Codesign Case Study on Transport-Triggered Architectures

39

Jarmo Takala, Pekka Jääskeläinen, and Teemu Pitkänen

Abstract

Application-specific processors are used to obtain the efficiency of fixed-function application-specific integrated circuits and flexibility of software implementations on programmable processors. The efficiency is achieved by tailoring the processor architecture according to the requirements of the application while the flexibility is provided by the programmability. In this chapter, we introduce a hardware/software codesign environment for developing application-specific processors, which is using processor templates based on the transport-triggering paradigm, hence the name transport-triggered architecture (TTA). Fast Fourier transform (FFT) is used as an example application to illustrate the customization. Specific features of FFTs are discussed, and we show how those can be exploited in FFT implementations. We have customized a TTA processor for FFT, and its energy efficiency is compared against several other FFT implementations to prove the potential of the concept.

Acronyms

ADF	Architecture Description File
ASIC	Application-Specific Integrated Circuit
ASP	Application-Specific Processor
CORDIC	COordinate Rotational DIgital Computer
DFT	Discrete Fourier Transfrom
DIF	Decimation-in-Frequency
DIT	Decimation-in-Time
DSP	Digital Signal Processor

J. Takala (✉) • P. Jääskeläinen
Tampere University of Technology, Tampere, Finland
e-mail: jarmo.takala@tut.fi; pekka.jaaskelainen@tut.fi

Teemu Pitkänen
Ajat Oy, Espoo, Finland
e-mail: teemu.pitkanen@ajat.fi

S. Ha, J. Teich (eds.), *Handbook of Hardware/Software Codesign*,
DOI 10.1007/978-94-017-7267-9_39

FFT	Fast Fourier Transform
HDB	Hardware Database
IR	Intermediate Representation
OSAL	Operation Set Abstraction Layer
RTL	Register Transfer Level
TCE	TTA-based Codesign Environment
TTA	Transport-Triggered Architecture

Contents

39.1 Introduction

Application-Specific Processors (ASPs) are used to obtain the efficiency of fixed-function Application-Specific Integrated Circuits (ASICs) and flexibility of software implementations on programmable processors. The efficiency is obtained by tailoring the processor architecture according to the requirements of the application as discussed earlier in ▸ Chap. 12, "Application-Specific Processors" and ▸ Chap. 33, "Hardware/Software Codesign Across Many Cadence Technologies". The simplest customization is to start with existing architecture and remove all the resources, which are not needed to execute the given application. In a similar fashion, an ASP may be constructed from library components, i.e., components are reused; thus the design process is shorter than in ASIC design. Even better efficiency can be obtained if the specific computation patterns in the application are identified and those are converted as accelerators or user-specific function units in the architecture.

The design space for application-specific processors is huge, and finding a suitable architecture for a given application will be an exhaustive work. The optimization strategies for design space exploration are covered in ▸ Chap. 6, "Optimization Strategies in Design Space Exploration" and architecture design space exploration in ▶ Chap. 8, "Architecture and Cross-Layer Design Space Exploration". Determining the machine code for an arbitrary processor architecture is extremely a difficult and time-consuming task especially when the processor contains parallelism. This work can be alleviated by limiting the search space. Such a constraint can be created by defining a processor template, which provides a set of customization parameters to vary the processor candidates. The limited set of parameters allows retargeting the software development toolchain; thus machine code of the given application can be generated on the customized architecture. The processor customization is an iterative process, thus during each iteration there is a need to port the application program to the new processor. This calls either for a manual assembly language program rewrite or a retargetable compiler, which can adapt to the changes in the architecture.

In this chapter, we introduce a processor template based on transport triggering paradigm and a software/hardware codesign environment supporting this template. We illustrate the use of the template and design environment by using Fast Fourier Transform (FFT) as an example application. FFT is used to compute Discrete Fourier Transfrom (DFT) of a sequence, which in turn converts the time domain representation of a digital signal to a frequency domain representation. The definition of DFT contains redundancy, and several methods have been proposed to avoid these redundancies. In general any method for computing DFT with lower arithmetic complexity than DFT is called a fast Fourier transform. FFT has been considered as "the most important numerical algorithm of our lifetime" [36], and nowadays it has gained popularity as frequency division has been used in many modern wireless communications standards.

In this chapter, we discuss FFT algorithms and illustrate some of their properties, which can be exploited when implementing the transform. This chapter shows how these properties can be exploited in implementations and a processor tailored for FFT is described. The energy efficiency of the tailored processor is compared against several implementations from the literature to show the efficiency of the approach.

39.2 Transport-Triggered Architecture Template

In this chapter, we use exposed data path as one of the characteristics of the architectural template, i.e., a template where many processor data-path details are visible to the programmer who can directly control those resources. Examples of such architectures are, e.g., MOVE [11], MOVE-Pro [18], FlexCore [42], STA [8], and ELM [12]. In particular, we exploit transport triggering paradigm [10], which defines that operation execution is initiated by data transport rather than operation defining the data transports as in traditional programming models. In Transport-Triggered Architecture (TTA) programming model, the program defines only data moves and the operations occur as side effects of data transports. In a way, transport triggering evokes the traditional data-flow model of execution. Operands to a function unit are moved via an interconnection network to input ports, and one of

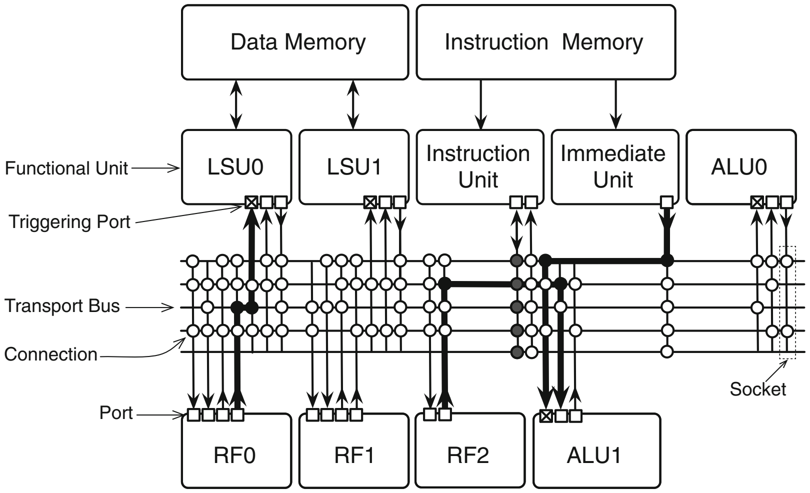


Fig. 39.1 TTA processor organization

the ports is dedicated as a trigger. Whenever data is moved to the trigger port, the operation execution is initiated. The program defines only the data moves on the interconnection network; thus the TTA processor has only one instruction: move. As the program defines moves in the interconnection network, the TTA processors have a programmer-exposed interconnection network.

An example TTA processor is depicted in Fig. 39.1. The interconnection network in this processor contains five transport buses implying that at most five data transports can be executed simultaneously. This also implies that each instruction contains five move slots, where each slot specifies the data transport carried out in each bus. The figure illustrates execution of an instruction with three parallel moves, i.e., instruction has three move slots:

$$\#4 \rightarrow \text{ALU1.i0.ADD};\ \text{RF2.r3} \rightarrow \text{ALU1.i1};\ \text{RF0.r1} \rightarrow \text{LSU0.i0.STW}$$

On the first transport bus, an immediate value is moved to the input port 0 of the function unit ALU1. The immediate value is actually obtained from the immediate unit, which has only one output port. As the function units can perform several operations, the move carries also information about the operation to be executed; opcode ADD is transported to function unit along with the operand. The second bus transports an operand from register r3 through the output port 0 of the register file RF2 to the input port 1 of the ALU1. The third bus is used to transport a value from register r1 in the register file RF0 through the output port 1 to the input port 0 of the load-store unit LSU0. The third move contains an opcode indicating that the transported word is to be stored to memory. The actual store address has been defined by another move to port 1 of the LSU0. The remaining two move slots are empty; thus the corresponding two buses are not used in this instruction. Thus they can be considered executing a NOP.

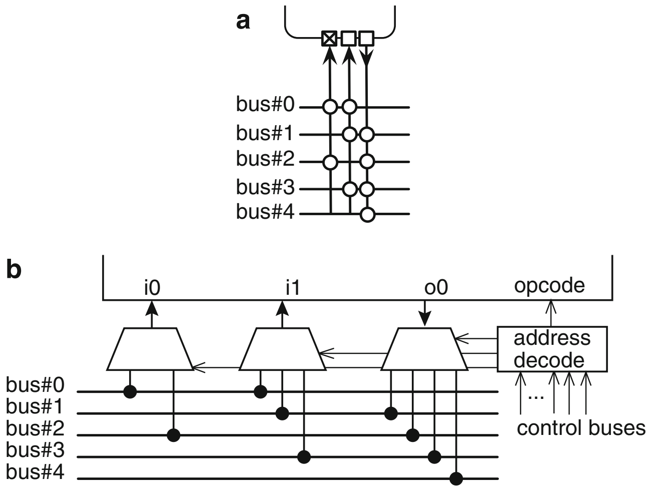


Fig. 39.2 Principal socket interface for function units: (**a**) high-abstraction-level representation and (**b**) structure

Figure 39.1 shows that the instructions control the operation of each transport bus through the instruction unit. The connection to each bus convoys control information, e.g., the source and destination of the transport move, possible opcode for the operation to be executed, etc. The function units are connected to the transport buses with the aid of sockets. The interconnection network in the architectural template consists of buses and sockets. The principal concept of sockets is illustrated in Fig. 39.2; each port of a function unit has a socket, which defines the connections to the buses. When the control information in a bus indicates that the port is the destination for the current move instruction, data from the bus is passed to the port. In a similar fashion, data from the source port is forwarded to the bus.

The architecture template defines that one of the input ports is a trigger port and a move to this port triggers the specified operation. The concept is illustrated in Fig. 39.3, where the trigger port is indicated by a cross in the input port. It should be noted that a function unit has only one trigger port. A move to this port will latch data from the bus to trigger register, and the operation execution starts with operands from the trigger port and other operand registers; the function unit in Fig. 39.3 expects two operands; thus there is one trigger register and one operand register. The operand to the operand register can be moved by an earlier instruction. The operand can also be moved in the same instruction as the trigger port moves if there are buses available to carry out the move. In Fig. 39.1, the first bus is used to transport an operand to trigger port of ALU1. The second bus moves the other operand from RF3; thus the operands for the specified ADD operation are moved to function unit at the same cycle. The move over the third bus triggers the store operation, but the actual store address has been moved to the input port 1 of LSU0 by an earlier instruction.

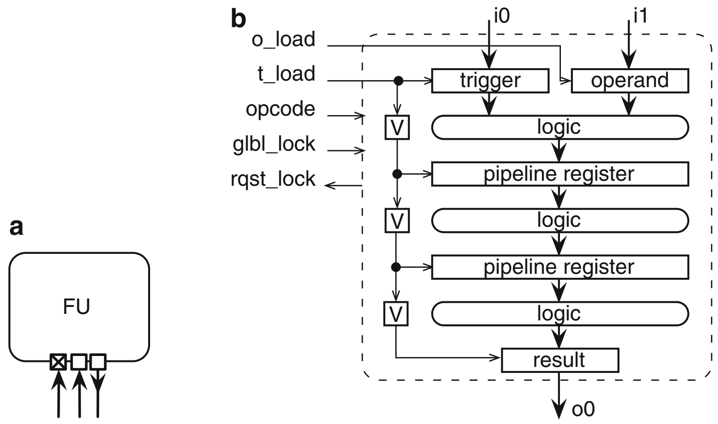


Fig. 39.3 Pipelined function unit based on semi-virtual latching: (**a**) high-abstraction-level representation and (**b**) principal block diagram. The result register in the output is optional

In TTAs, function units can be pipelined, and in the template, semi-virtual time latching [10] method is used, where valid bits control the pipeline as depicted in Fig. 39.3b. The pipeline starts an operation whenever there is a move to the trigger port, i.e., the *o_load* signal is active. As valid bits control the pipeline, a single pipeline stage is active only once for one trigger move. For example, Fig. 39.3b illustrates a case, where the result can be read from the result register three instructions after the trigger move.

An external or internal event can lock the processor (*glbl_lock* signal is active); all the function units in the processor have their pipelines stalled. The architectural template requires each operation in a function unit to have a deterministic latency such that the result read for the operation can be scheduled properly. If the function unit faces an unexpected longer latency operation, e.g., a memory refresh cycle or a function unit has iterative operation of which latency depends on the inputs, the unit can request the processor to be locked by activating the *rqst_lock* signal until the ongoing operation is completed.

In traditional statically scheduled machines, the timing between operand load, operation execution, and result store is fixed at design time. In TTAs, the timing is defined at compile time. For example, multiplication instruction defines completely when the operands are read from the registers R0 and R1 and result is stored to register R3:
MUL R3, R2, R1
while in TTA the corresponding operation can be specified with three different moves. When assuming a single transport bus and a function unit, which performs only multiplication and that the input port 0 is the trigger port, the previous instruction would be:

```
RF.r2 → MUL.i1;
RF.r1 → MUL.i0;
MUL.o0 → RF.r3
```

or if two transport buses are available:

RF.r2 $\rightarrow$ MUL.i1; RF.r1 $\rightarrow$ MUL.i0;
MUL.o0 $\rightarrow$ RF.r3;

However, the moves can even be scheduled over a large block of instructions:

RF.r2 $\rightarrow$ MUL.i1; . . . ; . . . ;
$\vdots$
. . . ; . . . ; RF.r1 $\rightarrow$ MUL.i0;
$\vdots$
MUL.o0 $\rightarrow$ RF.r3; . . . ; . . . ;

The previous examples show that there is a high degree of freedom on scheduling the moves over move slots in neighboring instructions compared to traditional statically scheduled machines, which makes the TTA scheduling a challenging problem.

The TTA instruction format reminds horizontal microcode, which usually shows poor instruction density. However, experiments show that the instruction overhead due to the exposed data-path control is negligible when comparing to the savings if the workload is data-intensive and the interconnection network is carefully optimized [21, 43].

The exposed data-path template opens unique optimization opportunities. For example, due to the explicit result transfers, the function units are independently executing isolated modular components in the data path. In the point of view of processor design methodology, the modularity allows point-and-click style tailoring of the data-path resources from existing processor component databases. It also means the function units can have arbitrary latencies and pipeline lengths from a single cycle because there is no hazard detection hardware. There is no practical limit to the number of outputs produced by operations.

The TTA template allows the processor to be customized in various ways. User can define the sets of basic TTA components to be included in the architecture. The number of register files can be varied, and each register file can have additional specifications: the number of registers, word width, and the number of read/write ports. Function units can be tailored by varying the number of function units, and for each function unit, it is possible to define the operation set implemented by the function unit, the number of input and output ports, the width of the ports, resource sharing/pipelining, and accessed address space (in case of a load-store unit). The operation set can be varied, and for each operation the number of operands, the number and data type of results and operands, and operation state data can be parametrized. Instruction encoding can be varied and the core can support a number of instruction formats. For each instruction format, the immediate (constant) support can be determined. Parameters related to address spaces include the number of address spaces. For each address space, size, address range, and the numerical id (referred to from program code) can be varied. Finally the parametrization allows even specification of multi-core systems.

An interesting customizable aspect in TTA processors is the interconnection network. As it is visible to the programmer, user can carefully tailor the connectivity according to the application. Another useful feature is the support for multiple disjoint address spaces: one can add one or more private address spaces for local memories inside a core that can be accessed using address space type qualifier attributes in the input C code.

39.3 Design Flow for Customizing Transport-Triggered Architectures

The design work described in this chapter is carried out with the TTA-based Codesign Environment TTA-based Codesign Environment (TCE) [40]. The codesign process supported by the TCE tools is illustrated in Fig. 39.4. Initially, the designer has a set of requirements and goals placed to the end result.

The iterative customization process starts with an initial predesigned architecture, which contains minimal resources to compile an arbitrary C program to run on the machine. The designer can add, modify, and remove architecture components using a graphical user interface tool called Processor Designer (ProDe) shown in Fig. 39.5, which creates the processor description in Architecture Description File (ADF) format. Each iteration of the processor can be evaluated by compiling the application code on the architecture with retargetable high-level language compiler and simulating the resulting parallel assembly code with the instruction-set simulator.

The simulator shows statistics of the run time of the program and the utilization of the different data-path components, indicating bottlenecks in the design. The processor simulator provides a compiled simulation engine for fast evaluation cycles and a more accurate interpretive engine for software debugging which supports common software debugging features such as breakpoints.

An essential feature in the processor customization process is the inclusion of custom operations. For a completely new processor operation, the designer describes the operation simulation behavior in C/C++ to Operation Set Abstraction

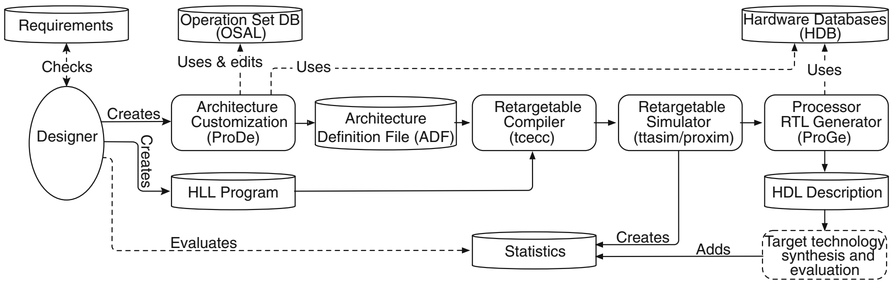

Requirements
Operation Set DB (OSAL)
Hardware Databases (HDB)
Checks
Uses & edits
Uses
Uses
Designer
Creates
Architecture Customization (ProDe)
Architecture Definition File (ADF)
Retargetable Compiler (tcecc)
Retargetable Simulator (ttasim/proxim)
Processor RTL Generator (ProGe)
Creates
HLL Program
HDL Description
Evaluates
Statistics
Creates
Adds
Target technology synthesis and evaluation

Fig. 39.4 TCE design flow for tailoring TTA processors

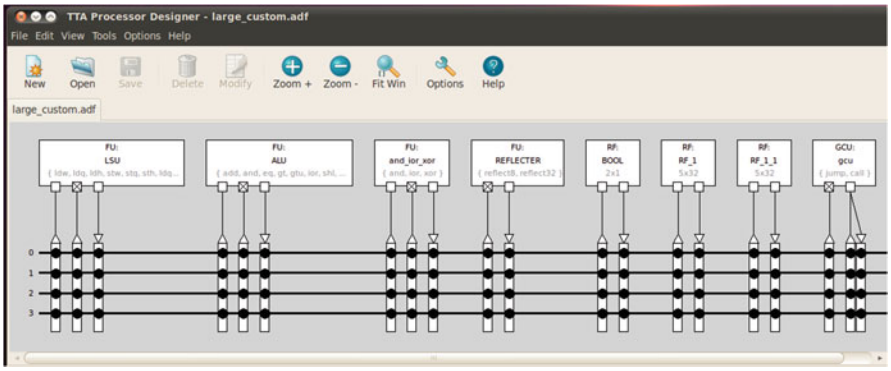

TTA Processor Designer - large_custom.adf
File Edit View Tools Options Help
New
Open
Save
Delete
Modify
Zoom +
Zoom -
Fit Win
Options
Help
large_custom.adf
FU: LSU { ldw, ldq, ldh, stw, stq, sth, ldq...
FU: ALU { add, and, eq, gt, gtu, ior, shl, ...
FU: and_ior_xor { and, ior, xor }
FU: REFLECTER { reflect8, reflect32 }
RF: BOOL 2x1
RF: RF_1 5x32
RF: RF_1_1 5x32
GCU: gcu { jump, call }
0
1
2
3

Fig. 39.5 Graphical user interface for ProDe tool

Layer (OSAL) database, estimates its latency in instruction cycles when implemented in hardware, and adds the operation to one of the function units in the architecture. This way it is possible to see the effects of the custom hardware to the cycle count, before deciding whether to include it in the design or not.

When a design point fulfilling the requirements has been found, or more accurate statistics of a design point is needed, the designer can generate synthesizable Register Transfer Level (RTL) description of the processor with processor generator (ProGe). For this step, the designer has to add RTL descriptions of the user-specific custom function units to Hardware Database (HDB). In order to alleviate this process, the function unit implementation is automatically verified against its architecture simulation model. The generated RTL can be synthesized with third party synthesis and simulation tools to obtain more detailed statistics of the processor. The TCE environment has an automated process to optimize the interconnection network, e.g., merging buses [43], removing function units until the performance does not increase, or removing connections which do not decrease the performance. The connectivity between components in larger TTA designs is hard to manage manually due to the huge space of options.

Manual assembly language coding would be the last optimization step after the final processor architecture has been selected. During the design process, however, assembly language is not feasible due to the architecture iteration process; whenever the architecture is changed, the affected parts of the assembly code would need to be rewritten.

In general, high-level language compilers cannot automatically exploit the complex custom operations in the processor for accelerating the program execution. Often compilers cannot extract all the inherent parallelism from the program description to exploit all the parallel processor resources. As high-level programming is typically preferred, the key tool in TCE is the retargetable software compiler, tcecc. The compiler uses LLVM [29] compiler framework as a backbone. The compiler supports C/C++ languages and has also support for the parallel OpenCL standard [22], in particular with *pocl* library [23]. The frontend supports the ISO

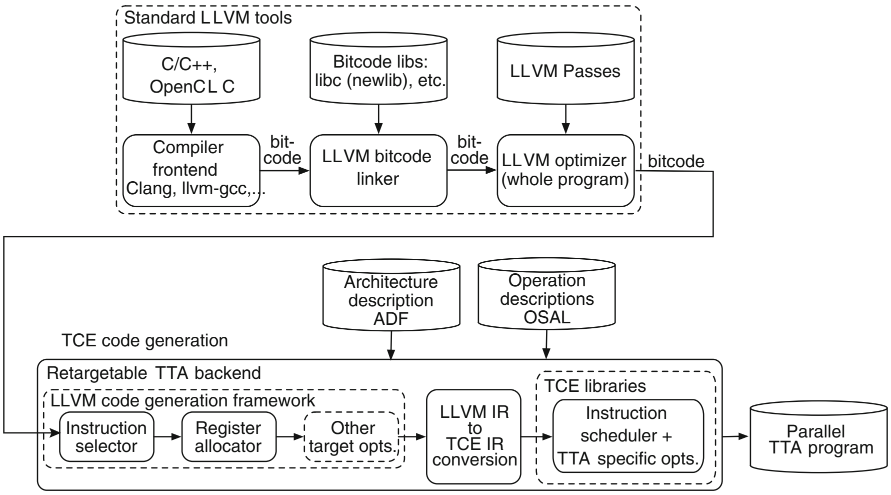

Standard LLVM tools
C/C++, OpenCL C
Bitcode libs: libc (newlib), etc.
LLVM Passes
Compiler frontend Clang, llvm-gcc,...
bit-code
LLVM bitcode linker
bit-code
LLVM optimizer (whole program)
bitcode
Architecture description ADF
Operation descriptions OSAL
TCE code generation
Retargetable TTA backend
LLVM code generation framework
Instruction selector
Register allocator
Other target opts.
LLVM IR to TCE IR conversion
TCE libraries
Instruction scheduler + TTA specific opts.
Parallel TTA program

Fig. 39.6 tcecc compiler

C99 standard with a few exceptions, most of the C++98 language constructs, and a subset of OpenCL C. Although TCE tools support multi-threading and multi-core systems [24], in this chapter we limit the discussion to single thread operation.

The main compilation phases of tcecc are shown in Fig. 39.6. Initially, LLVM's Clang frontend converts the source code to the LLVM internal representation. After the frontend has compiled the source code to LLVM bytecode, the utility software libraries are linked in, producing a fully linked self-contained bytecode program. Then standard LLVM Intermediate Representation (IR) optimization passes are applied to the bytecode-level program, and the whole-program optimizations can be applied aggressively. The optimized bytecode is then passed to the TCE retargetable code generation.

User-specific custom operations can be described in OSAL database as data-flow graphs consisting of primitive operations, which the LLVM instruction selector automatically attempts to detect and replace in the program code. Complex custom operations consisting of several primitive operations and dependencies between them or custom operations producing multiple results may not be automatically detected from intermediate code. Therefore, tcecc produces intrinsics that can be used manually in the source code.

39.4 Discrete Fourier Transform and Its Fast Algorithms

DFT is used to convert a finite sequence of equally spaced samples to a sequence of coefficients of a finite combination of complex sinusoids. In other words, the time domain representation of an N-point discrete time signal $x(n)$ is converted to frequency domain representation $X(r)$ as follows [31]:

$$X(r) = \sum_{n=0}^{N-1} x(n) W_N^{rn},\ r = 0, 1, \cdots, N-1, \tag{39.1}$$

where the coefficients W_N are defined as

$$W_N = e^{-j2\pi/N} = \cos(2\pi/N) - j\sin(2\pi/N), \tag{39.2}$$

where j denotes the imaginary unit. As the coefficients W_N are composed of sine and cosine functions, the coefficients W_N^{rn} have symmetry and periodicity properties, which implies that the DFT defined in (39.1) contains redundancy. By exploiting the underlying properties of the coefficients W_N^{rn}, several fast algorithms for DFT, i.e., FFTs, have been developed over the years. The most popular FFT is the Cooley-Tukey algorithm [9], where divide and conquer paradigm is used to decompose DFT into a set of smaller DFTs. In particular, the Cooley-Tukey principle states that a DFT of length $N = PQ$ can be computed with the aid of P-point DFT and Q-point DFT.

39.4.1 Radix-*p* Algorithms

If a factor N is not a prime, the Cooley-Tukey principle can be recursively applied and the larger DFT will be computed with the aid of several smaller DFTs. Especially, when the DFT length is a power of a prime, i.e., $N = p^q$, then the N-point DFT can be computed with the aid of p-point DFTs constructed in q computing stages. As the resulting fast algorithm contains only p-point DFTs, it is called a radix-p FFT. The most popular approach is radix-2 FFT algorithm, where the DFT is decomposed recursively until the entire algorithm is computed with the aid of 2-point DFTs as follows:

$$\begin{aligned} X(r) &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_N^{2nr} + \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_N^{2nr+1} \\ &= \sum_{n=0}^{\frac{N}{2}-1} x(2n) W_{\frac{N}{2}}^{2nr} + W_N^r \sum_{n=0}^{\frac{N}{2}-1} x(2n+1) W_{\frac{N}{2}}^{2nr},\ r = 0, 1, \cdots, N-1. \end{aligned} \tag{39.3}$$

This equation shows coefficients

$$W_N = e^{-j2\pi/N} \tag{39.4}$$

for the N-th root of unity. Its powers are referred to as *twiddle factors*.

The DFT decomposition can be carried out with two principal approaches: Decimation-in-Time (DIT) and Decimation-in-Frequency (DIF). In DIT approach,

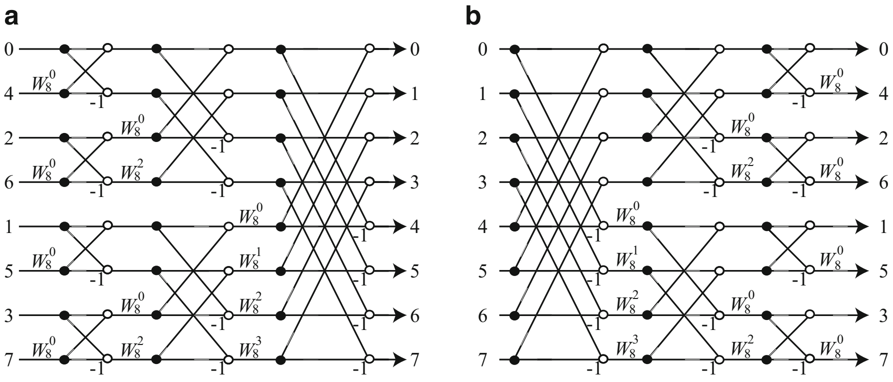


Fig. 39.7 Signal-flow graphs of 8-point radix-2 FFT: (**a**) decimation-in-time and (**b**) decimation-in-frequency algorithm. Circles represent addition

the decomposition is started in time domain sequence, while in DIF approach, the decomposition is started on frequency domain sequence. Both the approaches are illustrated in Fig. 39.7, where the signal-flow graphs of 8-point FFT derived with both approaches are shown. In the 8-point transform, the computations are carried out in three computing stages, where each column contains four 2-point DFTs. The principal computation building block in the radix-2 FFT is 2-point DFT in (39.3), which is also called a radix-2 butterfly. In Fig. 39.7, the weights -1 and W_N^r denote multiplication.

This work concentrates on the DIT approach, while both the DIT and DIF approaches result in the same arithmetic complexity but can be some other implementation-related differences. When implementing the algorithms with fixed-point arithmetic, there will be difference in the numeric accuracy due to quantizations carried out during the computations. Although the differences in signal-to-noise ratio (SNR) can be small, the DIT approach will result in better SNR in radix-2 algorithms [2]. Therefore, in this chapter, we exploit the DIT algorithms.

In the radix-2 butterfly, one complex multiplication and two complex additions are needed, each stage contains $N/2$ butterflies, and the number of the stages is $\log_2 N$, which gives the total of $\frac{N}{2}\log_2 N$ complex multiplications and $N\log_2 N$ additions for an N-point transform.

39.4.2 Radix-2^r Algorithms

Traditionally the most popular FFT have been the radix-2 FFTs, where computations are based on 2-input, 2-output butterflies depicted in Fig. 39.8. The radix-2 FFT is a special case in the class of radix-2^s FFTs [7]. The arithmetic complexity of FFT can be reduced by using greater than two radix if many of the complex coefficients turn out to be trivial (± 1 or $\pm j$). Let us consider the basic equation of the DFT in

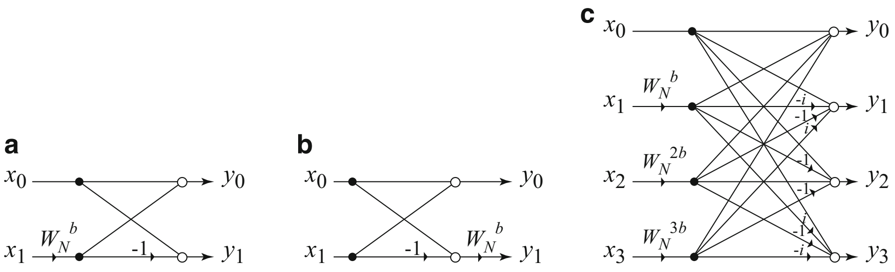


Fig. 39.8 FFT butterflies according to (**a**) radix-2 DIT algorithm in Fig. 39.7a, (**b**) radix-2 DIF algorithm in Fig. 39.7b, and radix-4 DIT algorithm in Fig. 39.9b

(39.1) and divide the original N-point problem to four partial sums by dividing the system to four sub problems, where the length of problem is $N/4$:

$$\begin{aligned} X(r) &= \sum_{n=1}^{N-1} x(n)W_N^{rn} \\ &= \sum_{n=0}^{N/4-1} x(4n)W_N^{r(4n)} + \sum_{n=0}^{N/4-1} x(4n+1)W_N^{r(4n+1)} + \sum_{n=0}^{N/4-1} x(4n+2)W_N^{r(4n+2)} \\ &\quad + \sum_{n=0}^{N/4-1} x(4n+3)W_N^{r(4n+3)}, r = 0, 1, \cdots, N-1. \end{aligned} \tag{39.5}$$

This method results in a radix-4 algorithm, where computations are based on 4-point DFT. This approach has benefits in terms of arithmetic complexity as 4-point DFT can be computed with trivial coefficients. In matrix form, the 4-point and 2-point DFT, F_4 and F_2, respectively, can be defined as

$$F_4 = \begin{pmatrix} 1 & 1 & 1 & 1 \\ 1 & -i & -1 & i \\ 1 & -1 & 1 & -1 \\ 1 & i & -1 & -i \end{pmatrix}; F_2 = \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix}. \tag{39.6}$$

While the radix-2 FFT has $\log_2 N$ computing stages, the radix-4 algorithm has only $\log_4 N$ stages, which results in significant savings in arithmetic complexity; e.g., a 64-point FFT can be computed in three stages while the radix-2 algorithm requires six computing stages. The arithmetic complexity for an N-point radix-4 FFT is $\frac{3N}{4} \log_4 N$ complex multiplications and $3N \log_4 N$ complex additions. The savings in multiplications (twiddle factors) are illustrated in Fig. 39.9.

From the implementation point of view, the lower number of arithmetic operations provides potential for faster computation and energy savings. In addition,

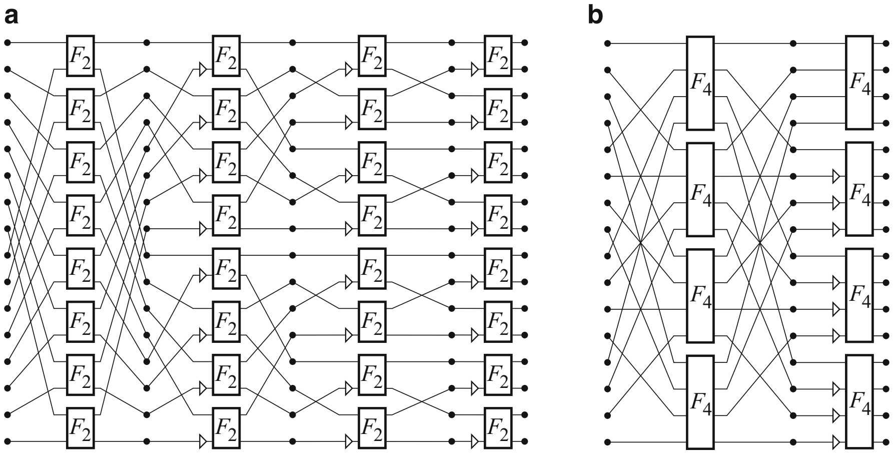


Fig. 39.9 Signal-flow graphs of in-place DIT FFTs: 16-point (**a**) radix-2 and (**b**) radix-4 FFT. Triangles represent a non-trivial twiddle factor

the latency of computations can be shorter. Besides lower arithmetic complexity, the radix-4 FFT provides also other advantages. The lower number of butterfly computation stages implies that, in memory based systems, less memory accesses are required. This speeds up the computations, reduces energy consumption, and relaxes the memory bandwidth requirements.

In this chapter, we exploit the in-order input, permuted output DIT radix-4 FFT defined as

$$F_{2^{2n}} = R_{2^{2n}} \left[\prod_{s=n-1}^{0} [P_{2^{2n}}^{s}]^{T} (I_{2^{(2n-2)}} \otimes F_4) D_{2^{2n}}^{s} P_{2^{2n}}^{s} \right], \tag{39.7}$$

where R_N is a permutation matrix defined as

$$R_{4^n} = \prod_{k=2}^{n} I_{4^{(n-k)}} \otimes P_{4^k,4}, \tag{39.8}$$

P_N^s is a permutation matrix of order N defined as

$$P_{2^n}^{s} = I_{4^s} \otimes P_{2^{(n-2s)},2^{(n-2s-2)}}, \tag{39.9}$$

and D_N^s is a diagonal matrix containing $N = 4^n$ twiddle factors as follows:

$$D_N^s = Q_N^s \left[\bigoplus_{k=0}^{N/4-1} \operatorname{diag}\left(1, W_{4^{s+1}}^{\lfloor k2^{s+1}/N \rfloor}, W_{4^{s+1}}^{2\lfloor k2^{s+1}/N \rfloor}, W_{4^{s+1}}^{3\lfloor k2^{s+1}/N \rfloor}\right) \right]; \tag{39.10}$$

$$Q_N^s = \prod_{k=0}^{s} P_{4^{(s-k)},4} \otimes I_{N/4^{(s-k)}} \ . \tag{39.11}$$

In the previous, $\otimes$ denotes tensor product, i.e.,

$$\begin{pmatrix} 0 & 1 \\ 2 & 1 \end{pmatrix} \otimes A = \begin{pmatrix} 0 & A \\ 2A & A \end{pmatrix} \tag{39.12}$$

and $\oplus$ denotes direct sum, i.e.,

$$A \oplus B = \begin{pmatrix} A & 0 \\ 0 & B \end{pmatrix}. \tag{39.13}$$

An example of FFT from (39.7) is depicted in Fig. 39.10.

The arithmetic complexity can further be reduced by selecting on even higher radix. However, in radix-8 and higher, the butterflies will contain nontrivial coefficients, and therefore, the relative arithmetic complexity is not decreasing as much. While radix-8 computations are applicable as they provide some advantages in specific implementation styles, higher radices are seldom used.

The main drawback of the radix-p FFTs is the fact that the length of the transform has to be a power of radix, $N = p^s$; i.e., radix-4 algorithms can only be applied when the transform length is a power of four. When the radix is higher, there are less sequence sizes where the algorithm can be applied. Due to this fact, radix-2 FFTs have been popular.

39.4.3 Mixed-Radix FFT

A method to reduce the arithmetic complexity compared to radix-2 FFT but still to support power of two transform lengths is mixed-radix approach, where the DFT decomposition contains several radices, e.g., the results of a 32-point FFT can be computed with two radix-4 stages and a single radix-2 stage. An example of mixed-radix FFT is shown in Fig. 39.11, where the signal-flow graph of a 32-point in-order input, permuted output DIT FFT based on radix-4 and radix-2 is illustrated.

In this chapter, we exploit the mixed-radix approach consisting of radix-4 and radix-2 computations, which provides best of the both worlds: lower arithmetic complexity of radix-4 FFTs and support for all the power-of-two transform sizes of radix-2 FFTs. The mixed-radix FFT consisting of radix-4 processing columns followed by a single radix-2 column can be defined as

$$F_{2^{2n+1}} = O_{2^{(2n+1)}} \, (I_{2^{2n}} \otimes F_2) \, C_{2^{(2n+1)}} \left[\prod_{s=n-1}^{0} [P^s_{2^{(2n+1)}}]^T \, (I_{2^{(2n-1)}} \otimes F_4) D^s_{2^{(2n+1)}} \, P^s_{2^{(2n+1)}} \right], \tag{39.14}$$

where the matrices P^s_N and D^s_N are defined in (39.9) and (39.10), respectively. The matrix O_N is a permutation matrix given as

$$O_N = (I_2 \otimes R_{4^n}) \, P_{N,2} \, , N = 2^{2n+1}. \tag{39.15}$$

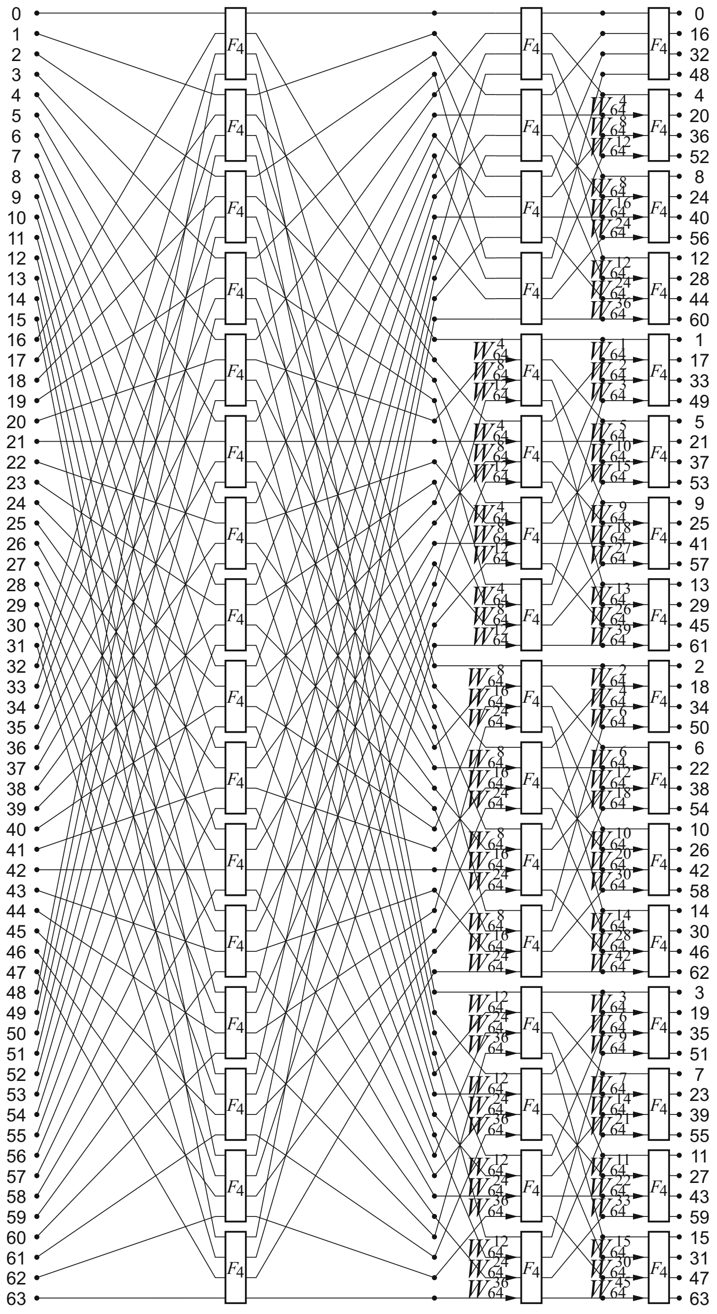


Fig. 39.10 Signal-flow graph of 64-point radix-4 DIT FFT. Numbers in the processing columns denote exponent k of twiddle factor, W_{64}^{k}

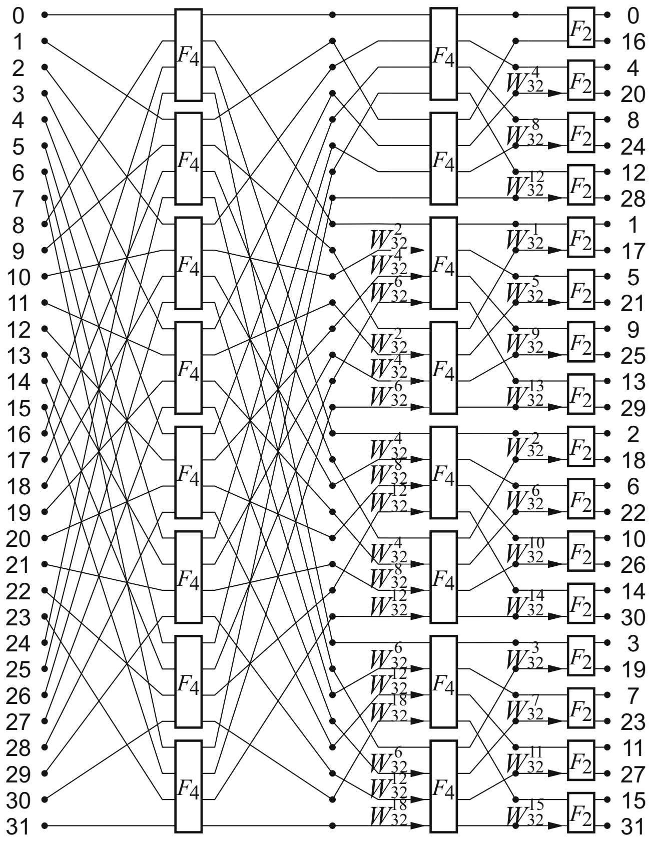


Fig. 39.11 Signal-flow graph of 32-point mixed-radix DIT in-place FFT algorithm

The matrix C_N contains the twiddle factors for the radix-2 processing stage, and it is defined as

$$C_N = Q_N^{\log_4(N/2)} \bigoplus_{k=0}^{N/2-1} \operatorname{diag}\left(1, W_N^k\right) , N = 2^{2n+1}, \qquad (39.16)$$

where the permutation matrix Q_N^s is defined in (39.11).

The mixed-radix approach allows us to design a system supporting multiple power-of-two FFT sizes. For example, in order to support the IEEE 802.16.1 OFDMA PHY [20], 256-point and 2048-point FFT transforms have to be realized, but the radix-4 FFT cannot be used to compute a 2048-point FFT, while mixed-radix approach is usable.

39.5 Building Blocks and Optimizations

In TTAs, application-specific function units could be exploited. One possible candidate for a special unit can be found by chaining up the operations executed; if an operation pattern is repeated in the application, it is a good candidate for a user-specific function unit. Such operation patterns can be, e.g., memory address generation, complex-valued addition, and complex-valued multiplication. The special units can also contain more specialized and complex functions like twiddle factor generator. The TTA template supports different latencies; thus the special function units can be pipelined to an arbitrary number of stages.

In this section, we discuss several properties of the previous FFT algorithms, which can be exploited when implementing the algorithm. In particular, the special features are used to construct user-specific functional units, which can be used in a TTA processor to speed up FFT computations.

39.5.1 In-Place Computations

In general, FFT algorithms are block processing algorithms, where computing is performed in processing stages consisting of butterfly computations. This is depicted in the signal-flow graphs of the algorithms, e.g., Figs. 39.9 and 39.11. Often in software implementations, double buffering [16] is used, i.e., operands are stored in an array and results are stored to another array and the role of buffers is exchanged for the next iteration. However, the previous signal-flow graphs illustrate that after the input operands for the butterfly operations are available and read from the memory locations, those operands are not needed any more, and the corresponding memory locations can be used to store the results of the butterfly. The results are used as operands for the butterflies in the following computing stage, i.e., computations can be performed in-place [26]. Exploitation of this property reduces significantly the memory requirements of software implementations.

39.5.2 Permutations and Operand Access

The FFT computations can be divided in butterfly computations, e.g., radix-2 FFT shown in Fig. 39.9a consists of 2-point butterfly computations, which each requires two operands and produces two results. The operands for butterflies are obtained with stride access, i.e., if the input sequence is stored in a memory array in order, the operands for butterfly computations in the first computing stage are located $N/2$ apart in the memory. In the second stage, the operands are $N/4$ apart. In software implementations, the operand index computation requires arithmetic operations, but, in application-specific implementations, this can be realized with lower complexity. When investigating the index addressing at bit level, it can be noted that addresses $N/2$ apart can be obtained from a linear address simply with the aid of rotation [7].

A linear address $(a_{N-1}, a_{N-2}, \ldots, a_0)$ is rotated to the right to obtain the operand access index $(a_0, a_{N-1}, a_{N-2}, \ldots, a_1)$. For example, the 2nd butterfly in the first processing stage in Fig. 39.9a reads operands from addresses 1 and 9; thus the mappings are $2_{10} = 0010_2 \rightarrow 1_{10} = 0001_2$ and $3_{10} = 0011_2 \rightarrow 9_{10} = 1001_2$. It should be noted that the access pattern of the operands for butterfly computations depends on the butterfly state s in the FFT signal-flow graph.

Different FFT algorithms have different operand access patterns. The operand indices for the first two processing stages of 64-point radix-4 DIT FFT in Fig 39.10 are listed in Fig. 39.12. The figure shows the decimal and binary representations of the indices, which reveal that the address mapping from linear address to operand index is a rotation. In the first stage shown in Fig. 39.12a, the bit-level mapping is rotation of two bits to the right in a 6-bit address. In the second stage listed in Fig. 39.12b, we can still see the same rotation of two bits to the right, but at this time the field to be rotated contains only four bits. This can be extended to a systematic method illustrated in Fig. 39.13; operand address mapping in 2^{2k}-point radix-4 DIT FFT in bit level is a rotation of two bits to the right in the $(2(k-s))$ least significant bits in the $2k$-bit linear address.

The mixed-radix approach uses yet another mechanism. Let us consider the mixed-radix FFT in Fig. 39.11. It should be noted that the length of the transform is now $N = 2^{2k+1}$, i.e., the index has an odd number of bits. The operand access sequence in the first processing stage is listed in Fig. 39.14a, which shows that the mapping in the bit level is rotation of two bits to the right in the 5-bit address. The addressing sequence in the second processing stage is listed in Fig. 39.14b, which indicates that the mapping is again 2-bit rotation, but the bit field to be rotated contains the three least significant bits in the address. The systematic mapping for mixed-radix FFT defined in (39.14) is illustrated in Fig. 39.15: operand address

a

Linear idx		Operand idx	
0	000000	0	000000
1	000001	16	010000
2	000010	32	100000
3	000011	48	110000
4	000100	1	000001
5	000101	17	010001
6	000110	33	100001
7	000111	49	110001
8	001000	2	000010
9	001001	18	010010
10	001010	34	100010
11	001011	50	110010
12	001100	3	000011
13	001101	19	010011
14	001110	35	100011
15	001111	51	110011

b

Linear idx		Operand idx	
0	000000	0	000000
1	000001	4	000100
2	000010	8	001000
3	000011	12	001100
4	000100	1	000001
5	000101	5	000101
6	000110	9	001001
7	000111	13	001101
8	001000	2	000010
9	001001	6	000110
10	001010	10	001010
11	001011	14	001110
12	001100	3	000011
13	001101	7	000111
14	001110	11	001011
15	001111	15	001111

Fig. 39.12 Operand address sequences 64-point FFT in Fig. 39.10: (**a**) the first computation stage, $s = 0$, and (**b**) the second computation stage, $s = 1$

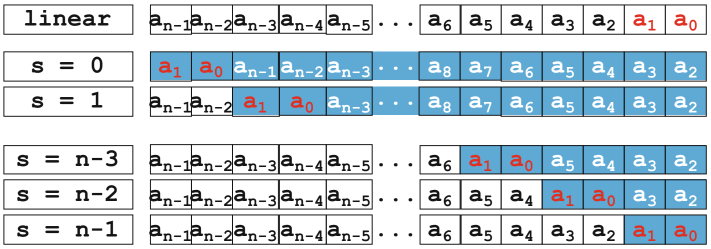
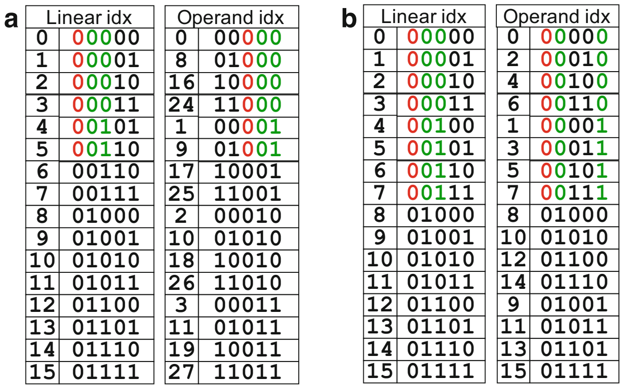

linear
s = 0
s = 1
s = n-3
s = n-2
s = n-1
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0
a_1 a_0 a_{n-1} a_{n-2} a_{n-3} . . . a_8 a_7 a_6 a_5 a_4 a_3 a_2
a_{n-1} a_{n-2} a_1 a_0 a_{n-3} . . . a_8 a_7 a_6 a_5 a_4 a_3 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_1 a_0 a_5 a_4 a_3 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_1 a_0 a_3 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0

Fig. 39.13 Bit-level operand address mapping for a 2^{2k}-point in-order, permuted output radix-4 DIT FFT. $n = 2k$



a

Linear idx		Operand idx	
0	00000	0	00000
1	00001	8	01000
2	00010	16	10000
3	00011	24	11000
4	00101	1	00001
5	00110	9	01001
6	00110	17	10001
7	00111	25	11001
8	01000	2	00010
9	01001	10	01010
10	01010	18	10010
11	01011	26	11010
12	01100	3	00011
13	01101	11	01011
14	01110	19	10011
15	01111	27	11011

b

Linear idx		Operand idx	
0	00000	0	00000
1	00001	2	00010
2	00010	4	00100
3	00011	6	00110
4	00100	1	00001
5	00101	3	00011
6	00110	5	00101
7	00111	7	00111
8	01000	8	01000
9	01001	10	01010
10	01010	12	01100
11	01011	14	01110
12	01100	9	01001
13	01101	11	01011
14	01110	13	01101
15	01111	15	01111

Fig. 39.14 Operand address sequences for 32-point mixed radix-4 and radix-2 FFT in Fig. 39.11: (**a**) the first computation stage, $s = 0$, and (**b**) the second computation stage, $s = 1$

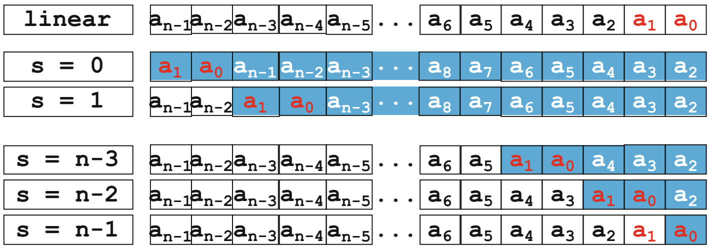
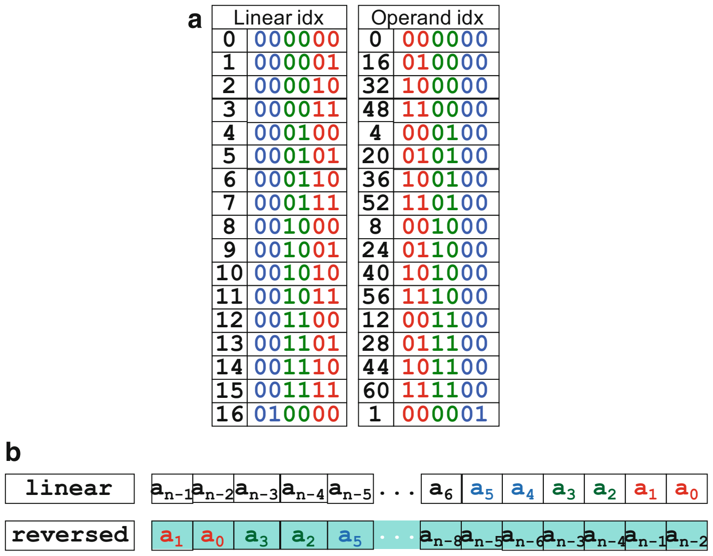

linear
s = 0
s = 1
s = n-3
s = n-2
s = n-1
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0
a_1 a_0 a_{n-1} a_{n-2} a_{n-3} . . . a_8 a_7 a_6 a_5 a_4 a_3 a_2
a_{n-1} a_{n-2} a_1 a_0 a_{n-3} . . . a_8 a_7 a_6 a_5 a_4 a_3 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_1 a_0 a_4 a_3 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_1 a_0 a_2
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0

Fig. 39.15 Bit-level operand address mapping for a 2^{2k+1}-point in-order, permuted output mixed radix-4 and radix-2 DIT FFT. $n = 2k + 1$

mapping in 2^{2k+1}-point mixed-radix DIT FFT in bit level is a rotation of two bits to the right in the $(2(k-s)+1)$ least significant bits in the $(2k+1)$-bit linear address.

There is yet another address mapping-related property in FFTs; the transforms contain permutations either in input or output or both. In radix-2 FFTs, the input or output permutations are the well-known bit-reversed permutations as seen in


a

Linear idx		Operand idx	
0	000000	0	000000
1	000001	16	010000
2	000010	32	100000
3	000011	48	110000
4	000100	4	000100
5	000101	20	010100
6	000110	36	100100
7	000111	52	110100
8	001000	8	001000
9	001001	24	011000
10	001010	40	101000
11	001011	56	111000
12	001100	12	001100
13	001101	28	011100
14	001110	44	101100
15	001111	60	111100
16	010000	1	000001

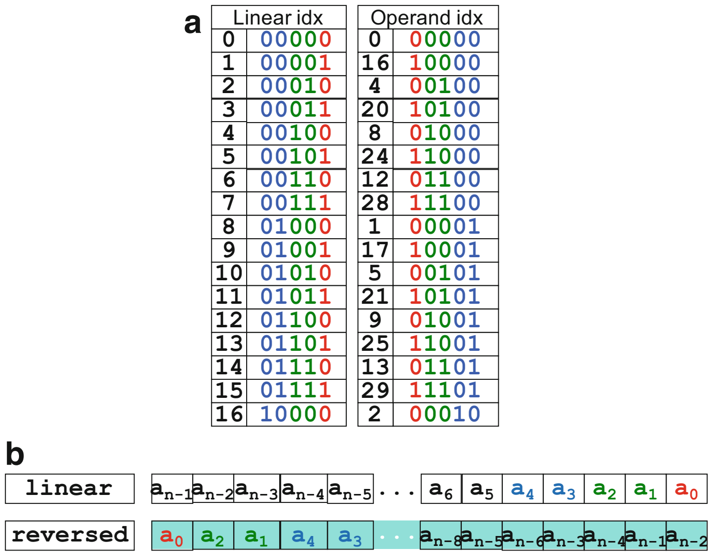
b
linear
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0
reversed
a_1 a_0 a_3 a_2 a_5 . . . a_{n-8} a_{n-5} a_{n-6} a_{n-3} a_{n-4} a_{n-1} a_{n-2}

Fig. 39.16 Bit-level address mapping for output permutation in a 2^n-point in-order, permuted output radix-4 DIT FFT. $n = 2k$

Fig. 39.9a. The address mapping is obtained simply by reversing the bit-level representation of the address. For example, in Fig. 39.9a, the addressing sequence is $(0, 8, 4, 12, 2, \ldots, 15)$, which is obtained from the linear address in bit level as $0_{10} = 0000_2 \rightarrow 0_{10} = 0000_2$, $1_{10} = 0001_2 \rightarrow 8_{10} = 1000_2$, $2_{10} = 0010_2 \rightarrow 0_4 = 0100_2$, $3_{10} = 0011_2 \rightarrow 12_{10} = 1100_2$, etc. It should also be noted that the inverse permutation of bit reversal is the same bit reversal.

The transforms considered in this chapter, radix-4 and mixed-radix algorithms defined in Eqs. (39.7) and (39.14), respectively, contain output permutations, and the permutations are different than in radix-2 algorithms. The output reordering in 64-point radix-4 algorithm illustrated in Fig. 39.10 is listed in Fig. 39.16a. The bit-level representation shows that the 6-bit linear address is reversed in 2-bit fields to obtain the index of the permuted element. The general method for address mapping for output permutation in a 2^{2k}-point radix-4 DIT FFT is depicted in Fig. 39.16b; the address mapping is reversal of 2-bit fields in a $2k$-bit address.

The mixed-radix FFT has a different output permutation. The 32-point FFT in Fig. 39.11 has the output index sequence listed in Fig. 39.17a, which again shows the reversal of 2-bit fields. However, this time the address field contains an odd number of bits; thus the least significant bit is moved to the most significant bit. The general case for 2^{2k+1}-point FFT is depicted in Fig. 39.17b. By using the previous bit-level presentations, the complexity of address generation can be reduced significantly compared to using worldwide arithmetic operations.


a

Linear idx		Operand idx	
0	00000	0	00000
1	00001	16	10000
2	00010	4	00100
3	00011	20	10100
4	00100	8	01000
5	00101	24	11000
6	00110	12	01100
7	00111	28	11100
8	01000	1	00001
9	01001	17	10001
10	01010	5	00101
11	01011	21	10101
12	01100	9	01001
13	01101	25	11001
14	01110	13	01101
15	01111	29	11101
16	10000	2	00010

b
linear
a_{n-1} a_{n-2} a_{n-3} a_{n-4} a_{n-5} . . . a_6 a_5 a_4 a_3 a_2 a_1 a_0
reversed
a_0 a_2 a_1 a_4 a_3 . . . a_{n-8} a_{n-5} a_{n-6} a_{n-3} a_{n-4} a_{n-1} a_{n-2}

Fig. 39.17 Bit-level address mapping for output permutation in a 2^n-point in-order, permuted output mixed-radix DIT FFT. $n = 2k + 1$

39.5.3 Twiddle Factors

The twiddle factors defined in (39.4) are an integral part of FFT algorithms, and often these coefficients are stored in a look-up table and fetched during computation of the algorithms. While this is a simple and quick method for short transforms, the table size increases superlinearly with the transform size. Therefore, the coefficients are created at run time when working with longer transforms.

The twiddle factors are actually complex roots of unity evenly spaced in the unit circle on complex plane [7] as seen in Fig. 39.18. The number of different factors depends on the FFT size N and the type of the fast algorithm. For example, radix-2 algorithms contain $\frac{N}{2}\log_2 N$ twiddle factors, but there are only $\frac{N}{2}$ different factors. According to (39.10), in a radix-4 algorithm, there are three nontrivial twiddle factors in each butterfly, thus there is a total of $\frac{3}{4}N\log_4 N$ nontrivial twiddle factors while only $\left(\frac{N}{2}-1\right)$ are unique.

The twiddle factors can be formed by using trigonometric functions, which are, however, expensive. The coefficients can be computed with fast algorithms, which exploit the trigonometric identities of twiddle factors, e.g., Singleton's method [35]. Then the coefficients can be computed on the fly, when they are needed. The twiddle factors can be generated as piecewise polynomial approximation of a function. Polynomial approximation requires multiplications and additions to compute the value of a function with given parameters. It should also be noted that the complexity of the polynomial-based algorithm increases significantly with the required

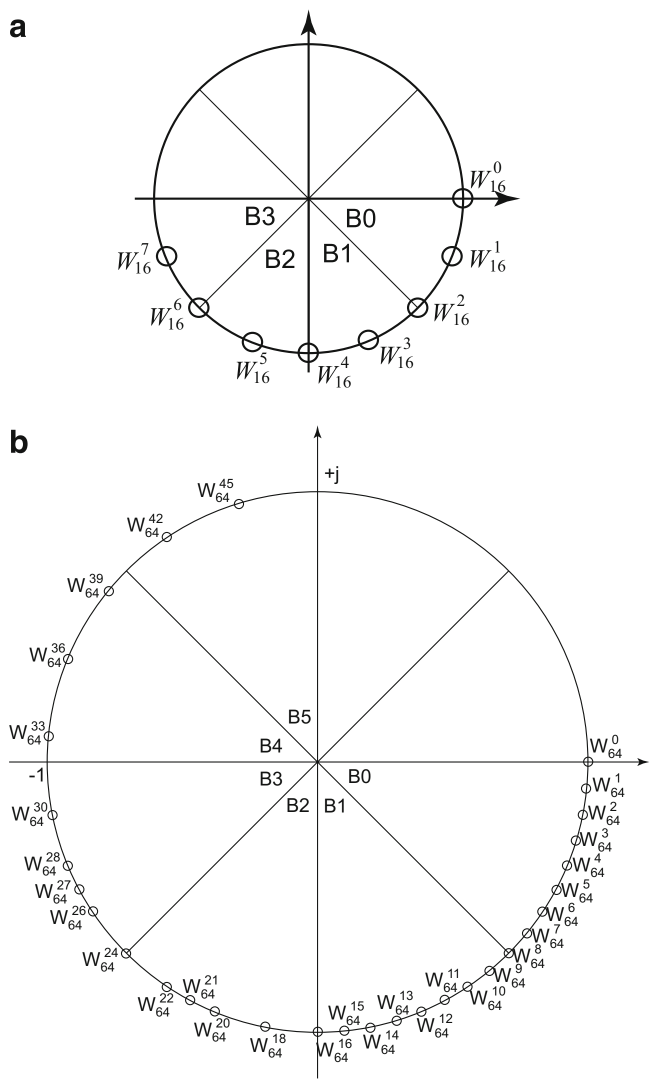


Fig. 39.18 Twiddle factors of (**a**) 16-point radix-2 and (**b**) 64-point radix-4 FFT in the complex plane

output precision. In [14], second-order polynomial approximation is combined with Horner's rule to compute the sine and cosine values.

Recursive twiddle factor generation is based on recursive feedback difference equations for sine and cosine functions. This approach is less complex compared to the polynomial, one iteration uses two real-valued multiplications and two real-valued additions to produce a complex-valued result. The drawback of the algorithm is error propagation of the finite numbers due to the feedback structure of the algorithm. In [6], a method to reduce the complexity of error propagation circuit is proposed. The accuracy is improved with a correction table containing $\frac{N}{8}$ 3-bit entries. The area cost is reduced by sharing the same multiplier and adder for both real and imaginary parts, which doubles the latency. The method uses two look-up tables for cosine and sine values, and both tables require $\log_2 N - 2$ entries. The drawback is that the method generates an ordered sequence of twiddle factors; thus it supports only a specific type of FFT algorithms and the reported unit supports only radix-2 DIF FFT. In these algorithms, the large number of iterations will increase the length of computation kernel. This might increase the need for intermediate storage, i.e., registers. Also the large number of multiplications will increase the power consumption of twiddle factor generation.

Another method to compute the twiddle factors is to exploit the COordinate Rotational DIgital Computer (CORDIC) algorithm [13, 25, 44]. All of the trigonometric functions can be evaluated by rotating a unit vector in complex plane. This operation is effectively performed iteratively with the CORDIC algorithm. The general rotation transform at iteration t can be given as

$$\begin{cases} X_{t+1} = X_t \cos\phi - Y_t \sin\phi \\ Y_{t+1} = Y_t \cos\phi + X_t \sin\phi \end{cases}, \tag{39.17}$$

where (X_{t+1}, Y_{t+1}) is the resulting vector generated by rotation of an angle ϕ from the original vector (X_t, Y_t), i.e., the resulting vector rotates in the unit circle in similar fashion as the twiddle factors. Therefore, the CORDIC algorithm can be used to compute the twiddle factors, generating the sine and cosine values. In particular, CORDIC is used for replacing the twiddle factors with rotation information and, therefore, avoids multiplication with the twiddle factor by replacing it with rotation realized with additions.

The CORDIC multiplier consumes less power compared to a traditional multiplier. For example, in [47], a pipelined CORDIC unit consumed roughly 20% less power than the traditional complex-valued multiplier while the area cost was about the same. Recursive CORDIC iteration saves area compared to look-up-based twiddle factors, but it introduces longer latency. In [47], the rotation angle constants for generating all the twiddle factors for an N-point FFT are stored in a look-up table with $\log_2 N$ entries, while in [15], the twiddle factors are generated without pre-calculated coefficients. The CORDIC algorithm is iterative; thus it can be pipelined easily and it lends itself to pipelined FFT architectures. However, the dynamic power consumption with a large number of iterations and/or long pipeline will be higher than in a look-up table-based approach. This will be the case, when longer word widths are used, i.e., increased accuracy calls for more iterations. Traditionally the CORDIC has mainly been used in fixed-function ASICs, but it can be used to accelerate computations in a programmable processor as reported in [34]. The authors describe instruction extensions for CORDIC operations, and there are separate instructions for vectoring and rotation mode.

Another approach is to exploit look-up tables and read the coefficients from the tables. In many cases, the look-up tables are stored in ROM, but, in software implementations, data memory is used to store the coefficients. The simplest design, radix-2, requires $\frac{N}{2}\log_2 N$ twiddle factors to be stored in the table. However, such a table contains redundancy as many of the coefficients are the same. In [27, 46], a method to reduce the number of coefficients in radix-2 algorithms to $\frac{N}{2}$ is proposed. Such a table can be used only for a sequential implementation, but, in [28], a method is proposed, which allows the $N/2$ entries to be distributed over $2^P, P = 0, 1, \ldots, \log_2\left(\frac{N}{2}\right) - 1$ sub-tables such that those can be accessed by 2^P butterfly units simultaneously.

The previous twiddle factor table contains still redundancy: as the twiddle factors are equally spaced in unit circle on the complex plane, there is symmetry as illustrated in Fig. 39.18a. We can note that the real and imaginary parts of the twiddle factors in the octants B_0 and B_1 can be used to obtain the twiddle factors required in the octants B_2 and B_3. The number of look-up table entries in the radix-2 case can be reduced down to $\frac{N}{4} + 1$. Such an approach has been presented in [3, 30, 39]. In this method, the twiddle factors from the octants B_0 and B_1 in Fig. 39.18 are stored to the look-up table, and the rest of twiddle factors are generated simply by interchanging the real and imaginary parts of the coefficient and changing the sign according to the octant.

In the previous, the symmetry among different quadrants is exploited, but the symmetry between real and imaginary parts of the twiddle factors is not exploited. This would allow all the twiddle factors to be generated from only one octant, B_0 in Fig. 39.18a. In [17], method is shown, which avoids these redundancy twiddle factors for radix-2 FFTs are created with the aid of $\frac{N}{8}+1$ complex-valued constants. In [32], this is extended to cover also radix-4 FFTs. The method can be used to construct twiddle factors for several transform sizes.

The redundancy in twiddle factors can easily be seen in Fig. 39.19; in order to represent all the different twiddle factors in 64-point radix-4 FFT, only the nine twiddle factors in the octant B0 are needed. For example, twiddle factor W_{64}^{14} in the octant B1 is obtained with the aid of W_{64}^{2} in octant B0: $W_{64}^{14} = -i{W_{64}^{2}}^{*}$, where * denotes complex conjugate. In a similar fashion, $W_{64}^{18} = -iW_{64}^{2}$. In general case, for an N-point transform, we store the values from B0 to a table M:

$$M = (M_0, M_1, \ldots, M_{N/8}), \tag{39.18}$$

where an entry M_k in the table represents a twiddle factor W_N^k, which is computed based on the exponent k as follows:

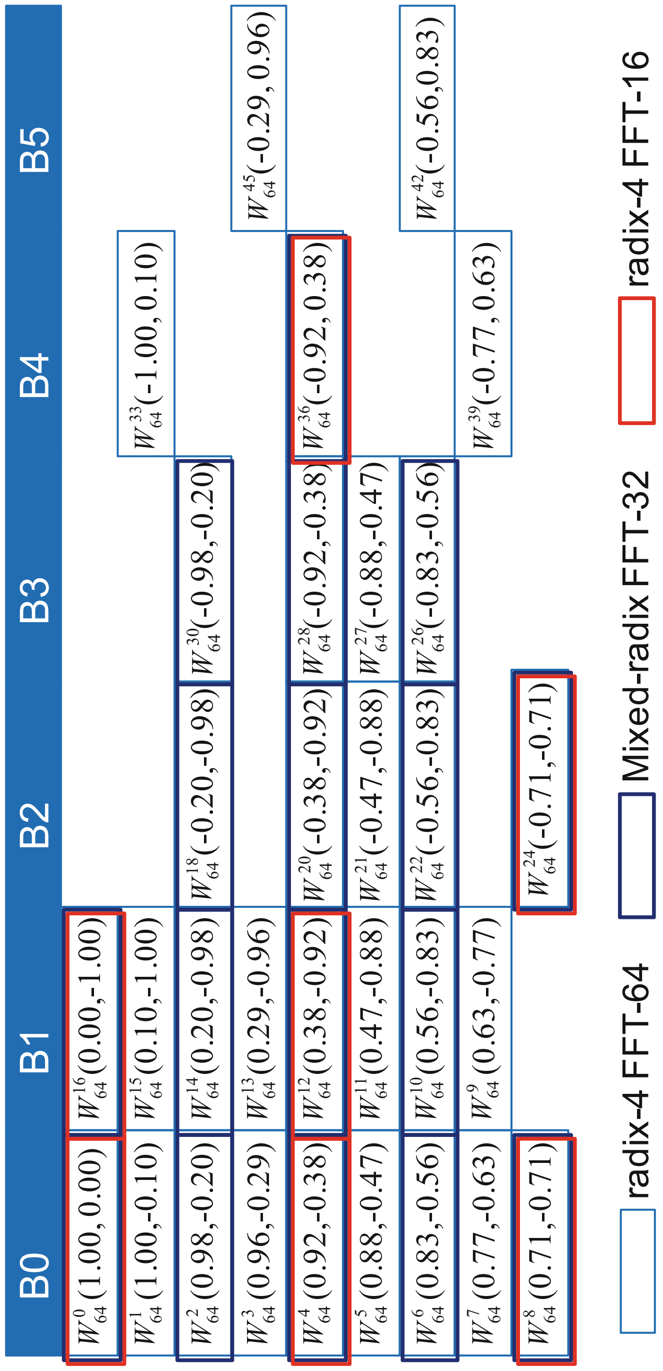

B0
B1
B2
B3
B4
B5
W_{64}^{0}(1.00,0.00)
W_{64}^{16}(0.00,-1.00)
W_{64}^{1}(1.00,-0.10)
W_{64}^{15}(0.10,-1.00)
W_{64}^{33}(-1.00,0.10)
W_{64}^{2}(0.98,-0.20)
W_{64}^{14}(0.20,-0.98)
W_{64}^{18}(-0.20,-0.98)
W_{64}^{30}(-0.98,-0.20)
W_{64}^{3}(0.96,-0.29)
W_{64}^{13}(0.29,-0.96)
W_{64}^{45}(-0.29,0.96)
W_{64}^{4}(0.92,-0.38)
W_{64}^{12}(0.38,-0.92)
W_{64}^{20}(-0.38,-0.92)
W_{64}^{28}(-0.92,-0.38)
W_{64}^{36}(-0.92,0.38)
W_{64}^{5}(0.88,-0.47)
W_{64}^{11}(0.47,-0.88)
W_{64}^{21}(-0.47,-0.88)
W_{64}^{27}(-0.88,-0.47)
W_{64}^{6}(0.83,-0.56)
W_{64}^{10}(0.56,-0.83)
W_{64}^{22}(-0.56,-0.83)
W_{64}^{26}(-0.83,-0.56)
W_{64}^{42}(-0.56,0.83)
W_{64}^{7}(0.77,-0.63)
W_{64}^{9}(0.63,-0.77)
W_{64}^{39}(-0.77,0.63)
W_{64}^{8}(0.71,-0.71)
W_{64}^{24}(-0.71,-0.71)
radix-4 FFT-64
Mixed-radix FFT-32
radix-4 FFT-16

Fig. 39.19 Twiddle factors in 16-point radix-4 FFT, 32-point mixed-radix, and 64-point radix-4 FFTs in the different octants in complex plane in Fig. 39.18b

$$W_N^k = \begin{cases} M_A \text{ , when } 0 \le k \le \frac{N}{8} \\ -jM_A{}^* \text{ , when } \frac{N}{8} < k < \frac{N}{4} \\ -jM_A \text{ , when } \frac{N}{4} \le k \le \frac{3N}{8} \\ -M_A{}^* \text{ , when } \frac{3N}{8} < k < \frac{N}{2} \\ -M_A \text{ , when } \frac{N}{2} \le k \le \frac{5N}{8} \\ jM_A{}^* \text{ , when } \frac{5N}{8} < k \end{cases}, \tag{39.19}$$

where A is an index to the look-up table M obtained from the given exponent k. For N-point FFT, $N = 2^n$, k is represented with n bits; thus when using two's complement representation, the $(n-2)$-bit index A is obtained simply as

$$A = \begin{cases} k[n-3:0] \text{ , when } 0 \le k \le \frac{N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{N}{8} < k < \frac{N}{4} \\ k[n-3:0] \text{ , when } \frac{N}{4} \le k \le \frac{3N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{3N}{8} < k < \frac{N}{2} \\ k[n-3:0] \text{ , when } \frac{N}{2} \le k \le \frac{5N}{8} \\ \sim k[n-3:0]+1 \text{ , when } \frac{5N}{8} < k \end{cases}, \tag{39.20}$$

where $k[a:b]$ denotes the bit field $(k_a, k_{a-1}, \ldots, k_{b+1}, k_b)$ of a two's complement number $k = (k_{n-1}, \ldots, k_1, k_0)$ and $\sim$ is the bit-wise complement operation.

The look-up table can be used to create the twiddle factors in all the power-of-two FFTs smaller than N. This can be seen in Fig. 39.19: the twiddle factors in a 32-point mixed-radix FFT are a subset of twiddle factors in 64-point radix-4 FFT. The access to the table requires only a simple manipulation of parameter k as the twiddle factors in a 32-point FFT are every second twiddle factor in a 64-point FFT. In a similar fashion, the twiddle factors in a 16-point radix-4 FFT are a subset of twiddle factors in the 32-point FFT. A block diagram of a twiddle factor unit supporting all the power-of-two FFTs with 16-bit real and 16-bit imaginary precision is illustrated in Fig. 39.20. The actual twiddle factor generation requires only negation of sine and cosine values read from the look-up table as defined in (39.18). According to (39.20), the index to the look-up table is formed with simple operations: increment and complement, and modification of complex entries from the table uses only few simple gates and two full adders.

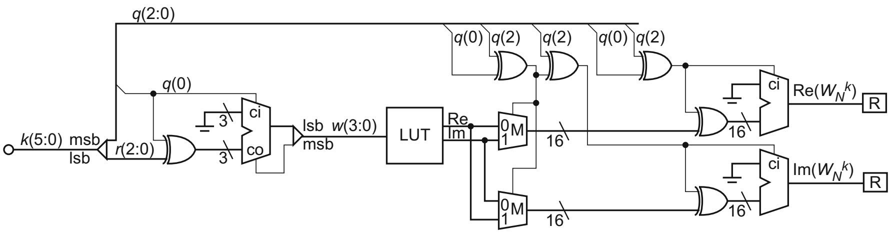


Fig. 39.20 Twiddle factor unit supporting all the power-of-two FFTs up to 64-points [32]

39.6 Customized FFT Architecture Based on Transport Triggering

The properties and optimizations discussed in the previous section can be used to tailor a single-thread transport-triggered processor for FFT computations. In this section, we describe the architecture, which is tailored by incorporating special function units discussed in the previous section.

The TTA template has been tailored according to the needs of radix-4 and mixed-radix FFT, and the resulting architecture can be seen in Fig. 39.21. The processor contains a set of standard function units, which has simply been taken from TCE hardware database, i.e., no design effort has been used to those units. The standard units include an instruction unit for controlling the operation, an immediate unit for extracting an immediate value from an instruction and passing it to the interconnection network, load/store units for accessing the data memories, a logical unit for standard logical operations, a comparator unit, and a shifter unit for arithmetic and logical shifts. There are several register files, which imply that the several temporary variables are accessed in the iteration kernel. There is one Boolean register for storing results of comparison, and this register can be used for predication to avoid costly branches.

The processor uses 32-bit arithmetic and packs a complex number in a single 32-bit word. There are 18 buses in the interconnection network; 17 are 32-bit buses (many of those are point-to-point buses), and there is a single 1-bit bus, which is used to transfer the Boolean results from comparisons. All these buses are generated by the ProGe tool once the processor architecture has been described with ProDe tool. The data memory is organized as a parallel memory consisting of two single-port memories with switching, which allows memories to be accessed through either of the two load/store units. This organization allows two memory accesses per clock

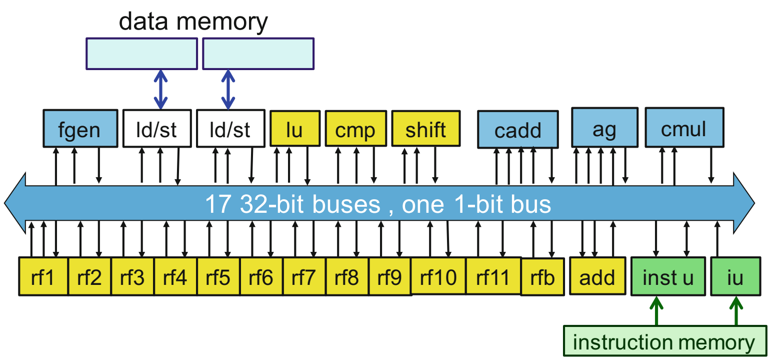


Fig. 39.21 Block diagram of TTA processor tailored for FFT computations. *fgen* twiddle factor generation unit, *ld/st* load/store unit, *lu* logical unit, *cmp* compare unit, *shift* shift unit, *cadd* complex-valued butterfly adder, *ag* operand address generation unit, *cmul* complex-valued multiplier, *rf* register file, *rfb* Boolean register, *add* adder unit, *inst u* instruction unit, *iu* immediate unit

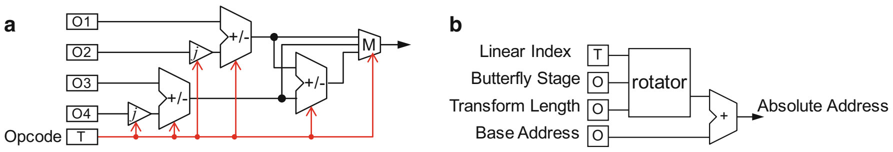


Fig. 39.22 Block diagram of (**a**) complex-valued butterfly adder and (**b**) operand address generator

cycle. The energy efficiency of the parallel memory is significantly better than the corresponding dual-port memory.

Finally, the processor has four special function units. The hardware structure of the units has been designed manually by exploiting the standard unit interface of the TTA template. There are separate units for complex-valued multiplication and complex-valued butterfly addition. The complex adder unit computes for different additions of four operands defined in 4-point DFT and two summations from 2-point DFT in (39.6). The block diagram of the complex butterfly unit is depicted in Fig. 39.22a. The unit has four operand ports and computes one of the outputs of butterfly operation when the opcode is transferred to the trigger port. The idea is that the four operands can be stored in the input registers over four consecutive clock cycles; thus there is no need to move operands, which reduces power consumption. The same unit can also be used to compute radix-2 butterflies when realizing mixed-radix FFTs.

The operand addresses are computed with a dedicated address generator unit illustrated in Fig. 39.22b. This is simply a rotator with an adder for adding the rotated index to the base address of the memory array. Once again, the linear index address is used as trigger port; thus during the FFT computations, all the other parameters are kept in input registers and operand moves can be avoided. If standard native arithmetic operations would be used for operand generation, it would take up to six operations. Here the customized unit can generate operand address at every clock cycle, which is sufficient to support two load/store units, as we perform in-place computations, i.e., the same address is used to read operand and store result.

The complex-valued twiddle factors are generated with a dedicated twiddle factor unit, which is based on the principle illustrated in Fig. 39.20. The unit can generate a new twiddle factor at rate of one per cycle. There are also some standard functional units for supporting control code. Many of the functional units are pipelined to support high clock frequency. The units have been designed such that during the FFT kernel computations, the throughput is one operation per clock cycle.

The programmability of the processor is usually limited if heavy customizations are used, i.e., when general-purpose functional units are removed. However, this architecture is still programmable, but the performance in general applications is limited. The TCE tools can be used to compile code on the customized architecture.

The code for FFT application is developed by exploiting heavily software pipelining and loop unrolling. Figure 39.23 shows the reservation table of the 17 buses during the computation of radix-4 FFT. Each color in the figure denotes move

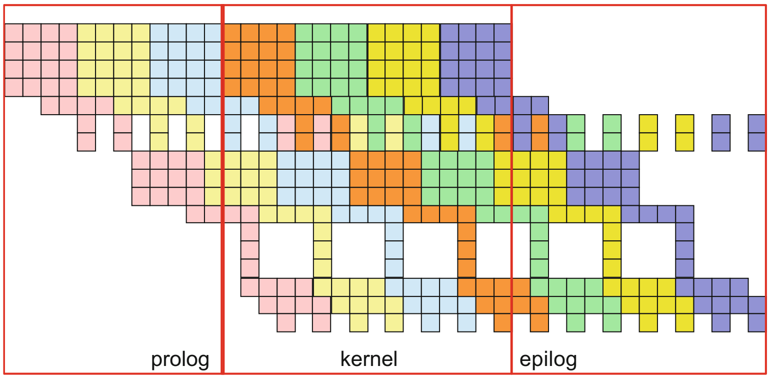


Fig. 39.23 Reservation table for the radix-4 FFT code

instructions related to computation of a single radix-4 butterfly with twiddle factors. The figure shows that the many of the resources are fully reserved during the kernel computation. The actual computation kernel contains 16 instructions while it could have been also shorter. However, the intermediate results are stored in register files over few clock cycles; thus the shorter kernel would require that the intermediate values would be stored in a FIFO type of storage. This would need data to be moved from register to register every cycle, which would consume extra power. Therefore, the intermediate values are kept in registers and kernel code need to be accessed from memory. However, in this specific case, we exploited code compression [19]; thus the instruction word is short.

Two versions of processors are developed: one with larger memory for supporting all the power-of-two FFT up to 16k-points and another version with smaller memory to support only 1-point FFT. Both the processors have the same processor core, but data memory and software are different.

Both the processors can perform two memory accesses per cycle; thus the overhead of the system can be compared by determining the theoretical lower bound for the number of memory cycles required for computing 1024-point radix-4 FFT. The radix-4 computation requires $1024 \log_4(1024)$ memory reads and memory writes; thus for a two-port memory, a total of 5121 memory accesses are needed. In the customized TTA core, computation of 1k-point FFT takes 5208 cycles; thus it shows really low overhead. The processors have been synthesized on a 130nm IC technology, and analysis shows maximum clock frequency of 250 MHz at 1.5 V supply voltage and 140 MHz at 1.1 V. The processor core takes 33 kgates, and in the smaller memory configuration for 1k-point FFT, memory is 30 kgates, while the larger memory supporting 16k-point FFT takes 240 kgates. The total power consumption for 1k-point FFT is 59 mW@250 MHz with 1.5 V supply voltage, where the smaller memory uses 16 mW. The most power hungry unit in the core is the twiddle factor generator, which takes about 23% of the core area and 7% of the power consumption. When several transform sizes are supported with the larger memory, the power consumption is higher when mixed-radix code is executed, e.g., computing

The major effort in the actual design work was spent on finding out the specific features of the algorithm, which can be exploited to speed up the computations. The effect of candidate features was analyzed by creating a high-abstraction-level model of the function unit, which was then used in simulator. Once the unit was verified to be useful, only then RTL code for the unit was developed; thus there was a need to develop RTL code only for four units. The RTL for the processor was generated with ProGe tool, and the design was synthesized with commercial IC design tools.

39.7 Energy Efficiency Comparison

Power consumption is a usual design metric when designing energy-efficient systems. However, power consumption depends on several issues: computing resources, memories, caches, computation cycles, operating voltage, and operation frequency. The energy efficiency can be compared by measuring the energy consumed for performing a reference task. Here we compare the energy efficiency by measuring how many 1024-point FFTs can be computed with energy of 1 mJ. This approach tries to compensate the effect of computational speed, but there are other implementation-specific parameters, which have a great effect on the result.

There are still some parameters, which may differ, such the implementations should be normalized. Although exact scaling of the characteristics of an implementation on a specific IC technology to another technology is difficult, even impossible, there are several normalization methods proposed in the literature. A normalization method for IC technologies is proposed in [38], which tries to take into account many architectural aspects and implementation-specific features; the normalized energy consumption of a system, E_N, is defined as

$$E_N = E\frac{L_r U_r^2 \left(\frac{1}{3}W_r^2 + \frac{2}{3}W_r\right)}{LU^2 \left(\frac{1}{3}W^2 + \frac{2}{3}W\right)}, \tag{39.21}$$

where E is the energy consumption of the system implemented on a specific IC technology, W is the word length of the system, U is the supply voltage of the implementation, and L is feature size of the specific IC technology on which the system has been implemented. The energy consumption of the implemented system is normalized for the same system implemented on reference technology, where L_r is the feature size of the reference IC technology, U_r is the supply voltage of reference technology, and W_r is the word length of the reference design.

We have compared the energy efficiency of the developed TTA processor against several other FFT implementations by using the previous normalization. The following parameter set has been used: $L_f = 130\,\text{nm}$, $U_r = 1.5\,\text{V}$, and $W_r = 16$ bits.

The energy efficiency comparisons are shown in Table 39.1. As expected FFT implementation on a general-purpose processor [5] has low energy efficiency.

Table 39.1 Energy efficiency comparison of various normalized FFT implementations measured as the number of executed 1024-point FFTs with energy of 1 mJ.

Design	Tech.	Class	WL	V_{CC}	t_{clk}	t_{FFT}	Efficiency	
	[nm]		[bits]	[V]	[MHz]	[μs]	[FFT/mJ]	
[5]	65	GPP	16	1.2	1000	63	1	†
[41]	130	DSP	16	1.5	720	8	100	†
[37]	45	ASIC	32	0.9	650	2	1007	
[4]	180	ASIC	13	1.8	51	61	748	
[38]	180	ASIC	14	1.8	5	220	755	
[45]	65	ASP	16	1.2	150	6	633	†
[16]	130	ASP	16	1.5	320	14	1170	†
[1]	180	ASP	16	1.8	280	37	61	†
TTA [33]	130	ASP	16	1.5	250	21	809	

V_{CC} Supply voltage, *WL* Word length, t_{clk} Clock period, t_{FFT} FFT execution time, *GPP* General-purpose processor, *DSP* Digital signal processor, *ASIC* Application-specific integrated circuit, *ASP* Application-specific processor
† Energy does not include memories.

The Digital Signal Processor (DSP) in [41] can achieve high performance, but the energy efficiency in high-speed mode is lower than in low-power mode, i.e., lower frequency and supply voltage. In addition, the high performance calls for manually optimized assembly code. It should be noted that the energy figures exclude memories.

The application-specific processor in [1] contains user-specific function units, e.g., for address generation and butterfly computations. There are two complex-valued multipliers and three complex-valued adders. The twiddle factors are stored in the main memory. The pipeline architecture in [37] realizes data permutation with the aid of delay lines, where data traverses through the registers introducing high dynamic power consumption. It is not known if the twiddle factor memories and address generators are included in the energy figures. Another pipelined processor is proposed in [38], which uses block floating-point number representation with 10-bit mantissa and 4-bit shared exponent. The short floating-point word allows CORDIC pipeline to be shortened. If larger word lengths are needed, e.g., to support larger FFTs or to improve the signal-to-noise ratio, the pipeline depth needs to be increased, which increases the power consumption.

A cache-memory architecture is described in [4], where a small data cache is used to reduce accesses to the main memory. The processor uses 13-bit complex data type and supports FFT size up to 1024-points. In [45], a small cache memory is also used. The twiddle factors are stored in the main memory, which adds power consumption. Unfortunately, caches or memories are excluded from the energy figures.

The application-specific processor in [16] has two small caches to reduce access to the main memory, and these are accessed in ping-pong fashion to avoid stall cycles when transferring the results to the main memory. The processor uses $\left(\frac{N}{8}+1\right)$ complex-valued coefficients to compute the twiddle factors. External data memories are not included in the energy figures. In addition, the power consumption figures are coarse estimates obtained from a processor design tool.

39.8 Conclusions

In this chapter, we described transport-triggered architecture template, which can be used to develop application-specific processors. In addition, we introduced the TCE hardware/software codesign environment for developing tailored implementations based on TTA processors. The TCE provides tool support for iterative processor customization starting from high-level programming languages and contains retargetable compiler, which speeds up the iterative customization significantly. The tools produce synthesizable RTL description of the TTA processor and generates instruction parallel binary code. TCE is available as a liberally licensed open-source project and can be downloaded from the web page [40]. We also customized a TTA processor for FFT application and showed that the highly customized but still programmable processor possesses energy efficiency close to fixed-function ASIC implementations.

Acknowledgments The authors thank the Finnish Funding Agency for Innovation in the context of the FiDiPro project StreamPro (decision no. 40142/14).

References

1. Baek JH, Kim SD, Sunwoo MH (2008) SPOCS: application specific signal processor for OFDM communication systems. J Signal Process Syst 53(3):383–397. doi: 10.1007/s11265-008-0240-4
2. Chang WH, Nguyen TQ (2008) On the fixed-point accuracy analysis of FFT algorithms. IEEE Trans Signal Proc 56(10):4673–4682
3. Chang YN, Parhi KK (1999) Efficient FFT implementation using digit-serial arithmetic. In: Proceedings of IEEE international workshop signal processing system, Taipei, pp 645–653. doi: 10.1109/SIPS.1999.822371
4. Chen CM, Hung CC, Huang YH (2010) An energy-efficient partial FFT processor for the OFDMA communication system. IEEE Trans Circuits Syst II 57(2):136–140. doi: 10.1109/TCSII.2010.2040318
5. Cheng KT, Wang YC (2011) Using mobile GPU for general-purpose computing: a case study of face recognition on smartphones. In: Proceedings of international symposium VLSI design automation test, Hsinchu, pp 1–4. doi: 10.1109/VDAT.2011.5783575
6. Chi JC, Chen SG (2004) An efficient FFT twiddle factor generator. In: Proceeding of European signal processing conference, Vienna, pp 1533–1536
7. Chu E, George, A (2000) Inside the FFT black box: serial and parallel fast Fourier transform algorithms. CRC Press, Boca Raton
8. Cichon G, Robelly P, Seidel H, Matúš E, Bronzel M, Fettweis G (2004) Synchronous transfer architecture (STA). In: Computer systems: architectures, modeling, and simulation. Lecture notes in computer science, vol 3133. Springer, Berlin/Heidelberg, pp 193–207. doi: 10.1007/978-3-540-27776-7_36
9. Cooley JW, Tukey JW (1965) An algorithm for the machine calculation of complex Fourier series. Math Comput 19(90):297–301

10. Corporaal H (1997) Microprocessor architectures: from VLIW to TTA. Wiley, Chichester
11. Corporaal H, Mulder H (1991) MOVE: a framework for high-performance processor design. In: Proceedings of ACM/IEEE conference on supercomputing, Albuquerque, pp 692–701. doi: 10.1145/125826.126159
12. Dally W, Balfour J, Black-Shaffer D, Chen J, Harting R, Parikh V, Park J, Sheffield D (2008) Efficient embedded computing. Computer 41:27–32. doi: 10.1109/MC.2008.224
13. Despain AM (1974) Fourier transform computers using CORDIC iterations. IEEE Trans Comput C-23(10):993–1001. doi: 10.1109/T-C.1974.223800
14. Fanucci L, Roncella R, Saletti R (2001) A sine wave digital synthesizer based on a quadratic approximation. In: Proceedings of IEEE international frequency control symposium PDA exhibition, pp 806–810. doi: 10.1109/FREQ.2001.956385
15. Garrido M, Grajal J (2007) Efficient memoryless CORDIC for FFT computation. In: Proceedings of IEEE international conference acoustics speech signal processing, Honolulu, vol 2, pp 113–116. doi: 10.1109/ICASSP.2007.366185
16. Guan X, Fei Y, Lin H (2012) Hierarchical design of an application-specific instruction set processor for high-throughput and scalable FFT processing. IEEE Trans VLSI Syst 20(3): 551–563. doi: 10.1109/TVLSI.2011.2105512
17. Hasan M, Arslan T (2002) FFT coefficient memory reduction technique for OFDM applications. In: IEEE international conference acoustics speech signal process, Orlando, vol 1, pp 1085–1088
18. He Y, She D, Mesman B, Corporaal H (2011) MOVE-Pro: a low power and high code density TTA architecture. In: Proceedings of international conference on embedded computer system: architectures modeling simulation, pp 294–301. doi: 10.1109/SAMOS.2011.6045474
19. Heikkinen J, Takala J, Corporaal H (2009) Dictionary-based program compression on customizable processor architectures. Microprocess Microsyst 33(2):139–153. doi: 10.1016/j.micpro.2008.10.001
20. IEEE 802.16.1 (2012) IEEE standard for wireless MAN – advanced air interface for broadband wireless access systems. Std 802.16.1–2012. IEEE
21. Jääskeläinen P, Kultala H, Viitanen T, Takala J (2014) Code density and energy efficiency of exposed datapath architectures. J Signal Process Syst 1–16. doi: 10.1007/s11265-014-0924-x
22. Jääskeläinen P, de La Lama C, Huerta P, Takala J (2011) OpenCL-based design methodology for application-specific processors. Transactions on HiPEAC 5. Available online
23. Jääskeläinen P, de La Lama CS, Schnetter E, Raiskila K, Takala J, Berg H (2014) pocl: a performance-portable OpenCL implementation. Int J Parallel Prog 1–34. doi: 10.1007/s10766-014-0320-y
24. Jääskeläinen P, Salminen E, de La Lama C, Takala J, Ignacio Martinez J (2011) TCEMC: a co-design flow for application-specific multicores. In: Proceeding of international conference on embedded computer system: architectures modeling and simulations, Samos, pp 85–92. doi: 10.1109/SAMOS.2011.6045448
25. Jiang RM (2007) An area-efficient FFT architecture for OFDM digital video broadcasting. IEEE Trans Consum Electron 53(4):1322–1326. doi: 10.1109/TCE.2007.4429219
26. Johnson H, Burrus C (1984) An in-order, in-place radix-2 FFT. In: IEEE international conference on acoustics speech signal processing, vol 9, San Diego, pp 473–476. doi: 10.1109/ICASSP.1984.1172660
27. Johnsson SL, Krawitz RL, Frye R, MacDonald D (1989) A radix-2 FFT on connection machine. In: Proceeding of ACM/IEEE conference on supercomputing, Reno, pp 809–819. doi: 10.1145/76263.76355
28. Jui PC, Wey CL, Shiue MT (2013) Low-cost parallel FFT processors with conflict-free ROM-based twiddle factor generator for DVB-T2 applications. In: Proceedings of IEEE international midwest symposium circuits system, Columbus, pp 1003–1006
29. Lattner C, Adve V (2004) LLVM: a compilation framework for lifelong program analysis & transformation. In: Proceedings of the 2004 international symposium on code generation and optimization (CGO'04), Palo Alto

30. Ma Y, Wanhammar L (2000) A hardware efficient control of memory addressing for high-performance FFT processors. IEEE Trans Signal Process 48(3):917–921. doi: 10.1109/78.824693
31. Oppenheim AV, Schafer RW (2010) Discrete-time signal processing, 3rd edn. Pearson, Upper Saddle River
32. Pitkänen T, Partanen T, Takala J (2007) Low-power twiddle factor unit for FFT computation. In: Vassiliadis S, Berekovic M, Hämäläinen T (eds) Embedded computer systems: architectures, modeling, and simulation. Proceeding of 7th international workshop SAMOS VII, vol LNCS 4599. Springer, Berlin, pp 233–240. doi: 10.1007/978-3-540-73625-7_9
33. Pitkänen T, Takala J (2011) Low-power application-specific processor for FFT computations. J Signal Process Syst 63(1):165–176. doi: 10.1007/s11265-010-0528-z
34. Senthilvelan M, Sima M, Iancu D, Schulte M, Glossner J (2013) Instruction set extensions for matrix decompositions on software defined radio architectures. J Signal Process Syst 70:289–303. doi: 10.1007/s11265-012-0665-7
35. Singleton R (1967) A method for computing the fast Fourier transform with auxiliary memory and limited high-speed memory. IEEE Trans Audio Electroacoust 15(2):91–98
36. Strang G (1994) Wavelets. Am Sci 82(3):250–255
37. Suleiman A, Saleh H, Hussein A, Akopian D (2008) A family of scalable FFT architectures and an implementation of 1024-point radix-2 FFT for real-time communications. In: IEEE international conference on computer design, Lake Tahoe, pp 321–327. doi: 10.1109/ICCD.2008.4751880
38. Tang SN, Liao CH, Chang TY (2012) An area- and energy-efficient multimode FFT processor for WPAN/WLAN/WMAN systems. IEEE J Solid-State Circuits 47(6):1419–1435. doi: 10.1109/JSSC.2012.2187406
39. Tang Y, Qian L, Wang Y, Savaria Y (2003) A new memory reference reduction method for FFT implementation on DSP. In: Proceedings of ISCAS, Bangkok, vol 4, pp 496–499. doi: 10.1109/ISCAS.2003.1205932
40. TTA-based co-design environment (2015). http://tce.cs.tut.fi. Accessed: 15 Jan 2016
41. Texas Instruments, Inc. (2003) TMS320C64x DSP Library programmer's reference, Dallas
42. Thuresson M, Själander M, Björk M, Svensson L, Larsson-Edefors P, Stenström P (2007) FlexCore: utilizing exposed datapath control for efficient computing. In: Proceedings of international conference on embedded computer system: architectures modeling simulation, Samos, pp 18–25. doi: 10.1109/ICSAMOS.2007.4285729
43. Viitanen T, Kultala H, Jääskeläinen P, Takala J (2014) Heuristics for greedy transport triggered architecture interconnect exploration. In: Proceedings of international conference compilers architecture synthesis embedded system, New Delhi, pp 2:1–2:7. doi: 10.1145/2656106.2656123
44. Volder JE (1959) The CORDIC trigonometric computing technique. IRE Trans Electron Comput EC–8(3):330–334. doi: 10.1109/TEC.1959.5222693
45. Wang W, Li L, Zhang G, Liu D, Qiu J (2011) An application specific instruction set processor optimized for FFT. In: IEEE international midwest symposium circuits and systems, Seoul, pp 1–4. doi: 10.1109/MWSCAS.2011.6026391
46. Wanhammar L (1999) DSP integrated circuits. Academic Press, San Diego
47. Yu CY, Chen SG, Chih JC (2006) Efficient CORDIC designs for multi-mode OFDM FFT. In: Proceedings IEEE international conference acoustics speech signal processing, vol 3, Toulouse, pp III-1036–III-1039. doi: 10.1109/ICASSP.2006.1660834