A Flexible Framework for Implementing FFT Processors

Michael A. Balog, III
and
Jeremy R. Johnson

Technical Report DU-CS-02-03
Department of Computer Science
Drexel University
Philadelphia, PA 19104
May 2002
© Copyright 2002

Michael A. Balog, III. All Rights Reserved.
# Table of Contents

List of Figures ......................................................... v

Abstract ................................................................. vii

1 Introduction ............................................................ 1

2 Dimensionless FFT ...................................................... 3
  2.1 Multidimensional DFT ............................................. 3
  2.2 Tensor Product Formulation ...................................... 4
  2.3 Fast Fourier Transform ........................................... 7
    2.3.1 One-Dimensional FFT ....................................... 7
    2.3.2 Dimensionless FFT .......................................... 19
  2.4 Matlab Implementation of the Dimensionless FFT ............. 26

3 Universal FFT Processor ............................................. 36
  3.1 Abstract FFT Processor .......................................... 36
    3.1.1 Address and Twiddle Generation ........................... 39
  3.2 Distributed Memory FFT Processor .............................. 41
    3.2.1 Matlab Simulation of Distributed Memory FFT Processor .. 47

4 Architecture Design ............................................... 52
  4.1 Interface Unit (IU) .............................................. 53
  4.2 Inter Connection Network (ICN) ................................. 54
  4.3 Processing Element (PE) ....................................... 55
    4.3.1 Computation Unit (CU) .................................... 56
    4.3.2 Twiddle Factor Generator (TG) ............................. 56
4.3.3 Address Generator(AG) ............................................. 57
4.3.4 Data Control Unit(DCU) ............................................. 58
4.3.5 Inter Connection Network Interface(ICN-IF) ..................... 60
4.3.6 Memory Interface(MEM-IF) ........................................... 60

5 Hardware Implantation ................................................. 61
5.1 Processing Element ................................................... 61
   5.1.1 The FFT Package .................................................. 61
   5.1.2 Cooley-Tukey Address Generator ................................ 62
   5.1.3 Cooley-Tukey Twiddle Generator ................................ 62
   5.1.4 Data and Address queues ........................................ 62
   5.1.5 Input and Output State Machines ............................... 63
   5.1.6 Inter Connection Network Interface ........................... 63
   5.1.7 Memory and Memory Interface .................................. 64
5.2 Inter Connetion Network ............................................. 64
5.3 FFT Machine Test Bench ............................................. 64

Bibliography .................................................................. 66

A Appendix: Hardware Implantation ..................................... 68
   A.1 Processing Element .................................................... 68
   A.2 FFT Package ............................................................ 80
   A.3 Cooley-Tukey Address Generator .................................. 81
   A.4 Cooley-Tukey Twiddle Generator .................................. 84
   A.5 Radix-2 Computation Unit .......................................... 88
   A.6 Data Control Unit - Input State Machine ....................... 90
   A.7 Data Control Unit - Output State Machine ..................... 98
   A.8 Inter-Connection Network Interface ............................ 103
   A.9 ICN Input Queue ..................................................... 107
<table>
<thead>
<tr>
<th>Section Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A.10 ICN Output Queue</td>
<td>110</td>
</tr>
<tr>
<td>A.11 Data Queue</td>
<td>112</td>
</tr>
<tr>
<td>A.12 Address Queue</td>
<td>114</td>
</tr>
<tr>
<td>A.13 Memory Interface</td>
<td>117</td>
</tr>
<tr>
<td>A.14 Inter Connection Network</td>
<td>118</td>
</tr>
<tr>
<td>A.15 FFT Machine Test Bench</td>
<td>120</td>
</tr>
</tbody>
</table>
List of Figures

2.1 Matlab implementation of bit-reversal .......................................................... 28
2.2 Matlab implementation of multi-dimensional bit-reversal ............................... 29
2.3 Matlab implementation of the Pease dimensionless FFT ............................... 30
2.4 Matlab implementation of Multi-dimensional twiddle factors ....................... 31
2.5 Matlab implementation of stride permutations .............................................. 32
2.6 Matlab implementation of the Cooley-Tukey dimensionless FFT .................. 33
2.7 Matlab standard basis generator .................................................................... 34
2.8 Matlab standard basis generator .................................................................... 35
3.1 16-point universal FFT machine .................................................................... 37
3.2 16-point universal Pease FFT machine .......................................................... 38
3.3 16-point universal Cooley-Tukey FFT machine ............................................. 38
3.4 16-point universal Pease FFT machine configured to compute 1D DFT ......... 39
3.5 16-point universal Pease FFT machine configured to compute 4 × 4 DFT ...... 40
3.6 Matlab address calculation for the Pease machine ........................................ 41
3.7 Matlab address calculation for the Cooley-Tukey machine ............................ 42
3.8 Matlab twiddle factor calculation for the Pease machine .............................. 43
3.9 Matlab twiddle factor calculation for the Cooley-Tukey machine ............... 44
3.10 Universal FFT Processor Architecture ......................................................... 46
3.11 Round robin schedule for universal FFT processor ..................................... 47
3.12 Matlab Implementation of the distributed Pease FFT Processor .................. 49
3.13 Matlab Implementation of the distributed Pease Processing Element .......... 50
4.1 Distributed Memory FFT Processor ............................................................... 52
4.2 Interface Unit .............................................................................................. 53
4.3 Processing Element ..................................................................................... 55
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.4</td>
<td>Basic Butterfly Operation</td>
<td>56</td>
</tr>
<tr>
<td>4.5</td>
<td>N number of Mux-N</td>
<td>57</td>
</tr>
<tr>
<td>4.6</td>
<td>Data Control Unit</td>
<td>59</td>
</tr>
</tbody>
</table>
Abstract
A Flexible Framework for Implementing FFT Processors
Michael A. Balog, III
Dr. Jeremy Johnson and Dr. Prawat Nagvajara

This thesis presents a framework for implementing a family of distributed memory fast Fourier transform (FFT) processors. A methodology is presented for mapping an FFT algorithm to a set of hardware components for address generation, twiddle factor generation, and memory configuration. The architecture is flexible in that it consists of cores that can be easily exchanged alternative implementations. Three levels of abstraction are presented to simplify verification and derivation of the instantiated processor.
1 Introduction

In this thesis we describe a systematic and flexible framework for implementing a universal fast Fourier transform (FFT) processor. The notion of a universal FFT processor was described in [7] and is based on the dimensionless FFT [9]. The dimensionless FFT is an algorithm which computes the discrete Fourier transform (DFT) on a fixed number of points independent of the dimension of the data. A dimensionless FFT can be used to compute different multi-dimensional DFTs simply by reordering the input data and changing the twiddle factors used by the FFT. This implies that it is possible to design a hardwired FFT processor that can be configured to compute different dimensional DFTs by changing the internal constants (twiddle factors) and be loading the data in a different order. By “hardwired” we mean that the internal dataflow (memory access pattern) and the corresponding control is fixed in the hardware. For a dimensionless FFT this hardwired control and communication pattern is fixed independent of dimension. This contrasts with other approaches (e.g. row-column algorithm [12] where different data access patterns are required for different dimensional DFTs.

The dimensionless FFT is not unique and consequently selecting the best algorithm for implementing a universal FFT processor becomes an optimization problem [1]. In previous work [7] a distributed memory architecture was proposed for implementing a family of dimensionless FFT algorithms. A high-level performance model was used to select the optimal dimensionless FFT for the implementation of the FFT processor. A prototype implementation of the selected design, using four processors was implemented using the Annapolis Microsystem Wildforce board with five Xilinx XC4085XLA FPGA processors [2]. This implementation was limited to computing one, two, and three dimensional DFTs and was designed specifically for the Wildforce board. Moreover, the entire design was restricted to the optimal algo-
algorithm that was selected by the performance model. Consequently it was not possible to experiment with different algorithms in hardware, nor was it possible to scale the implementation to use more processors or port the implementation to other hardware environments.

In this thesis we provide a framework that makes it easy to experiment with different implementations, scale to a larger number of processors, and to port to different hardware environments. In addition, the constraint on the dimension is removed. In order to be flexible, the processor is organized into a set of components with well defined specifications implemented as “cores” which can be easily replaced to try alternative implementation strategies and hardware technologies. This allows us to build a more general processor that supports the entire family of dimensionless FFT algorithms, or to optimize the implementation to a specific algorithm as was the case in [7].

The thesis is organized as follows. The first chapter discusses the DFT and the mathematics of the dimensionless FFT. In particular, we explicitly show how to compute the twiddle factors for different dimensionless FFT algorithms. The chapter concludes with a Matlab program to verify the correctness of different algorithms. The second chapter starts with a straightforward interpretation of the dimensionless FFT algorithm as a virtual machine for computing multi-dimensional DFTs. The universal FFT processor is obtained by folding the columns and rows of the virtual machine. An abstract of the resulting processor is specified and described using a Matlab simulation. The third chapter provides specifications for the hardware implementation of the processor along with suggested implementation approaches. The final chapter provides a sample hardware implementation written using the hardware description language VHDL.
2 Dimensionless FFT

This chapter reviews the dimensionless FFT [9]. The presentation of the dimensionless FFT follows that given in [8], except that an in-place variant is used. The FFT is described and derived using matrix factorizations [5, 11, 12] and the notation and presentation follows [5]. Proofs for most of the theorems are not given as they can be obtained from these references; instead examples are provided that clarify the theorems and illustrate the main ideas. Proofs are given in those cases where the computations have not previously appeared in the literature.

2.1 Multidimensional DFT

Let \(X(a_1, \ldots, a_t)\) be a complex function of \(t\) variables where \((0 \leq a_i < n_i)\) for \(i = 1, \ldots, t\). The \(n_1 \times \cdots \times n_t\) \(t\)-dimensional DFT \(Y = F_{(n_1, \ldots, n_t)}X\) is defined by the following equation.

\[
Y(b_1, \cdots, b_t) = \sum_{0 \leq a_1 < n_1} \cdots \sum_{0 \leq a_t < n_t} W_{n_1}^{a_1 b_1} \cdots W_{n_t}^{a_t b_t} X(a_1, \ldots, a_t),
\]

for \((0 \leq b_j < n_j)\) and \(j = (0, \cdots, t)\)

This sum can be written as a nested sum which reduces the computation of \(F_{(n_1, \ldots, n_t)}\) to the computation of one-dimension DFTs.

\[
Y(b_1, \cdots, b_t) = \sum_{0 \leq a_1 < n_1} W_{n_1}^{a_1 b_1} \left( \cdots \left( \sum_{0 \leq a_t < n_t} W_{n_t}^{a_t b_t} X(a_1, \ldots, a_t) \right) \right)
\]

The computation in Equation 2.2 proceeds in \(t\) stages. In the \(i\)-th stage, \(N/n_{t-i} n_{t-i}\)-point DFTs are applied to the result of the previous stage. If \(Y_{t-1}\) contains the result of the \((i - 1)\)-st stage \((Y_0 = X)\), an \(n_{t-i}\)-point DFT is applied to each subvector of \(Y_{t-1}(a_1, \ldots, a_t)\) obtained by fixing \(a_1, \ldots, a_{t-i-1}, a_{t-i+1}, \ldots, a_t\) and varying \(0 \leq a_{t-i} < n_{t-i}\). In two-dimensions, if the data is viewed as a \(n_1 \times n_2\) array, the first stage applies \(n_2\)-point DFTs to the rows of the input array and the second stage applies \(n_1\)-point
DFTs to the columns of the array obtained from the first stage. Consequently, this algorithm is commonly called the “row-column” algorithm.

2.2 Tensor Product Formulation

In order to derive and manipulate FFT algorithms it is beneficial to work with a matrix formulation of the multi-dimensional DFT. This involves a matrix construction called the tensor product (also called the Kronecker product).

**Definition 1 (Tensor Product)** Let

\[
A = \begin{pmatrix}
a_{1,1} & \cdots & a_{1,n_1} \\
\vdots & \ddots & \vdots \\
a_{m_1,1} & \cdots & a_{m_1,n_1}
\end{pmatrix}
\quad \text{and} \quad
B = \begin{pmatrix}
b_{1,1} & \cdots & b_{1,n_2} \\
\vdots & \ddots & \vdots \\
b_{m_2,1} & \cdots & b_{m_2,n_2}
\end{pmatrix}
\]

The tensor product, \( A \otimes B \), of \( A \) and \( B \) is the block matrix obtained by replacing each element of the matrix \( A \) by the matrix \( a_{i,j}B \).

\[
A \otimes B = \begin{pmatrix}
a_{1,1}B & \cdots & a_{1,n_1}B \\
\vdots & \ddots & \vdots \\
a_{m_1,1}B & \cdots & a_{m_1,n_1}B
\end{pmatrix}
\]

Multi-dimensional DFTs can be represented by a tensor product of DFT matrices.

**Definition 2 (Fourier Matrix)** The Discrete Fourier Transform (DFT) of vector \( x \) of size \( N \) is the matrix-vector product \( F_Nx \), where

\[
F_N = (\omega_N^{jk}), \quad \text{where} \quad \omega_N = e^{(2\pi i j k/N)}, \quad 0 \leq j, k < N,
\]

(2.3) is the \( N \times N \) discrete Fourier matrix.

**Theorem 1 (Multi-dimensional DFT)** Let \( X(a_1, \cdots, a_t) \) be a complex function of \( t \) variables where \( 0 \leq a_i < n_i \) for \( i = 1, \ldots, t \), and let \( x \) be the vector of size
$N = n_1 \cdots n_t$ obtained from the function $X$ by ordering the values $X(a_1, \ldots, a_t)$ lexicographically. If $Y = F_{(n_1, \ldots, n_t)}X$ and $y$ is the lexicographically ordered vector of values of $Y$, then

$$y = (F_{n_1} \otimes \cdots \otimes F_{n_t})x.$$  

**Proof:** This theorem follows from the definition of the multi-dimensional DFT and the tensor product. When $t = 2$, fix $a_2$ and $b_2$ and observe, using Matlab notation, $Y(b_2,:)=\omega_{n_1}^{a_2h_2}F_{n_2}X(a_2,:)$. Therefore, ordering $X$ and $Y$ lexicographically, the result follows from the definition of the tensor product. The general case follows from induction on the dimension $T$.

**Example:** Let $Y = F_{(2,2)}X$. Then $Y(b_1, b_2) = \sum_{a_1=0}^{1} \sum_{a_2=0}^{1} \omega_{n_1}^{a_2h_2} \omega_{n_2}^{a_1h_1}X(a_1, a_2)$. Let $\mathbf{x}$ and $\mathbf{y}$ be the vectors of size 4 obtained by ordering the elements of $X$ and $Y$ lexicographically; that is

$$\mathbf{x} = \begin{pmatrix} X(0,0) \\ X(0,1) \\ X(1,0) \\ X(1,1) \end{pmatrix}, \quad \mathbf{y} = \begin{pmatrix} Y(0,0) \\ Y(0,1) \\ Y(1,0) \\ Y(1,1) \end{pmatrix}.$$  

Then

$$\mathbf{y} = \begin{pmatrix} Y(0,0) \\ Y(0,1) \\ Y(1,0) \\ Y(1,1) \end{pmatrix} = \begin{pmatrix} X(0,0) + X(0,1) + X(1,0) + X(1,1) \\ X(0,0) - X(0,1) + X(1,0) - X(1,1) \\ X(0,0) + X(0,1) - X(1,0) - X(1,1) \\ X(0,0) - X(0,1) - X(1,0) + X(1,1) \end{pmatrix} = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -1 & 1 & -1 \\ 1 & 1 & -1 & -1 \\ 1 & -1 & -1 & 1 \end{pmatrix} \begin{pmatrix} X(0,0) \\ X(0,1) \\ X(1,0) \\ X(1,1) \end{pmatrix}$$
\[
\begin{pmatrix}
1 & 1 \\
1 & -1
\end{pmatrix}
\otimes
\begin{pmatrix}
1 & 1 \\
1 & -1
\end{pmatrix}
\begin{pmatrix}
X(0,0) \\
X(0,1) \\
X(1,0) \\
X(1,1)
\end{pmatrix}
= (F_2 \otimes F_2)x.
\]

The tensor product satisfies many properties that can be used in deriving and manipulating algorithms for computing the DFT.

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

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.

Property 1.11 implies that we can compute $y = (F_{n_1} \otimes F_{n_2})x$, by first computing $t = (I_{n_1} \otimes F_{n_2})x$ and then computing $y = (F_{n_1} \otimes I_{n_2})t$. This corresponds to the row-column algorithm. Using indication we obtain the following matrix version of the multi-dimensional generalization of the row-column algorithm.

**Theorem 2 (Row-Column Algorithm)** Let $\mathcal{F}_N$ be the $t$-dimensional DFT $(F_{n_1} \otimes \cdots \otimes F_{n_t})$.

$$\mathcal{F}_N = \prod_{i=1}^{t} \left( I_{N(i-1)} \otimes F_{n_i} \otimes I_{\tilde{N}(i)} \right),$$

(2.4)

where $N(i) = n_1 \cdots n_i$ and $\tilde{N}(i) = N/N(i)$ with the following boundary conditions $N(0) = 1$ and $N(t) = N$.

### 2.3 Fast Fourier Transform

In this section we review the mathematics behind the FFT using the ideas presented in [5, 11, 12]. The notation used is from [5].

#### 2.3.1 One-Dimensional FFT

The FFT can be viewed as a factorization of the DFT matrix into a product of sparse structured matrices. The $O(N \log(N))$ computing time can be seen from the fact that each matrix in the factorization can be computed in $O(N)$ time and there are $O(\log(N))$ factors. Before reviewing the factorization, a class of permutation and diagonal matrices arising in the factorization need to be defined.

**Definition 3 (Standard Basis)** Let $\mathbb{C}^N$ denote the vector space of $N$-tuples of complex numbers. The standard basis for $\mathbb{C}^N$ is the set of vectors

$$\{e_i^N \mid 0 \leq i < N\},$$

(2.5)
where $e_i^N$ is the vector with 1 in the $i^{th}$ component and zeros elsewhere.

When performing computations with tensor products it is convenient to view the elements of the standard basis as tensor products of standard basis vectors. The following property describes the tensor product of basis vectors.

**Property 2 (Tensor Product of Standard Basis)**

\[ e_i^p \otimes e_j^q = e_{qi+j}^{pq}; \quad 0 \leq i < p, 0 \leq j < q. \]  

(2.6)

More generally,

\[ e_{i_1}^{n_1} \otimes \cdots \otimes e_{i_t}^{n_t} = e_{i_1n_2\cdots n_t+i_{t-1}n_{t-1}+\cdots+i_1}^{n_1\cdots n_t}; \quad 0 \leq i_j < n_j. \]  

(2.7)

A permutation matrix can be defined by permuting the elements of the standard basis.

**Definition 4 (Permutation Matrix)**

Let $\sigma$ be a permutation of the set \{0, \ldots, N - 1\} denoted by

\[
\sigma = \begin{pmatrix}
0 & 1 & \cdots & N - 1 \\
\sigma(0) & \sigma(1) & \cdots & \sigma(N - 1)
\end{pmatrix}.
\]

The permutation matrix $P_\sigma$ is the change of basis matrix obtained by permuting the standard basis elements corresponding to $\sigma$.

\[ P_\sigma(e_i^N) = e_{\sigma(i)}^N. \]

The inverse of $\sigma$ is the permutation $\sigma^{-1}$ that maps $\sigma(i)$ to $i$, and $P_{\sigma^{-1}} = P_\sigma^{-1} = P_\sigma^T$.

The product of two permutations $\sigma\tau$ maps $i$ to $\sigma(\tau(i))$, and $P_{\sigma\tau} = P_\sigma P_\tau$. Observe that

\[ P_\sigma x = P_\sigma(\sum_{i=0}^{N-1} x_i e_i^N) = \sum_{i=0}^{N-1} x_i e_{\sigma(i)}^N = \sum_{i=0}^{N-1} x_{\sigma^{-1}(i)} e_i^N. \]

**Example:** Let

\[
\sigma = \begin{pmatrix}
0 & 1 & 2 & 3 \\
1 & 2 & 3 & 0
\end{pmatrix} \quad \text{and} \quad \sigma^{-1} = \begin{pmatrix}
0 & 1 & 2 & 3 \\
3 & 0 & 1 & 2
\end{pmatrix}.
\]
Then

\[
P_\sigma = \begin{pmatrix}
0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0
\end{pmatrix}, \quad P_{\sigma^{-1}} = \begin{pmatrix}
0 & 1 & 0 & 0 \\
0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 \\
1 & 0 & 0 & 1
\end{pmatrix}
\text{ and } P_\sigma \begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3
\end{pmatrix} = \begin{pmatrix}
x_3 \\
x_0 \\
x_1 \\
x_2
\end{pmatrix}.
\]

An important class of permutation matrices, that arise in FFT algorithms, are obtained by permuting the factors in the tensor product of basis vectors. These permutations are called digit permutations because the permute the digits of the multi-radix number given in Equation 2.7.

**Definition 5 (Digit Permutation)**

Let \( N = N_1 \cdots N_t \) and \( \sigma \) be a permutation of the numbers \( 1, \ldots, t \).

\[
P_\sigma^{(N_1, \ldots, N_t)} (e_{i_1}^{N_1} \otimes \cdots \otimes e_{i_t}^{N_t}) = e_{i_{\sigma(1)}}^{N_{\sigma(1)}} \otimes \cdots \otimes e_{i_{\sigma(t)}}^{N_{\sigma(t)}}.
\]

The load-stride permutation is a digit permutation that occurs in the FFT.

**Definition 6 (Load-stride Permutation Matrix)** The load-stride permutation, \( L_{rst}^{rs} \), is the tensor permutation matrix of size \( rs \times rs \) defined as following.

\[
L_{rs}^{rs} (e_i^r \otimes e_j^s) = e_j^s \otimes e_i^r
\]  \hspace{1cm} (2.8)

**Property 3**

1. \( L_{s}^{rst} L_{t}^{rst} = L_{st}^{rst} \).

2. \( L_{n}^{N} L_{N/n}^{N} = I_{N} \).

3. \( L_{rst}^{rs} = (L_{t}^{rt} \otimes I_{s})(I_{r} \otimes L_{t}^{st}) \).
Example: Let $\sigma$ be the permutation of size 3 with $\sigma(0) = 2$, $\sigma(1) = 0$, and $\sigma(2) = 1$. Then $L_2^8 = P_\sigma^{(2,2,2)}$.

$$L_2^8 = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1
\end{pmatrix} \quad \text{and} \quad 
\begin{pmatrix}
x_0 \\
x_2 \\
x_4 \\
x_6 \\
x_1 \\
x_3 \\
x_5 \\
x_7
\end{pmatrix} = L_2^8 
\begin{pmatrix}
x_0 \\
x_1 \\
x_2 \\
x_3 \\
x_4 \\
x_5 \\
x_6 \\
x_7
\end{pmatrix}.$$  

The twiddle factor matrix is a diagonal matrix that occurs in the FFT.

Definition 7 (Twiddle Factor Matrix)

The diagonal matrix $T^{rs}_s$ is defined as follows.

$$T^{rs}_s(e^r_i \otimes e^s_j) = \omega^{ij}_{rs}(e^r_i \otimes e^s_j), \quad 0 \leq i < r, \quad 0 \leq j < s \quad (2.9)$$

$$T^{rs}_s = \bigoplus_{i=0}^{r-1} \bigoplus_{j=0}^{s-1} \omega^{ij}_{rs} = \bigoplus_{i=0}^{r-1} (W_s(\omega_{rs}))^i \quad (2.10)$$

where

$$W_n(\alpha) = \text{diag}(1, \alpha, \ldots, \alpha^{n-1}) \quad (2.11)$$

The following theorem is the basis of the divide and conquer step in the FFT.

Theorem 3 (Cooley-Tukey)

Let $N = rs$. Then the DFT matrix of size $N$ can be factored into a product of matrices involving DFTs of size $r$ and $s$.

$$F_N = (F_r \otimes I_s)T^{rs}_s(I_r \otimes F_s)\quad (2.12)$$

Applying this theorem recursively to $F_r$ and $F_s$ leads to a recursive FFT algorithm. Using properties of the tensor product, the recursive algorithm can be converted into an iterative algorithm.
Example: [Derivation of an iterative 16-point FFT] Applying the Cooley-Tukey theorem we obtain.

\[ F_{16} = (F_2 \otimes I_8)T_8^{16}(I_2 \otimes F_8)L_2^{16}, \]
\[ F_8 = (F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_2^8, \]
\[ F_4 = (F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4. \]

Substituting the factorization for \( F_8 \) into the factorization for \( F_{16} \) and using properties of the tensor product gives

\[ F_{16} = (F_2 \otimes I_8)T_8^{16}(I_2 \otimes ((F_2 \otimes I_4)T_4^8(I_2 \otimes F_4)L_2^8)L_2^{16}) \]
\[ = (F_2 \otimes I_8)T_8^{16}(I_2 \otimes F_2 \otimes I_4)(I_2 \otimes T_2^4)(I_4 \otimes F_4)(I_2 \otimes T_2^4)L_2^{16}. \]

Substituting for \( F_4 \) produces

\[ F_{16} = (F_2 \otimes I_8)T_8^{16}(I_2 \otimes F_2 \otimes I_4)(I_2 \otimes T_2^4) \]
\[ (I_4 \otimes (F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)L_2^4)(I_2 \otimes L_2^8)L_2^{16} \]
\[ = (F_2 \otimes I_8)T_8^{16}(I_2 \otimes F_2 \otimes I_4)(I_2 \otimes T_2^4) \]
\[ (I_4 \otimes F_2 \otimes I_2)(I_4 \otimes T_2^4)(I_8 \otimes F_2)(I_4 \otimes L_2^4)(I_2 \otimes L_2^8)L_2^{16}. \]

Since factors of the form \((I_m \otimes F_2 \otimes I_n)D\), where \( D = (I_m \otimes T_n^{2n}) \) can be implemented using a doubly nested loop, this factorization leads to a program with a triply nested loop called the iterative FFT.

Since the product of permutations is a permutation and the tensor product of permutations is a permutation, \((I_4 \otimes L_2^4)(I_2 \otimes L_2^8)L_2^{16}\) is a permutation. Applying this permutation to \( e_{i_3}^2 \otimes e_{i_2}^2 \otimes e_{i_1}^2 \otimes e_{i_0}^2 \) we see that \( e_{i_08+i_24+i_12+i_0}^{16} \) gets mapped to \( e_{i_08+i_14+i_22+i_3}^{16} \). For this reason, it is called bit-reversal.
Definition 8 (Digit-Reversal Permutation Matrix)

Let $N = N_1 \times \cdots \times N_t$. Then, the digit-reversal permutation, $R^{(N_1 \times \cdots \times N_t)}$, is defined as following.

$$R^{(N_1 \times \cdots \times N_t)} e_i^{N_1} \otimes \cdots \otimes e_i^{N_t} = e_i^{N_1} \otimes \cdots \otimes e_i^{N_t} \quad (2.13)$$

For $N = 2 \times \cdots \times 2 = 2^t$, digit reversal is called bit-reversal since it reverses the bits in the binary representation of the indices. In this case we will simply use the notation $R_N$.

Generalizing Example 2.3.1 leads to the following theorem.

Theorem 4 (Single-Dimension Radix-2 FFT) Let $N = 2^K$ denoting radix-2 FFT and $R_N$ being the $K$-digit, address bit-reversal permutation.

$$F_N = \left\{ \prod_{i=1}^K (I_{2^{i-1}} \otimes F_2 \otimes I_{2^{K-i}}) T_i \right\} R_N, \quad (2.14)$$

where $T_i = (I_{2^{i-1}} \otimes T_{2^{K-i+1}}^{2^{K-i}})$.

This factorization can be grouped into $K$ stages, where in each stage $N/2$ butterfly operations are performed. A butterfly operation computes $F_2$ combined with multiplication by a twiddle factor.

Definition 9 (Butterfly Operation)

$$\begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} 1 & 0 \\ 0 & \alpha \end{pmatrix}, \text{ where } \alpha = \omega_N^j \text{ for some } j.$$ 

Prior to the first stage, the data is permuted by the bit-reversal permutation.

Example: [Example of Single-Dimension Radix-2 FFT] Breaking the last example into stages and determining the values of the twiddle factors and bit-reversal permutation leads to the following calculations.

$$F_{16} = (F_2 \otimes I_8) T_1$$
\[(I_2 \otimes F_2 \otimes I_4)T_2\]
\[(I_4 \otimes F_2 \otimes I_2)T_3\]
\[(I_8 \otimes F_2)T_4\]

\[R_{16}\]
\[T_1 = T_{16}^8\]
\[= \text{diag}(1, 1, 1, 1, 1, 2, \omega_{16}^1, \omega_{16}^2, \omega_{16}^3, \omega_{16}^4, \omega_{16}^5, \omega_{16}^6, \omega_{16}^7)\]
\[T_2 = (I_2 \otimes T_4^8)\]
\[= \text{diag}(1, 1, 1, 1, 1, 2, \omega_{8}^1, \omega_{8}^2, \omega_{8}^3, 1, 1, 1, 1, 1, \omega_{8}^1, \omega_{8}^2, \omega_{8}^3)\]
\[T_3 = (I_4 \otimes T_2^4)\]
\[= \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, \omega_{4}^1, \omega_{4}^2)\]
\[T_4 = I_{16}\]
\[= \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)\]
\[R_{16} = (I_4 \otimes L_2^4)(I_2 \otimes L_2^8)L_{2}^{16}\]
\[= \begin{pmatrix}
0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\
0 & 8 & 4 & 12 & 2 & 10 & 6 & 14 & 1 & 9 & 5 & 13 & 3 & 11 & 7 & 15
\end{pmatrix}\]

**Parallel In-Place FFT**

In preparation for our FFT processor it is convenient to rewrite the FFT in a canonical form where the addressing for the butterfly operations is given explicitly as a permutation. The algorithms we consider will be in-place, which means that each butterfly operation stores the result in the same two locations where the inputs were obtained. The FFT factorization will consist of \(\log(N)\) stages of the form \(P^{-1}(I_{N/2} \otimes F_2)TP\), where \(P\) is a permutation and \(T\) is a diagonal matrix with all of the odd elements equal to 1. Since the inverse of \(P\) occurs on the left, the implied algorithm is in-place.

**Definition 10 (Parallel In-Place FFT)** Assume \(N = 2^K\), then a canonical in-

The Fast Fourier Transform (FFT) algorithm is given by the factorization

\[ F_N = \left\{ \prod_{i=1}^{K} \left( P_i^{-1} (I_{2^{K-i}} \otimes F_2) T_i^i P_i \right) \right\} R, \tag{2.15} \]

where the internal permutations \( P_i, i = 1, \ldots, K \) and the initial permutation \( R \) are digit permutations, and \( T_i, i = 1, \ldots, K \), are diagonal matrices with elements in the odd diagonal locations are all equal to 1.

Many such factorizations with different \( P_i \) and \( T_i \) can be obtained from Theorem 4 through the use of the following theorem.

**Theorem 5** (Commutation) If \( A \) is an \( m \times m \) matrix, and \( B \) is an \( n \times n \) matrix, then

\[ A \otimes B = L_m^{m,n} (B \otimes A)L_n^{m,n}. \tag{2.16} \]

More generally, if \( A_i \) is a \( n_i \times n_i \) matrix then

\[ (A_1 \otimes \cdots \otimes A_i) = P_{\sigma(1), \ldots, \sigma(i)}^{(n_1, \ldots, n_i)}(A_{\sigma(1)} \otimes \cdots \otimes A_{\sigma(i)}) P_{\sigma}^{(n_1, \ldots, n_i)}, \]

where \( P_{\sigma}^{(n_1, \ldots, n_i)} \) is a digit permutation.

Using this theorem the factors \( I_{2^{K-i}} \otimes F_2 \otimes I_{2^{K-i}} \) occurring in Theorem 4 can be commuted to the form \( I_{2^{K-i}} \otimes F_2 \) using any digit permutation that \( F_2 \) to the rightmost location (i.e. \( P_{\sigma}^{(2, \ldots, 2)} \), with \( \sigma(K) = i \)). This leads to factors of the form \( P^{-1}(I_{2^{K-i}} \otimes F_2) P T \). These factors can be rewritten as \( P^{-1}(I_{2^{K-i}} \otimes F_2) T' P \), allowing the twiddle factor computation to be combined with \( F_2 \) to obtain butterfly operations, using the following theorem.

**Theorem 6** Let \( P_{\sigma} \) be a permutation matrix of size \( N \) and \( D = \text{diag}(d_1, \ldots, d_N) \) be a diagonal matrix. Then \( P_{\sigma}D = D' P_{\sigma} \), where \( D' = \text{diag}(d_{\sigma(1)}, \ldots, d_{\sigma(N)}) \).

Different permutations lead to different data access patterns with potentially different performance. There are \((K - 1)!\) such permutations. In the next two sections we explicitly compute two of the possible algorithms. The procedure used in the examples can easily be used for any of the \((K - 1)!\) algorithms obtained in this way.
Pease Algorithm

The in-place version of the Pease algorithm [10] is an example of Definition 10 where
\( P_i = L_{2k-1} \). It is similar to the algorithm described by Pease except that by not
combining permutations between stages an in-place algorithm is retained. The twiddle
factor computations shown are from [4].

**Theorem 7 (Pease Algorithm)** Assume \( N = 2^K \). Then

\[
F_N = \left\{ \prod_{i=1}^{K} L_{2i}^N \left( I_{N/2} \otimes F_2 \right) T_i' L_{2k-1}^N \right\} R_N, \tag{2.17}
\]

where

\[
T_i' = L_{2k-1}^N (I_{2^i-1} \otimes T_{2k-1}^{2K-i+1} L_{2k-1}^N) \tag{2.18}
\]

**Proof:** Starting with Theorem 4

\[
F_N = \left\{ \prod_{i=1}^{K} (I_{2k-1} \otimes F_2 \otimes I_{2k-1}) T_i \right\} R_N,
\]

where \( T_i = (I_{2^i-1} \otimes T_{2k-1}^{2K-i+1}) \). Applying the commutation theorem (5) to \( (I_{2k-1} \otimes F_2 \otimes I_{2k-1}) \) this becomes

\[
F_N = \left\{ \prod_{i=1}^{K} L_{2i}^N (I_{N/2} \otimes F_2) L_{2k-1}^N T_i \right\} R_N.
\]

By Theorem 6 and Property 3.2

\[
F_N = \left\{ \prod_{i=1}^{K} L_{2i}^N (I_{N/2} \otimes F_2) T_i' L_{2k-1}^N \right\} R_N,
\]

where \( L_{2k-1}^N T_i = T_i' L_{2k-1}^N \). Consequently, \( T_i' = L_{2k-1}^N T_i L_{2k-1}^N \).

The twiddle factors in the Pease algorithm are made explicit by the following corollary.

**Corollary:** [Twiddle Factors for Pease Algorithm]

\[
T_i' = \bigoplus_{a=0}^{r-1} \bigoplus_{b=0}^{s-1} \bigoplus_{c=0}^{1} \omega_{2r}^a \omega_{2s}^b \omega_{2c}^c, \quad \text{where} \ r = 2^{K-i} \text{ and } s = 2^{i-1} \tag{2.19}
\]

Since \( \omega_{2r} = \omega_N^a \), \( \omega_{2s}^b \), and \( \omega_{2c}^c \) can be replaced by \( \omega_N^{a+b+c} \) which allows all of the twiddle factors
to be written as \( N \)-th roots of unity.
Proof: Let \( r = 2^{K-i} \), \( s = 2^{i-1} \), and \( N = 2rs = 2^K \). Then,

\[
T_i = (I_{2^{i-1}} \otimes T_{2^{K-i+1}}^{2^{K-i}})
\]
\[
T_i' = L_{2^{K-i}}^{2^{K}}(I_{2^{i-1}} \otimes T_{2^{K-i+1}}^{2^{K-i+1}})L_{2^{K-i}}^{2^{K}}
\]
\[
= L_{2^{r}}^{2^{r}}(I_{s} \otimes T_{r}^{2^{r}})L_{2^{s}}^{2^{r}}
\]

and

\[
T_i'(e_a^r \otimes e_b^s \otimes e_c^2) = L_{2^{r}}^{2^{r}}(I_{s} \otimes T_{r}^{2^{r}})L_{2^{s}}^{2^{r}}(e_a^r \otimes e_b^s \otimes e_c^2)
\]
\[
= L_{2^{r}}^{2^{r}}(I_{s} \otimes T_{r}^{2^{r}})(e_b^s \otimes e_c^2 \otimes e_a^r)
\]
\[
= L_{2^{r}}^{2^{r}}(I_{s}e_b^s \otimes T_{r}^{2^{r}}(e_c^2 \otimes e_a^r))
\]
\[
= \omega_{2^{r}}^{ac}L_{2^{r}}^{2^{r}}(e_b^s \otimes e_c^2 \otimes e_a^r)
\]
\[
= \omega_{2^{r}}^{ac}(e_a^r \otimes e_b^s \otimes e_c^2).
\]

Therefore,

\[
T_i' = \bigoplus_{a=0}^{r-1} \bigoplus_{b=0}^{s-1} \bigoplus_{c=0}^{1} \omega_{2^{r}}^{ac}, \text{ where } r = 2^{K-i} \text{ and } s = 2^{i-1}.
\]

Example: [Pease Algorithm]

\[
F_{16} = L_{2}^{16}(I_{8} \otimes F_{2})T_{4} L_{8}^{16}
\]
\[
L_{4}^{16}(I_{8} \otimes F_{2})T_{2} L_{8}^{16}
\]
\[
L_{8}^{16}(I_{8} \otimes F_{2})T_{1} L_{2}^{16}
\]
\[
L_{16}^{16}(I_{8} \otimes F_{2})T_{1} L_{1}^{16}
\]
\[
R_{16}
\]

The twiddle factors \( T_i' \) are,

\[
T_1' = L_{8}^{16}T_{1}^{16}L_{2}^{16}
\]
\[
= \bigoplus_{a=0}^{7} \bigoplus_{b=0}^{0} \bigoplus_{c=0}^{1} \omega_{16}^{ac} = \text{diag}(1, 1, 1, \omega_{16}^{1}, 1, \omega_{16}^{2}, 1, \omega_{16}^{3}, 1, \omega_{16}^{4}, 1, \omega_{16}^{5}, 1, \omega_{16}^{6}, 1, \omega_{16}^{7})
\]
\[ T'_2 = L_4^{16}(I_2 \otimes T^6_4)L_4^{16} \]
\[ = \bigoplus_{a=0}^{3} \bigoplus_{b=0}^{1} \bigoplus_{c=0}^{1} \omega_{16}^{ac} = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]
\[ T'_3 = L_2^{16}(I_4 \otimes T^4_2)L_2^{16} \]
\[ = \bigoplus_{a=0}^{3} \bigoplus_{b=0}^{1} \bigoplus_{c=0}^{1} \omega_{16}^{ac} = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]
\[ T'_4 = I_{16} \]
\[ = \bigoplus_{a=0}^{0} \bigoplus_{b=0}^{7} \bigoplus_{c=0}^{1} \omega_{16}^{ac} = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

**Cooley-Tukey Algorithm**

The in-place version of the Cooley-Tukey algorithm [3] is an example of Definition 10 where \( P_i = (I_{2^j-1} \otimes L_{2^{2K-1-i}}) \). This algorithm is standard radix-two iterative FFT with the addressing made explicit with the insertion of permutations.

**Theorem 8 (Cooley-Tukey Algorithm)** Assume \( N = 2^K \). Then

\[
F_N = \left\{ \prod_{i=1}^{K} (I_{2^j-1} \otimes L_{2^{2K-1-i}})(I_{N/2} \otimes F_2)T'_i(I_{2^j-1} \otimes L_{2^{2K-1-i}}) \right\} R_N, \quad (2.20)
\]

where

\[
T'_i = (I_{2^j-1} \otimes L_{2^{2K-1-i}})(I_{2^{j-1}} \otimes T_{2^i-1}^{2K-1-i})(I_{2^{j-1}} \otimes L_{2^{2j-1}}^{2K-1-i}). \quad (2.21)
\]

**Proof:** [Cooley-Tukey Algorithm] The proof is identical to the proof of Theorem 7 except that the Commutation theorem is applied to the factors \((F_2 \otimes I_{2^{2K-1}})\) rather than \((I_{2^{2K-1}} \otimes F_2 \otimes I_{2^{2K-1}})\).

**Corollary:** [Twiddle Factors for Cooley-Tukey Algorithm]

\[
T'_i = \bigoplus_{a=0}^{s-1} \bigoplus_{b=0}^{r-1} \bigoplus_{c=0}^{1} \omega_{2^r}^{ab}, \quad \text{where } r = 2^{K-i} \text{ and } s = 2^{i-1} \quad (2.22)
\]

**Proof:** [Twiddle Factors for Cooley-Tukey Algorithm] Let \( r = 2^{K-i}, s = 2^{i-1}, \) and \( N = 2rs = 2^K \). Then,

\[
T_i = (I_2 \otimes L_{2^{2K-1-i}})
\]
\[
T'_i = (I_{2^{j-1}} \otimes L_{2^{2K-1-i}})(I_{2^{j-1}} \otimes T_{2^i}^{2K-1-i})(I_{2^{j-1}} \otimes L_{2^{2j-1}}^{2K-1-i})
\]
\[
= (I_s \otimes L_r^s)(I_s \otimes T_r^s)(I_s \otimes L_2^r),
\]
and

\[
T'_i (e^*_a \otimes e^*_b \otimes e^2_c) = (I_s \otimes L^s_r)(I_s \otimes T^s_r)(I_s \otimes L^s_2)(e^*_a \otimes e^*_b \otimes e^2_c)
\]
\[
= (I_s \otimes L^s_r)(I_s \otimes T^s_r)(I_s e^*_a \otimes L^s_2 (e^*_b \otimes e^2_c))
\]
\[
= (I_s \otimes L^s_r)(I_s \otimes T^s_r)(e^*_a \otimes e^2_c \otimes e^*_b)
\]
\[
= (I_s \otimes L^s_r)(I_s e^*_a \otimes T^s_r (e^2_c \otimes e^*_b))
\]
\[
= \omega^{c,b}_{2^r}(I_s \otimes L^s_r)(e^*_a \otimes e^2_c \otimes e^*_b)
\]
\[
= \omega^{c,b}_{2^r}(I_s e^*_a \otimes L^s_r (e^2_c \otimes e^*_b))
\]
\[
= \omega^{c,b}_{2^r}(e^*_a \otimes e^*_b \otimes e^2_c)
\]

Consequently,

\[
T'_i = \bigoplus_{a=0}^{s-1} \bigoplus_{b=0}^{r-1} \omega^{c,b}_{2^r}, \text{ where } r = 2^{K-i} \text{ and } s = 2^{l-1}.
\]

**Example:** [Cooley-Tukey Algorithm]

\[
F_{16} = L_{16}^{16}(I_8 \otimes F_2)T'_1 L_{16}^{16}
\]
\[
(I_2 \otimes L_{2}^8)(I_8 \otimes F_2)T'_2 (I_2 \otimes L_{4}^8)
\]
\[
(I_4 \otimes L_{2}^4)(I_8 \otimes F_2)T'_3 (I_4 \otimes L_{1}^4)
\]
\[
(I_8 \otimes L_{2}^2)(I_8 \otimes F_2)T'_4 (I_8 \otimes L_{1}^2)
\]
\[
R_{16}
\]

The twiddle factors \( T'_i \) are,

\[
T'_1 = L_{16}^{16}(T_{16}^8)L_{2}^{16}
\]
\[
= \bigoplus_{a=0}^{7} \bigoplus_{b=0}^{1} \omega_{16}^{c,b} = \text{diag}(1, 1, 1, \omega_{16}^1, 1, \omega_{16}^2, 1, \omega_{16}^3, 1, \omega_{16}^4, 1, \omega_{16}^5, 1, \omega_{16}^6, 1, \omega_{16}^7)
\]

\[
T'_2 = (I_2 \otimes L_{4}^8)(I_2 \otimes T_{4}^8)(I_2 \otimes L_{2}^8)
\]
\[
= \bigoplus_{a=0}^{3} \bigoplus_{b=0}^{1} \omega_{8}^{c,b} = \text{diag}(1, 1, 1, \omega_{8}^1, 1, \omega_{8}^2, 1, \omega_{8}^3, 1, 1, 1, \omega_{8}^1, 1, \omega_{8}^2, 1, \omega_{8}^3)\]
\[ T_3' = \left( I_4 \otimes L_2^3 \right) \left( I_4 \otimes T_2^4 \right) \left( I_4 \otimes L_2^4 \right) \]
\[ = \bigoplus_{a=0}^3 \bigoplus_{b=0}^1 \bigoplus_{c=0}^1 \omega_4^{abc} = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]
\[ T_4' = \left( I_8 \otimes L_2^2 \right) \left( I_{16} \otimes L_2^2 \right) \]
\[ = \bigoplus_{a=0}^7 \bigoplus_{b=0}^1 \bigoplus_{c=0}^1 \omega_4^{abc} = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]
\[ R_{16} = \left( I_4 \otimes L_2^3 \right) \left( I_2 \otimes L_2^5 \right) L_2^{16} \]

2.3.2 Dimensionless FFT  
A dimensionless FFT algorithm is an algorithm that, subject to minor changes, is valid for all multi-dimensional DFTs independent of dimension. In particular, the internal “data flow” (data access pattern) remains the same for all multi-dimensional DFTs of a given size. Let \( \mathcal{F}_N \) denote an arbitrary multi-dimensional DFT of size \( N \). If \( N = 2^K \), there are \( 2^{K-1} \) different multi-dimensional DFTs. For example, \( F_{16} \), \( F_8 \otimes F_2 \), \( F_4 \otimes F_4 \), \( F_2 \otimes F_8 \), \( F_4 \otimes F_2 \otimes F_2 \), \( F_2 \otimes F_4 \otimes F_2 \), \( F_2 \otimes F_2 \otimes F_4 \), \( F_2 \otimes F_2 \otimes F_2 \otimes F_2 \) are the eight possible 16-point DFTs.

The factorization in Theorem 4 underlying the iterative radix-two FFT holds for all multi-dimensional DFTs if the initial permutation and twiddle factor matrices are modified.

\[ \mathcal{F}_N = \left( \prod_{i=1}^K \left( I_{2^{-1}} \otimes F_2 \otimes I_{2^{K-1}} \right) T_i \right) R, \quad (2.23) \]

where \( R \) is a permutation matrix and \( T_i, i = 1, \ldots, K \) are diagonal matrices both dependent on the dimension of \( \mathcal{F}_N \). The basic form of the factorization, including internal dataflow, remains the same independent of dimension. Hence this factorization provides a dimensionless FFT. Since the form of equation is the same as Theorem 4, the techniques applied in Section 2.3.1 can be used for generating alternative dimensionless FFTs.

The following example illustrates how to derive the factorization in Equation 2.23 for multi-dimensional DFTs.

**Example:** [Derivation of 16-point dimensionless FFT] The factorization is ob-
tained by starting with the row-column algorithm in Theorem 2.4 and substituting the one-dimensional FFT factorization in Theorem 4 for each of the DFTs. The final form of the factorization is obtained by appropriate application of the multiplicative property of the tensor product.

\[
F_4 \otimes F_4 = (F_4 \otimes I_4)(I_4 \otimes F_4)
\]
\[
= ((F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4 \otimes I_4)
\]
\[
(I_4 \otimes [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)R_4])
\]
\[
= ((F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)] \otimes I_4)
\]
\[
(I_4 \otimes [(F_2 \otimes I_2)T_2^4(I_2 \otimes F_2)])(R_4 \otimes R_4)
\]
\[
= (F_2 \otimes I_8)(T_2^4 \otimes I_4)(I_2 \otimes F_2 \otimes I_4)
\]
\[
(I_8 \otimes F_2)(I_4 \otimes T_2^4)(I_4 \otimes F_2 \otimes I_2)(R_4 \otimes R_4)
\]

Generalizing the computation in the previous example, we obtain the following theorem whose proof can be found in [8].

**Theorem 9 (Multi-Dimensional Radix 2 FFT)** Assume \(N = 2^K\) with \(N = n_1 \times \cdots \times n_t\) where \(n_i = 2^{k_i}\). Let \(K(i) = k_1 + \cdots + k_i\), with the boundary conditions \(K(t) = K\) and \(K(0) = 0\) denote the \(i\)-th prefix sum. Let \(\mathcal{F}_N = F_{n_1} \otimes \cdots \otimes F_{n_t}\) denote an arbitrary multi-dimensional DFT of size \(N = 2^K\).

\[
\mathcal{F}_N = \left\{ \prod_{i=1}^{K} (I_{2^{-1}} \otimes F_2 \otimes I_{2^{K-K(i)}}) T_i \right\} R_i \tag{2.24}
\]

where

\[
R = R_{n_1} \otimes \cdots \otimes R_{n_t} \tag{2.25}
\]

and

\[
T_i = I_{2^{k_{(j-1)+i-1}}} \otimes T_{2^{k_j-i+1}} \otimes I_{2^{K-K(j)}} \tag{2.26}
\]

with \(i = K(j-1)+l\) and \(0 \leq l < k_j\).
Example: [Example of 2-Dimension Radix 2 FFT]

\[ F_{16} = (F_2 \otimes I_8)T_1 \]
\[ = (I_2 \otimes F_2 \otimes I_4)T_2 \]
\[ = (I_4 \otimes F_2 \otimes I_2)T_3 \]
\[ = (I_8 \otimes F_2)T_4 \]
\[ = R \]

\[ T_1 = (T_2^4 \otimes I_4) \]
\[ = \text{diag}(1, 1, 1, \omega_4^1, 1, 1, 1, 1, 1, 1, 1, 1, \omega_4^1) \]

\[ T_2 = I_{16} \]
\[ = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

\[ T_3 = (I_4 \otimes T_2^4) \]
\[ = \text{diag}(1, 1, 1, \omega_4^1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

\[ T_4 = I_{16} \]
\[ = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

\[ R = (R_4 \otimes R_4) \]
\[ = \begin{pmatrix} 0 & 1 & 2 & 3 & 4 & 5 & 6 & 7 & 8 & 9 & 10 & 11 & 12 & 13 & 14 & 15 \\ 0 & 2 & 1 & 3 & 8 & 10 & 9 & 11 & 4 & 6 & 5 & 7 & 12 & 14 & 13 & 15 \end{pmatrix} \]

Parallel In-Place FFT

The canonical FFT factorization given in Definition 10 for the one-dimensional DFT carries over to the multi-dimensional case and will serve as the basis for the universal FFT processor presented in Chapter 3. Derivation of alternative instances of this canonical dimensionless factorization follow in exactly the same way as do one-dimensional instances except that the starting point is Theorem 9 instead of Theorem 4. In the next two sections explicit calculations are given for the Pease and Cooley-Tukey variants. Other variants can be derived in the same way.
Definition 11 (Parallel In-Place Dimensionless FFT) Assume $N = 2^K$, and let $\mathcal{F}_N = (F_{n_1} \otimes \cdots \otimes F_{n_t})$ be an arbitrary multi-dimensional DFT of size $N$, then a canonical in-place dimensionless FFT algorithm is given by the factorization

$$\mathcal{F}_N = \left\{ \prod_{i=1}^{K} P_i^{-1} (I_{2^{K-1}} \otimes F_2) T_i' P_i \right\} R,$$

(2.27)

where the internal permutations $P_i$, $i = 1, \ldots, K$ and the initial permutation $R$ are digit permutations, and $T_i$, $i = 1, \ldots, K$, are diagonal matrices with elements in the odd diagonal locations are all equal to 1. The initial permutation $R$ and the diagonal matrices $T_i$ depend on the dimension of $\mathcal{F}_N$, whereas the internal permutations $P_i$ are fixed independent of dimension.

Pease Algorithm

The following theorem is the dimensionless version of Theorem 7. The proof follows from Theorem 9 and is identical to the proof of Theorem 7.

Theorem 10 (Dimensionless Pease Algorithm) Assume $N = 2^K$ with $N = n_1 \times \cdots \times n_t$ where $n_i = 2^{k_i}$. Let $K(i) = k_1 + \cdots + k_i$, with the boundary conditions $K(t) = K$ and $K(0) = 0$ denote the $i$-th prefix sum. Let $\mathcal{F}_N = F_{n_1} \otimes \cdots \otimes F_{n_t}$ denote an arbitrary multi-dimensional DFT of size $N = 2^K$. Then

$$\mathcal{F}_N = \left\{ \prod_{i=1}^{K} L_{2^i}^N \left( I_{N/2} \otimes F_2 \right) T_i' L_{2^{K-1-i}}^N \right\} R,$$

(2.28)

where $R = R_{n_1} \otimes \cdots \otimes R_{n_t}$ and the twiddle factor matrices are

$$T_i' = L_{2^{K-1-i}}^N \left( I_{2^{K(j-1)+l-1}} \otimes T_{2^{K-j-i}}^{2^{K-j-1}} \otimes I_{2^{K-K(j)}} \right) L_{2^i}^N.$$

(2.29)

Corollary: [Twiddle Factors for Dimensionless Pease Algorithm]

$$T_i' = \bigoplus_{a=0}^{p-1} \bigoplus_{b=0}^{q-1} \bigoplus_{c=0}^{r-1} \omega_{2p}^{a \cdot d}, \quad \omega_{2p}^{a \cdot d} = \omega_{N}^{a \cdot d \cdot q \cdot r},$$

(2.30)

where $r = 2^{i-1}$, $p = 2^{K-j-l}$, $q = 2^{K-K(j)}$, and $i = K(j - 1) + l = K(j) - k_j + l$. 
Proof: Let \( r = 2^{i-1}, p = 2^{k_j-l}, q = 2^{K-K(j)} \), and \( i = K(j-1) + l = K(j) - k_j + l \).

Then \( N = 2pqr = 2^K \), and

\[
T_i = (I_{2^K_{(j-1)+l-1}} \otimes T_{2^{k_j-l}}^{2^{k_j-l+1}} \otimes I_{2K-K(j)})
\]

\[
T'_i = \ L_{2K-i}^{2K} (I_{2^{k_j-l+1}} \otimes I_{2K-K(j)}) L_{2K}^{2K}
\]

\[
= L_{2^{2pqr}}^{2pq} (I_r \otimes T_p^{2p} \otimes I_q) L_{2^{2pq}}^{2pq},
\]

and

\[
T'_i (e^p_a \otimes e^q_b \otimes e^r_c \otimes e^d_d) = L_{2^{2pq}}^{2pq} (I_r \otimes T_p^{2p} \otimes I_q) L_{2^{2pq}}^{2pq} (e^p_a \otimes e^2_d \otimes e^r_c \otimes e^d_d)
\]

\[
= L_{2^{2pq}}^{2pq} (I_r \otimes T_p^{2p} \otimes I_q) (e^r_c \otimes e^2_d \otimes e^p_a \otimes e^q_b)
\]

\[
= \omega^{a,d}_{2p} L_{2^{2pq}}^{2pq} (e^r_c \otimes e^2_d \otimes e^p_a \otimes e^q_b)
\]

\[
= \omega^{a,d}_{2p} (e^p_a \otimes e^q_b \otimes e^r_c \otimes e^d_d).
\]

Therefore,

\[
T_i = \bigoplus_{a=0}^{p-1} \bigoplus_{b=0}^{q-1} \bigoplus_{c=0}^{r-1} \bigoplus_{d=0}^{1} \omega^{a,d}_{2p}.
\]

Example: [Dimensionless Pease Algorithm]

\[
F_4 \otimes F_4 = L_2^{16} (I_8 \otimes F_2) T'_i L_8^{16}
\]

\[
L_4^{16} (I_8 \otimes F_2) T_2^{16} L_4^{16}
\]

\[
L_8^{16} (I_8 \otimes F_2) T_2^{16} L_2^{16}
\]

\[
L_{16}^{16} (I_8 \otimes F_2) T_2^{16} L_1^{16}
\]

\[
R_4 \otimes R_4
\]

The twiddle factors \( T'_i \) are,

\[
T'_i = L_8^{16} (T_2^{16} \otimes I_4) L_2^{16}
\]

\[
= \bigoplus_{a=0}^{1} \bigoplus_{b=0}^{3} \bigoplus_{c=0}^{0} \bigoplus_{d=0}^{1} \omega^{a,d}_4
\]
\[ T_2' = L_4^{16} I_16 L_4^{16} \]
\[ = \bigoplus_{a=0}^{1} \bigoplus_{b=0}^{3} \bigoplus_{c=0}^{1} \bigoplus_{d=0}^{1} \omega_{2}^{a b c d} \]
\[ = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

\[ T_3' = L_8^{16} (I_4 \otimes T_2^3) L_8^{16} \]
\[ = \bigoplus_{a=0}^{1} \bigoplus_{b=0}^{3} \bigoplus_{c=0}^{1} \bigoplus_{d=0}^{1} \omega_{4}^{a b c d} \]
\[ = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

\[ T_4' = L_4^{16} I_16 L_4^{16} \]
\[ = \bigoplus_{a=0}^{1} \bigoplus_{b=0}^{3} \bigoplus_{c=0}^{1} \bigoplus_{d=0}^{1} \omega_{2}^{a b c d} \]
\[ = \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1) \]

Cooley-Tukey Algorithm

The following theorem is the dimensionless version of Theorem 8. The proof follows from Theorem 9 and is identical to the proof of Theorem 8.

**Theorem 11 (Dimensionless Cooley-Tukey Algorithm)** Assume \( N = 2^K \) with \( N = n_1 \times \cdots \times n_t \) where \( n_i = 2^{k_i} \). Let \( K(i) = k_1 + \cdots + k_i \), with the boundary conditions \( K(t) = K \) and \( K(0) = 0 \) denote the \( i \)-th prefix sum. Let \( \mathcal{F}_N = F_{n_1} \otimes \cdots \otimes F_{n_t} \) denote an arbitrary multi-dimensional DFT of size \( N = 2^K \). Then

\[ \mathcal{F}_N = \left\{ \prod_{i=1}^{K} (I_{2^{K-i}+1} \otimes L_2^{2^{K-i}}) (I_{N/2} \otimes F_2) T'_i (I_{2^{K-i}+1} \otimes L_2^{2^{K-i}}) \right\} R, \]

where, \( R = R_{n_1} \otimes \cdots \otimes R_{n_t} \) and the twiddle factor matrices are

\[ T'_i = (I_{2^{K-i}} \otimes L_2^{2^{K-i}}) (I_{2^{K-i}+1} \otimes T_{2^{K-i}}^{2^{K-j}}) (I_{2^{K-i}} \otimes L_2^{2^{K-i}}). \]

**Corollary:** [Twiddle Factors for Dimensionless Cooley-Tukey Algorithm]

\[ T'_i = \bigoplus_{a=0}^{r-1} \bigoplus_{b=0}^{p-1} \bigoplus_{c=0}^{q-1} \bigoplus_{d=0}^{1} \omega_{2p}^{a b c d}, \quad \omega_{2p}^{a b c d} = \omega_N^{a b q_p r}, \]
where \( r = 2^{i-1}, p = 2^{k_j-l}, q = 2^{K-K(j)}, \) and \( i = K(j-1)+l = K(j) - k_j + l. \)

**Proof:** Let \( r = 2^{i-1}, p = 2^{k_j-l}, q = 2^{K-K(j)}, \) and \( i = K(j-1)+l = K(j) - k_j + l. \)

Then, \( N = 2pq, \) and

\[
T_i = \left(I_{2^{K(j-1)+i-1} \otimes T_{2^{k_j-1} \otimes I_{2^{K-K(j)}}}}\right)
\]

\[
T_i' = \left(I_{2^{q-1} \otimes L_{2^{K-K(j)}}^{2^{k_j-1}+1}} \otimes T_{2^{2k_j-1} \otimes I_{2^{K-K(j)}}} \left(I_{2^{q-1} \otimes L_{2}^{2K-K(j)+1}}\right)\right)
\]

\[
= \left(I_{r \otimes L_{p}^{2pq}} \right) \left(I_{r \otimes T_{2}^{2p} \otimes I_{q}} \right) \left(I_{r \otimes L_{2}^{2pq}}\right)
\]

and,

\[
T_i'(e_a^r \otimes e_b^p \otimes e_c^q \otimes e_d^2) = \left(I_{r \otimes L_{p}^{2pq}} \right) \left(I_{r \otimes T_{2}^{2p} \otimes I_{q}} \right) \left(I_{r \otimes L_{2}^{2pq}}\right) \left(e_a^r \otimes e_b^p \otimes e_c^q \otimes e_d^2\right)
\]

\[
= \left(I_{r \otimes L_{p}^{2pq}} \right) \left(I_{r \otimes T_{2}^{2p} \otimes I_{q}} \right) \left(e_a^r \otimes e_d^2 \otimes e_b^p \otimes e_c^q\right)
\]

\[
= \omega_{2p}^{a,b} \left(I_{r \otimes L_{p}^{2pq}} \right) \left(e_a^r \otimes e_d^2 \otimes e_b^p \otimes e_c^q\right)
\]

\[
= \omega_{2p}^{a,b} \left(e_a^r \otimes e_b^p \otimes e_c^q \otimes e_d^2\right).
\]

Thus,

\[
T_i' = \bigoplus_{a=0}^{r-1} \bigoplus_{b=0}^{p-1} \bigoplus_{c=0}^{q-1} \bigoplus_{d=0}^{1} \omega_{2p}^{a,b}.
\]

**Example:** [Dimensionless Cooley-Tukey Algorithm]

\[ F_4 \otimes F_4 = \left(L_{2}^{16}\right)\left(I_8 \otimes F_2\right)T_i'(L_{8}^{16}) \]

\[ = \left(I_2 \otimes L_{2}^{8}\right)\left(I_8 \otimes F_2\right)T_2'(I_2 \otimes L_{8}^{8}) \]

\[ = \left(I_4 \otimes L_{2}^{4}\right)\left(I_8 \otimes F_2\right)T_3'(I_4 \otimes L_{8}^{4}) \]

\[ = \left(I_8 \otimes L_{2}^{2}\right)\left(I_8 \otimes F_2\right)T_4'(I_8 \otimes L_{8}^{2}) \]

\[ = (R_4 \otimes R_4). \]

The twiddle factors \( T_1' \) are

\[
T_1' = \left(L_{8}^{16}\right)\left(T_2^4 \otimes I_4\right)\left(L_{2}^{16}\right)
\]

\[
= \bigoplus_{a=0}^{0} \bigoplus_{b=0}^{1} \bigoplus_{c=0}^{3} \bigoplus_{d=0}^{1} \omega_{4}^{a,b}
\]
\[
T'_2 = (I_2 \otimes L_4^8) I_{16} (I_2 \otimes L_2^8)
\]
\[
= \bigoplus_{a=0}^{1} \bigoplus_{b=0}^{3} \bigoplus_{c=0}^{1} \bigoplus_{d=0}^{1} \omega_2^{d,b}
\]
\[
= \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, \omega_4^1, 1, \omega_4^1, 1, \omega_4^1, 1, \omega_4^1, 1)
\]
\[
T'_3 = (I_4 \otimes L_2^4)(I_4 \otimes T'_2)(I_4 \otimes L_2^4)
\]
\[
= \bigoplus_{a=0}^{3} \bigoplus_{b=0}^{1} \bigoplus_{c=0}^{1} \bigoplus_{d=0}^{1} \omega_4^{d,b}
\]
\[
= \text{diag}(1, 1, 1, \omega_4^1, 1, 1, 1, \omega_4^1, 1, 1, 1, \omega_4^1, 1, 1, 1, \omega_4^1)
\]
\[
T'_4 = (I_8 \otimes L_2^8) I_{16} (I_8 \otimes L_2^8)
\]
\[
= \bigoplus_{a=0}^{7} \bigoplus_{b=0}^{0} \bigoplus_{c=0}^{0} \bigoplus_{d=0}^{1} \omega_2^{d,b}
\]
\[
= \text{diag}(1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1)
\]

### 2.4 Matlab Implementation of the Dimensionless FFT

In this section we provide a Matlab implementation of the Dimensionless FFT. Implementations of the in-place Pease and Cooley-Tukey algorithms described in Theorems 10 and 11 are provided. These programs provide validation of the corresponding theorems and can be used to make sure that the twiddle factor computations and addressing related to the internal permutations are correct. The programs are not intended to provide efficient implementations, but rather are intended simply to validate the given algorithms. Additional variants of the in-place parallel form of the dimensionless FFT can easily be obtained by providing modified programs for the internal permutations and twiddle factors.

The initial permutation in both the Pease and Cooley-Tukey algorithms is a multidimensional bit-reversal. The function `mdbr` in Figure 2.2 implements the multidimensional bit-reversal computation \( y = (R_{n_1} \otimes \cdots \otimes R_{n_i}) x \), using the factorization

\[
R_{n_1} \otimes \cdots \otimes R_{n_t} = (R_{n_1} \otimes I_{N/n_1})(I_{n_1} \otimes (R_{n_2} \otimes \cdots \otimes R_{n_t})).
\]  

(2.34)
The algorithm recursively computes \((R_{n_2} \otimes \cdots \otimes R_{n_t})\) for \(n_1\) consecutive subvectors of size \(N/n_1\) of the input \(x\). The \(i\)-th permuted block of data is placed in the subvector of \(y\) of size \(N/n_1\) beginning with index \(\text{rev}(i) \cdot (N/n_1)\), where \(\text{rev}(i)\) is the index obtained by reversing the bits of binary representation of \(i\). In the base case, the bit-reversal algorithm in Figure 2.1 is used.

The function \texttt{md\_fft\_p} in Figure 2.3 implements the Pease variant of the dimensionless FFT. The internal permutations, \(P_i = L_{2^k-1}^N\), are implemented by the function \texttt{stride} (see Figure 2.5) which computes \(L_d^N\), and the twiddle factors are computed by the function \texttt{md\_tf\_p} (see Figure 2.4), which is based on Corollary 2.3.2.

The function \texttt{md\_fft\_ct} in Figure 2.6 implements the Cooley-Tukey variant of the dimensionless FFT. The internal permutations, \(P_i = (I_{2^i-1} \otimes L_{2^k-i+1}^N)\), are implemented by the function \texttt{itstride} (see Figure 2.5) which computes \(I_m \otimes L_d^N\), and the twiddle factors are computed by the function \texttt{md\_tf\_ct} (see Figure 2.4), which is based on Corollary 2.3.2. This function is identical to the Pease variant in Figure 2.3 except the call to \texttt{itstride} replaces the call to \texttt{stride} and the call to \texttt{md\_tf\_ct} replaces the call to \texttt{md\_tf\_p}.

Multi-dimensional DFT programs can be tested exhaustively, for a given size, by applying the program to a set of basis vectors (this assumes that the program is linear and ignores roundoff error). Our programs were tested by applying them to the standard basis. Let \(A\) be an \(n \times n\) matrix, then \(Ae_i^n\) is equal to the \(i\)-th column of the matrix \(A\). In the case of the multi-dimensional DFT,

\[
(F_n \otimes \cdots \otimes F_n)(e_{i_1}^{n_1} \otimes \cdots \otimes e_{i_t}^{n_t}) = (f_{i_1}^{n_1} \otimes \cdots \otimes f_{i_t}^{n_t}),
\]

where \(f_i^n = \sum_{j=0}^{n-1} \omega_n^j e_j^n\) is the \(i\)-th column of the DFT matrix.

The Matlab function \texttt{ebasis}, shown in Figure 2.7, computes elements of the standard basis \((e_{i_1}^{n_1} \otimes \cdots \otimes e_{i_t}^{n_t})\), and the function \texttt{fbasis}, shown in Figure 2.8, computes elements of the Fourier basis \((f_{i_1}^{n_1} \otimes \cdots \otimes f_{i_t}^{n_t})\). The function \texttt{ebasis} is used to generate inputs to test DFT programs, and the function \texttt{fbasis} is used to check the results.
% Bit Reversal Permutation.
% Usage: bitreversal(x)
% Input: x a vector of length n. n = 2^t, t an integer, t >= 0.
% Output: y = R_{2^t} x

function y = bitreversal(x)
n = length(x);
t = floor(log2(n));
y = zeros(n,1);
for i=0:n-1
    j = bitrevind(t,i);
    y(j+1) = x(i+1);
end

% Usage: bitrevind(t,index)
% Inputs:
%   t : positive integer
%   index : nonnegative integer. 0 <= index < 2^t
% Output:
%   revindex : integer with revindex = rev_t(index)

function revindex = bitrevind(t,index)
revindex = 0;
tempindex = index;

for i=0:t-1
    bit = mod(tempindex,2);
    tempindex = floor(tempindex/2);
    revindex = revindex*2 + bit;
end

Figure 2.1: Matlab implementation of bit-reversal
% Multi-dimensional Bit Reversal Permutation.
% Usage: bitreversal(x)
% Input: x a vector of length N. N = 2^K, K an integer, K >= 0.
% k = (k_1,...,k_t), K = k_1 + ... + k_t., n_i = 2^k_i
% Output: y = (R_{n_i} tensor ... tensor R_{n_t}) x,
% where n_i = 2^k_i.

function y = mdb(x,k)
    t = length(k);
    if (t == 1)
        y = bitreversal(x);
    else
        K = sum(k(1:t));
        Nb = 2^K-k(1);
        for i=1:2^K(1)
            j = bitrevind(k(1),i-1)+1;
            y((j-1)*Nb+1:1:j*Nb) = mdb(x((i-1)*Nb+1:1:i*Nb),k(2:t));
        end
    end
end

Figure 2.2: Matlab implementation of multi-dimensional bit-reversal
computed by the program being tested.
function y = md_fft_p(x,k)
% Mulit-dimensional FFT using the Pease formulation
% Usage: md_fft_p(x,k)
% Input:  x a vector of length N.  N = 2^K, K >= 0.
%  k: (k_1,...,k_t), with K = k_1 + ... + k_t.
% Output: y = (F_{n_1} tensor ... tensor F_{n_t}) x,
% where n_i = 2^{-k_i}.
N = length(x);
K = sum(k);
t=length(k);
xt = mdbr(x,k);
for i=K:-1:1
% determine j and l such that i = K(j-1) + l, 0 <= l < k_j
j = 0;
while (sum(k(1:j)) < i)
    j = j + 1;
end
l = i - sum(k(1:j-1));
q = 2^{K-sum(k(1:j))};
p = 2^{k(j)-1};
r = 2^{i-1};
Tp = (md_tf_p(p,q,r))';
xt=stride(xt,N,2^{K-i});
xt=Tp .* xt;
for a = 0:N/2-1,
    t = xt(a*2+1) + xt(a*2+2);
    xt(a*2+2) = xt(a*2+1) - xt(a*2+2);
    xt(a*2+1) = t;
end
xt=stride(xt,N,2^{i});
end
y = xt;

Figure 2.3: Matlab implementation of the Pease dimensionless FFT
% Multi-dimension twiddle factors for Pease dimensionless FFT
% Usage: function w = md_tf_p(p,q,r)
% Input:
%   p, q, r: positive integers with p = 2^(Kj - 1), q = 2^(K-K(j)),
%            and r = 2^(i-1)
% Output:
%   w = dirsum_{a=0}^{p-1} dirsum_{b=0}^{q-1} dirsum_{c=0}^{r-1}
%        dirsum_{d=0}^{1} w_{ad}_{2p}.

function w=md_tf_p(p,q,r)
idx = 1; for a = 0 : p-1,
    for b = 0: q-1,
        for c = 0:r-1,
            for d = 0:1,
                w(idx) = exp((2*pi*sqrt(-1))/(2*p))^(a*d);
                idx=idx+1;
            end
        end
    end
end
end

% Multi-dimension twiddle factors for Cooley-Tukey dimensionless FFT
% Usage: function w = md_tf_ct(p,q,r)
% Input:
%   p, q, r: positive integers with p = 2^(Kj - 1), q = 2^(K-K(j)),
%            and r = 2^(i-1)
% Output:
%   w = dirsum_{a=0}^{r-1} dirsum_{b=0}^{p-1} dirsum_{c=0}^{q-1}
%        dirsum_{d=0}^{1} w_{db}_{2p}.

function w=md_tf_ct(p,q,r)
idx = 1;
for a = 0 : r-1,
    for b = 0: p-1,
        for c = 0:q-1,
            for d = 0:1,
                w(idx) = exp((2*pi*sqrt(-1))/(2*p))^(d*b);
                idx=idx+1;
            end
        end
    end
end
end

Figure 2.4: Matlab implementation of Multi-dimensional twiddle factors
% Identity Tensor Stride permutation
% Usage: y = itstride(x,M,N,d)
% Input:
% M, N, d : positive integers such that d|N.
% x a vector of size M*N
% Output:
% y = (I_M tensor L^N_d)x.

function y = itstride(x,M,N,d)
y = zeros(M*N,1);
for i=1:M
    y((i-1)*N+1:i*N) = stride(x((i-1)*N+1:i*N),N,d);
end

% Stride permutation
% Usage: y = stride(x,N,d)
% Input:
% N, d : positive integers such that d|N.
% x a vector of size N
% Output:
% y = L^N_d x.

function y = stride(x,N,d)
y = zeros(N,1);
M = N/d;
for i=1:d
    y((i-1)*M+1:i*M) = x(i:d:N);
end

Figure 2.5: Matlab implementation of stride permutations
function y = md_fft_ct(x,k)
% Multidimensional FFT using the Cooley-Tukey formulation
% Usage: md_fft_ct(x,k)
% Input: x a vector of length N. N = 2^K, K >= 0.
% k: (k_1,...,k_t), with K = k_1 + ... + k_t.
% Output: y = (F_{n_1} tensor ... tensor F_{n_t}) x, where n_i = 2^{k_i}.
N = length(x);
K = sum(k);
t=length(k);
xt = mdbin(x,k);
for i=K:-1:1
% determine j and l such that i = k(j-1) + l, 0 <= l < k_j
    j = 0;
    while (sum(k(1:j)) < i)
        j = j + 1;
    end
    l = i - sum(k(1:j-1));
    q = 2^((K-sum(k(1:j))));
    p = 2^((k(j)-1));
    r = 2^((i-1));
    Tp = (md_tf_ct(p,q,r))';
    xt=itStride(xt,r,2^((K-i+1)),2);
    xt=Tp.*xt;
    for a = 0:N/2-1,
        t = xt(a*2+1) + xt(a*2+2);
        xt(a*2+2) = xt(a*2+1) - xt(a*2+2);
        xt(a*2+1) = t;
    end
    xt=itStride(xt,r,2^((K-i+1)),2^((K-i)));
end
y = xt;

Figure 2.6: Matlab implementation of the Cooley-Tukey dimensionless FFT
% Standard basis element.
% usage:   ebasis(N,J)
% Input:
%   N : positive integer or array = [n1 ... nt],
%       where ni is an integer >= 2.
%   J : if N is an integer, then J is an integer 0 <= J < N.
%       otherwise J = [j1,...,jt], where 0 <= j_i < n_i.
% Output:
%   y : vector.
%       if N is a scalar, y = e^N_j,
%       if N is a vector, y = e^{n1_j1} tensor ... tensor e^{nt_jt} = e^N_j,
%       where n = prod(N), and j = j1*n2*...*nt + ... + j_{t-1}*nt + jt.

function y = ebasis(N,J)
if (isequal(size(N,[1 1])))
    y = zeros(N,1);
    if (not(isequal(size(J,[1,1]))))
        error('J must be a scalar when N is a scalar');
    end
    if (or(J < 0,J > N))
        error('index out of range');
    end
    y(J+1) = 1;
else
    if (length(J) ~= length(N))
        error('incompatible dimensions');
    end
    t = length(N);
    n = prod(N);
    y = zeros(n,1);
    ind = 0;
    for i=1:t
        if (or(J(i) < 0,J(i) > N(i)))
            error('index out of range');
        end
        ind = ind*N(i) + J(i);
    end
    y(ind+1) = 1;
end

Figure 2.7: Matlab standard basis generator
% Fourier basis element.
% Usage: fbasis(N,J)
% Input:
% N : positive integer or array = [n1 ... nt],
% where ni is an integer >= 2.
% J : if N is an integer, then J is an integer 0 <= J < N.
% otherwise J = [j1,...,jt], where 0 <= j_i < n_i.
% Output:
% y : vector.
% if N is a scalar, y = f^N_J,
% if N is a vector, y = f^n1_j1 tensor ... tensor f^nt_jt.

function y = fbasis(N,J)
t = length(N)
if (t == 1)
y = exp((2*pi*i/N(1))*(-J(1)*(0:N(1)-1)'));
else
    yt = fbasis(N(2:t),J(2:t));
    Nt = prod(N(2:t));
    y1 = exp((2*pi*i/N(1))*(-J(1)*(0:N(1)-1)'));
    y = kron(y1,yt);
end

Figure 2.8: Matlab standard basis generator
3 Universal FFT Processor

Equation 2.27 in the definition of an in-place dimensionless FFT algorithm can be interpreted as an abstract machine for computing multi-dimensional DFTs. Abstractly the machine is equivalent to the dataflow graph indicated by the formula. The internal dataflow pattern is determined by the set of permutations $P_1, \ldots, P_K$. Once these permutations are fixed, the computation performed is determined by the values of the twiddle factors, $T'_1, \ldots, T'_K$, and the initial permutation $R$. By choosing appropriate values of the twiddle factors and the initial permutation, any multi-dimensional DFT for a fixed number of points can be computed. Since the internal permutations are fixed, independent of dimension, a machine can be designed with a fixed data path that is capable of computing arbitrary multi-dimensional DFTs provided it can be configured to use appropriate twiddle factors and load the data according to the necessary initial permutation. Such a machine is called a universal FFT processor [7].

In this chapter we describe the key features a universal FFT processor. First we directly interpret Equation 2.15 as an abstract processor. The abstract processor has a processing element for each butterfly operation and memory for each stage in the factorization. Second we map the abstract processor to a distributed memory architecture with a parameterized number of processors. This mapping involves scheduling butterfly operations to the given number of processors and reusing memory.

3.1 Abstract FFT Processor

Equation 2.15 consists of an initial permutation, $R$, followed by $K$ stages, where a stage corresponds to a factor of the form $P^{-1}(I_{N/2} \otimes F_2)T^TP$, where $P$ is a permutation, $T$ a diagonal matrix 1's alternating with roots of unity. When applied to an input
vector, the factor \((I_{N/2} \otimes F_2)\) combined with \(T\) corresponds to \(N/2\) butterfly operations (see Definition 9). The permutations \(P\) determines the two data elements used in the butterfly operation.

Figure 3.1 depicts the computation implied by Equation 2.15 when \(N = 2^4\). The computation starts on the left with the input vector \(x = (x_0, x_1, \ldots, x_{15})^t\) and completes on the right with the output vector \(y = \mathcal{F}_{16} x = (y_0, y_1, \ldots, y_{15})\). Each stage contains 8 butterfly operations. Butterfly operations are represented by a box with an “X”. The small box labeled with a “T” connected to the butterfly contains an unspecified twiddle factor. A universal FFT processor is obtained once a valid set of permutations are provided for \(P_1, \ldots, P_K\). A valid set of permutations are \(K\) permutations which can be used to compute a dimensionless FFT. Figures 3.2 and 3.3 show the 16-point universal FFT processors based on Theorems 11 and 10. A universal FFT processor of size \(N = 2^K\) can be configured to compute an arbitrary multidimensional DFT. Configuring the processor requires setting each of the unspecified twiddle factors and the initial permutation. Figures 3.4 and 3.5 show the 16-point Pease universal FFT processor configured to compute a one-dimensional 16-point DFT (see
Figure 3.2: 16-point universal Pease FFT machine

Figure 3.3: 16-point universal Cooley-Tukey FFT machine
Figure 3.4: 16-point universal Pease FFT machine configured to compute 1D DFT

Example 2.3.1) and a two-dimensional $4 \times 4$ DFT (see Example 2.3.2).

3.1.1 Address and Twiddle Generation

As indicated the addressing for the butterfly operations in stage $c$ are determined by the permutation $P_c$. If we label the butterfly operations by a pair, $(r, c)$, of row and column coordinates with $0 \leq r < 2^{K-1}$ and $1 \leq c \leq K$, the address of the inputs and outputs to the butterfly operation labeled by $(r, c)$ is uniquely determined by the coordinates $r$ and $c$. Let $A(r, c)$ be the function that returns the address for the $(r, c)$ butterfly operation. The function $A(r, c)$ is determined by the permutation $P_c$. Since we have restricted $P_c$ to be a bit permutation (i.e. a digit permutation with all of the digits binary bits), the function $A(r, c)$ can be computed using bit operations. For the Pease algorithm, $P_c = L_{2K-c}^N$, and

$$A(r, c) = (\text{rot}_c(r||0), \text{rot}_c(r||1)),$$

(3.1)
Figure 3.5: 16-point universal Pease FFT machine configured to compute $4 \times 4$ DFT

where $r || b$ indicates the concatenation of the binary representation of $r$ and the bit $b$, and $\text{rot}_c$ rotates the bits in a binary number $c$ bits to the left. A Matlab program to compute this function is given in Figure 3.6.

For the Cooley-Tukey machine, $P_c = I_{2^l-1} \otimes L^{2^{K-c+1}}_{2^K-c}$, and

$$A(r, c) = (r_1 || \text{rot}_c (r_0 || 0), r_1 || \text{rot}_c (r_0 || 1)),$$

(3.2)

where $r = r_1 || r_0$ and $r_0$ is the first $K - i$ bits in the binary representation of $r$. A Matlab program to compute this function is given in Figure 3.7.

Computation of the twiddle factor needed for the $(r, c)$ butterfly operation depends on both the permutation $P_c$ and the dimension the machine is configured to compute. Corollaries 2.3.2 and 2.3.2 show how to compute the twiddle factors for the Pease and Cooley-Tukey machines respectively. These computations are similar to the address functions and the necessary root of unity can easily be computed using bit operations.

Let $k = (k_1, \ldots, k_i)$ specify the desired dimension information and let $c = K(j -$
% Address computation for the Pease universal FFT processor.
% usage: addr(t,r,c)
% Inputs:
%   t : positive integer.
%   r, c : nonnegative integers with 0 <= c < t,
%          0 <= r < 2^((t-1).
% Output
%   [a0,a1] : vector of integers of size 2. The addresses
%          for the (r,c) butterfly operation of a 2^t-point
%          conjugated Pease FFT.
function a = addr(t,r,c)

    cp = c + 1;
    r0 = bitshift(r,1);  % add trailing 0-bit
    r1 = bitor(bitshift(r,1),1);  % add trailing 1-bit
    % rotate bits.
    a0 = bitrot(r0,-cp,t);  a1 = bitrot(r1,-cp,t);
    a = [a0,a1];

Figure 3.6: Matlab address calculation for the Pease machine

1) + l, then the twiddle factor function for the Pease algorithm

\[ T(r,c,k,j) = \omega_{2^p}^{r_1^1}, \tag{3.3} \]

where \( p = 2^{k_j-l} \) and \( r_1 \) is the leading \( k_j - l \) bits of \( r \). A Matlab program to compute this function is given in Figure 3.8.

The twiddle factor function for the Cooley-Tukey machine can be determined in a similar fashion. In this case,

\[ T(r,c,k,j) = \omega_{2^p}^{r_m^m}, \tag{3.4} \]

where \( p = 2^{k_j-l} \) and \( r_m \) are the \( k_j - l \) bits after the leading \( c - 1 \) of \( r \). A Matlab program to compute this function is given in Figure 3.9.

3.2 Distributed Memory FFT Processor

The abstract universal FFT processor of size \( N = 2^K \) requires \((N/2)\lg(N)\) butterfly processors and \( \lg(N) \) memories of size \( N \). In order to accommodate varying numbers
% Address computation for the Cooley-Tukey universal FFT processor.
% usage: addr(t,r,c)
% Inputs:
%  t : positive integer.
%  r, c : nonnegative integers with 0 <= c < t, 0 <= r < 2^t-1.
% Output
%  [a0,a1] : vector of integers of size 2. The addresses
%  for the (r,c) butterfly operation of a 2^t-point
%  conjugated Cooley-Tukey FFT.
function a = addr_ct(t,r,c)

r0 = bitshift(r,1);  % add trailing 0-bit
r1 = bitor(bitshift(r,1),1);  % add trailing 1-bit

B=bitand(r0,1);  %bottom bit of address
M=bitand(r0,(2^t-2^t-c-2));  %middle bit(s) of address
T=bitand(r0,(2^t-2^t-c-2));  %top bit(s) of address
% the top bits stay, the bottom bit shifts to the middle position and
% the middle bits shift to the bottom
a0 = T+bitshift(B,t-c-1)+bitshift(M,-1);

% the same as above is done for the second address
B=bitand(r1,1);  M=bitand(r1,(2^t-c-2));  T=bitand(r1,(2^t-c-2));
30 = T+bitshift(B,t-c-1)+bitshift(M,-1);  a = [a0,a1];

Figure 3.7: Matlab address calculation for the Cooley-Tukey machine
% Twiddle factor computation for the Pease universal FFT processor.
% usage: md_twiddle_p(r,c,k)
% Inputs:
% r, c: nonnegative integers with 0 <= c < t, 0 <= r < 2^(t-1).
% k: k = (k_1,...,k_t)
% Output:
% The twiddle factor for the (r,c) butterfly operation configured to compute F_{n_1} tensor ... tensor F_{n_t},
% where n_i = 2^(k_i).

function w = md_twiddle_p(r,c,k)
    t=sum(k);
    diff=t-c;
    mk=length(k);
    for i = mk:-1:1,
        ck=sum(k(i:mk))-diff;
        if ck >= 0
            ki=i;
            break;
        end
    end
    p=sum(k(1:ki))-c;
    omega = exp(-2*pi*sqrt(-1)/2^p);
    r = bitshift(r,(-(t-p)));
    w = omega^r;

Figure 3.8: Matlab twiddle factor calculation for the Pease machine
% Twiddle factor computation for the Cooley-Tukey
% universal FFT processor.
% usage : md_twiddle_ct(r,c,k)
% Inputs:
% r, c : nonnegative integers with 0 <= c < t, 0 <= r < 2^(t-1).
% k : k = (k_1,...,k_t)
% Output:
% The twiddle factor for the (r,c) butterfly operation
% configured to compute F_{n_1} tensor ... tensor F_{n_t},
% where n_i = 2^{k_i}.

function w = md_twiddle_ct(r,c,k)
t=sum(k);
diff=t-c;
mk=length(k);
for i = mk:-1:1,
    ck=sum(k(i:mk))-diff;
    if ck >= 0
        ki=i;
        break;
    end
end
lg_p=((sum(k(1:ki))-c-1));
lg_q=(t-sum(k(1:ki)));
omega = exp(-2*pi*sqrt(-1)/2^(lg_p+1));
mask=bitshift(((2^lg_p)-1),lg_q); r=bitand(r,mask);
r=bitshift(r,-lg_q);
w = omega^(r);

Figure 3.9: Matlab twiddle factor calculation for the Cooley-Tukey machine
of processors and to reduce the amount of memory, the abstract processor is mapped
to an architecture with a specified number of processing elements and one memory
of size \( N \). This reduction can be viewed as folding the rows and columns of the ab-
stract machine. A distributed memory architecture, with the data partitioned equally
amongst the processors, was chosen. Since most of the computation can be performed
locally, a distributed memory design can be scaled to use many processing elements.
Since the addresses and twiddle factor for each butterfly operation can be determined
in dependently global communication is limited to accessing data stored remotely
and to synchronizing computation between stages. An interconnection network is
provided for accessing data stored in a remote processor. The choice of different
algorithms, i.e. internal permutations, leads to varying amounts of remote memory
access and contention on the interconnection network. See [6] for a discussion of this
optimization problem.

Figure 3.2 shows a four processor distributed-memory FFT processor. Each pro-
cessing element computes a sequence of butterfly operations. Let \( P \) be the number
of processors. Consecutive segments of size \( N/P \) are distributed to the \( P \) memory
units. Thus an address is broken into a memory id (mid), equal to the Processor ID
(PID), and an offset. If \( P = 2^p \), the mid is equal to the leading \( p \) bits of the address
and the offset is equal to the trailing \( K - p \) bits. To perform a butterfly operation,
two data elements must be loaded, the appropriate twiddle factor generated, and the
results must be computed and stored. Since the algorithm is in-place, the inputs
are loaded from and the outputs stored to the same two locations. For each butterfly
operation the addresses for these two memory locations must be generated. As shown
previously the addresses can be computed from the butterfly coordinates \((r, c)\) and
depend on the permutations defining the algorithm.

Butterfly operations are allocated to the processing elements using a round robin
scheduling algorithm. The tasks performed by processor \( \text{pid} \) are \((\text{pid}, c), (\text{pid} +
Figure 3.10: Universal FFT Processor Architecture
\[
P, c), \ldots, (\text{pid} + (N/P - 1) \cdot P, c) \text{ for } c = 1, \ldots, K. \text{ Figure 3.11 shows this schedule.}
\]
Using a fixed scheduling algorithm does not limit scheduling choices, since different schedules can be obtained by changing the algorithm. In this setting, the space of allowable schedules correspond to the set of permutations \( P_1, \ldots, P_K \) that lead to valid FFT algorithms.

The sequence of addresses generated by processor \text{pid} are equal to \( A(\text{pid}, c), A(\text{pid} + P, c), \ldots, A(\text{pid} + (N/P - 1) \cdot P, c) \) for \( c = 1, \ldots, K \) and the twiddle factors needed to compute the transform \( F_n_1 \otimes \cdots \otimes F_n_\ell \) specified by \( k = (k_1, \ldots, k_\ell) \) with \( n_\ell = 2^{k_\ell} \) are \( T(\text{pid}, c, k), A(\text{pid} + P, c, k), \ldots, A(\text{pid} + (N/P - 1) \cdot P, c, k) \) for \( c = 1, \ldots, K \). Depending on the schedule, these addresses may be local or remote and there will be varying amounts of contention on the interconnect.

### 3.2.1 Matlab Simulation of Distributed Memory FFT Processor

In order to demonstrate and specify the design of the distributed memory universal FFT processor a Matlab simulation is provided. The program uses a two-dimensional array to simulate distributed memory (the first index is the mid and the second index is
the offset). Computation is performed in stages where each stage simulates parallel computation of the butterfly tasks. Tasks are allocated using the round-robin schedule discussed in the previous section. A function call parameterized by the pid simulates each processing element. The number of processors is a runtime parameter. Each processing element generates its own sequence of tasks using its pid. The code for the processing elements uses the functions from the previous section to generate addresses and twiddle factors. The function \texttt{mdbr} from the previous chapter is used to simulate loading the data.

Figure 3.12 shows the Matlab program for the distributed memory Pease machine. The function \texttt{md-pe machine} calls \texttt{md-pe}, see Figure 3.13, which simulates the execution of the processing elements. These programs can be tested in the same way that the Matlab implementations of the dimensionless FFT in Section 2.4 were tested.

The Matlab implementation of the Pease machine can easily be modified to simulate other universal FFT processors. All that is required is to change the functions \texttt{addr} and \texttt{md-twiddle} in the function simulating the processing element. For example, a Cooley-Tukey machine could be obtained by using the Cooley-Tukey address and twiddle functions from the previous section.
% Distributed Memory Universal Pease Machine
% usage: md_p_machine(N,K,P,x,t,Dim)
% Inputs
% K : non-negative integer.
% N : positive integers with N = 2^K.
% P : positive integer.  P = 2^p, P <= N/2.
% x : vector of size N
% t : Number of Dimensions
% Dim : Dimension Vector = (k_1,...,k_t), where k_1 + ... + k_t = K.
% Output
% y : vector of size N.  y = (F_{n_1} tensor ... tensor F_{n_t})x,
% where n_i = 2^{k_i}.

function y = md_p_machine(N,t,P,x, mxDim, Dim)

global M;  % P x (N/P) matrix representing distributed memory

M = reshape(md_br(x,Dim),N/P,P)';  % map to distributed memory
Dim_idx = mxDim;
for c = t-1:-1:0
    for pid = 0:P-1
        md_p_pe(N,t,P,pid,c,Dim,Dim_idx);
    end
    if t-c >= sum(Dim(1:Dim_idx))
        Dim_idx = Dim_idx - 1;
    end
end
y = reshape(M',N,1);

Figure 3.12: Matlab Implementation of the distributed Pease FFT Processor
function md_p_pe(N,t,P,pid,c,Dim,Dim_Idx)

global M; F2 = [1 1; 1 -1];
for r = pid:P:2^(t-1)-1
    a = addr(t,r,c);
    m1 = mid(N,P,a(1)); o1 = moffset(N,P,a(1));
    m2 = mid(N,P,a(2)); o2 = moffset(N,P,a(2));
    w = md_twiddle_p(r,c,Dim,Dim_Idx);
    yt(1,1) = M(m1+1,o1+1);
    yt(2,1) = M(m2+1,o2+1);
    yt = F2 * (yt .* [1;w]);
    M(m1+1,o1+1) = yt(1);
    M(m2+1,o2+1) = yt(2);
end

Figure 3.13: Matlab Implementation of the distributed Pease Processing Element
4 Architecture Design

This chapter discusses the hardware implementation of the Universal FFT Machine, called the Main Processing Element(MPE), shown in figure(3.2.1). The MPE is the top level of the entire design for the on-board FPGA. There is a HOST which passes the data, transform size and dimension information to the MPE for the FFT. From there the MPE performs the FFT and the HOST retrieves the data. The specific parts of the MPE that accomplish these tasks are the Interface Unit(IU) handling communication to and from the HOST, the Inter Connection Network(ICN) handling communication between the PEs and the IU, and the Processing Elements(PE) which perform the FFT in a distributed manner.

![Diagram of Distributed Memory FFT Processor](image)

Figure 4.1: Distributed Memory FFT Processor
4.1 Interface Unit (IU)

The Interface Unit, shown in figure (4.1), is responsible for handling the communications with the HOST. The HOST transmits the size, dimension and data of the transform that is processed by the MPE. The FFT can have as many dimension parameters as there are bits to decode the address; therefore, the IU first decodes the size and dimension of the transform that it is performing. These parameters, describing the size and dimension of the FFT, are sent from the IU to the PEs to perform the transform. The IU decodes the locations of the dimensions and correctly permutes the data as it is passed it to the proper PEs. The communication from the IU is always sent across the ICN to the proper PE. The IU consists of an input FIFO that

![Image of Interface Unit diagram]

**Figure 4.2: Interface Unit**

queues the data passed by the HOST. It will have a register describing the size and dimension of the transform that is written to by the HOST. The IU begins decoding the dimension information after the HOST has given the enable signal. Once the IU has completed decoding the size and dimension information, it passes that informa-
tion to all of the PEs. The IU continues by passing the input data, in a permuted manner described by the dimension, to all of the PEs. During this operation the HOST keeps the input FIFO full so the IU will not need to stall in the loading phase.

The IU configures the ICN to pass the parameters and the data to the PEs. There is a Processor Identification (PID) number associated with each piece of data, in order for, the ICN to know where to route the data. There is also, a simple communication protocol or enable signal for the ICN and IU to verify that the data is accepted.

4.2 Inter Connection Network (ICN)

The Inter Connection Network is a cross-bar which connects the PEs and the IU together. Since we are using a distributed memory model, each PE has its own memory; therefore, some data needs to be passed from one PE to another. The ICN connects to the IU and the PEs for communication and routes the proper data to the proper PE by use of the PID.

The smallest FFT that we are computing is of Radix-2 and requires two data points. At any given iteration there will be two pieces of data to be computed. The data may need to be routed between as many as two different PEs. The ICN needs to handle these communications from different PEs trying to pass data from one PE to another or from two PE’s, at once, to another. An arbitration scheme, or controller handling the requests to transmit and receive data for each PE, is required. It is this complexity that makes the design of the ICN for the general case a difficult task. For some algorithms there is a known pattern in the communication scheme, that may be used in the design of a specialized ICN.
4.3 Processing Element (PE)

The Processing Element (PE), shown in figure (4.3) is responsible for the distributed FFT computation. This unit performs the computation with the use of an addressing and twiddle scheme that is generated locally for the PE. Even though each PE is performing the same operation, the PEs are asynchronous to each other by iteration but not by stage. At the beginning of each remote stage the PEs synchronize to guarantee that the data being passed is valid. The PEs route the data to the proper PE over the ICN. The FFT as discussed earlier is broken down into $F_2$'s. These $F_2$'s are accomplished by a Computation Unit (CU) which performs the butterfly operation. For this butterfly operation to be completed, a twiddle factor needs to be fed into the CU. These are generated by the Twiddle Factor Generator (TG). The current address for the pair of data points is given by the Address Generator (AG), which can generate all the different addresses needed for the FFT. The PE needs to be able to handle communication with its local memory and the ICN. These are handled by
the Interconnection and Memory Interfaces. While performing the computation, the Data Control Unit (DCU) ensures that the CU does not have to stall by queuing data and Twiddle Factors. The DCU also handles the AG and queues the addresses for the data to be stored, back into the proper memory location.

### 4.3.1 Computation Unit (CU)

The CU performs the Radix-2 FFT computation. This computation is comprised of three inputs $X_0, X_1,$ and $W_n$. It then takes these inputs and performs the butterfly operation, shown in figure (4.4). The CU

![Figure 4.4: Basic Butterfly Operation](image)

performs the operations $X_0 + 1 \times X_1 = Y_0$ and $X_0 + W_n \times X_1 = Y_1$. The outputs $Y_0$ and $Y_1$ are stored in an output queue where they are taken by the DCU when ready and stored back into the proper memory location. The proper $W_n$, or twiddle factor, is generated by the TG. This data is stored in a queue and given to the CU at the proper time, with the corresponding input data that was generated by the AG.

### 4.3.2 Twiddle Factor Generator (TG)

The Twiddle Factor Generator (TG) creates the Twiddle Factors needed to compute the FFT. These are queued and fed into the CU at the appropriate time for the butterfly operation to be completed, successfully. This device is complicated since it needs to change due to the different size transforms and also how the dimensions are defined.
4.3.3 Address Generator (AG)  The Address Generator (AG) creates all the local addresses needed to perform the FFT for the PE. The AG begins in the loading stage, where it generates that address for the initial data. This is defined by the dimension information that are given to the PEs by the IU. When the loading is complete the AG will decode the address that corresponds to the current stage and iteration. The data is fetched from memory and processed by the CU for the transform. The AG receives the stage and an iteration to generate an address. The addresses have an associated PID that decodes the specific PE to which the data is going, and in return where the data for that point will be coming from if it is remote. This was accomplished with a set of muxes. The muxes route a certain bit of the

![Diagram](image)

**Figure 4.5: N number of Mux-N**

address to the proper location. This is shown in 4.5, where each bit of the input is sent to the muxes and then collected by the “OR” gate at the end. The “OR” gate re-combines the signal with the correct permutation of the address. This permutation is done for the entire stage. The addresses will start at 0 and increment till the PE
has completed the stage. The next stage will be sent to the AG, which will in-turn generate the control signals for the muxes in that stage. The addresses will start over again at 0 and increment till the end of that stage.

The PE tells the AG the stage and, from there, the AG decodes the mux control signals. When this is complete the AG is enabled for decoding the address and the PE may request the first permutation of the addresses. This is done by applying an input iteration to the AG. The PID bits are generated at the same time with the local memory address. The control signals for the muxes are the only difference between the Pease and Cooley-Tukey algorithms; therefore, to change the AG for a different algorithm, only the mux signals need to be modified.

4.3.4 Data Control Unit (DCU)  The Data Control Unit (DCU) is comprised of state machines and internal queues that connect the devices of the PE together. The state machines control the operation of the Address Generator, Twiddle Generator, Inter Connection Network, and Memory Interface. The DCU’s internal queues buffer the data between the Memory, ICN and CU. The DCU also stores the local addresses which will be used to write the data back into the local memory. First considering the local stages, where data for a stage is from the PEs own memory are considered. This data is passed to the CU, where it is computed with the proper twiddle factor, and then sent back out to be stored locally. This data flow is possible only for the local stages of the FFT. The second set of stages are remote, where one or possibly both pieces of data can be from other PEs. For some algorithms of the FFT the PEs will be passing most of their data. The particular algorithm is chosen depending on how the internal stages and addressing is generated. A worse-case would be if the data was from two different PEs and then processed by a third. When finished, the data will be sent back to the PEs who originated the data. It is important that the remote stages happen with minimum overhead.
Queues have to be maintained in order that the CU does not stall due to lack of input. CU stalls during the transform are a waste of resources and need to be minimized. At the same time, only a certain amount of data can be queued before it becomes excessive. The data used locally stays, and data needed to be sent to remote processors get routed to the proper PE. Since the operation is pair-wise, we know that the remote data arriving in from a PE is used in the transform as either $X_0$ or $X_1$.

In the worse-case described earlier we had two PEs feeding data to a third. This is the most difficult situation to deal with, since the data is coming from two different PEs. The data order could be backwards where $X_1$ was first and $X_0$ was second. The PE is assuming that $X_0$ will arrive first; and therefore, needs to be queued into the proper location. This ordering is handled by the PID. Which denotes what PE has $X_0$ and $X_1$. The algorithm is recognized by the AG and it will have to communicate this information to the DCU.
4.3.5 Inter Connection Network Interface (ICN-IF)  The ICN-IF communicates with the ICN. The ICN-IF allows any changes to the ICN to be made in this component. This ensures plug-and-play capabilities of the PE. In other words, if the ICN were to be modified for a more stringent protocol the change would be made here, by plugging a new ICN-IF into the PE instead of having to change intricate state machines in the DCU.

4.3.6 Memory Interface (MEM-IF)  The MEM-IF is for the PE to communicate to the external memory that is located on the board. If there were a different memory architecture, only the Memory Interface (MEM-IF) for the DCU would need to be changed; the DCU state machines would not need to be modified.
5 Hardware Implementation

This chapter discusses the hardware behavior model of the Fast Fourier Processor that was created. This behavior model is shown in Appendix A and is broken into three major sections: the Processing Element, the Inter Connection Network, and the test bench.

5.1 Processing Element

The processing element (A.1) is a structural description of how the entire design is wired together. The heart of the Processing Element is the computation unit which has two input queues, called the $X_1$ and $X_2$ data queues. When both the input queues and the twiddle factor queue have data, they send a ready signal for the Computation Unit to process that iteration of the transform. The X data queues are filled by either the local memory or the Inter Connection Network’s interface (ICN-IF). The ICN-IF can send data to the X queues to be processed or send data that has been computed by a remote Processing Element back to the memory. The input ports of the ICN-IF are wired to the memory, for sending data to remote processors to be computed, or the Computation unit, for sending computed data back to remote Processing Elements. There are two state machines that control the flow of this data and they are the Input, or X State Machine, and the output, or Y State Machine.

5.1.1 The FFT Package

This FFT Package (A.2) is loaded at compile time and used to define the number of processors. It also defines a record type called the global information bus. This bus is connected to all the Processing Elements and its information is sent by the Interface Unit (IU). This Global information is used only at the beginning of the transform to set the starting parameters. These parameters
are: the size of the transform, number of stages, and initial permutation information for the address generators to store the first set data give by the IU.

5.1.2 Cooley-Tukey Address Generator

The Address Generator (A.3) was written for the Cooley-Tukey algorithm. It utilizes a series of muxes to permute the address bits. The generator re-creates the bit permutation scheme for the Cooley-Tukey algorithm by shifting a register that contains the schedule of address bits per stage. The schedule is shifted as defined by the algorithm for each stage. The generator will produce a certain Memory ID indicating where the data is going and where the data is from for that particular iteration, and the local offset for its own memory bank.

5.1.3 Cooley-Tukey Twiddle Generator

The Twiddle Generator (A.4) creates and stores the twiddle factors or constants, for the transform. In the first stage, this device generates the constants described by the Cooley-Tukey algorithm, size, stage, and dimension. It stores these constants in a local queue and monitors the queue to keep it full. Whenever a constant is used by the Computation Unit, the Twiddle Generator will create and store the next constant in its schedule. The Twiddle Generator keeps track of this schedule by counting iterations and stages of Twiddle Constants that is generating. This is done by the knowing the size and stages of the transform, and is passed to the Generator by the Global Information bus that is sent to the Processing Elements at start up by the Interface Unit.

5.1.4 Data and Address queues

The Address (A.12) and Data (A.11) queues perform as basic FIFOs with some added abilities for the Address Queue. The Address Queue, like the Data Queue, stores input data as a FIFO. The first data that is placed in, is the first that is taken out. The difference in the Address Queue is when data is
taken out. There is an enable that will store the local address in the Address Queue to an internal secondary queue for remote data that may not be present at that time. Since the remoted data will not always be ready at the time the local address is needed, it is stored in a secondary queue and taken out as remote data comes in from the ICN-IF.

5.1.5 Input and Output State Machines  The Data Control Unit (DCU) is comprised of two different state machines. These state machines are responsible for controlling the muxes that connect the memory to the $X_1$ and $X_2$ input queues, the read, and write signals on the ICN-IF. The input or X State Machine (A.6) controls the initial loading and queueing of the data for the computation. The X State Machine will send the current iteration and stage to the Address Generator which will create the Memory IDs and the Local Offset for that iteration. The output or Y State Machine is then stored into the secondary queue in the address queue to be used later. These two state machines make up the DCU.

5.1.6 Inter Connection Network Interface  The ICN-IF is the Processing Element’s set of queues to the ICN. There are two parts to the device, sending (A.10) and receiving (A.9). When data needs to be sent, if it requires processing it is sent from the X State Machine. Otherwise, if it needs to be returned to storage to memory, it is sent from the Y State Machine. The ICN-IF will queue this data and send a flag, containing the address of the Processing Element to which the data needs to be sent, to the ICN. There are two different sending queues: computation or storing. The second half of the ICN-IF is for receiving data. This is done by receiving a signal from the ICN that tells the interface whether the data was for computation or storing. This ICN will tell the interface this information when it sends the data and an enable signal to take the data. Once there is data ready to be taken out of the ICN-IF, flags
are generated and the X or Y State Machine will take this data out.

5.1.7 Memory and Memory Interface For this behavior model the memory (A.13) was simulated as a simple array. There was an request signal for data from memory, and it was assumed that at the following clock that data would be present. This model was simulated and can be replaced by an actual memory controller in future implementations of the hardware.

5.2 Inter Connetion Network

The ICN (A.14) is the device that connects the Processing Elements together. This device has an arbiter that scans through the Processing Elements in a cyclic manner. This device goes through each Processing Element and checks to see if they have data that needs to be sent. This data is then put in an output register and a flag is set. The other Processing Element receives an ID that tells it where the data came from. The ICN has one extra connection. This connection is to the IU and allows the data to come in and be sent to a specific Processing Element, and is used only for loading but can be configured for unloading.

5.3 FFT Machine Test Bench

The Test Bench (A.15) takes the ICN and wires it to an array of $2^p$ Processing Elements. The Test Bench acts as the IU or Interface Unit between the Host and the array of Processing Elements. This device is dependant on the hardware architecture that the FFT Machine is implemented on. To simulate the FFT Machine the IU needs to be simulated. The IU would pass the size, data, and dimension information to the Processing Elements to perform the transform. The Test Bench sends this necessary information across the Global Information bus and passes the data across the ICN
to be distributed to the Processing Elements. When all Processing Elements have completed the transform, their memories are programmed to dump all the data to a file where it can be manually inspected to see that the transform was complete.
Bibliography


A Appendix: Hardware Implementation

A.1 Processing Element

--
-- VHDL Entity thesis.PE.symbol
--
-- Created:
--                by - Mike Balog
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

LIBRARY thesis;
USE thesis.fft_pak.all;

ENTITY PE IS
  GENERIC(
    PID : natural := 0
  );
  PORT(
    clk : IN std_logic;
    glb_info : IN tranfm_info;
    icn_ctl_in : IN DataType;
    icn_data_in : IN complex;
    icn_mid_in : IN natural;
    icn_out_ack : IN std_logic;
    reset : IN std_logic;
    icn_ctl_out : OUT DataType;
    icn_data_out : OUT complex;
    icn_in_ack : OUT std_logic;
    icn_mid_out : OUT natural;
    iu_pe_done : OUT std_logic
  );
END PE ;
-- VHDL Architecture thesis_PE.struct
--
-- Created:
--    by - Mike Balog
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

LIBRARY thesis;
USE thesis.FFT_pak.ALL;

LIBRARY thesis;

ARCHITECTURE struct OF PE IS

-- Architecture declarations

-- Internal signal declarations
SIGNAL X_data_from_icn : complex;
SIGNAL X_data_to_f2     : complex;
SIGNAL Y_data_from_icn : complex;
SIGNAL addr_w          : natural;
SIGNAL aq_e            : std_logic;
SIGNAL aq_e_l          : std_logic;
SIGNAL aq_e_r          : std_logic;
SIGNAL aq_ff           : std_logic;
SIGNAL aq_mode         : a_que_mode;
SIGNAL aq_r_en         : std_logic;
SIGNAL aq_w_en         : std_logic;
SIGNAL f2_ie           : std_logic;
SIGNAL f2_oe           : std_logic;
SIGNAL f2_y_e         : std_logic;
SIGNAL icn_XQ_e        : std_logic;
SIGNAL icn_XQ_ff       : std_logic;
SIGNAL icn_YQ_e        : std_logic;
SIGNAL icn_YQ_ff       : std_logic;
SIGNAL icn_x_r_en      : std_logic;
SIGNAL icn_x_w_en      : std_logic;
SIGNAL icn_y_r_en      : std_logic;
SIGNAL icn_y_w_en : std_logic;
SIGNAL icn_yq_xsm_r_en : std_logic;
SIGNAL icn_yq_ysm_r_en : std_logic;
SIGNAL iter : natural;
SIGNAL load_stg : std_logic;
SIGNAL lof : natural;
SIGNAL lof_out : natural;
SIGNAL mem_data_in : complex;
SIGNAL mem_data_out : complex;
SIGNAL mem_r_en : std_logic;
SIGNAL mem_w_en : std_logic;
SIGNAL mem_x_w_en : std_logic;
SIGNAL mem_y_w_en : std_logic;
SIGNAL stage : natural;
SIGNAL tw_in : complex;
SIGNAL x0_data : complex;
SIGNAL x0_data_ee : std_logic;
SIGNAL x0_data_ff : std_logic;
SIGNAL x0_w_en : std_logic;
SIGNAL x1_data : complex;
SIGNAL x1_data_ee : std_logic;
SIGNAL x1_data_ff : std_logic;
SIGNAL x1_w_en : std_logic;
SIGNAL x_01_ready : std_logic;
SIGNAL x_done : std_logic;
SIGNAL x_ff : std_logic;
SIGNAL x_mid_f : natural;
SIGNAL x_mid_from_icn : natural;
SIGNAL x_mid_t : natural;
SIGNAL x_rl : std_logic;
SIGNAL y_data_from_f2 : complex;
SIGNAL y_done : std_logic;
SIGNAL y_mid : natural;
SIGNAL y_rl : std_logic;

SIGNAL iu_pe_done_internal : std_logic;

-- Component Declarations
COMPONENT CT_TG
PORT ( 
    clk : IN std_logic ;
    f2_ie : IN std_logic ;
    glb_info : IN tranfm_info ;
    loading : IN std_logic ;
);
reset : IN    std_logic;
tw_in : OUT   complex
);
END COMPONENT;
COMPONENT ICN_IF
PORT (  clk             : IN    std_logic;
icn_ctl_in      : IN    DataType;
icn_data_in     : IN    complex;
icn_mid_in       : IN    natural;
icn_out_ack      : IN    std_logic;
icn_x_r_en       : IN    std_logic;
icn_x_w_en       : IN    std_logic;
icn_y_r_en       : IN    std_logic;
icn_y_w_en       : IN    std_logic;
mem_data_out     : IN    complex;
reset            : IN    std_logic;
x_mid_t           : IN    natural;
y_data_from_f2    : IN    complex;
y_mid             : IN    natural;
x_data_from_icn   : OUT   complex;
y_data_from_icn   : OUT   complex;
icn_XQ_ee        : OUT   std_logic;
icn_XQ_ff         : OUT   std_logic;
icn_YQ_ee         : OUT   std_logic;
icn_YQ_ff         : OUT   std_logic;
icn_ctl_out       : OUT   DataType;
icn_data_out      : OUT   complex;
icn_in_ack        : OUT   std_logic;
icn_mid_out        : OUT   natural;
x_mid_from_icn    : OUT   natural
);
END COMPONENT;
COMPONENT X_sm
GENERIC (  pid : natural := 0;
            q_size : natural := 5
);
PORT (  clk             : IN    std_logic;
gib_info          : IN    tranfm_info;
icn_XQ_ee         : IN    std_logic;
icn_XQ_ff         : IN    std_logic;
icn_YQ_ee         : IN    std_logic;
reset             : IN    std_logic;
x_ff : IN std_logic;
x_mid_f : IN natural;
x_mid_from_icn : IN natural;
x_mid_t : IN natural;
aq_w_en : OUT std_logic;
icn_x_r_en : OUT std_logic;
icn_x_w_en : OUT std_logic;
icn_yq_xsm_r_en : OUT std_logic;
iter : OUT natural;
load_stg : OUT std_logic;
mem_r_en : OUT std_logic;
mem_x_w_en : OUT std_logic;
stage : OUT natural;
x0_w_en : OUT std_logic;
x1_w_en : OUT std_logic;
x_done : OUT std_logic;
x_r1 : OUT std_logic
);
END COMPONENT;
COMPONENT y_sm
;1;1;128;128;1;0x;1;1;128;128;1;0x GENERIC (pid : natural := 0);
PORT (aq_e : IN std_logic;
aq_e_l : IN std_logic;
aq_e_r : IN std_logic;
clk : IN std_logic;
f2_y_e : IN std_logic;
icn_YQ_e : IN std_logic;
icn_YQ_ff : IN std_logic;
load_stg : IN std_logic;
reset : IN std_logic;
x_done : IN std_logic;
y_mid : IN natural;
aq_mode : OUT a_que_mode;
aq_r_en : OUT std_logic;
f2_oe : OUT std_logic;
icn_y_w_en : OUT std_logic;
icn_yq_ysm_r_en : OUT std_logic;
mem_y_w_en : OUT std_logic;
y_done : OUT std_logic;
y_r1 : OUT std_logic
);
END COMPONENT;
COMPONENT a_que
GENERIC (  
    q_size : natural := 10
)
);
PORT (  
    clk    : IN    std_logic ;
    reset  : IN    std_logic ;
    lof_in : IN    natural ;
    mid_in : IN    natural ;
    aq_ea : OUT   std_logic ;
    w_en   : IN    std_logic ;
    mode   : IN    a_que_mode ;
    r_en   : IN    std_logic ;
    aq_ff  : OUT   std_logic ;
    aq_ea_rq : OUT std_logic ;
    lof_out : OUT  natural ;
    mid_out : OUT  natural
);
END COMPONENT;
COMPONENT ct_perm
GENERIC (  
    pid : natural := 0
)
);
PORT (  
    glb_info : IN    trnfm_info ;
    iter     : IN    natural ;
    stage    : IN    natural ;
    reset    : IN    std_logic ;
    mid_to   : OUT   natural ;
    mid_from : OUT   natural ;
    lof      : OUT   natural
);
END COMPONENT;
COMPONENT d_que
GENERIC (  
    q_size : natural := 10
)
);
PORT (  
    clk    : IN    std_logic ;
    reset  : IN    STD_LOGIC ;
    data_in : IN    complex ;
    data_ea : OUT   std_logic ;
    w_en   : IN    std_logic ;
    r_en   : IN    std_logic ;
    data_ff : OUT   std_logic ;
data_out : OUT complex
);
END COMPONENT;

COMPONENT f2_unit
PORT ( 
clk      : IN  std_logic ;
oe       : IN  std_logic ;
reset    : IN  std_logic ;
tw_in    : IN  complex ;
x0       : IN  complex ;
x1       : IN  complex ;
x_01_ready : IN  std_logic ;
ie       : OUT std_logic ;
y       : OUT complex ;
y_e    : OUT  std_logic
);
END COMPONENT;

COMPONENT memory_array
GENERAL ( 
  pid     : natural := 0;
  absize  : natural := 6
);
PORT ( 
addr_r   : IN  natural ;
addr_w   : IN  natural ;
clk       : IN  std_logic ;
data_in    : IN  complex ;
w_en     : IN  std_logic ;
r_en     : IN  std_logic ;
data_out  : OUT complex
;1;1;128;128;1;0x;1;1;128;128;1;0x   pe_done  : IN  std_logic
);
END COMPONENT;

-- Optional embedded configurations
-- pragma translate_off
FOR ALL : CT_TG USE ENTITY thesis.CT_TG;
FOR ALL : ICN_IF USE ENTITY thesis.ICN_IF;
FOR ALL : X_sm USE ENTITY thesis.X_sm;
FOR ALL : Y_sm USE ENTITY thesis.Y_sm;
FOR ALL : a_que USE ENTITY thesis.a_que;
FOR ALL : ct_perm USE ENTITY thesis.ct_perm;
FOR ALL : d_que USE ENTITY thesis.d_que;
FOR ALL : f2_unit USE ENTITY thesis.f2_unit;
FOR ALL : memory_array USE ENTITY thesis.memory_array;
-- pragma translate_on

BEGIN
  -- Architecture concurrent statements
  -- HDL Embedded Text Block 1 AQ_and
  -- AQAND
  aq_ee<=aq_ee_1 and aq_ee_r;

  -- HDL Embedded Text Block 2 Mux_Y
  -- Mux_X 1
  with y_rl select
    mem_data_in <=
      y_data_from_f2 when '0',
      Y_data_from_icn when '1',
      math_czero when others;

  -- HDL Embedded Text Block 3 X_ready
  -- ANDX
  x_01_ready <= (not x0_data_ee) and (not x1_data_ee);

  -- HDL Embedded Text Block 4 MUX_AG
  -- MUX_AG 4
  process(lof,lof_out,load_stg)
  begin
    if load_stg = '1' then
      addr_w <= lof;
    else
      addr_w <= lof_out;
    end if;
  end process;

  -- HDL Embedded Text Block 5 Mux_X
  -- Mux_X 1
  with x_rl select
    x_data_to_f2 <=
      mem_data_out when '0',
      X_data_from_icn when '1',
      math_czero when others;

  -- HDL Embedded Text Block 7 donesig
  -- DONESIG
  iu_pe_done_internal <= x_done and y_done;

  -- HDL Embedded Text Block 8 mw_en
  -- ANDX
mem_w_en <= mem_y_w_en or mem_x_w_en;

-- HDL Embedded Text Block 9 X_ff
-- XY_EE
x_ff <= x0_data_ff or x1_data_ff;

-- HDL Embedded Text Block 10 icn_yq_r_en
-- ANDX
icn_y_r_en <= icn_yq_ysm_r_en or icn_yq_xsm_r_en;

-- Instance port mappings.
TG : CT_TG
PORT MAP (
  clk => clk,
  f2_ie => f2_ie,
  glb_info => glb_info,
  loading => load_stg,
  reset => reset,
  tw_in => tw_in
);

icnf : ICN_IF
PORT MAP (
  clk => clk,
  icn_ctl_in => icn_ctl_in,
  icn_data_in => icn_data_in,
  icn_mid_in => icn_mid_in,
  icn_out_ack => icn_out_ack,
  icn_x_r_en => icn_x_r_en,
  icn_x_w_en => icn_x_w_en,
  icn_y_r_en => icn_y_r_en,
  icn_y_w_en => icn_y_w_en,
  mem_data_out => mem_data_out,
  reset => reset,
  x_mid_t => x_mid_t,
  y_data_from_f2 => y_data_from_f2,
  y_mid => y_mid,
  X_data_from_icn => X_data_from_icn,
  Y_data_from_icn => Y_data_from_icn,
  icn_XQ_ee => icn_XQ_ee,
  icn_XQ_ff => icn_XQ_ff,
  icn_YQ_ee => icn_YQ_ee,
  icn_YQ_ff => icn_YQ_ff,
  icn_ctl_out => icn_ctl_out,
  icn_data_out => icn_data_out,
  icn_in_ack => icn_in_ack,
icn_mid_out  => icn_mid_out,
x_mid_from_icn => x_mid_from_icn
);
xsm : X_sm
GENERIC MAP ( pid => pid, q_size => 5 )
PORT MAP ( clk => clk,
    glb_info  => glb_info,
    icn_XQ_ee => icn_XQ_ee,
    icn_XQ_ff => icn_XQ_ff,
    icn_YQ_ee => icn_YQ_ee,
    reset    => reset,
    x_ff     => x_ff,
    x_mid_f  => x_mid_f,
    x_mid_from_icn => x_mid_from_icn,
    x_mid_t  => x_mid_t,
    aq_w_en  => aq_w_en,
    icn_x_r_en => icn_x_r_en,
    icn_x_w_en => icn_x_w_en,
    icn_yq_xsm_r_en => icn_yq_xsm_r_en,
    iter    => iter,
    load_stg => load_stg,
    mem_r_en => mem_r_en,
    mem_x_w_en => mem_x_w_en,
    stage   => stage,
    x0_w_en => x0_w_en,
    x1_w_en => x1_w_en,
    x_done  => x_done,
    x_rl    => x_rl
);
ysm : Y_sm
GENERIC MAP ( pid => pid )
PORT MAP ( aq_e  => aq_e,
aq_e_l => aq_e_l,
aq_e_r => aq_e_r,
clk   => clk,
f2_y_e  => f2_y_e,
icn_YQ_ee => icn_YQ_ee,
icn_YQ_ff => icn_YQ_ff,
load_stg   => load_stg,
reset     => reset,
x_done    => x_done,
y_mid     => y_mid,
aq_mode   => aq_mode,
aq_r_en   => aq_r_en,
f2_oe     => f2_oe,
icn_y_w_en => icn_y_w_en,
icn_yq_ysm_r_en => icn_yq_ysm_r_en,
mem_y_w_en => mem_y_w_en,
y_done    => y_done,
y_rl      => y_rl
);
AQ : a_que
GENERIC MAP (  
  q_size => 22
)
PORT MAP (  
  clk   => clk,
  reset => reset,
  lof_in => lof,
  mid_in => x_mid_t,
  aq_ee  => aq_ee_l,
  w_en   => aq_w_en,
  mode   => aq_mode,
  r_en   => aq_r_en,
  aq_ff  => aq_ff,
  aq_ee_rq => aq_ee_r,
  lof_out => lof_out,
  mid_out => y_mid
);
AG : ct_perm
GENERIC MAP (  
  pid => PID
)
PORT MAP (  
  glb_info => glb_info,
  iter    => iter,
  stage   => stage,
  reset   => reset,
  mid_to  => x_mid_t,
  mid_from => x_mid_f,
  lof    => lof
);
X1Q : d_que
GENERIC MAP (  
    q_size => 10  
)  
PORT MAP (  
    clk => clk,  
    reset => reset,  
    data_in => X_data_to_f2,  
    data_ee => x1_data_ee,  
    w_en => x1_w_en,  
    r_en => f2_ie,  
    data_ff => x1_data_ff,  
    data_out => x1_data  
);  
XOQ : d_que  
GENERIC MAP (  
    q_size => 10  
)  
PORT MAP (  
    clk => clk,  
    reset => reset,  
    data_in => X_data_to_f2,  
    data_ee => x0_data_ee,  
    w_en => x0_w_en,  
    r_en => f2_ie,  
    data_ff => x0_data_ff,  
    data_out => x0_data  
);  
f2unit : f2_unit  
PORT MAP (  
    clk => clk,  
    oe => f2_oe,  
    reset => reset,  
    tw_in => tw_in,  
    x0 => x0_data,  
    x1 => x1_data,  
    x_01_ready => x_01_ready,  
    ie => f2_ie,  
    y => y_data_from_f2,  
    y_ee => f2_y_ee  
);  
MEMIF : memory_array  
GENERIC MAP (  
    pid => PID,  
    absize => absize  
)
PORT MAP (  
addr_r  => lof,  
addr_w  => addr_w,  
clk     => clk,  
data_in  => mem_data_in,  
w_en    => mem_w_en,  
r_en    => mem_r_en,  
data_out => mem_data_out,  
pe_done => iu_pe_done_internal  
);  

-- Implicit buffered output assignments  
iu_pe_done <= iu_pe_done_internal;  

END struct;  

A.2 FFT Package  

--  
-- VHDL Package Header thesis.fft_pak  
--  
LIBRARY ieee;  
USE ieee.std_logic_1164.all;  
use ieee.math_complex.all;  
PACKAGE fft_pak IS  
------------------------------------------------------------------  
-- This is needed to be defined at Compile time and therefore need to be  
-- defined in this package and changed. Only the number of processing  
-- elements and the number of bits to decode them.  
------------------------------------------------------------------  
constant P : natural := 4;  -- Number of Processing elements  
constant Pb : natural := 2;  -- Processor ID bus size  
constant abs_size : natural := 4;  -- size of mem banks  
------------------------------------------------------------------  
constant ab : natural := 20;  -- Max size of Transform  
constant LOF_size : natural := ab-Pb;  

Type MuxSel_array is array (natural range <>) of natural;  
Type dv_array is array (natural range <>) of natural;  
Type Data_Type is (NO_OP,T0_X,T0_Y,Loading);  
type icn_data_array is array (natural range <>) of complex;
type icn_mid_array is array (natural range <>) of natural;
type icn_ctl_array is array (natural range <>) of DataType;
Type tranfm_info is record
  num_pe : natural;
  size_two : natural;
  --2^\text{size_two}=\text{size_nat}
  size_nat : natural;
  --size of transform natural
  size_bit : std_logic_vector(ab-1 downto 0);
  -- Size of Transfrom in bits
  stage_max: natural;
  --number of stages(incl. loading)
  dv_bit : std_logic_vector(ab-1 downto 0);
  dv_nat : dv_array(ab-1 downto 0);
  dv_len : natural; -- length of \text{dv}
  init_perm : MuxSel_array(ab-1 downto 0);
end record;
Type a_que_mode is (remote, local, store);
END fft_pak;

A.3 Cooley-Tukey Address Generator

--
-- VHDL Architecture thesis.ct_perm.sim
--
-- Created:
-- by - Mike Balog
-- at - 11:47:14 04/09/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
library thesis;
use thesis.FFT_pak.all;

ENTITY ct_perm IS
  GENERIC(
    pid : natural := 0
  );
PORT(
    glb_info : IN tranfm_info;
    iter : IN natural;
    stage : IN natural;
    reset : IN std_logic;
    mid_to : OUT natural;
    mid_from : OUT natural;
    lof : OUT natural
);
END ct_perm ;

ARCHITECTURE sim OF ct_perm IS
component NMUXN
    GENERIC(
        size_mux : natural := 20
    );
    PORT(
        mux_in : IN natural;
        mux_sel : IN muxsel_array (size_mux-1 DOWNTO 0);
        mux_out : OUT natural
    );
end component;
    -- pragma translate_off
FOR ALL : NMUXN USE ENTITY thesis.NMUXN;
    -- pragma translate_on

signal WHOLE_MID_TO, WHOLE_MID_FROM,WHOLE_LOF,TRSH : natural;
signal LOF_sel,MID_FROM_sel,MID_TO_sel
    : muxsel_array(ab-1 downto 0);

begin
    -- LOF and MID generation
    with reset select
        TRSH <=
            0 when '1',
            conv_integer(glb_info.size_two_pb) when others;
    -- Memory ID FROM generation
    MID_FROM <= conv_integer(shr(conv_unsigned(WHOLE_MID_FROM,ab),
                                         (conv_unsigned(TRSH,ab ) ) ) ) ;
    -- Memory ID TO generation
    MID_TO <= conv_integer(shr(conv_unsigned(WHOLE_MID_TO,ab),
                                         (conv_unsigned('1',ab ) ) ) ) ;
    -- Local Offset Calculations and address bit swap
    -- till the last address
    LOF <= conv_integer(unsigned(conv_std_logic_vector(WHOLE_LOF,ab)
and conv_std_logic_vector((2**TRSH)-1,ab));

-- L0F select signals generated in this process
process(reset,stage)
var d : natural;
var size,shft : natural;
var temp, temp_to : muxsel_array(ab-1 downto 0);
begin
if reset = '1' then
  temp := (others => ab+1); -- init dummy value
  temp_to := (others => ab+1); -- init dummy value
  shft := 1;
else
  if stage = glb_info.stage_max then
    -- set up the initial param
    for i in glb_info.size_two - 1 downto 0 loop
      temp(i) := i;
    end loop;
    for i in pb downto 1 loop
      temp_to(i) := glb_info.size_two - 1 - pb +i;
    end loop; -- i
    temp_to(0) := 0;
    MID_TO_Sel <= temp_to;
    MID_FROM_sel <= temp;
    L0F_sel <= glb_info.init_perm;
  elsif stage > 0 then
    size := glb_info.size_two - stage;
    -- size of the local offset
    if size > 0 then
      d := temp(size-1);
      temp(size-1) := temp(size);
      temp(size) := d;
    end if;
    if stage <= pb then
      shft := pb - stage + 1;
      d := temp_to(shft);
      temp_to(shft) := temp_to(0);
      temp_to(0) := d;
    end if;
    MID_TO_Sel <= temp_to;
    MID_FROM_sel <= temp;
  else
    if size < glb_info.size_two - pb then
      L0F_sel <= temp;
    end if;
  end if;
end if;
end process;

MID_FromMuxs : NMUXN
GENERIC MAP (  
    size_mux => ab
)
PORT MAP (  
    mux_in  => iter,  
    mux_sel => MID_FROM_sel,  
    mux_out => WHOLE_MID_FROM
);

MID_ToMuxs : NMUXN
GENERIC MAP (  
    size_mux => ab
)
PORT MAP (  
    mux_in  => iter,  
    mux_sel => MID_TO_sel,  
    mux_out => WHOLE_MID_TO
);

LOF_Muxs : NMUXN
GENERIC MAP (  
    size_mux => ab
)
PORT MAP (  
    mux_in  => iter,  
    mux_sel => LOF_sel,  
    mux_out => WHOLE_LOF
);

END sim;

\section*{A.4 Cooley-Tukey Twiddle Generator}

--
-- VHDL Entity thesis.CT_TG.symbol
--
-- Created:
-- by - Mike Balog
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.math_complex.all;
LIBRARY thesis;
USE thesis.FFT pak.ALL;

ENTITY CT_TG IS
  PORT(
    clk : IN std_logic;
    f2 ie : IN std_logic;
    glb info : IN tranfm info;
    loading : IN std_logic;
    reset : IN std_logic;
    tw in : OUT complex
  );
END CT_TG;

ARCHITECTURE struct OF CT_TG IS

  -- Architecture declarations

  -- Internal signal declarations
  SIGNAL data_ff : std_logic;
  SIGNAL data_in : complex;
  SIGNAL w en : std_logic;

  -- Component Declarations
  COMPONENT d que
  GENERIC (
    q size : natural := 10
  );
  PORT ( 
    clk : IN std_logic ;
    reset : IN STD_LOGIC ;
    data_in : IN complex ;
    data ee : OUT std_logic ;
    w_en : IN std_logic ;
    r en : IN std_logic ;
    data_ff : OUT std_logic ;
    data_out : OUT complex
  );
END COMPONENT;

-- Optional embedded configurations
-- pragma translate_off
FOR ALL : d_que USE ENTITY thesis.d_que;
-- pragma translate_on

BEGIN
-- Architecture concurrent statements

-- This process will generate all the twiddle factors
-- starting at the Stage Max and down to stage 1. It
-- Only generates the twiddle constant at each interation
-- pair and stores them for the Computation unit to take.

process (clk, reset, loading)
  variable stg_max, stg_cur, dim_num, iter, stage : natural;
  variable ck : integer;
  variable lg_p, lg_q, r, ki, sum_k_1_ki : natural;
  variable mask : natural;
begin -- process
  if reset = '1' then
    null;
  elsif loading = '1' then
    stage := glb_info.stage_max-1;
    iter := 0;
  elsif rising_edge(clk) then
    w_en <= '0';
    if data_ff = '0' then
      if iter = (glb_info.size_nat/(glb_info.num_pe*2)) then
        iter := 0;
        stage := stage - 1;
      end if;
      if stage /= 0 then
        w_en <= '1';
        stg_max := glb_info.stage_max - 1;
        stg_cur := stg_max - stage+1;
        dim_num := glb_info.dv_len;
        for i in dim_num-1 downto 0 loop
          ck := 0;
          for j in i to dim_num-1 loop
            ck := ck + integer(glb_info.dv_nat(i));
          end loop; -- j
        end loop;
      end if;
    end if;
  end if;
end process;
ki := i;
exit;
end if;
end loop; -- i
-- lg_p is log2(p) or where 2*p = 2^(lg_p)
sum_k_1_ki := 0;
for i in 0 to ki loop
    sum_k_1_ki := sum_k_1_ki + glb_info.dv_nat(i);
end loop; -- i
lg_p := sum_k_1_ki - stage;
lg_q := stg_max - sum_k_1_ki;
molygon := conv_integer(shl(conv_unsigned((2**lg_p)-1,16),
                           conv_unsigned(1g_q,16)));
    r := conv_integer(unsigned(shr(
                      unsigned(conv_std_logic_vector(iter,16) and
                      conv_std_logic_vector(mask,16)),
                      conv_unsigned(lg_q,16)))
    data_in <= EXP(((0.0,-6.28318530718) /
    cmplx(real(2**(lg_p+1)))) * cmplx(real(r)));
    iter := iter + 1;
end if;
end if;
end if;
end process;

-- Instance port mappings.
TG_Q : d_que
GENERIC MAP (q_size => 10)
PORT MAP (clk => clk,
          reset => reset,
          data_in => data_in,
          data_out => OPEN,
          w_en => w_en,
          r_en => f2_ie,
          data_ff => data_ff,
          data_out => tw_in)
); END struct;
A.5 Radix-2 Computation Unit

--
-- VHDL Entity thesis.f2_unit.symbol
--
-- Created:
--     by - Mike Balog
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

ENTITY f2_unit IS
    PORT(
        clk : IN std_logic;
        oe : IN std_logic;
        reset : IN std_logic;
        tw_in : IN complex;
        x0 : IN complex;
        x1 : IN complex;
        x_01_ready : IN std_logic;
        ie : OUT std_logic;
        y : OUT complex;
        y_ee : OUT std_logic
    );
END f2_unit;

ARCHITECTURE struct OF f2_unit IS

    -- Architecture declarations
    signal ie_q : std_logic;
    signal y0,y1: complex;

    -- Internal signal declarations
    SIGNAL data_ff : std_logic;
    SIGNAL data_in : complex;
    SIGNAL w_en : std_logic;

    -- Component Declarations
    COMPONENT d_que
    GENERIC (
q_size : natural := 10
);
PORT (  
clk   : IN std_logic;
reset : IN STD_LOGIC;
data_in : IN complex;
data_ee : OUT std_logic;
w_en  : IN std_logic;
r_en  : IN std_logic;
data_ff : OUT std_logic;
data_out : OUT complex
);
END COMPONENT;

-- Optional embedded configurations
-- pragma translate_off
FOR ALL : d_que USE ENTITY thesis.d_que;
-- pragma translate_on

BEGIN
-- Architecture concurrent statements

-- The Radix-2 Computation, and controller for
-- Taking data from the X and Twiddle Queues and
-- stores the data in an internal Queue.

process(clk,reset)
variable toggle : boolean;
begin
if reset = '1' then
  Y0 <= MATH_CZERO;
  Y1 <= MATH_CZERO;
  toggle := true;
  w_en <= '0';
  ie_q <= '0';
elsif rising_edge(clk) then
  ie_q <= '0';
  if data_ff = '0' and ie_q = '0' then
    if x_01_ready = '1' then
      ie_q <= '1';
      Y0 <= X0 + TW_IN * X1;
      Y1 <= X0 - TW_IN * X1;
    end if;
  end if;
  if ie_q = '1' and toggle then

toggle := false;
w_en <= '1';
data_in <= Y0;
elsif not toggle then
toggle := true;
w_en <= '1';
data_in <= Y1;
else
w_en <= '0';
data_in <= MATH_CZERO ;
end if;
end if;
end process;
ie <= ie_q;

-- Instance port mappings.
I0 : d_que
  GENERIC MAP (  
    q_size => 10    
  )
  PORT MAP (  
    clk   => clk,   
    reset => reset,  
    data_in => data_in,  
    data_ee => y_ee,  
    w_en   => w_en,  
    r_en   => oe,  
    data_ff => data_ff,  
    data_out => y  
  );
END struct;

A.6  Data Control Unit - Input State Machine

--
-- VHDL Entity thesis.X_sm.interface
--
-- Created:
--    by - Mike Balog
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_real.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ENTITY X_sm IS
  GENERIC(
    pid : natural := 0;
    q_size : natural := 5
  );
  PORT(
    clk : IN std_logic;
    glb_info : IN tranfm_info;
    icn_XQ_ee : IN std_logic;
    icn_XQ_ff : IN std_logic;
    icn_YQ_ee : IN std_logic;
    reset : IN std_logic;
    x_ff : IN std_logic;
    x_mid_f : IN natural;
    x_mid_from_icn : IN natural;
    x_mid_t : IN natural;
    aq_w_en : OUT std_logic;
    icn_x_r_en : OUT std_logic;
    icn_x_w_en : OUT std_logic;
    icn_yq_xsm_r_en : OUT std_logic;
    iter : OUT natural;
    load_stg : OUT std_logic;
    mem_r_en : OUT std_logic;
    mem_x_w_en : OUT std_logic;
    stage : OUT natural;
    x0_w_en : OUT std_logic;
    xl_w_en : OUT std_logic;
    x_done : OUT std_logic;
    x_rl : OUT std_logic
  );
END X_sm;

--

-- VHDL Architecture thesis.X_sm.fsm
--
-- Created:
ARCHITECTURE fsm OF X_sm IS

-- Architecture Declarations
signal toggle : std_logic;
signal FNUM,INUM,p_left,x_mask : natural;

TYPE STATE_TYPE IS (compair,remote,local,incr_addr,Loading,Stage_Inc,done,IcnData,store,start,IcnLeft);

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

-- Declare current and next state signals
SIGNAL current_state : STATE_TYPE;
SIGNAL next_state : STATE_TYPE;

-- Declare any pre-registered internal signals
SIGNAL aq_w_en_int : std_logic;
SIGNAL icn_x_r_en_int : std_logic;
SIGNAL icn_x_w_en_int : std_logic;
SIGNAL icn_yq_xsm_r_en_int : std_logic;
BEGIN

clk, reset, glb_info

BEGIN
IF (reset = '1') THEN
  current_state <= start;
  -- Reset Values
  aq_w_en <= '0';
  icn_x_r_en <= '0';
  icn_x_w_en <= '0';
  icn_yq_xsm_r_en <= '0';
  iter_cld <= 0;
  load_stg <= '1';
  mem_r_en <= '0';
  mem_x_w_en <= '0';
  stage_cld <= glb_info.stage_max;
  x0_w_en <= '0';
  x1_w_en <= '0';
  x_done <= '0';
  x_rl <= '0';
  p_left <= 0;
  toggle <= '0';
  x_mask <= 0;
ELSIF (clk'EVENT AND clk = '1') THEN
  current_state <= next_state;
  -- Registered output assignments
  aq_w_en <= aq_w_en_int;
  icn_x_r_en <= icn_x_r_en_int;
  icn_x_w_en <= icn_x_w_en_int;
icn_yq_xsm_r_en <= icn_yq_xsm_r_en_int;
load_stg <= load_stg_int;
mem_r_en <= mem_r_en_int;
mem_x_w_en <= mem_x_w_en_int;
x0_w_en <= x0_w_en_int;
x1_w_en <= x1_w_en_int;
x_done <= x_done_int;
x_rl <= x_rl_int;

-- Default Assignment To Internals

-- Combined Actions for internal signals only
CASE current_state IS
WHEN remote =>
    iter_cld <= iter_cld + 1;
p_left <= p_left + 1;
toggle <= not toggle;
WHEN local =>
    iter_cld <= iter_cld + 1;
toggle <= not toggle;
WHEN incr_addr =>
    if iter_cld = INUM then
        x_mask <= x_mid_f;
    end if;
WHEN Stage_Inc =>
    iter_cld <= INUM;
    stage_cld <= stage_cld-1;
toggle<= '0';
WHEN IcnData =>
    --p_out<=unsigned( p_out)+'1';
p_left <= p_left - 1;
WHEN store =>
    iter_cld <= iter_cld + 1;
WHEN start =>
    IF (reset = '0') THEN
        iter_cld <= pid*(glb_info.size_nat/glb_info.num_pe);
    END IF;
WHEN OTHERS =>
    NULL;
END CASE;
END IF;

END PROCESS clocked;
nextstate : PROCESS (  
  FNUM,  
  current_state,  
  glb_info,  
  icn_XQ_e,  
  icn_XQ_f,  
  icn_YQ_e,  
  iter_cld,  
  p_left,  
  reset,  
  stage_cld,  
  toggle,  
  x_f,  
  x_mask,  
  x_mid_from_icn,  
  x_mid_t  
)

BEGIN  
-- Default Assignment  
aq_w_en_int <= '0';  
icn_x_r_en_int <= '0';  
icn_x_w_en_int <= '0';  
icn_yq_xsm_r_en_int <= '0';  
load_stg_int <= '0';  
mem_r_en_int <= '0';  
mem_x_w_en_int <= '0';  
x0_w_en_int <= '0';  
x1_w_en_int <= '0';  
x_done_int <= '0';  
x_rl_int <= '0';  
-- Default Assignment To Internals  

-- Combined Actions  
CASE current_state IS  
WHEN compair =>  
  mem_r_en_int<='1';  
aq_w_en_int <= '1';  
IF (x_mid_t =PID) THEN  
  next_state <= local;  
ELSIF (x_mid_t /=PID) THEN  
  icn_x_w_en_int <= '1';  
  next_state <= remote;  
ELSE
next_state <= compair;
END IF;
WHEN remote =>
    mem_r_en_int <= '0';
    icn_x_w_en_int <= '0';
    aq_w_en_int <= '0';
    next_state <= incr_addr;
WHEN local =>
    mem_r_en_int <= '0';
    aq_w_en_int <= '0';
    if toggle = '0' then
        x0_w_en_int <= '1';
    else
        x1_w_en_int <= '1';
    end if;
    next_state <= incr_addr;
WHEN incr_addr =>
    icn_x_w_en_int <= '0';
    icn_x_r_en_int <= '0';
    x0_w_en_int <= '0';
    x1_w_en_int <= '0';
    x_rl_int <= '0';
    IF (iter_cld = FNUM) THEN
        next_state <= Stage_Inc;
    ELSIF (icn_XQ_ee = '0' and p_left > 0) THEN
        if x_mask = x_mid_from_icn then
            x0_w_en_int <= '1';
        else
            x1_w_en_int <= '1';
        end if;
        x_rl_int <= '1';
        icn_x_r_en_int <= '1';
        next_state <= IcnData;
    ELSIF (x_ff = '0' AND icn_XQ_ff = '0') THEN
        next_state <= compair;
    ELSE
        next_state <= incr_addr;
    END IF;
WHEN Loading =>
    load_stg_int <= '1';
    IF (icn_YQ_ee = '0') THEN
        icn_yq_xsm_r_en_int <= '1';
        mem_x_w_en_int <= '1';
        next_state <= store;
    ELSIF (iter_cld = FNUM) THEN
load_stg_int <= '1';
next_state <= Stage_INC;
ELSE
next_state <= Loading;
END IF;
WHEN Stage_INC =>
mem_x_w_en_int <= '0';
IF (stage_cld = 1) THEN
next_state <= IcnLeft;
ELSE
next_state <= incr_addr;
END IF;
WHEN done =>
x_DONE_int<='1';
next_state <= done;
WHEN IcnData =>
x_rl_int <= '1';
IF (stage_cld = 0) THEN
next_state <= IcnLeft;
ELSE
next_state <= incr_addr;
END IF;
WHEN store =>
mem_x_w_en_int <= '0';
icn_yq_xsm_r_en_int <= '0';
load_stg_int <= '1';
next_state <= Loading;
WHEN start =>
load_stg_int <= '1';
IF (reset = '0') THEN
INUM <= pid*(glb_info.size_nat/glb_info.num_pe);
FNUM <= pid*(glb_info.size_nat/glb_info.num_pe)+
(glb_info.size_nat/glb_info.num_pe);
next_state <= Loading;
ELSE
next_state <= start;
END IF;
WHEN IcnLeft =>
icn_x_r_en_int <= '0';
x_rl_int <= '0';
IF (icn_XQ_e = '0' and p_left > 0) THEN
if x_mask = x_mid_from_icn then
x0_w_en_int <= '1';
else
x1_w_en_int <= '1';
end if;
end if;
x_r1_int <= '1';
icn_x_r_en_int <= '1';
next_state <= IcnData;
ELSIF (p_left = 0) THEN
    next_state <= done;
ELSE
    next_state <= IcnLeft;
END IF;
WHEN OTHERS =>
    next_state <= start;
END CASE;

END PROCESS nextstate;

-- Concurrent Statements
-- Clocked output assignments
iter <= iter_cld;
stage <= stage_cld;
END fsm;

A.7 Data Control Unit - Output State Machine

--
-- VHDL Entity thesis.Y_sm.interface
--
-- Created:
-- by - Mike Balog
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_real.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ENTITY Y_sm IS
    GENERIC(
        pid : natural := 0
    )
PORT(
    aq_e0 : IN  std_logic;
    aq_e0_l : IN  std_logic;
    aq_e0_r : IN  std_logic;
    clk : IN  std_logic;
    f2_y_e0 : IN  std_logic;
    icn_YQ_e0 : IN  std_logic;
    icn_YQ_ff : IN  std_logic;
    load_stg : IN  std_logic;
    reset : IN  std_logic;
    x_done : IN  std_logic;
    y_mid : IN  natural;
    aq_mode : OUT  a_qe_mode;
    aq_r_en : OUT  std_logic;
    f2_oe : OUT  std_logic;
    icn_y_w_en : OUT  std_logic;
    icn_yq_ym_r_en : OUT  std_logic;
    mem_y_w_en : OUT  std_logic;
    y_done : OUT  std_logic;
    y_rl : OUT  std_logic
);
END Y_sm ;

--
-- VHDL Architecture thesis.Y_sm.fsm
--
-- Created:
--    by - Mike Balog
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ARCHITECTURE fsm OF Y_sm IS

    -- Architecture Declarations
    TYPE STATE_TYPE IS (  
        WaitForF2,  
        CheckMID,


WaitForICN,
ydone

-- State vector declaration
ATTRIBUTE state_vector : string;
ATTRIBUTE state_vector OF fsm : ARCHITECTURE IS "current_state";

-- Declare current and next state signals
SIGNAL current_state : STATE_TYPE;
SIGNAL next_state : STATE_TYPE;

-- Declare any pre-registered internal signals
SIGNAL aq_mode_int : a_que_mode;
SIGNAL aq_r_en_int : std_logic;
SIGNAL f2_oe_int : std_logic;
SIGNAL icn_y_w_en_int : std_logic;
SIGNAL icn_yq_ysm_r_en_int : std_logic;
SIGNAL mem_y_w_en_int : std_logic;
SIGNAL y_done_int : std_logic;
SIGNAL y rl_int : std_logic;

BEGIN

-- Process

 clocked : PROCESS(
   clk,
   load_stg
 )

BEGIN

IF (load_stg = '1') THEN
  current_state <= WaitForF2;
  -- Reset Values
  aq_mode <= local;
  aq_r_en <= '0';
  f2_oe <= '0';
  icn_y_w_en <= '0';
  icn_yq_ysm_r_en <= '0';
  mem_y_w_en <= '0';
  y_done <= '0';
  y rl <= '1';
ELSIF (clk'EVENT AND clk = '1') THEN
  current_state <= next_state;
  -- Registered output assignments
aq_mode <= aq_mode_int;
aq_r_en <= aq_r_en_int;
f2_oe <= f2_oe_int;
  icn_y_w_en <= icn_y_w_en_int;
  icn_yq_ysm_r_en <= icn_yq_ysm_r_en_int;
mem_y_w_en <= mem_y_w_en_int;
y_done <= y_done_int;
y_r1 <= y_r1_int;

  -- Default Assignment To Internals

END IF;

END PROCESS clocked;

-----------------------------------------------------
nextstate : PROCESS (  
aq_e,  
aq_e_r,  
aq_mode_int,  
  current_state, 
f2_y_e,  
icn_YQ_e,  
icn_YQ_ff,  
x_done,  
y_mid
)
-----------------------------------------------------

BEGIN  
  -- Default Assignment  
aq_mode_int <= local;
aq_r_en_int <= '0';
f2_oe_int <= '0';
icn_y_w_en_int <= '0';
icn_yq_ysm_r_en_int <= '0';
mem_y_w_en_int <= '0';
y_done_int <= '0';
y_r1_int <= '0';  
  -- Default Assignment To Internals

  -- Combined Actions  
CASE current_state IS  
WHEN WaitForF2 =>  
y_r1_int <= '0';  
IF (f2_y_e = '0' AND icn_YQ_ff = '0') THEN
if y_mid = PID then
    mem_y_w_en_int <= '1';
else
    icn_y_w_en_int <= '1';
aq_mode_int <= store;
end if;
f2_oe_int <= '1';
aq_r_en_int <= '1';
next_state <= CheckMID;
ELSIF (icn_YQ_e = '0' and aq_e_r = '0') THEN
    y_rl_int <= '1';
aq_mode_int <= remote;
    mem_y_w_en_int <= '1';
aq_r_en_int <= '1';
    next_state <= CheckMID;
ELSIF (aq_e_e = '1' and x_done = '1') THEN
    next_state <= ydone;
ELSE
    next_state <= WaitForF2;
END IF;
WHEN CheckMID =>
aq_mode_int <= local;
icn_y_w_en_int <= '0';
mem_y_w_en_int <= '0';
f2_oe_int <= '0';
aq_r_en_int <= '0';
IF (aq_mode_int=remote) THEN
    next_state <= WaitForF2;
ELSE
    next_state <= WaitForICN;
END IF;
WHEN WaitForICN =>
y_rl_int <= '0';
IF (icn_YQ_e = '0' and aq_e_r = '0') THEN
    y_rl_int <= '1';
aq_mode_int <= remote;
    mem_y_w_en_int <= '1';
aq_r_en_int <= '1';
    next_state <= CheckMID;
ELSIF (f2_y_e = '0' AND icn_YQ_ff = '0') THEN
    if y_mid = PID then
        mem_y_w_en_int <= '1';
    else
        icn_y_w_en_int <= '1';
aq_mode_int <= store;
end if;
f2_oe_int <= '1';
aq_r_en_int <= '1';
next_state <= CheckMID;
ELSIF (aq_ee = '1' and x_done = '1') THEN
  next_state <= ydone;
ELSE
  next_state <= WaitForICN;
END IF;
WHEN ydone =>
  y_done_int<= '1';
  next_state <= ydone;
WHEN OTHERS =>
  next_state <= WaitForF2;
END CASE;

END PROCESS nextstate;
END fsm;

A.8 Inter-Connection Network Interface

--
-- VHDL Entity thesis.ICN_IF.interface
--
-- Created:
--     by - Mike Balog
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ENTITY ICN_IF IS
  PORT(
    clk       : IN     std_logic;
    icn_ctl_in : IN    DataType;
    icn_data_in : IN     complex;
  )
icn_mid_in    : IN    natural;
icn_out_ack   : IN    std_logic;
icn_x_r_en    : IN    std_logic;
icn_x_w_en    : IN    std_logic;
icn_y_r_en    : IN    std_logic;
icn_y_w_en    : IN    std_logic;
memo_data_out : IN    complex;
reset         : IN    std_logic;
x_mid_t       : IN    natural;
y_data_from_f2: IN    complex;
y_mid         : IN    natural;
X_data_from_icn: OUT   complex;
Y_data_from_icn: OUT   complex;
icn_XQ_e      : OUT   std_logic;
icn_XQ_f      : OUT   std_logic;
icn_YQ_e      : OUT   std_logic;
icn_YQ_f      : OUT   std_logic;
icn_ctl_out   : OUT   Data_Type;
icn_data_out  : OUT   complex;
icn_in_ack    : OUT   std_logic;
icn_mid_out   : OUT   natural;
x_mid_from_icn: OUT   natural
);
END ICN_IF;

--
-- VHDL Architecture thesis.ICN_IF.struct
--
-- Created:
--       by - Mike Balog
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

LIBRARY thesis;
USE thesis.FFT_pak.ALL;

LIBRARY thesis;

ARCHITECTURE struct OF ICN_IF IS
-- Component Declarations

COMPONENT ICNIF_IN_SQ

GENERIC (  
    q_size : natural := 10;
);

PORT (  
    clk : IN std_logic ;  
    icn_ctl_in : IN DataType ;  
    icn_data_in : IN complex ;  
    icn_mid_in : IN natural ;  
    icn_x_r_en : IN std_logic ;  
    icn_y_r_en : IN std_logic ;  
    reset : IN std_logic ;  
    x_data_from_icn : OUT complex ;  
    x_mid_from_icn : OUT natural ;  
    y_data_from_icn : OUT complex ;  
    icn_xq_ee : OUT std_logic ;  
    icn_yq_ee : OUT std_logic ;  
    icn_in_ack : OUT std_logic
);

END COMPONENT;

COMPONENT ICNIF_OUT_SQ

GENERIC (  
    q_size : natural := 10;  
    yf : natural := 2;  
    xf : natural := 5
);

PORT (  
    y_mid : IN natural ;  
    clk : IN std_logic ;  
    icn_out_ack : IN std_logic ;  
    icn_x_w_en : IN std_logic ;  
    icn_y_w_en : IN std_logic ;  
    mem_data_out : IN complex ;  
    reset : IN std_logic ;  
    x_mid : IN natural ;  
    y_data_from_f2 : IN complex ;  
    icn_xq_ff : OUT std_logic ;  
    icn_yq_ff : OUT std_logic ;  
    icn_ctl_out : OUT DataType ;  
    icn_data_out : OUT complex ;  
    icn_mid_out : OUT natural
);

END COMPONENT;
-- Optional embedded configurations
-- pragma translate_off
FOR ALL : ICNIF_IN_SQ USE ENTITY thesis.ICNIF_IN_SQ;
FOR ALL : ICNIF_OUT_SQ USE ENTITY thesis.ICNIF_OUT_SQ;
-- pragma translate_on

BEGIN
-- Instance port mappings.
IN_Q : ICNIF_IN_SQ
GENERIC MAP (q_size => 10)
PORT MAP (
clk => clk,
icn_ctl_in => icn_ctl_in,
icn_data_in => icn_data_in,
icn_mid_in => icn_mid_in,
icn_xr_en => icn_xr_en,
icn yr_en => icn_yr_en,
reset => reset,
x_data_from_icn => X_data_from_icn,
x_mid_from_icn => x_mid_from_icn,
y_data_from_icn => Y_data_from_icn,
icn_xq_e => icn_XQ_e,
icn_yq_e => icn_YQ_e,
icn_in_ack => icn_in_ack
);
OUT_Q : ICNIF_OUT_SQ
GENERIC MAP (q_size => 10,
yf => 2,
xf => 5)
PORT MAP (
y_mid => y_mid,
clk => clk,
icn_out_ack => icn_out_ack,
icn_xw_en => icn_xw_en,
icn_yw_en => icn_yw_en,
mem_data_out => mem_data_out,
reset => reset,
x_mid => x_mid_t,
y_data_from_f2 => y_data_from_f2,
icn_xq_ff => icn_XQ_ff,
icn_yq_ff => icn_YQ_ff,
icn_ctl_out => icn_ctl_out,
icn_data_out => icn_data_out,
icn_mid_out => icn_mid_out
);
END struct;

A.9 ICN Input Queue

--
-- VHDL Architecture thesis.ICNIF_IN_SQ.untitled
--
-- Created:
--       by - Mike Balog
--       at - 09:59:06 05/07/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ENTITY ICNIF_IN_SQ IS
   GENERIC(
      q_size : natural := 10
   );
PORT(
   clk                : IN    std_logic;
   icn_ctl_in        : IN    DataType;
   icn_data_in       : IN    complex;
   icn_mid_in        : IN    natural;
   icn_x_r_en        : IN    std_logic;
   icn_y_r_en        : IN    std_logic;
   reset             : IN    std_logic;
   x_data_from_icn   : OUT   complex;
   x_mid_from_icn    : OUT   natural;
   y_data_from_icn   : OUT   complex;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
   icn_yq_ee         : OUT   std_logic;
   icn_xq_ee         : OUT   std_logic;
icn_in_ack : OUT std_logic

END ICNIF_IN_SQ;

ARCHITECTURE behav OF ICNIF_IN_SQ IS
BEGIN
process(clk,reset)
  type ICNIN_type is record
    data : complex;
    mid : natural;
  end record;
  type SQIN_array is array (natural range <>) of ICNIN_Type;
  variable XIN_Array, YIN_Array : SQIN_Array(0 to q_size-1);
  variable x_ap, y_ap, dy_left, dx_left: natural;
  variable icn_x_ap, icn_y_ap : natural := 0;
begin
  if reset = '1' then
    x_ap := 0;
    y_ap := 0;
    icn_x_ap := 0;
    icn_y_ap := 0;
    dx_left := 0;
    dy_left := 0;
  elsif rising_edge(clk) then
    -- control for data coming in for X
    if icn_ctl_in = T0_X then
      XIN_Array(icn_x_ap).data := icn_data_in;
      XIN_Array(icn_x_ap).mid :=icn_mid_in;
      icn_x_ap := icn_x_ap + 1;
      dx_left := dx_left + 1;
      if icn_x_ap = q_size then
        icn_x_ap := 0;
      end if;
    -- control for data coming in for Y
    elsif icn_ctl_in = T0_Y or icn_ctl_in = Loading then
      YIN_Array(icn_y_ap).data := icn_data_in;
      YIN_Array(icn_y_ap).mid := icn_mid_in;
      icn_y_ap := icn_y_ap + 1;
      dy_left := dy_left + 1;
      if icn_y_ap = q_size then
        icn_y_ap := 0;
      end if;
  end if;
  -- control for the data going out T0 X
if icn_x_r_en = '1' then
  x_ap := x_ap + 1;
  dx_left := dx_left - 1;
  if x_ap = q_size then
    x_ap := 0;
  end if;
end if;
end if;

-- control for the data going out TO Y
if icn_y_r_en = '1' then
  y_ap := y_ap + 1;
  dy_left := dy_left - 1;
  if y_ap = q_size then
    y_ap := 0;
  end if;
end if;
end if;

X_data_from_icn <= XIN_Array(x_ap).data;
X_mid_from_icn <= XIN_Array(x_ap).mid;
Y_data_from_icn <= YIN_Array(y_ap).data;

-- flags for inputs to check if queue is full
if dx_left = 0 then
  icn_XQ_ee <= '1';
else
  icn_XQ_ee <= '0';
end if;
if dy_left = 0 then
  icn_YQ_ee <= '1';
else
  icn_YQ_ee <= '0';
end if;
if dy_left >= q_size-1 and icn_ctl_in /= NO_OP then
  assert false report "icn_loading_q full" severity note;
end if;
if dy_left >= q_size-1 or dx_left >= q_size-2 then
  icn_in_ack <= '1';
else
  icn_in_ack <= '0';
end if;
end process;
END behav;
A.10 ICN Output Queue

--
-- VHDL Architecture thesis.ICNIF_OUT_SQ.behav
--
-- Created:
--    by - Mike Balog
--    at - 09:59:48 05/07/2002
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;
LIBRARY thesis;
USE thesis.FFT_pak.ALL;

ENTITY ICNIF_OUT_SQ IS
  GENERIC(
    q_size : natural := 10;
    yf     : natural := 2;
    xf     : natural := 5
  );
  PORT(
    y_mid     : IN   natural;
    clk       : IN   std_logic;
    icn_out_ack : IN   std_logic;
    icn_x_w_en : IN   std_logic;
    icn_y_w_en : IN   std_logic;
    mem_data_out : IN   complex;
    reset     : IN   std_logic;
    x_mid     : IN   natural;
    y_data_from_f2 : IN   complex;
    icn_xq_ff : OUT  std_logic;
    icn_yq_ff : OUT  std_logic;
    icn_ctl_out : OUT  DataType;
    icn_data_out : OUT  complex;
    icn_mid_out : OUT  natural
  );
END ICNIF_OUT_SQ;

ARCHITECTURE behav OF ICNIF_OUT_SQ IS
BEGIN
  process(clk,reset)
  BEGIN
    --
Type ICNIF_type is record
   data : complex;
   mid_where : natural;
   ctl : DataType;
end record;

Type SQ_array is array (natural range <>) of ICNIF_Type;

variable OUT_Array : SQ_Array(0 to q_size-1);
variable in_ap, icn_ap, d_left : natural := 0;

begin
   if reset = '1' then
      in_ap := 0;
      icn_ap := 0;
      d_left := 0;
      OUT_Array(0).ctl := NO_OP;
   elsif rising_edge(clk) then
      -- control for data coming in from X
      if icn_x_w_en = '1' then
         OUT_Array(in_ap).data := mem_data_out;
         OUT_Array(in_ap).mid_where := x_mid;
         OUT_Array(in_ap).ctl := TO_X;
         in_ap := in_ap + 1;
         d_left := d_left + 1;
         if in_ap = q_size then
            in_ap := 0;
         end if;
      end if;
      -- control for data coming in from Y
      if icn_y_w_en = '1' then
         OUT_Array(in_ap).data := y_data_from_f2;
         OUT_Array(in_ap).mid_where := y_mid;
         OUT_Array(in_ap).ctl := TO_Y;
         in_ap := in_ap + 1;
         d_left := d_left + 1;
         if in_ap = q_size then
            in_ap := 0;
         end if;
      end if;
      -- control for the data going out to ICN
      if icn_out_ack = '1' then
         OUT_Array(icn_ap).ctl := NO_OP;
         icn_ap := icn_ap + 1;
         d_left := d_left - 1;
         if icn_ap = q_size then
            icn_ap := 0;
         end if;
   end if;
A.11 Data Queue

```vhdl
--
-- VHDL Architecture thesis_queue.sim
--
-- Created:
-- by - Mike Balog
-- at - 11:26:23 04/10/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

ENTITY d_que IS
  GENERIC(
    q_size : natural := 10
  );
```
PORT(
  clk     : IN    std_logic;
reset   : IN    STD_LOGIC;
data_in : IN    complex;
data_ee  : OUT   std_logic;
w_en    : IN    std_logic;
r_en    : IN    std_logic;
data_ff  : OUT   std_logic;
data_out : OUT   complex
);
END d_que;

ARCHITECTURE sim OF d_que IS
BEGIN

  -- A Fifo design to take data and store into
  -- a local array and generates empty and full
  -- flags.
  process(clk)
    type q_array is array (natural range <>) of complex;
    variable data : q_array(0 to q_size - 1);
    variable addr_w, addr_r, d_left : natural := 0;
    begin
      if reset = '1' then
        for i in data'range loop
          data(i) <= (others => '0');
        end loop;
        addr_w := 0;
        addr_r := 0;
        d_left := 0;
      elsif rising_edge(clk) then
        if w_en = '1' then
          data(addr_w) := data_in;
          addr_w := addr_w + 1;
          d_left := d_left + 1;
        end if;
        if r_en = '1' then
          addr_r := addr_r + 1;
          d_left := d_left - 1;
        end if;
      end if;
      if addr_w = q_size then
        addr_w := 0;
      end if;
    end process;
END sim;
if addr_r = q_size then
  addr_r := 0;
end if;

if d_left = 0 then
  data_e := '1';
  data_f := '0';
elsif d_left >= q_size - 2 then
  data_e := '0';
  data_f := '1';
else
  data_e := '0';
  data_f := '0';
end if;
data_out <= data(addr_r);
end process;
END sim;

A.12 Address Queue

--
-- VHDL Architecture thesis.que.sim
--
-- Created:
--    by - Mike Balog
--    at - 11:26:23 04/10/2002
--
-- Generated by Mentor Graphics’ HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY iee;
USE iee.std_logic_1164.all;
USE iee.std_logic_arith.all;
library thesis;
use thesis.fft_pak.all;

ENTITY a_que IS
  GENERIC(
    q_size : natural := 10
  );
  PORT(
    clk : IN std_logic;
    reset : IN std_logic;
  )
lof_in : IN  natural;
mid_in : IN  natural;
aq_e : OUT std_logic;
w_en : IN  std_logic;
mode : IN  a_que_mode;
r_en : IN  std_logic;
aq_ff : OUT std_logic;
aq_e : OUT std_logic;
lof_out : OUT natural;
mid_out : OUT natural
);
END a_que;

ARCHITECTURE sim OF a_que IS
signal LOF_Q int,LOF_int,MID_Q int,MID_int : natural := 0;
BEGIN
  process(clk)
    type q_array is array (natural range <>) of natural;
    variable LOF_Q,MID_Q : q_array(0 to q_size - 1);
    variable addr_w, addr_r, d_left : natural := 0;
    variable LOF_Q,MID_Q : q_array(0 to q_size - 1);
    variable addr_w_RQ, addr_r_RQ, d_left_RQ : natural := 0;
    begin
      if reset = '1' then
        addr_w := 0;
        addr_r := 0;
        d_left := 0;
        addr_w_RQ := 0;
        addr_r_RQ := 0;
        d_left_RQ := 0;
      elsif rising_edge(clk) then
        if w_en = '1' then
          LOF_Q(addr_w) := LOF_in;
          MID_Q(addr_w) := MID_in;
          addr_w := addr_w + 1;
          d_left := d_left + 1;
        end if;

        if r_en = '1' then
          if mode = local then
            addr_r := addr_r + 1;
            d_left := d_left - 1;
          elsif mode = remote then
            addr_r := addr_r + 1;
            d_left := d_left - 1;
          end if;

      end process;
elsif mode = store then
    LOF_RQ(addr_w_RQ) := LOF_Q(addr_r);
    MID_RQ(addr_w_RQ) := MID_Q(addr_r);
    addr_w_RQ := addr_w_RQ + 1;
    addr_r := addr_r + 1;
    d_left := d_left - 1;
    d_left_RQ := d_left_RQ + 1;
end if;
end if;

if addr_r = q_size then
    addr_r := 0;
end if;

if addr_w = q_size then
    addr_w := 0;
end if;
if d_left_RQ = 0 then
    aq_ee_RQ <= '1';
else
    aq_ee_RQ <= '0';
end if;

if d_left = 0 then
    aq_e <= '1';
    aq_f <= '0';
elsif d_left = q_size-1 then
    aq_e <= '0';
    aq_f <= '1';
else
    aq_e <= '0';
    aq_f <= '0';
end if;
end if;

LOF_RQ_int <= LOF_RQ(addr_r);
MID_RQ_int <= MID_RQ(addr_r);
LOF_int <= LOF_Q(addr_r);
MID_int <= MID_Q(addr_r);
end process;

with mode select
    LOF_out <=
        LOF_RQ_int when remote,
        LOF_int when others;
with mode select
  MID_out <=
  MID_RQ_int when remote,
  MID_int when others;
END sim;

A.13 Memory Interface

--
-- VHDL Architecture thesis.memory_array.symbol
--
-- Created:
--        by - Mike Balog
--        at - 23:30:42 04/08/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.1 (Build 12)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
use ieee.math_complex.all;
library std;
use std.textio.all;

ENTITY memory_array IS
  GENERIC(
    pid: natural := 0;
    absize : natural := 6
  );
  PORT(
    addr_r : IN   natural;
    addr_w : IN   natural;
    clk    : IN   std_logic;
    data_in : IN   complex;
    w_en   : IN   std_logic;
    r_en   : IN   std_logic;
    data_out : OUT complex;
    pe_done : IN   std_logic
  );
END memory_array ;
ARCHITECTURE sim OF memory_array IS
  file outfile : text open write_mode is "outfile.txt";
BEGIN
  process(clk,pe_done)
    variable buf : line;
    type mem_array is array (natural range <>) of complex;
    variable data : mem_array(0 to 2**absize - 1);
    begin
      if rising_edge(clk) then
        if (w_en='1') then
          data(addr_w) := data_in;
        end if;
        if r_en = '1' then
          data_out <= data(addr_r);
        end if;
      end if;
      if rising_edge(pe_done) then
        for i in data'range loop
          write(buf,i);
          write(buf,':');
          write(buf,pid);
          write(buf,':');
          write(buf,data(i).re);
          write(buf,':');
          write(buf,data(i).im);
          writeln(outfile,buf);
        end loop; -- i
      end if;
    end process;
  END sim;

A.14 Inter Connection Network

--
-- VHDL Architecture thesis.ICN.behav
--
-- Created:
--     by - Mike Balog
--     at - 14:50:28 05/07/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.all;

LIBRARY thesis;
USE thesis.fft_pak.all;

ENTITY ICN IS
  PORT(
    clk       : IN    std_logic;
    icn_ctl_out : IN   icn_ctl_array (p DOWNTO 0);
    icn_data_out : IN   icn_data_array (p downto 0);
    icn_in_ack : IN   std_logic_vector(p downto 0);
    icn_mid_out : IN   icn_mid_array (p downto 0);
    reset     : IN    std_logic;
    icn_ctl_in : OUT  icn_ctl_array (p DOWNTO 0);
    icn_data_in : OUT  icn_data_array (p downto 0);
    icn_mid_in : OUT  icn_mid_array(p downto 0);
    icn_out_ack : OUT  std_logic_vector(p downto 0)
  );
END ICN;

ARCHITECTURE behav OF ICN IS
BEGIN
  process (clk, reset)
  variable icn_out_ack_Q : std_logic_vector(p downto 0);
  variable busy : std_logic_vector(p downto 0);
  variable check : boolean;
  begin -- process
    if reset = '1' then
      for i in p downto 0 loop
        icn_out_ack_q(i) := '0';
        busy(i) := '0';
      end loop;
    elsif rising_edge(clk) then
      for i in p-1 downto 0 loop
        icn_ctl_in(i) <= NO_OP;
        icn_data_in(i) <= (0.0,0.0);
        icn_mid_in(i) <= 0;
        busy(i) := '0';
      end loop;
      if icn_ctl_out(p) = Loading then
        icn_data_in(icn_mid_out(p)) <= icn_data_out(p);
        icn_ctl_in (icn_mid_out(p)) <= icn_ctl_out(p);
      end if;
    end if;
  end process;
END behav;
icn_mid_in (icn_mid_out(p)) <= p;
else
  for i in p-1 downto 0 loop
    check := icn_ctl_out(i) = T0_X or icn_ctl_out(i) = T0_Y;
    if check and icn_out_ack_Q(i) = '0' then
      if busy(icn_mid_out(i)) = '0' then
        busy(icn_mid_out(i)) := '1';
        icn_data_in(icn_mid_out(i)) <= icn_data_out(i);
        icn_ctl_in(icn_mid_out(i)) <= icnCtl_out(i);
        icn_mid_in(icn_mid_out(i)) <= i;
        icn_out_ack_q(i) := '1';
      else
        icn_out_ack_q(i) := '0';
      end if;
    else
      icn_out_ack_q(i) := '0';
    end if;
  end loop; -- i
end if;
end if;
for i in p downto 0 loop
  icn_out_ack(i) <= icn_out_ack_q(i);
end loop;
end process;
END behav;

A.15 FFT Machine Test Bench

--
-- VHDL Architecture thesis_TB_PE.testbench
--
-- Created:
--          by - Mike Balog
--          at - 10:56:52 05/08/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
library thesis;
use thesis.fft_pak.all;

ENTITY TB_PES IS

END TB_PES ;

--
-- VHDL Architecture thesis.TB_PE.testbench
--
-- Created: 
--      by - Mike Balog
--      at - 10:56:19 05/08/2002
--
-- Generated by Mentor Graphics' HDL Designer(TM) 2001.5 (Build 170)
--
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.std_logic_arith.all;
USE ieee.math_complex.ALL;
LIBRARY thesis;
USE thesis.fft_pak.ALL;
LIBRARY thesis;

ARCHITECTURE testbench OF TB_PES IS
  -- Internal signal declarations
  SIGNAL clk : std_logic := '1';
  SIGNAL glb_info : tranfm_info;
  SIGNAL icn_ctl_in : icn_ctl_array (p DOWNTO 0);
  SIGNAL icn_ctl_out : icn_ctl_array (p DOWNTO 0);
  SIGNAL icn_data_in : icn_data_array (p downto 0);
  SIGNAL icn_data_out : icn_data_array (p downto 0);
  SIGNAL icn_in_ack : std_logic_vector(p downto 0);
  SIGNAL icn_mid_in : icn_mid_array (p downto 0);
  SIGNAL icn_mid_out : icn_mid_array (p downto 0);
  SIGNAL icn_out_ack : std_logic_vector(p downto 0);
  signal iu_pe_done : std_logic_vector(p-1 downto 0);
  SIGNAL reset : std_logic:= '1'
  signal muxin : natural;
  signal muxsel : muxsel_array (ab-1 DOWNTO 0);
  signal muxout : natural;

  -- Component Declarations
  COMPONENT PE
    GENERIC (
PID : natural := 0
);
PORT(
  clk : IN    std_logic;
  glb_info : IN    tranfm_info;
  icn_ctl_in : IN    DataType;
  icn_data_in : IN    complex;
  icn_mid_in : IN    natural;
  icn_out_ack : IN    std_logic;
  reset : IN    std_logic;
  icn_ctl_out : OUT   DataType;
  icn_data_out : OUT   complex;
  icn_in_ack : OUT    std_logic;
  icn_mid_out : OUT   natural;
  iu_pe_done : OUT    std_logic
);
END COMPONENT;
COMPONENT ICN
PORT(
  clk : IN    std_logic;
  icn_ctl_out : IN    icn_ctl_array (p DOWNTO 0);
  icn_data_out : IN    icn_data_array (p downto 0);
  icn_in_ack : IN    std_logic_vector(p downto 0);
  icn_mid_out : IN    icn_mid_array (p downto 0);
  reset : IN    std_logic;
  icn_ctl_in : OUT   icn_ctl_array (p DOWNTO 0);
  icn_data_in : OUT   icn_data_array (p downto 0);
  icn_mid_in : OUT   icn_mid_array(p downto 0);
  icn_out_ack : OUT   std_logic_vector(p downto 0)
);
END COMPONENT;
COMPONENT NMUXN
GENERIC(
  size_mux : natural := 20
);
PORT(
  mux_in : IN    natural;
  mux_sel : IN    muxsel_array (size_mux-1 DOWNTO 0);
  mux_out : OUT    natural
);
END COMPONENT;

-- Optional embedded configurations
-- pragma translate_off
FOR ALL : PE USE ENTITY thesis.PE;
-- pragma translate_on
BEGIN
   clock: clk <= not clk after 5 ns;

   rset : reset <= '1','0' after 20 ns;

   test: process(reset,clk)
      type cpx_array is array (natural range <>) of complex;
      variable mx : natural := 64;  -- max input data
      variable bv : natural := 0;  -- basis vector 1 @ bv
      variable data : cpx_array(mx-1 downto 0);
      variable logtwo_tf : complex;
      variable tf_size : natural := 16;  -- transform size
      variable pe_num : natural := p;  -- num of pes
      variable dvlen : natural := 2;  -- length of dimension
      variable dv : dv_array(ab-1 downto 0) :=
         (0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,0,2,2);  -- Dimension vector
      variable ip : muxsel_array(ab-1 downto 0) :=
         (21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,21,2,3,0,1);
      variable index,ld_mid : natural := 0;  -- Index used in loading
      variable pe_done : std_logic;

      begin
         if reset = '1' then
            for i in mx-1 downto 0 loop
               data(i).re := real(0);
               data(i).im := real(0);
            end loop;  -- i
            data(bv).re := real(1);
            logtwo_tf := LOG2(CMPLX(real(tf_size)));
            glb_info.num_pe <= pe_num;
            glb_info.size_two <= conv_integer(integer(logtwo_tf.re));
            glb_info.size_nat <= tf_size;
            glb_info.size_bit <= (others => '0');
            glb_info.stage_max <= conv_integer(integer(logtwo_tf.re))
               +integer(1);
            glb_info.dv_len <= dvlen;
            glb_info.dv_bit <= (others => '0');
            glb_info.dv_nat <= dv;
            glb_info.init_perm <= ip;
            index := 0;
            ld_mid := 0;
            pe_done := '1';
            icn_in_ack(p) <= '0';
            icn_mid_out(p) <= 0;
         end if;
      end test;
elsif rising_edge(clk) then
    if index < tf_size then
        ld_mid := conv_integer(
            shr( conv_unsigned(MUXOUT,ab),
                conv_unsigned(integer(logtwo_tf.re)-pb,ab)));
        if icn_in_ack(ld_mid) = '0' then
            icn_ctl_out(p) <= Loading;
            icn_mid_out(p) <= ld_mid;
            icn_data_out(p) <= data(index);
            index := index + 1;
        else
            icn_data_out(p) <= (0.0,0.0);
            icn_mid_out(p) <=0;
            icn_ctl_out(p) <= NO_OP;
        end if;
    else
        icn_data_out(p) <= (0.0,0.0);
        icn_mid_out(p) <=0;
        icn_ctl_out(p) <= NO_OP;
    end if;

    pe_done := '1';
    for i in p-1 downto 0 loop
        pe_done := iu_pe_done(i) and pe_done;
    end loop; -- i
    if pe_done = '1' then
        assert false report "DONE!!!" severity FAILURE;
    end if;
end if;
end if;
MUXIN <= index;
MUXSEL <= ip;
end process;

-- Instance port mappings.
IPERM : NMUXN
 GENERIC MAP(
    size_mux => ab
)
 PORT MAP(
    mux_in => MUXIN,
    mux_sel => MUXSEL,
    mux_out => MUXOUT
);

ICNS : ICN
PORT MAP (  
  clk => clk,  
  icn_ctl_out => icn_ctl_out,  
  icn_data_out => icn_data_out,  
  icn_in_ack => icn_in_ack,  
  icn_mid_out => icn_mid_out,  
  reset => reset,  
  icn_ctl_in => icn_ctl_in,  
  icn_data_in => icn_data_in,  
  icn_mid_in => icn_mid_in,  
  icn_out_ack => icn_out_ack  
);

peloo: for i in 0 to p-1 generate  
PES : PE  
GENERIC MAP (  
  PID => i  
)  
PORT MAP (  
  clk => clk,  
  glb_info => glb_info,  
  icn_ctl_out => icn_ctl_out(i),  
  icn_data_out => icn_data_out(i),  
  icn_in_ack => icn_in_ack(i),  
  icn_mid_out => icn_mid_out(i),  
  reset => reset,  
  icn_ctl_in => icn_ctl_in(i),  
  icn_data_in => icn_data_in(i),  
  icn_mid_in => icn_mid_in(i),  
  icn_out_ack => icn_out_ack(i),  
  iu_pe_done => iu_pe_done(i)  
);

end generate peloo;
END testbench;