ADSL, VDSL, and Multicarrier Modulation. John A. C. Bingham

Print ISBN 0-471-29099-8 Electronic ISBN 0-471-20072-7

APPENDIX C

EFFICIENT HARDWARE IMPLEMENTATIONS OF FFT ENGINES

Mitra Nasserbakht

Intel Corporation, E-mail: Mitra.Nasserbakht@intel.com or Bite 5000@aol.com

C.1 OVERVIEW

Efficient algorithms for computing the discrete fourier transform (DFT) have enabled widespread access to Fourier analysis in numerous fields. These application areas span diverse disciplines such as applied mechanics and structural modeling to biomedical engineering. In the signal-processing arena, Fourier theory has been widely used for signal recognition, estimation, and spectral analysis. Fourier analysis has been at the core of many communication system subblocks, such as those used for echo cancellation, filtering, coding, and compression. The ability to compute DFT in realtime and with minimal hardware is the key to the successful implementation of many of these complex systems.

The fast fourier transform (FFT) is an efficient algorithm for computing the DFT of time-domain signals. The focus of this chapter is on the necessary ingredients for the design of FFT processing engines capable of handling data of a real-time nature found in most digital signal processing and telecommunications applications.

This appendix starts with a brief overview of the FFT and its computation. Top-level system requirements, addressing, arithmetic processing, memory subsystem, and data ordering are discussed in Section C.3. Section C.4 is devoted to a discussion of implementation issues for a representative FFT processing engine.

C.2 FAST FOURIER TRANSFORM

In 1807, Joseph Fourier described the Fourier series representation of signals where any periodic signal could be represented by the sum of scaled sine and

cosine waveforms. The signal can thus be represented by a sequence of an infinite number of scale factors or coefficients. Given such a sequence, we can also determine the original waveform. For Fourier series representation of finite sequences, we deal exclusively with the discrete Fourier transform (DFT).

At the core of the computation are the following two equations that describe the N-point DFT of a data sequence $x(k)$:

$$X(i) = \sum_{k=0}^{N-1} x(k) W^{ik} \tag{C.1}$$

and the inverse DFT of a transform sequence $X(i)$:

$$x(k) = \frac{1}{N} \sum_{i=0}^{N-1} X(i) W^{-ki} \tag{C.2}$$

where $\{x(k)\}$ is a vector of N real samples, $\{X(i)\}$ is a N-complex vector with Hermitian symmetry, and $W = \exp[-j(2\pi/N)]$ are called the twiddle factors. The number of operations for computing DFT of a signal using direct DFT method is proportional to N^2.

The invention of FFT is attributed to Cooley and Tukey in 1965. Fast Fourier transforms compute the DFT with greatly reduced number of operations. By adding certain sequences of data after performing multiplication by the same fixed complex multipliers, FFT eliminates redundancies in brute-force DFT computations. This efficiency in computation is achieved at the expense of additional reordering steps to determine the final results. These additional steps, once implemented efficiently, have negligible overhead on the overall compute engine. As a result, FFT is an extremely robust algorithm that lends itself well to machine computation. Since most applications of FFT deal with real-time data sequences, full hardware implementation of the algorithm is desired. Due to the complexity and large amount of hardware required for this type of implementation, many trade-offs need to be considered; these are outlined in the upcoming sections.

For large values of N, computational efficiency can be achieved by breaking the DFT into successively smaller calculations. This can be achieved in the time or frequency domain, as discussed in the following sections. The main observation made by Cooley and Tukey was that when N is not prime, the DFT computation can be decomposed into a number of DFTs of smaller length. This decomposition can be continued until the prime radix for the given value of N is reached. For the case where N is a power of 2, the total number of operations is reduced to on the order of $N \log_2 N$.

C.2.1 Radix-2 FFT Computation

In a radix-2 implementation, the basic building block is a two-point *butterfly* shown in Figure C.1. It has two inputs ($x[0]$, $x[1]$) and two outputs ($X[0]$, $X[1]$)

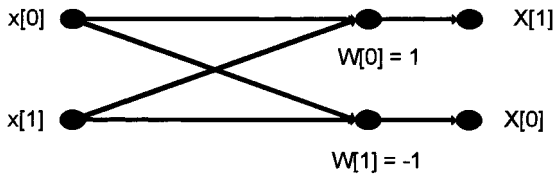


Figure C.1 Basic radix-2 butterfly.

and consists of one complex multiply and two complex adds:

$$\begin{aligned} X[1] &= x[0] + x[1] \\ X[0] &= x[0] + \mathrm{W}[1] * x[1] \end{aligned} \tag{C.3}$$

In computing the FFT for larger than two-point sequences, data points are successively partitioned according to methods outlined in Sections C.2.3 and C.2.4 until the basic butterfly, shown in Figure C.1, is reached. Figures C.3 and C.4 depict examples of such partitioning. Radix-2 implementations have minimal demand on memory subsystem sophistication but are not the most efficient for large data sequences.

C.2.2 Radix-4 FFT Computation

In a radix-4 implementation, the basic building block is a four-point butterfly, as depicted in Figure C.2. A basic implementation of this requires four additions and three complex multiplication operations:

$$\begin{aligned} X[0] &= x[0] + x[1] + x[2] + x[3] \\ X[1] &= (x[0] - \mathrm{j}\,x[1] - x[2] + \mathrm{j}\,x[3])\mathrm{W}^1 \\ X[2] &= (x[0] - x[1] + x[2] - x[3])\mathrm{W}^2 \\ X[3] &= (x[1] - \mathrm{j}\,x[1] - x[2] - \mathrm{j}\,x[3])\mathrm{W}^3 \end{aligned} \tag{C.4}$$

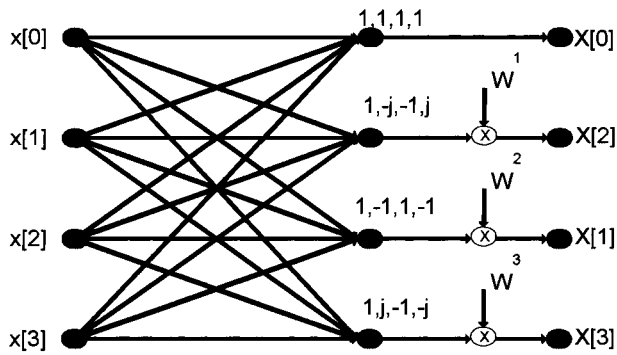


Figure C.2 Basic radix-4 butterfly.

Similar to radix-2 computation, larger values of N need to be divided into groups of 4 this time until the basic butterfly is reached.

C.2.3 Decimation in Time

As mentioned above, it is possible to break up the input sequence to be transformed into successively smaller time-domain sequences, hence the name *decimation in time* (DIT). The DIT can be interchangeably used with decimation in input for the case of a DFT-only engine. In a more general case, input can be the data sequence or its respective transform sequence. For the special case of N being a power of 2, the input sequence is divided into its even and odd parts, and each is so further divided until the base of the algorithm is reached (two points in a radix-2, four points in a radix-4 algorithm, etc.) This requires logr N stages of computation.[1] The total number of complex multiplies and adds will therefore be N logr N.

An example of FFT decomposition is depicted in Figure C.3 for $N = 8$ and a radix of 2. The DIT decomposition is sometimes referred to as the *Cooley–Tukey method* of computing the FFT.

C.2.4 Decimation in Frequency

When performing pure DFT functionality, the compute engine may be designed to divide the frequency domain (output in this case) into smaller sequences. The total number of computations remains the same as for DIT-type algorithms: N logr N, with the number of complex multiplies being (N/radix) logr N and the number of complex additions being N logr N.

In general, a great degree of correspondence exists between the DIT and decimation in frequency (DIF) algorithms whereby interchanging one's outputs and inputs and reversing the direction of the flow graph of computation would yield the other method. In addition, if the input data are entered in normal order, proper permutations must happen to produce bit-reversed ordered data before the final results are returned.

An example of FFT decomposition is depicted in Figure C.4 for $N = 8$ and a radix of 2. The DIF decomposition is sometimes referred to as the *Sande–Tukey method* of computing the FFT.

C.3 ARCHITECTURAL CONSIDERATIONS

In specifying the architecture and design of an FFT engine, there are several key factors to consider. First is the basic FFT computational building block. At a high level, this affects the radix chosen along with memory system availability and requirements. At a lower level, it determines the details of how the radix-x processor engine is to be implemented. These are discussed in Section C.1.1.

[1] For convenience, logr is written for $\log_{\text{radix}}$.

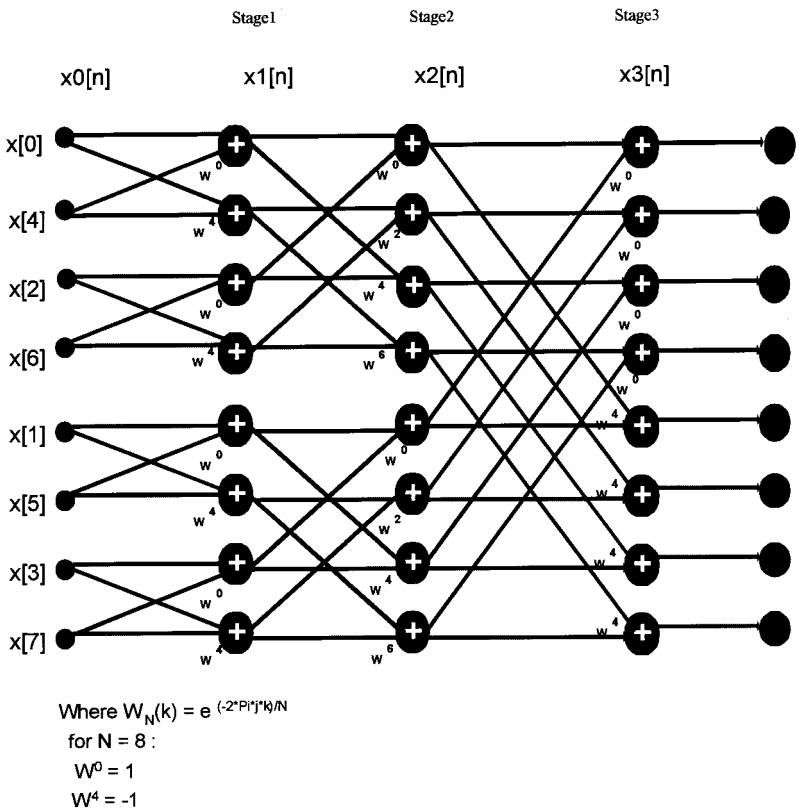


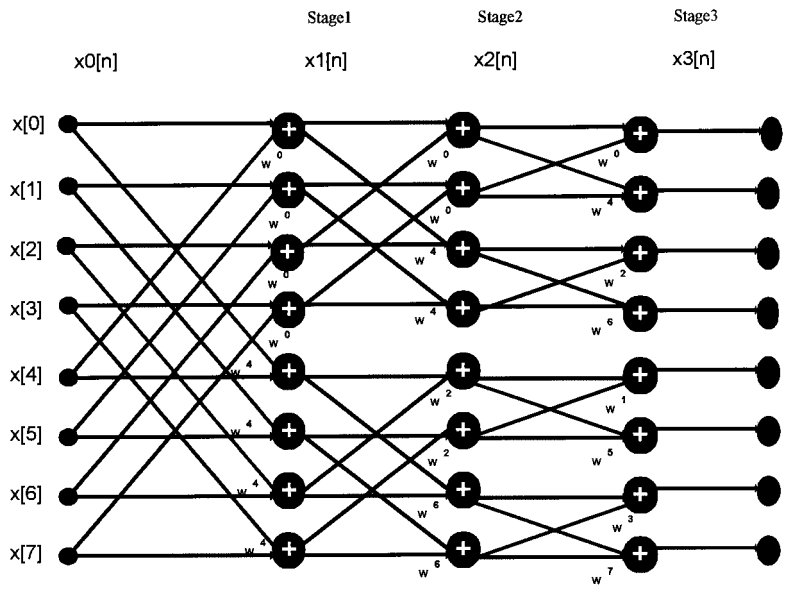
Figure C.3 Flow graph of FFT: decimation in time (DIT) with $N = 8$.

Another important consideration is memory access. This deals with the number of memory devices to be used, and most important, the addressing scheme used in the implementation of the FFT engine. For space conservation, in-place algorithms are almost universally preferable. In such cases, care must be taken in choosing an appropriate addressing scheme for both data and the corresponding twiddle factors used in different stages of FFT computation, as discussed in Section C.2.

Finally, there are details to consider pertaining to the order in which data and twiddle factors enter the computation engine as well as scrambling and descrambling of data. Some of these considerations are discussed in Sections C.1.1 and C.2.

C.3.1 Number Representation Scheme

One of the most important factors in deciding the architecture of a number-crunching machine is its number representation scheme. This decision directly affects the storage requirements of the machine. In applications such as FFT or



Where $W_N(k) = e^{(-2*Pi*j*k)/N}$
for N = 8 :
$W^0 = 1$
$W^4 = -1$

Figure C.4 Flow graph of FFT: decimation In frequency (DIT) with $N = 8$.

other signal processing applications, an increasingly larger amount of on-chip memory is required to support the real-time nature of these applications. Saving a few bits for each data point that needs to be stored in the RAMs will save a large percentage of chip area in such applications.

Twos-Complement Versus Sign-Magnitude. The number representation scheme dictates the type of computational elements used in the architecture. If any data representation system other than the standard twos-complement binary is chosen, one must make a trade-off between taking advantage of the number representation system in computational elements as well, or use standard twos-complement binary arithmetic elements and make appropriate conversions between data formats when necessary. More details of such a trade-off are covered in the Section C.4.

Floating-Point Versus Fixed-Point. The choice of number representation scheme is also driven by the need to achieve maximum "dynamic range" for the available hardware and to minimize error accumulation due to lack of precision. The first floating-point digital filter was built in 1973. Since then several general-purpose DSP machines have been developed to accommodate high-precision

calculations. Floating-point signal processing has several advantages over fixed-point arithmetic, the most notable of which is its superior dynamic range behavior. In fixed-point number representation schemes, the dynamic range grows only linearly (6 dB/bit) with increasing bit width, while for floating-point numbers it grows exponentially with increasing exponent bit width. The use of floating-point number representation improves the dynamic range, and it may be the only scheme that provides proper signal/noise ratio (SNR) in the presence of storage limitations.

The improvement in dynamic range comes at the expense of increased hardware complexity. In the case of an FFT engine, this hardware overhead is multiplied by the requirement of having to store initial and final results as well as intermediate values and keeping proper levels of precision.

Due to the inherent round-off noise in computation-intensive applications, the number representation scheme and the carrying of the round-off noise through the computation become essential factors affecting the final performance of the FFT engine. The processor needs to allocate enough internal storage between RAM stages to ensure that the SNR never falls below the minimum system requirements.

In general, the speed of operation for fixed-point multipliers and adders has increased due to technology scaling as well more efficient algorithms. Sub-1nS 64-bit adders have been reported in 0.5-μm technology. Floating-point arithmetic processors have started to enjoy similar performance improvements even without full custom implementations. As a result, system SNR requirements and available hardware are the main driving factors in the implementation of the arithmetic units for FFT engines rather than delay considerations.

C.3.2 Memory Subsystem

One of the more significant considerations in the design of a specialized high-performance DSP engine is its memory subsystem architecture. As the processing engine becomes capable of more speedy operations and as the sophistication of DSP algorithms increases, there is a greater need for fast, efficient memory-access algorithms independent of the advances in DRAM and SRAM technologies. The goal in most such architectures in general and FFT specifically is to satisfy the system bandwidth requirements while occupying the least amount of space. In addition, due to intensive pipelining and the emphasis on high throughput, decoupling of fetch hardware and execute engine is dictated. This allows a continuous stream of data flowing into the compute engine and similarly at the output for further processing and possibly transmission and receipt of data.

In an FFT-specific engine, the basic design of the memory subsystem is influenced by the chosen radix, the number and types of memory elements, and the type of storage. In-place storage is used almost universally to minimize area in applications where it is critical.

FFT Address Storage (FAST). In this section the "FAST" addressing scheme is introduced, which optimizes storage for an in-place FFT algorithm in any radix. Given the number of data points (N) and chosen radix (r), this scheme determines the optimum storage location for each data point in an r-bank memory system.

Addressing for FFT Modes. The N data points can be organized into r banks for a radix-r implementation with N/r data points residing in each data bank. Each data point is assigned a bank number (B), and an address (A) which determine its location within each bank. This assignment needs to be done such that the following conditions are met at all times:

1. Only the locations being read in the current FFT cycle are overwritten.
2. The destination locations are determined such that the data points required for all subsequent butterfly stages reside in different memory banks.

The second requirement poses more restrictions on the address generation hardware. The following algorithm computes the value of bank (B) and address (A) from each data point index (i);

1. Express the index in radix-r notation:

$$i = i(r-1)r^{(r-1)} + \cdots + i(3)r^3 + i(2)r^2 + i(1)r + i(0) \tag{C.5}$$

2. Compute the value of the bank (B) according to the following equation:

$$B = (i(r-1) + \ldots + i(3) + i(2) + i(1) + i(0)) \text{ modulo } r \tag{C.6}$$

3. Compute the address location (A) for each data point according to

$$A = i \text{ modulo}(N/r) \tag{C.7}$$

These equations yield the optimum address and bank assignments for any radix(r), provided that the system is capable of providing r-banks of memory per required storage device. This is sufficient to provide contention-free in-place storage assignments for all FFT stages; there would be logr N stages for a radix-r implementation of FFT. Each stage of the FFT algorithm requires fetching a different set of inputs that will be written back in-place once the computation on each data point is completed. This assignment remains unchanged from stage to stage while preserving the contention-free nature of the algorithm. Table C.1 shows the bank and address assignment for the first nine entries for a 256-point FFT implemented in radix-4, using the assignment scheme above.

Addressing for Pre-/Postprocessing. To reduce computational complexity, the same engine can be used to process FFT and IFFT data. Often, system

TABLE C.1 Memory Assignment for a 256-Point FFT in Radix-4

Data-Point Index i	Radix-4 Digits $i(3)\ i(2)\ i(1)\ i(0)$	Memory Bank/Address
0	0000	0/0
1	0001	1/1
2	0002	2/2
3	0003	3/3
4	0010	1/4
5	0011	2/5
6	0012	3/6
7	0013	0/7
8	0020	2/8

requirements dictate and greatly benefit from this reduced cost. An example of such a system would be a communication device that is receiving and transmitting information in time-multiplexed fashion. Since the inputs to the FFT and the outputs from the IFFT are "real," a significant reduction in computational resources can be achieved. Instead of zeroing out the imaginary part, we can pre/postprocess the data to enable it to use an engine that is half the size of what would be needed otherwise. Preprocessing is an additional stage before entering the IFFT mode of the engine in which data are prepared. Similarly, postprocessing is an additional stage after the FFT mode of the engine.

Preprocessing modifies the input vector so that when an IFFT is performed the real part of the output vector contains the even time samples and the imaginary part of the output contains the odd time samples. For an effective 512-point IFFT function, the preprocessing function is described by the following equation:

$$\mathrm{Y}[i] = \{[\mathrm{X}(i) + \mathrm{X}^*(256 - i)] + \mathrm{j}[\mathrm{X}(i) - \mathrm{X}^*(256 - i)]\mathrm{W}^{(256-i)}\}^* \qquad \text{(C.8)}$$

The input to the FFT engine is a 256-point complex vector that is formed from the real 512-point data vector. The real part of the vector is comprised of the even sample points while the odd samples make up the imaginary part of the vector. Postprocessing is done as the final stage according to the following equation:

$$\mathrm{Y}[i] = \left(\frac{1}{2}\right)[\mathrm{X}(i) + \mathrm{X}^*(256 - i)] - (\mathrm{j}/2)[\mathrm{X}(i) - \mathrm{X}^*(256 - i)]\mathrm{W}^i \qquad \text{(C.9)}$$

These two modes have special requirements in terms of storage and addressing modes.

As can be observed from (C.8) and (C.9) for pre/postprocessing, regardless of the radix used in the computational engine, only two data points are operated on at any instant. This is in line with the radix-2 FFT as only two inputs are fetched and operated on at any given time. In order to reduce the number of cycles

TABLE C.2 Data Ordering/Digit Reversal in Radix-2 and Radix-4 FFT

Data Point	Radix-2 Input	Radix-2 Output	Radix-4 Input	Radix-4 Output
0	0000	0000	00	00
1	0001	1000	01	10
2	0010	0100	02	20
3	0011	1100	03	30
4	0100	0010	10	01
5	0101	1010	11	11
6	0110	0110	12	21
7	0111	1110	13	31

required to perform these two operations and to take full advantage of existing hardware in cases of non-radix-2 FFT architecture, the addressing scheme is modified to pull and operate on "*r*" data-points. For a multiple of 2 such as a radix-4 case, this translates to keeping a dual set of symmetric addresses from top and bottom, and fetching four points at a time. The table is skipped due to simplicity of derivation.

C.3.3 Scrambling and Unscrambling of Data

Due to the inherent nature of the FFT algorithm, and depending on whether DIT or DIF is chosen, input data points or outputs will be in bit reversed order in radix-2 computation. As a result, coefficient entry into the FFT engine needs to be adjusted accordingly. Bit reversal in a radix-2 case can be generalized to "digit reversal" in any radix-*r* architectures. Many attempts have been made to study the feasibility of implementing this type of address generation in software, but due to the long delays of software implementations, the consensus has been to add the hardware to the system. Hardware implementation can be greatly simplified if the reordering of data is properly designed into the memory access blocks. An example of the duality between radix-2 and radix-4 data input and output orderings is given in Table C.2.

C.3.4 Twiddle Factor Generation

One of the hardware implementation considerations for an FFT engine is twiddle factor generation. Hardware requirements can be greatly reduced by exploiting the symmetry properties of the twiddle factor. As depicted in Figure C.5, only the values in region H need be stored; the rest of the required twiddle factors can be generated simply by inversion and transposition of the stored values. Table C.3 shows the twiddle factors for each point on the plane based on stored values for the real (R) and imaginary (I) values of the corresponding point in region H. Hence the minimum amount of read-only-memory required for twiddle factor storage is reduced to $N/8$.

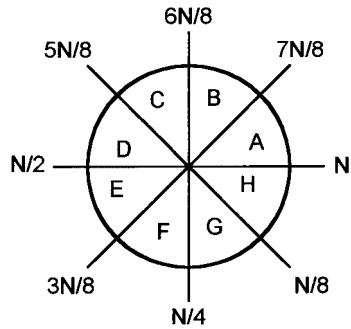


Figure C.5 Twiddle factors for *N*-Point FFT in the complex plane; only region marked as "H" is stored in ROM.

TABLE C.3 Twiddle Factor Generation

Region	Real[Twiddle Factor]	Imaginary[Twiddle Factor]
A	R	-I
B	-I	R
C	I	R
D	-R	-I
E	-R	I
F	I	-R
G	-I	-R
H	R	I

The points on this two-dimensional complex space depict values for W in counterclockwise direction, where

$$\mathrm{W} = e^{\mathrm{j}2\pi k/N} \qquad \text{for } k = 0, \ldots, N-1 \tag{C.10}$$

C.4 A REPRESENTATIVE FFT ENGINE IMPLEMENTATION

In this section, some of the design considerations for a representative FFT/IFFT engine are presented. The architecture is applicable to many different applications, including high-speed digital modems such as xDSL. In this implementation we are considering a 512-point FFT/IFFT engine. The processor performs a 512-point real-to-complex fast fourier transform in the FFT mode, and in the IFFT mode, it performs a 512-point complex-to-real inverse fourier transform.

C.4.1 Data Format

The data format should achieve the following objectives:

1. Maximize dynamic range of represented numbers.
2. Minimize representation error (maximize precision).

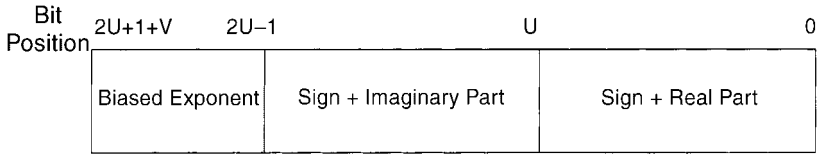


Figure C.6 Data format bit allocation.

3. Minimize storage area.
4. Facilitate FFT computations.

Signed-magnitude number representation was chosen to facilitate some of the FFT computational steps, and floating-point number representation was chosen for highest dynamic range and reduced storage area. Each data point is represented as a complex, sign-magnitude, floating-point number with one unsigned (implicitly negative) exponent that, in order to save memory space, is shared between the real and imaginary parts.[2] The mantissa is a fully fractional quantity, and the whole complex number is always less than 1, with 1 being used to indicate overflow conditions. That is,

$$\mathrm{DP} = (\mathrm{real} + \mathrm{j} * \mathrm{imaginary}) * 2^{-\mathrm{exp}} \tag{C.11}$$

Figure C.6 shows the data format comprising u real bits, u imaginary bits, ν exponent bits, and one sign bit. The dynamic range of the exponent is $L = 2^{\nu}$, so that

Magnitude of smallest represented number (precision)	$2^{-(L+u)}$
Magnitude of largest represented number	2^0
Range of biased exponent	$[0,2^{\nu}]$

NOTE: The same number representation must be followed on all the RAMs across the engine and all supporting system blocks. Overflow is prevented in each stage of the FFT operation by an effective scale-down performed at every stage. Furthermore, minimum exponent of each stage is tracked and it is ensured that all the exponents in the stage are scaled up by the complement of that minimum value, thus increasing the magnitude of the elements of each vector. The combination provides for smaller data sizes, prevention of overflow, and overscaling while maintaining low quantization noise levels.

C.4.2 FFT System Top-Level Architecture

The top-level system is designed as a superpipelined processing engine. Data are fetched from input files while computation is being done in the FFT processor

[2] In cases when it is not possible to normalize both parts of data using this scheme, there will be a slight precision loss.

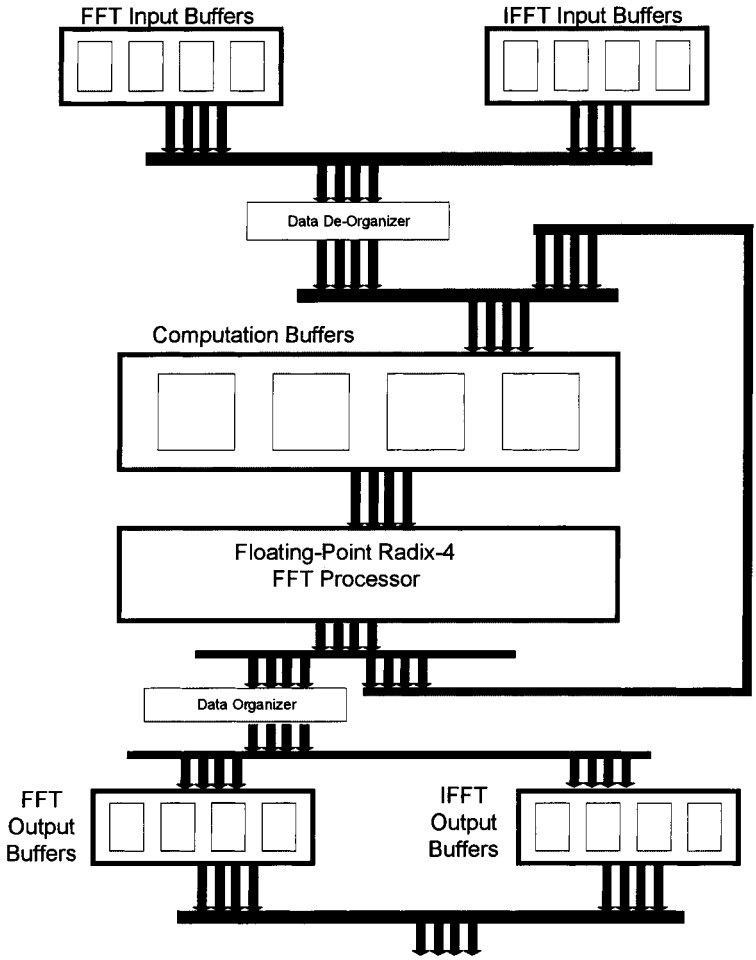


Figure C.7 Top-level block diagram.

and other data are being retired to the output stage. The processor itself is further pipelined to accommodate maximum throughput. The number of internal stages is designed to keep the output stage fully occupied as well as allowing real-time receipt of data into the input stage. Figure C.7 shows the block diagram of the FFT engine. The FFT processor is discussed further in the following sections.

C.4.3 Processor Pipeline Stages

As described earlier, for large values of N, DIT or DIF schemes are used to reduce the complex FFT operation into smaller values of N until the radix of the algorithm is reached. This process requires logr N stages of computation.

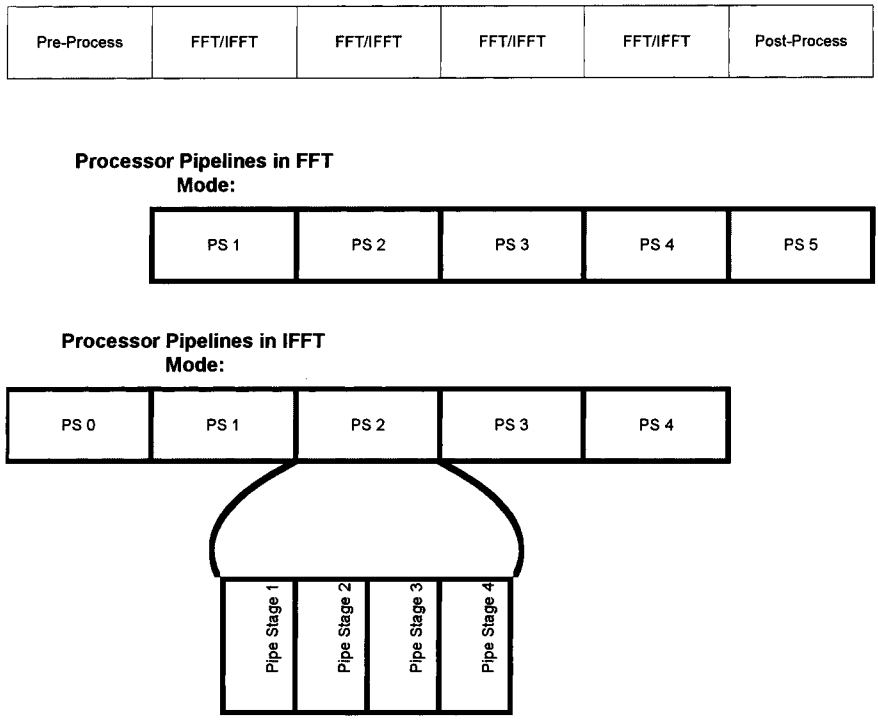


Figure C.8 Stages of the processor in FFT and IFFT modes.

Figure C.8 shows the pipeline stages for radix-4 processor for an effective 512-point FFT. The processor requires $\log_4(256) = 4$. Stages 1 through 4 refer to normal FFT stages, while stage 0 refers to preprocessing, which is active only in case of IFFT, and stage 5 refers to postprocessing, active only in FFT mode of operation.

There is a universal pipeline designed for each of the modes the FFT processor is operating in: raw_fft, pre_process and post_process. Some of the operations are modified or by-passed depending on the mode. A simplified list of operations that happen at each pipeline stage follows.

- *Stage 1*
  - Register incoming data
  - Perform sign-magnitude conversions
- *Stage 2*
  - Convert to fixed point
  - Multiplex data
  - Butterfly operations
  - Butterfly multiplications

  - Perform final additions for pre/post
  - Multiplex data
- *Stage 3*
  - Convert to floating for complex multiplies
  - Perform sign-magnitude conversions
  - Multiplex data
- *Stage 4*
  - Perform final adds
  - Convert to fixed point

C.4.4 Dedicated Storage Elements

Three types of storage systems are required for this type of processor:

1. Input buffers are used to receive data prior to entering the computation phase. In FFT mode they would store 512 real time-domain samples, and in IFFT mode they would store 256 complex frequency-domain taps.
2. Computation buffers store intermediate results for the various stages of in-place radix-4 FFT/IFFT computations.
3. Output buffers in FFT mode store 256 complex frequency-domain samples and in IFFT mode store 512 samples of time-domain data.

Dedicated input and output buffers for FFT and IFFT enable superpipelining requests for inverse and regular transforms. A top-level view of these storage modules is shown in Figure C.6.