt{INC}_2(i)$ in Stride
1	$(000000)_2$	$(000001)_2 = 1$	n/c	n/c
2	$(000000)_2$	$(000010)_2 = 2$	$(110011)_2 = -13$	$2^1 = 2$
3	$(000000)_2$	$(000100)_2 = 4$	$(110101)_2 = -11$	$2^2 = 4$
4	$(000000)_2$	$(001000)_2 = 8$	$(111001)_2 = -7$	$2^3 = 8$
5	$(000000)_2$	$(000001)_2 = 1$	n/c	n/c
6	$(000000)_2$	$(000001)_2 = 1$	n/c	n/c

date the carry propagation, the increment $\texttt{INC}_1(i)$ is equal to $(0\cdots010\cdots0)_2 = 2^{i-1}$, $1 \leq i \leq 4$, and the $\texttt{INC}_2(i)$ is equal to $-2^{n-m} + 2^{i-1} + 1 = -2^4 + 2^{i-1} + 1$. Table A-6 shows the initial and increment numbers for generating address using adders.

Table A-7 Addresses generated following Pease algorithm of size 64 points during stage 3 using adders

Count 1	ACC	INC	Stride 2^2 Count	inc2_hit
0	$(000000)_2 = 0$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(010000)_2$	0
1	$(000100)_2 = 4$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(100000)_2$	0
2	$(001000)_2 = 8$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(110000)_2$	0
3	$(001100)_2 = 12$	$\texttt{INC}_2(3) = (110101)_2 = -11$	$(000000)_2$	1
4	$(000001)_2 = 1$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(010000)_2$	0
5	$(000101)_2 = 5$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(100000)_2$	0
6	$(001001)_2 = 9$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(110000)_2$	0
7	$(001101)_2 = 13$	$\texttt{INC}_2(3) = (110101)_2 = -11$	$(000000)_2$	1
8	$(000010)_2 = 2$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(010000)_2$	0
9	$(000110)_2 = 6$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(100000)_2$	0
10	$(001010)_2 = 10$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(110000)_2$	0
11	$(001110)_2 = 14$	$\texttt{INC}_2(3) = (110101)_2 = -11$	$(000000)_2$	1
12	$(000011)_2 = 3$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(010000)_2$	0
13	$(000111)_2 = 7$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(100000)_2$	0
14	$(001011)_2 = 11$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(110000)_2$	0
15	$(001111)_2 = 15$	$\texttt{INC}_2(3) = (110101)_2 = -11$	$(000000)_2$	1

The carry from a_{i-2} to a_{i-1} occurs only when $(a_{i-2}\cdots a_0) = (1\cdots1)_2$. In other words, it occurs in stride 2^{i-1}. Therefore, if $(a_{i-2}\cdots a_0) = (1\cdots1)_2$, $\texttt{INC}_2(i)$ is selected as the increment number, else $\texttt{INC}_2(i)$ is selected. Another adder for which the initial number (INIT) and increment number (INC) at stage i are 2^{6-i-1} can be used for checking whether $\texttt{INC}_2(i)$ is selected. For example, at stage 3 the adder is initialized to $2^{6-3-1} = (010000)_2$ and incremented by the same number. This produces the carry out (called "inc2_hit" in Table A-7) in every 4 addresses (stride 4). Table A-7 shows how the addresses are generated during stage 3, where $\texttt{INC}_1(3) = (000100)_2 = 2^2$ and $\texttt{INC}_2(3) = (110101)_2 = -2^4 + 2^2 + 1 = -11$.

Since INIT(i) are constant, we need to generate only $\texttt{INC}_1(i)$ and $\texttt{INC}_2(i)$. Since $\texttt{INC}_1(i+1) = 2^{i+1} = 2 \cdot 2^i$, it can be generated by shifting its current value ($\texttt{INC}_1(i) = 2^i$) to the left by 1 bit.

The $\texttt{INC}_2(i)$ be generated as following. Since $\texttt{INC}_2(i) = (-2^4 + 1) + 2^{i-1}$, it can be generated by adding $\texttt{INC}_1(i)$ with $-2^4 + 1 = (110001)_2$. Moreover, for $1 \leq i < 4$, it is guaranteed that the adding has no carry. Therefore, it can be accomplished using **OR** gates; i.e. $\texttt{INC}_2(i)$ is equal to $\texttt{INC}_1(i)$ **OR** $(110001)_2$. For example, $\texttt{INC}_2(3)$ is equal to $(000100)_2$ **OR** $(110001)_2 = (110101)_2 = -11$.

A.1.3.2 Twiddle Fraction Generator.

As shown in Chapter 5, we can generate the twiddle fractions of any dimension by masking off unwanted bits of one-dimensional twiddle fraction. To illustrate this property, let us consider the bit patterns for one-dimensional 64-point and two-dimensional (16×4)-point twiddle fractions.

Table A-4 shows the bit patterns for generating twiddle fractions of one-dimensional and two-dimensional FFT of size 64 points following the Pease algorithm. Both twiddle fractions can be generated by (1) generating a twiddle fraction denoted as TF1 and (2) masking off unwanted bits of TF1. Table A-8 shows the bit patterns of TF1 and the masks for both one-dimensional 64-point and two-dimensional (16×4)-point FFT following the Pease algorithm. Following this implementation, we can parameterize the design such that the twiddle fraction TF1 depends solely on the algorithm and the size and that the mask depends solely on the dimension specification. For our example, the Pease algorithm of size 64 specifies that TF1 is $(0a_3a_2a_1p_1p_000)_2$ for

Table A-8 Generation of twiddle fractions for Pease algorithm of different dimensions using masks

Twiddle Fraction = TF1 **AND** M									
Stage i	Twiddle fraction before masking (TF1)	Mask (M) = $(2^j-1)\cdot 2^{8-j}$							
		1-D 64-point (2^{n_1}-point)				2-D (16×4)-point ($2^{n_1}\times 2^{n_2}$-point)			
		k	n_k	j	M	k	n_k	j	M
1	$(0a_3a_2a_1p_1p_000)_2$	1	6	0	$(00000000)_2$	2	2	0	$(00000000)_2$
2	$(0a_3a_2a_1p_1p_000)_2$			1	$(01000000)_2$			1	$(01000000)_2$
3	$(0a_3a_2a_1p_1p_000)_2$			2	$(01100000)_2$	1	4	0	$(00000000)_2$
4	$(0a_3a_2a_1p_1p_000)_2$			3	$(01110000)_2$			1	$(01000000)_2$
5	$(0a_3a_2a_1p_1p_000)_2$			4	$(01111000)_2$			2	$(01100000)_2$
6	$(0a_3a_2a_1p_1p_000)_2$			5	$(01111100)_2$			3	$(01110000)_2$

every stage and the mask at stage i, $1\le i\le 6$, is equal to $(2^j-1)\cdot 2^{8-j-1}$, where $0\le j<n_k$, $k=D,\cdots,1$ and D is number of dimensions. For instance, the two-dimensional (16×4)-point DFT is specified by $D=2$, $n_1=4$ and $n_2=2$. This results that $j=0,1,0,1,2,3$ and that the masks are $(2^0-1)\cdot 2^{8-1}=0$, $(2^1-1)\cdot 2^{8-2}=2^6$, $(2^0-1)\cdot 2^{8-1}=0$, $(2^1-1)\cdot 2^{8-2}=2^6$, $(2^2-1)\cdot 2^{8-3}=3\cdot 2^5$ and $(2^3-1)\cdot 2^{8-4}=7\cdot 2^4$ respectively. The implementation of the masks is explained in Chapter 5.

The TF1 can be generated using MUXs or adders. When using MUXs, we need 8 7-to-1 MUXs whose MUX selects are parameterized during the runtime by the size. Let $(r_7\cdots r_0)$ be the 8-bit TF1 and r_j, $0\le j<8$, be the output of a 8-to-1 MUX whose MUX select is sf_j. Then,

$$r_j = \begin{cases} p_0 & \texttt{if } sf_j=0 \\ p_1 & \texttt{if } sf_j=1 \\ a_1 & \texttt{if } sf_j=2 \\ a_2 & \texttt{if } sf_j=3 \\ a_3 & \texttt{if } sf_j=4 \\ a_4 & \texttt{if } sf_j=5 \\ a_5 & \texttt{if } sf_j=6 \end{cases}, \quad 0\le j<8.$$

For the size 2^6 points, the MUX selects are following.

$$\begin{aligned} sf_0 &= 0, \quad sf_1 = 0, \quad sf_2 = 0, \quad sf_3 = 1 \\ sf_4 &= 2, \quad sf_5 = 3, \quad sf_6 = 4, \quad sf_7 = 5 \end{aligned}$$

When using adders, the initial number (INIT) and the increment number (INC) are parameterized during the runtime by the size. For the size 2^n points, the INIT is equal to $(p_{m-1} \cdots p_0)_2 \cdot 2^{8-n}$ and the INC is equal to $2^{8-(n-m)}$. Note that the increment is done in every 2 addresses. For 64-point and 4 processors (n = 6 and m = 2), the INIT is equal to $(p_1p_0)_2 \cdot 2^{8-6} = (0000p_1p_000)_2$ and the INC is equal to $2^{8-6+2} = (00010000)_2$. Table A-9 shows the generation of TF1 during any stages in each PE.

Table A-9 Generation of twiddle fractions before masking (TF1) following Pease algorithm of size 64 points using adders

INIT = $(p_1p_0) \cdot 2^{8-6} = (0000p_1p_000)$ and INC = $2^{8-6+2} = (00010000)$				
$a_3a_2a_1$	Twiddle Fraction (TF1)			
	PE0	PE1	PE2	PE3
	$p_1p_0 = 00$	$p_1p_0 = 01$	$p_1p_0 = 10$	$p_1p_0 = 11$
0	$(00000000) = \frac{0}{64}$	$(00000100) = \frac{1}{64}$	$(00001000) = \frac{2}{64}$	$(00001100) = \frac{3}{64}$
1	$(00010000) = \frac{4}{64}$	$(00010100) = \frac{5}{64}$	$(00011000) = \frac{6}{64}$	$(00011100) = \frac{7}{64}$
2	$(00100000) = \frac{8}{64}$	$(00100100) = \frac{9}{64}$	$(00101000) = \frac{10}{64}$	$(00101100) = \frac{11}{64}$
3	$(00110000) = \frac{12}{64}$	$(00110100) = \frac{13}{64}$	$(00111000) = \frac{14}{64}$	$(00111100) = \frac{15}{64}$
4	$(01000000) = \frac{16}{64}$	$(01000100) = \frac{17}{64}$	$(01001000) = \frac{18}{64}$	$(01001100) = \frac{19}{64}$
5	$(01010000) = \frac{20}{64}$	$(01010100) = \frac{21}{64}$	$(01011000) = \frac{22}{64}$	$(01011100) = \frac{23}{64}$
6	$(01100000) = \frac{24}{64}$	$(01100100) = \frac{25}{64}$	$(01101000) = \frac{26}{64}$	$(01101100) = \frac{27}{64}$
7	$(01110000) = \frac{28}{64}$	$(01110100) = \frac{29}{64}$	$(01111000) = \frac{30}{64}$	$(01111100) = \frac{31}{64}$

A.2 Example 2: The Optimal Algorithm

In this section, we will walk through all steps in the design methodology when the optimal algorithm is chosen.

A.2.1 Formula Representing the Optimal Algorithm

The optimal algorithm is obtained from the optimization step described in Chapter 4. For FFT of size 64 points and 4 processors, the optimal algorithm is described by following formula in Equation A-1, where the internal permutation P_i, $1 \leq i \leq 6$, can be specified a permutation function σ_i and its inverse σ_i^{-1} as follows.

$$\texttt{Stage 1}: \quad \sigma_1 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 4 & 5 & 1 & 2 & 3 \end{pmatrix} = (0,4,5,1,2,3) \tag{A-9}$$

$$\sigma_1^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 0 & 3 & 4 & 5 & 1 & 2 \end{pmatrix} = (0,3,4,5,1,2)$$

$$\texttt{Stage 2}: \quad \sigma_2 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 1 & 4 & 5 & 0 & 2 & 3 \end{pmatrix} = (1,4,5,0,2,3) \tag{A-10}$$

$$\sigma_2^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 0 & 4 & 5 & 1 & 2 \end{pmatrix} = (3,0,4,5,1,2)$$

$$\texttt{Stage 3}: \quad \sigma_3 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 2 & 4 & 5 & 0 & 1 & 3 \end{pmatrix} = (2,4,5,0,1,3) \tag{A-11}$$

$$\sigma_3^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 0 & 5 & 1 & 2 \end{pmatrix} = (3,4,0,5,1,2)$$

$$\texttt{Stage 4}: \quad \sigma_4^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 1 & 2 \end{pmatrix} = (3,4,5,0,1,2) \tag{A-12}$$

$$\sigma_4^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 0 & 1 & 2 \end{pmatrix} = (3,4,5,0,1,2)$$

$$\texttt{Stage 5}: \quad \sigma_5 = \begin{pmatrix} 5 & 4 & 3 & 2 & 1 & 0 \\ 4 & 3 & 5 & 0 & 1 & 2 \end{pmatrix} = (4,3,5,0,1,2) \tag{A-13}$$

$$\sigma_5^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 1 & 0 & 2 \end{pmatrix} = (3,4,5,1,0,2)$$

$$\texttt{Stage 6}: \quad \sigma_6 = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 5 & 4 & 3 & 0 & 1 & 2 \end{pmatrix} = (5,4,3,0,1,2) \tag{A-14}$$

$$\sigma_6^{-1} = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 \\ 3 & 4 & 5 & 2 & 1 & 0 \end{pmatrix} = (3,4,5,2,1,0)$$

This set of permutation functions specifies the permutation P_i shown as following.

$$\begin{aligned}
&\texttt{Stage 1:} \quad && P_1 = (L_8^{32} \otimes I_2)\\
&\texttt{Stage 2:} \quad && P_2 = (L_4^{32} \otimes I_2)L_2^{64}\\
&\texttt{Stage 3:} \quad && P_3 = (L_2^{32} \otimes I_2)L_4^{64}\\
&\texttt{Stage 4:} \quad && P_4 = L_8^{64}\\
&\texttt{Stage 5:} \quad && P_5 = (L_{16}^{32} \otimes I_2)L_{16}^{64}\\
&\texttt{Stage 6:} \quad && P_6 = (L_8^{16} \otimes I_4)(L_{16}^{32} \otimes I_2)L_{32}^{64},
\end{aligned}$$

The initial permutation, P_0, and the twiddle factors T_i, $1 \leq i \leq 6$, depend on the dimension specification. The following table shows the initial permutation and the twiddle factors of the optimal algorithm for one-dimensional 64-point DFT and two-dimensional (16×4)-point DFT respectively. The twiddle factors follow Theorem 9.

$$\begin{array}{ll}
\multicolumn{2}{c}{\textbf{One} - \textbf{dimensional 64} - \textbf{point DFT}}\\
 & P_0 = R_{64}\\
\texttt{Stage 1:} & T_1 = P_1 I_{64} P_1^{-1} = I_{64}\\
\texttt{Stage 2:} & T_2 = P_2(I_{16} \otimes T_2^4)P_2^{-1}\\
\texttt{Stage 3:} & T_3 = P_3(I_8 \otimes T_4^8)P_3^{-1}\\
\texttt{Stage 4:} & T_4 = P_4(I_4 \otimes T_8^{16})P_4^{-1}\\
\texttt{Stage 5:} & T_5 = P_5(I_2 \otimes T_{16}^{32})P_5^{-1}\\
\texttt{Stage 6:} & T_6 = P_6 T_{32}^{64} P_6^{-1}
\end{array}$$

$$\begin{array}{ll}
\multicolumn{2}{c}{\textbf{Two} - \textbf{dimensional } \mathbf{16 \times 4} - \textbf{point DFT}}\\
 & P_0 = R_{16} \otimes R_4\\
\texttt{Stage 1:} & T_1 = P_1 I_{64} P_1^{-1} = I_{64}\\
\texttt{Stage 2:} & T_2 = P_2(I_{16} \otimes T_2^4)P_2^{-1}\\
\texttt{Stage 3:} & T_3 = P_3 L_{16}^{64} I_{64} L_4^{64} P_3^{-1} = I_{64}\\
\texttt{Stage 4:} & T_4 = P_4 L_{16}^{64}(I_{16} \otimes T_2^4)L_4^{64} P_4^{-1}\\
\texttt{Stage 5:} & T_5 = P_5 L_{16}^{64}(I_8 \otimes T_4^8)L_4^{64} P_5^{-1}\\
\texttt{Stage 6:} & T_6 = P_6 L_{16}^{64}(I_4 \otimes T_8^{16})L_4^{64} P_6^{-1}
\end{array}$$

Following the definition of T_r^{rs} defined in Section 2.1.5, we compute twiddle factors in each stage for both cases as follows.

$$\textbf{One}-\textbf{dimensional 64}-\textbf{point FFT}$$

`Stage 1`:

$$\begin{aligned} T_1 &= P_1 I_{64} P_1^{-1} = I_{64} \\ T_1 &= \mathbf{diag}(1, \ldots, 1) \end{aligned}$$

`Stage 2`:

$$\begin{aligned} T_2 &= P_2(I_{16} \otimes T_2^4)P_2^{-1} \\ P_2(I_{16} &\otimes T_2^4)P_2^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\ &= P_2(I_{16} \otimes T_2^4)(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_0}^2 \otimes e_{b_3}^2) \\ &= P_2(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes T_2^4(e_{b_0}^2 \otimes e_{b_3}^2)) \\ &= \omega_4^{(b_0)_2 \cdot (b_3)_2} P_2(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_0}^2 \otimes e_{b_3}^2) \\ &= \omega_4^{(b_0)_2 \cdot (b_3)_2} (e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\ T_2 &= \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_3)_2} \end{aligned}$$

`Stage 3`:

$$\begin{aligned} T_3 &= P_3(I_8 \otimes T_4^8)P_3^{-1} \\ & \quad P_3(I_8 \otimes T_4^8)P_3^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\ &= P_3(I_8 \otimes T_4^8)(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_0}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\ &= P_3(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes T_4^8(e_{b_0}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2)) \\ &= \omega_8^{(b_0)_2 \cdot (b_4 b_3)_2} P_3(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_0}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\ &= \omega_8^{(b_0)_2 \cdot (b_4 b_3)_2} (e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\ T_3 &= \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_4 b_3)_2} \end{aligned}$$

Stage 4 :

$$\begin{aligned}
T_4 &= P_4(I_4 \otimes T_8^{16})P_4^{-1} \\
& P_4(I_4 \otimes T_8^{16})P_4^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= P_4(I_4 \otimes T_8^{16})(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= P_4(e_{b_2}^2 \otimes e_{b_1}^2 \otimes T_8^{16}(e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2)) \\
&= \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2} P_4(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_4 &= \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}
\end{aligned}$$

Stage 5 :

$$\begin{aligned}
T_5 &= P_5(I_2 \otimes T_{16}^{32})P_5^{-1} \\
& P_5(I_2 \otimes T_{16}^{32})P_5^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= P_5(I_2 \otimes T_{16}^{32})(e_{b_2}^2 \otimes e_{b_4}^0 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= P_5(e_{b_2}^2 \otimes T_{16}^{32}(e_{b_0}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2)) \\
&= \omega_{32}^{(b_0)_2 \cdot (b_1 b_5 b_4 b_3)_2} P_5(e_{b_2}^2 \otimes e_{b_0}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2)) \\
&= \omega_{32}^{(b_0)_2 \cdot (b_1 b_5 b_4 b_3)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_5 &= \bigoplus_{b_5,\ldots,b_0} \omega_{32}^{(b_0)_2 \cdot (b_1 b_5 b_4 b_3)_2}
\end{aligned}$$

`Stage 6 :`

$$\begin{aligned}
T_6 &= P_6 T_{32}^{64} P_6^{-1} \\
& \quad P_6 T_{32}^{64} P_6^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= P_6 T_{32}^{64}(e_{b_0}^2 \otimes e_{b_1}^2 \otimes e_{b_2}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= \omega_{64}^{(b_0)_2 \cdot (b_1 b_2 b_5 b_4 b_3)_2} P_6(e_{b_0}^2 \otimes e_{b_1}^2 \otimes e_{b_2}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2) \\
&= \omega_{64}^{(b_0)_2 \cdot (b_1 b_2 b_5 b_4 b_3)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_6 &= \bigoplus_{b_5,\ldots,b_0} \omega_{64}^{(b_0)_2 \cdot (b_1 b_2 b_5 b_4 b_3)_2}
\end{aligned}$$

Two − dimensional (16 × 4) − point FFT

`Stage 1 :` $T_1 = P_1 I_{64} P_1^{-1} = I_{64}$

$$T_1 = \mathbf{diag}(1, \ldots, 1)$$

`Stage 2 :` $T_2 = P_2(I_{16} \otimes T_2^4) P_2^{-1}$

$$\begin{aligned}
& P_2(I_{16} \otimes T_2^4) P_2^{-1}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
&= P_2(I_{16} \otimes T_2^4)(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_0}^2 \otimes e_{b_3}^2)
\end{aligned}$$

$$\begin{aligned}
&= P_2(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes T_2^4(e_{b_0}^2 \otimes e_{b_3}^2)) \\
&= \omega_4^{(b_0)_2 \cdot (b_3)_2} P_2(e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_0}^2 \otimes e_{b_3}^2) \\
&= \omega_4^{(b_0)_2 \cdot (b_3)_2}(e_{b_5}^2 \otimes e_{b_4}^2 \otimes e_{b_3}^2 \otimes e_{b_2}^2 \otimes e_{b_1}^2 \otimes e_{b_0}^2) \\
T_2 &= \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_3)_2}
\end{aligned}$$

$$\texttt{Stage 3}: \quad T_3 = P_3 I_{64} P_3^{-1} = I_{64}$$

$$T_3 = \mathbf{diag}(1,\ldots,1)$$

$$\texttt{Stage 4}: \quad T_4 = P_4 L_{16}^{64}(I_{16}\otimes T_2^4)L_4^{64}P_4^{-1}$$

$$\begin{aligned}
& P_4 L_{16}^{64}(I_{16}\otimes T_2^4)L_4^{64}P_4^{-1}(e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2)\\
= {} & P_4 L_{16}^{64}(I_{16}\otimes T_2^4)L_4^{64}(e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2\otimes e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2)\\
= {} & P_4 L_{16}^{64}(I_{16}\otimes T_2^4)(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2\otimes e_{b_5}^2)\\
= {} & P_4 L_{16}^{64}(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes T_2^4(e_{b_0}^2\otimes e_{b_5}^2))\\
= {} & \omega_4^{(b_0)_2\cdot(b_5)_2} P_4 L_{16}^{64}(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2\otimes e_{b_5}^2)\\
= {} & \omega_4^{(b_0)_2\cdot(b_5)_2} P_4(e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2\otimes e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2)\\
= {} & \omega_4^{(b_0)_2\cdot(b_5)_2}(e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2)
\end{aligned}$$

$$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2\cdot(b_5)_2}$$

$$\texttt{Stage 5}: \quad T_5 = P_5 L_{16}^{64}(I_8\otimes T_4^8)L_4^{64}P_5^{-1}$$

$$\begin{aligned}
& P_5 L_{16}^{64}(I_8\otimes T_4^8)L_4^{64}P_5^{-1}(e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2)\\
= {} & P_4 L_{16}^{64}(I_8\otimes T_4^8)L_4^{64}(e_{b_2}^2\otimes e_{b_0}^2\otimes e_{b_1}^2\otimes e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2)\\
= {} & P_5 L_{16}^{64}(I_8\otimes T_4^8)(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_0}^2\otimes e_{b_1}^2\otimes e_{b_5}^2)\\
= {} & P_5 L_{16}^{64}(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes T_2^4(e_{b_0}^2\otimes e_{b_1}^2\otimes e_{b_5}^2))\\
= {} & \omega_8^{(b_0)_2\cdot(b_1b_5)_2} P_5 L_{16}^{64}(e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_0}^2\otimes e_{b_1}^2\otimes e_{b_5}^2)
\end{aligned}$$

$$\begin{aligned}
= {} & \omega_8^{(b_0)_2\cdot(b_1b_5)_2} P_5(e_{b_2}^2\otimes e_{b_0}^2\otimes e_{b_1}^2\otimes e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2)\\
= {} & \omega_8^{(b_0)_2\cdot(b_1b_5)_2}(e_{b_5}^2\otimes e_{b_4}^2\otimes e_{b_3}^2\otimes e_{b_2}^2\otimes e_{b_1}^2\otimes e_{b_0}^2)
\end{aligned}$$

$$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2\cdot(b_1b_5)_2}$$

`Stage 6 :` $T_6 = P_6 L^{64}_{16}(I_4 \otimes T^{16}_8) L^{64}_4 P_6^{-1}$

$$\begin{aligned}
& P_6 L^{64}_{16}(I_4 \otimes T^{16}_8) L^{64}_4 P_6^{-1}(e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
= \; & P_6 L^{64}_{16}(I_4 \otimes T^{16}_8) L^{64}_4 (e^2_{b_0} \otimes e^2_{b_1} \otimes e^2_{b_2} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3}) \\
= \; & P_6 L^{64}_4 (I_4 \otimes T^{16}_8)(e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_0} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_5}) \\
= \; & P_6 L^{64}_{16}(e^2_{b_4} \otimes e^2_{b_3} \otimes T^{16}_8(e^2_{b_0} \otimes e^2_{b_1} \otimes e^2_{b_2} \otimes e^2_{b_5})) \\
= \; & \omega_{16}^{(b_0)_2 \cdot (b_1 b_2 b_5)_2} P_6 L^{64}_{16}(e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_0} \otimes e^2_{b_1} \otimes e^2_{b_2} \otimes e^2_{b_5}) \\
= \; & \omega_{16}^{(b_0)_2 \cdot (b_1 b_2 b_5)_2} P_6 (e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0} \otimes e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3}) \\
= \; & \omega_{16}^{(b_0)_2 \cdot (b_1 b_2 b_5)_2} (e^2_{b_5} \otimes e^2_{b_4} \otimes e^2_{b_3} \otimes e^2_{b_2} \otimes e^2_{b_1} \otimes e^2_{b_0}) \\
T_6 = \; & \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_1 b_2 b_5)_2}
\end{aligned}$$

Table A-10 summarizes the twiddle factors of optimal algorithm for one-dimensional

Table A-10 Twiddle factors of optimal algorithm for one-dimensional 64-point DFT and two-dimensional (16×4)-point DFT

Stage	T_i	
i	1-D 64-point	2-D (16×4)-point
1	$T_1 = \mathbf{diag}(1, \ldots, 1)$	$T_1 = \mathbf{diag}(1, \ldots, 1)$
2	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_3)_2}$	$T_2 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_3)_2}$
3	$T_3 = \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_4 b_3)_2}$	$T_1 = \mathbf{diag}(1, \ldots, 1)$
4	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_5 b_4 b_3)_2}$	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_4^{(b_0)_2 \cdot (b_5)_2}$
5	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{32}^{(b_0)_2 \cdot (b_1 b_5 b_4 b_3)_2}$	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_8^{(b_0)_2 \cdot (b_1 b_5)_2}$
6	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{64}^{(b_0)_2 \cdot (b_1 b_2 b_5 b_4 b_3)_2}$	$T_1 = \bigoplus_{b_5,\ldots,b_0} \omega_{16}^{(b_0)_2 \cdot (b_1 b_2 b_5)_2}$

64-point DFT and two-dimensional (16×4)-point DFT.

A.2.2 Mapping the Optimal Algorithm

Following the mapping methodology in Chapter 3, the 64-point optimal algorithm is mapped to 4 processors as follows.

A.2.2.1 Address Mapping.

The global addresses generated in stage i can be completely defined by the permutation function σ_i. Let $(b_5b_4b_3b_2b_1b_0)_2$ be a binary number representing the output of a counter. Then, the butterfly addresses in stage i can be generated by permuting $(b_5b_4b_3b_2b_1b_0)_2$ with σ_i defined in Equation A-9-A-14 while counting $(b_5b_4b_3b_2b_1b_0)_2$.

$$\texttt{Stage 1}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_1}{\rightarrow} b_2b_1b_5b_4b_3b_0$$

$$\texttt{Stage 2}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_2}{\rightarrow} b_2b_1b_4b_3b_0b_5$$

$$\texttt{Stage 3}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_3}{\rightarrow} b_2b_1b_3b_0b_5b_4$$

$$\texttt{Stage 4}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_4}{\rightarrow} b_2b_1b_0b_5b_4b_3$$

$$\texttt{Stage 5}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_5}{\rightarrow} b_2b_0b_1b_5b_4b_3$$

$$\texttt{Stage 6}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_6}{\rightarrow} b_0b_1b_2b_5b_4b_3$$

Table A-51-A-54 in the appendix shows the sequence of addresses generated in stage 1, 2, 3, 4, 5, and 6 following the optimal algorithm.

The addresses are mapped to PE0, PE1, PE2 and PE3 in round-robin fashion. The result is that all addresses containing $(b_2b_1)_2 = (00)_2$ are mapped to PE0, that all addresses containing $(b_2b_1)_2 = (01)_2$ are mapped to PE1, that all addresses containing $(b_2b_1)_2 = (10)_2$ are mapped to PE2, and that all addresses containing $(b_2b_1)_2 = (10)_2$ are mapped to PE3. Table A-55-A-58 in the appendix shows the sequence of addresses that are generated in stage 1, 2, 3, 4, 5, and 6 following the optimal algorithm, and mapped to PE0, PE1, PE2 and PE3.

The address generator in each processor generates local addresses, target PID, and source MID. The bit patterns of generating the local addresses, the target PID, and the source MID are derived from σ_i. The following steps demonstrate the steps used for obtaining these bit patterns.

(1) Permute the global counter bits by σ_i. Then, set the 2 most significant bits to the MID ($p_1 p_0$).

$$\texttt{Stage 1}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_1}{\rightarrow} b_2b_1b_5b_4b_3b_0 \rightarrow \underline{p_1p_0}\ b_5b_4b_3b_0$$

$$\texttt{Stage 2}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_2}{\rightarrow} b_2b_1b_4b_3b_0b_5 \rightarrow \underline{p_1p_0}\ b_4b_3b_0b_5$$

$$\texttt{Stage 3}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_3}{\rightarrow} b_2b_1b_3b_0b_5b_4 \rightarrow \underline{p_1p_0}\ b_3b_0b_5b_4$$

$$\texttt{Stage 4}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_4}{\rightarrow} b_2b_1b_0b_5b_4b_3 \rightarrow \underline{p_1p_0}\ b_0b_5b_4b_3$$

$$\texttt{Stage 5}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_5}{\rightarrow} b_2b_0b_1b_5b_4b_3 \rightarrow \underline{p_1p_0}\ b_1b_5b_4b_3$$

$$\texttt{Stage 6}: b_5b_4b_3b_2b_1b_0 \overset{\sigma_6}{\rightarrow} p_1p_0b_5b_4b_3b_2 \rightarrow \underline{p_1p_0}\ b_5b_4b_3b_2$$

(2) Permute the permuted bits back to their original order by the inverse permutation σ_i^{-1}. However, now there are 2 bits that are fixed to the MID ($p_1 p_0$). The remaining bits are the 4-bit counter that will count the local memory addresses. To make it clear, we relabel these bits as $a_3,\ a_2,\ a_1,\ a_0$ while keeping the order of the bits. This procedure maps one-to-one the label $(b_5, b_4, b_3, b_2, b_1, b_0)$ to the new label $(a_3, a_2, a_1, a_0, p_1, p_0)$. The order of the new label depends on the

permutation function σ_i. Since $(b_2b_1)_2$ in the old label specifies the target PID, whatever replaces it becomes bit patterns of the target PID.

$$\begin{array}{ll}
\texttt{Stage 1}: & p_1p_0b_5b_4b_3b_0 \overset{\sigma_1^{-1}}{\rightarrow} b_5b_4b_3p_1p_0b_0 \rightarrow a_3a_2a_1\underline{p_1p_0}a_0 \\
 & \texttt{PID} := p_1p_0 \\
\texttt{Stage 2}: & p_1p_0b_4b_3b_0b_5 \overset{\sigma_2^{-1}}{\rightarrow} b_5b_4b_3p_1p_0b_0 \rightarrow a_3a_2a_1\underline{p_1p_0}a_0 \\
 & \texttt{PID} := p_1p_0 \\
\texttt{Stage 3}: & p_1p_0b_3b_0b_5b_4 \overset{\sigma_3^{-1}}{\rightarrow} b_5b_4b_3p_1p_0b_0 \rightarrow a_3a_2a_1\underline{p_1p_0}a_0 \\
 & \texttt{PID} := p_1p_0 \\
\texttt{Stage 4}: & p_1p_0b_0b_5b_4b_3 \overset{\sigma_4^{-1}}{\rightarrow} b_5b_4b_3p_1p_0b_0 \rightarrow a_3a_2a_1\underline{p_1p_0}a_0 \\
 & \texttt{PID} := p_1p_0 \\
\texttt{Stage 5}: & p_1p_0b_1b_5b_4b_3 \overset{\sigma_5^{-1}}{\rightarrow} b_5b_4b_3p_1b_1p_0 \rightarrow a_3a_2a_1\underline{p_1a_0}p_0 \\
 & \texttt{PID} := p_1a_0 \\
\texttt{Stage 6}: & p_1p_0b_5b_4b_3b_2 \overset{\sigma_6^{-1}}{\rightarrow} b_5b_4b_3b_2p_0p_1 \rightarrow a_3a_2a_1\underline{a_0p_0}p_1 \\
 & \texttt{PID} := a_0p_0
\end{array}$$

(3) Permute the new label by σ_i to get the bit patterns of the local addresses.

$$\begin{array}{ll}
\texttt{Stage 1}: & a_3a_2a_1p_1p_0a_0 \overset{\sigma_1}{\rightarrow} p_1p_0\ \underline{a_3a_2a_1a_0} \\
\texttt{Stage 2}: & a_3a_2a_1p_1p_0a_0 \overset{\sigma_2}{\rightarrow} p_1p_0\ \underline{a_3a_2a_0a_1} \\
\texttt{Stage 3}: & a_3a_2a_1p_1p_0a_0 \overset{\sigma_3}{\rightarrow} p_1p_0\ \underline{a_3a_0a_2a_1} \\
\texttt{Stage 4}: & a_3a_2a_1p_1p_0a_0 \overset{\sigma_4}{\rightarrow} p_1p_0\ \underline{a_0a_3a_2a_1} \\
\texttt{Stage 5}: & a_3a_2a_1p_1a_0p_0 \overset{\sigma_5}{\rightarrow} p_1p_0\ \underline{a_0a_3a_2a_1} \\
\texttt{Stage 6}: & a_3a_2a_1a_0p_0p_1 \overset{\sigma_6}{\rightarrow} p_1p_0\ \underline{a_0a_3a_2a_1}
\end{array}$$

(4) The source data operated by PE number $(p_1p_0)_2$ can be generated by relabeling $(b_5b_4b_3b_1b_0)_2$ such that the $(b_2b_1)_2$ is replaced by $(p_1p_0)_2$. The remaining of the bits are replaced by the local counter in the same order; i.e. $b_5b_4b_3b_0$ is relabeled to $a_3a_2a_1a_0$. The 2 most significant bits after the permutation σ_i specify the bit pattern of the source MID.

$$
\begin{array}{ll}
\texttt{Stage 1:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_1} \underline{p_1p_0}\ a_3a_2a_1a_0 \\
 & \texttt{MID} := p_1p_0 \\
\texttt{Stage 2:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_2} \underline{p_1p_0}\ a_3a_2a_0a_1 \\
 & \texttt{MID} := p_1p_0 \\
\texttt{Stage 3:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_3} \underline{p_1p_0}\ a_3a_0a_2a_1 \\
 & \texttt{MID} := p_1p_0 \\
\texttt{Stage 4:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_4} \underline{p_1p_0}\ a_0a_3a_2a_1 \\
 & \texttt{MID} := p_1p_0 \\
\texttt{Stage 5:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_5} \underline{p_1a_0}\ p_0a_3a_2a_1 \\
 & \texttt{MID} := p_1a_0 \\
\texttt{Stage 6:} & a_3a_2a_1p_1p_0a_0 \xrightarrow{\sigma_6} \underline{a_0p_0}\ a_3a_2a_1p_1 \\
 & \texttt{MID} := a_0p_0
\end{array}
$$

Table A-11 concludes the bit patterns of local address, target PID and source MID following the optimal algorithm. Table A-59-A-64 in the appendix list these numbers in each PE and each stage.

A.2.2.2 Twiddle Factor Mapping.

Table A-10 shows the twiddle factors for one-dimensional 64-point DFT and two-dimensional (16 × 4)-point DFT using the optimal algorithm. Since, we represent a

Table A-11 Bit patterns for generating local address, target PID and source MID using the optimal algorithm

Stage	source MID	target PID	Local Address
1	p_1p_0	p_1p_0	$a_3a_2a_1a_0$
2	p_1p_0	p_1p_0	$a_3a_2a_0a_1$
3	p_1p_0	p_1p_0	$a_3a_0a_2a_1$
4	p_1p_0	p_1p_0	$a_0a_3a_2a_1$
5	p_1a_0	p_1a_0	$a_0a_3a_2a_1$
6	a_0p_0	a_0p_1	$a_0a_3a_2a_1$

fraction by 8-bit binary number, we convert the twiddle factors in Table A-10 to $w^r_{2^8}$ resulting in the twiddle factors in Table A-12. The value of b_0 identifies whether the

Table A-12 Twiddle factors of optimal algorithm for one-dimensional 64-point DFT and two-dimensional (16 × 4)-point DFT

Stage	T_i	
i	1-D 64-point	2-D (16 × 4)-point
1	$T_1 = \mathbf{diag}(1, \ldots, 1)$	$T_1 = \mathbf{diag}(1, \ldots, 1)$
2	$T_2 = \bigoplus_{b_9,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_3)_2 \cdot 2^6}$	$T_2 = \bigoplus_{b_9,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_3)_2 \cdot 2^6}$
3	$T_3 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_4b_3)_2 \cdot 2^5}$	$T_3 = \mathbf{diag}(1, \ldots, 1)$
4	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5b_4b_3)_2 \cdot 2^4}$	$T_4 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_5)_2 \cdot 2^6}$
5	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_1b_5b_4b_3)_2 \cdot 2^3}$	$T_5 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_1b_5)_2 \cdot 2^5}$
6	$T_6 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_1b_2b_5b_4b_3)_2 \cdot 2^2}$	$T_1 = \bigoplus_{b_5,\ldots,b_0} \omega_{256}^{(b_0)_2 \cdot (b_1b_2b_5)_2 \cdot 2^4}$

address is the first or second address of the butterfly operation. When $b_0 = 0$, it is the first address and the twiddle factor for this datum is always equal to 1. When $b_1 = 1$ signifies that the address is for the second datum which will be multiplied by twiddle factor ω^r_{256}. Therefore, the twiddle factors in Table A-12 can be written as the twiddle fractions in Table A-13. Since the twiddle factors are mapped to PE0, PE1, PE2 and PE3 in round-robin fashion, (b_2b_1) is replaced by PID (p_1p_0) and $(b_5b_4b_3b_0)$ is replaced by the output of the local counter $(a_3a_2a_1a_0)$. Table A-65 to A-68 show

Table A-13 Twiddle fractions of optimal algorithm for one-dimensional 64-point DFT and two-dimensional (16×4)-point DFT

Stage	Twiddle Fraction = $(r_7 \cdots r_0)$	
i	1-D 64-point	2-D (16×4)-point
1	$(000000)_2$	$(000000)_2$
2	$(0a_1000000)_2$	$(0a_1000000)_2$
3	$(0a_2a_2000000)_2$	$(00000000)_2$
4	$(0a_3a_2a_10000)_2$	$(0a_3000000)_2$
5	$(0p_0a_3a_2b_1000)_2$	$(0p_0a_3000000)_2$
6	$(0p_0p_1a_3a_2a_100)_2$	$(0p_0p_1a_30000)_2$

the twiddle fractions of 1-D 64-point DFT using the optimal algorithm generated in each stage. Table A-69 to A-72 show the twiddle fractions of 1-D 64-point DFT using the optimal algorithms generated in each stage inside each PE. Table A-73 to A-76 show the twiddle fractions of 2-D (16×4)-point the optimal algorithms generated in each stage and the round-robin mapping.

A.2.3 Implementation

The address and twiddle fraction generators can be implemented using MUXs or adders. In both cases, we can systematically translate the bit patterns into parameters needed for the corresponding implementation.

In addition to the algorithm specified by P_i and T_i and number of processors specified by m, where 2^m is the number of processors, the other compile-time parameter is the maximum size specified by n_{max}. This number specifies that the universal FFT engine can compute FFT of size ranging from 2^m to $2^{n_{max}}$ points. For the purpose of demonstration, we set n_{max} to 8. This implies that a local address is represented by a 6-bit binary number while a twiddle fraction is represented by an 8-bit binary number.

A.2.3.1 Address Generation Unit.

Implementation using MUXs

The bit patterns for generating local address, target PID, and source MID following the optimal algorithm of size 64 point shown in Table A-11 provides us the following nice properties.

- Target PID and source MID are the same pattern. Moreover, they are equal to the PID (p_1p_0) during stage 1 to 4 and there is only 1 bit different between them and (p_1p_0) during stage 5 and 6. Therefore, we do not need to generate the whole target PID and source MID. Only a flag identifying whether a particular address is remote or local is needed. This can be done because at particular stage, data involving a processor can either come or go to its memory or another remote memory. In other words, only a pair wise interconnection is required at each stage. Therefore, we can deterministically configure the interconnection network in each stage to accommodate such communication. Moreover, there are only 2 stages that need the interconnection.

  For our example, all addresses are local during stage 1 to 4. During stage 5, the pair wise connections are that PE0 connects with PE1 and that PE2 connects with PE3. During stage 6, PE0 connects with PE2 while PE1 connects with PE3.

- There are at most only 3 possible bits that are permuted to become an output bit. In other words, instead of using 8-to-1 MUX, we need only 3-to-1 MUXs regardless of the size. Specifically, either a_0, a_j or a_{j+1} is permuted to become the j^{th} bit of the output. Let $(c_5c_4c_3c_2c_1c_0)_2$ be the local address. Let a_0, a_j or

a_{j+1} be the 3 possible choices of the 3-to-1 mux whose output is c_j and s_j is its MUX select. Then,

$$c_j = \begin{cases} a_0 & \texttt{if } s_j = 0 \\ a_j & \texttt{if } s_j = 1 \\ a_{j+1} & \texttt{if } s_j = 2 \end{cases}, \quad 0 \leq j < 6 \texttt{ and } a_6 = a_5$$

Figure A-2 shows the address generator for optimal algorithm of size 64 using

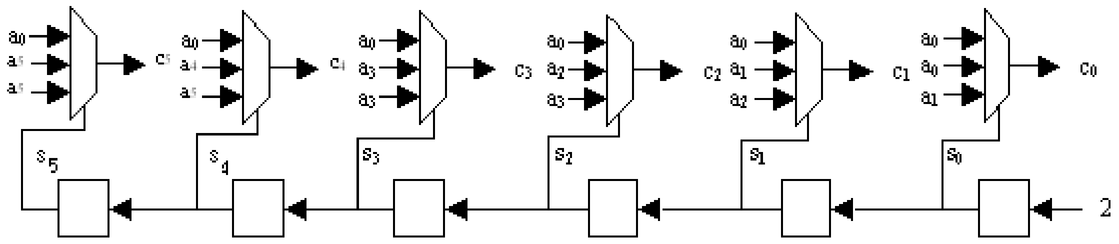


Figure A-2 Address generator for optimal FFT algorithm of size 64 points using 3-to-1 MUXs

Table A-14 MUX selects for generating local address of the optimal algorithm

Stage	s_5	s_4	s_3	s_2	s_1	s_0
1	1	1	1	1	1	0
2	1	1	1	1	0	2
3	1	1	1	0	2	2
4	1	1	0	2	2	2
5	1	1	0	2	2	2
6	1	1	0	2	2	2

3-to-1 MUXs. Table A-11 shows the MUX selects at each stage. Note that the MUX selects can be generated easily using shift registers as shown Figure A-2. The MUX select s_j is the output of a 2-bit register. At the first stage, the registers are initialized such that $s_0 = 0$, $s_j = 1$, $1 \leq j < 4$. Then, when changing the stage and the current stage is less than 4, the s_0 is loaded with 2 and s_j, $1 \leq j < 4$ is with s_{j-1} resulting in the MUXs selects of the next stage.

For the remaining stages (stage 5 and 6), the MUX selects stay the same as in stage 4.

Implementation Using Adders

The address generator can also be generated using an adder as shown in Figure 5.3. As describing in above section, we do not need to generate the target PID and source MID. We will focus on how the local addresses are generated using adders.

Since the bit patterns for generating an address have at most 3 portions of consecutive bits, we needs 3 increment numbers at each stage. At the first stage, the pattern is $(00a_3a_2a_1a_0)$ which can be generated by setting the initial number (INIT) to $(000000)_2$ and the increment number to $(000001)_2$. Since we always start from address 0, the INIT value at the beginning of stage is always equal to $(000000)_2$. For $2 \leq i \leq 4$ the pattern becomes $(00a_3 \cdots a_i a_0 a_{i-1} \cdots a_1)$. The portions of consecutive bits are (a_0), $(a_i \cdots a_1)$ and $(00a_3 \cdots a_i)$, $1 \leq i \leq 4$. This implies that we need 3 increment numbers. Let $\mathtt{INC}_1(i)$ be the increment number at stage i when there is no carry, $\mathtt{INC}_2(i)$ be the increment number at stage i when there is carry from a_0 to a_1 but no carry from a_{i-1} to a_{i-1}, and $\mathtt{INC}_3(i)$ be the increment number at stage i when there is carry from a_{i-1} to a_i. To accommodate the carry propagation, the

Table A-15 Increment and initial number for generating address using adders

Stage i	INIT(i)	Increment Number		
		$\mathtt{INC}_1(i)$	$\mathtt{INC}_2(i)$	$\mathtt{INC}_3(i)$
1	$(000000)_2$	$(000001)_2 = 1$	n/c	$(000001)_2 = 1$
2	$(000000)_2$	$(000010)_2 = 2$	$(111111)_2 = -1$	$(000001)_2 = 1$
3	$(000000)_2$	$(000100)_2 = 4$	$(111101)_2 = -3$	$(000001)_2 = 1$
4	$(000000)_2$	$(001000)_2 = 8$	$(111001)_2 = -7$	$(000001)_2 = 1$
5	$(000000)_2$	$(001000)_2 = 8$	$(111001)_2 = -7$	$(000001)_2 = 1$
6	$(000000)_2$	$(001000)_2 = 8$	$(111001)_2 = -7$	$(000001)_2 = 1$

increment $\texttt{INC}_1(i)$ is equal to $(0\cdots010\cdots0)_2 = 2^{i-1}$, $1 \le i \le 4$, the $\texttt{INC}_2(i)$ is equal to $(1\cdots10\cdots01)_2 = 1-2^{i-1}$ and the $\texttt{INC}_3(i)$ is always equal to $(0\cdots1)_2$. Table A-15 shows the initial and increment numbers for generating the address using adders.

Table A-16 Addresses generated during stage 3 using adders

Addr. Count	ACC	INC	Stride 2^3 Count	inc3_ hit
0	$(0000)_2 = 0$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(001000)_2$	0
1	$(000100)_2 = 4$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(010000)_2$	0
2	$(000001)_2 = 1$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(011000)_2$	0
3	$(000101)_2 = 5$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(100000)_2$	0
4	$(000010)_2 = 2$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(101000)_2$	0
5	$(000110)_2 = 6$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(110000)_2$	0
6	$(000011)_2 = 3$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(111000)_2$	0
7	$(000111)_2 = 7$	$\texttt{INC}_3(3) = (000001)_2 = 1$	$(000000)_2$	1
8	$(001000)_2 = 8$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(001000)_2$	0
9	$(001100)_2 = 12$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(010000)_2$	0
10	$(001001)_2 = 9$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(011000)_2$	0
11	$(001101)_2 = 13$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(100000)_2$	0
12	$(001010)_2 = 10$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(101000)_2$	0
13	$(001110)_2 = 14$	$\texttt{INC}_2(3) = (111101)_2 = -3$	$(110000)_2$	0
14	$(001011)_2 = 11$	$\texttt{INC}_1(3) = (000100)_2 = 4$	$(111000)_2$	0
15	$(001111)_2 = 15$	$\texttt{INC}_3(3) = (000001)_2 = 1$	$(000000)_2$	1

The carry from a_{i-1} to a_i occurs only when $(a_{i-1}\cdots a_0) = (1\cdots1)_2$. In other words, it occurs in stride 2^i. Another adder similar to Figure 5.3 may be used for checking whether $\texttt{INC}_2(i)$ is selected. If we set both initial number (INIT) and increment number (INC) of the adder at stage i to 2^{6-i}, the adder counts in stride 2^i. For example, at stage 3, the INIT and INC is $2^{6-3} = (001000)_2$. This produces a carry out called "inc3_hit" in every 8 counts.

The carry from a_0 to a_1 occurs in every other count (stride 2). This simply is a toggle flip-flop. Therefore, if "inc3_hit" is active, $\texttt{INC}_3(i)$ is selected as the increment number, else if $a_0 = $ '1', $\texttt{INC}_2(i)$ is selected; otherwise, $\texttt{INC}_1(i)$ is selected. Table A-16 shows how the addresses are generated during stage 3, where $\texttt{INC}_1(3) = (000100)_2 = 2^2$, $\texttt{INC}_2(i) = (111101)_2 = 1 - 2^2 = -3$ and $\texttt{INC}_3(i) = (000001)_2 = 1$.

Since $\texttt{INC}_3(i)$ and INIT(i) are constant, we need to generate only $\texttt{INC}_1(i)$ and $\texttt{INC}_2(i)$. Since $\texttt{INC}_1(i+1) = 2^{i+1} = 2 \cdot 2^i$, it can be generated by shifting its current value ($\texttt{INC}_1(i) = 2^i$) to the left by 1 bit. Similarly, if we keep the LSB bit of $\texttt{INC}_2(i+1)$ equal to '1', then its remaining value is $-2^{i+1} = -2 \cdot 2^i$ which can also be generated by shifting its current value (-2^i) to the left by 1 bit.

A.2.3.2 Twiddle Fraction Generator.

As shown in Chapter 5, we can generate the twiddle fractions of any dimension by masking off unwanted bits of one-dimensional twiddle fraction.

Table A-13 shows the bit patterns for generating twiddle fractions of one-dimensional and two-dimensional FFT of size 64 points following the optimal algorithm. Both

Table A-17 Generation of twiddle fractions for the optimal algorithm of different dimensions using masks

Twiddle Fraction = TF1 **AND** M									
Stage i	Twiddle fraction before masking (TF1)	Mask (M) = $(2^j - 1) \cdot 2^{8-j}$							
		1-D 64-point (2^{n_1}-point)				2-D (16 × 4)-point ($2^{n_1} \times 2^{n_2}$-point)			
		k	n_k	j	M	k	n_k	j	M
1	$(00000000)_2$	1	6	0	$(00000000)_2$	2	2	0	$(00000000)_2$
2	$(0a_1000000)_2$			1	$(01000000)_2$			1	$(01000000)_2$
3	$(0a_2a_100000)_2$			2	$(01100000)_2$	1	4	0	$(00000000)_2$
4	$(0a_3a_2a_10000)_2$			3	$(01110000)_2$			1	$(01000000)_2$
5	$(0p_0a_3a_2a_1000)_2$			4	$(01111000)_2$			2	$(01100000)_2$
6	$(0p_0p_1a_3a_2a_100)_2$			5	$(01111100)_2$			3	$(01110000)_2$

twiddle fractions can be generated by (1) generating a twiddle fraction denoted as TF1 and (2) masking off unwanted bits of TF1. Table A-17 shows the bit patterns of

TF1 and the masks for both one-dimensional 64-point and two-dimensional (16×4)-point FFT following the Pease algorithm. Following this implementation, we can parameterize the design such that the twiddle fraction TF1 depends solely on the algorithm and the size and that the mask depends solely on the dimension specification. The implementation of the masks is explained in Chapter 5.

The TF1 can be generated using MUXs or adders. When using MUXs, we need 8 7-to-1 MUXs whose MUX selects are parameterized during the runtime by the size. Let $(r_7 \cdots r_0)$ be the 8-bit TF1 and r_j, $0 \leq j < 8$, be the output of a 8-to-1 MUX whose MUX select is sf_j. Then,

$$r_j = \begin{cases} p_0 & \texttt{if } sf_j = 0 \\ p_1 & \texttt{if } sf_j = 1 \\ a_1 & \texttt{if } sf_j = 2 \\ a_2 & \texttt{if } sf_j = 3 \\ a_3 & \texttt{if } sf_j = 4 \\ a_4 & \texttt{if } sf_j = 5 \\ a_5 & \texttt{if } sf_j = 6 \end{cases}, \quad 0 \leq j < 8.$$

Table A-18 MUX selects for generating local address of the optimal algorithm

Stage	sf_7	sf_6	sf_5	sf_4	sf_3	sf_2	sf_1	sf_0
1	0	2	2	2	2	2	2	2
2	0	2	2	2	2	2	2	2
3	0	3	2	2	2	2	2	2
4	0	4	3	2	2	2	2	2
5	0	0	4	3	2	2	2	2
6	0	1	0	4	3	2	2	2

When using adders, the initial number (INIT) and the increment number (INC) are parameterized during the runtime by the size. For the size 2^6 points, the INIT and INC are shown in Table A-19.

Table A-19 Increment and initial number for generating twiddle fractions of optimal algorithm using adders

Stage	INIT(i)	INC(i)
1	$(00000000)_2$	$(00000000)_2 = \frac{0}{64}$
2	$(00000000)_2$	$(01000000)_2 = \frac{16}{64}$
3	$(00000000)_2$	$(00100000)_2 = \frac{8}{64}$
4	$(00000000)_2$	$(00010000)_2 = \frac{4}{64}$
5	$(0p_0000000)_2$	$(00001000)_2 = \frac{2}{64}$
6	$(0p_0p_1000000)_2$	$(00000100)_2 = \frac{1}{64}$

Note that the increment is done in every 2 addresses. Table A-20 shows the generation of TF1 during any stages in each PE.

Table A-20 Generation of twiddle fractions at stage 6 before masking (TF1) following optimal algorithm of size 64 points using adders

INIT = $(0p_0p_1000000)$ and INC = $2^{8-6} = (00000100)$				
$a_3a_2a_1$	Twiddle Fraction (TF1)			
	PE0	PE1	PE2	PE3
	$p_1p_0 = 00$	$p_1p_0 = 01$	$p_1p_0 = 10$	$p_1p_0 = 11$
0	$(00000000) = \frac{0}{64}$	$(01000000) = \frac{16}{64}$	$(00100000) = \frac{8}{64}$	$(01100000) = \frac{24}{64}$
1	$(00000100) = \frac{1}{64}$	$(01000100) = \frac{17}{64}$	$(00100100) = \frac{9}{64}$	$(01100100) = \frac{25}{64}$
2	$(00001000) = \frac{2}{64}$	$(01001000) = \frac{18}{64}$	$(00101000) = \frac{10}{64}$	$(01101000) = \frac{26}{64}$
3	$(00001100) = \frac{3}{64}$	$(01001100) = \frac{19}{64}$	$(00101100) = \frac{11}{64}$	$(01101100) = \frac{27}{64}$
4	$(00010000) = \frac{4}{64}$	$(01010000) = \frac{20}{64}$	$(00110000) = \frac{12}{64}$	$(01110000) = \frac{28}{64}$
5	$(00010100) = \frac{5}{64}$	$(01010100) = \frac{21}{64}$	$(00110100) = \frac{13}{64}$	$(01110100) = \frac{29}{64}$
6	$(00011000) = \frac{6}{64}$	$(01011000) = \frac{22}{64}$	$(00111000) = \frac{14}{64}$	$(01111000) = \frac{30}{64}$
7	$(00011100) = \frac{7}{64}$	$(01011100) = \frac{23}{64}$	$(00111100) = \frac{15}{64}$	$(01111100) = \frac{31}{64}$

A.3 Tables for 64-point Pease Algorithm

Table A-21 Addresses sequences for 64-bit Pease algorithm at stage 1, 2 and 3

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3b_2b_1b_0$		$b_4b_3b_2b_1b_0b_5$		$b_3b_2b_1b_0b_5b_4$	
000000	0	000000	0	000000	0	000000	0
000001	0	000001	1	000010	2	000100	4
000010	1	000010	2	000100	4	001000	8
000011	1	000011	3	000110	6	001100	12
000100	2	000100	4	001000	8	010000	16
000101	2	000101	5	001010	10	010100	20
000110	3	000110	6	001100	12	011000	24
000111	3	000111	7	001110	14	011100	28
001000	0	001000	8	010000	16	100000	32
001001	0	001001	9	010010	18	100100	36
001010	1	001010	10	010100	20	101000	40
001011	1	001011	11	010110	22	101100	44
001100	2	001100	12	011000	24	110000	48
001101	2	001101	13	011010	26	110100	52
001110	3	001110	14	011100	28	111000	56
001111	3	001111	15	011110	30	111100	60
010000	0	010000	16	100000	32	000001	1
010001	0	010001	17	100010	34	000101	5
010010	1	010010	18	100100	36	001001	9
010011	1	010011	19	100110	38	001101	13
010100	2	010100	20	101000	40	010001	17
010101	2	010101	21	101010	42	010101	21
010110	3	010110	22	101100	44	011001	25
010111	3	010111	23	101110	46	011101	29
011000	0	011000	24	110000	48	100001	33
011001	0	011001	25	110010	50	100101	37
011010	1	011010	26	110100	52	101001	41
011011	1	011011	27	110110	54	101101	45
011100	2	011100	28	111000	56	110001	49
011101	2	011101	29	111010	58	110101	53
011110	3	011110	30	111100	60	111001	57
011111	3	011111	31	111110	62	111101	61

Table A-22 Addresses sequences for 64-bit Pease algorithm at stage 1, 2 and 3 (cont.)

Counter (l)	Mapped	Addresses (k)					
	to PE	Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3b_2b_1b_0$		$b_4b_3b_2b_1b_0b_5$		$b_3b_2b_1b_0b_5b_4$	
100000	0	100000	32	000001	1	000010	2
100001	0	100001	33	000011	3	000110	6
100010	1	100010	34	000101	5	001010	10
100011	1	100011	35	000111	7	001110	14
100100	2	100100	36	001001	9	010010	18
100101	2	100101	37	001011	11	010110	22
100110	3	100110	38	001101	13	011010	26
100111	3	100111	39	001111	15	011110	30
101000	0	101000	40	010001	17	100010	34
101001	0	101001	41	010011	19	100110	38
101010	1	101010	42	010101	21	101010	42
101011	1	101011	43	010111	23	101110	46
101100	2	101100	44	011001	25	110010	50
101101	2	101101	45	011011	27	110110	54
101110	3	101110	46	011101	29	111010	58
101111	3	101111	47	011111	31	111110	62
110000	0	110000	48	100001	33	000011	3
110001	0	110001	49	100011	35	000111	7
110010	1	110010	50	100101	37	001011	11
110011	1	110011	51	100111	39	001111	15
110100	2	110100	52	101001	41	010011	19
110101	2	110101	53	101011	43	010111	23
110110	3	110110	54	101101	45	011011	27
110111	3	110111	55	101111	47	011111	31
111000	0	111000	56	110001	49	100011	35
111001	0	111001	57	110011	51	100111	39
111010	1	111010	58	110101	53	101011	43
111011	1	111011	59	110111	55	101111	47
111100	2	111100	60	111001	57	110011	51
111101	2	111101	61	111011	59	110111	55
111110	3	111110	62	111101	61	111011	59
111111	3	111111	63	111111	63	111111	63

Table A-23 Addresses sequences for 64-bit Pease algorithm at stage 4, 5 and 6

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_1b_0b_5b_4b_3b_2$		$b_0b_5b_4b_3b_2b_1$	
000000	0	000000	0	000000	0	000000	0
000001	0	001000	8	010000	16	100000	32
000010	1	010000	16	100000	32	000001	1
000011	1	011000	24	110000	48	100001	33
000100	2	100000	32	000001	1	000010	2
000101	2	101000	40	010001	17	100010	34
000110	3	110000	48	100001	33	000011	3
000111	3	111000	56	110001	49	100011	35
001000	0	000001	1	000010	2	000100	4
001001	0	001001	9	010010	18	100100	36
001010	1	010001	17	100010	34	000101	5
001011	1	011001	25	110010	50	100101	37
001100	2	100001	33	000011	3	000110	6
001101	2	101001	41	010011	19	100110	38
001110	3	110001	49	100011	35	000111	7
001111	3	111001	57	110011	51	100111	39
010000	0	000010	2	000100	4	001000	8
010001	0	001010	10	010100	20	101000	40
010010	1	010010	18	100100	36	001001	9
010011	1	011010	26	110100	52	101001	41
010100	2	100010	34	000101	5	001010	10
010101	2	101010	42	010101	21	101010	42
010110	3	110010	50	100101	37	001011	11
010111	3	111010	58	110101	53	101011	43
011000	0	000011	3	000110	6	001100	12
011001	0	001011	11	010110	22	101100	44
011010	1	010011	19	100110	38	001101	13
011011	1	011011	27	110110	54	101101	45
011100	2	100011	35	000111	7	001110	14
011101	2	101011	43	010111	23	101110	46
011110	3	110011	51	100111	39	001111	15
011111	3	111011	59	110111	55	101111	47

Table A-24 Addresses sequences for 64-bit Pease algorithm at stage 4, 5 and 6 (cont.)

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_1b_0b_5b_4b_3b_2$		$b_0b_5b_4b_3b_2b_1$	
100000	0	000100	4	001000	8	010000	16
100001	0	001100	12	011000	24	110000	48
100010	1	010100	20	101000	40	010001	17
100011	1	011100	28	111000	56	110001	49
100100	2	100100	36	001001	9	010010	18
100101	2	101100	44	011001	25	110010	50
100110	3	110100	52	101001	41	010011	19
100111	3	111100	60	111001	57	110011	51
101000	0	000101	5	001010	10	010100	20
101001	0	001101	13	011010	26	110100	52
101010	1	010101	21	101010	42	010101	21
101011	1	011101	29	111010	58	110101	53
101100	2	100101	37	001011	11	010110	22
101101	2	101101	45	011011	27	110110	54
101110	3	110101	53	101011	43	010111	23
101111	3	111101	61	111011	59	110111	55
110000	0	000110	6	001100	12	011000	24
110001	0	001110	14	011100	28	111000	56
110010	1	010110	22	101100	44	011001	25
110011	1	011110	30	111100	60	111001	57
110100	2	100110	38	001101	13	011010	26
110101	2	101110	46	011101	29	111010	58
110110	3	110110	54	101101	45	011011	27
110111	3	111110	62	111101	61	111011	59
111000	0	000111	7	001110	14	011100	28
111001	0	001111	15	011110	30	111100	60
111010	1	010111	23	101110	46	011101	29
111011	1	011111	31	111110	62	111101	61
111100	2	100111	39	001111	15	011110	30
111101	2	101111	47	011111	31	111110	62
111110	3	110111	55	101111	47	011111	31
111111	3	111111	63	111111	63	111111	63

Table A-25 Addresses mapped to PE0 and PE1 for 64-bit Pease algorithm at stage 1, 2 and 3

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3b_2b_1b_0$		$b_4b_3b_2b_1b_0b_5$		$b_3b_2b_1b_0b_5b_4$	
000000	0	000000	0	000000	0	000000	0
000001	0	000001	1	000010	2	000100	4
001000	0	001000	8	010000	16	100000	32
001001	0	001001	9	010010	18	100100	36
010000	0	010000	16	100000	32	000001	1
010001	0	010001	17	100010	34	000101	5
011000	0	011000	24	110000	48	100001	33
011001	0	011001	25	110010	50	100101	37
100000	0	100000	32	000001	1	000010	2
100001	0	100001	33	000011	3	000110	6
101000	0	101000	40	010001	17	100010	34
101001	0	101001	41	010011	19	100110	38
110000	0	110000	48	100001	33	000011	3
110001	0	110001	49	100011	35	000111	7
111000	0	111000	56	110001	49	100011	35
111001	0	111001	57	110011	51	100111	39
000010	1	000010	2	000100	4	001000	8
000011	1	000011	3	000110	6	001100	12
001010	1	001010	10	010100	20	101000	40
001011	1	001011	11	010110	22	101100	44
010010	1	010010	18	100100	36	001001	9
010011	1	010011	19	100110	38	001101	13
011010	1	011010	26	110100	52	101001	41
011011	1	011011	27	110110	54	101101	45
100010	1	100010	34	000101	5	001010	10
100011	1	100011	35	000111	7	001110	14
101010	1	101010	42	010101	21	101010	42
101011	1	101011	43	010111	23	101110	46
110010	1	110010	50	100101	37	001011	11
110011	1	110011	51	100111	39	001111	15
111010	1	111010	58	110101	53	101011	43
111011	1	111011	59	110111	55	101111	47

Table A-26 Addresses mapped to PE0 and PE1 for 64-bit Pease algorithm at stage 4, 5 and 6

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_1b_0b_5b_4b_3b_2$		$b_0b_5b_4b_3b_2b_1$	
000000	0	000000	0	000000	0	000000	0
000001	0	001000	8	010000	16	100000	32
001000	0	000001	1	000010	2	000100	4
001001	0	001001	9	010010	18	100100	36
010000	0	000010	2	000100	4	001000	8
010001	0	001010	10	010100	20	101000	40
011000	0	000011	3	000110	6	001100	12
011001	0	001011	11	010110	22	101100	44
100000	0	000100	4	001000	8	010000	16
100001	0	001100	12	011000	24	110000	48
101000	0	000101	5	001010	10	010100	20
101001	0	001101	13	011010	26	110100	52
110000	0	000110	6	001100	12	011000	24
110001	0	001110	14	011100	28	111000	56
111000	0	000111	7	001110	14	011100	28
111001	0	001111	15	011110	30	111100	60
000010	1	010000	16	100000	32	000001	1
000011	1	011000	24	110000	48	100001	33
001010	1	010001	17	100010	34	000101	5
001011	1	011001	25	110010	50	100101	37
010010	1	010010	18	100100	36	001001	9
010011	1	011010	26	110100	52	101001	41
011010	1	010011	19	100110	38	001101	13
011011	1	011011	27	110110	54	101101	45
100010	1	010100	20	101000	40	010001	17
100011	1	011100	28	111000	56	110001	49
101010	1	010101	21	101010	42	010101	21
101011	1	011101	29	111010	58	110101	53
110010	1	010110	22	101100	44	011001	25
110011	1	011110	30	111100	60	111001	57
111010	1	010111	23	101110	46	011101	29
111011	1	011111	31	111110	62	111101	61

Table A-27 Addresses mapped to PE2 and PE3 for 64-bit Pease algorithm at stage 1, 2 and 3

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3b_2b_1b_0$		$b_4b_3b_2b_1b_0b_5$		$b_3b_2b_1b_0b_5b_4$	
000100	2	000100	4	001000	8	010000	16
000101	2	000101	5	001010	10	010100	20
001100	2	001100	12	011000	24	110000	48
001101	2	001101	13	011010	26	110100	52
010100	2	010100	20	101000	40	010001	17
010101	2	010101	21	101010	42	010101	21
011100	2	011100	28	111000	56	110001	49
011101	2	011101	29	111010	58	110101	53
100100	2	100100	36	001001	9	010010	18
100101	2	100101	37	001011	11	010110	22
101100	2	101100	44	011001	25	110010	50
101101	2	101101	45	011011	27	110110	54
110100	2	110100	52	101001	41	010011	19
110101	2	110101	53	101011	43	010111	23
111100	2	111100	60	111001	57	110011	51
111101	2	111101	61	111011	59	110111	55
000110	3	000110	6	001100	12	011000	24
000111	3	000111	7	001110	14	011100	28
001110	3	001110	14	011100	28	111000	56
001111	3	001111	15	011110	30	111100	60
010110	3	010110	22	101100	44	011001	25
010111	3	010111	23	101110	46	011101	29
011110	3	011110	30	111100	60	111001	57
011111	3	011111	31	111110	62	111101	61
100110	3	100110	38	001101	13	011010	26
100111	3	100111	39	001111	15	011110	30
101110	3	101110	46	011101	29	111010	58
101111	3	101111	47	011111	31	111110	62
110110	3	110110	54	101101	45	011011	27
110111	3	110111	55	101111	47	011111	31
111110	3	111110	62	111101	61	111011	59
111111	3	111111	63	111111	63	111111	63

Table A-28 Addresses mapped to PE2 and PE3 for 64-bit Pease algorithm at stage 4, 5 and 6

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3b_2b_1b_0$		$b_4b_3b_2b_1b_0b_5$		$b_3b_2b_1b_0b_5b_4$	
000100	2	100000	32	000001	1	000010	2
000101	2	101000	40	010001	17	100010	34
001100	2	100001	33	000011	3	000110	6
001101	2	101001	41	010011	19	100110	38
010100	2	100010	34	000101	5	001010	10
010101	2	101010	42	010101	21	101010	42
011100	2	100011	35	000111	7	001110	14
011101	2	101011	43	010111	23	101110	46
100100	2	100100	36	001001	9	010010	18
100101	2	101100	44	011001	25	110010	50
101100	2	100101	37	001011	11	010110	22
101101	2	101101	45	011011	27	110110	54
110100	2	100110	38	001101	13	011010	26
110101	2	101110	46	011101	29	111010	58
111100	2	100111	39	001111	15	011110	30
111101	2	101111	47	011111	31	111110	62
000110	3	110000	48	100001	33	000011	3
000111	3	111000	56	110001	49	100011	35
001110	3	110001	49	100011	35	000111	7
001111	3	111001	57	110011	51	100111	39
010110	3	110010	50	100101	37	001011	11
010111	3	111010	58	110101	53	101011	43
011110	3	110011	51	100111	39	001111	15
011111	3	111011	59	110111	55	101111	47
100110	3	110100	52	101001	41	010011	19
100111	3	111100	60	111001	57	110011	51
101110	3	110101	53	101011	43	010111	23
101111	3	111101	61	111011	59	110111	55
110110	3	110110	54	101101	45	011011	27
110111	3	111110	62	111101	61	111011	59
111110	3	110111	55	101111	47	011111	31
111111	3	111111	63	111111	63	111111	63

Table A-29 Source MID, target PID and local addresses generated in PE0 and PE1 at stage 1 and 2 following the Pease algorithm

Counter	Stage 1				Stage 2			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	a_3a_2	a_2a_1	$a_3a_2a_1a_0$		a_2a_1	a_2a_1	$a_2a_1a_0a_3$	
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0001	1	0	0	0010	2
0010	0	1	0010	2	1	1	0100	4
0011	0	1	0011	3	1	1	0110	6
0100	1	2	0100	4	2	2	1000	8
0101	1	2	0101	5	2	2	1010	10
0110	1	3	0110	6	3	3	1100	12
0111	1	3	0111	7	3	3	1110	14
1000	2	0	1000	8	0	0	0001	1
1001	2	0	1001	9	0	0	0011	3
1010	2	1	1010	10	1	1	0101	5
1011	2	1	1011	11	1	1	0111	7
1100	3	2	1100	12	2	2	1001	9
1101	3	2	1101	13	2	2	1011	11
1110	3	3	1110	14	3	3	1101	13
1111	3	3	1111	15	3	3	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0001	1	0	0	0010	2
0010	0	1	0010	2	1	1	0100	4
0011	0	1	0011	3	1	1	0110	6
0100	1	2	0100	4	2	2	1000	8
0101	1	2	0101	5	2	2	1010	10
0110	1	3	0110	6	3	3	1100	12
0111	1	3	0111	7	3	3	1110	14
1000	2	0	1000	8	0	0	0001	1
1001	2	0	1001	9	0	0	0011	3
1010	2	1	1010	10	1	1	0101	5
1011	2	1	1011	11	1	1	0111	7
1100	3	2	1100	12	2	2	1001	9
1101	3	2	1101	13	2	2	1011	11
1110	3	3	1110	14	3	3	1101	13
1111	3	3	1111	15	3	3	1111	15

Table A-30 Source MID, target PID and local addresses generated in PE0 and PE1 at stage 3 and 4 following the Pease algorithm

Counter	Stage 3				Stage 4			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	a_1p_1	p_0a_1	$a_1a_0a_3a_2$		p_1p_0	p_1p_0	$a_0a_3a_2a_1$	
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0100	4	0	0	1000	8
0010	2	1	1000	8	0	0	0001	1
0011	2	1	1100	12	0	0	1001	9
0100	0	0	0001	1	0	0	0010	2
0101	0	0	0101	5	0	0	1010	10
0110	2	1	1001	9	0	0	0011	3
0111	2	1	1101	13	0	0	1011	11
1000	0	0	0010	2	0	0	0100	4
1001	0	0	0110	6	0	0	1100	12
1010	2	1	1010	10	0	0	0101	5
1011	2	1	1110	14	0	0	1101	13
1100	0	0	011	3	0	0	0110	6
1101	0	0	0111	7	0	0	1110	14
1110	2	1	1011	11	0	0	0111	7
1111	2	1	1111	15	0	0	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	0	2	0000	0	1	1	0000	0
0001	0	2	0100	4	1	1	1000	8
0010	2	3	1000	8	1	1	0001	1
0011	2	3	1100	12	1	1	1001	9
0100	0	2	0001	1	1	1	0010	2
0101	0	2	0101	5	1	1	1010	10
0110	2	3	1001	9	1	1	0011	3
0111	2	3	1101	13	1	1	1011	11
1000	0	2	0010	2	1	1	0100	4
1001	0	2	0110	6	1	1	1100	12
1010	2	3	1010	10	1	1	0101	5
1011	2	3	1110	14	1	1	1101	13
1100	0	2	011	3	1	1	0110	6
1101	0	2	0111	7	1	1	1110	14
1110	2	3	1011	11	1	1	0111	7
1111	2	3	1111	15	1	1	1111	15

Table A-31 Source MID, target PID and local addresses generaged at stage 5 and 6 in PE0 and PE1 following the Pease algorithm

Counter	Stage 5				Stage 6			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_0a_0	a_0p_1	$a_1a_0a_3a_2$		a_0a_3	a_1a_0	$a_3a_2a_1a_0$	
0000	0	0	0000	0	0	0	0000	0
0001	1	2	0001	1	2	1	0001	1
0010	0	0	0010	2	0	2	0010	2
0011	1	2	0011	3	2	3	0011	3
0100	0	0	0100	4	0	0	0100	4
0101	1	2	0101	5	2	1	0101	5
0110	0	0	0110	6	0	2	0110	6
0111	1	2	0111	7	2	3	0111	7
1000	0	0	1000	8	1	0	1000	8
1001	1	2	1001	9	3	1	1001	9
1010	0	0	1010	10	1	2	1010	10
1011	1	2	1011	11	3	3	1011	11
1100	0	0	1100	12	1	0	1100	12
1101	1	2	1101	13	3	1	1101	13
1110	0	0	1110	14	1	2	1110	14
1111	1	2	1111	15	3	3	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	2	0	0000	0	0	0	0000	0
0001	3	2	0001	1	2	1	0001	1
0010	2	0	0010	2	0	2	0010	2
0011	3	2	0011	3	2	3	0011	3
0100	2	0	0100	4	0	0	0100	4
0101	3	2	0101	5	2	1	0101	5
0110	2	0	0110	6	0	2	0110	6
0111	3	2	0111	7	2	3	0111	7
1000	2	0	1000	8	1	0	1000	8
1001	3	2	1001	9	3	1	1001	9
1010	2	0	1010	10	1	2	1010	10
1011	3	2	1011	11	3	3	1011	11
1100	2	0	1100	12	1	0	1100	12
1101	3	2	1101	13	3	1	1101	13
1110	2	0	1110	14	1	2	1110	14
1111	3	2	1111	15	3	3	1111	15

Table A-32 Source MID, target PID and local addresses generated in PE2 and PE3 at stage 1 and 2 following Pease algorithm

Counter	Stage 1				Stage 2			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	a_3a_2	a_2a_1	$a_3a_2a_1a_0$		a_2a_1	a_2a_1	$a_2a_1a_0a_3$	
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0001	1	0	0	0010	2
0010	0	1	0010	2	1	1	0100	4
0011	0	1	0011	3	1	1	0110	6
0100	1	2	0100	4	2	2	1000	8
0101	1	2	0101	5	2	2	1010	10
0110	1	3	0110	6	3	3	1100	12
0111	1	3	0111	7	3	3	1110	14
1000	2	0	1000	8	0	0	0001	1
1001	2	0	1001	9	0	0	0011	3
1010	2	1	1010	10	1	1	0101	5
1011	2	1	1011	11	1	1	0111	7
1100	3	2	1100	12	2	2	1001	9
1101	3	2	1101	13	2	2	1011	11
1110	3	3	1110	14	3	3	1101	13
1111	3	3	1111	15	3	3	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0001	1	0	0	0010	2
0010	0	1	0010	2	1	1	0100	4
0011	0	1	0011	3	1	1	0110	6
0100	1	2	0100	4	2	2	1000	8
0101	1	2	0101	5	2	2	1010	10
0110	1	3	0110	6	3	3	1100	12
0111	1	3	0111	7	3	3	1110	14
1000	2	0	1000	8	0	0	0001	1
1001	2	0	1001	9	0	0	0011	3
1010	2	1	1010	10	1	1	0101	5
1011	2	1	1011	11	1	1	0111	7
1100	3	2	1100	12	2	2	1001	9
1101	3	2	1101	13	2	2	1011	11
1110	3	3	1110	14	3	3	1101	13
1111	3	3	1111	15	3	3	1111	15

Table A-33 Source MID, target PID and local addresses generated in PE2 and PE3 at stage 3 and 4 following Pease algorithm

Counter	Stage 3				Stage 4			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	a_1p_1	p_0a_1	$a_1a_0a_3a_2$		p_1p_0	p_1p_0	$a_0a_3a_2a_1$	
0000	1	0	0000	0	2	2	0000	0
0001	1	0	0100	4	2	2	1000	8
0010	3	1	1000	8	2	2	0001	1
0011	3	1	1100	12	2	2	1001	9
0100	1	0	0001	1	2	2	0010	2
0101	1	0	0101	5	2	2	1010	10
0110	3	1	1001	9	2	2	0011	3
0111	3	1	1101	13	2	2	1011	11
1000	1	0	0010	2	2	2	0100	4
1001	1	0	0110	6	2	2	1100	12
1010	3	1	1010	10	2	2	0101	5
1011	3	1	1110	14	2	2	1101	13
1100	1	0	0011	3	2	2	0110	6
1101	1	0	0111	7	2	2	1110	14
1110	3	1	1011	11	2	2	0111	7
1111	3	1	1111	15	2	2	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	1	2	0000	0	3	3	0000	0
0001	1	2	0100	4	3	3	1000	8
0010	3	3	1000	8	3	3	0001	1
0011	3	3	1100	12	3	3	1001	9
0100	1	2	0001	1	3	3	0010	2
0101	1	2	0101	5	3	3	1010	10
0110	3	3	1001	9	3	3	0011	3
0111	3	3	1101	13	3	3	1011	11
1000	1	2	0010	2	3	3	0100	4
1001	1	2	0110	6	3	3	1100	12
1010	3	3	1010	10	3	3	0101	5
1011	3	3	1110	14	3	3	1101	13
1100	1	2	011	3	3	3	0110	6
1101	1	2	0111	7	3	3	1110	14
1110	3	3	1011	11	3	3	0111	7
1111	3	3	1111	15	3	3	1111	15

Table A-34 Source MID, target PID and local addresses at stage 5 and 6 in PE2 and PE3 following Pease algorithm

Counter	Stage 5				Stage 6			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_0a_0	a_0p_1	$a_1a_0a_3a_2$		a_0a_3	a_1a_0	$a_3a_2a_1a_0$	
0000	0	1	0000	0	0	0	0000	0
0001	1	3	0001	1	2	1	0001	1
0010	0	1	0010	2	0	2	0010	2
0011	1	3	0011	3	2	3	0011	3
0100	0	1	0100	4	0	0	0100	4
0101	1	3	0101	5	2	1	0101	5
0110	0	1	0110	6	0	2	0110	6
0111	1	3	0111	7	2	3	0111	7
1000	0	1	1000	8	1	0	1000	8
1001	1	3	1001	9	3	1	1001	9
1010	0	1	1010	10	1	2	1010	10
1011	1	3	1011	11	3	3	1011	11
1100	0	1	1100	12	1	0	1100	12
1101	1	3	1101	13	3	1	1101	13
1110	0	1	1110	14	1	2	1110	14
1111	1	3	1111	15	3	3	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	2	1	0000	0	0	0	0000	0
0001	3	3	0001	2	1	1	0001	1
0010	2	1	0010	0	2	2	0010	2
0011	3	3	0011	2	3	3	0011	3
0100	2	1	0100	0	4	0	0100	4
0101	3	3	0101	2	5	1	0101	5
0110	2	1	0110	0	6	2	0110	6
0111	3	3	0111	2	7	3	0111	7
1000	2	1	1000	0	8	0	1000	8
1001	3	3	1001	2	9	1	1001	9
1010	2	1	1010	0	10	2	1010	10
1011	3	3	1011	2	11	3	1011	11
1100	2	1	1100	0	12	0	1100	12
1101	3	3	1101	2	13	1	1101	13
1110	2	1	1110	0	14	2	1110	14
1111	3	3	1111	2	15	3	1111	15

Table A-35 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm at stage 1 to 3

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{r}{N}$	b_5	$\frac{r}{N}$	b_5b_4	$\frac{r}{N}$
000000	0	0	0	0	0	00	0
000001	0	0	0	0	0	00	0
000010	1	0	0	0	0	00	0
000011	1	0	0	0	0	00	0
000100	2	0	0	0	0	00	0
000101	2	0	0	0	0	00	0
000110	3	0	0	0	0	00	0
000111	3	0	0	0	0	00	0
001000	0	0	0	0	0	00	0
001001	0	0	0	0	0	00	0
001010	1	0	0	0	0	00	0
001011	1	0	0	0	0	00	0
001100	2	0	0	0	0	00	0
001101	2	0	0	0	0	00	0
001110	3	0	0	0	0	00	0
001111	3	0	0	0	0	00	0
010000	0	0	0	0	0	01	0
010001	0	0	0	0	0	01	$\frac{1}{8}$
010010	1	0	0	0	0	01	0
010011	1	0	0	0	0	01	$\frac{1}{8}$
010100	2	0	0	0	0	01	0
010101	2	0	0	0	0	01	$\frac{1}{8}$
010110	3	0	0	0	0	01	0
010111	3	0	0	0	0	01	$\frac{1}{8}$
011000	0	0	0	0	0	01	0
011001	0	0	0	0	0	01	$\frac{1}{8}$
011010	1	0	0	0	0	01	0
011011	1	0	0	0	0	01	$\frac{1}{8}$
011100	2	0	0	0	0	01	0
011101	2	0	0	0	0	01	$\frac{1}{8}$
011110	3	0	0	0	0	01	0
011111	3	0	0	0	0	01	$\frac{1}{8}$

Table A-36 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm at stage 1 to 3 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{r}{N}$	b_5	$\frac{r}{N}$	b_5b_4	$\frac{r}{N}$
100000	0	0	0	1	0	10	0
100001	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
100010	1	0	0	1	0	10	0
100011	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
100100	2	0	0	1	0	10	0
100101	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
100110	3	0	0	1	0	10	0
100111	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
101000	0	0	0	1	0	10	0
101001	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
101010	1	0	0	1	0	10	0
101011	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
101100	2	0	0	1	0	10	0
101101	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
101110	3	0	0	1	0	10	0
101111	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
110000	0	0	0	1	0	11	0
110001	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
110010	1	0	0	1	0	11	0
110011	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
110100	2	0	0	1	0	11	0
110101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
110110	3	0	0	1	0	11	0
110111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111000	0	0	0	1	0	11	0
111001	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111010	1	0	0	1	0	11	0
111011	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111100	2	0	0	1	0	11	0
111101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111110	3	0	0	1	0	11	0
111111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-37 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm at stage 4 to 6

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3$	$\frac{r}{N}$	$b_5b_4b_3b_2$	$\frac{r}{N}$	$b_5b_4b_3b_2b_1$	$\frac{r}{N}$
000000	0	000	0	0000	0	00000	0
000001	0	000	0	0000	0	00000	0
000010	1	000	0	0000	0	00001	0
000011	1	000	0	0000	0	00001	$\frac{1}{64}$
000100	2	000	0	0001	0	00010	0
000101	2	000	0	0001	$\frac{1}{32}$	00010	$\frac{2}{64}$
000110	3	000	0	0001	0	00011	0
000111	3	000	0	0001	$\frac{1}{32}$	00011	$\frac{3}{64}$
001000	0	001	0	0010	0	00100	0
001001	0	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00100	$\frac{4}{64}$
001010	1	001	0	0010	0	00101	0
001011	1	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00101	$\frac{5}{64}$
001100	2	001	0	0011	0	00110	0
001101	2	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00110	$\frac{6}{64}$
001110	3	001	0	0011	0	00111	0
001111	3	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00111	$\frac{7}{64}$
010000	0	010	0	0100	0	01000	0
010001	0	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01000	$\frac{8}{64}$
010010	1	010	0	0100	0	01001	0
010011	1	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01001	$\frac{9}{64}$
010100	2	010	0	0101	0	01010	0
010101	2	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01010	$\frac{10}{64}$
010110	3	010	0	0101	0	01011	0
010111	3	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01011	$\frac{11}{64}$
011000	0	011	0	0110	0	01100	0
011001	0	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01100	$\frac{12}{64}$
011010	1	011	0	0110	0	01101	0
011011	1	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01101	$\frac{13}{64}$
011100	2	011	0	0111	0	01110	0
011101	2	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01110	$\frac{14}{64}$
011110	3	011	0	0111	0	01111	0
011111	3	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01111	$\frac{15}{64}$

Table A-38 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm at stage 4 to 6 (cont.)

Counter	Mapped	Twiddle Fractions ($\frac{r}{N}$)					
	to	Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_10b_0$	b_2b_1	$b_5b_4b_3$	$\frac{r}{N}$	$b_5b_4b_3b_2$	$\frac{r}{N}$	$b_5b_4b_3b_2b_1$	$\frac{r}{N}$
100000	0	100	0	1000	0	10000	0
100001	0	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10000	$\frac{16}{64}$
100010	1	100	0	1000	0	10001	0
100011	1	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10001	$\frac{17}{64}$
100100	2	100	0	1001	0	10010	0
100101	2	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10010	$\frac{18}{64}$
100110	3	100	0	1001	0	10011	0
100111	3	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10011	$\frac{19}{64}$
101000	0	101	0	1010	0	10100	0
101001	0	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10100	$\frac{20}{64}$
101010	1	101	0	1010	0	10101	0
101011	1	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10101	$\frac{21}{64}$
101100	2	101	0	1011	0	10110	0
101101	2	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10110	$\frac{22}{64}$
101110	3	101	0	1011	0	10111	0
101111	3	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10111	$\frac{23}{64}$
110000	0	110	0	1100	0	11000	0
110001	0	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11000	$\frac{24}{64}$
110010	1	110	0	1100	0	11001	0
110011	1	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11001	$\frac{25}{64}$
110100	2	110	0	1101	0	11010	0
110101	2	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11010	$\frac{26}{64}$
110110	3	110	0	1101	0	11011	0
110111	3	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11011	$\frac{27}{64}$
111000	0	111	0	1110	0	11100	0
111001	0	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11100	$\frac{28}{64}$
111010	1	111	0	1110	0	11101	0
111011	1	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11101	$\frac{29}{64}$
111100	2	111	0	1111	0	11110	0
111101	2	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11110	$\frac{30}{64}$
111110	3	111	0	1111	0	11111	0
111111	3	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11111	$\frac{31}{64}$

Table A-39 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm mapped to PE0 and PE1 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_3b_2b_1b_0$	p_1p_0	0	$\frac{r}{N}$	b_3	$\frac{r}{N}$	b_3b_2	$\frac{r}{N}$
0000	0	0	0	0	0	00	0
0001	0	0	0	0	0	00	0
0010	0	0	0	0	0	00	0
0011	0	0	0	0	0	00	0
0100	0	0	0	0	0	01	0
0101	0	0	0	0	0	01	$\frac{1}{8}$
0110	0	0	0	0	0	01	0
0111	0	0	0	0	0	01	$\frac{1}{8}$
1000	0	0	0	1	0	10	0
1001	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	0	0	0	1	0	10	0
1011	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	0	0	0	1	0	11	0
1101	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	0	0	0	1	0	11	0
1111	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	1	0	0	0	0	00	0
0001	1	0	0	0	0	00	0
0010	1	0	0	0	0	00	0
0011	1	0	0	0	0	00	0
0100	1	0	0	0	0	01	0
0101	1	0	0	0	0	01	$\frac{1}{8}$
0110	1	0	0	0	0	01	0
0111	1	0	0	0	0	01	$\frac{1}{8}$
1000	1	0	0	1	0	10	0
1001	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	1	0	0	1	0	10	0
1011	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	1	0	0	1	0	11	0
1101	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	1	0	0	1	0	11	0
1111	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-40 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm mapped to PE0 and PE1 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_3b_2b_1b_0$	p_1p_0	$b_3b_2b_1$	$\frac{r}{N}$	$b_3b_2b_1p_1$	$\frac{r}{N}$	$b_3b_2b_1p_1p_0$	$\frac{r}{N}$
0000	0	000	0	0000	0	00000	0
0001	0	000	0	0000	0	00000	0
0010	0	001	0	0010	0	00100	0
0011	0	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00100	$\frac{4}{64}$
0100	0	010	0	0100	0	01000	0
0101	0	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01000	$\frac{8}{64}$
0110	0	011	0	0110	0	01100	0
0111	0	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01100	$\frac{12}{64}$
1000	0	100	0	1000	0	10000	0
1001	0	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10000	$\frac{16}{64}$
1010	0	101	0	1010	0	10100	0
1011	0	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10100	$\frac{20}{64}$
1100	0	110	0	1100	0	11000	0
1101	0	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11000	$\frac{24}{64}$
1110	0	111	0	1110	0	11100	0
1111	0	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11100	$\frac{28}{64}$
0000	1	000	0	0000	0	00001	0
0001	1	000	0	0000	0	00001	$\frac{1}{64}$
0010	1	001	0	0010	0	00101	0
0011	1	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00101	$\frac{5}{64}$
0100	1	010	0	0100	0	01001	0
0101	1	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01001	$\frac{9}{64}$
0110	1	011	0	0110	0	01101	0
0111	1	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01101	$\frac{13}{64}$
1000	1	100	0	1000	0	10001	0
1001	1	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10001	$\frac{17}{64}$
1010	1	101	0	1010	0	10101	0
1011	1	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10101	$\frac{21}{64}$
1100	1	110	0	1100	0	11001	0
1101	1	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11001	$\frac{25}{64}$
1110	1	111	0	1110	0	11101	0
1111	1	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11101	$\frac{29}{64}$

Table A-41 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm mapped to PE2 and PE3 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_3b_2b_1b_0$	p_1p_0	0	$\frac{r}{N}$	b_3	$\frac{r}{N}$	b_3b_4	$\frac{r}{N}$
0000	2	0	0	0	0	00	0
0001	2	0	0	0	0	00	0
0010	2	0	0	0	0	00	0
0011	2	0	0	0	0	00	0
0100	2	0	0	0	0	01	0
0101	2	0	0	0	0	01	$\frac{1}{8}$
0110	2	0	0	0	0	01	0
0111	2	0	0	0	0	01	$\frac{1}{8}$
1000	2	0	0	1	0	10	0
1001	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	2	0	0	1	0	10	0
1011	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	2	0	0	1	0	11	0
1101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	2	0	0	1	0	11	0
1111	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	3	0	0	0	0	00	0
0001	3	0	0	0	0	00	0
0010	3	0	0	0	0	00	0
0011	3	0	0	0	0	00	0
0100	3	0	0	0	0	01	0
0101	3	0	0	0	0	01	$\frac{1}{8}$
0110	3	0	0	0	0	01	0
0111	3	0	0	0	0	01	$\frac{1}{8}$
1000	3	0	0	1	0	10	0
1001	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	3	0	0	1	0	10	0
1011	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	3	0	0	1	0	11	0
1101	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	3	0	0	1	0	11	0
1111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-42 Twiddle factors, represented by fractions, for 1-D 64-point FFT using Pease algorithm mapped to PE2 and PE3 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_3b_2b_10b_0$	p_1p_0	$b_3b_2b_1$	$\frac{r}{N}$	$b_3b_2b_1p_1$	$\frac{r}{N}$	$b_3b_2b_1p_1p_0$	$\frac{r}{N}$
0000	2	000	0	0001	0	00010	0
0001	2	000	0	0001	$\frac{1}{32}$	00010	$\frac{2}{64}$
0010	2	001	0	0011	0	00110	0
0011	2	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00110	$\frac{6}{64}$
0100	2	010	0	0101	0	01010	0
0101	2	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01010	$\frac{10}{64}$
0110	2	011	0	0111	0	01110	0
0111	2	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01110	$\frac{14}{64}$
1000	2	100	0	1001	0	10010	0
1001	2	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10010	$\frac{18}{64}$
1010	2	101	0	1011	0	10110	0
1011	2	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10110	$\frac{22}{64}$
1100	2	110	0	1101	0	11010	0
1101	2	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11010	$\frac{26}{64}$
1110	2	111	0	1111	0	11110	0
1111	2	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11110	$\frac{30}{64}$
0000	3	000	0	0001	0	00011	0
0001	3	000	0	0001	$\frac{1}{32}$	00011	$\frac{3}{64}$
0010	3	001	0	0011	0	00111	0
0011	3	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00111	$\frac{7}{64}$
0100	3	010	0	0101	0	01011	0
0101	3	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01011	$\frac{11}{64}$
0110	3	011	0	0111	0	01111	0
0111	3	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01111	$\frac{15}{64}$
1000	3	100	0	1001	0	10011	0
1001	3	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10011	$\frac{19}{64}$
1010	3	101	0	1011	0	10111	0
1011	3	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10111	$\frac{23}{64}$
1100	3	110	0	1101	0	11011	0
1101	3	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11011	$\frac{27}{64}$
1110	3	111	0	1111	0	11111	0
1111	3	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11111	$\frac{31}{64}$

Table A-43 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm at stage 1 to 3

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{r}{N}$	b_5	$\frac{r}{N}$	0	$\frac{r}{N}$
000000	0	0	0	0	0	0	0
000001	0	0	0	0	0	0	0
000010	1	0	0	0	0	0	0
000011	1	0	0	0	0	0	0
000100	2	0	0	0	0	0	0
000101	2	0	0	0	0	0	0
000110	3	0	0	0	0	0	0
000111	3	0	0	0	0	0	0
001000	0	0	0	0	0	0	0
001001	0	0	0	0	0	0	0
001010	1	0	0	0	0	0	0
001011	1	0	0	0	0	0	0
001100	2	0	0	0	0	0	0
001101	2	0	0	0	0	0	0
001110	3	0	0	0	0	0	0
001111	3	0	0	0	0	0	0
010000	0	0	0	0	0	0	0
010001	0	0	0	0	0	0	0
010010	1	0	0	0	0	0	0
010011	1	0	0	0	0	0	0
010100	2	0	0	0	0	0	0
010101	2	0	0	0	0	0	0
010110	3	0	0	0	0	0	0
010111	3	0	0	0	0	0	0
011000	0	0	0	0	0	0	0
011001	0	0	0	0	0	0	0
011010	1	0	0	0	0	0	0
011011	1	0	0	0	0	0	0
011100	2	0	0	0	0	0	0
011101	2	0	0	0	0	0	0
011110	3	0	0	0	0	0	0
011111	3	0	0	0	0	0	0

Table A-44 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm at stage 1 to 3 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1 0 b_0$	b_2b_1	0	$\frac{r}{N}$	b_5	$\frac{r}{N}$	0	$\frac{r}{N}$
100000	0	0	0	1	0	0	0
100001	0	0	0	1	$\frac{1}{4}$	0	0
100010	1	0	0	1	0	0	0
100011	1	0	0	1	$\frac{1}{4}$	0	0
100100	2	0	0	1	0	0	0
100101	2	0	0	1	$\frac{1}{4}$	0	0
100110	3	0	0	1	0	0	0
100111	3	0	0	1	$\frac{1}{4}$	0	0
101000	0	0	0	1	0	0	0
101001	0	0	0	1	$\frac{1}{4}$	0	0
101010	1	0	0	1	0	0	0
101011	1	0	0	1	$\frac{1}{4}$	0	0
101100	2	0	0	1	0	0	0
101101	2	0	0	1	$\frac{1}{4}$	0	0
101110	3	0	0	1	0	0	0
101111	3	0	0	1	$\frac{1}{4}$	0	0
110000	0	0	0	1	0	0	0
110001	0	0	0	1	$\frac{1}{4}$	0	0
110010	1	0	0	1	0	0	0
110011	1	0	0	1	$\frac{1}{4}$	0	0
110100	2	0	0	1	0	0	0
110101	2	0	0	1	$\frac{1}{4}$	0	0
110110	3	0	0	1	0	0	0
110111	3	0	0	1	$\frac{1}{4}$	0	0
111000	0	0	0	1	0	0	0
111001	0	0	0	1	$\frac{1}{4}$	0	0
111010	1	0	0	1	0	0	0
111011	1	0	0	1	$\frac{1}{4}$	0	0
111100	2	0	0	1	0	0	0
111101	2	0	0	1	$\frac{1}{4}$	0	0
111110	3	0	0	1	0	0	0
111111	3	0	0	1	$\frac{1}{4}$	0	0

Table A-45 Twiddle factors, represented by fractions, for 16×4-point FFT using Pease algorithm at stage 4 to 6

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	b_5	$\frac{r}{N}$	b_5b_4	$\frac{r}{N}$	$b_5b_4b_3$	$\frac{r}{N}$
000000	0	0	0	00	0	000	0
000001	0	0	0	00	0	000	0
000010	1	0	0	00	0	000	0
000011	1	0	0	00	0	000	0
000100	2	0	0	00	0	000	0
000101	2	0	0	00	0	000	0
000110	3	0	0	00	0	000	0
000111	3	0	0	00	0	000	0
001000	0	0	0	00	0	001	0
001001	0	0	0	00	0	001	$\frac{1}{16}$
001010	1	0	0	00	0	001	0
001011	1	0	0	00	0	001	$\frac{1}{16}$
001100	2	0	0	00	0	001	0
001101	2	0	0	00	0	001	$\frac{1}{16}$
001110	3	0	0	00	0	001	0
001111	3	0	0	00	0	001	$\frac{1}{16}$
010000	0	0	0	01	0	010	0
010001	0	0	0	01	$\frac{1}{8}$	010	$\frac{2}{16}$
010010	1	0	0	01	0	010	0
010011	1	0	0	01	$\frac{1}{8}$	010	$\frac{2}{16}$
010100	2	0	0	01	0	010	0
010101	2	0	0	01	$\frac{1}{8}$	010	$\frac{2}{16}$
010110	3	0	0	01	0	010	0
010111	3	0	0	01	$\frac{1}{8}$	010	$\frac{2}{16}$
011000	0	0	0	01	0	011	0
011001	0	0	0	01	$\frac{1}{8}$	011	$\frac{3}{16}$
011010	1	0	0	01	0	011	0
011011	1	0	0	01	$\frac{1}{8}$	011	$\frac{3}{16}$
011100	2	0	0	01	0	011	0
011101	2	0	0	01	$\frac{1}{8}$	011	$\frac{3}{16}$
011110	3	0	0	01	0	011	0
011111	3	0	0	01	$\frac{1}{8}$	011	$\frac{3}{16}$

Table A-46 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm at stage 4 to 6 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_10b_0$	b_2b_1	b_5	$\frac{r}{N}$	b_5b_4	$\frac{r}{N}$	$b_5b_4b_3$	$\frac{r}{N}$
100000	0	1	0	10	0	100	0
100001	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$	100	$\frac{4}{16}$
100010	1	1	0	10	0	100	0
100011	1	1	$\frac{1}{4}$	10	$\frac{2}{8}$	100	$\frac{4}{16}$
100100	2	1	0	10	0	100	0
100101	2	1	$\frac{1}{4}$	10	$\frac{2}{8}$	100	$\frac{4}{16}$
100110	3	1	0	10	0	100	0
100111	3	1	$\frac{1}{4}$	10	$\frac{2}{8}$	100	$\frac{4}{16}$
101000	0	1	0	10	0	101	0
101001	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$	101	$\frac{5}{16}$
101010	1	1	0	10	0	101	0
101011	1	1	$\frac{1}{4}$	10	$\frac{2}{8}$	101	$\frac{5}{16}$
101100	2	1	0	10	0	101	0
101101	2	1	$\frac{1}{4}$	10	$\frac{2}{8}$	101	$\frac{5}{16}$
101110	3	1	0	10	0	101	0
101111	3	1	$\frac{1}{4}$	10	$\frac{2}{8}$	101	$\frac{5}{16}$
110000	0	1	0	11	0	110	0
110001	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$	110	$\frac{6}{16}$
110010	1	1	0	11	0	110	0
110011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	110	$\frac{6}{16}$
110100	2	1	0	11	0	110	0
110101	2	1	$\frac{1}{4}$	11	$\frac{3}{8}$	110	$\frac{6}{16}$
110110	3	1	0	11	0	110	0
110111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	110	$\frac{6}{16}$
111000	0	1	0	11	0	111	0
111001	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
111010	1	1	0	11	0	111	0
111011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
111100	2	1	0	11	0	111	0
111101	2	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
111110	3	1	0	11	0	111	0
111111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$

Table A-47 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm mapped to PE0 and PE1 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_3b_2b_1b_0$	p_1p_0	0	$\frac{r}{N}$	b_3	$\frac{r}{N}$	b_3b_2	$\frac{r}{N}$
0000	0	0	0	0	0	00	0
0001	0	0	0	0	0	00	0
0010	0	0	0	0	0	00	0
0011	0	0	0	0	0	00	0
0100	0	0	0	0	0	01	0
0101	0	0	0	0	0	01	$\frac{1}{8}$
0110	0	0	0	0	0	01	0
0111	0	0	0	0	0	01	$\frac{1}{8}$
1000	0	0	0	1	0	10	0
1001	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	0	0	0	1	0	10	0
1011	0	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	0	0	0	1	0	11	0
1101	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	0	0	0	1	0	11	0
1111	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	1	0	0	0	0	00	0
0001	1	0	0	0	0	00	0
0010	1	0	0	0	0	00	0
0011	1	0	0	0	0	00	0
0100	1	0	0	0	0	01	0
0101	1	0	0	0	0	01	$\frac{1}{8}$
0110	1	0	0	0	0	01	0
0111	1	0	0	0	0	01	$\frac{1}{8}$
1000	1	0	0	1	0	10	0
1001	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	1	0	0	1	0	10	0
1011	1	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	1	0	0	1	0	11	0
1101	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	1	0	0	1	0	11	0
1111	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-48 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm mapped to PE0 and PE1 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_3b_2b_1b_0$	p_1p_0	$b_3b_2b_1$	$\frac{r}{N}$	$b_3b_2b_1p_1$	$\frac{r}{N}$	$b_3b_2b_1p_1p_0$	$\frac{r}{N}$
0000	0	000	0	0000	0	00000	0
0001	0	000	0	0000	0	00000	0
0010	0	001	0	0010	0	00100	0
0011	0	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00100	$\frac{4}{64}$
0100	0	010	0	0100	0	01000	0
0101	0	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01000	$\frac{8}{64}$
0110	0	011	0	0110	0	01100	0
0111	0	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01100	$\frac{12}{64}$
1000	0	100	0	1000	0	10000	0
1001	0	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10000	$\frac{16}{64}$
1010	0	101	0	1010	0	10100	0
1011	0	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10100	$\frac{20}{64}$
1100	0	110	0	1100	0	11000	0
1101	0	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11000	$\frac{24}{64}$
1110	0	111	0	1110	0	11100	0
1111	0	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11100	$\frac{28}{64}$
0000	1	000	0	0000	0	00001	0
0001	1	000	0	0000	0	00001	$\frac{1}{64}$
0010	1	001	0	0010	0	00101	0
0011	1	001	$\frac{1}{16}$	0010	$\frac{2}{32}$	00101	$\frac{5}{64}$
0100	1	010	0	0100	0	01001	0
0101	1	010	$\frac{2}{16}$	0100	$\frac{4}{32}$	01001	$\frac{9}{64}$
0110	1	011	0	0110	0	01101	0
0111	1	011	$\frac{3}{16}$	0110	$\frac{6}{32}$	01101	$\frac{13}{64}$
1000	1	100	0	1000	0	10001	0
1001	1	100	$\frac{4}{16}$	1000	$\frac{8}{32}$	10001	$\frac{17}{64}$
1010	1	101	0	1010	0	10101	0
1011	1	101	$\frac{5}{16}$	1010	$\frac{10}{32}$	10101	$\frac{21}{64}$
1100	1	110	0	1100	0	11001	0
1101	1	110	$\frac{6}{16}$	1100	$\frac{12}{32}$	11001	$\frac{25}{64}$
1110	1	111	0	1110	0	11101	0
1111	1	111	$\frac{7}{16}$	1110	$\frac{14}{32}$	11101	$\frac{29}{64}$

Table A-49 Twiddle factors, represented by fractions, for 16 × 4-point FFT using Pease algorithm mapped to PE2 and PE3 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_3b_2b_1b_0$	p_1p_0	0	$\frac{r}{N}$	b_3	$\frac{r}{N}$	b_3b_4	$\frac{r}{N}$
0000	2	0	0	0	0	00	0
0001	2	0	0	0	0	00	0
0010	2	0	0	0	0	00	0
0011	2	0	0	0	0	00	0
0100	2	0	0	0	0	01	0
0101	2	0	0	0	0	01	$\frac{1}{8}$
0110	2	0	0	0	0	01	0
0111	2	0	0	0	0	01	$\frac{1}{8}$
1000	2	0	0	1	0	10	0
1001	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	2	0	0	1	0	10	0
1011	2	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	2	0	0	1	0	11	0
1101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	2	0	0	1	0	11	0
1111	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	3	0	0	0	0	00	0
0001	3	0	0	0	0	00	0
0010	3	0	0	0	0	00	0
0011	3	0	0	0	0	00	0
0100	3	0	0	0	0	01	0
0101	3	0	0	0	0	01	$\frac{1}{8}$
0110	3	0	0	0	0	01	0
0111	3	0	0	0	0	01	$\frac{1}{8}$
1000	3	0	0	1	0	10	0
1001	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1010	3	0	0	1	0	10	0
1011	3	0	0	1	$\frac{1}{4}$	10	$\frac{2}{8}$
1100	3	0	0	1	0	11	0
1101	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1110	3	0	0	1	0	11	0
1111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-50 Twiddle factors, represented by fractions, for 16×4-point FFT using Pease algorithm mapped to PE2 and PE3 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_3b_2b_10b_0$	p_1p_0	$b_3b_2b_1$	$\frac{r}{N}$	$b_3b_2b_1p_1$	$\frac{r}{N}$	$b_3b_2b_1p_1p_0$	$\frac{r}{N}$
0000	2	000	0	0001	0	00010	0
0001	2	000	0	0001	$\frac{1}{32}$	00010	$\frac{2}{64}$
0010	2	001	0	0011	0	00110	0
0011	2	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00110	$\frac{6}{64}$
0100	2	010	0	0101	0	01010	0
0101	2	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01010	$\frac{10}{64}$
0110	2	011	0	0111	0	01110	0
0111	2	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01110	$\frac{14}{64}$
1000	2	100	0	1001	0	10010	0
1001	2	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10010	$\frac{18}{64}$
1010	2	101	0	1011	0	10110	0
1011	2	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10110	$\frac{22}{64}$
1100	2	110	0	1101	0	11010	0
1101	2	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11010	$\frac{26}{64}$
1110	2	111	0	1111	0	11110	0
1111	2	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11110	$\frac{30}{64}$
0000	3	000	0	0001	0	00011	0
0001	3	000	0	0001	$\frac{1}{32}$	00011	$\frac{3}{64}$
0010	3	001	0	0011	0	00111	0
0011	3	001	$\frac{1}{16}$	0011	$\frac{3}{32}$	00111	$\frac{7}{64}$
0100	3	010	0	0101	0	01011	0
0101	3	010	$\frac{2}{16}$	0101	$\frac{5}{32}$	01011	$\frac{11}{64}$
0110	3	011	0	0111	0	01111	0
0111	3	011	$\frac{3}{16}$	0111	$\frac{7}{32}$	01111	$\frac{15}{64}$
1000	3	100	0	1001	0	10011	0
1001	3	100	$\frac{4}{16}$	1001	$\frac{9}{32}$	10011	$\frac{19}{64}$
1010	3	101	0	1011	0	10111	0
1011	3	101	$\frac{5}{16}$	1011	$\frac{11}{32}$	10111	$\frac{23}{64}$
1100	3	110	0	1101	0	11011	0
1101	3	110	$\frac{6}{16}$	1101	$\frac{13}{32}$	11011	$\frac{27}{64}$
1110	3	111	0	1111	0	11111	0
1111	3	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11111	$\frac{31}{64}$

A.4 Tables for 64-point Optimal Algorithm

Table A-51 Addresses following the optimal algorithm at stage 1, 2 and 3

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_5b_4b_3b_0$		$b_2b_1b_5b_4b_0b_3$		$b_2b_1b_5b_0b_4b_3$	
000000	0	000000	0	000000	0	000000	0
000001	0	000001	1	000010	2	000100	4
000010	1	010000	16	010000	16	010000	16
000011	1	010001	17	010010	18	010100	20
000100	2	100000	32	100000	32	100000	32
000101	2	100001	33	100010	34	100100	36
000110	3	110000	48	110000	48	110000	48
000111	3	110001	49	110010	50	110100	52
001000	0	000010	2	000001	1	000001	1
001001	0	000011	3	000011	3	000101	5
001010	1	010010	18	010001	17	010001	17
001011	1	010011	19	010011	19	010101	21
001100	2	100010	34	100001	33	100001	33
001101	2	100011	35	100011	35	100101	37
001110	3	110010	50	110001	49	110001	49
001111	3	110011	51	110011	51	110101	53
010000	0	000100	4	000100	4	000010	2
010001	0	000101	5	000110	6	000110	6
010010	1	010100	20	010100	20	010010	18
010011	1	010101	21	010110	22	010110	22
010100	2	100100	36	100100	36	100010	34
010101	2	100101	37	100110	38	100110	38
010110	3	110100	52	110100	52	110010	50
010111	3	110101	53	110110	54	110110	54
011000	0	000110	6	000101	5	000011	3
011001	0	000111	7	000111	7	000111	7
011010	1	010110	22	010101	21	010011	19
011011	1	010111	23	010111	23	010111	23
011100	2	100110	38	100101	37	100011	35
011101	2	100111	39	100111	39	100111	39
011110	3	110110	54	110101	53	110011	51
011111	3	110111	55	110111	55	110111	55

Table A-52 Addresses following the optimal algorithm at stage 1, 2 and 3 (cont.)

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_5b_4b_3b_0$		$b_2b_1b_5b_4b_0b_3$		$b_2b_1b_5b_0b_4b_3$	
100000	0	001000	8	001000	8	001000	8
100001	0	001001	9	001010	10	001100	12
100010	1	011000	24	011000	24	011000	24
100011	1	011001	25	011010	26	011100	28
100100	2	101000	40	101000	40	101000	40
100101	2	101001	41	101010	42	101100	44
100110	3	111000	56	111000	56	111000	56
100111	3	111001	57	111010	58	111100	60
101000	0	001010	10	001001	9	001001	9
101001	0	001011	11	001011	11	001101	13
101010	1	011010	26	011001	25	011001	25
101011	1	011011	27	011011	27	011101	29
101100	2	101010	42	101001	41	101001	41
101101	2	101011	43	101011	43	101101	45
101110	3	111010	58	111001	57	111001	57
101111	3	111011	59	111011	59	111101	61
110000	0	001100	12	001100	12	001010	10
110001	0	001101	13	001110	14	001110	14
110010	1	011100	28	011100	28	011010	26
110011	1	011101	29	011110	30	011110	30
110100	2	101100	44	101100	44	101010	42
110101	2	101101	45	101110	46	101110	46
110110	3	111100	60	111100	60	111010	58
110111	3	111101	61	111110	62	111110	62
111000	0	001110	14	001101	13	001011	11
111001	0	001111	15	001111	15	001111	15
111010	1	011110	30	011101	29	011011	27
111011	1	011111	31	011111	31	011111	31
111100	2	101110	46	101101	45	101011	43
111101	2	101111	47	101111	47	101111	47
111110	3	111110	62	111101	61	111011	59
111111	3	111111	63	111111	63	111111	63

Table A-53 Addresses following the optimal algorithm at stage 4, 5 and 6

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_2b_0b_1b_5b_4b_3$		$b_0b_1b_5b_4b_3b_2$	
000000	0	000000	0	000000	0	000000	0
000001	0	001000	8	010000	16	100000	32
000010	1	010000	16	001000	8	010000	16
000011	1	011000	24	011000	24	110000	48
000100	2	100000	32	100000	32	000001	1
000101	2	101000	40	110000	48	100001	33
000110	3	110000	48	101000	40	010001	17
000111	3	111000	56	111000	56	110001	49
001000	0	000001	1	000001	1	000010	2
001001	0	001001	9	010001	17	100010	34
001010	1	010001	17	001001	9	010010	18
001011	1	011001	25	011001	25	110010	50
001100	2	100001	33	100001	33	000011	3
001101	2	101001	41	110001	49	100011	35
001110	3	110001	49	101001	41	010011	19
001111	3	111001	57	111001	57	110011	51
010000	0	000010	2	000010	2	000100	4
010001	0	001010	10	010010	18	100100	36
010010	1	010010	18	001010	10	010100	20
010011	1	011010	26	011010	26	110100	52
010100	2	100010	34	100010	34	000101	5
010101	2	101010	42	110010	50	100101	37
010110	3	110010	50	101010	42	010101	21
010111	3	111010	58	111010	58	110101	53
011000	0	000011	3	000011	3	000110	6
011001	0	001011	11	010011	19	100110	38
011010	1	010011	19	001011	11	010110	22
011011	1	011011	27	011011	27	110110	54
011100	2	100011	35	100011	35	000111	7
011101	2	101011	43	110011	51	100111	39
011110	3	110011	51	101011	43	010111	23
011111	3	111011	59	111011	59	110111	55

Table A-54 Addresses following the optimal algorithm at stage 4, 5 and 6 (cont.)

Counter (l)	Mapped to PE	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_2b_0b_1b_5b_4b_3$		$b_0b_1b_5b_4b_3b_2$	
100000	0	000100	4	000100	4	001000	8
100001	0	001100	12	010100	20	101000	40
100010	1	010100	20	001100	12	011000	24
100011	1	011100	28	011100	28	111000	56
100100	2	100100	36	100100	36	001001	9
100101	2	101100	44	110100	52	101001	41
100110	3	110100	52	101100	44	011001	25
100111	3	111100	60	111100	60	111001	57
101000	0	000101	5	000101	5	001010	10
101001	0	001101	13	010101	21	101010	42
101010	1	010101	21	001101	13	011010	26
101011	1	011101	29	011101	29	111010	58
101100	2	100101	37	100101	37	001011	11
101101	2	101101	45	110101	53	101011	43
101110	3	110101	53	101101	45	011011	27
101111	3	111101	61	111101	61	111011	59
110000	0	000110	6	000110	6	001100	12
110001	0	001110	14	010110	22	101100	44
110010	1	010110	22	001110	14	011100	28
110011	1	011110	30	011110	30	111100	60
110100	2	100110	38	100110	38	001101	13
110101	2	101110	46	110110	54	101101	45
110110	3	110110	54	101110	46	011101	29
110111	3	111110	62	111110	62	111101	61
111000	0	000111	7	000111	7	001110	14
111001	0	001111	15	010111	23	101110	46
111010	1	010111	23	001111	15	011110	30
111011	1	011111	31	011111	31	111110	62
111100	2	100111	39	100111	39	001111	15
111101	2	101111	47	110111	55	101111	47
111110	3	110111	55	101111	47	011111	31
111111	3	111111	63	111111	63	111111	63

Table A-55 Addresses mapped to PE0 and PE1 at stage 1, 2 and 3 following the optimal alogrithm

Counter	PID	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_5b_4b_3b_0$		$b_2b_1b_5b_4b_0b_3$		$b_2b_1b_5b_0b_4b_3$	
000000	0	000000	0	000000	0	000000	0
000001	0	000001	1	000010	2	000100	4
001000	0	000010	2	000001	1	000001	1
001001	0	000011	3	000011	3	000101	5
010000	0	000100	4	000100	4	000010	2
010001	0	000101	5	000110	6	000110	6
011000	0	000110	6	000101	5	000011	3
011001	0	000111	7	000111	7	000111	7
100000	0	001000	8	001000	8	001000	8
100001	0	001001	9	001010	10	001100	12
101000	0	001010	10	001001	9	001001	9
101001	0	001011	11	001011	11	001101	13
110000	0	001100	12	001100	12	001010	10
110001	0	001101	13	001110	14	001110	14
111000	0	001110	14	001101	13	001011	11
111001	0	001111	15	001111	15	001111	15
000010	1	010000	16	010000	16	010000	16
000011	1	010001	17	010010	18	010100	20
001010	1	010010	18	010001	17	010001	17
001011	1	010011	19	010011	19	010101	21
010010	1	010100	20	010100	20	010010	18
010011	1	010101	21	010110	22	010110	22
011010	1	010110	22	010101	21	010011	19
011011	1	010111	23	010111	23	010111	23
100010	1	011000	24	011000	24	011000	24
100011	1	011001	25	011010	26	011100	28
101010	1	011010	26	011001	25	011001	25
101011	1	011011	27	011011	27	011101	29
110010	1	011100	28	011100	28	011010	26
110011	1	011101	29	011110	30	011110	30
111010	1	011110	30	011101	29	011011	27
111011	1	011111	31	011111	31	011111	31

Table A-56 Addresses mapped to PE0 and PE1 at stage 4, 5 and 6 following the optimal algorithm

Counter	PID	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_2b_0b_1b_5b_4b_3$		$b_0b_1b_5b_4b_3b_2$	
000000	0	000000	0	000000	0	000000	0
000001	0	001000	8	010000	16	100000	32
001000	0	000001	1	000001	1	000010	2
001001	0	001001	9	010001	17	100010	34
010000	0	000010	2	000010	2	000100	4
010001	0	001010	10	010010	18	100100	36
011000	0	000011	3	000011	3	000110	6
011001	0	001011	11	010011	19	100110	38
100000	0	000100	4	000100	4	001000	8
100001	0	001100	12	010100	20	101000	40
101000	0	000101	5	000101	5	001010	10
101001	0	001101	13	010101	21	101010	42
110000	0	000110	6	000110	6	001100	12
110001	0	001110	14	010110	22	101100	44
111000	0	000111	7	000111	7	001110	14
111001	0	001111	15	010111	23	101110	46
000010	1	010000	16	001000	8	010000	16
000011	1	011000	24	011000	24	110000	48
001010	1	010001	17	001001	9	010010	18
001011	1	011001	25	011001	25	110010	50
010010	1	010010	18	001010	10	010100	20
010011	1	011010	26	011010	26	110100	52
011010	1	010011	19	001011	11	010110	22
011011	1	011011	27	011011	27	110110	54
100010	1	010100	20	001100	12	011000	24
100011	1	011100	28	011100	28	111000	56
101010	1	010101	21	001101	13	011010	26
101011	1	011101	29	011101	29	111010	58
110010	1	010110	22	001110	14	011100	28
110011	1	011110	30	011110	30	111100	60
111010	1	010111	23	001111	15	011110	30
111011	1	011111	31	011111	31	111110	62

Table A-57 Addresses mapped to PE2 and PE3 at stage 1, 2 and 3 following the optimal algorithm

Counter	PID	Addresses (k)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_5b_4b_3b_0$		$b_2b_1b_5b_4b_0b_3$		$b_2b_1b_5b_0b_4b_3$	
000100	2	100000	32	100000	32	100000	32
000101	2	100001	33	100010	34	100100	36
001100	2	100010	34	100001	33	100001	33
001101	2	100011	35	100011	35	100101	37
010100	2	100100	36	100100	36	100010	34
010101	2	100101	37	100110	38	100110	38
011100	2	100110	38	100101	37	100011	35
011101	2	100111	39	100111	39	100111	39
100100	2	101000	40	101000	40	101000	40
100101	2	101001	41	101010	42	101100	44
101100	2	101010	42	101001	41	101001	41
101101	2	101011	43	101011	43	101101	45
110100	2	101100	44	101100	44	101010	42
110101	2	101101	45	101110	46	101110	46
111100	2	101110	46	101101	45	101011	43
111101	2	101111	47	101111	47	101111	47
000110	3	110000	48	110000	48	110000	48
000111	3	110001	49	110010	50	110100	52
001110	3	110010	50	110001	49	110001	49
001111	3	110011	51	110011	51	110101	53
010110	3	110100	52	110100	52	110010	50
010111	3	110101	53	110110	54	110110	54
011110	3	110110	54	110101	53	110011	51
011111	3	110111	55	110111	55	110111	55
100110	3	111000	56	111000	56	111000	56
100111	3	111001	57	111010	58	111100	60
101110	3	111010	58	111001	57	111001	57
101111	3	111011	59	111011	59	111101	61
110110	3	111100	60	111100	60	111010	58
110111	3	111101	61	111110	62	111110	62
111110	3	111110	62	111101	61	111011	59
111111	3	111111	63	111111	63	111111	63

Table A-58 Addresses mapped to PE2 and PE3 at stage 4, 5 and 6 following the optimal algorithm

Counter	PID	Addresses (k)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_2b_1b_0b_5b_4b_3$		$b_2b_0b_1b_5b_4b_3$		$b_0b_1b_5b_4b_3b_2$	
000100	2	100000	32	100000	32	000001	1
000101	2	101000	40	110000	48	100001	33
001100	2	100001	33	100001	33	000011	3
001101	2	101001	41	110001	49	100011	35
010100	2	100010	34	100010	34	000101	5
010101	2	101010	42	110010	50	100101	37
011100	2	100011	35	100011	35	000111	7
011101	2	101011	43	110011	51	100111	39
100100	2	100100	36	100100	36	001001	9
100101	2	101100	44	110100	52	101001	41
101100	2	100101	37	100101	37	001011	11
101101	2	101101	45	110101	53	101011	43
110100	2	100110	38	100110	38	001101	13
110101	2	101110	46	110110	54	101101	45
111100	2	100111	39	100111	39	001111	15
111101	2	101111	47	110111	55	101111	47
000110	3	110000	48	101000	40	010001	17
000111	3	111000	56	111000	56	110001	49
001110	3	110001	49	101001	41	010011	19
001111	3	111001	57	111001	57	110011	51
010110	3	110010	50	101010	42	010101	21
010111	3	111010	58	111010	58	110101	53
011110	3	110011	51	101011	43	010111	23
011111	3	111011	59	111011	59	110111	55
100110	3	110100	52	101100	44	011001	25
100111	3	111100	60	111100	60	111001	57
101110	3	110101	53	101101	45	011011	27
101111	3	111101	61	111101	61	111011	59
110110	3	110110	54	101110	46	011101	29
110111	3	111110	62	111110	62	111101	61
111110	3	110111	55	101111	47	011111	31
111111	3	111111	63	111111	63	111111	63

Table A-59 Source MID, target PID and local ddresses in PE0 and PE1 at stage 1 and 2 following the optimal algorithm

Counter	Stage 1				Stage 2			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1p_0	p_1p_0	$a_3a_2a_1a_0$		p_1p_0	p_1p_0	$a_3a_2a_0a_1$	
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0001	1	0	0	0010	2
0010	0	0	0010	2	0	0	0001	1
0011	0	0	0011	3	0	0	0011	3
0100	0	0	0100	4	0	0	0100	4
0101	0	0	0101	5	0	0	0110	6
0110	0	0	0110	6	0	0	0101	5
0111	0	0	0111	7	0	0	0111	7
1000	0	0	1000	8	0	0	1000	8
1001	0	0	1001	9	0	0	1010	10
1010	0	0	1010	10	0	0	1001	9
1011	0	0	1011	11	0	0	1011	11
1100	0	0	1100	12	0	0	1100	12
1101	0	0	1101	13	0	0	1110	14
1110	0	0	1110	14	0	0	1101	13
1111	0	0	1111	15	0	0	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	1	1	0000	0	1	1	0000	0
0001	1	1	0001	1	1	1	0010	2
0010	1	1	0010	2	1	1	0001	1
0011	1	1	0011	3	1	1	0011	3
0100	1	1	0100	4	1	1	0100	4
0101	1	1	0101	5	1	1	0110	6
0110	1	1	0110	6	1	1	0101	5
0111	1	1	0111	7	1	1	0111	7
1000	1	1	1000	8	1	1	1000	8
1001	1	1	1001	9	1	1	1010	10
1010	1	1	1010	10	1	1	1001	9
1011	1	1	1011	11	1	1	1011	11
1100	1	1	1100	12	1	1	1100	12
1101	1	1	1101	13	1	1	1110	14
1110	1	1	1110	14	1	1	1101	13
1111	1	1	1111	15	1	1	1111	15

Table A-60 Source MID, target PID and local ddresses in PE0 and PE1 at stage 3 and 4 followig the optimal algorithm

Counter	Stage 3				Stage 4			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1p_0	p_1p_0	$a_3a_0a_2a_1$		p_1p_0	p_1p_0	$a_0a_3a_2a_1$	
0000	0	0	0000	0	0	0	0000	0
0001	0	0	0100	4	0	0	1000	8
0010	0	0	0001	1	0	0	0001	1
0011	0	0	0101	5	0	0	1001	9
0100	0	0	0010	2	0	0	0010	2
0101	0	0	0110	6	0	0	1010	10
0110	0	0	0011	3	0	0	0011	3
0111	0	0	0111	7	0	0	1011	11
1000	0	0	1000	8	0	0	0100	4
1001	0	0	1100	12	0	0	1100	12
1010	0	0	1001	9	0	0	0101	5
1011	0	0	1101	13	0	0	1101	13
1100	0	0	1010	10	0	0	0110	6
1101	0	0	1110	14	0	0	1110	14
1110	0	0	1011	11	0	0	0111	7
1111	0	0	1111	15	0	0	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	1	1	0000	0	1	1	0000	0
0001	1	1	0100	4	1	1	1000	8
0010	1	1	0001	1	1	1	0001	1
0011	1	1	0101	5	1	1	1001	9
0100	1	1	0010	2	1	1	0010	2
0101	1	1	0110	6	1	1	1010	10
0110	1	1	0011	3	1	1	0011	3
0111	1	1	0111	7	1	1	1011	11
1000	1	1	1000	8	1	1	0100	4
1001	1	1	1100	12	1	1	1100	12
1010	1	1	1001	9	1	1	0101	5
1011	1	1	1101	13	1	1	1101	13
1100	1	1	1010	10	1	1	0110	6
1101	1	1	1110	14	1	1	1110	14
1110	1	1	1011	11	1	1	0111	7
1111	1	1	1111	15	1	1	1111	15

Table A-61 Source MID, target PID and local ddresses in PE0 and PE1 at stage 5 and 6 following the optimal algorithm

Counter	Stage 5				Stage 6			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1a_0	p_1a_0	$a_0a_3a_2a_1$		a_0p_0	a_0p_0	$a_3a_2a_1a_0$	
0000	0	0	0000	0	0	0	0000	0
0001	1	1	1000	8	2	2	0001	1
0010	0	0	0001	1	0	0	0010	2
0011	1	1	1001	9	2	2	0011	3
0100	0	0	0010	2	0	0	0100	4
0101	1	1	1010	10	2	2	0101	5
0110	0	0	0011	3	0	0	0110	6
0111	1	1	1011	11	2	2	0111	7
1000	0	0	0100	4	0	0	1000	8
1001	1	1	1100	12	2	2	1001	9
1010	0	0	0101	5	0	0	1010	10
1011	1	1	1101	13	2	2	1011	11
1100	0	0	0110	6	0	0	1100	12
1101	1	1	1110	14	2	2	1101	13
1110	0	0	0111	7	0	0	1110	14
1111	1	1	1111	15	2	2	1111	15
PE1 $((p_1p_0)_2 = (01)_2)$								
0000	0	0	0000	0	1	1	0000	0
0001	1	1	1000	8	3	3	0001	1
0010	0	0	0001	1	1	1	0010	2
0011	1	1	1001	9	3	3	0011	3
0100	0	0	0010	2	1	1	0100	4
0101	1	1	1010	10	3	3	0101	5
0110	0	0	0011	3	1	1	0110	6
0111	1	1	1011	11	3	3	0111	7
1000	0	0	0100	4	1	1	1000	8
1001	1	1	1100	12	3	3	1001	9
1010	0	0	0101	5	1	1	1010	10
1011	1	1	1101	13	3	3	1011	11
1100	0	0	0110	6	1	1	1100	12
1101	1	1	1110	14	3	3	1101	13
1110	0	0	0111	7	1	1	1110	14
1111	1	1	1111	15	3	3	1111	15

Table A-62 Source MID, target PID and local ddresses in PE2 and PE3 at stage 1 and 2 following the optimal algorithm

Counter	Stage 1				Stage 2			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1p_0	p_1p_0	$a_3a_2a_1a_0$		p_1p_0	p_1p_0	$a_3a_2a_0a_1$	
0000	2	2	0000	0	2	2	0000	0
0001	2	2	0001	1	2	2	0010	2
0010	2	2	0010	2	2	2	0001	1
0011	2	2	0011	3	2	2	0011	3
0100	2	2	0100	4	2	2	0100	4
0101	2	2	0101	5	2	2	0110	6
0110	2	2	0110	6	2	2	0101	5
0111	2	2	0111	7	2	2	0111	7
1000	2	2	1000	8	2	2	1000	8
1001	2	2	1001	9	2	2	1010	10
1010	2	2	1010	10	2	2	1001	9
1011	2	2	1011	11	2	2	1011	11
1100	2	2	1100	12	2	2	1100	12
1101	2	2	1101	13	2	2	1110	14
1110	2	2	1110	14	2	2	1101	13
1111	2	2	1111	15	2	2	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	3	3	0000	0	3	3	0000	0
0001	3	3	0001	1	3	3	0010	2
0010	3	3	0010	2	3	3	0001	1
0011	3	3	0011	3	3	3	0011	3
0100	3	3	0100	4	3	3	0100	4
0101	3	3	0101	5	3	3	0110	6
0110	3	3	0110	6	3	3	0101	5
0111	3	3	0111	7	3	3	0111	7
1000	3	3	1000	8	3	3	1000	8
1001	3	3	1001	9	3	3	1010	10
1010	3	3	1010	10	3	3	1001	9
1011	3	3	1011	11	3	3	1011	11
1100	3	3	1100	12	3	3	1100	12
1101	3	3	1101	13	3	3	1110	14
1110	3	3	1110	14	3	3	1101	13
1111	3	3	1111	15	3	3	1111	15

Table A-63 Source MID, target PID and local ddresses in PE2 and PE3 at stage 3 and 4 following the optimal algorithm

Counter	Stage 3				Stage 4			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1p_0	p_1p_0	$a_3a_0a_2a_1$		p_1p_0	p_1p_0	$a_0a_3a_2a_1$	
0000	2	2	0000	0	2	2	0000	0
0001	2	2	0100	4	2	2	1000	8
0010	2	2	0001	1	2	2	0001	1
0011	2	2	0101	5	2	2	1001	9
0100	2	2	0010	2	2	2	0010	2
0101	2	2	0110	6	2	2	1010	10
0110	2	2	0011	3	2	2	0011	3
0111	2	2	0111	7	2	2	1011	11
1000	2	2	1000	8	2	2	0100	4
1001	2	2	1100	12	2	2	1100	12
1010	2	2	1001	9	2	2	0101	5
1011	2	2	1101	13	2	2	1101	13
1100	2	2	1010	10	2	2	0110	6
1101	2	2	1110	14	2	2	1110	14
1110	2	2	1011	11	2	2	0111	7
1111	2	2	1111	15	2	2	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	3	3	0000	0	3	3	0000	0
0001	3	3	0100	4	3	3	1000	8
0010	3	3	0001	1	3	3	0001	1
0011	3	3	0101	5	3	3	1001	9
0100	3	3	0010	2	3	3	0010	2
0101	3	3	0110	6	3	3	1010	10
0110	3	3	0011	3	3	3	0011	3
0111	3	3	0111	7	3	3	1011	11
1000	3	3	1000	8	3	3	0100	4
1001	3	3	1100	12	3	3	1100	12
1010	3	3	1001	9	3	3	0101	5
1011	3	3	1101	13	3	3	1101	13
1100	3	3	1010	10	3	3	0110	6
1101	3	3	1110	14	3	3	1110	14
1110	3	3	1011	11	3	3	0111	7
1111	3	3	1111	15	3	3	1111	15

Table A-64 Source MID, target PID and local ddresses in PE2 and PE3 at stage 5 and 6 following the optimal algorithm

Counter	Stage 5				Stage 6			
	MID	PID	Address		MID	PID	Address	
$a_3a_2a_1a_0$	p_1a_0	p_1a_0	$a_0a_3a_2a_1$		a_0p_0	a_0p_0	$a_3a_2a_1a_0$	
0000	2	2	0000	0	0	0	0000	0
0001	3	3	1000	8	2	2	0001	1
0010	2	2	0001	1	0	0	0010	2
0011	3	3	1001	9	2	2	0011	3
0100	2	2	0010	2	0	0	0100	4
0101	3	3	1010	10	2	2	0101	5
0110	2	2	0011	3	0	0	0110	6
0111	3	3	1011	11	2	2	0111	7
1000	2	2	0100	4	0	0	1000	8
1001	3	3	1100	12	2	2	1001	9
1010	2	2	0101	5	0	0	1010	10
1011	3	3	1101	13	2	2	1011	11
1100	2	2	0110	6	0	0	1100	12
1101	3	3	1110	14	2	2	1101	13
1110	2	2	0111	7	0	0	1110	14
1111	3	3	1111	15	2	2	1111	15
PE3 $((p_1p_0)_2 = (11)_2)$								
0000	2	2	0000	0	1	1	0000	0
0001	3	3	1000	8	3	3	0001	1
0010	2	2	0001	1	1	1	0010	2
0011	3	3	1001	9	3	3	0011	3
0100	2	2	0010	2	1	1	0100	4
0101	3	3	1010	10	3	3	0101	5
0110	2	2	0011	3	1	1	0110	6
0111	3	3	1011	11	3	3	0111	7
1000	2	2	0100	4	1	1	1000	8
1001	3	3	1100	12	3	3	1001	9
1010	2	2	0101	5	1	1	1010	10
1011	3	3	1101	13	3	3	1011	11
1100	2	2	0110	6	1	1	1100	12
1101	3	3	1110	14	3	3	1101	13
1110	2	2	0111	7	1	1	1110	14
1111	3	3	1111	15	3	3	1111	15

Table A-65 Twiddle factors of 1-D 64-point FFT following optimal algorithm at stage 1 to 3

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{0}{N}$	b_3	$\frac{r}{N}$	b_4b_3	$\frac{r}{N}$
000000	0	0	0	0	0	00	0
000001	0	0	0	0	0	00	0
000010	1	0	0	0	0	00	0
000011	1	0	0	0	0	00	0
000100	2	0	0	0	0	00	0
000101	2	0	0	0	0	00	0
000110	3	0	0	0	0	00	0
000111	3	0	0	0	0	00	0
001000	0	0	0	1	0	01	0
001001	0	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
001010	1	0	0	1	0	01	0
001011	1	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
001100	2	0	0	1	0	01	0
001101	2	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
001110	3	0	0	1	0	01	0
001111	3	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
010000	0	0	0	0	0	10	0
010001	0	0	0	0	0	10	$\frac{2}{8}$
010010	1	0	0	0	0	10	0
010011	1	0	0	0	0	10	$\frac{2}{8}$
010100	2	0	0	0	0	10	0
010101	2	0	0	0	0	10	$\frac{2}{8}$
010110	3	0	0	0	0	10	0
010111	3	0	0	0	0	10	$\frac{2}{8}$
011000	0	0	0	1	0	11	0
011001	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
011010	1	0	0	1	0	11	0
011011	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
011100	2	0	0	1	0	11	0
011101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
011110	3	0	0	1	0	11	0
011111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-66 Twiddle factors of 1-D 64-point FFT following the optimal algorithm at stage 1 to 3 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{0}{N}$	b_3	$\frac{r}{N}$	b_4b_3	$\frac{r}{N}$
100000	0	0	0	0	0	00	0
100001	0	0	0	0	0	00	0
100010	1	0	0	0	0	00	0
100011	1	0	0	0	0	00	0
100100	2	0	0	0	0	00	0
100101	2	0	0	0	0	00	0
100110	3	0	0	0	0	00	0
100111	3	0	0	0	0	00	0
101000	0	0	0	1	0	01	0
101001	0	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
101010	1	0	0	1	0	01	0
101011	1	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
101100	2	0	0	1	0	01	0
101101	2	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
101110	3	0	0	1	0	01	0
101111	3	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
110000	0	0	0	0	0	10	0
110001	0	0	0	0	0	10	$\frac{2}{8}$
110010	1	0	0	0	0	10	0
110011	1	0	0	0	0	10	$\frac{2}{8}$
110100	2	0	0	0	0	10	0
110101	2	0	0	0	0	10	$\frac{2}{8}$
110110	3	0	0	0	0	10	0
110111	3	0	0	0	0	10	$\frac{2}{8}$
111000	0	0	0	1	0	11	0
111001	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111010	1	0	0	1	0	11	0
111011	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111100	2	0	0	1	0	11	0
111101	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
111110	3	0	0	1	0	11	0
111111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-67 Twiddle factors of 1-D 64-point FFT following the optimal algorithm at stage 5 to 6

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3$	$\frac{r}{N}$	$b_1b_5b_4b_3$	$\frac{r}{N}$	$b_1b_2b_5b_4b_3$	$\frac{r}{N}$
000000	0	000	0	0000	0	00000	0
000001	0	000	0	0000	0	00000	0
000010	1	000	0	1000	0	10000	0
000011	1	000	0	1000	$\frac{8}{32}$	10000	$\frac{16}{64}$
000100	2	000	0	0000	0	01000	0
000101	2	000	0	0000	0	01000	$\frac{8}{64}$
000110	3	000	0	1000	0	11000	0
000111	3	000	0	1000	$\frac{8}{32}$	11000	$\frac{24}{64}$
001000	0	001	0	0001	0	00001	0
001001	0	001	$\frac{1}{16}$	0001	$\frac{1}{32}$	00001	$\frac{1}{64}$
001010	1	001	0	1001	0	10001	0
001011	1	001	$\frac{1}{16}$	1001	$\frac{9}{32}$	10001	$\frac{17}{64}$
001100	2	001	0	0001	0	01001	0
001101	2	001	$\frac{1}{16}$	0001	$\frac{1}{32}$	01001	$\frac{9}{64}$
001110	3	001	0	1001	0	11001	0
001111	3	001	$\frac{1}{16}$	1001	$\frac{9}{32}$	11001	$\frac{25}{64}$
010000	0	010	0	0010	0	00010	0
010001	0	010	$\frac{2}{16}$	0010	$\frac{2}{32}$	00010	$\frac{2}{64}$
010010	1	010	0	1010	0	10010	0
010011	1	010	$\frac{2}{16}$	1010	$\frac{10}{32}$	10010	$\frac{18}{64}$
010100	2	010	0	0010	0	01010	0
010101	2	010	$\frac{2}{16}$	0010	$\frac{2}{32}$	01010	$\frac{10}{64}$
010110	3	010	0	1010	0	11010	0
010111	3	010	$\frac{2}{16}$	1010	$\frac{10}{32}$	11010	$\frac{26}{64}$
011000	0	011	0	0011	0	00011	0
011001	0	011	$\frac{3}{16}$	0011	$\frac{3}{32}$	00011	$\frac{3}{64}$
011010	1	011	0	1011	0	10011	0
011011	1	011	$\frac{3}{16}$	1011	$\frac{11}{32}$	10011	$\frac{19}{64}$
011100	2	011	0	0011	0	01011	0
011101	2	011	$\frac{3}{16}$	0011	$\frac{3}{32}$	01011	$\frac{11}{64}$
011110	3	011	0	1011	0	11011	0
011111	3	011	$\frac{3}{16}$	1011	$\frac{11}{32}$	11011	$\frac{27}{64}$

Table A-68 Twiddle factors of 1-D 64-point FFT following the optimal algorithm at stage 5 to 6 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3$	$\frac{r}{N}$	$b_1b_5b_4b_3$	$\frac{r}{N}$	$b_1b_5b_4b_3b_2$	$\frac{r}{N}$
100000	0	100	0	0100	0	00100	0
100001	0	100	$\frac{4}{16}$	0100	$\frac{4}{32}$	00100	$\frac{4}{64}$
100010	1	100	0	1100	0	10100	0
100011	1	100	$\frac{4}{16}$	1100	$\frac{12}{32}$	10100	$\frac{20}{64}$
100100	2	100	0	0100	0	01100	0
100101	2	100	$\frac{4}{16}$	0100	$\frac{4}{32}$	01100	$\frac{12}{64}$
100110	3	100	0	1100	0	11100	0
100111	3	100	$\frac{4}{16}$	1100	$\frac{12}{32}$	11100	$\frac{28}{64}$
101000	0	101	0	0101	0	00101	0
101001	0	101	$\frac{5}{16}$	0101	$\frac{5}{32}$	00101	$\frac{5}{64}$
101010	1	101	0	1101	0	10101	0
101011	1	101	$\frac{5}{16}$	1101	$\frac{13}{32}$	10101	$\frac{21}{64}$
101100	2	101	0	0101	0	01101	0
101101	2	101	$\frac{5}{16}$	0101	$\frac{5}{32}$	01101	$\frac{13}{64}$
101110	3	101	0	1101	0	11101	0
101111	3	101	$\frac{5}{16}$	1101	$\frac{13}{32}$	11101	$\frac{29}{64}$
110000	0	110	0	0110	0	00110	0
110001	0	110	$\frac{6}{16}$	0110	$\frac{6}{32}$	00110	$\frac{6}{64}$
110010	1	110	0	1110	0	10110	0
110011	1	110	$\frac{6}{16}$	1110	$\frac{14}{32}$	10110	$\frac{22}{64}$
110100	2	110	0	0110	0	01110	0
110101	2	110	$\frac{6}{16}$	0110	$\frac{6}{32}$	01110	$\frac{14}{64}$
110110	3	110	0	1110	0	11110	0
110111	3	110	$\frac{6}{16}$	1110	$\frac{14}{32}$	11110	$\frac{30}{64}$
111000	0	111	0	0111	0	00111	0
111001	0	111	$\frac{7}{16}$	0111	$\frac{7}{32}$	00111	$\frac{7}{64}$
111010	1	111	0	1111	0	10111	0
111011	1	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	10111	$\frac{23}{64}$
111100	2	111	0	0111	0	01111	0
111101	2	111	$\frac{7}{16}$	0111	$\frac{7}{32}$	01111	$\frac{15}{64}$
111110	3	111	0	1111	0	11111	0
111111	3	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11111	$\frac{31}{64}$

Table A-69 Twiddle factors of 1-D 64-point FFT following the optimal algorithm mapped to PE0 and PE1 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$a_3a_2a_1a_0$	p_1p_0	0	$\frac{0}{N}$	a_1	$\frac{r}{N}$	a_2a_1	$\frac{r}{N}$
0000	0	0	0	0	0	00	0
0001	0	0	0	0	0	00	0
0010	0	0	0	1	0	01	0
0011	0	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
0100	0	0	0	0	0	10	0
0101	0	0	0	0	0	10	$\frac{2}{8}$
0110	0	0	0	1	0	11	0
0111	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1000	0	0	0	0	0	00	0
1001	0	0	0	0	0	00	0
1010	0	0	0	1	0	01	0
1011	0	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
1100	0	0	0	0	0	10	0
1101	0	0	0	0	0	10	$\frac{2}{8}$
1110	0	0	0	1	0	11	0
1111	0	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	1	0	0	0	0	00	0
0001	1	0	0	0	0	00	0
0010	1	0	0	1	0	01	0
0011	1	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
0100	1	0	0	0	0	10	0
0101	1	0	0	0	0	10	$\frac{2}{8}$
0110	1	0	0	1	0	11	0
0111	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1000	1	0	0	0	0	00	0
1001	1	0	0	0	0	00	0
1010	1	0	0	1	0	01	0
1011	1	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
1100	1	0	0	0	0	10	0
1101	1	0	0	0	0	10	$\frac{2}{8}$
1110	1	0	0	1	0	11	0
1111	1	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-70 Twiddle factors of 1-D 64-point FFT following the optimal algorithm mapped to PE0 and PE1 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$a_3a_2a_1a_0$	p_1p_0	$a_3a_2a_1$	$\frac{r}{N}$	$p_0a_3a_2a_1$	$\frac{r}{N}$	$p_0p_1a_3a_2a_1$	$\frac{r}{N}$
0000	0	000	0	0000	0	00000	0
0001	0	000	0	0000	0	00000	0
0010	0	001	0	0001	0	00010	0
0011	0	001	$\frac{1}{16}$	0001	$\frac{1}{32}$	00001	$\frac{1}{64}$
0100	0	010	0	0010	0	00010	0
0101	0	010	$\frac{2}{16}$	0010	$\frac{2}{32}$	00010	$\frac{2}{64}$
0110	0	011	0	0011	0	00011	0
0111	0	011	$\frac{3}{16}$	0011	$\frac{3}{32}$	00011	$\frac{3}{64}$
1000	0	100	0	0100	0	00100	0
1001	0	100	$\frac{4}{16}$	0100	$\frac{4}{32}$	00100	$\frac{4}{64}$
1010	0	101	0	0101	0	00101	0
1011	0	101	$\frac{5}{16}$	0101	$\frac{5}{32}$	00101	$\frac{5}{64}$
1100	0	110	0	0110	0	00110	0
1101	0	110	$\frac{6}{16}$	0110	$\frac{6}{32}$	00110	$\frac{6}{64}$
1110	0	111	0	0111	0	00111	0
1111	0	111	$\frac{7}{16}$	0111	$\frac{7}{32}$	00111	$\frac{7}{64}$
0000	1	000	0	1000	0	10000	0
0001	1	000	0	1000	$\frac{8}{32}$	10000	$\frac{16}{64}$
0010	1	001	0	1001	0	10001	0
0011	1	001	$\frac{1}{16}$	1001	$\frac{9}{32}$	10001	$\frac{17}{64}$
0100	1	010	0	1010	0	10010	0
0101	1	010	$\frac{2}{16}$	1010	$\frac{10}{32}$	10010	$\frac{18}{64}$
0110	1	011	0	1011	0	10011	0
0111	1	011	$\frac{3}{16}$	1011	$\frac{11}{32}$	10011	$\frac{19}{64}$
1000	1	100	0	1100	0	10100	0
1001	1	100	$\frac{4}{16}$	1100	$\frac{12}{32}$	10100	$\frac{20}{64}$
1010	1	101	0	1101	0	10101	0
1011	1	101	$\frac{5}{16}$	1101	$\frac{13}{32}$	10101	$\frac{21}{64}$
1100	1	110	0	1110	0	10110	0
1101	1	110	$\frac{6}{16}$	1110	$\frac{14}{32}$	10110	$\frac{22}{64}$
1110	1	111	0	1111	0	10111	0
1111	1	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	10111	$\frac{23}{64}$

Table A-71 Twiddle factors of 1-D 64-point FFT following the optimal algorithm mapped to PE2 and PE3 at stage 1 to 3

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$a_3a_2a_1a_0$	p_1p_0	0	$\frac{0}{N}$	a_1	$\frac{r}{N}$	a_2a_1	$\frac{r}{N}$
0000	2	0	0	0	0	00	0
0001	2	0	0	0	0	00	0
0010	2	0	0	1	0	01	0
0011	2	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
0100	2	0	0	0	0	10	0
0101	2	0	0	0	0	10	$\frac{2}{8}$
0110	2	0	0	1	0	11	0
0111	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1000	2	0	0	0	0	00	0
1001	2	0	0	0	0	00	0
1010	2	0	0	1	0	01	0
1011	2	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
1100	2	0	0	0	0	10	0
1101	2	0	0	0	0	10	$\frac{2}{8}$
1110	2	0	0	1	0	11	0
1111	2	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
0000	3	0	0	0	0	00	0
0001	3	0	0	0	0	00	0
0010	3	0	0	1	0	01	0
0011	3	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
0100	3	0	0	0	0	10	0
0101	3	0	0	0	0	10	$\frac{2}{8}$
0110	3	0	0	1	0	11	0
0111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$
1000	3	0	0	0	0	00	0
1001	3	0	0	0	0	00	0
1010	3	0	0	1	0	01	0
1011	3	0	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$
1100	3	0	0	0	0	10	0
1101	3	0	0	0	0	10	$\frac{2}{8}$
1110	3	0	0	1	0	11	0
1111	3	0	0	1	$\frac{1}{4}$	11	$\frac{3}{8}$

Table A-72 Twiddle factors of 1-D 64-point FFT following the optimal algorithm mapped to PE2 and PE3 at stage 4 to 6

Counter	PID	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$a_3a_2a_1a_0$	p_1p_0	$a_3a_2a_1$	$\frac{r}{N}$	$p_0a_3a_2a_1$	$\frac{r}{N}$	$p_0p_1a_3a_2a_1$	$\frac{r}{N}$
0000	2	000	0	0000	0	01000	0
0001	2	000	0	0000	0	01000	$\frac{1}{64}$
0010	2	001	0	0001	0	01001	0
0011	2	001	$\frac{1}{16}$	0001	$\frac{1}{32}$	01001	$\frac{3}{64}$
0100	2	010	0	0010	0	01010	0
0101	2	010	$\frac{2}{16}$	0010	$\frac{2}{32}$	01010	$\frac{5}{64}$
0110	2	011	0	0011	0	01011	0
0111	2	011	$\frac{3}{16}$	0011	$\frac{3}{32}$	01011	$\frac{7}{64}$
1000	2	100	0	0100	0	01100	0
1001	2	100	$\frac{4}{16}$	0100	$\frac{4}{32}$	01100	$\frac{9}{64}$
1010	2	101	0	0101	0	01101	0
1011	2	101	$\frac{5}{16}$	0101	$\frac{5}{32}$	01101	$\frac{11}{64}$
1100	2	110	0	0110	0	01110	0
1101	2	110	$\frac{6}{16}$	0110	$\frac{6}{32}$	01110	$\frac{13}{64}$
1110	2	111	0	0111	0	01111	0
1111	2	111	$\frac{7}{16}$	0111	$\frac{7}{32}$	01111	$\frac{15}{64}$
0000	3	000	0	1000	0	11000	0
0001	3	000	0	1000	$\frac{8}{32}$	11000	$\frac{24}{64}$
0010	3	001	0	1001	0	11001	0
0011	3	001	$\frac{1}{16}$	1001	$\frac{9}{32}$	11001	$\frac{25}{64}$
0100	3	010	0	1010	0	11010	0
0101	3	010	$\frac{2}{16}$	1010	$\frac{10}{32}$	11010	$\frac{26}{64}$
0110	3	011	0	1011	0	11011	0
0111	3	011	$\frac{3}{16}$	1011	$\frac{11}{32}$	11011	$\frac{27}{64}$
1000	3	100	0	1100	0	11100	0
1001	3	100	$\frac{4}{16}$	1100	$\frac{12}{32}$	11100	$\frac{28}{64}$
1010	3	101	0	1101	0	11101	0
1011	3	101	$\frac{5}{16}$	1101	$\frac{13}{32}$	11101	$\frac{29}{64}$
1100	3	110	0	1110	0	11110	0
1101	3	110	$\frac{6}{16}$	1110	$\frac{14}{32}$	11110	$\frac{30}{64}$
1110	3	111	0	1111	0	11111	0
1111	3	111	$\frac{7}{16}$	1111	$\frac{15}{32}$	11111	$\frac{31}{64}$

Table A-73 Twiddle factors of 2-D (16×4)-point FFT following optimal algorithm at stage 1 to 3

Counter	Mapped	Twiddle Fractions ($\frac{r}{N}$)					
	to	Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{0}{N}$	b_3	$\frac{r}{N}$	0	$\frac{r}{N}$
000000	0	0	0	0	0	0	0
000001	0	0	0	0	0	0	0
000010	1	0	0	0	0	0	0
000011	1	0	0	0	0	0	0
000100	2	0	0	0	0	0	0
000101	2	0	0	0	0	0	0
000110	3	0	0	0	0	0	0
000111	3	0	0	0	0	0	0
001000	0	0	0	1	0	0	0
001001	0	0	0	1	$\frac{1}{4}$	0	0
001010	1	0	0	1	0	0	0
001011	1	0	0	1	$\frac{1}{4}$	0	0
001100	2	0	0	1	0	0	0
001101	2	0	0	1	$\frac{1}{4}$	0	0
001110	3	0	0	1	0	0	0
001111	3	0	0	1	$\frac{1}{4}$	0	0
010000	0	0	0	0	0	0	0
010001	0	0	0	0	0	0	0
010010	1	0	0	0	0	0	0
010011	1	0	0	0	0	0	0
010100	2	0	0	0	0	0	0
010101	2	0	0	0	0	0	0
010110	3	0	0	0	0	0	0
010111	3	0	0	0	0	0	0
011000	0	0	0	1	0	0	0
011001	0	0	0	1	$\frac{1}{4}$	0	0
011010	1	0	0	1	0	0	0
011011	1	0	0	1	$\frac{1}{4}$	0	0
011100	2	0	0	1	0	0	0
011101	2	0	0	1	$\frac{1}{4}$	0	0
011110	3	0	0	1	0	0	0
011111	3	0	0	1	$\frac{1}{4}$	0	0

Table A-74 Twiddle factors of 2-D (16 $\times$ 4)-point FFT following the optimal algorithm at stage 1 to 3 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 1		Stage 2		Stage 3	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	0	$\frac{0}{N}$	b_3	$\frac{r}{N}$	0	$\frac{r}{N}$
100000	0	0	0	0	0	0	0
100001	0	0	0	0	0	0	0
100010	1	0	0	0	0	0	0
100011	1	0	0	0	0	0	0
100100	2	0	0	0	0	0	0
100101	2	0	0	0	0	0	0
100110	3	0	0	0	0	0	0
100111	3	0	0	0	0	0	0
101000	0	0	0	1	0	0	0
101001	0	0	0	1	$\frac{1}{4}$	0	0
101010	1	0	0	1	0	0	0
101011	1	0	0	1	$\frac{1}{4}$	0	0
101100	2	0	0	1	0	0	0
101101	2	0	0	1	$\frac{1}{4}$	0	0
101110	3	0	0	1	0	0	0
101111	3	0	0	1	$\frac{1}{4}$	0	0
110000	0	0	0	0	0	0	0
110001	0	0	0	0	0	0	0
110010	1	0	0	0	0	0	0
110011	1	0	0	0	0	0	0
110100	2	0	0	0	0	0	0
110101	2	0	0	0	0	0	0
110110	3	0	0	0	0	0	0
110111	3	0	0	0	0	0	0
111000	0	0	0	1	0	0	0
111001	0	0	0	1	$\frac{1}{4}$	0	0
111010	1	0	0	1	0	0	0
111011	1	0	0	1	$\frac{1}{4}$	0	0
111100	2	0	0	1	0	0	0
111101	2	0	0	1	$\frac{1}{4}$	0	0
111110	3	0	0	1	0	0	0
111111	3	0	0	1	$\frac{1}{4}$	0	0

Table A-75 Twiddle factors of 2-D (16×4)-point FFT following the optimal algorithm at stage 5 to 6

Counter	Mapped	Twiddle Fractions ($\frac{r}{N}$)					
	to	Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	b_5	$\frac{r}{N}$	b_1b_5	$\frac{r}{N}$	$b_1b_2b_5$	$\frac{r}{N}$
000000	0	0	0	00	0	000	0
000001	0	0	0	00	0	000	0
000010	1	0	0	10	0	100	0
000011	1	0	0	10	$\frac{2}{8}$	100	$\frac{4}{16}$
000100	2	0	0	00	0	010	0
000101	2	0	0	00	0	010	$\frac{2}{16}$
000110	3	0	0	10	0	110	0
000111	3	0	0	10	$\frac{2}{8}$	110	$\frac{6}{16}$
001000	0	0	0	00	0	000	0
001001	0	0	0	00	0	000	0
001010	1	0	0	10	0	100	0
001011	1	0	0	10	$\frac{2}{8}$	100	$\frac{4}{16}$
001100	2	0	0	00	0	010	0
001101	2	0	0	00	0	010	$\frac{2}{16}$
001110	3	0	0	10	0	110	0
001111	3	0	0	10	$\frac{2}{8}$	110	$\frac{6}{16}$
010000	0	0	0	00	0	000	0
010001	0	0	0	00	0	000	0
010010	1	0	0	10	0	100	0
010011	1	0	0	10	$\frac{2}{8}$	100	$\frac{4}{16}$
010100	2	0	0	00	0	010	0
010101	2	0	0	00	0	010	$\frac{2}{16}$
010110	3	0	0	10	0	110	0
010111	3	0	0	10	$\frac{2}{8}$	110	$\frac{6}{16}$
011000	0	0	0	00	0	000	0
011001	0	0	0	00	0	000	0
011010	1	0	0	10	0	100	0
011011	1	0	0	10	$\frac{2}{8}$	100	$\frac{4}{16}$
011100	2	0	0	00	0	010	0
011101	2	0	0	00	0	010	$\frac{2}{16}$
011110	3	0	0	10	0	110	0
011111	3	0	0	10	$\frac{2}{8}$	110	$\frac{6}{16}$

Table A-76 Twiddle factors of 1-D (16 × 4)-point FFT following the optimal algorithm at stage 5 to 6 (cont.)

Counter	Mapped to	Twiddle Fractions ($\frac{r}{N}$)					
		Stage 4		Stage 5		Stage 6	
$b_5b_4b_3b_2b_1b_0$	b_2b_1	$b_5b_4b_3$	$\frac{r}{N}$	$b_1b_5b_4b_3$	$\frac{r}{N}$	$b_1b_5b_4b_3b_2$	$\frac{r}{N}$
100000	0	1	0	01	0	001	0
100001	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$	001	$\frac{1}{16}$
100010	1	1	0	11	0	101	0
100011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	101	$\frac{5}{16}$
100100	2	1	0	01	0	011	0
100101	2	1	$\frac{1}{4}$	01	$\frac{1}{8}$	011	$\frac{3}{16}$
100110	3	1	0	11	0	111	0
100111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
101000	0	1	0	01	0	001	0
101001	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$	001	$\frac{1}{16}$
101010	1	1	0	11	0	101	0
101011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	101	$\frac{5}{16}$
101100	2	1	0	01	0	011	0
101101	2	1	$\frac{1}{4}$	01	$\frac{1}{8}$	011	$\frac{3}{16}$
101110	3	1	0	11	0	111	0
101111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
110000	0	1	0	01	0	001	0
110001	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$	001	$\frac{1}{16}$
110010	1	1	0	11	0	101	0
110011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	101	$\frac{5}{16}$
110100	2	1	0	01	0	011	0
110101	2	1	$\frac{1}{4}$	01	$\frac{1}{8}$	011	$\frac{3}{16}$
110110	3	1	0	11	0	111	0
110111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$
111000	0	1	0	01	0	001	0
111001	0	1	$\frac{1}{4}$	01	$\frac{1}{8}$	001	$\frac{1}{16}$
111010	1	1	0	11	0	101	0
111011	1	1	$\frac{1}{4}$	11	$\frac{3}{8}$	101	$\frac{5}{16}$
111100	2	1	0	01	0	011	0
111101	2	1	$\frac{1}{4}$	01	$\frac{1}{8}$	011	$\frac{3}{16}$
111110	3	1	0	11	0	111	0
111111	3	1	$\frac{1}{4}$	11	$\frac{3}{8}$	111	$\frac{7}{16}$

VITA

Pinit Kumhom was born on December 25, 1964 in RoiEt province, Thailand. He is a citizen of Thailand. In 1988, he received a Bachelor of Engineering (B.E.) in Electrical Engineering with second honor from King Mongkut's University of Technology Thonburi (KMUTT), Bangkok, Thailand. He started his career as an instructor in the Department of Electrical Engineering at KMUTT. His teaching experience included Electronic and Digital Designs and Microprocessor-based System. Since September, 1992, he has been studying in department of Electrical and Computer Engineering at Drexel University. He received his doctoral degree in Electrical engineering in March 2001. His research interests include the design of application-specific hardware, hardware/software codesign and system-on-chip (SOC) design.

1. P. Kumhom, J.R. Johnson, P. Nagvajara, "Design, Optimization and Implementation of a Universal FFT Processor", ASIC/SOC Conference, Arlington VA., September 2000.

2. J.R. Johnson, P. Kumhom, P. Nagvajara, R. Johnson, "Implementation and Optimization of a Distributed Memory FFT Processor", HPEC 2000, MIT, Cambrige MA, September 2000.