FACHHOCHSCHULE HOCHSCHULE FÜR
STUTTGART TECHNIK

UNIVERSITY OF APPLIED SCIENCES

Department of Mathematics and
Computer Science

Winter Term 2001/2002

Implementation of a JPEG Decoder for a 16-bit Microcontroller

by

Stefan Kuhr

A Thesis Submitted

in Partial Fulfillment of the

Requirements for the Degree

MASTER OF SCIENCE

in

Software Technology

Thesis Committee:

Prof. Dr. Uwe Müßigmann, Chair
Prof. Dr. Peter Hauber
Rolf Kofink, Sony Corp.

For the best
of all mothers.

Contents

Acknowledgements

The author would like to thank Sony-Wega Corporation for the opportunity to work on this interesting topic as a master thesis. Special thanks go to all the fine folks of the VEE team of the Sony Advanced Technology Center Stuttgart, especially to the author's supervisor, Rolf Kofink, for all the fruitful discussions, to Andreas Beermann and Thor Opl for insightful comments, to Matthias Jerye for his outstanding technical expertise and cooperativeness, and to Mark Blaxall for reviewing this document over and over again, providing useful tips from a native English speaker.

Special thanks go to Thomas G. Lane from the Independent JPEG Group for reviewing chapter 5.

Finally, the author would like to thank all the helpful people in the German speaking TEX usenet newsgroup de.comp.text.tex for helping troubleshoot all kinds of weirdnesses involved with writing a document like this in LATEX.

Notational Conventions

The following notational conventions will be used throughout this document:

Symbol	**Meaning**
$F(u)$	One-dimensional Fourier Transform
$F(u, v)$	Two-dimensional Fourier Transform
$\tilde{F}(u)$	One-dimensional Cosine Transform
$\tilde{F}(u, v)$	Two-dimensional Cosine Transform
$\Re\{\ldots\}$	Real Part
$\Im\{\ldots\}$	Imaginary Part
j	$\sqrt{-1}$
I_n	$n \times n$ Identity Matrix

List of Acronyms

2D-DCT Two-dimensional Discrete Cosine Transform. The 2D-DCT is the two-dimensional variant of the DCT, see section 3.1.2.

2D-FDCT Two-dimensional Forward Discrete Cosine Transform. The 2D-FDCT is the two-dimensional variant of the FDCT, see section 3.1.2.

2D-IDCT Two-dimensional Inverse Discrete Cosine Transform. The 2D-IDCT is the two-dimensional variant of the IDCT, see section 3.1.2.

AC Alternating Current. In electricity, Alternating Current occurs when charge carriers in a conductor or semiconductor periodically reverse their direction of movement.

AC-3 Audio Code number 3. AC-3 is a multichannel music compression technology, also known as Dolby Digital, that has been developed by Dolby Laboratories.

ANSI American National Standards Institute. The American National Standards Institute is the primary organization for fostering the development of technology standards in the United States of America. ANSI works with industry groups and is the United States' member of the ISO and the IEC.

ASCII American Standard Code for Information Interchange. ASCII is the most common format for text files in computers and on the Internet. It was developed by the ANSI.

CCITT International Telegraph and Telephone Consultative Committee. The CCITT is an organ of the ITU.

CMYK Cyan, Magenta, Yellow, Black. CMYK is a so-called color space and thus refers to a system for representing colors. The CMYK color space is mainly used for printed color illustrations.

COM Component Object Model. COM is a framework and an architecture for creation, distribution and managing of distributed objects in a network, as specified by Microsoft.

cos-DFT Cosine DFT. The Cosine DFT is the definition of Vetterli and Nussbaumer for the real part of a DFT, see section 3.2.2.

CORBA Common Object Request Broker Architecture. CORBA is an architecture and specification for creation, distribution and managing of distributed objects in a network, as defined by the OMG (Object Management Group).

DC Direct Current. DC (Direct Current) is the unidirectional flow or movement of electric charge carriers.

DCT Discrete Cosine Transform. The Discrete Cosine Transform is an orthonormal transform that is suitable for image compression, see chapter 3.

DCT* Scaled variant of the FDCT. The DCT* is a scaled variant of the FDCT, see section 3.2.2.

DFT Discrete Fourier Transform. The Discrete Fourier Transform is an orthonormal transform that is widely used in system theory, see section 3.2.1.

DRAM Dynamic Random Access memory. Dynamic Random Access memory is the most common kind of random access memory (RAM) for personal computers.

EPROM Erasable programmable read-only memory. EPROM is programmable read-only memory that can be erased by exposing it to ultraviolet light in order to reprogram it.

FDCT Forward Discrete Cosine Transform. See the complete definition of the Forward Discrete Cosine Transform in 3.1.1.

GDI Graphics Device Interface. The GDI is a software library that was used throughout this thesis in conjunction with the microcontroller in use for the rendering of bitmaps on a monitor or TV set.

GIF Graphics Interchange Format. GIF is one of the most common file formats for graphic images on the World Wide Web.

GNU GNU is not Unix. The recursive acronym GNU stands for a UNIX-like operating system that comes with source code that can be copied, modified, and redistributed. Often the term GNU is also associated with "The GNU project" of the Free Software Foundation.

HDTV High Definition Television. HDTV is is a television format in 16:9 aspect ratio with at least twice the horizontal and vertical resolution of the standard format.

IDCT Inverse Discrete Cosine Transform. The Inverse Discrete Cosine Transform is the reverse process to the FDCT. See the complete definition of the Inverse Discrete Cosine Transform in 3.1.1.

IEC International Electrotechnical Commission. The IEC is a nongovernmental standards association that coordinates and unifies electrotechnical standards.

IJG Independent JPEG Group. An informal group that writes and distributes a widely used free library for JPEG image compression.

IRAM Internal dual-port RAM. The IRAM is on-chip memory of the controller used throughout this thesis.

ISO International Organization for Standardization. The ISO is a worldwide federation of national standards bodies."ISO" is actually not an abbreviation. It is a word, derived from the Greek isos, meaning "equal".

ITU International Telecommunication Union. The ITU is the United Nations' Specialized Agency in the field of telecommunications.

JFIF JPEG File Interchange Format. JFIF is the standard file format for exchanging JPEG images, see section 2.3.

JPEG Joint Photographic Expert Group. JPEG is the standard for continuous tone still images, see chapter 2.

MIPS Million instructions per second. The number of MIPS corresponds to the number of operations a computer can execute within a given time and thus serves as a general measure of computing performance.

MMX Multimedia Extensions. MMX is an extension to the instruction set of the Intel Pentium processor family for improved performance of multimedia applications.

MPEG Moving Picture Experts Group. The Moving Picture Experts Group is a working group in ISO that develops standards for digital video and audio compression.

OS Operating System. An Operating System is the program that is being initially loaded into a computer by a boot procedure and that manages all the other programs in the computer.

PAL Phase Alternation Line. PAL is an analog television display standard that is mainly used in Europe but also in other parts of the world.

ROM Read-only Memory. Read-only Memory is memory that cannot be written to.

RGB Red, Green, Blue. RGB is a so-called color space and thus refers to a system for representing colors. Pixels are represented as tupels of red, green and blue components.

sin-DFT Sine DFT. The Sine DFT is the definition of Vetterli and Nussbaumer for the imaginary part of a DFT, see section 3.2.2.

SVGA Super Video Graphics Array. SVGA is a display mode for computer monitors as specified by the Video Electronics Standards Assocation (VESA).

SMP Symmetrical Multiprocessing. In systems that employ Symmetrical Multiprocessing, more than one processor is used and a single copy of the operating system is in charge of all the processors. The operating system does not monopolize a single CPU, i.e. both operating system code and application code can run on all available processors.

UNICODE The Unicode Worldwide Character Standard. UNICODE is a format for binary coding text files in computers. UNICODE encompasses all the diverse languages of the modern world as well as many classical and historical languages.

UI User Interface. The User Interface is the part of a computer program that accepts input from the user and produces output for the user.

Win32 32-bit Windows. 32-bit Windows is the operating system environment that the Microsoft Windows operating systems provide to applications.

XRAM Internal XBUS RAM. The XRAM is on-chip memory of the controller used throughout this thesis.

YCCK Luminance, Chroma, Chroma, Black. YCCK is a so-called color space and thus refers to a system for representing colors. The YCCK color space is a rarely used color space defined by Adobe.

YCbCr Luminance and Chroma. YCbCr is a so-called color space and thus refers to a system for representing colors. Pixels are represented as tupels of one luminance (Y) and two chroma components (Cb and Cr). For a definition, see section 2.3.

List of Figures

List of Tables

Chapter 1

Introduction

> *All programmers are optimists. Perhaps this modern sorcery especially attracts those who believe in happy endings and fairy godmothers. Perhaps the hundred of nitty frustrations drive away all but those who habitually focus on the end goal. Perhaps it is merely that computers are young, programmers are younger, and the young are always optimists.*
>
> **– From the book "The mythical man-month" by Frederick P. Brooks jr., 1975**

The aim of this chapter is to provide the reader with the initial expectations prior to the beginning of this thesis, as well as with information regarding the organization of the rest of this document. For the hasty reader it is probably enough to read the following section, section 1.1, and the summary in chapter 7, to get an impression of what this thesis is all about. For all other readers, section 1.2 contains an overview of what the reader will expect throughout the rest of this document.

1.1 Subject and conceptual formulation of this thesis

This thesis is considered a feasibility study, that should examine whether a typical 16-bit microcontroller with a graphics accelerator unit (Micronas SDA 6000), as it is used for On-Screen-Displays in High-End TV sets, is capable of decoding JPEG files of commonly used file sizes and image resolutions. Of particular interest are those resolutions that are common for digital still cameras. In order to estimate the feasibility, which first of all includes acceptable performance, a JPEG decoder has to be implemented on the microcontroller.

The following issues have been identified beforehand as potential obstacles for this goal:

- Limited memory resources of the controller: The controller can use a maximum of 8 MBytes of DRAM. While this seems to be enough for JPEG decoding at first glance, it has to be considered that an embedded device such as a microcontroller doesn't have virtual memory like PCs or workstations due to the lack of a harddisk. Also, typical

JPEG images from digital still cameras at the time of writing are "3.3 Megapixel" images with a resolution of 2048×1536 pixels. One image of this size with 1 byte per color component clearly exceeds the available memory space of 8 MBytes.

- Processor speed: The microcontroller runs at a clock rate of 33 MHz. Typical machine instructions need two clock cycles, so the controller can be roughly considered a 16.5 MIPS machine. Typical desktop computers today have several hundred MIPS of processing power and optimized architectures for floating point arithmetics or instruction sets that are more suitable for the fast calculation of the DCT (Discrete Cosine Transform), like the MMX technology from Intel.
- Display functionality: With standard software packages, the microcontroller's graphics accelerator unit is only capable of rendering RGB tupels with a color depth of 4 bits per color component on the output device (4-4-4 mode). Though the graphics accelerator unit has a mode (5-6-5 mode)in which it can display RGB tupels with 5, 6 and 5 bits for R, G and B, respectively, there is no software available that exploits this mode.

The following tasks are considered prerequisites for a successful implementation of a JPEG decoder:

- Availability of suitable software: Part of the work is considered market research for suitable software packages that could be ported to the microcontroller. The choice of programming languages is limited to C or C++ (via a C++ frontend compiler).
- Familiarity with the development environment for the microcontroller: Mastering the "tool chain" of C/C++ compiler, assembler, linker/locator and debugger is a prerequisite for writing any software on any platform, but is especially true for the development for an embedded system with a cross-compiler and a remote debugger, where also the quality of the development tools is generally not as high as for the PC and workstation market.

The following issues are considered additional development goals:

- It should be taken into account, that the TV set used to render a decoded JPEG file typically has only a very limited spatial resolution. This limitation is imposed by the capabilities of the microcontroller (800×600 pixels maximum). Therefore, the ability to scale an image fast enough, either during decoding or after decoding, in order to fit the screen, is a requirement.
- It should be possible to identify and fix possible performance bottlenecks in the software being used for JPEG decoding. Therefore intimate knowledge of all the steps involved in JPEG decoding, especially the IDCT (Inverse Discrete Cosine Transform), is a requirement.

The minimum result of this thesis should be to make a statement, as to whether or not the microcontroller is capable of doing JPEG decoding. If it is not, the reasons for this should be given. If it is capable for doing so, the limitations for this should be determined. Also, the performance should be determined and increased, and it should be at least possible to decode images of standard VGA resolution (640×480 pixels). If performance is unacceptable, it should be possible to predict the processing power that is required for the next generations of microcontrollers to deliver acceptable performance.

1.2 Overview of the subsequent chapters

The next chapters are organized as follows: Chapter 2 will contain a brief introduction to JPEG, including some historical retrospect and an overview of those parts of this international standard that are actually in use at the time of writing. Chapter 3 will dive into the heart of JPEG compression, the discrete cosine transform. Several historical algorithms will be presented, including the fastest one-dimensional algorithms known up to now, and a fast two-dimensional algorithm that is based on the fastest scaled one-dimensional algorithm. Chapter 4 will deal with fast image scaling, either in the spatial domain, after decoding a JPEG image, or during the decoding process. Chapter 5 will give an overview of the most popular software package being used for JPEG encoding and decoding, the Independent JPEG Group's JPEGLib. Chapter 6 will include information about the source code written during this thesis along with detailed numbers on the performance of the Micronas SDA 6000 microcontroller for JPEG decoding. Chapter 7 will then finally conclude this document with a summary and recommendations for future work.

Chapter 2

A brief introduction to JPEG encoding and decoding

Not for nothing does it say in the Commandments "Thou shalt not make unto thee any image" ... Every image is a sin ... When you love someone you leave every possibility open to them, and in spite of all the memories of the past you are ready to be surprised, again and again surprised, at how different they are, how various, not a finished image.

– From the novel "Stiller" by Max Frisch, 1954

THE aim of this chapter is to provide an overview of JPEG encoding and decoding without going too far into the technical details themselves. The interested reader may consult the various references for more in-depth details on the actual file format or other subtle details of the standard. First we will give some historical overview and will make clear, that the JPEG standard is not, at least in its initial design, a simple file format specification, but rather an architecture for a set of image compression functions with a rich set of capabilities, making it suitable for a wide range of applications that use image compression. We will give an overview of how the encoding and decoding processes actually work and what modes of operation are possible, which ones are actually in use at the time of writing, and which ones are the dominant ones. After that, the definition of the baseline system and the JFIF file format, which is the prevalent JPEG file format in use today will be given. Finally, we will conclude this chapter and show, where actually compression and data loss comes into play in JPEG encoding.

2.1 History and Motivation

In the early eighties of the 20^{th} century, researchers interested in color image data compression initiated some activity in ISO (*International Organization for Standardization*) on a standard in the area of color image data compression. At about the same time, several working groups of the CCITT (*International Telegraph and Telephone Consultative*

Committee), an organ of the ITU (*International Telecommunication Union*), the United Nations' Specialized Agency in the field of telecommunications, were driven by the same goal. In order to avoid the definition of two different competing standards, the groups in CCITT joined the working groups in ISO in 1986. In 1987, ISO and the IEC (*International Electrotechnical Commission*) created the Joint Technical Committee 1 (JTC1) in order to standardize on the field of information technology, and one of these collaborations under JTC1 was the now called JPEG committee (pronounce: "Jay-Peg")[1]. JPEG stands for "Joint Photographic Experts Group" and the term "Joint" reflects the fact, that this standard was a joint development of the three aforementioned standards bodies.

Finally, in 1992, the JPEG's standard document ([5]) with the title "Information Technology - Digital Compression and coding of continuous-tone still images - requirements and guidelines" was approved as ISO International Standard 10918-1 (ISO IS 10918-1) and as CCITT Recommendation T.81. To quote from its introduction, this standards document

> "... sets out requirements and implementation guidelines for continuous-tone still image encoding and decoding processes, and for the coded representation of compressed image data for interchange between applications. These processes and representations are intended to be generic, that is, to be applicable to a broad range of applications for color and grayscale still images within communications and computer systems. [...] In addition to the applications addressed by the CCITT and ISO/IEC, the JPEG committee has developped a compression standard to meet the needs of other applications as well, including desktop publishing, graphic arts, medical imaging and scientific imaging." [5].

2.2 The JPEG Standard

The JPEG standards document specifies three elements: an encoder, a decoder and an interchange format. The encoder takes digital source image data and table specifications as input, and generates compressed image data via a specified set of procedures. The decoder takes compressed image data and table specifications as input, and generates digitally reconstructed image data via another specified set of procedures. The interchange format is a compressed image data representation that includes all table specifications used in the encoding and decoding process. The interchange format is for exchange between different application environments, e.g. two different computers, computer applications or telecommunication devices. Both the encoder and the decoder can be implemented in hardware or software. The interchange format can be data that is transmitted via communications facilities or a file in a computer system. Also, the interchange format does not specify a complete coded image representation, e.g. application-dependent information such as color space is outside the scope of the JPEG specification. This means, that JPEG as such is a "colorblind" specification (see also section 2.3).

In the following sections we will give a rough introduction into some of the JPEG coding and decoding techniques, based on the standards document ([5]) and the textbook-like approach of [22], whose authors were members of the JPEG working group and wrote substantial parts of the standards document during the standardization process. By no

[1]The reason why three standards bodies bother about standardization on a single area, like in the case of color image data compression, lies in the fact that this area is both part of the telecommunications technology domain (CCITT) and the computer technology domain (ISO and IEC).

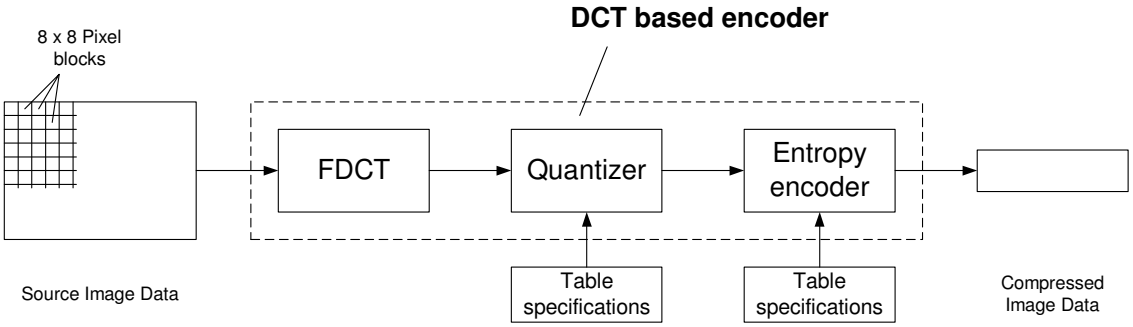

Figure 2.1: Simplified diagram of a DCT-based encoder

means should the following be considered a detailed description of the JPEG standard or the file formats in use today; rather it should give the reader a rough overview of the processes involved in JPEG encoding and decoding. For a detailed description of these issues, the reader may consult [5] and [22].

2.2.1 Compression classes

The JPEG specification specifies two fundamental classes of encoding and decoding processes: lossy and lossless processes. The processes that are based on the discrete cosine transform (see chapter 3) are lossy, and thereby allow for substantial compression while producing a reconstructed image that shows high visual fidelity to the encoder's original source image. In order to meet the needs of applications requiring lossless compression, the JPEG specification provides the second class of coding processes which is not based on the DCT. For the DCT-based processes, two alternative sample precisions are specified in the standards document: either 8 bits or 12 bits per sample. 12 bits per sample are only in use in specialized applications such as medical imaging. For lossless processes the sample precision is specified to be from 2 to 16 bits.

In the following, we will disregard the lossless JPEG compression class, because it is not in widespread use[2].

2.2.2 DCT-based Encoding

Figure 2.1 shows the encoding process for the case of an image with only one color component (i.e. a grayscale image) in a simplified form. In the case of more than one color component (the number of allowed color components per image in the JPEG specification is virtually unlimited) each color component is treated in the same manner, independently of the other color components.

In the encoding process the input component's samples are grouped into 8×8 pixel blocks, and each individual block is transformed by the forward DCT (FDCT) into a set of 64 values referred to as the "DCT coefficients" (for an in-depth introduction into the FDCT, see chapter 3). The upper left value of these values is commonly referred to as the

[2]Ironically, for lossless compression in digital still cameras, TIFF Rev. 6.0 was adopted in [14] instead of the lossless JPEG compression class, whereas for lossy compression the JPEG baseline system (see section 2.2.4) was adopted.

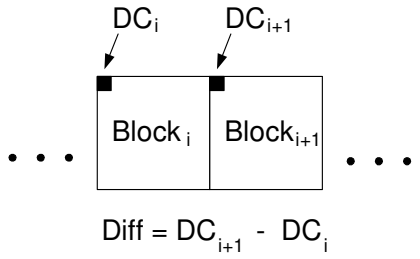

Figure 2.2: Differential encoding of the DC values of two subsequent 8×8 blocks

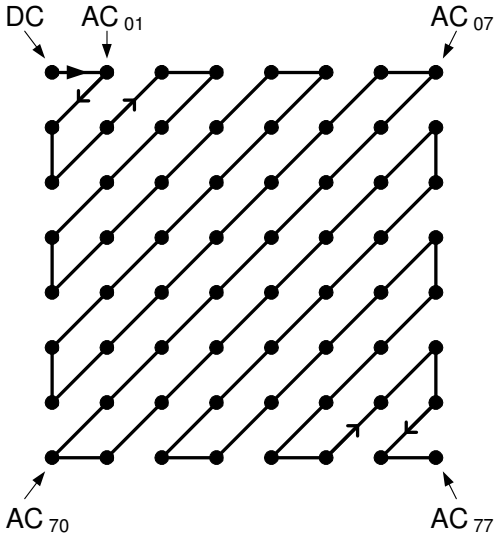

Figure 2.3: Zig-Zag encoding of AC coefficients within one 8×8 block

"DC coefficient" or "DC value" (see figure 2.3), because it contains the average value of all pixels prior to the FDCT step. The other 63 values are commonly referred to as the "AC coefficients" or "AC values". Each of these 64 coefficients is then divided ("quantized") by one of 64 corresponding values from a quantization table. There are no default values for quantization tables specified in the JPEG specification[3]; applications or their users may specify values in order to customize image quality for their particular viewing conditions, display devices or preferred image characteristics.

After quantization, the DC coefficient and the 63 AC coefficients are prepared for entropy encoding, as shown in figures 2.2 and 2.3: Figure 2.2 exemplifies how the current quantized DC coefficient is used to predict the subsequent quantized DC coefficient in that only the difference between the two is encoded. The underlying heuristics here are, that

[3]However, the JPEG standard ([5]) gives example tables whose properties are described as follows: "These are based on psychovisual thresholding and are derived empirically using luminance and chrominance and 2:1 horizontal subsampling. These tables are provided as examples only and are not necessarily suitable for any particular application."

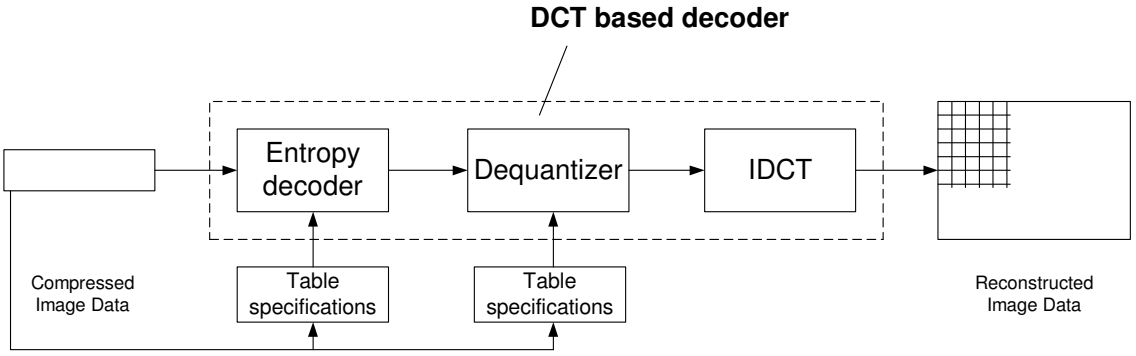

Figure 2.4: Simplified diagram of a DCT-based decoder

the DC values of two subsequent 8×8 blocks will not differ too much and that therefore their difference can be efficiently encoded with an entropy encoding scheme. The 63 quantized AC coefficients do not undergo such a differential encoding technique, but are rather converted into a one-dimensional zig-zag sequence, as shown in figure 2.3. After that, the quantized coefficients in zig-zag sequence are passed to an entropy encoding procedure which compresses the data further. One of two entropy coding procedures can be used, Huffman coding or arithmetic coding. For Huffman encoding, the Huffman table specifications must be provided to the encoder. For each color component, there must be one Huffman table for encoding the DC coeffients and one Huffman table for encoding the AC coefficients, although two color components may share a pair of Huffman tables, as is often the case for the chroma components (the Cb and Cr components in the YCbCr color space[4]). In the following we will disregard arithmetic coding, since it is not in widespread use[5], although it is generally considered to be slightly more efficient than Huffman coding. Huffman encoding is an entropy encoding procedure that assigns a variable length code to each input symbol. In Huffman encoding, symbols are assigned a frequency that determines their position in a binary tree: Rarely used symbols are at the bottom of the tree whereas frequently used symbols are nearer to the tree's root. A symbol can now be expressed by its Huffman code, which is the path from the root to its location in the tree. Frequently used symbols are assigned shorter Huffman codes whereas rarely used symbols are expressed by longer Huffman codes, resulting in overall compression. In JPEG, the Huffman code length (the depth of the tree) is restricted to 16 bits and the Huffman tables consist of 16 values that correspond to the number of counts of Huffman codes for the associated code length and a list of symbol values that are sorted by Huffman code (see [5]) in order of increasing code length. From this information, the binary tree can be easily reconstructed, but the JPEG standard also provides algorithms that can reconstruct the symbols in the input stream of the Huffman decoder directly from this information (see [5]).

Figure 2.4 now shows the decoding process in a simplified form. Before the start of the

[4] In practical usage of JPEG, the preferred color space is not RGB, but rather YCbCr, see also section 2.3

[5] Arithmetic coding is also subject to a patent issue, which caused application developers – or more specific: the implementors of the Independent JPEG Group's JPEGLib (see chapter 5) – to refrain from implementing arithmetic coding.

actual image in the decoder's stream of input data, the encoder has placed the tables it used for encoding the image[6]. This means that the decoder can first extract the required tables from its input data stream, in order to process the rest of the input data with the help of these tables. The first step is to reverse the entropy encoding process, which produces the quantized DCT coefficients. With the help of the quantization tables, the DCT coefficients can be dequantized and finally be transformed back via the IDCT process (for an in-depth introduction into the IDCT, see chapter 3).

2.2.3 Modes of operation

The JPEG standard distinguishes four modes of operation under which the various coding processes are defined: sequential DCT-based, progressive DCT-based, lossless, and hierarchical.

Sequential DCT based mode is by far the most popular mode, almost all JPEG files found on the internet today use this mode of operation. "Sequential" means, that the image can be decoded line by line with a minimum of memory requirements during the decoding process and a line of the image can be rendered on the output device as soon as it is fully decoded. In order to minimize memory requirements during decoding for images with more than one color component, the sequential mode encoding process may also interleave pixels from two or more color components in the input data stream. This is particularly important if during the decoding process a color space conversion such as a conversion from YCbCr to RGB data is required. If scans are not interleaved, complete lines of each component must be decoded before a complete line in the target color space can be calculated. If scans are interleaved, this color space conversion can be done "on the fly" on a per-pixel basis during line processing. For more information on interleaving of scans, the reader may consult the standard ([5]).

The progressive DCT-based mode is probably the second-most popular mode of operation. "Progressive" means, that the components are encoded in multiple scans. The first scan contains a rough version of the image and subsequent scans refine the image. This way, the user can get a rough idea of an image at a very early state, while it is transmitted via a low bandwidth connection such as a modem to a web browser or the like and, if desired, can cancel further downloading of the rest of the image[7]. Web users typically experience a similar effect from so-called "interlaced GIF" files which are far more popular than progressive mode JPEG files. The drawback of the progressive DCT-based mode are the memory requirements: A memory buffer for the complete image must be provided during the whole decoding process, whereas for the sequential DCT based mode only a line buffer is required.

Similar to the lossless mode, the hierarchical mode is not in general use. We will therefore disregard it in the following with a quote from the JPEG standard ([5]), describing its purpose:

> "... In hierarchical mode, an image is encoded as a sequence of frames. These frames provide reference reconstructed components which are usually needed

[6] This could be considered as some sort of a "file header".

[7] Unfortunately, the only web browser at the time of writing which really supports progressive display of JPEG files in progressive DCT-based mode is the Netscape 4.x browser suite, which is not the predominant browser anymore. Later Versions of the Netscape browser suite based on the Mozilla project, and the Internet Explorer versions from Microsoft can display JPEG files in progressive DCT-based mode, but do not display the state after intermediate scans but rather the final output only.

> for prediction in subsequent frames. Except for the first frame for a given component, differential frames encode the difference between source components and reference reconstructed components. The coding of the differences may be done using only DCT-based processes, only lossless processes, or DCT-based processes with a final lossless process for each component. Downsampling and upsampling filters may be used to provide a pyramid of spatial resolutions [...]. Alternatively, the hierarchical mode can be used to improve the quality of the reconstructed components at a given spatial resolution. Hierarchical mode offers a progressive presentation similar to the progressive DCT-based mode but is useful in environments which have multi-resolution requirements. Hierarchical mode also offers the capability of progressive coding to a final lossless stage."

2.2.4 The baseline process

The "baseline process" provides a capability which is sufficient for many applications and is the simplest form of a DCT-based JPEG decoder. It is also a requirement for all DCT-based decoders to be capable of the baseline process. This also means, that encoding for the baseline process guarantees, that a particular image can be decoded by every application that decodes JPEG data. The following are the requirements of the baseline process:

- DCT-based process
- 8-bit samples within each component of the source image
- Sequential mode of operation
- Maximum of 2 AC and 2 DC tables for Huffman coding in total for all color components
- Maximum of 4 color components
- Interleaved and non-interleaved scans possible

Any DCT-based JPEG decoder, that provides additional capabilities to the baseline process is a decoder that uses an "extended (DCT-based) process"[8].

2.3 The JPEG File Interchange Format (JFIF)

In section 2.2 we already briefly mentioned that JPEG is a "colorblind" standard, which means that the number of color components and the choice of the color space is left up to JPEG applications or application developers. In practice however, it is desirable to have a color space with 3 components (or 4 in the case of CMYK) that can be transformed into the RGB color space. A good adaption to the properties of the human visual system is the YCbCr color space, that separates luminosity information Y (*luminance*) from the color information Cb and Cr (*chroma*). From the standpoint of data compression, it is very important to note that the contrast sensitivity of the human visual system for luminance (rod vision) is much higher than for the chroma components (cone vision) because different

[8] The standard defines "extended (DCT-based) process" as "a descriptive term for DCT-based encoding and decoding processes in which additional capabilities are added to the baseline sequential process" [5].

visual receptors are used to perceive luminance (rods) and chroma information (cones). In other words: Luminance information is much more important, so chroma information can be suppressed with no perceivable quality loss. This is called *chroma subsampling* and typically works such that in one line only one chroma pixel is used per 2 luminance pixels (4:2:2 chroma subsampling) or even one chroma pixel per 2 luminance pixels in both horizontal and vertical direction (4:2:0 chroma subsampling)[9]. The JPEG standard provides mechanisms for such subsampling of individual components and thus suggests the use of the YCbCr color space. The following set of equations ([11]) can be used to accomplish a color space conversion from RGB to YCbCr:

$$\begin{aligned} \mathrm{Y} &= 0.299 \cdot \mathrm{R} + 0.587 \cdot \mathrm{G} + 0.114 \cdot \mathrm{B} \\ \mathrm{Cb} &= -0.1687 \cdot \mathrm{R} - 0.3313 \cdot \mathrm{G} + 0.5 \cdot \mathrm{B} + 128 \\ \mathrm{Cr} &= 0.5 \cdot \mathrm{R} - 0.4187 \cdot \mathrm{G} - 0.0813 \cdot \mathrm{B} + 128 \end{aligned} \tag{2.1}$$

The reverse process, a color space conversion from YCbCr to RGB can be made with the following equations ([11]):

$$\begin{aligned} \mathrm{R} &= \mathrm{Y} + 1.402 \cdot (\mathrm{Cr} - 128) \\ \mathrm{G} &= \mathrm{Y} - 0.34414 \cdot (\mathrm{Cb} - 128) - 0.71414 \cdot (\mathrm{Cr} - 128) \\ \mathrm{B} &= \mathrm{Y} + 1.772 \cdot (\mathrm{Cb} - 128) \end{aligned} \tag{2.2}$$

In 1992, Eric Hamilton of C-Cube Microsystems, a member of the JPEG committee and chair of the editing group of part 2 of the JPEG standard[10] (ISO IS 10918-2), published a paper ([11]) of 9 pages that described a JPEG compatible file format that he named "JPEG File Interchange Format" (JFIF), with the following features:

- PC or Mac or Unix workstation compatible
- Standard color space: one or three components. For three components, YCbCr is used, for one component only Y is used
- Extensions in private fields ("APP_0 markers") to identify the file as a JFIF file
- Extensions in private fields to encode pixel density (aspect ratio)
- Extensions in private fields to encode pixel units (no units, dots per inch or dots per cm)
- Extensions in private fields for thumbnail data[11]

All extensions to the already defined JPEG format for the JFIF format are made in so-called APP_0 markers that may contain application specific data, therefore JFIF files completely adhere to the JPEG standard. Throughout the format specification of JPEG, the term

[9] The case where no subsampling is used, i.e. one chroma pixel per luminance pixel, is called 4:4:4 chroma subsampling.

[10] All ISO standards require compliance tests. The purpose of part 2 of the JPEG standard document is to provide such compliance tests for JPEG.

[11] A thumbnail is a low resolution version of the complete image that can be extracted very fast. This way applications can give the user a quick overview over all JPEG files in a directory without actually having to completely decode all images for this purpose.

"marker" is used for a byte sequence starting with FF_{16}. The byte following FF_{16} then specifies the marker type. Markers with data of arbitrary length are followed by two bytes containing the length of the marker, including the two length bytes but not the two bytes of the marker iself. The JFIF format demands that the starting SOI ("Start Of Image") marker of a JPEG file is directly followed by an APP_0 marker, its two length bytes and the zero terminated ASCII string "JFIF". The SOI marker is the byte sequence $FFD8_{16}$ and the APP_0 marker is made up of the byte sequence $FFE0_{16}$, the string "JFIF" corresponds to the byte sequence $4A46494600_{16}$. This way, a JFIF file can be identified with high probability by examining the first 11 bytes of the file in question: If the first four bytes are the sequence $FFD8FFE0_{16}$ and after skipping the next two bytes the zero-terminated ASCII string "JFIF" or the byte sequence $4A46494600_{16}$ is found, the file is very likely to be a JFIF file.

The definition of the JFIF file format was widely adopted by application developers, and today virtually all JPEG files in use are actually JFIF files. Only very few applications, such as Adobe Illustrator, allow the creation of JPEG files that are not JFIF files[12].

2.4 Compression and information loss in JPEG encoding

The astute reader might have wondered, where in the encoding or decoding processes discussed so far, any compression or data loss can be achieved at all besides from the chroma subsampling mentioned in section 2.3. Maybe some reader also wondered, what these "quantization tables" are all about.

The truth is, that through the FDCT and IDCT steps, no compression is achieved at all. Apart from rounding errors, resulting from floating-point or fixed-point arithmetics, these steps don't even introduce any information loss. The key to JPEG compression is the combination of the FDCT and the proper selection of quantization tables. The FDCT process has the property of concentrating the most important parts of an 8×8 block of pixels, with regard to the human visual system, in the upper- and leftmost corner of the 64 DCT coefficients, just around the DC-coefficient. By carefully selecting the quantization tables, these important values around the DC coefficient can be preserved, whereas the other DCT coefficients are quantized to zeroes. This typically leads to the situation, that the zig-zag encoded blocks contain some useful non-zero values right at the start of an 8×8 pixel block, followed by long runs of zero values with a few interspersed non-zero values inbetween. Entropy encoding can now minimize the storage requirements for this block in that it accounts for these runs of identical values, which makes up the compression effect in JPEG encoding. Information loss is solely introduced by non-uniform quantization table values, varying these tables is the means of getting a user-specified compression vs. image quality tradeoff. If all quantization table values are 1, JPEG encoding and decoding comprises no information loss, apart from rounding errors. In theory, the JPEG standard allows the usage of arbitrary quantization tables, but application programs for JPEG encoding typically allow the user to specify the desired compression vs. image quality tradeoff on a higher level of abstraction than specifying these 64 quantization values per color component

[12]Probably, the Independent JPEG Group's (IJG) JPEGLib (see chapter 5) also played an important role in the widespread adoption of the JFIF format, since this is the builtin format of this library. Another boost probably came from the success of the World Wide Web and the fact that the major browser vendors used the IJG's code in their browsers for decoding JPEG images.

Figure 2.5: Original picture at size 480000 bytes

individually. In most cases the user only specifies some "quality percentage" value, and the application program determines suitable quantization tables from this.

As an example of how the selection of the quantization table affects image quality and compression, we will look at a greyscale image[13] and various compressed variants of it in the JFIF format (all with 4:4:4 chroma subsampling). Figure 2.5 shows the original file with a resolution of 800×600 pixels. Figures 2.6, 2.7, 2.8 and 2.9 show a JPEG-compressed version at various file sizes (54680 bytes, 35336 bytes, 18573 bytes and 10807 bytes, respectively).

Figure 2.6 uses the following quantization table:

8	6	6	7	6	5	8	7
7	7	9	9	8	10	12	20
13	12	11	11	12	25	18	19
15	20	29	26	31	30	29	26
28	28	32	36	46	39	32	34
44	35	28	28	40	55	41	44
48	49	52	52	52	31	39	57
61	56	50	60	46	51	52	50

A comparison with the original file in 2.5 shows no real apparent difference while this file has only about 11 % of the original file's size. This quantization table already favours the values around the DC coefficients, the highest value can be found in the last row (61). The lowest values are in the upper left corner.

[13]This image is provided courtesy of Arwed Sienitzki, Sony-Wega Corporation.

Figure 2.6: JPEG file at size 54680 bytes

Figure 2.7: JPEG file at size 35336 bytes

Figure 2.8: JPEG file at size 18573 bytes

Figure 2.9: JPEG file at size 10807 bytes

Figure 2.7 uses the following quantization table:

16	11	12	14	12	10	16	14
13	14	18	17	16	19	24	40
26	24	22	22	24	49	35	37
29	40	58	51	61	60	57	51
56	55	64	72	92	78	64	68
87	69	55	56	80	109	81	87
95	98	103	104	103	62	77	113
121	112	100	120	92	101	103	99

Still, only few artifacts can be found in this image, while it has only about 7 % of the original file's size.

Figure 2.8 uses the following quantization table:

40	28	30	35	30	25	40	35
33	35	45	43	40	48	60	100
65	60	55	55	60	123	88	93
73	100	145	128	153	150	143	128
140	138	160	180	230	195	160	170
218	173	138	140	200	255	203	218
238	245	255	255	255	155	193	255
255	255	250	255	230	253	255	248

In this quantization table already lots of table entries have the maximum value of 255, the lowest values are again in the upper left corner. Quite some artifacts can now be seen in this image, that has about 4 % of the original file's size.

Figure 2.9 uses the following quantization table:

80	55	60	70	60	50	80	70
65	70	90	85	80	95	120	200
130	120	110	110	120	245	175	185
145	200	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255
255	255	255	255	255	255	255	255

In this quantization table only the values in the upper left corner don't have the maximum value of 255. This image is clearly of unæsthetic quality but occupies only about 2 % of the original file's size. In this image also the drawback of grouping pixels into 8×8 blocks can be seen in that so called "blocking artifacts" appear and the grouping into these blocks becomes immediately visible to the viewer.

Chapter 3

The Discrete Cosine Transform

And when God gave out rhythm,
he sure was good to you.
You can add, subtract, multiply and divide ... by two.

– From the song "Popsicle Toes" by Michael Franks, 1976

The Discrete Cosine Transform (DCT) is the transformation that is at the heart of JPEG compression and decompression. It was proposed in 1972 to the American National Science Foundation by Nasir Ahmed as an algorithm to achieve bandwidth compression. According to Ahmed, the proposal was not funded because the reviewers found "the whole idea seemed too simple" [1]. Ahmed continued to work on the DCT together with his Ph.D. student Raj Natarajan and Ram Mohan Rao and published it in a paper ([20]) in 1974.

In applications of the DCT, such as JPEG encoding and decoding, the bandwidth compression is achieved by transforming the digital image via the DCT into the *DCT Frequency Domain* where those portions of the image that are less important or imperceptible for the human eye appear clearly separated from the perceptible features of the image and thus can be removed. But the DCT is not only relevant for still images like in JPEG, it is also an integral part of moving picture standards like MPEG-1 and MPEG-2.

The DCT is also of particular importance for other compression standards, such as Dolby AC-3 which is used as the audio compression standard for HDTV, where a modified version of the DCT has been adopted.

This chapter deals with the mathematical definition and gives an overview over several fast DCT implementations. It also contains a comparison of the DCT and the Discrete Fourier Transform (DFT) that shows that the DCT is a special case of the DFT for a real and symmetrical input vector and how a DFT on a real input vector can be split into several DFTs and DCTs and vice versa, following a Divide-and-Conquer scheme.

3.1 Mathematical Definition of the DCT

3.1.1 The one-dimensional DCT

The process of transforming a one-dimensional set of N real valued samples into the DCT Frequency Domain is called the *forward discrete cosine transform* (FDCT) and creates a set

with the same number of real valued samples which are often called the *DCT coefficients*. The reverse process that recreates the original image from the DCT coefficients is called the *inverse discrete cosine transform* (IDCT). The definitions of the N-point FDCT and IDCT are as follows ([20]):

$$\text{FDCT:}\quad \tilde{F}(u) = \frac{2}{N}\ c(u) \sum_{n=0}^{N-1} f(n) \cos \frac{(2n+1)\,u\pi}{2N}, \quad u = 0, \ldots, N-1 \tag{3.1}$$

$$\text{IDCT:}\quad f(n) = \sum_{u=0}^{N-1} c(u)\tilde{F}(u) \cos \frac{(2n+1)\,u\pi}{2N}, \quad n = 0, \ldots, N-1 \tag{3.2}$$

where $n, u \in \mathbb{N}$ and

$$\begin{array}{lll} c(u) & = \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ c(u) & = 1 & \text{for } u > 0 \\ f(n) & = & \text{1-D sample value} \\ \tilde{F}(u) & = & \text{1-D DCT coefficient} \end{array} \tag{3.3}$$

In the following, we will use the definition from ([23]) and ([22]) where the normalization constant 2/N of the FDCT is equally distributed over the FDCT and IDCT:

$$\text{FDCT:}\quad \tilde{F}(u) = \sqrt{\frac{2}{N}}\ c(u) \sum_{n=0}^{N-1} f(n) \cos \frac{(2n+1)\,u\pi}{2N}, \quad u = 0, \ldots, N-1 \tag{3.4}$$

$$\text{IDCT:}\quad f(n) = \sqrt{\frac{2}{N}}\ \sum_{u=0}^{N-1} c(u)\tilde{F}(u) \cos \frac{(2n+1)\,u\pi}{2N}, \quad n = 0, \ldots, N-1 \tag{3.5}$$

For JPEG encoding and decoding, the number of samples is always $N = 8$, so equations (3.4) and (3.5) become equations (3.6) and (3.7):

$$\text{FDCT:}\quad \tilde{F}(u) = \frac{c(u)}{2} \sum_{n=0}^{7} f(n) \cos \frac{(2n+1)\,u\pi}{16}, \quad u = 0, \ldots, 7 \tag{3.6}$$

$$\text{IDCT:}\quad f(n) = \sum_{u=0}^{7} \frac{c(u)}{2}\tilde{F}(u) \cos \frac{(2n+1)\,u\pi}{16}, \quad n = 0, \ldots, 7 \tag{3.7}$$

The FDCT can be exemplified as a decomposition of an input sample vector into a scaled set of cosine base function vectors with the DCT coefficients as the scaling values. Summing up the scaled cosine base functions vectors then corresponds to the IDCT process. The DCT as defined in equation (3.4) is an *orthonormal transform* (see [23]), which simply means, that the inner product of any two different cosine base function vectors is always zero[1] and

[1] Two vector's inner product is 0 if the vectors are orthogonal, therefore the *ortho-* in *orthonormal transform*

that the inner product of a cosine base function vector with itself is always one[2]. This also means, that no cosine base function can be represented by a scaled sum of different cosine base functions and that all possible input vectors can be represented by a sum of scaled cosine base function vectors with the scaling values (DCT coefficients) being unique for each input vector. In this respect, the Cosine Transform is very similar to the Fourier Transform while having the advantage of being a real transform. The obvious similarities and relations between the DCT and the Discrete Fourier Transform are covered in chapter 3.2. A nice and intuitive introduction into the DCT and its similarities with the Fourier Transform can also be found in [2].

3.1.2 The two-dimensional DCT

If the DCT is to be applied to two-dimensional image arrays, the one-dimensional DCT is extended to the two-dimensional DCT (2D-DCT). Again, the two-dimensional FDCT (2D FDCT) transforms the image samples into the DCT Frequency Domain and the two-dimensional IDCT (2D IDCT) is the reverse operation that recreates the original sample values. The definitions for JPEG ([5], [33]) and other standards that employ an 8-point DCT are as follows:

2D FDCT:

$$\tilde{F}(u,v) = \frac{c(u)}{2}\frac{c(v)}{2}\sum_{n=0}^{7}\sum_{m=0}^{7} f(n,m)\cos\frac{(2n+1)\,u\pi}{16}\cos\frac{(2m+1)\,v\pi}{16}, \quad u,v = 0,\ldots,7 \tag{3.8}$$

2D IDCT:

$$f(n,m) = \sum_{u=0}^{7}\frac{c(u)}{2}\sum_{v=0}^{7}\frac{c(v)}{2}\tilde{F}(u,v)\cos\frac{(2n+1)\,u\pi}{16}\cos\frac{(2m+1)\,v\pi}{16}, \quad n,m = 0,\ldots,7 \tag{3.9}$$

where $n, m, u, v \in \mathbb{N}$ and

$$\begin{array}{lll} c(u) & = \frac{1}{\sqrt{2}} & \text{for } u = 0 \\ c(u) & = 1 & \text{otherwise} \\ c(v) & = \frac{1}{\sqrt{2}} & \text{for } v = 0 \\ c(v) & = 1 & \text{otherwise} \\ f(n,m) & = & \text{2-D sample value} \\ \tilde{F}(u,v) & = & \text{2-D DCT coefficient} \end{array} \tag{3.10}$$

Equations (3.8) and (3.9) suggest that the calculation of one DCT coefficient requires 64 multiplications and 63 additions, but the two-dimensional DCT can be separated into 16

[2]Two vector's inner product is 1, if they are normalized, therefore the *-normal* in *orthonormal transform*

one-dimensional DCTs, as (3.11) and (3.12) illustrate:

2D FDCT:

$$\tilde{F}(u,v) = \frac{c(u)}{2}\frac{c(v)}{2}\sum_{n=0}^{7}\left(\cos\frac{(2n+1)\,u\pi}{16}\sum_{m=0}^{7} f(n,m)\cos\frac{(2m+1)\,u\pi}{16}\right), \quad u,v = 0,\ldots,7 \tag{3.11}$$

2D IDCT:

$$f(n,m) = \sum_{u=0}^{7}\left(\frac{c(u)}{2}\cos\frac{(2n+1)\,u\pi}{16}\sum_{v=0}^{7}\frac{c(v)}{2}\tilde{F}(u,v)\cos\frac{(2m+1)\,u\pi}{16}\right), \quad n,m = 0,\ldots,7 \tag{3.12}$$

This means, that a two-dimensional DCT can be obtained by applying first 8 one-dimensional DCTs over the rows, followed by another 8 one-dimensional DCTs to the columns of the input data matrix. Furthermore, there are quite a number of algorithms that reduce computational complexity drastically. Most of them are based on earlier work done on the Fast Fourier Transform, that preceeded the DCT about 10 years[3], and on the Discrete Fourier Transform in general. Because of the similarities between the Discrete Fourier Transform and the DCT it is therefore worthwile looking into the relations between these two orthogonal transforms.

3.2 Relations between the DCT and the DFT

In their introductory paper to the DCT ([20]), Ahmed et al. suggested to use existing Fast Fourier Transform algorithms to compute an N-point DCT from a 2N-point DFT. Section 3.2.1 elaborates on their initial proposal. The DCT is also very closely related to the Discrete Fourier Transform (DFT) on real inputs (which is the case for 2D image arrays). This relationship is the basis for the most efficient DCT algorithms known up to now. Section 3.2.2 shows, how an N-point DFT can be calculated from one N/2-point DFT and two N/4-point DCTs, which is the basis for the Ligtenberg-Vetterli Fast DCT and similar algorithms. Section 3.2.3 then shows the relations between the N-point DCT and the 2N-point DFT for the special case of a symmetrical input vector for the DFT, which is the basis for the fastest scaled one-dimensional DCT algorithm known up to now, the Arai-Agui-Nakajima Fast DCT.

[3]The famous Cooley-Tukey algorithm ([6]) was presented in 1965

3.2.1 Computing an N-point DCT from a 2N-point DFT

An N-point DFT ([3]) is defined as:

$$F(u) = \sum_{n=0}^{N-1} f(n) e^{-j\frac{2\pi nu}{N}} = \sum_{n=0}^{N-1} f(n) \left[\cos\left(\frac{2\pi nu}{N}\right) - j \sin\left(\frac{2\pi nu}{N}\right) \right], \quad u = 0, \ldots, N-1 \tag{3.13}$$

with $j = \sqrt{-1}$. Consequently, a 2N-point DFT is defined as:

$$F(u) = \sum_{n=0}^{2N-1} f(n) e^{-j\frac{2\pi nu}{2N}}, \quad u = 0, \ldots, 2N-1 \tag{3.14}$$

If we scale equation (3.14)with $e^{-ju\pi/(2N)}$ and assume that the input samples $f(N) \ldots f(2N-1)$ are zero, we get:

$$\begin{aligned} e^{-j\frac{\pi u}{2N}} \cdot \sum_{n=0}^{N-1} f(n) e^{-j\frac{2\pi nu}{2N}} &= \sum_{n=0}^{N-1} f(n) e^{-j\frac{u\pi(2n+1)}{2N}} \\ &= \sum_{n=0}^{N-1} f(n) \left(\cos\left(\frac{u\pi(2n+1)}{2N}\right) - j \sin\left(\frac{u\pi(2n+1)}{2N}\right) \right), \quad u = 0, \ldots, 2N-1 \end{aligned} \tag{3.15}$$

From equation (3.15) and a comparison with equations (3.1) and (3.4) we can deduce, that an N-point DCT can be computed by taking the real part of a 2N-point DFT that was scaled by the complex constant $e^{-ju\pi/(2N)}$. Note that this was an early suggestion of Ahmed et al. in their introductory paper to the DCT ([20]) on how to calculate a DCT leveraging existing algorithms or hardware for the calculation of the DFT. By no means, this calculation method should be considered a particularly efficient one.

3.2.2 A Divide-and-Conquer Scheme for real input vectors

In [32], Vetterli and Nussbaumer present a simple approach to break the task of calculating an N-point DFT into transformations of reduced complexity for real input vectors. For this purpose they define two operations, the Cosine DFT and the Sine DFT, that together represent the DFT operation. The Cosine DFT represents the real part of the DFT operation, whereas the Sine DFT represents the imaginary part of the DFT operation. They also use a slightly modified version of the DCT, which, in order to not confuse the reader, is called DCT* in the following. Vetterli and Nussbaumer also introduce the following notation for the N-point DFT of a function $f(n)$ which is the same as we already introduced in equation (3.13):

$$\mathrm{DFT}(u, N, f) = \sum_{n=0}^{N-1} f(n) e^{-j\frac{2\pi nu}{N}} = \sum_{n=0}^{N-1} f(n) \left[\cos\left(\frac{2\pi nu}{N}\right) - j \sin\left(\frac{2\pi nu}{N}\right) \right], \quad u = 0, \ldots, N-1 \tag{3.16}$$

with $j = \sqrt{-1}$. Vetterli and Nussbaumer define the Sine DFT (sin-DFT) and the Cosine DFT (cos-DFT) as:

$$\text{sin-DFT}(u, N, f) = \sum_{n=0}^{N-1} f(n) \sin\left(\frac{2\pi nu}{N}\right), \quad u = 0, \ldots, N-1 \tag{3.17}$$

$$\text{cos-DFT}(u, N, f) = \sum_{n=0}^{N-1} f(n) \cos\left(\frac{2\pi nu}{N}\right), \quad u = 0, \ldots, N-1 \tag{3.18}$$

The Sine DFT from equation (3.17) and the Cosine DFT from equation (3.18) together constitute the DFT from equation (3.16):

$$\text{DFT}(u, N, f) = \text{cos-DFT}(u, N, f) - j \cdot \text{sin-DFT}(u, N, f), \quad u = 0, \ldots, N-1 \tag{3.19}$$

The DCT* Operation is defined as:

$$\text{DCT*}(u, N, f) = \sum_{n=0}^{N-1} f(n) \cos\left(\frac{2\pi(2n+1)u}{4N}\right), \quad u = 0, \ldots, N-1 \tag{3.20}$$

Note that the DCT* in equation (3.20) is just a simplified form of the forward DCT in equation (3.4) without the scaling of $\sqrt{\frac{2}{N}}$ and the factor $1/\sqrt{2}$ for $u = 0$. By exploiting symmetry properties of the Sine and Cosine functions and under the assumption that N is divisible by 4, equation (3.18) can be rewritten as:

$$\begin{aligned} \text{cos-DFT}(u, N, f) &= \sum_{n=0}^{N/2-1} f(2n) \cos\left(\frac{2\pi nu}{N/2}\right) \\ &+ \sum_{n=0}^{N/4-1} (f(2n+1) + f(N-2n-1)) \cos\frac{2\pi(2n+1)u}{4N/4}, \\ &u = 0, \ldots, N-1 \end{aligned} \tag{3.21}$$

or with $f_1(n) = f(2n)$ for $n = 0, \ldots N/2-1$ and $f_2(n) = f(2n+1) + f(N-2n-1)$ for $n = 0, \ldots N/4-1$ as:

$$\text{cos-DFT}(u, N, f) = \text{cos-DFT}(u, N/2, f_1) + \text{DCT*}(u, N/4, f_2), \quad u = 0 \ldots, N-1 \tag{3.22}$$

Equation (3.22) shows, that an N-point Cosine DFT can be divided into an N/2-point Cosine DFT and an N/4-point DCT*. The N/2-point Cosine DFT can further be subdivided until only trivial operations are left. Similarly, the Sine DFT in equation (3.17) can be rewritten as:

$$\begin{aligned} \text{sin-DFT}(u, N, f) &= \sum_{n=0}^{N/2-1} f(2n) \sin\left(\frac{2\pi nu}{N/2}\right) \\ &+ \sum_{n=0}^{N/4-1} (f(2n+1) + f(N-2n-1)) \sin\frac{2\pi(2n+1)u}{4N/4}, \\ &u = 0, \ldots, N-1 \end{aligned} \tag{3.23}$$

Using the identity:

$$\sin\frac{2\pi n(2n+1)u}{N} = (-1)^n \cos\frac{2\pi(2n+1)(N/4-u)}{N} \tag{3.24}$$

equation (3.23) can be written in a more succinct form as:

$$\text{sin-DFT}(u,N,f) = \text{sin-DFT}(u,N/2,f_1) + \text{DCT*}(N/4-u,N/4,f_3), \quad u = 0\ldots,N-1 \tag{3.25}$$

with $f_3(n) = (-1)^n\,(f(2n+1) - f(N-2n-1))$. Equation (3.25) shows, that an N-point Sine DFT can be subdivided into an N/2 point Sine DFT and an N/4-point DCT*. Together with equation (3.22) this means, that an N-point DFT can be subdivided into one N/2-point DFT and two N/4-point DCT* operations.

We will now show how to do the reverse process and therefore compute a DCT* operation from a Sine DFT and a Cosine DFT:

With the mapping:

$$\begin{aligned} &f_4(n) = f(2n),\\ &f_4(N-n-1) = f(2n+1), \quad n = 0\ldots,N/2-1 \end{aligned}$$

the DCT* in equation (3.20) becomes:

$$\text{DCT*}(u,N,f) = \sum_{n=0}^{N-1} f_4(n)\cos\left(\frac{2\pi(4n+1)u}{4N}\right), \quad u = 0,\ldots,N-1 \tag{3.26}$$

With the basic geometrical identity $\cos(\alpha+\beta) = \cos\alpha\cos\beta - \sin\alpha\sin\beta$, equation (3.26) becomes:

$$\begin{aligned} \text{DCT*}(u,N,f) = &\cos\left(\frac{2\pi u}{4N}\right)\text{cos-DFT}(u,N,f_4)\\ &-\sin\left(\frac{2\pi u}{4N}\right)\text{sin-DFT}(u,N,f_4), \quad u = 0,\ldots,N-1 \end{aligned} \tag{3.27}$$

Using the symmetry properties of trigonometric functions and the Sine and Cosine DFT, this can be written as:

$$\begin{aligned} \begin{pmatrix} \text{DCT*}(u,N,f)\\ \text{DCT*}(N-u,N,f)\end{pmatrix} = &\begin{pmatrix} \cos(\frac{2\pi u}{4N}) & -\sin(\frac{2\pi u}{4N})\\ \sin(\frac{2\pi u}{4N}) & \cos(\frac{2\pi u}{4N})\end{pmatrix}\begin{pmatrix}\text{cos-DFT}(u,N,f_4)\\ \text{sin-DFT}(u,N,f_4)\end{pmatrix}\\ &u = 0\ldots,N/2-1 \end{aligned} \tag{3.28}$$

Equation (3.28) is a rotation of a vector consisting of the real and the imaginary part of the DFT with a rotation angle of $2\pi u/4N$. Thus we have shown how to compute a DCT* (and thus a DCT) from a DFT (equation (3.28)) and vice versa (equations (3.22) and (3.25)). We will show in section 3.3.2.1 that this rotation only requires 3 multiplications and 3 additions. In section 3.3.2.2 we will show how to transform this into flowgraphs for an 8-point DCT.

3.2.3 DCT and DFT for real and symmetrical input vectors

In [29], Tseng and Miller show, that if the mirror image of a sequence of N samples is appended to itself, the first N points of the 2N-point DFT performed on this vector of size 2N are scaled values of the DCT coefficients of the original N-point sequence. They conclude that an N-point DCT can therefore be very efficiently calculated by taking one double length real DFT [4] and a final output scaling, which will be shown in the following. If we use the shortcut $W_K = e^{-j\frac{2\pi}{K}}$, the K-point DFT (see equation (3.13)) can be written as:

$$F(u) = \sum_{n=0}^{K-1} f(n) W_K^{un}, \quad u = 0, \ldots, K-1 \tag{3.29}$$

If we set $K = 2N$ and extend the N-point sequence $f(n), n = 0, \ldots, N-1$ such that there is symmetry around the index $(2N-1)/2$, we get:

$$f(n) = f(2N-1-n), \quad n = 0, \ldots, 2N-1 \tag{3.30}$$

and equation (3.29) becomes:

$$F(u) = \sum_{n=0}^{N-1} f(n) W_{2N}^{un} + \sum_{n=N}^{2N-1} f(2N-n-1) W_{2N}^{un}, \quad u = 0, \ldots, 2N-1 \tag{3.31}$$

Now if we define a new index $k = (2N-n-1)$ for the the second sum in equation (3.31) this becomes:

$$F(u) = \sum_{n=0}^{N-1} f(n) W_{2N}^{un} + \sum_{k=0}^{N-1} f(k) W_{2N}^{-u(k+1)}, \quad u = 0, \ldots, 2N-1 \tag{3.32}$$

If k is now replaced with n and equation (3.32) is multiplied by $\frac{1}{2} W_{2N}^{\frac{u}{2}}$, we obtain:

$$\begin{aligned} \frac{1}{2} F(u) W_{2N}^{\frac{u}{2}} &= \frac{1}{2} \sum_{n=0}^{N-1} f(n) \left(W_{2N}^{\frac{u}{2}(2n+1)} + W_{2N}^{\frac{-u}{2}(2n+1)} \right) \\ &= \frac{1}{2} \sum_{n=0}^{N-1} f(n) \left(e^{-j\pi\frac{u}{2N}(2n+1)} + e^{j\pi\frac{u}{2N}(2n+1)} \right) \\ &= \sum_{n=0}^{N-1} f(n) \cos\frac{\pi u}{2N}(2n+1), \quad u = 0, \ldots, 2N-1 \end{aligned} \tag{3.33}$$

From equation (3.33) we can deduce that the first N DFT coefficients of a 2N-point DFT are the DCT coefficients of an N point DCT, scaled by a complex scaling factor, provided that equation (3.30) holds for the input vector.

Our goal will now be to calculate this scaling factor: Since the right side of equation (3.33) is real, the left side must also be real and we therefore can obtain the real part A_u and the imaginary part B_u of $F(u)$ with:

$$F(u) W_{2N}^{\frac{u}{2}} = (A_u + jB_u)(\cos\frac{\pi u}{2N} - j\sin\frac{\pi u}{2N}), \quad u = 0, \ldots, 2N-1 \tag{3.34}$$

[4] A real DFT denotes a DFT with real input, in contrast to a complex DFT with a complex input vector

Separating equation (3.34) into a real and an imaginary parts and setting the imaginary part to zero we get:

$$B_u = A_u \frac{\sin \frac{\pi u}{2N}}{\cos \frac{\pi u}{2N}} \tag{3.35}$$

which can be substituted back into equation (3.34):

$$\begin{aligned} F(u)W_{2N}^{\frac{u}{2}} &= A_u \cos \frac{\pi u}{2N} + A_u \frac{\sin^2 \frac{\pi u}{2N}}{\cos \frac{\pi u}{2N}} \\ &= A_u \frac{1}{\cos \frac{\pi u}{2N}} (\cos^2 \frac{\pi u}{2N} + \sin^2 \frac{\pi u}{2N}) \\ &= A_u \frac{1}{\cos \frac{\pi u}{2N}}, \quad u = 0, \ldots, 2N-1 \end{aligned} \tag{3.36}$$

From equations (3.33) and (3.36) we now get:

$$\sum_{n=0}^{N-1} f(n) \cos \frac{\pi u}{2N}(2n+1) = \frac{1}{2} \Re(F(u)) \frac{1}{cos \frac{\pi u}{2N}}, \quad u = 0, \ldots, 2N-1 \tag{3.37}$$

This shows that the DCT coefficients of an N-point DCT can simply be derived from scaling the real part of a 2N-point DFT with symmetrical input vector. Chapter 3.3.6 will show that for $N = 8$ this property is the basis of the Arai-Agui-Nakajima Fast DCT.

3.3 Fast one-dimensional 8-point DCTs

This section will cover in detail several fast one-dimensional DCT algorithms that have been developed over the years both for software and hardware implementations. Because of the widespread use of 8-point DCTs and IDCTs in continuous tone image encoding and decoding, we will strictly focus on algorithms for an 8-point input vector. This also means that we don't evaluate the complexity of algorithms in terms of "Big-Oh-Notation", such as $O(N)$, $O(N^2)$, $O(logN)$, because $N = 8$ is a fixed value. We rather compare algorithms by evaluating the number of additions and multiplications they require. In order to describe the algorithms a graphical notation, the flowgraph, will be introduced. We will start with a very simple 8-point DCT, taken from [22], that demonstrates how the symmetry of the sine and cosine function can be exploited to reduce computational complexity of the DCT. We will then proceed in section 3.3.2 to the first classical DCT algorithm, the Ligtenberg-Vetterli DCT (1986), that we analyze first with an algebraic approach (section 3.3.2.1) and then with a more graphical and intuitive approach (section 3.3.2.2). From there we will turn to the Loeffler-Ligtenberg-Moschytz DCT (1989) and its inverse, which is the fastest unscaled one-dimensional DCT known up to now. The section on fast one-dimensional 8-point DCTs is then concluded with section 3.3.6 that contains an analysis of the Arai-Agui-Nakajima Fast DCT (1988) and its inverse, which is the fastest scaled one-dimensional DCT known up to now. For this algorithm the "Winograd 16-point small-N DFT" is of particular importance. Therefore we will analyze this algorithm as well.

3.3.1 A simple and fast 8-point DCT

A simple approach ([22]) for a fast DCT takes advantage of the symmetry of sinusoidal functions. With the following definitions:

$$C_k = cos\frac{k\pi}{16}, S_k = sin\frac{k\pi}{16}, \quad k = 0, \ldots, 7 \tag{3.38}$$

we get from the symmetry property of the cosine and sine functions:

$$C_1 = S_7, C_2 = S_6, ..., C_7 = S_1 \tag{3.39}$$

and

$$C_1 = -C_{15}, C_2 = -C_{14}, ..., C_7 = -C_9. \tag{3.40}$$

We therefore define the following shortcuts for sums (3.41) and differences (3.42) of the samples $f(n)$:

$$\begin{aligned} s_{07} &= f(0) + f(7) \\ s_{16} &= f(1) + f(6) \\ s_{25} &= f(2) + f(5) \\ s_{34} &= f(3) + f(4) \\ s_{0734} &= s_{07} + s_{34} \\ s_{1625} &= s_{16} + s_{25} \end{aligned} \tag{3.41}$$

$$\begin{aligned} d_{07} &= f(0) - f(7) \\ d_{16} &= f(1) - f(6) \\ d_{25} &= f(2) - f(5) \\ d_{34} &= f(3) - f(4) \\ d_{0734} &= s_{07} - s_{34} \\ d_{1625} &= s_{16} - s_{25} \end{aligned} \tag{3.42}$$

With the shortcuts from equations (3.41) and (3.42), the DCT coefficients from (3.6) can be written as:

$$2\tilde{F}(0) = \frac{1}{\sqrt{2}} \sum_{n=0}^{7} f(n)\cos(0) = \cos\frac{4\pi}{16} \sum_{n=0}^{7} f(n) = C_4(s_{0734} + s_{1625}) \tag{3.43}$$

$$\begin{aligned} 2\tilde{F}(1) &= \cos\frac{\pi}{16}f(0) + \cos\frac{3\pi}{16}f(1) + \cos\frac{5\pi}{16}f(2) + \cos\frac{7\pi}{16}f(3) \\ &+ \cos\frac{9\pi}{16}f(4) + \cos\frac{11\pi}{16}f(5) + \cos\frac{13\pi}{16}f(6) + \cos\frac{15\pi}{16}f(7) \\ &= \cos\frac{\pi}{16}[f(0) - f(7)] + \cos\frac{3\pi}{16}[f(1) - f(6)] \\ &+ \cos\frac{5\pi}{16}[f(2) - f(5)] + \cos\frac{7\pi}{16}[f(3) - f(4)] \\ &= C_1 d_{07} + C_3 d_{16} + C_5 d_{25} + C_7 d_{34} \end{aligned} \tag{3.44}$$

$$
\begin{aligned}
2\tilde{F}(2) &= \cos\frac{2\pi}{16}f(0) + \cos\frac{6\pi}{16}f(1) + \cos\frac{10\pi}{16}f(2) + \cos\frac{14\pi}{16}f(3) \\
&+ \cos\frac{18\pi}{16}f(4) + \cos\frac{22\pi}{16}f(5) + \cos\frac{26\pi}{16}f(6) + \cos\frac{30\pi}{16}f(7) \\
&= \cos\frac{2\pi}{16}\left[f(0) + f(7) - f(3) - f(4)\right] \\
&+ \cos\frac{6\pi}{16}\left[f(1) + f(6) - f(2) - f(5)\right] \\
&= C_2 d_{0734} + C_6 d_{1625} \qquad (3.45)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(3) &= \cos\frac{3\pi}{16}f(0) + \cos\frac{9\pi}{16}f(1) + \cos\frac{15\pi}{16}f(2) + \cos\frac{21\pi}{16}f(3) \\
&+ \cos\frac{27\pi}{16}f(4) + \cos\frac{33\pi}{16}f(5) + \cos\frac{39\pi}{16}f(6) + \cos\frac{45\pi}{16}f(7) \\
&= \cos\frac{3\pi}{16}\left[f(0) - f(7)\right] - \cos\frac{7\pi}{16}\left[f(1) - f(6)\right] \\
&+ \cos\frac{7\pi}{16}\left[f(2) - f(5)\right] - \cos\frac{5\pi}{16}\left[f(3) - f(4)\right] \\
&= C_3 d_{07} - C_7 d_{16} - C_1 d_{25} - C_5 d_{34} \qquad (3.46)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(4) &= \cos\frac{4\pi}{16}f(0) + \cos\frac{12\pi}{16}f(1) + \cos\frac{20\pi}{16}f(2) + \cos\frac{28\pi}{16}f(3) \\
&+ \cos\frac{36\pi}{16}f(4) + \cos\frac{44\pi}{16}f(5) + \cos\frac{52\pi}{16}f(6) + \cos\frac{60\pi}{16}f(7) \\
&= \cos\frac{4\pi}{16}\left[f(0) + f(3) + f(4) + f(7) - f(1) - f(2) - f(5) - f(6)\right] \\
&= C_4(s_{0734} - s_{1625}) \qquad (3.47)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(5) &= \cos\frac{5\pi}{16}f(0) + \cos\frac{15\pi}{16}f(1) + \cos\frac{25\pi}{16}f(2) + \cos\frac{35\pi}{16}f(3) \\
&+ \cos\frac{45\pi}{16}f(4) + \cos\frac{55\pi}{16}f(5) + \cos\frac{65\pi}{16}f(6) + \cos\frac{75\pi}{16}f(7) \\
&= \cos\frac{5\pi}{16}\left[f(0) - f(7)\right] - \cos\frac{\pi}{16}\left[f(1) - f(6)\right] \\
&+ \cos\frac{7\pi}{16}\left[f(2) - f(5)\right] + \cos\frac{3\pi}{16}\left[f(3) - f(4)\right] \\
&= C_5 d_{07} - C_1 d_{16} + C_7 d_{25} + C_3 d_{34} \qquad (3.48)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(6) &= \cos\frac{6\pi}{16}f(0) + \cos\frac{18\pi}{16}f(1) + \cos\frac{30\pi}{16}f(2) + \cos\frac{42\pi}{16}f(3) \\
&+ \cos\frac{54\pi}{16}f(4) + \cos\frac{66\pi}{16}f(5) + \cos\frac{78\pi}{16}f(6) + \cos\frac{90\pi}{16}f(7) \\
&= \cos\frac{6\pi}{16}\left[f(0) + f(7) - f(3) - f(4)\right] + cos\frac{2\pi}{16}\left[f(2) + f(5) - f(1) - f(6)\right] \\
&= C_6 d_{0734} - C_2 d_{1625} \qquad (3.49)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(7) &= \cos\frac{7\pi}{16}f(0) + \cos\frac{21\pi}{16}f(1) + \cos\frac{35\pi}{16}f(2) + \cos\frac{49\pi}{16}f(3) \\
&+ \cos\frac{63\pi}{16}f(4) + \cos\frac{77\pi}{16}f(5) + \cos\frac{91\pi}{16}f(6) + \cos\frac{105\pi}{16}f(7) \\
&= \cos\frac{7\pi}{16}\left[f(0) - f(7)\right] - \cos\frac{5\pi}{16}\left[f(1) - f(6)\right] \\
&+ \cos\frac{3\pi}{16}\left[f(2) - f(5)\right] - \cos\frac{\pi}{16}\left[f(3) - f(4)\right] \\
&= C_7 d_{07} - C_5 d_{16} + C_3 d_{25} - C_1 d_{34} \qquad (3.50)
\end{aligned}
$$

By arranging the terms in equations (3.43)-(3.50), the sums and differences from equations (3.41) and (3.42) can be reused to calculate the DCT coefficients and thus the total amount of operations is reduced to 22 multiplications and 28 additions. This is quite some progress, compared to the naïve approach of calculating each of the 8 coefficients by 7 additions and 8 multiplications.

3.3.2 The Ligtenberg-Vetterli-DCT

In this section an algorithm is presented that was introduced by Ligtenberg and Vetterli and implemented in hardware as early as 1986 ([31]). It employs the Divide-and-Conquer scheme presented in section 3.2.2. We will first use in section 3.3.2.1 the algebraic approach from [22], which should be enough to successfully implement this algorithm and requires no prior understanding of section 3.2.2. Section 3.3.2.2 will then use a more intuitive approach and will show how to graphically derive a flowgraph for this algorithm based on chapter 3.2.2.

3.3.2.1 An algebraic approach to the Ligtenberg-Vetterli-DCT

The Ligtenberg-Vetterli-DCT uses equations (3.43)-(3.50) and the sums and differences from equations (3.41) and (3.42), but with slightly different grouping:

$$2\tilde{F}(0) = C_4\left[(s_{07} + s_{12}) + (s_{34} + s_{56})\right] \qquad (3.51)$$

$$
\begin{aligned}
2\tilde{F}(1) &= C_1 d_{07} + C_3 d_{16} + C_5 d_{25} + C_7 d_{34} \\
&= (C_1 d_{07} + S_1 d_{34}) + (C_3 d_{16} + S_3 d_{25}) \qquad (3.52)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(2) &= C_2(s_{07} - s_{34}) + C_6(d_{12} - d_{56}) \\
&= C_6(d_{12} - d_{56}) + S_6(s_{07} - s_{34}) = C_{-6}(d_{12} - d_{56}) - S_{-6}(s_{07} - s_{34}) \qquad (3.53)
\end{aligned}
$$

$$
\begin{aligned}
2\tilde{F}(3) &= C_3 d_{07} - (C_5 d_{34} + C_7 d_{16} + C_1 d_{25}) \\
&= (C_3 d_{07} - S_3 d_{34}) - (S_1 d_{16} + C_1 d_{25}) \qquad (3.54)
\end{aligned}
$$

$$2\tilde{F}(4) = C_4\left[(s_{07} + s_{34}) - (s_{12} + s_{56})\right] \qquad (3.55)$$

$$
\begin{aligned}
2\tilde{F}(5) &= C_5 d_{07} + C_7 d_{25} + C_3 d_{34} - C_1 d_{16} \\
&= (S_3 d_{07} + C_3 d_{34}) + (S_1 d_{25} - C_1 d_{16}) \qquad (3.56)
\end{aligned}
$$

$$\begin{aligned} 2\tilde{F}(6) &= C_6(s_{07} - s_{34}) + C_2(s_{25} - s_{16}) \\ &= -S_6(d_{12} - d_{56}) + C_6(s_{07} - s_{34}) = S_{-6}(d_{12} - d_{56}) + C_{-6}(s_{07} - s_{34}) \end{aligned} \quad (3.57)$$

$$\begin{aligned} 2\tilde{F}(7) &= C_7 d_{07} + C_3 d_{25} - (C_5 d_{16} + C_1 d_{34}) \\ &= (S_1 d_{07} - C_1 d_{34}) + (C_3 d_{25} - C_5 d_{16}) \end{aligned} \quad (3.58)$$

From the standard geometric identities

$$\cos\alpha - \cos\beta = -2\sin\frac{\alpha+\beta}{2}\sin\frac{\alpha-\beta}{2} \quad (3.59)$$

and

$$\cos\alpha + \cos\beta = 2\cos\frac{\alpha+\beta}{2}\cos\frac{\alpha-\beta}{2} \quad (3.60)$$

we get for $k, l \in \mathbb{N}$:

$$C_k - C_l = -2sin\frac{(k+l)\pi}{32}sin\frac{(k-l)\pi}{32} = -2S_{(k+l)/2}S_{(k-l)/2} \quad (3.61)$$

$$C_k + C_l = 2cos\frac{(k+l)\pi}{32}cos\frac{(k-l)\pi}{32} = 2C_{(k+l)/2}C_{(k-l)/2} \quad (3.62)$$

From (3.51) and (3.55) we can deduce, that $\tilde{F}(0)$ and $\tilde{F}(4)$ can be calculated very straightforward by simple additions and a multiplication. For all other DCT coefficients, the Ligtenberg-Vetterli Fast DCT employs the rotation operation, that was introduced in section 3.2.2. The rotation operation rotates two input values (x, y) to output values (X, Y) by an angle Θ using the following equations:

$$X = x\cos\Theta - y\sin\Theta \quad (3.63)$$

$$Y = x\sin\Theta + y\cos\Theta \quad (3.64)$$

For a rotation angle $\Theta = \frac{k\pi}{16}$, equations (3.63) and (3.64) become:

$$X = C_k x - S_k y \quad (3.65)$$

$$Y = S_k x + C_k y \quad (3.66)$$

which can be expressed as:

$$X = C_k(x+y) - (S_k + C_k)y \quad (3.67)$$

$$Y = (S_k - C_k)x + C_k(x+y) \quad (3.68)$$

with S_k and C_k defined as in equation (3.38). Note that equations (3.65) and (3.66) require four multiplications and two additions whereas equations (3.67) and (3.68) require only three multiplications and three additions ($S_k \pm C_k$ are constants that can be precalculated) which is advantageous if a multiplication is costlier than an addition on a given computer hardware.

From (3.53) and (3.57) we can see that calculating $2\tilde{F}(2)$ and $2\tilde{F}(6)$ is simply a rotation of the inputs $x = (d_{12} - d_{56})$ and $y = (s_{07} - s_{34})$ with a rotation angle of $\Theta = \frac{-6\pi}{16}$. In order to calculate the other DCT coefficients with a rotation, some additional recasting is required to bring the terms for the DCT coefficients into a form that is suitable for the rotation operation:

For $\tilde{F}(3)$, the term $(S_1 d_{16} + C_1 d_{25})$ in (3.54) can be expressed in terms of a sum of s_{12}, d_{12}, s_{56} and d_{56} with four unknown coefficients A, B, C and D:

$$S_1 d_{16} + C_1 d_{25} = A s_{12} + B d_{12} + C s_{56} + D d_{56} \tag{3.69}$$

Solving the coefficients with four separate equations for $f(1)$, $f(2)$, $f(5)$ and $f(6)$ yields:

$$\begin{array}{lrcl} f(1): & S_1 & = & A + B \\ f(2): & C_1 & = & A - B \\ f(5): & -C_1 & = & C + D \\ f(6): & -S_1 & = & C - D \end{array} \tag{3.70}$$

Adding and subtracting the equations in (3.70) the unknown coefficients A, B, C and D become:

$$\begin{array}{l} A = \frac{1}{2}(S_1 + C_1) = \frac{1}{2}(C_7 + C_1) = C_4 C_3 \\ B = \frac{1}{2}(S_1 - C_1) = \frac{1}{2}(C_7 - C_1) = -S_4 S_3 = -C_4 S_3 \\ C = -\frac{1}{2}(C_1 + S_1) = -\frac{1}{2}(C_7 + C_1) = -C_4 C_3 \\ D = \frac{1}{2}(S_1 - C_1) = \frac{1}{2}(C_7 - C_1) = -S_4 S_3 = -C_4 S_3 \end{array} \tag{3.71}$$

With (3.71) the term $(S_1 d_{16} + C_1 d_{25})$ can be expressed as:

$$S_1 d_{16} + C_1 d_{25} = C_4 \left[C_3 (s_{12} - s_{56}) - S_3 (d_{12} + d_{56}) \right] \tag{3.72}$$

and thus (3.54) becomes:

$$2\tilde{F}(3) = \left[C_3 d_{07} - S_3 d_{34} \right] - C_4 \left[C_3 (s_{12} - s_{56}) - S_3 (d_{12} + d_{56}) \right] \tag{3.73}$$

For $\tilde{F}(5)$ we again express the term $(S_1 d_{25} - C_1 d_{16})$ in (3.56) in terms of a sum of s_{12}, d_{12}, s_{56} and d_{56} with four unknown coefficients A, B, C and D:

$$S_1 d_{25} - C_1 d_{16} = A s_{12} + B d_{12} + C s_{56} + D d_{56} \tag{3.74}$$

Solving the coefficients with four separate equations for $f(1)$, $f(2)$, $f(5)$ and $f(6)$ yields:

$$\begin{array}{lrcl} f(1): & -C_1 & = & A + B \\ f(2): & S_1 & = & A - B \\ f(5): & -S_1 & = & C + D \\ f(6): & C_1 & = & C - D \end{array} \tag{3.75}$$

Again, adding and subtracting the equations in (3.75) the unknown coefficients A, B, C and D become:

$$\begin{array}{l} A = -\frac{1}{2}(C_1 - S_1) = \frac{1}{2}(C_7 - C_1) = -S_4 S_3 = -C_4 S_3 \\ B = -\frac{1}{2}(C_1 + S_1) = -\frac{1}{2}(C_1 + C_7) = -C_4 C_3 \\ C = -\frac{1}{2}(S_1 - C_1) = -\frac{1}{2}(C_7 - C_1) = S_4 S_3 = C_4 S_3 \\ D = -\frac{1}{2}(S_1 + C_1) = -\frac{1}{2}(C_7 + C_1) = -C_4 C_3 \end{array} \tag{3.76}$$

With (3.76) the term $S_1 d_{25} - C_1 d_{16}$ can be expressed as:

$$S_1 d_{25} - C_1 d_{16} = C_4 \left[-S_3(s_{12} - s_{56}) - C_3(d_{12} + d_{56})\right] \tag{3.77}$$

and thus (3.56) becomes:

$$2\tilde{F}(5) = \left[S_3 d_{07} + C_3 d_{34}\right] + C_4 \left[-S_3(s_{12} - s_{56}) - C_3(d_{12} + d_{56})\right] \tag{3.78}$$

Rearranging equations (3.73) and (3.78) gives:

$$\begin{array}{l} 2\tilde{F}(3) = C_3 \left[d_{07} - C_4(s_{12} - s_{56})\right] - S_3 \left[d_{34} - C_4(d_{12} + d_{56})\right] \\ 2\tilde{F}(5) = S_3 \left[d_{07} - C_4(s_{12} - s_{56})\right] + C_3 \left[d_{34} - C_4(d_{12} + d_{56})\right] \end{array} \tag{3.79}$$

which is a rotation of $x = d_{07} - C_4(s_{12} - s_{56})$ and $y = d_{34} - C_4(d_{12} + d_{56})$ with a rotation angle of $\Theta = \frac{3\pi}{16}$.

The two remaining DCT coefficients, $\tilde{F}(1)$ and $\tilde{F}(7)$, can be calculated in a similar way to $\tilde{F}(3)$ and $\tilde{F}(5)$:

First the term $(C_3 d_{16} + S_3 d_{25})$ in (3.52) is expressed in terms of a sum of s_{12}, d_{12}, s_{56} and d_{56} with four unknown coefficients A, B, C and D:

$$C_3 d_{16} + S_3 d_{25} = A s_{12} + B d_{12} + C s_{56} + D d_{56} \tag{3.80}$$

Solving the coefficients with four separate equations for $f(1)$, $f(2)$, $f(5)$ and $f(6)$ yields:

$$\begin{array}{lrcl} f(1): & C_3 & = & A + B \\ f(2): & S_3 & = & A - B \\ f(5): & -S_3 & = & C + D \\ f(6): & -C_3 & = & C - D \end{array} \tag{3.81}$$

Again, adding and subtracting the equations in (3.81) the unknown coefficients A, B, C and D become:

$$\begin{array}{l} A = \frac{1}{2}(C_3 + S_3) = \frac{1}{2}(C_3 + C_5) = C_4 C_1 \\ B = \frac{1}{2}(C_3 - S_3) = -\frac{1}{2}(C_5 - C_3) = S_4 S_1 = C_4 S_1 \\ C = -\frac{1}{2}(S_3 + C_3) = -\frac{1}{2}(C_5 + C_3) = -C_4 C_1 \\ D = -\frac{1}{2}(S_3 - C_3) = -\frac{1}{2}(C_5 - C_3) = S_4 S_1 = C_4 S_1 \end{array} \tag{3.82}$$

With (3.82) the term $C_3 d_{16} + S_3 d_{25}$ can be expressed as:

$$C_3 d_{16} + S_3 d_{25} = C_4 \left[C_1(s_{12} - s_{56}) + S_1(d_{12} + d_{56})\right] \tag{3.83}$$

and thus (3.52) becomes:

$$2\tilde{F}(1) = (C_1 d_{07} + S_1 d_{34}) + C_4 \left[C_1(s_{12} - s_{56}) + S_1(d_{12} + d_{56})\right] \tag{3.84}$$

For S(7) we again express the term $(C_3 d_{25} - C_5 d_{16})$ in (3.58) in terms of a sum of s_{12}, d_{12}, s_{56} and d_{56} with four unknown coefficients A, B, C and D:

$$C_3 d_{25} - C_5 d_{16} = A s_{12} + B d_{12} + C s_{56} + D d_{56} \tag{3.85}$$

Solving the coefficients with four separate equations for $f(1), f(2), f(5)$ and $f(6)$ yields:

$$\begin{array}{lrcl} f(1): & -C_5 & = & A + B \\ f(2): & C_3 & = & A - B \\ f(5): & -C_3 & = & C + D \\ f(6): & C_5 & = & C - D \end{array} \tag{3.86}$$

Again, adding and subtracting the equations in (3.86) the unknown coefficients A, B, C and D become:

$$\begin{array}{l} A = \frac{1}{2}(C_3 - C_5) = S_4 S_1 = C_4 S_1 \\ B = -\frac{1}{2}(C_5 + C_3) = -C_4 C_1 \\ C = \frac{1}{2}(C_5 - C_3) = -S_4 S_1 = -C_4 S_1 \\ D = -\frac{1}{2}(C_5 + C_3) = -C_4 C_1 \end{array} \tag{3.87}$$

With (3.87) the term $C_3 d_{25} - C_5 d_{16}$ can be expressed as:

$$C_3 d_{25} - C_5 d_{16} = C_4 \left[S_1 (s_{12} - s_{56}) - C_1 (d_{12} + d_{56}) \right] \tag{3.88}$$

and thus (3.58) becomes:

$$2\tilde{F}(7) = (S_1 d_{07} - C_1 d_{34}) + C_4 \left[S_1 (s_{12} - s_{56}) - C_1 (d_{12} + d_{56}) \right] \tag{3.89}$$

Rearranging equations (3.84) and (3.89) gives:

$$\begin{array}{l} 2\tilde{F}(1) = C_1 \left[d_{07} + C_4 (s_{12} - s_{56}) \right] - S_1 \left[-d_{34} - C_4 (d_{12} + d_{56}) \right] \\ 2\tilde{F}(7) = S_1 \left[d_{07} + C_4 (s_{12} - s_{56}) \right] + C_1 \left[-d_{34} - C_4 (d_{12} + d_{56}) \right] \end{array} \tag{3.90}$$

which is a rotation of $x = d_{07} + C_4(s_{12} - s_{56})$ and $y = -d_{34} - C_4(d_{12} + d_{56})$ with a rotation angle of $\Theta = \frac{\pi}{16}$.

We can now calculate the 8 point DCT with 3 rotations (9 multiplications), the two multiplications from equations (3.51) and (3.55) and another two multiplications of C_4 with $(d_{12} + d_{56})$ and $(s_{12} - s_{56})$ which serve as an input for the rotations in equations (3.79) and (3.90). The total number of operations sums up to 13 multiplications and 29 additions, which will become much more obvious in section 3.3.2.2.

3.3.2.2 A graphical approach to the Ligtenberg-Vetterli-DCT

In this section, we will take a more intuitive approach to the computation of the Ligtenberg-Vetterli DCT than in the preceeding chapter:

If an N point real DFT or a DCT* operation is simply considered a "black box" with N input values and N output values, equations (3.22) and (3.25) can be described graphically like in figure 3.1 for $N = 8$. Such a flowgraph is to be read from left to right, a filled dot represents an addition of two values and an arrow represents a negation of a value [5]. The

[5] A hollow dot represents the reuse of a value for more than one following operation and thus is not really an operation that contributes to computational complexity.

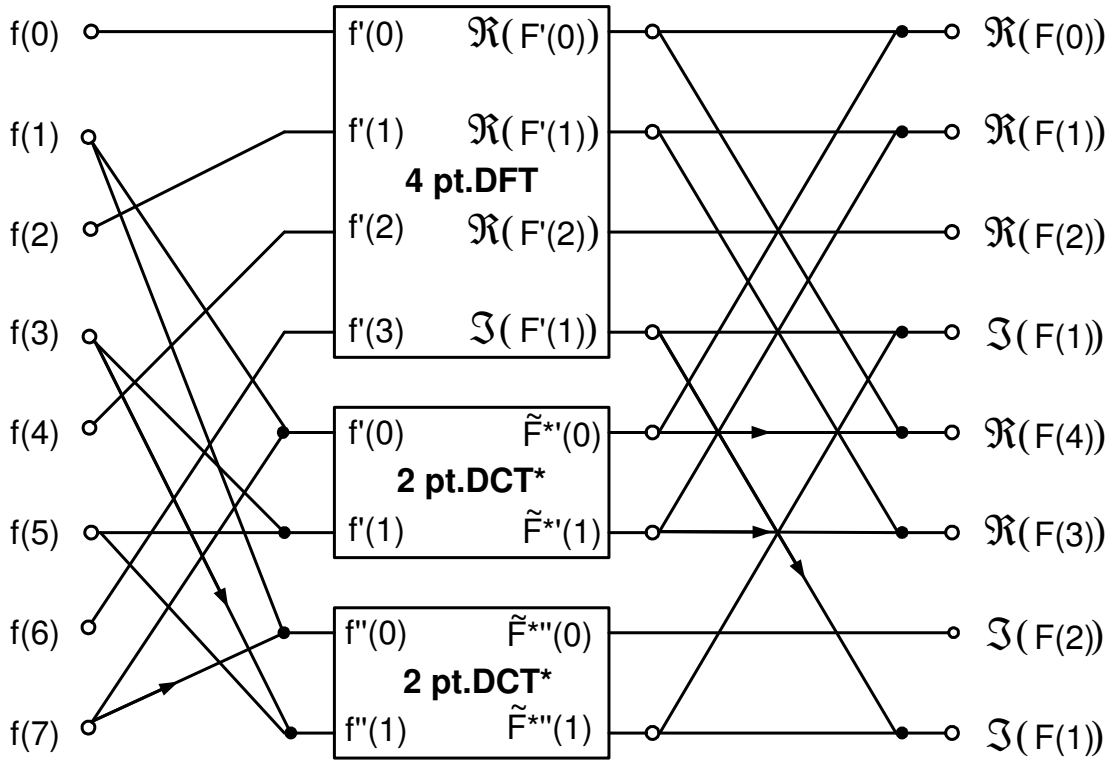

Figure 3.1: A graphical description of an 8 point real DFT

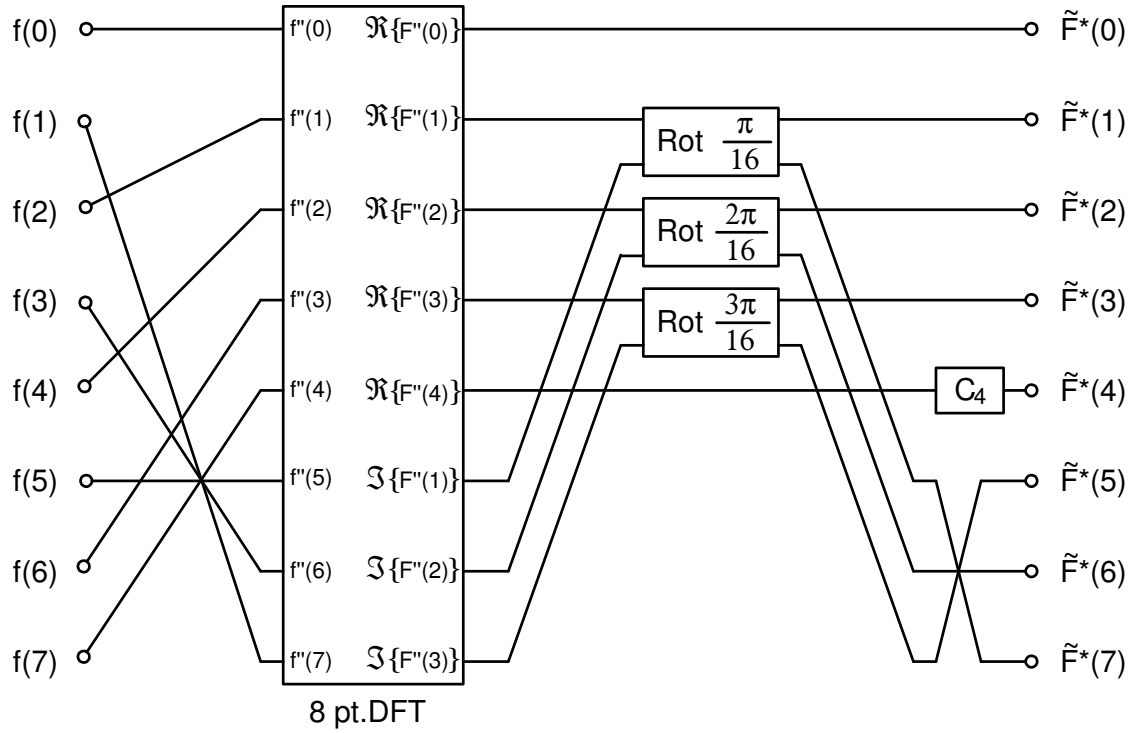

Figure 3.2: A graphical description of an 8 point DCT*

astute reader might object at this point that an N-point DFT should yield an N-point real part and an N-point imaginary part, but it can be easily shown, that for a real input signal function the Fourier Transform yields an even function for its real part and an odd function for its imaginary part. Therefore

$$\Im\{F(0)\} = \text{sin-DFT}(0, N, f) = 0 \tag{3.91}$$

and

$$\Im\{F(N/2)\} = \text{sin-DFT}(N/2, N, f) = 0 \tag{3.92}$$

and

$$\Im\{F(u)\} = \text{sin-DFT}(u, N, f) = \text{-sin-DFT}(N-u, N, f) = -\Im\{X(N-u)\}. \tag{3.93}$$

Similarly, for the real part, we can find

$$\Re\{F(u)\} = \text{cos-DFT}(u, N, f) = \text{cos-DFT}(N-u, N, f) = \Re\{F(N-u)\}. \tag{3.94}$$

This reduces the total number of distinct and nonzero values of the N-point real DFT to N values.

The computation of a DCT* from a DFT with rotation operations, as expressed in equation (3.28), can be described graphically like in figure 3.2 for $N = 8$. In this flowgraph, a box labeled with $\text{Rot}\frac{n\pi}{16}$ denotes a rotation operation of the two input values as in equation

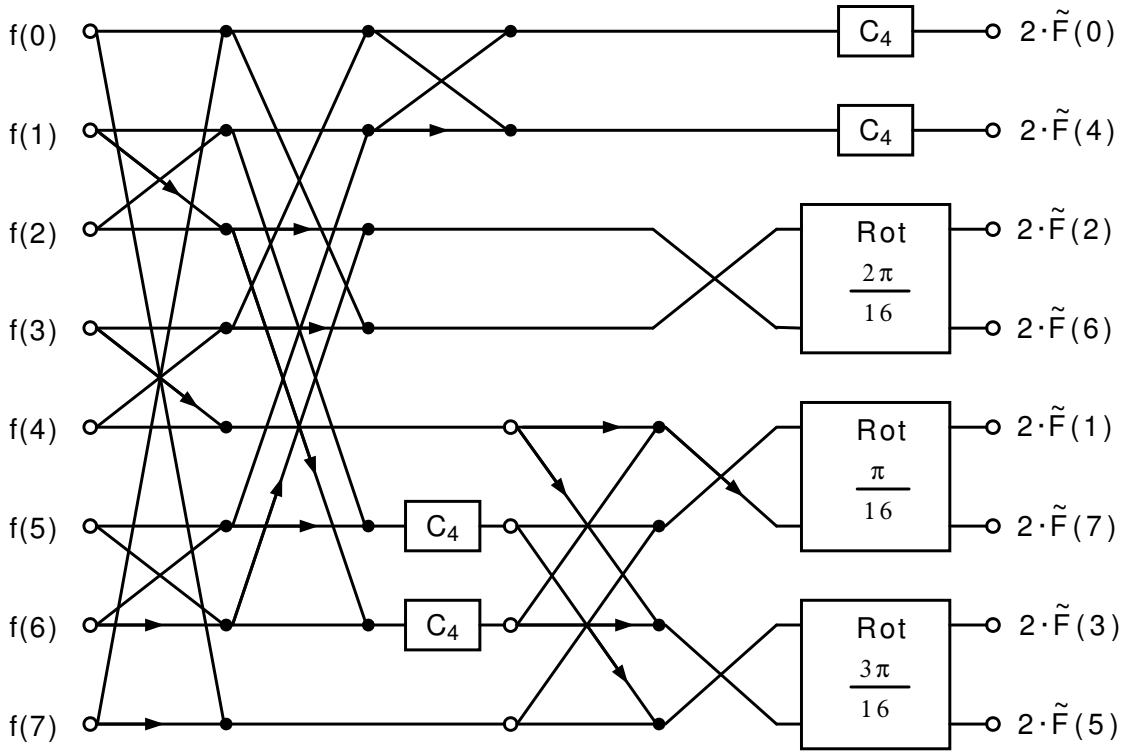

Figure 3.3: A combination of figures 3.1 and 3.2 to derive the DCT from the DCT*

(3.28) with a rotation angle of $n\pi/16$. A box labeled C_4 represents a multiplication with $C_4 = 1/\sqrt{2}$. In order to not confuse the reader, the output vector of the DCT* in figure 3.2 is denoted $\tilde{F}^*(0) \dots \tilde{F}^*(7)$. If the two figures 3.1 and 3.2 are cleverly combined, such that the black box labeled "8-pt. DFT" in figure 3.2 is replaced by the complete flowgraph in figure 3.1, it is very easy to derive a flowgraph for the DCT* and finally, for the DCT, like in figure (3.3). Note that we have to add an additional multiplication with C_4 for $S(0)$ in order to derive the DCT from the DCT*. From figure (3.3), some optimization in the number of required negations leads to the flowgraph in figure 3.4. As can be easily seen from the flowgraph in figure 3.4, the computational complexity of the Ligtenberg-Vetterli-DCT sums up to a total of 13 multiplications and 29 additions. Note that one rotation takes 3 multiplications and 3 additions. The output of the flowgraph is 2 times the value of the DCT coefficients, but the required division by 2 can easily be done by a right shift operation which is negligible in computation costs, compared to an addition or a multiplication. In figure 3.4 the boxes for the rotation operation get additional designators where a minus sign means a negation of the input value. Such a negation of a rotation's input vector does not contribute to computational complexity, since it can be absorbed into the precalculated constants of the rotation operation. Ligtenberg and Vetterli actually sacrificed some computational complexity for the sake of a simpler design of their chip. Figure 3.5 shows the flowgraph for their final design where one "butterfly"-operation (2 additions) in the third stage and the two multiplications by $1/\sqrt{2}$ in the last stage are replaced by a single rotation with a rotation angle of $4\pi/16$ (three multiplications, three

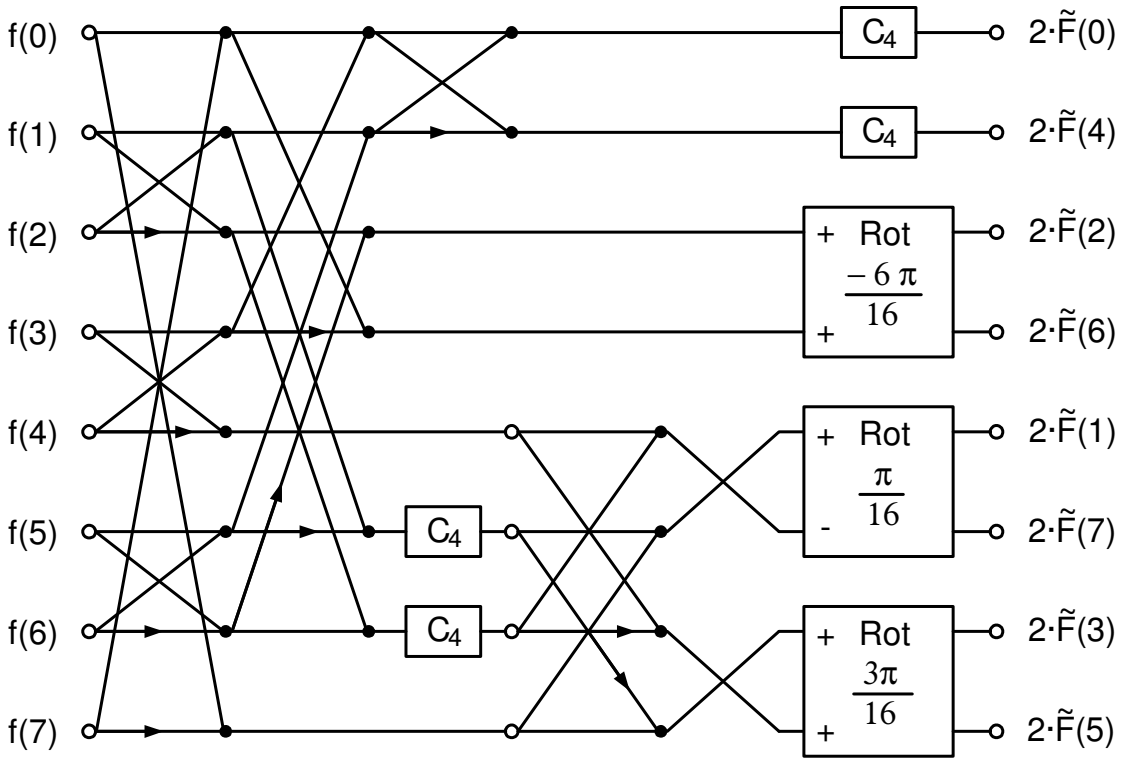

Figure 3.4: Flowgraph for the Ligtenberg-Vetterli Fast DCT

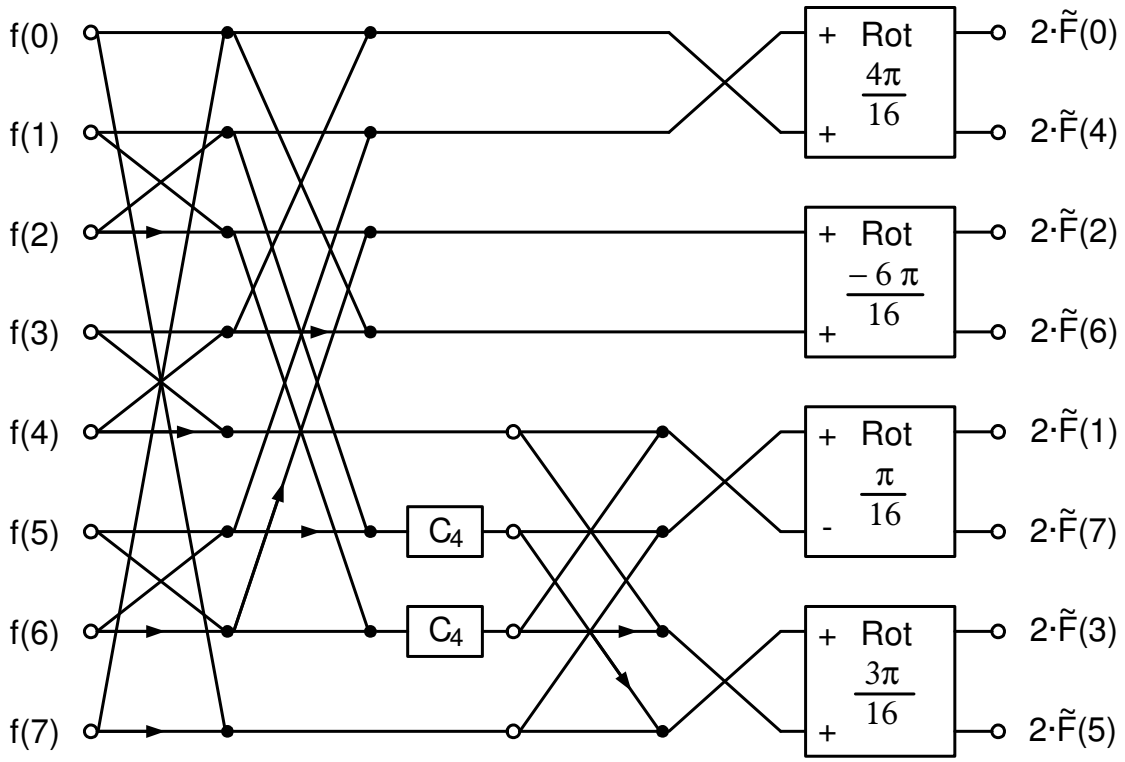

Figure 3.5: Hardware implementation of the Ligtenberg-Vetterli Fast DCT

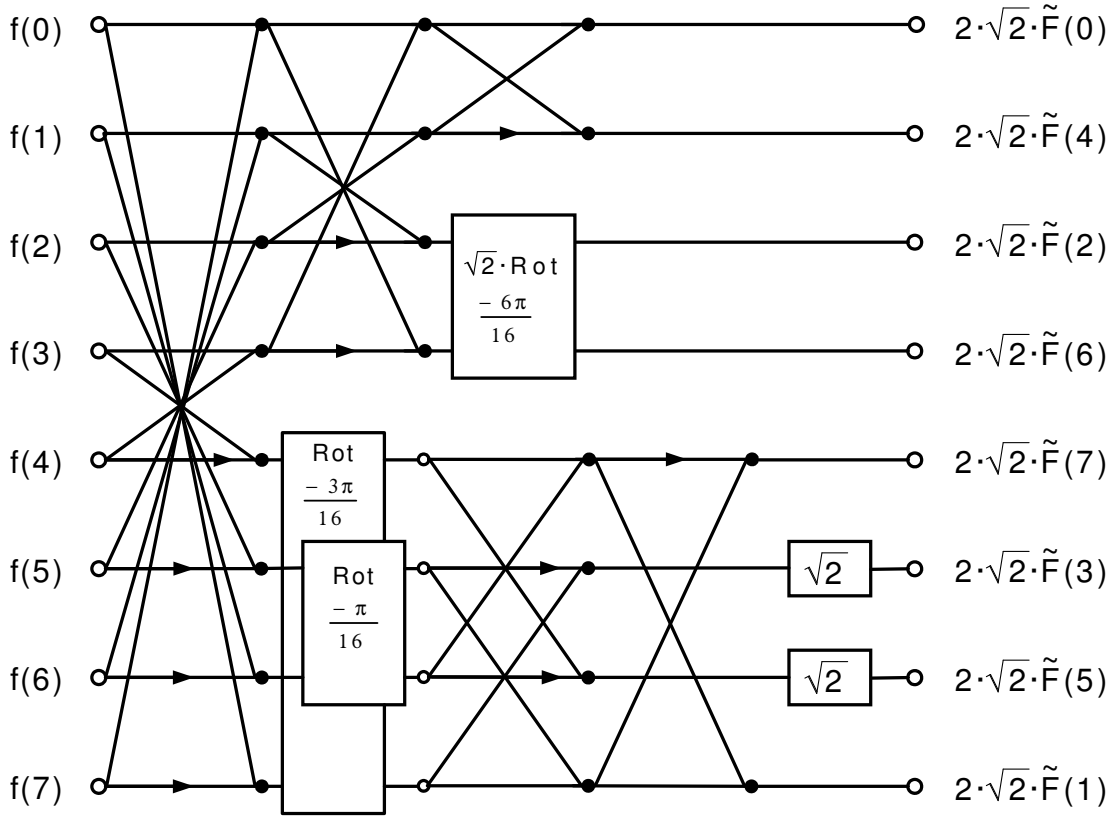

Figure 3.6: Flowgraph for the Loeffler-Ligtenberg-Moschytz Fast DCT

additions). Note that a software implementation of their algorithm should preferrably use the flowgraph in figure 3.4 instead of the flowgraph in figure 3.5.

3.3.3 The Loeffler-Ligtenberg-Moschytz-DCT

In [4], Loeffler et al. present an 8-point DCT algorithm that needs only 11 multiplications and 29 additions. Figure 3.6 shows the flowgraph of this algorithm. This algorithm uses the same kind of rotation as in equations (3.65) and (3.66), denoted by a box labeled with RotΘ, with Θ being the rotation angle. The rotation in the third stage with a rotation angle of $\frac{-6\pi}{16}$ is additionally scaled by $\sqrt{2}$, therefore it is labeled $\sqrt{2}\text{Rot}\frac{-6\pi}{16}$. Note that this scaling can be absorbed into the constants of the rotation operation and therefore does not contribute to computational complexity. The output of this algorithm is scaled to $2\sqrt{2}$ times the DCT coefficients of the input vector, but after a 2D DCT with a total scaling of $4\sqrt{2}\sqrt{2}$, this can easily be reverted by a single right shift operation for each coefficient.

The Loeffler-Ligtenberg-Moschytz Fast DCT can now be written as a matrix-vector

product:

$$
\begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} = \frac{1}{2\sqrt{2}} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2}C_6 & \sqrt{2}S_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & -\sqrt{2}S_6 & \sqrt{2}C_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & C_3 & 0 & 0 & S_3 \\ 0 & 0 & 0 & 0 & 0 & C_1 & S_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -S_1 & C_1 & 0 \\ 0 & 0 & 0 & 0 & -S_3 & 0 & 0 & C_3 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} \tag{3.95}
$$

In [4], Loeffler et al. show, that algorithms like the Ligtenberg-Vetterli-DCT, which allow the parallel execution of multiplications within one stage, will always take at least 12 multiplications, whereas their own algorithm belongs to a class of algorithms that uses a cascaded multiplication in one stage but therefore needs one multiplication less. Note that in [4] the scaled rotation in stage 3 of figure 1 is wrong. It should read $\sqrt{2}\ c6$ instead of $\sqrt{2}\ c1$. Note also, that Loeffler, Ligtenberg and Moschytz omitted a scaling factor of $\sqrt{2}$ in all constants used in figure 8 in [4], section 4.3.

In [15], an excellent introduction into this algorithm and how to translate it into source code, along with performance figures for different DCT implementations, can be found. The C source code that accompanies this article is available electronically from the website of *Dr. Dobb's Journal* (http://www.ddj.com).

3.3.4 The Inverse Loeffler-Ligtenberg-Moschytz-DCT

The Inverse Loeffler-Ligtenberg-Moschytz-DCT can easily be derived by simply reversing the order of matrices in equation (3.95) and inverting the matrices as well. The Inverse Loeffler-Ligtenberg-Moschytz-DCT can then be given as:

$$
\begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} = \frac{1}{2\sqrt{2}} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & C_3 & 0 & 0 & -S_3 \\ 0 & 0 & 0 & 0 & 0 & C_1 & -S_1 & 0 \\ 0 & 0 & 0 & 0 & 0 & S_1 & C_1 & 0 \\ 0 & 0 & 0 & 0 & S_3 & 0 & 0 & C_3 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2}C_6 & -\sqrt{2}S_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2}S_6 & \sqrt{2}C_6 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \tag{3.96}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix}
$$

From equation (3.96) it is very easy to derive a a flowgraph which is depicted in figure 3.7. Notice that the rotations simply changed their signs and the stages of the algorithm have just been mirrored horizontally.

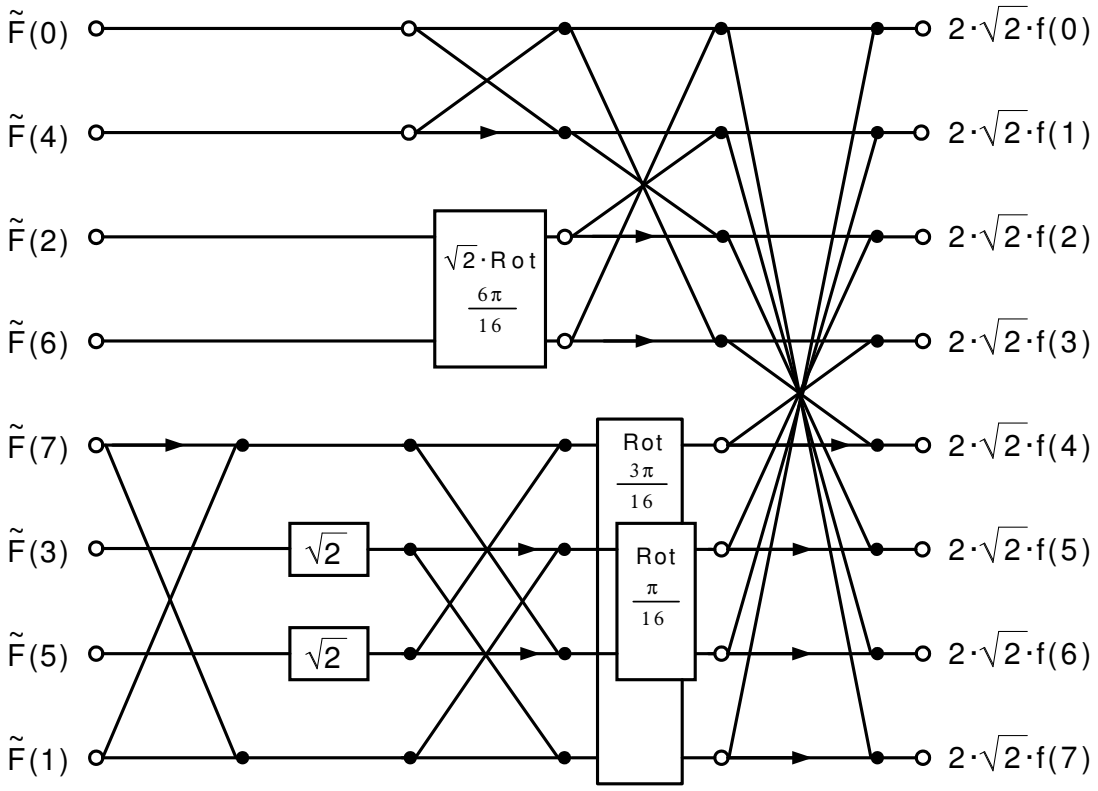

Figure 3.7: Inverse Loeffler-Ligtenberg-Moschytz Fast DCT

3.3.5 The Winograd 16-point "small-N" DFT

In [34], Shmuel Winograd presented his research work at the IBM Thomas J. Watson Research Center on the calculation of Fourier Transforms of arbitrary lengths[6]. His idea was to break up the sample length N of the Fourier Transform into its prime factors and calculate and combine DFTs of the size of the prime factors thus achieving a computational complexity of $O(N)$.[7] For this purpose, Winograd derived the so-called "small-N" DFTs of the prime sizes 2, 3, 5, 7, and for reasons of convenience, the "small-N" DFTs of the composite sizes 4, 8, 9 and 16. Applying the results from section 3.2.3 to the 16-point Winograd "small-N DFT" leads directly to the Arai-Agui-Nakajima DCT which will be covered in section 3.3.6. A general introduction into the Winograd DFT can also be found in [27].

The DFT from equation (3.29)[8] can be written as a matrix-vector product in the following form:

$$\begin{pmatrix} F(0) \\ F(1) \\ F(2) \\ \vdots \\ F(N-1) \end{pmatrix} = M \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(N-1) \end{pmatrix} \tag{3.97}$$

For $N = 16$ and $W = e^{-j\pi/8}$ the matrix M is defined as:

$$M = \begin{pmatrix}
W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 \\
W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 & -W^0 & -W^1 & -W^2 & -W^3 & -W^4 & -W^5 & -W^6 & -W^7 \\
W^0 & W^2 & W^4 & W^6 & -W^0 & -W^2 & -W^4 & -W^6 & W^0 & W^2 & W^4 & W^6 & -W^0 & -W^2 & -W^4 & -W^6 \\
W^0 & W^3 & W^6 & -W^1 & -W^4 & -W^7 & W^2 & W^5 & -W^0 & -W^3 & -W^6 & W^1 & W^4 & W^7 & -W^2 & -W^5 \\
W^0 & W^4 & -W^0 & -W^4 & W^0 & W^4 & -W^0 & -W^4 & W^0 & W^4 & -W^0 & -W^4 & W^0 & W^4 & -W^0 & -W^4 \\
W^0 & W^5 & -W^2 & -W^7 & W^4 & -W^1 & -W^6 & W^3 & -W^0 & -W^5 & W^2 & W^7 & -W^4 & W^1 & W^6 & -W^3 \\
W^0 & W^6 & -W^4 & W^2 & -W^0 & -W^6 & W^4 & -W^2 & W^0 & W^6 & -W^4 & W^2 & -W^0 & -W^6 & W^4 & -W^2 \\
W^0 & W^7 & -W^6 & W^5 & -W^4 & W^3 & -W^2 & W^1 & -W^0 & -W^7 & W^6 & -W^5 & W^4 & -W^3 & W^2 & -W^1 \\
W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 \\
W^0 & -W^1 & W^2 & -W^3 & W^4 & -W^5 & W^6 & -W^7 & -W^0 & W^1 & -W^2 & W^3 & -W^4 & W^5 & -W^6 & W^7 \\
W^0 & -W^2 & W^4 & -W^6 & -W^0 & W^2 & -W^4 & W^6 & W^0 & -W^2 & W^4 & -W^6 & -W^0 & W^2 & -W^4 & W^6 \\
W^0 & -W^3 & W^6 & W^1 & -W^4 & W^7 & W^2 & -W^5 & -W^0 & W^3 & -W^6 & -W^1 & W^4 & -W^7 & -W^2 & W^5 \\
W^0 & -W^4 & -W^0 & W^4 & W^0 & -W^4 & -W^0 & W^4 & W^0 & -W^4 & -W^0 & W^4 & W^0 & -W^4 & -W^0 & W^4 \\
W^0 & -W^5 & -W^2 & W^7 & W^4 & W^1 & -W^6 & -W^3 & -W^0 & W^5 & W^2 & -W^7 & -W^4 & -W^1 & W^6 & W^3 \\
W^0 & -W^6 & -W^4 & -W^2 & -W^0 & W^6 & W^4 & W^2 & W^0 & -W^6 & -W^4 & -W^2 & -W^0 & W^6 & W^4 & W^2 \\
W^0 & -W^7 & -W^6 & -W^5 & -W^4 & -W^3 & -W^2 & -W^1 & -W^0 & W^7 & W^6 & W^5 & W^4 & W^3 & W^2 & W^1
\end{pmatrix} \tag{3.98}$$

We will now index the rows of M in equation (3.98) from 0 to 15. Let P_1 be the permutation matrix reordering the rows of the matrix in (3.98) as follows: 0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15. Let I_n denote the $n \times n$ identity matrix and define $R_1 = L \otimes I_8$ with

[6] The classic Cooley-Tukey algorithm requires the number of samples to be a power of two.

[7] Cooley-Tukey style DFTs have a computational complexity of $O(N \log N)$.

[8] Note that in both [34] and [27], the Fourier Transform is defined as $F(u) = \sum_{n=0}^{K-1} f(n) W_K^{un}, \quad u = 0, \ldots, K-1$ with $W_K = e^{j\frac{2\pi}{K}}$, therefore the sign of the imaginary part must be reversed when applying their results.

$L = \left(\begin{smallmatrix} 1 & 1 \\ 1 & -1 \end{smallmatrix}\right)$.[9] M can then be written as follows:

$$M = P_1 \cdot \begin{pmatrix} W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & W^0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & W^2 & W^4 & W^6 & -W^0 & -W^2 & -W^4 & -W^6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & W^4 & -W^0 & -W^4 & W^0 & W^4 & -W^0 & -W^4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & W^6 & -W^4 & W^2 & -W^0 & -W^6 & W^4 & -W^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & W^0 & -W^0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & -W^2 & W^4 & -W^6 & -W^0 & W^2 & -W^4 & W^6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & -W^4 & -W^0 & W^4 & W^0 & -W^4 & -W^0 & W^4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ W^0 & -W^6 & -W^4 & -W^2 & -W^0 & W^6 & W^4 & W^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^3 & W^6 & -W^1 & -W^4 & -W^7 & W^2 & W^5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^5 & -W^2 & -W^7 & W^4 & -W^1 & -W^6 & W^3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^7 & -W^6 & W^5 & -W^4 & W^3 & -W^2 & W^1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^1 & W^2 & -W^3 & W^4 & -W^5 & W^6 & -W^7 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^3 & W^6 & W^1 & -W^4 & W^7 & W^2 & -W^5 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^5 & -W^2 & W^7 & W^4 & W^1 & -W^6 & -W^3 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^7 & -W^6 & -W^5 & -W^4 & -W^3 & -W^2 & -W^1 \end{pmatrix} \cdot R_1 \tag{3.99}$$

The permutation matrix P_1 is defined as:

$$P_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.100}$$

[9] If A is the $m \times n$ matrix $(a_{i,j})$ and B is the $p \times q$ matrix $(b_{i,j})$, then the tensor product $A \otimes B$ is the $mp \times nq$ matrix composed of the $m \times n$ blocks $(a_{i,j}B)$.

The Matrix $R_1 = L \otimes I_8$ is defined as:

$$R_1 = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1
\end{pmatrix} \tag{3.101}$$

We will now reorder the first eight rows of the matrix in equation (3.99). Let P_2 be the permutation matrix reordering the first eight rows of the matrix in (3.99) as follows: 0, 2, 4, 6, 1, 3, 5, 7. We define the matrix R_2 which is a 16×16 identity matrix with the upper left 8×8 block replaced by $L \otimes I_4$. The matrix M can then be written as follows:

$$\begin{aligned}
M &= P_1 \cdot P_2 \\
&\cdot \begin{pmatrix}
W^0 & W^0 & W^0 & W^0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
W^0 & W^4 & -W^0 & -W^4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
W^0 & -W^0 & W^0 & -W^0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
W^0 & -W^4 & -W^0 & W^4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & W^0 & W^2 & W^4 & W^6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & W^0 & W^6 & -W^4 & W^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & W^0 & -W^2 & W^4 & -W^6 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & W^0 & -W^6 & -W^4 & -W^2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^3 & W^6 & -W^1 & -W^4 & -W^7 & W^2 & W^5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^5 & -W^2 & -W^7 & W^4 & -W^1 & -W^6 & W^3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & W^7 & -W^6 & W^5 & -W^4 & W^3 & -W^2 & W^1 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^1 & W^2 & -W^3 & W^4 & -W^5 & W^6 & -W^7 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^3 & W^6 & W^1 & -W^4 & W^7 & W^2 & -W^5 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^5 & -W^2 & W^7 & W^4 & W^1 & -W^6 & -W^3 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & W^0 & -W^7 & -W^6 & -W^5 & -W^4 & -W^3 & -W^2 & -W^1
\end{pmatrix} \\
&\cdot \quad R_2 \cdot R_1
\end{aligned} \tag{3.102}$$

The permutation matrix P_2 is defined as:

$$P_2 = \begin{pmatrix}
1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\
0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\
0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 1
\end{pmatrix} \tag{3.103}$$

The Matrix R_2 is defined as:

$$R_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 & \cdots & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & 1 \end{pmatrix} \tag{3.104}$$

For reasons of notational simplicity, we define a shortcut for the lower right half of the matrix in equations (3.99) and (3.102):

$$G_8 = \begin{pmatrix} W^0 & W^1 & W^2 & W^3 & W^4 & W^5 & W^6 & W^7 \\ W^0 & W^3 & W^6 & -W^1 & -W^4 & -W^7 & W^2 & W^5 \\ W^0 & W^5 & -W^2 & -W^7 & W^4 & -W^1 & -W^6 & W^3 \\ W^0 & W^7 & -W^6 & W^5 & -W^4 & W^3 & -W^2 & W^1 \\ W^0 & -W^1 & W^2 & -W^3 & W^4 & -W^5 & W^6 & -W^7 \\ W^0 & -W^3 & W^6 & W^1 & -W^4 & W^7 & W^2 & -W^5 \\ W^0 & -W^5 & -W^2 & W^7 & W^4 & W^1 & -W^6 & -W^3 \\ W^0 & -W^7 & -W^6 & -W^5 & -W^4 & -W^3 & -W^2 & -W^1 \end{pmatrix} \tag{3.105}$$

We will now reorder the first four rows of the matrix in equation (3.102). Let P_3 be the permutation matrix reordering the first four rows of the matrix in equation (3.102) as follows: 0, 2, 1, 3. We define the matrix R_3 which is a 16×16 identity matrix with the upper left 4×4 block replaced by $L \otimes I_2$. The matrix M can then be written as follows:

$$M = P_1 \cdot P_2 \cdot P_3 \cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -j & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & j & 0 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}}(1-j) & -j & -\frac{1}{\sqrt{2}}(1+j) & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & -\frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(1-j) & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}}(j-1) & -j & \frac{1}{\sqrt{2}}(1+j) & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(j-1) & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & G_8 & & \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & & & \end{pmatrix} \cdot R_3 \cdot R_2 \cdot R_1 \tag{3.106}$$

The permutation matrix P_3 is defined as:

$$P_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & & 1 \end{pmatrix} \tag{3.107}$$

The Matrix $R_3 = L \otimes I_2$ is defined as:

$$R_3 = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 1 & 0 & \cdots & 0 \\ 1 & 0 & -1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & -1 & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & 0 & 0 & & 1 \end{pmatrix} \tag{3.108}$$

In order to diagonalise the upper left 2×2 block in equation (3.106) we use matrix R_4 which is a 16×16 identity matrix with the upper left 2×2 block replaced by L:

$$R_4 = \begin{pmatrix} 1 & 1 & 0 & \cdots & 0 \\ 1 & -1 & 0 & \cdots & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \\ 0 & 0 & 0 & & 1 \end{pmatrix} \tag{3.109}$$

The product $P_1 \cdot P_2 \cdot P_3$ of all permutation matrices can now be combined into one single permutation matrix:

$$P = P_1 \cdot P_2 \cdot P_3 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.110}$$

Again, we define a shortcut:

$$G_4 = \begin{pmatrix} 1 & \frac{1}{\sqrt{2}}(1-j) & -j & -\frac{1}{\sqrt{2}}(1+j) \\ 1 & -\frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(1-j) \\ 1 & \frac{1}{\sqrt{2}}(j-1) & -j & \frac{1}{\sqrt{2}}(1+j) \\ 1 & \frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(j-1) \end{pmatrix} \tag{3.111}$$

Now matrix M can be written as follows:

$$M = P \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -j & 0 & \cdots & 0 \\ 0 & 0 & 1 & j & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & G_4 & & \\ \vdots & \vdots & \vdots & \vdots & & G_8 & \\ 0 & 0 & 0 & 0 & & & \end{pmatrix} \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1 \tag{3.112}$$

We will now look at the matrices G_4 and G_8 separately: Matrix G_4 can be factorized as follows:

$$\begin{aligned} G_4 &= \begin{pmatrix} 1 & \frac{1}{\sqrt{2}}(1-j) & -j & -\frac{1}{\sqrt{2}}(1+j) \\ 1 & -\frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(1-j) \\ 1 & \frac{1}{\sqrt{2}}(j-1) & -j & \frac{1}{\sqrt{2}}(1+j) \\ 1 & \frac{1}{\sqrt{2}}(1+j) & j & \frac{1}{\sqrt{2}}(j-1) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 1 & 0 \\ 1 & 0 & -1 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & -j & 0 \\ 0 & -\frac{1}{\sqrt{2}}j & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \end{pmatrix} \end{aligned} \tag{3.113}$$

With $\gamma = \frac{\pi}{8}$, matrix G_8 from equation (3.105) can be factorized as follows:

$$\begin{aligned} G_8 = & \\ & \begin{pmatrix} 1 & \cos\gamma - j\sin\gamma & \cos 2\gamma - j\sin 2\gamma & \cos 3\gamma - j\sin 3\gamma & -j & -\cos 3\gamma - j\sin 3\gamma & -\cos 2\gamma - j\sin 2\gamma & -\cos\gamma - j\sin\gamma \\ 1 & \cos 3\gamma - j\sin 3\gamma & -\cos 2\gamma - j\sin 2\gamma & -\cos\gamma + j\sin\gamma & j & \cos\gamma + j\sin\gamma & \cos 2\gamma - j\sin 2\gamma & -\cos 3\gamma - j\sin 3\gamma \\ 1 & -\cos 3\gamma - j\sin 3\gamma & -\cos 2\gamma + j\sin 2\gamma & \cos\gamma + j\sin\gamma & -j & -\cos\gamma + j\sin\gamma & \cos 2\gamma + j\sin 2\gamma & \cos 3\gamma - j\sin 3\gamma \\ 1 & -\cos\gamma - j\sin\gamma & \cos 2\gamma + j\sin 2\gamma & -\cos 3\gamma - j\sin 3\gamma & j & \cos 3\gamma - j\sin 3\gamma & -\cos 2\gamma + j\sin 2\gamma & \cos\gamma - j\sin\gamma \\ 1 & -\cos\gamma + j\sin\gamma & \cos 2\gamma - j\sin 2\gamma & -\cos 3\gamma + j\sin 3\gamma & -j & \cos 3\gamma + j\sin 3\gamma & -\cos 2\gamma - j\sin 2\gamma & \cos\gamma + j\sin\gamma \\ 1 & -\cos 3\gamma + j\sin 3\gamma & -\cos 2\gamma - j\sin 2\gamma & \cos\gamma - j\sin\gamma & j & -\cos\gamma - j\sin\gamma & \cos 2\gamma - j\sin 2\gamma & \cos 3\gamma + j\sin 3\gamma \\ 1 & \cos 3\gamma + j\sin 3\gamma & -\cos 2\gamma + j\sin 2\gamma & -\cos\gamma - j\sin\gamma & -j & \cos\gamma - j\sin\gamma & \cos 2\gamma + j\sin 2\gamma & -\cos 3\gamma + j\sin 3\gamma \\ 1 & \cos\gamma + j\sin\gamma & \cos 2\gamma + j\sin 2\gamma & \cos 3\gamma + j\sin 3\gamma & j & -\cos 3\gamma + j\sin 3\gamma & -\cos 2\gamma + j\sin 2\gamma & -\cos\gamma + j\sin\gamma \end{pmatrix} \\ & = T_1 \cdot T_2 \cdot T_3 \cdot T_4 \cdot T_5 \cdot T_6 \cdot T_7 \end{aligned} \tag{3.114}$$

The matrices T_1, T_2, T_3, T_4, T_5, T_6 and T_7 are as follows:

$$T_1 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \end{pmatrix} \tag{3.115}$$

$$T_2 = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{pmatrix} \tag{3.116}$$

$$T_3 = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix} \tag{3.117}$$

$$T_4 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -j & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -j\sin 2\gamma & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \cos 2\gamma & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -j\sin 3\gamma & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & j(\sin 3\gamma - \sin\gamma) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -j(\sin 3\gamma + \sin\gamma) & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cos 3\gamma & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cos\gamma + \cos 3\gamma & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & \cos 3\gamma - \cos\gamma \end{pmatrix} \tag{3.118}$$

$$T_5 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.119}$$

$$T_6 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \end{pmatrix} \tag{3.120}$$

$$T_7 = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.121}$$

The number of operations that are needed to calculate the 16-point Winograd "small-N" DFT can now easily be obtained by looking at the rows of the matrices in equations (3.101), (3.104), (3.108), (3.109), (3.112), (3.113), (3.115), (3.116), (3.117), (3.119) and (3.120): Each row that contains more than one ± 1, is an addition, 74 in total. The number of multiplications is 10, with 8 multiplications originating from equation (3.118) and 2 multiplications from equation (3.113).

3.3.6 The Arai-Agui-Nakajima-DCT

The Arai-Agui-Nakajima DCT ([35]) is the most efficient one-dimensional DCT known up to now for DCT coefficients that need to be quantized[10]. This algorithm actually uses 13 multiplications, but eight of them can be absorbed into the quantization values[11]. This means that for DCT applications such as JPEG, where a quantization needs to be calculated anyway, moving 8 of the 13 multiplications into the quantization coefficients reduces this algorithm's complexity to only 5 multiplications and 29 additions for an 8 point DCT. In this section we will derive the Arai-Agui-Nakajima-DCT from the Winograd 16-point "small-N" DFT that was presented in section 3.3.5 and from the results of section 3.2.3, where we used a 2N sample symmetrical input vector to derive an N-point DCT from a 2N-point DFT. In order to use only 8 samples as the input vector to the Winograd 16-point "small-N" DFT and satisfy condition (3.30), we have to multiply matrix M in (3.112) with another matrix U. Let M' be the resulting matrix from multiplying matrix M in equation (3.112) with U:

$$M' = M \cdot U \tag{3.122}$$

[10] This is the case for the DCT in JPEG encoding and decoding, see section 2.2.2.

[11] For JPEG, the quantization values remain the same for each color component of an image during the whole encoding or decoding process, see section 2.4.

Matrix U is defined as follows:

$$U = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.123}$$

Matrix U can easily be absorbed into matrix R_1 from equation (3.101) which leads to matrix R_1':

$$R_1' = R_1 \cdot U = M \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.124}$$

Now matrix M' can be written as follows (compare with equation (3.112)):

$$M' = P \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & 0 & 0 & \cdots & 0 \\ 0 & 0 & 1 & -j & 0 & \cdots & 0 \\ 0 & 0 & 1 & j & 0 & \cdots & 0 \\ 0 & 0 & 0 & 0 & G_4 & & \\ \vdots & \vdots & \vdots & \vdots & & G_8 & \\ 0 & 0 & 0 & 0 & & & \end{pmatrix} \cdot R_4 \cdot R_3 \cdot R_2 \cdot R_1' \tag{3.125}$$

From section 3.2.3 we know that another interesting property of applying a DFT on a real and symmetrical input vector is the fact, that the imaginary part of the result vector is zero. Since there are no multiplication paths in equation (3.125) that multiply a complex term with another complex term, we can therefore simply disregard the imaginary part of all terms from now on and replace them with zero.

The derivation of the Arai-Agui-Nakajima-DCT from the Winograd 16-point "small-N" DFT can probably be best described by dividing the result vector into two parts, the first part containing all even indices and the second part containing all the odd indices. By looking at the permutation matrix P in equation (3.110), we find that the indices 0, 2, 4 and 6 are built from lines 0, 4, 2 and 5 in the matrix in equation (3.125), therefore only the top left 8×8 block is of any interest to us. From equation (3.125) we can now deduce the following:

$$\begin{aligned}
\begin{pmatrix} F(0) \\ F(2) \\ F(4) \\ F(6) \end{pmatrix} &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 & 1 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \\
&\cdot \quad R'_4 \cdot R'_3 \cdot R'_2 \cdot R''_1 \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix} \\
&= P' \cdot Q \cdot R'_4 \cdot R'_3 \cdot R'_2 \cdot R''_1 \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix}
\end{aligned} \tag{3.126}$$

The matrices P', Q and R''_1 are defined as follows:

$$P' = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.127}$$

$$Q = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & \frac{1}{\sqrt{2}} & 0 & -\frac{1}{\sqrt{2}} \\ 0 & 0 & 0 & 0 & 1 & -\frac{1}{\sqrt{2}} & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \tag{3.128}$$

$$R_1'' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.129}$$

The matrices R_2', R_3' and R_4' are simply the the top left 8×8 blocks of the matrices R_2, R_3 and R_4 from equations (3.104), (3.108) and (3.109):

$$R_2' = \begin{pmatrix} 1 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & -1 \end{pmatrix} \tag{3.130}$$

$$R_3' = \begin{pmatrix} 1 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.131}$$

$$R_4' = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.132}$$

It can be easily shown that matrix Q can be factorized as follows:

$$\begin{aligned} Q &= \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \\ &\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \end{pmatrix} \end{aligned} \tag{3.133}$$

Our goal will now be to simplify matrices Q, R_1'', R_2', R_3' and R_4' as much as possible, taking advantage of the symmetry of the matrix that follows each of these matrices in the matrix product in equation (3.126): Because rows 0 and 7 (as well as 1 and 6, 2 and 5, 3 and 4) in matrix R_1'' in equation are identical, R_2' can be replaced by R_2'':

$$R_2'' = \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ -1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.134}$$

Doing so, we can also replace the complete lower half of R_1'' with zeroes.

Similarly, we can make use of the symmetry in R_2'' in equation (3.134): Rows 0 and 3 are identical, also 1 and 2. Row 7 is just the inverse of row 4 and row 6 is the inverse of row 5. Therefore, R_3' can be replaced by R_3'':

$$R_3'' = \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ -1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix} \tag{3.135}$$

Doing so, we can also replace rows 2, 3, 6 and 7 of R_2'' with zeroes.

Exploiting the symmetries in R_3'' (rows 0 and 1 are identical, rows 2 and 3, 4 and 7, 5 and 6 are just their own inverses), we can now replace R_4' with R_4'':

$$R_4'' = \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 0 \end{pmatrix} \tag{3.136}$$

Doing so, we can also replace rows 1, 3, 6 and 7 of R_3'' with zeroes.

As a last optimization we make use of the symmetry in R_4'', where row 4 and 7 are their inverses. We can then change row 5 of the rightmost matrix of the factorization in equation (3.133) from $(\,0\,0\,0\,0\,0\,1\,0\,-1\,)$ to $(\,0\,0\,0\,0\,1\,1\,0\,0\,)$. Deleting columns with all zeroes and their associated rows in their respective following matrix, we can now reduce the dimensions of the matrices and calculate the even coefficients of the Arai-Agai-Nakajima Fast DCT as follows:

$$\begin{pmatrix} F(0) \\ F(2) \\ F(4) \\ F(6) \end{pmatrix} = \begin{pmatrix} 2 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & \frac{1}{\sqrt{2}} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$$

$$\cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 1 & 0 & 0 & -1 \\ 0 & 1 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix} \tag{3.137}$$

From equation (3.137) we can now easily derive a flowgraph for the even coefficients of the Arai-Agui-Nakajima Fast DCT as depicted in figure 3.8.

For the vector with the odd indices, we again look at the permutation matrix P in equation (3.110). We find that the indices 1, 3, 5 and 7 are built from lines 8, 9, 10 and 11 in the matrix in equation (3.125), therefore only the bottom right 8×8 block is of any interest to us. Fortunately, the bottom right 8×8 block of the matrices R_4', R_3' and R_2' are the 8×8 identity matrices, so we can disregard them. Therefore we can deduce the following from equation (3.125):

$$\begin{aligned} \begin{pmatrix} F(1) \\ F(3) \\ F(5) \\ F(7) \end{pmatrix} &= \begin{pmatrix} 1 & \cos\gamma & \cos 2\gamma & \cos 3\gamma & 0 & -\cos 3\gamma & -\cos 2\gamma & -cos\gamma \\ 1 & \cos 3\gamma & -\cos 2\gamma & -\cos\gamma & 0 & \cos\gamma & \cos 2\gamma & -cos3\gamma \\ 1 & -\cos 3\gamma & -\cos 2\gamma & \cos\gamma & 0 & -\cos\gamma & \cos 2\gamma & cos3\gamma \\ 1 & -\cos\gamma & \cos 2\gamma & -\cos 3\gamma & 0 & \cos 3\gamma & -\cos 2\gamma & cos\gamma \end{pmatrix} \cdot R_1''' \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix} \\ &= V \cdot R_1''' \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix} \end{aligned} \tag{3.138}$$

The matrix V is defined as:

$$V = \begin{pmatrix} 1 & \cos\gamma & \cos 2\gamma & \cos 3\gamma & 0 & -\cos 3\gamma & -\cos 2\gamma & -cos\gamma \\ 1 & \cos 3\gamma & -\cos 2\gamma & -\cos\gamma & 0 & \cos\gamma & \cos 2\gamma & -cos3\gamma \\ 1 & -\cos 3\gamma & -\cos 2\gamma & \cos\gamma & 0 & -\cos\gamma & \cos 2\gamma & cos3\gamma \\ 1 & -\cos\gamma & \cos 2\gamma & -\cos 3\gamma & 0 & \cos 3\gamma & -\cos 2\gamma & cos\gamma \end{pmatrix} \tag{3.139}$$

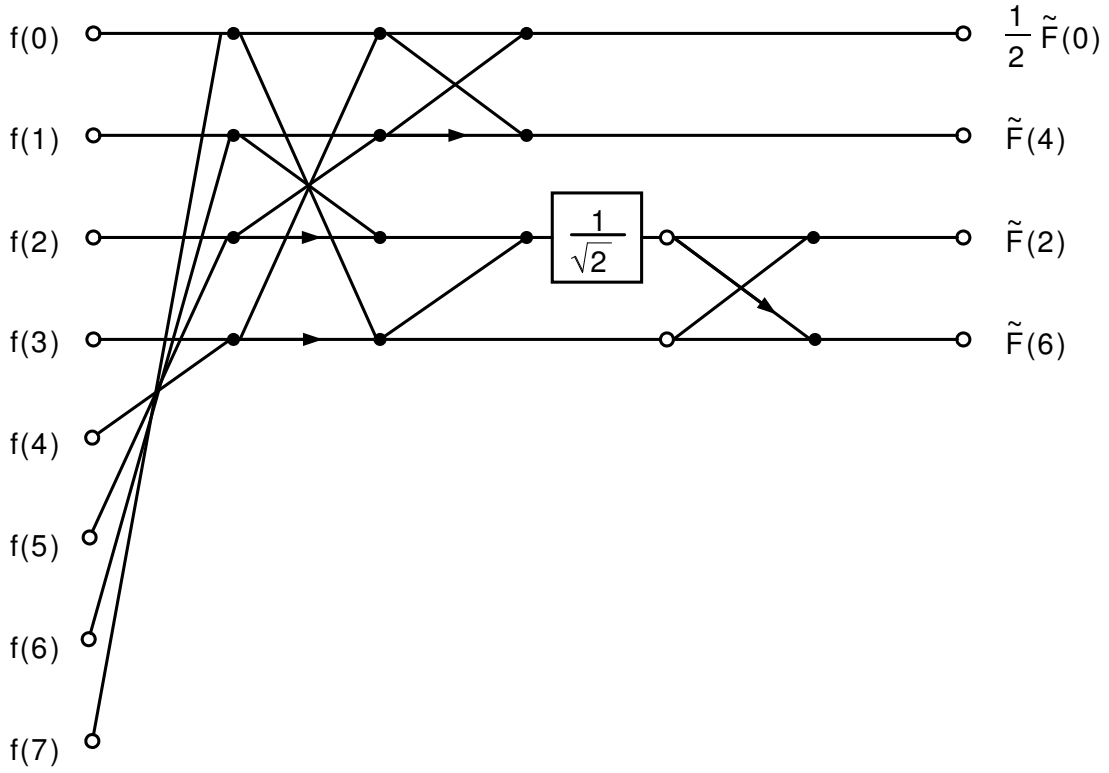

Figure 3.8: Flowgraph for the even coefficients of the Arai-Agui-Nakajima Fast DCT

Matrix R_1''' is simply the lower half of matrix R_1' from equation (3.124):

$$R_1''' = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 & 1 & 0 & 0 & 0 \\ 0 & 0 & -1 & 0 & 0 & 1 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 0 & 1 & 0 \\ -1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.140}$$

Matrix V can be factorized as follows:

$$\begin{aligned} V &= \begin{pmatrix} 1 & \cos\gamma & \cos 2\gamma & \cos 3\gamma & 0 & -\cos 3\gamma & -\cos 2\gamma & -cos\gamma \\ 1 & \cos 3\gamma & -\cos 2\gamma & -\cos\gamma & 0 & \cos\gamma & \cos 2\gamma & -cos3\gamma \\ 1 & -\cos 3\gamma & -\cos 2\gamma & \cos\gamma & 0 & -\cos\gamma & \cos 2\gamma & cos3\gamma \\ 1 & -\cos\gamma & \cos 2\gamma & -\cos 3\gamma & 0 & \cos 3\gamma & -\cos 2\gamma & cos\gamma \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix} \\ &\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos 2\gamma & 0 & 0 & 0 \\ 0 & 0 & -\cos 3\gamma & 0 & 0 \\ 0 & 0 & 0 & \cos\gamma + \cos 3\gamma & 0 \\ 0 & 0 & 0 & 0 & \cos 3\gamma - \cos\gamma \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\ &\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 \end{pmatrix} \end{aligned} \tag{3.141}$$

If we now take advantage of the symmetry in matrix R_1''', the right half of the last matrix in equation (3.141) can be inverted and mirrored to the left half and the odd indices of the

Arai-Agui-Nakajima Fast DCT can be computed as follows:

$$\begin{pmatrix} F(1) \\ F(3) \\ F(5) \\ F(7) \end{pmatrix} = \begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & -1 \\ 0 & 1 & 0 & 1 \\ 1 & 0 & -1 & 0 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 1 & 0 & 1 \end{pmatrix}$$
$$\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & \cos 2\gamma & 0 & 0 & 0 \\ 0 & 0 & -\cos 3\gamma & 0 & 0 \\ 0 & 0 & 0 & \cos\gamma + \cos 3\gamma & 0 \\ 0 & 0 & 0 & 0 & \cos 3\gamma - \cos\gamma \end{pmatrix}$$
$$\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 \\ 0 & 0 & -1 & -1 \end{pmatrix}$$
$$\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \end{pmatrix} \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ \vdots \\ f(7) \end{pmatrix} \tag{3.142}$$

From equation (3.142) we can now easily derive a flowgraph for the odd coefficients of the Arai-Agui-Nakajima Fast DCT as depicted in figure 3.9. Figure 3.10 shows the complete flowgraph as a combination of figure 3.8 and figure 3.9. From the flowgraph in figure 3.10, which can also be found in [35], [22] and [15], we can now determine the computational complexity of the Arai-Agui-Nakajima Fast DCT as consisting of 5 multiplications and 29 additions. Note however, that the scaling of the output vector in [15] (which is merely an adaption of [22] with respect to the the Arai-Agui-Nakajima-DCT) is incorrect. The correct values for the scaling factors m_n of the Arai-Agui-Nakajima-DCT are:

$$m_n = 4 \cdot c(n) \cdot \cos\frac{n\pi}{16} = 4 \cdot c(n) \cdot C_n \quad , n = 0, \ldots, 7 \tag{3.143}$$

with $c(n)$ defined according to equation (3.3) and C_n defined according to equation (3.38).

The factors $a_1, \ldots, a_5$ are defined as follows:

$$\begin{aligned} a_1 &= a_3 = C_4 = \frac{1}{\sqrt{2}} \\ a_2 &= \cos\frac{\pi}{8} - \cos\frac{3\pi}{8} = \sqrt{1 - \frac{1}{\sqrt{2}}} \\ a_4 &= \cos\frac{\pi}{8} + \cos\frac{3\pi}{8} = \sqrt{1 + \frac{1}{\sqrt{2}}} \\ a_5 &= \cos\frac{3\pi}{8} = \frac{1}{2}\sqrt{2 - \sqrt{2}} \end{aligned} \tag{3.144}$$

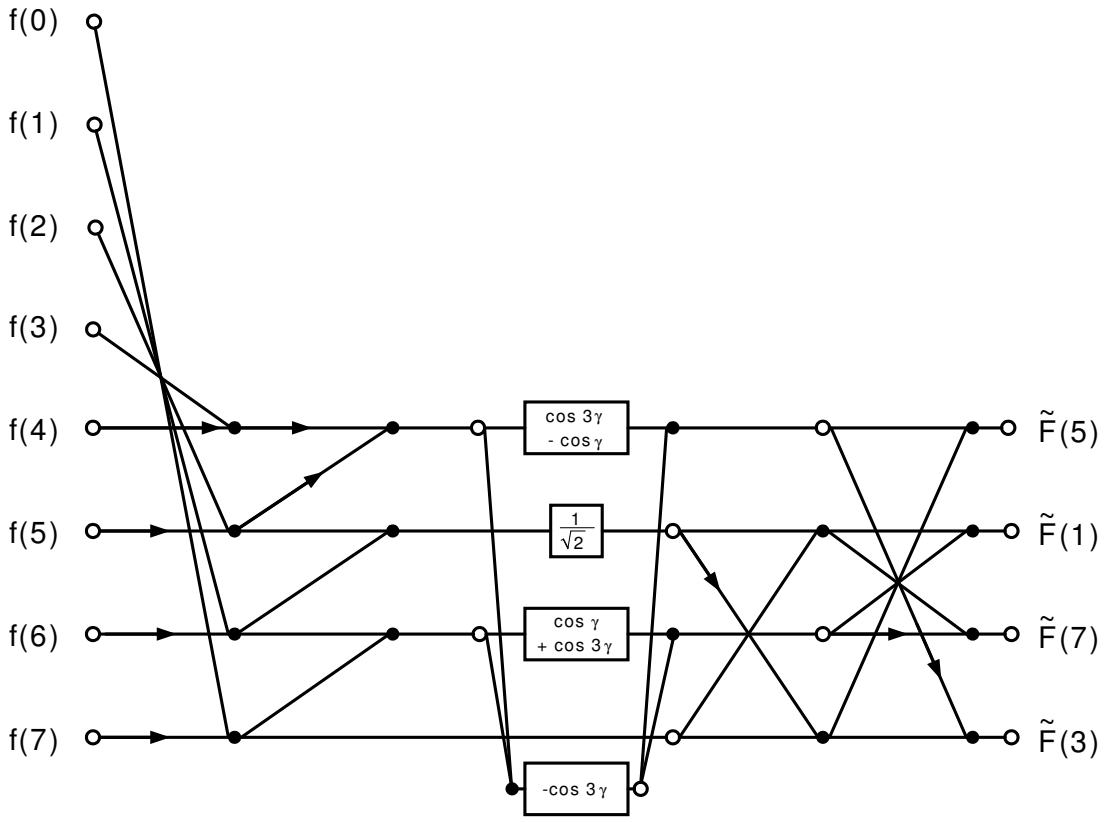

Figure 3.9: Flowgraph for the odd coefficients of the Arai-Agui-Nakajima Fast DCT

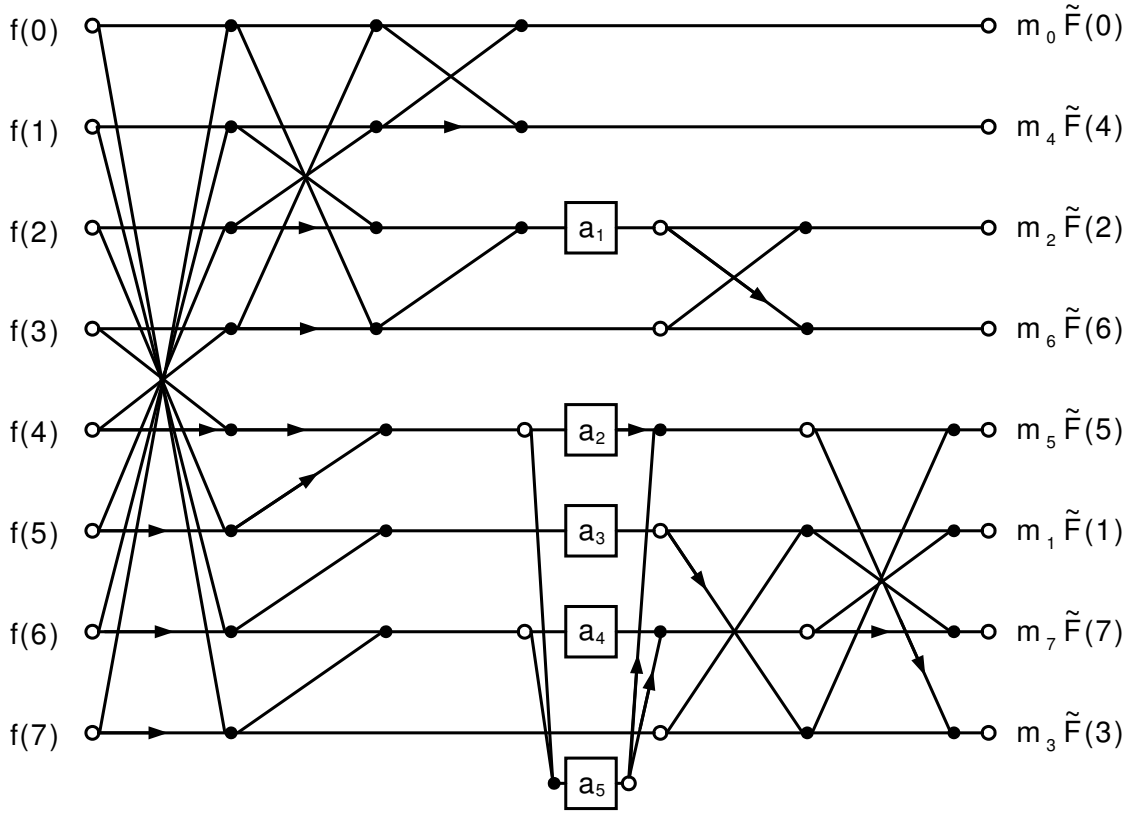

Figure 3.10: Flowgraph for the Arai-Agui-Nakajima Fast DCT

With the help of the flowgraph in figure 3.10 and equations (3.137) and (3.142) the Arai-Agui-Nakajima Fast DCT can be computed as follows:

$$
\begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} = \begin{pmatrix} \frac{1}{2\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4C_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4C_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4C_3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_6} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_7} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix} \tag{3.145}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\cos\frac{\pi}{8} & 0 & -\cos\frac{3\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & -\cos\frac{3\pi}{8} & 0 & \cos\frac{\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix}
$$

Note that the first matrix in equation (3.145) is a diagonal matrix containing the scaling values from equation (3.144). In a DCT application that uses quantization, such as JPEG, this matrix would be absorbed into the quantization values. The matrix immediately following the diagonal scaling matrix is a permutation matrix that brings the outcome of figure 3.10 into natural order. We will use the matrix-vector product representation of the Arai-Agui-Nakajima Fast DCT from equation (3.145) in subsequent sections on a fast two-dimensional DCT.

3.3.7 The Inverse Arai-Agui-Nakajima-DCT

From equation (3.145) it is now trivial to derive the inverse Arai-Agui-Nakajima Fast DCT: Simply the order of all matrices in this equation needs to be inverted and all matrices themselves as well:

$$
\begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} = \frac{1}{8} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\cos\frac{\pi}{8} & 0 & -2\cos\frac{3\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\cos\frac{3\pi}{8} & 0 & 2\cos\frac{\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} 2\sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 4C_1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 4C_2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4C_3 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 2\sqrt{2} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 4C_5 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 4C_6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 4C_7 \end{pmatrix}
$$

$$
\cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} \tag{3.146}
$$

Notice that the lower right half of the second matrix in equation (3.146) can be factored as follows:

$$\begin{pmatrix} -1 & -1 & 1 & -1 \\ 0 & 1 & -1 & 1 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} = \begin{pmatrix} -1 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 1 \end{pmatrix} \quad (3.147)$$

If we now consider that the lower right half of the fourth matrix in equation (3.146) contains a rotation of rows 4 and 6 of the following matrix which can be written as $\begin{pmatrix} -2\cos\frac{\pi}{8} & -2\cos\frac{3\pi}{8} \\ -2\cos\frac{3\pi}{8} & 2\cos\frac{\pi}{8} \end{pmatrix} = \begin{pmatrix} -1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2\cos\frac{\pi}{8} & 0 & 0 \\ 0 & 2\left(\cos\frac{\pi}{8}-\cos\frac{3\pi}{8}\right) & 0 \\ 0 & 0 & -2\left(\cos\frac{\pi}{8}+\cos\frac{3\pi}{8}\right) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 0 & 1 \\ 1 & 0 \end{pmatrix}$, we are able to derive a flowgraph for the inverse Arai-Agui-Nakajima Fast DCT as depicted in figure 3.11.

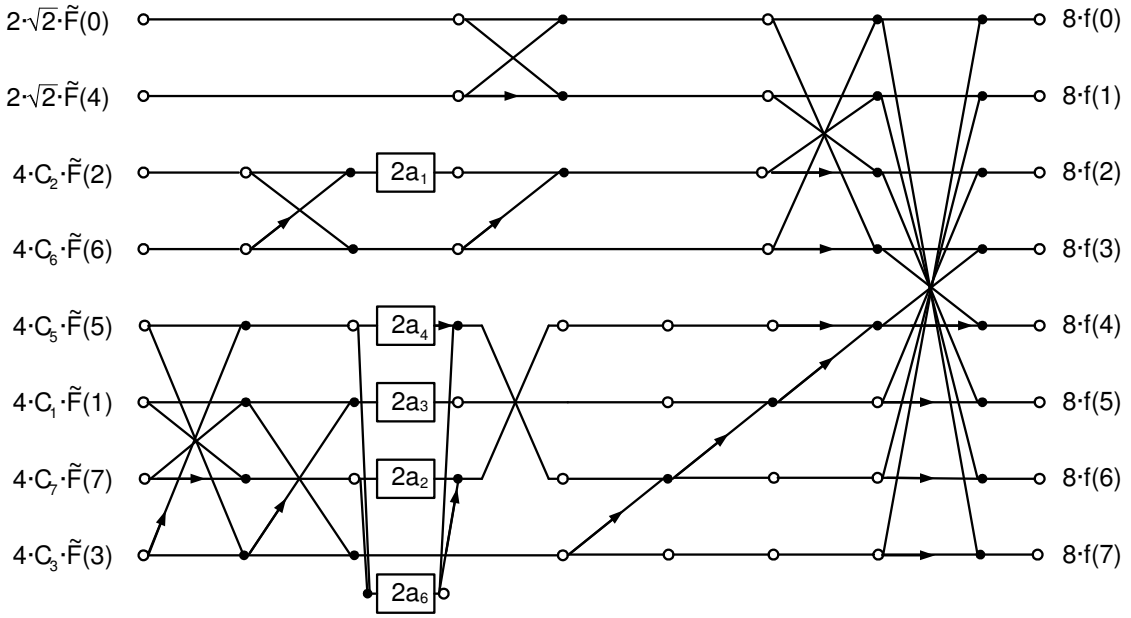

Figure 3.11: Flowgraph for the inverse Arai-Agui-Nakajima Fast DCT

Figure 3.11 uses a new constant besides from the ones in equation (3.144):

$$a_6 = \cos\frac{\pi}{8} \quad (3.148)$$

3.4 Fast two-dimensional DCTs

In this section we are going to develop a fast two-dimensional 8-point DCT from the Arai-Agui-Nakajima DCT in section 3.3.6. We will show that the two-dimensional DCT is actually the tensor product of the DCT matrix with itself. Therefore the tensor product and its most important properties will first be covered in more detail.

3.4.1 The tensor product and its properties

In section 3.3.5 we already introduced the tensor product. A good introduction on this topic can also be found in [13]. The tensor product is a binary matrix operator that provides a mechanism for combining two matrices to form one single larger matrix. Let A_{n_1,n_2} and B_{m_1,m_2} be two arbitrary matrices of dimension $n_1 \times n_2$ and $m_1 \times m_2$, respectively:

$$A_{n_1,n_2} = \begin{pmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,n_2-1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,n_2-1} \\ \vdots & \vdots & & \vdots \\ a_{n_1-1,0} & a_{n_1-1,1} & \dots & a_{n_1-1,n_2-1} \end{pmatrix} \tag{3.149}$$

$$B_{m_1,m_2} = \begin{pmatrix} b_{0,0} & b_{0,1} & \dots & b_{0,m_2-1} \\ b_{1,0} & b_{1,1} & \dots & b_{1,m_2-1} \\ \vdots & \vdots & & \vdots \\ b_{m_1-1,0} & b_{m_1-1,1} & \dots & b_{m_1-1,m_2-1} \end{pmatrix} \tag{3.150}$$

The tensor product $C = A \otimes B$ is defined as the $n_1 m_1 \times n_2 m_2$ matrix given by:

$$C = A_{n_1,n_2} \otimes B_{m_1,m_2} = \begin{pmatrix} a_{0,0} \cdot B & a_{0,1} \cdot B & \dots & a_{0,n_2-1} \cdot B \\ a_{1,0} \cdot B & a_{1,1} \cdot B & \dots & a_{1,n_2-1} \cdot B \\ \vdots & \vdots & & \vdots \\ a_{n_1-1,0} \cdot B & a_{n_1-1,1} \cdot B & \dots & a_{n_1-1,n_2-1} \cdot B \end{pmatrix} \tag{3.151}$$

This means that the tensor product of A and B is formed by replacing an arbitrary element a_{ij} of A with the $m_1 \times m_2$ matrix $a_{ij} \cdot B$.

There are quite a few interesting properties of the tensor product, for instance, it is associative:

$$A \otimes (B \otimes C) = (A \otimes B) \otimes C \tag{3.152}$$

If the involved matrices are of appropriate dimension, the tensor product is also distributive over matrix multiplication:

$$(A \otimes B) \cdot (C \otimes D) = AC \otimes BD \tag{3.153}$$

Another property is the inversion property: If A and B are nonsingular matrices, so is $C = A \otimes B$ and:

$$C^{-1} = (A \otimes B)^{-1} = A^{-1} \otimes B^{-1} \tag{3.154}$$

This means that if we want to calculate the inverse matrix of a large matrix C that can be expressed as a tensor product, the inverse can be much more easily calculated from an inversion of the factors A and B.

Another interesting property of the tensor product is the *transposition property*:

$$(A \otimes B)^t = A^t \otimes B^t \tag{3.155}$$

The last property to be examined will be needed quite often in the following sections. It allows us to reverse the order of operators in the tensor product with the help of permutation matrices: In section 3.3.5 we used the permutation matrix that permuted the rows of a vector or a matrix in the following order: 0, 2, 4, 6, 8, 10, 12, 14, 1, 3, 5, 7, 9, 11, 13, 15. This matrix is actually called a *stride-by-2* matrix. Generally, a *stride-by-s* matrix works like this: Let $X_n = \begin{pmatrix} x_0 \\ x_1 \\ \vdots \\ x_n \end{pmatrix}$ be an n-point vector and let $P_{n,s}$ denote the stride-by-s permutation matrix with $n = r \cdot s$. For the caculation of $Y_n = P_{n,s} \cdot X_n$, the first r elements of Y_n are obtained by starting at element x_0 and selecting each s^{th} element of X_n. The next r elements are obtained in the same manner, but this time starting at x_1. This process is repeated s times in total and the vector Y_n is built from the following elements: $x_0, x_s, x_{2s}, \ldots, x_{(r-1)s}, x_1, x_{s+1}, x_{2s+1}, \ldots, x_{(r-1)s+1}, \ldots, x_{s-1}, x_{2s-1}, \ldots, x_{rs-1}$. Now let A_s and B_r be any sqare matrices of order s and r, respectively. Then we can reverse the order of the tensor product with a stride-by-s and a stride-by-r permutation matrix:

$$(A_s \otimes B_r) = P_{n,s} \cdot (B_r \otimes A_s) \cdot P_{n,r} \tag{3.156}$$

3.4.2 The two-dimensional DCT as a tensor product

Let K_8 be the DCT matrix that is built from the product of all matrices in equation (3.145) such that equation (3.145) can be written as:

$$\begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} = K_8 \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} \tag{3.157}$$

For calculating a two-dimensional n-point DCT, we use an n × n input vector that is built from appending all n rows together. We then apply a one-dimensional DCT individually on each group of n points, which corresponds to calculating the row-wise DCT. Then we permute the result vector such that all n first vector elements of each group are grouped together, then all n second vector elements, and so on. On the resulting vector, again to each group of n points, individual n-point DCTs are applied, which corresponds to calculating the column-wise DCT. Finally, the elements of the result vector have to be permuted again in the same way as between the first row-wise DCT and the column-wise DCT. For a two-dimensional 8-point DCT this can now be written in the form of a matrix-vector product

as follows:

$$\begin{pmatrix} \tilde{F}(0,0) \\ \tilde{F}(0,1) \\ \vdots \\ \tilde{F}(0,7) \\ \tilde{F}(1,0) \\ \tilde{F}(1,1) \\ \vdots \\ \tilde{F}(1,7) \\ \vdots \\ \tilde{F}(7,0) \\ \tilde{F}(7,1) \\ \vdots \\ \tilde{F}(7,7) \end{pmatrix} = P_{64,8} \cdot \begin{pmatrix} K_8 & & & & & & \dots & 0 \\ & K_8 & & & & & \ddots & \vdots \\ & & K_8 & & & & & \\ & & & K_8 & & & & \\ & & & & K_8 & & & \\ & & & & & K_8 & & \\ \vdots & \ddots & & & & & K_8 & \\ 0 & \dots & & & & & & K_8 \end{pmatrix}$$

$$\cdot P_{64,8} \cdot \begin{pmatrix} K_8 & & & & & & \dots & 0 \\ & K_8 & & & & & \ddots & \vdots \\ & & K_8 & & & & & \\ & & & K_8 & & & & \\ & & & & K_8 & & & \\ & & & & & K_8 & & \\ \vdots & \ddots & & & & & K_8 & \\ 0 & \dots & & & & & & K_8 \end{pmatrix} \cdot \begin{pmatrix} f(0,0) \\ f(0,1) \\ \vdots \\ f(0,7) \\ f(1,0) \\ f(1,1) \\ \vdots \\ f(1,7) \\ \vdots \\ f(7,0) \\ f(7,1) \\ \vdots \\ f(7,7) \end{pmatrix} \tag{3.158}$$

$$= P_{64,8} \cdot (I_8 \otimes K_8) \cdot P_{64,8} \cdot (I_8 \otimes K_8) \cdot \begin{pmatrix} f(0,0) \\ f(0,1) \\ \vdots \\ f(0,7) \\ f(1,0) \\ f(1,1) \\ \vdots \\ f(1,7) \\ \vdots \\ f(7,0) \\ f(7,1) \\ \vdots \\ f(7,7) \end{pmatrix}$$

Now, since $(I_8 \otimes K_8) = P_{64,8} \cdot (K_8 \otimes I_8) \cdot P_{64,8}$ and $P_{64,8} \cdot P_{64,8} = I_8$ we can write:

$$P_{64,8} \cdot (I_8 \otimes K_8) \cdot P_{64,8} \cdot (I_8 \otimes K_8) = P_{64,8} \cdot P_{64,8} \cdot (K_8 \otimes I_8) \cdot P_{64,8} \cdot P_{64,8} \cdot (I_8 \otimes K_8) = (K_8 \otimes I_8) \cdot (I_8 \otimes K_8) = (K_8 \cdot I_8) \otimes (I_8 \cdot K_8) = K_8 \otimes K_8 \tag{3.159}$$

With equation (3.159) we now get from equation (3.158):

$$\begin{pmatrix} \tilde{F}(0,0) \\ \tilde{F}(0,1) \\ \vdots \\ \tilde{F}(0,7) \\ \tilde{F}(1,0) \\ \tilde{F}(1,1) \\ \vdots \\ \tilde{F}(1,7) \\ \vdots \\ \tilde{F}(7,0) \\ \tilde{F}(7,1) \\ \vdots \\ \tilde{F}(7,7) \end{pmatrix} = (K_8 \otimes K_8) \begin{pmatrix} f(0,0) \\ f(0,1) \\ \vdots \\ f(0,7) \\ f(1,0) \\ f(1,1) \\ \vdots \\ f(1,7) \\ \vdots \\ f(7,0) \\ f(7,1) \\ \vdots \\ f(7,7) \end{pmatrix} \tag{3.160}$$

This shows, that the matrix for the two-dimensional DCT can be built from applying the tensor product of the one-dimensional DCT to itself.

3.4.3 Feig's fast two-dimensional DCT

This section will deal with a true two-dimensional approach to the calculation of the 2D 8-point DCT. It was published by E. Feig in 1990 ([8]). A good introduction into this algorithm can also be found in [22] (1993) and in [9] (1992). Feig's algorithm takes the Arai-Agui-Nakajima DCT as the basis and achieves a computational complexity of 54 multiplications, 6 shift operations and 462 additions, whereas an Arai-Agui-Nakajima DCT in row-column scheme would require $16 \times 5 = 80$ multiplications and $16 \times 29 = 464$ additions.

From section 3.4.2 we know, that the two-dimensional DCT can be expressed as the tensor product of the one-dimensional DCT with itself (see equations (3.159) and (3.160)). From equation (3.145) we also know, that the Arai-Agui-Nakajima DCT can be expressed as a product of matrices with an 8-point input vector as follows:

$$\begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} = K_8 \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} = D_F \cdot P_F \cdot B_F \cdot M_F \cdot A_F \cdot \begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} \tag{3.161}$$

The matrices D_F, P_F, B_F, M_F and A_F are as follows (compare with equation(3.145)):

$$D_F = \begin{pmatrix} \frac{1}{2\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{1}{4C_1} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{4C_2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{1}{4C_3} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_5} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_6} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{1}{4C_7} \end{pmatrix} \tag{3.162}$$

$$P_F = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix} \tag{3.163}$$

$$B_F = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & -1 & 0 & 1 \end{pmatrix} \tag{3.164}$$

$$M_F = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -\cos\frac{\pi}{8} & 0 & -\cos\frac{3\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{\sqrt{2}} & 0 & 0 \\ 0 & 0 & 0 & 0 & -\cos\frac{3\pi}{8} & 0 & \cos\frac{\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.165}$$

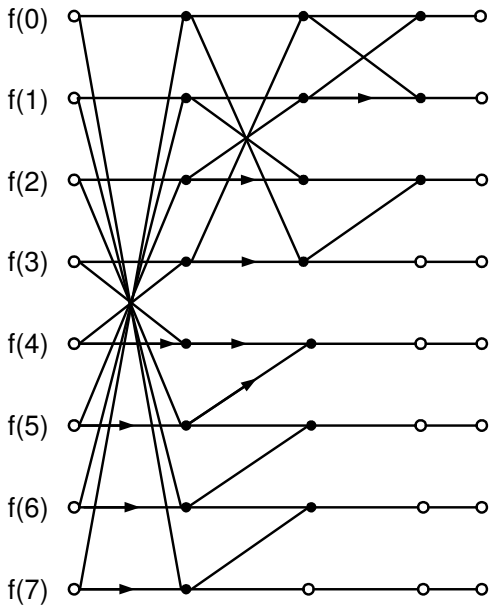

Figure 3.12: Flowgraph for matrix A_F

$$
\begin{aligned}
A_F \;=\; & \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\
& \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix}
\end{aligned} \tag{3.166}
$$

Figures 3.12, 3.14 and 3.13 show the flowgraphs of matrices A_F, B_F and M_F, respectively. For figure 3.13, the values of a_1, a_2, a_3, a_4 and a_5 are the ones from equation (3.144).

The tensor product $K_8 \otimes K_8$ can be now written as:

$$K_8 \otimes K_8 = (D_F \cdot P_F \cdot B_F \cdot M_F \cdot A_F) \otimes (D_F \cdot P_F \cdot B_F \cdot M_F \cdot A_F) \tag{3.167}$$

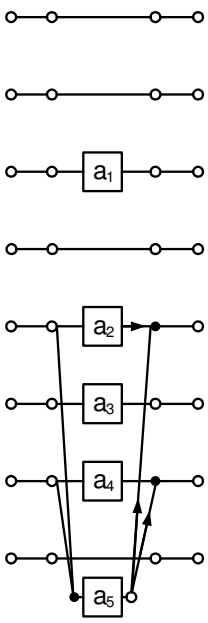

Figure 3.13: Flowgraph for matrix M_F

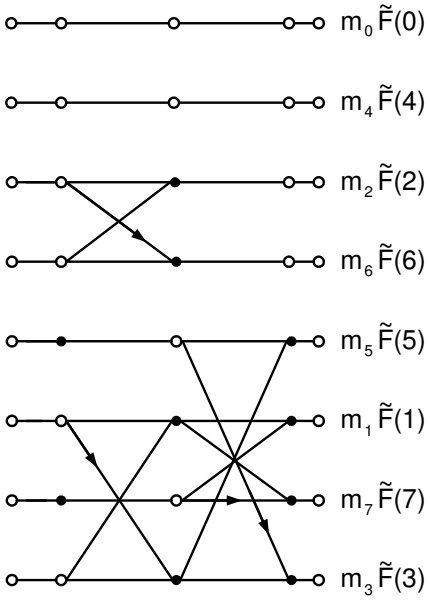

Figure 3.14: Flowgraph for matrix B_F

From the distributive property of the tensor product we get:

$$K_8 \otimes K_8 = (D_F \otimes D_F) \cdot ((P_F \cdot B_F \cdot M_F \cdot A_F) \otimes (P_F \cdot B_F \cdot M_F \cdot A_F)) \tag{3.168}$$

With the shortcut $K'_8 = P_F \cdot B_F \cdot M_F \cdot A_F$ we can now write:

$$K_8 \otimes K_8 = (D_F \otimes D_F) \cdot (K'_8 \otimes K'_8) \tag{3.169}$$

The matrix $(D_F \otimes D_F)$ is a diagonal matrix of dimension 64×64, containing the 64 scaling factors that can be absorbed into the quantization values. For practical reasons, in the following we will rather deal with $(K'_8 \otimes K'_8)$ than $(K_8 \otimes K_8)$, keeping in mind, that the result vector of $(K'_8 \otimes K'_8)$ needs to be scaled by $(D_F \otimes D_F)$.

Again using the distributive property of tensor products, we can express $(K'_8 \otimes K'_8)$ as follows:

$$(K'_8 \otimes K'_8) = (P_F \otimes P_F) \cdot (B_F \otimes B_F) \cdot (M_F \otimes M_F) \cdot (A_F \otimes A_F) \tag{3.170}$$

The matrix $(P_F \otimes P_F)$ in equation (3.170) is simply a 64×64 permutation matrix that reorders the output to natural order, but does not contribute to computational complexity. For the other factors in equation (3.170) we can now choose whether we want to apply a row-column scheme, or whether we want to calculate the tensor product to form a 64×64 matrix. Feig's approach was to use a row-column scheme for the tensor products $(B_F \otimes B_F)$ and $(A_F \otimes A_F)$, which contain merely additions, but to calculate the tensor product $(M_F \otimes M_F)$, which contains all the multiplications.

The tensor product $(A_F \otimes A_F)$ can be written as follows:

$$(A_F \otimes A_F) = (I_8 \cdot A_F) \otimes (A_F \cdot I_8) = (I_8 \otimes A_F) \cdot (A_F \otimes I_8) \tag{3.171}$$

Using equation (3.156), this can be written as follows:

$$(A_F \otimes A_F) = (I_8 \otimes A_F) \cdot P_{64,8} \cdot (I_8 \otimes A_F) \cdot P_{64,8} \tag{3.172}$$

Matrix $P_{64,8}$ simply permutes the 64 elements of the input vector, and $(I_8 \otimes A_F)$ can be expanded to the following 64×64 matrix:

$$(I_8 \otimes A_F) = \begin{pmatrix} A_F & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_F & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_F & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_F & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & A_F & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_F & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & A_F & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_F \end{pmatrix} \tag{3.173}$$

Equations (3.172) and (3.173) are illustrated in the form of a flowgraph in figure 3.15. Note that this flowgraph uses a hollow square at line ends to represent an 8-element vector input and output of an 8×8 matrix. The boxes labeled A_F and $P_{64,8}$ represent the respective matrices. Also, the initial permutation with the stride-by-8 permutation matrix is already represented by the choice of the input vectors to the left row of the matrices labeled with A_F.

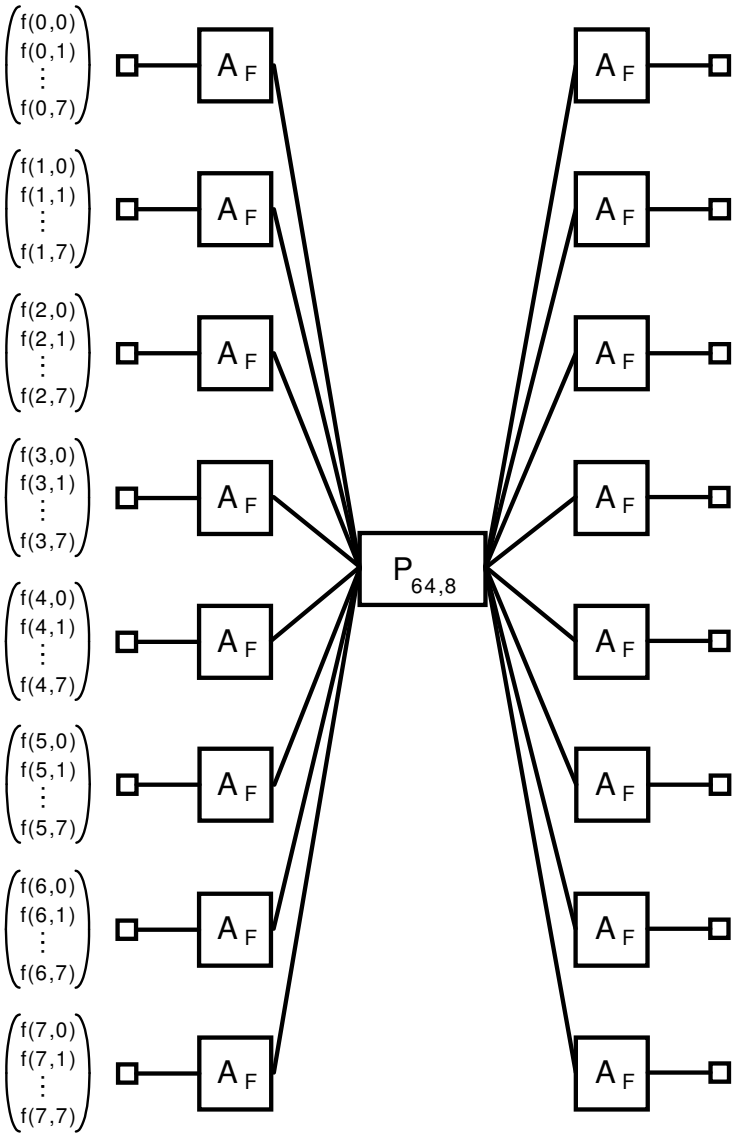

Figure 3.15: Flowgraph for matrix $A_F \otimes A_F$

Similar to $(A_F \otimes A_F)$ in equation (3.172), $(B_F \otimes B_F)$ can be expressed as follows:

$$(B_F \otimes B_F) = B_F \cdot I_8 \otimes I_8 \cdot B_F = (B_F \otimes I_8) \cdot (I_8 \otimes B_F) = P_{64,8} \cdot (I_8 \otimes B_F) \cdot P_{64,8} \cdot (I_8 \otimes B_F) \quad (3.174)$$

Matrix $(I_8 \otimes B_F)$ can be expanded to the following 64×64 matrix:

$$(I_8 \otimes B_F) = \begin{pmatrix} B_F & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & B_F & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_F & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & B_F & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & B_F & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & B_F & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & B_F & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & B_F \end{pmatrix} \quad (3.175)$$

Equation (3.175) is illustrated in the form of a flowgraph in figure 3.16. Like the initial permutation in figure 3.15, the final permutation with the stride-by-8 permutation matrix is already represented by the choice of the output vectors of the right row of the matrices labeled with B_F. The output vectors in figure 3.16 are the values of the DCT with the scaling factors from $(D_F \otimes D_F)$ omitted and without the final permutation with $(P_F \otimes P_F)$, therefore they are labeled $\tilde{F}'(u, v)$ to avoid confusion with the output $\tilde{F}(n, m)$ of the DCT. Again, this flowgraph uses a hollow square at line ends to represent an 8-element vector input and output of an 8×8 matrix and the boxes labeled B_F and $P_{64,8}$ represent the respective matrices.

The last missing tensor product in equation (3.170), $M_F \otimes M_F$, can be written in the form of a 64×64 matrix as follows:

$$(M_F \otimes M_F) = \begin{pmatrix} M_F & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & M_F & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C_4 \cdot M_F & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & M_F & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_2 \cdot M_F & 0 & -C_6 \cdot M_F & 0 \\ 0 & 0 & 0 & 0 & 0 & C_4 \cdot M_F & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_6 \cdot M_F & 0 & C_2 \cdot M_F & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & M_F \end{pmatrix} \quad (3.176)$$

Rows 0, 1, 3 and 7 represent 4 matrices $M_1 = M_F$, each operating on a sequence of 8 elements of the input vector, therefore contributing $4 \times 3 = 12$ additions and $4 \times 5 = 20$ multiplications to the overall complexity of this algorithm. See figure 3.13 for a flowgraph of $M_1 = M_F$.

Rows 2 and 5 each represent a variant of matrix M_F that is scaled by C_4. Each of these two matrices $M_2 = C_4 \cdot M_F$ is operating on a sequence of 8 input elements of the input vector. While M_2 requires 9 multiplications, 2 of them are actually multiplications with $C_4 \cdot C_4 = \frac{1}{2}$, which can be easily performed by a right shift operation. The 8×8 matrix

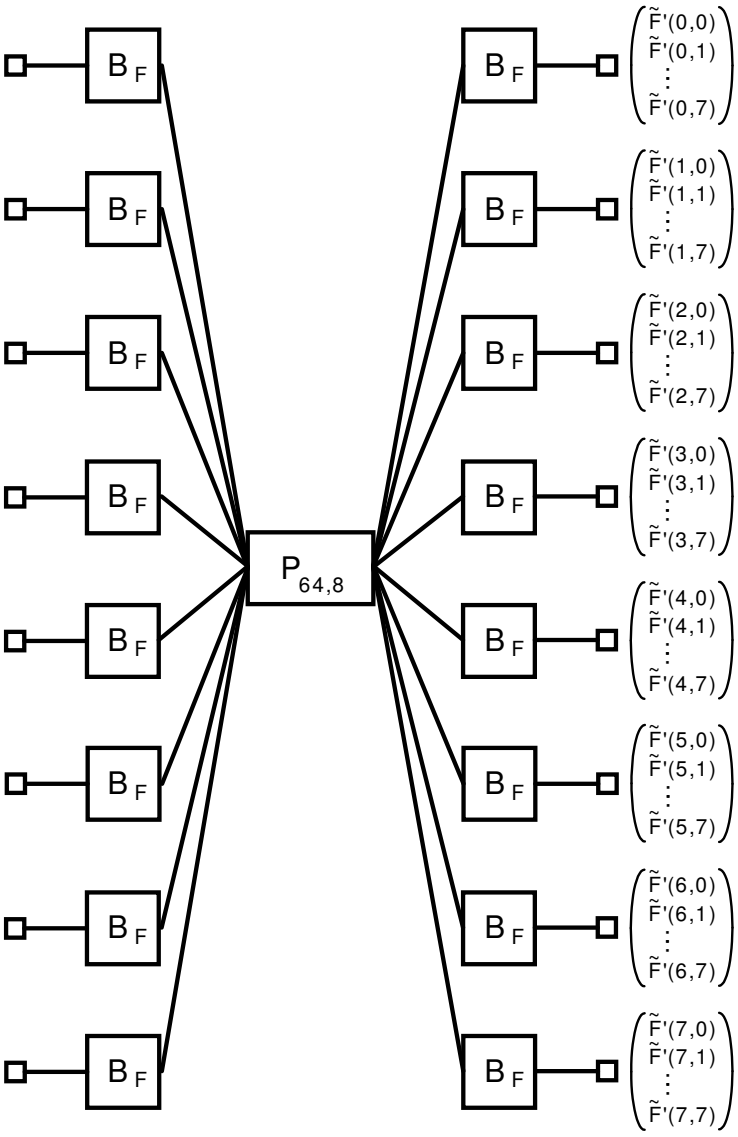

Figure 3.16: Flowgraph for matrix $B_F \otimes B_F$

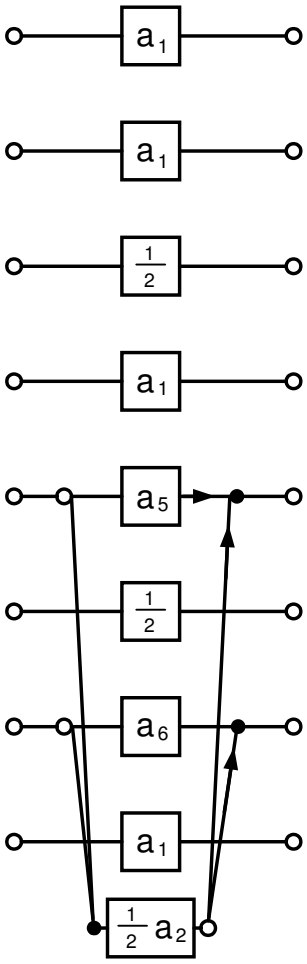

Figure 3.17: Flowgraph for matrix M_2

$M_2 = C_4 \cdot M_F$ can therefore be written as follows:

$$M_2 = C_4 \cdot M_F = \begin{pmatrix} C_4 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & C_4 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & C_4 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_2 \cdot C_4 & 0 & -C_6 \cdot C_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{1}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_6 \cdot C_4 & 0 & C_2 \cdot C_4 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & C_4 \end{pmatrix} \tag{3.177}$$

Figure 3.17 shows the flowgraph of M_2 from which we can see, that the two rows in equation (3.176) with $M_2 = C_4 \cdot M_F$ contribute $2 \times 7 = 14$ multiplications, $2 \times 3 = 6$ additions and $2 \times 2 = 4$ shift operations to the overall complexity of the Feig 2D-DCT. The constants used in figure 3.17 are the ones from equations (3.144) and (3.148).

The remaining rows and columns in equation (3.176) can be represented by the tensor

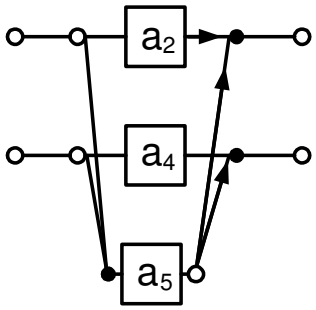

Figure 3.18: Flowgraph for matrix N_1

product $M_3 = \begin{pmatrix} -C_2 & -C_6 \\ -C_6 & C_2 \end{pmatrix} \otimes M_F = \tilde{N} \otimes M_F$ with

$$\tilde{N} = \begin{pmatrix} -C_2 & -C_6 \\ -C_6 & C_2 \end{pmatrix} = \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -\cos\frac{3\pi}{8} & 0 & 0 \\ 0 & (\cos\frac{3\pi}{8} - \cos\frac{\pi}{8}) & 0 \\ 0 & 0 & \left(\cos\frac{\pi}{8} + \cos\frac{3\pi}{8}\right) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \tag{3.178}$$

With equation (3.156) the tensor product $M_3 = \tilde{N} \otimes M_F$ can be written as:

$$M_3 = \tilde{N} \otimes M_F = P_{16,2} \cdot (M_F \otimes \tilde{N}) \cdot P_{16,8} \tag{3.179}$$

The 16×16 matrix $M_F \otimes \tilde{N}$ is defined as:

$$(M_F \otimes \tilde{N}) = \begin{pmatrix} \tilde{N} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \tilde{N} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & C_4 \cdot \tilde{N} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \tilde{N} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_2 \cdot \tilde{N} & 0 & -C_6 \cdot \tilde{N} & 0 \\ 0 & 0 & 0 & 0 & 0 & C_4 \cdot \tilde{N} & 0 & 0 \\ 0 & 0 & 0 & 0 & -C_6 \cdot \tilde{N} & 0 & C_2 \cdot \tilde{N} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \tilde{N} \end{pmatrix} \tag{3.180}$$

This means, that the first, second, fourth and eighth pair of input vector elements of $M_F \otimes \tilde{N}$ are matrix-multiplied by $N_1 = \tilde{N}$ and that the third and fifth pair of input vector elements of $M_F \otimes \tilde{N}$ are matrix-multiplied by a scaled variant of $\tilde{N}$, $N_2 = C_4 \cdot \tilde{N}$. These 6 matrix multiplications require each 3 multiplications and 3 additions. Figures 3.18 and 3.19 show the flowgraphs of N_1 and N_2, respectively. The constants used in figure 3.19 are the ones from equations (3.144) and (3.148).

The remaining rows in equation (3.180) can be expressed as the tensor product

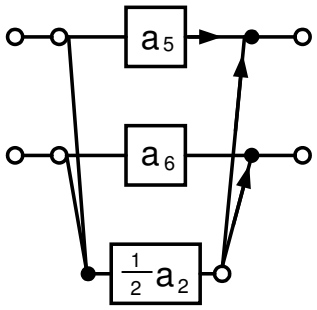

Figure 3.19: Flowgraph for matrix N_2

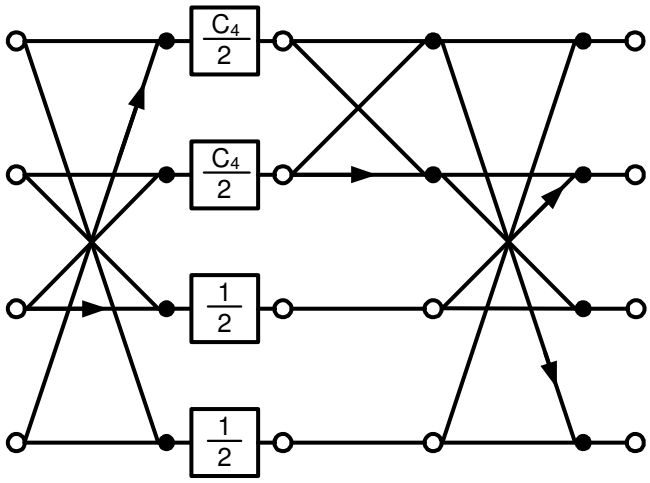

Figure 3.20: Flowgraph for matrix $N_3 = \tilde{N} \otimes \tilde{N}$

$N_3 = \tilde{N} \otimes \tilde{N}$ and can be written in matrix form and factorized as follows:

$$
\begin{aligned}
N_3 &= (\tilde{N} \otimes \tilde{N}) = \frac{1}{2} \cdot \begin{pmatrix} (1+C_4) & C_4 & C_4 & (1-C_4) \\ C_4 & -(1+C_4) & (1-C_4) & -C_4 \\ C_4 & (1-C_4) & -(1+C_4) & -C_4 \\ (1-C_4) & -C_4 & -C_4 & (1+C_4) \end{pmatrix} \\
&= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} \frac{C_4}{2} & 0 & 0 & 0 \\ 0 & \frac{C_4}{2} & 0 & 0 \\ 0 & 0 & \frac{1}{2} & 0 \\ 0 & 0 & 0 & \frac{1}{2} \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix}
\end{aligned}
\tag{3.181}
$$

Thus, $N_3 = \tilde{N} \otimes \tilde{N}$ can be calculated with 10 additions, 2 multiplications and 2 shift operations. Figure 3.20 shows the flowgraph of $N_3 = \tilde{N} \otimes \tilde{N}$ with C_4 defined according to equation (3.38). If we combine the flowgraphs in figures 3.18, 3.19, and 3.20 for the matrices N1, N2 and N3, we get the flowgraph for matrix $M_3 = \tilde{N} \otimes M_F$ in figure 3.21. The constants used in figure 3.21 are the ones from equations (3.38), (3.144) and (3.148).

Now that we have all individual flowgraphs for $A_F \otimes A_F$, $B_F \otimes B_F$, M_1, M_2 and M_3, we can finally derive a flowgraph for the calculation of $K'_8 \otimes K'_8$ without the final permutation

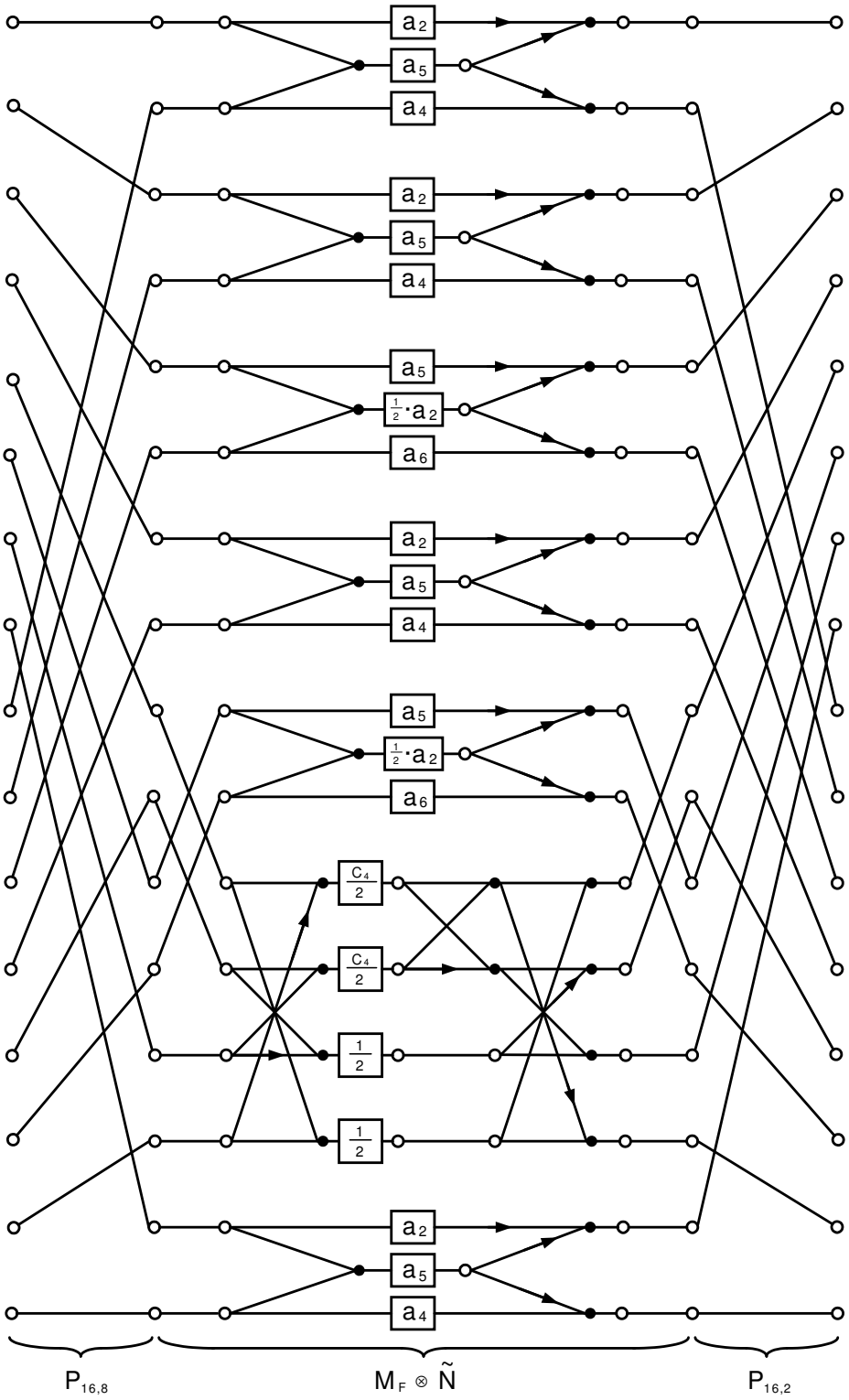

Figure 3.21: Flowgraph for matrix $M_3 = \tilde{N} \otimes M_F$

Stage	**Multiplications**	**Additions**	**Shifts**
$B_F \otimes B_F$	0	16×18	0
$A_F \otimes A_F$	0	16×8	0
$M_F \otimes M_F$:			
Rows 1, 2, 4, 8 (M_1)	4×5	4×3	0
Rows 3, 6 (M_2)	2×7	2×3	2×2
Rows 5, 7 ($M_F \otimes \tilde{N}$):			
Rows 1, 2, 4, 8 (N_1)	4×3	4×3	0
Rows 3, 6 (N_2)	2×3	2×3	0
Rows 5, 7 (N_3)	2	10	2
Totals:	**54**	**462**	**6**

Table 3.1: Computational complexity of Feig's 2D DCT

by $P_F \otimes P_F$ in figure 3.22 and sum up the operations of the different stages in Feig's scaled 2D DCT in table 3.1.

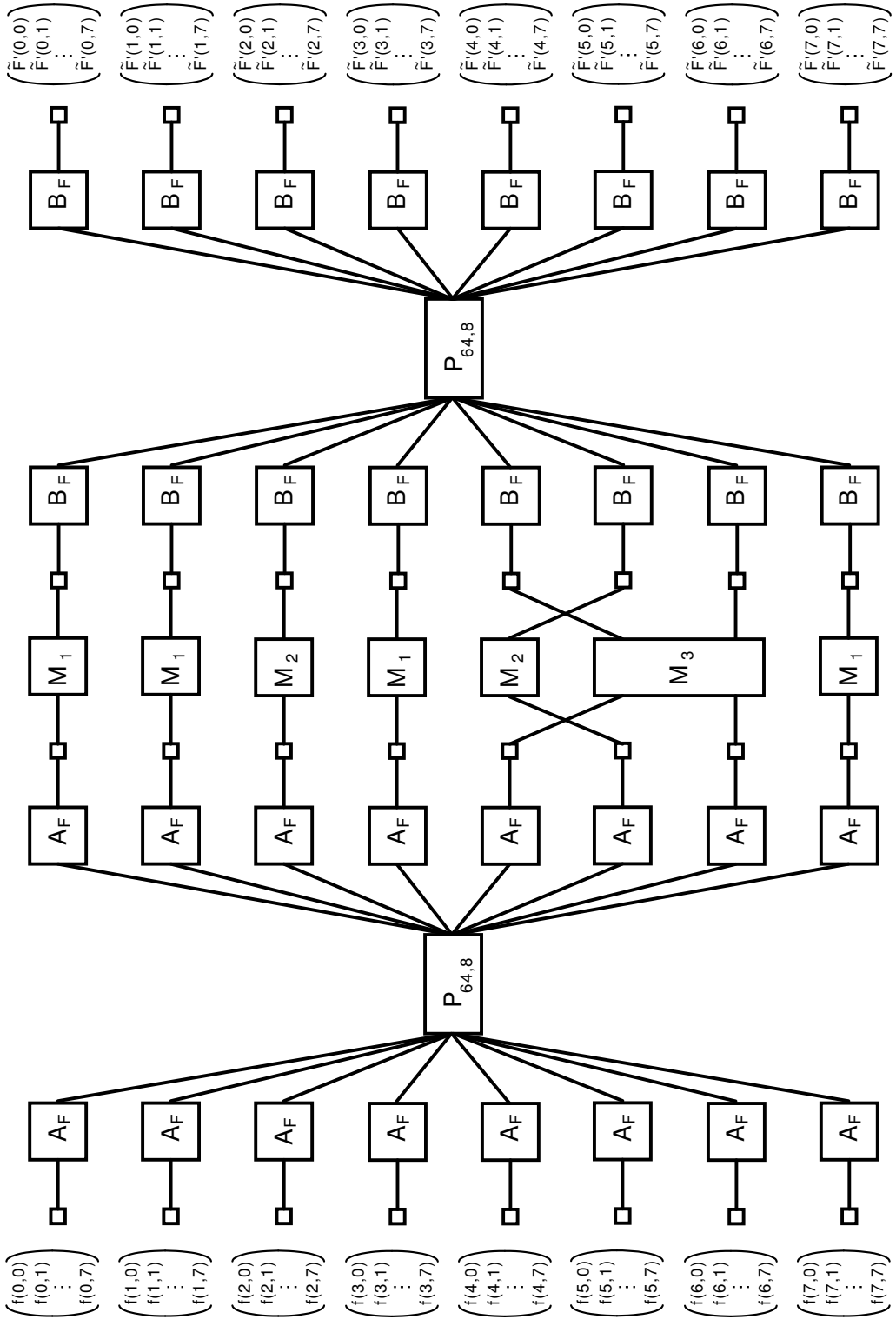

Figure 3.22: Flowgraph of $K'_8 \otimes K'_8$ without the final permutation by $(P \otimes P)$

3.4.4 Feig's fast two-dimensional inverse DCT

This section will show in a similar manner to section 3.4.3 how the inverse Arai-Agui-Nakajima DCT can be extended to a true two-dimensional DCT.

From equation (3.146) we can derive the following for the inverse Arai-Agui-Nakajima DCT:

$$\begin{pmatrix} f(0) \\ f(1) \\ f(2) \\ f(3) \\ f(4) \\ f(5) \\ f(6) \\ f(7) \end{pmatrix} = {K_8}^{-1} \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} = A_I \cdot M_I \cdot B_I \cdot P_I \cdot D_I \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(4) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix} \tag{3.182}$$

The matrices A_I, M_I, B_I, P_I and D_I are as follows (compare with equation(3.146)):

$$\begin{aligned} A_I \quad = \quad & \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & -1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & -1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & -1 & 0 \\ 1 & 0 & 0 & 0 & 0 & 0 & 0 & -1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & -1 & 0 & 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & -1 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\ \cdot \quad & \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{3.183}$$

$$M_I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\cos\frac{\pi}{8} & 0 & -2\cos\frac{3\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\cos\frac{3\pi}{8} & 0 & 2\cos\frac{\pi}{8} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \tag{3.184}$$

$$B_I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 \end{pmatrix} \tag{3.185}$$

$$P_I = \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 \end{pmatrix} \tag{3.186}$$

$$D_I = \begin{pmatrix} \frac{1}{2\sqrt{2}} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \frac{C_1}{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \frac{C_2}{2} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \frac{C_3}{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & \frac{1}{2\sqrt{2}} & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & \frac{C_5}{2} & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & \frac{C_6}{2} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \frac{C_7}{2} \end{pmatrix} \tag{3.187}$$

Figures 3.23, 3.24 and 3.25 show the flowgraphs of matrices A_I, B_I and M_I, respectively. For figure 3.25, the values of a_1, a_2, a_3, a_4 and a_5 are the ones from equation (3.144). With the help of the inversion property (see equation (3.154)) of the tensor product, $(K_8 \otimes K_8)^{-1}$ can now be written as:

$$(K_8 \otimes K_8)^{-1} = K_8^{-1} \otimes K_8^{-1} = (A_I \cdot M_I \cdot B_I \cdot P_I \cdot D_I) \otimes (A_I \cdot M_I \cdot B_I \cdot P_I \cdot D_I) \tag{3.188}$$

From the distributive property of the tensor product we get:

$$(K_8 \otimes K_8)^{-1} = ((A_I \cdot M_I \cdot B_I \cdot P_I) \otimes (A_I \cdot M_I \cdot B_I \cdot P_I)) \cdot (D_I \otimes D_I) \tag{3.189}$$

With the shortcut ${K'_8}^{-1} = A_I \cdot M_I \cdot B_I \cdot P_I$ we can now write:

$$(K_8 \otimes K_8)^{-1} = ({K'_8}^{-1} \otimes {K'_8}^{-1}) \cdot (D_I \otimes D_I) = (K'_8 \otimes K'_8)^{-1} \cdot (D_I \otimes D_I) \tag{3.190}$$

The matrix $(D_I \otimes D_I)$ is a diagonal matrix of dimension 64×64, containing the 64 scaling factors that can be absorbed into the quantization values. Again, for practical reasons, in the following we will rather deal with $(K'_8 \otimes K'_8)^{-1}$ than $(K_8 \otimes K_8)^{-1}$, keeping in mind, that the input vector of $(K'_8 \otimes K'_8)^{-1}$ needs to be scaled by $(D_I \otimes D_I)$.

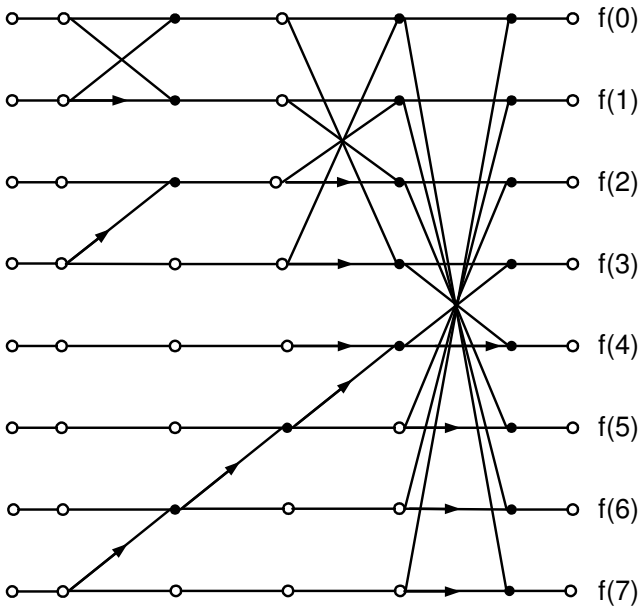

Figure 3.23: Flowgraph for matrix A_I

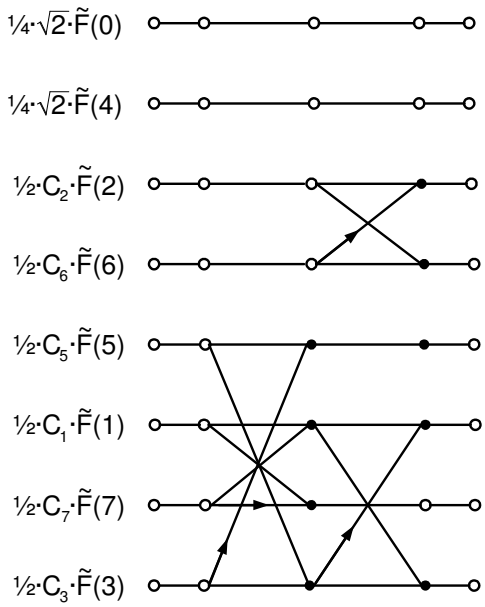

Figure 3.24: Flowgraph for matrix B_I

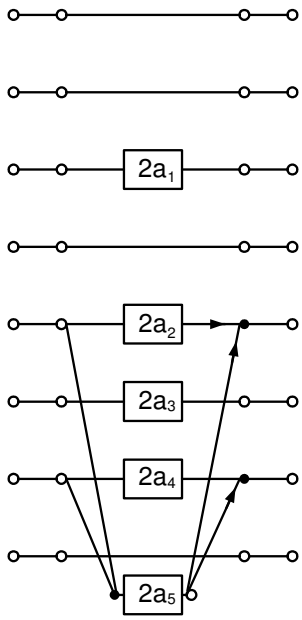

Figure 3.25: Flowgraph for matrix M_I

Again using the distributive property of tensor products, we can express $(K'_8 \otimes K'_8)$ as follows:

$$(K'_8 \otimes K'_8)^{-1} = (A_I \otimes A_I) \cdot (M_I \otimes M_I) \cdot (B_I \otimes B_I) \cdot (P_I \otimes P_I) \tag{3.191}$$

The matrix $(P_I \otimes P_I)$ in equation (3.191) is simply a 64×64 permutation matrix that reorders the input from natural order to an input order suitable for $B_I \otimes B_I$, but does not contribute to computational complexity. For the other factors in equation (3.191) we can again choose whether we want to apply a row-column scheme, or whether we want to calculate the tensor product to form a 64×64 matrix. Like in section 3.4.3 we will follow Feig's approach and use a row-column scheme for the tensor products $(A_I \otimes A_I)$ and $(B_I \otimes B_I)$, which contain merely additions, but to calculate the tensor product $(M_I \otimes M_I)$, which contains all the multiplications.

The tensor product $(B_I \otimes B_I)$ can be written as follows:

$$(B_I \otimes B_I) = (I_8 \cdot B_I) \otimes (B_I \cdot I_8) = (I_8 \otimes B_I) \cdot (B_I \otimes I_8) \tag{3.192}$$

Using equation (3.156), this can be written as follows:

$$(B_I \otimes B_I) = (I_8 \otimes B_I) \cdot P_{64,8} \cdot (I_8 \otimes B_I) \cdot P_{64,8} \tag{3.193}$$

Matrix $P_{64,8}$ simply permutes the 64 elements of the input vector, and $(I_8 \otimes B_I)$ can be

expanded to the following 64×64 matrix:

$$(I_8 \otimes B_I) = \begin{pmatrix} B_I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & B_I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & B_I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & B_I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & B_I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & B_I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & B_I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & B_I \end{pmatrix} \tag{3.194}$$

Equations (3.193) and (3.194) are illustrated in the form of a flowgraph in figure 3.26. Again, this flowgraph uses a hollow square at line ends to represent an 8-element vector input and output of an 8×8 matrix. The boxes labeled B_I and $P_{64,8}$ represent the respective matrices. Also, the initial permutation with the stride-by-8 permutation matrix is already represented by the choice of the input vectors to the left row of the matrices labeled with B_I. The input vectors in figure 3.26 are the values of the DCT coefficients but with the scaling factors from $(D_I \otimes D_I)$ and the permutation from $(P_I \otimes P_I)$ omitted, therefore they are labeled $\tilde{F}'(u, v)$ to avoid confusion with the coefficients $\tilde{F}(u, v)$ of the DCT.

Similar to $(B_I \otimes B_I)$ in equation (3.193), $(A_I \otimes A_I)$ can be expressed as follows:

$$(A_I \otimes A_I) = A_I \cdot I_8 \otimes I_8 \cdot A_I = (A_I \otimes I_8) \cdot (I_8 \otimes A_I) = P_{64,8} \cdot (I_8 \otimes A_I) \cdot P_{64,8} \cdot (I_8 \otimes A_I) \tag{3.195}$$

Matrix $(I_8 \otimes A_I)$ can be expanded to the following 64×64 matrix:

$$(I_8 \otimes A_I) = \begin{pmatrix} A_I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & A_I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & A_I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & A_I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & A_I & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & A_I & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & A_I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & A_I \end{pmatrix} \tag{3.196}$$

Equation (3.196) is illustrated in the form of a flowgraph in figure 3.27. Like the initial permutation in figure 3.26, the final permutation with the stride-by-8 permutation matrix is already represented by the choice of the output vectors of the right row of the matrices labeled with A_I. Again, this flowgraph uses a hollow square at line ends to represent an 8-element vector input and output of an 8×8 matrix and the boxes labeled A_I and $P_{64,8}$ represent the respective matrices.

The last missing tensor product in equation (3.191), $M_I \otimes M_I$, can be written in the form of a 64×64 matrix as follows:

$$(M_I \otimes M_I) = \begin{pmatrix} M_I & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & M_I & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2} \cdot M_I & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & M_I & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2C_2 \cdot M_I & 0 & -2C_6 \cdot M_I & 0 \\ 0 & 0 & 0 & 0 & 0 & \sqrt{2} \cdot M_I & 0 & 0 \\ 0 & 0 & 0 & 0 & -2C_6 \cdot M_I & 0 & 2C_2 \cdot M_I & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & M_I \end{pmatrix} \tag{3.197}$$

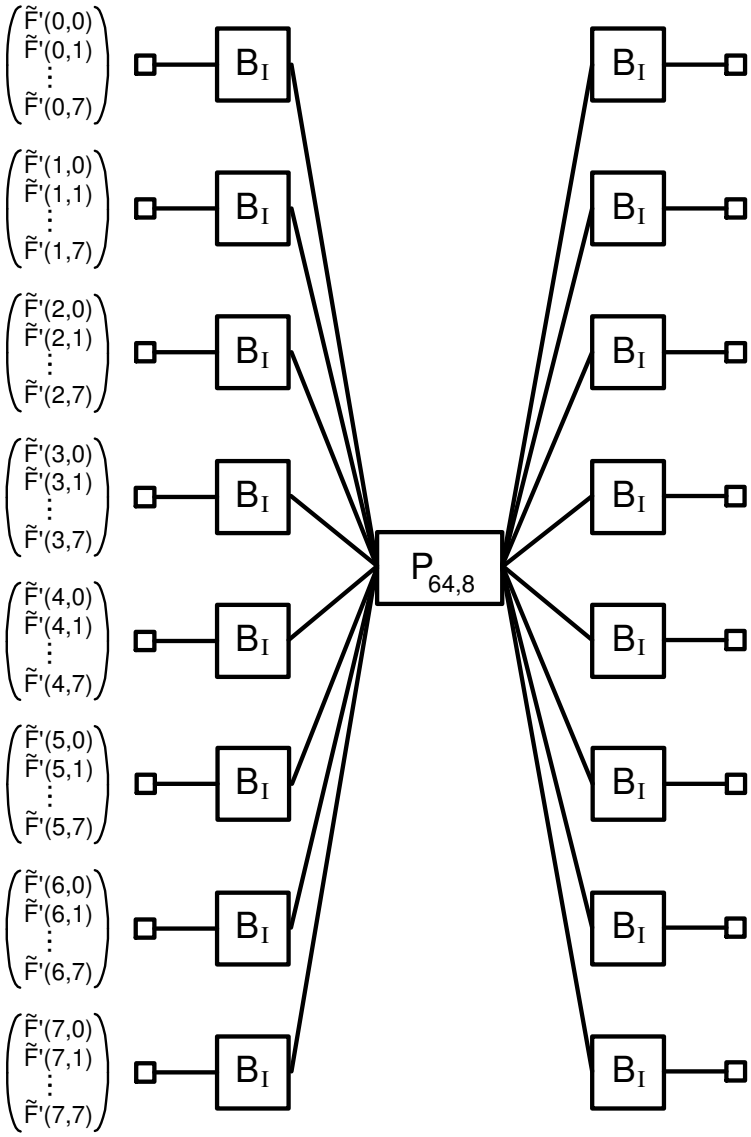

Figure 3.26: Flowgraph for matrix $B_I \otimes B_I$

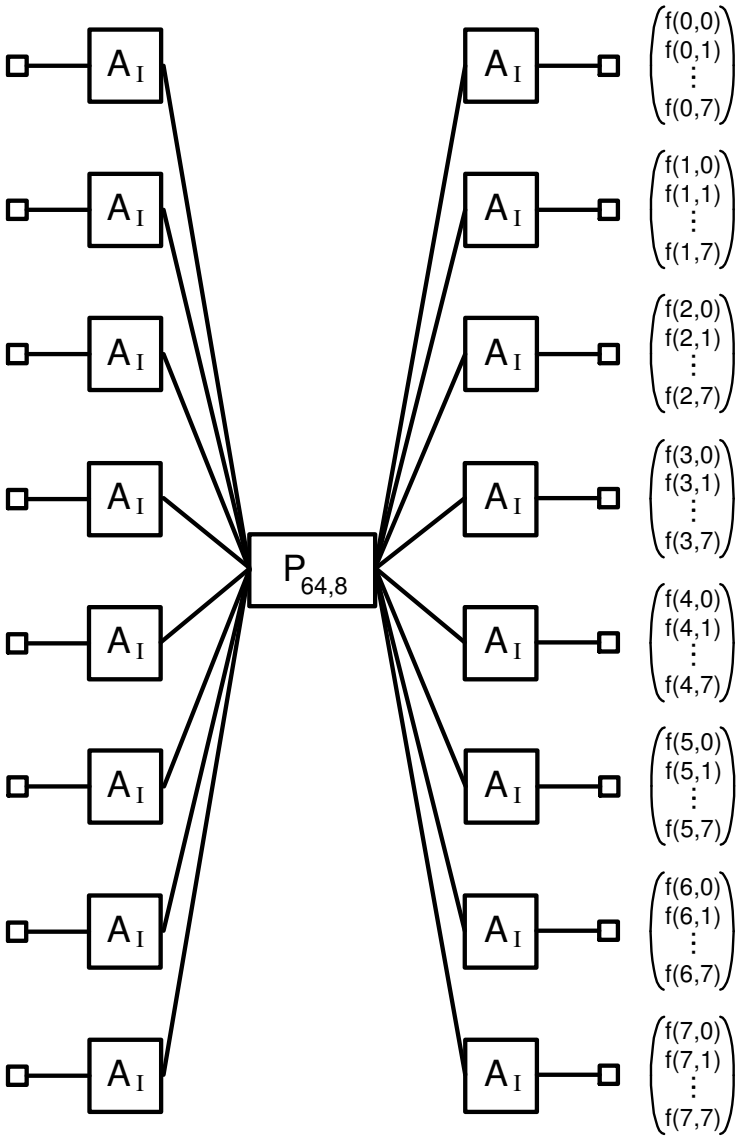

Figure 3.27: Flowgraph for matrix $A_I \otimes A_I$

Rows 0, 1, 3 and 7 represent 4 matrices $\tilde{M}_1 = M_I$, each operating on a sequence of 8 elements of the input vector, therefore contributing $4 \times 3 = 12$ additions and $4 \times 5 = 20$ multiplications to the overall complexity of this algorithm. See figure 3.25 for a flowgraph of $\tilde{M}_1 = M_I$.

Rows 2 and 5 each represent a variant of matrix M_I that is scaled by $\sqrt{2}$. Each of these two matrices $\tilde{M}_2 = \sqrt{2} \cdot M_I$ is operating on a sequence of 8 input elements of the input vector. While $\tilde{M}_2$ requires 9 multiplications, 2 of them are actually multiplications with $\sqrt{2} \cdot \sqrt{2} = 2$, which can be easily performed by a left shift operation. The 8×8 matrix $\tilde{M}_2 = \sqrt{2} \cdot M_I$ can be written as follows:

$$\tilde{M}_2 = \sqrt{2} \cdot M_I = \begin{pmatrix} \sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & \sqrt{2} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\sqrt{2} \cdot C_2 & 0 & -2\sqrt{2} \cdot C_6 & 0 \\ 0 & 0 & 0 & 0 & 0 & 2 & 0 & 0 \\ 0 & 0 & 0 & 0 & -2\sqrt{2} \cdot C_6 & 0 & 2\sqrt{2} \cdot C_2 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \sqrt{2} \end{pmatrix} \tag{3.198}$$

Figure 3.28 shows the flowgraph of $\tilde{M}_2$ from which we can see, that the two rows in equation (3.197) with $\tilde{M}_2 = \sqrt{2} \cdot M_I$ contribute $2 \times 7 = 14$ multiplications, $2 \times 3 = 6$ additions and $2 \times 2 = 4$ shift operations to the overall complexity of the inverse Feig 2D-DCT. The constants used in figure 3.28 are the ones from equations (3.144) and (3.148).

The remaining rows and columns in equation (3.197) can be represented by the tensor product $\tilde{M}_3 = 2 \cdot \left(\begin{smallmatrix} -C_2 & -C_6 \\ -C_6 & C_2 \end{smallmatrix} \right) \otimes M_I = 2 \cdot \tilde{N} \otimes M_I$ with $\tilde{N}$ as defined in equation (3.178). The factor 2 can be absorbed into the diagonal matrix in the factorization of $\tilde{N}$, so $2\tilde{N}$ can be written as:

$$\begin{aligned} 2\tilde{N} &= 2 \cdot \begin{pmatrix} -C_2 & -C_6 \\ -C_6 & C_2 \end{pmatrix} \qquad (3.199) \\ &= \begin{pmatrix} 1 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} -2 \cdot \cos\frac{3\pi}{8} & 0 & 0 \\ 0 & 2 \cdot (\cos\frac{3\pi}{8} - \cos\frac{\pi}{8}) & 0 \\ 0 & 0 & 2 \cdot \left(\cos\frac{\pi}{8} + \cos\frac{3\pi}{8}\right) \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 \\ 1 & 0 \\ 0 & 1 \end{pmatrix} \end{aligned}$$

With equation (3.156) the tensor product $\tilde{M}_3 = 2\tilde{N} \otimes M_I$ can be written as:

$$\tilde{M}_3 = 2\tilde{N} \otimes M_I = P_{16,2} \cdot (M_I \otimes 2\tilde{N}) \cdot P_{16,8} \tag{3.200}$$

The 16×16 matrix $M_I \otimes 2\tilde{N}$ is defined as:

$$(M_I \otimes 2\tilde{N}) = \begin{pmatrix} 2\tilde{N} & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 2\tilde{N} & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 2\sqrt{2} \cdot \tilde{N} & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 2\tilde{N} & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -4C_2 \cdot \tilde{N} & 0 & -4C_6 \cdot \tilde{N} & 0 \\ 0 & 0 & 0 & 0 & 0 & 2\sqrt{2} \cdot \tilde{N} & 0 & 0 \\ 0 & 0 & 0 & 0 & -4C_6 \cdot \tilde{N} & 0 & 4C_2 \cdot \tilde{N} & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & 2\tilde{N} \end{pmatrix} \tag{3.201}$$

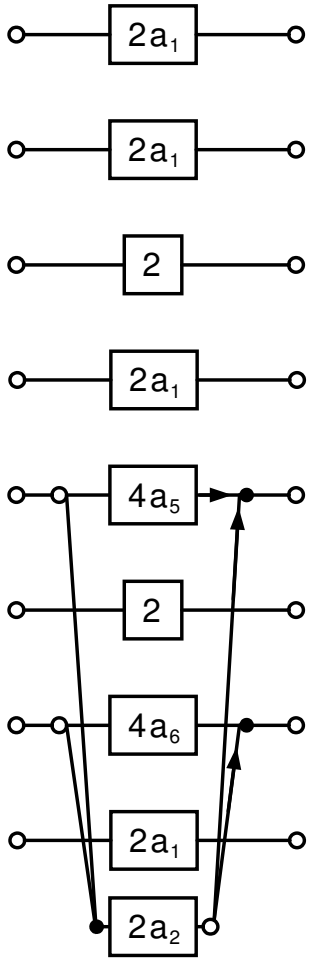

Figure 3.28: Flowgraph for matrix $\tilde{M}_2$

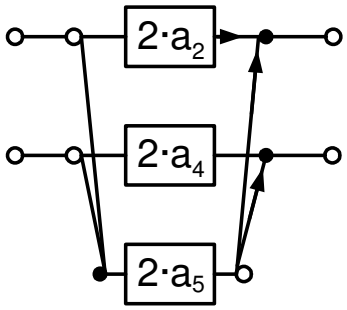

Figure 3.29: Flowgraph for matrix $\tilde{N}_1$

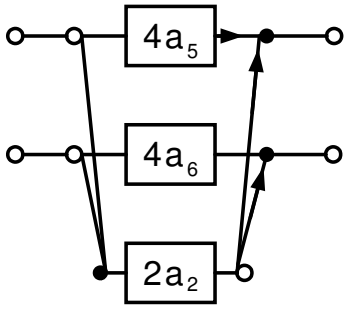

Figure 3.30: Flowgraph for matrix $\tilde{N}_2$

This means, that the first, second, fourth and eighth pair of input vector elements of $M_I \otimes 2\tilde{N}$ are matrix-multiplied by $\tilde{N}_1 = 2\tilde{N}$ and that the third and fifth pair of input vector elements of $\tilde{M} \otimes 2\tilde{N}$ are matrix-multiplied by a scaled variant of $\tilde{N}$, $\tilde{N}_2 = 2\sqrt{2} \cdot \tilde{N}$. These 6 matrix multiplications require each 3 multiplications and 3 additions. Figures 3.29 and 3.30 show the flowgraphs of $\tilde{N}_1$ and $\tilde{N}_2$, respectively. The constants used in figures 3.29 and 3.30 are the ones from equations (3.144) and (3.148).

The remaining rows in equation (3.201) can be expressed as the tensor product $\tilde{N}_3 = 4\tilde{N} \otimes \tilde{N}$ and can be written in matrix form and factorized as follows:

$$\begin{aligned} \tilde{N}_3 &= (4\tilde{N} \otimes \tilde{N}) = 2 \cdot \begin{pmatrix} (1+C_4) & C_4 & C_4 & (1-C_4) \\ C_4 & -(1+C_4) & (1-C_4) & -C_4 \\ C_4 & (1-C_4) & -(1+C_4) & -C_4 \\ (1-C_4) & -C_4 & -C_4 & (1+C_4) \end{pmatrix} \\ &= \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & -1 & 0 \\ 0 & 1 & 1 & 0 \\ -1 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 1 & 1 & 0 & 0 \\ 1 & -1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix} \cdot \begin{pmatrix} 2 \cdot C_4 & 0 & 0 & 0 \\ 0 & 2 \cdot C_4 & 0 & 0 \\ 0 & 0 & 2 & 0 \\ 0 & 0 & 0 & 2 \end{pmatrix} \cdot \begin{pmatrix} 1 & 0 & 0 & -1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & 1 \end{pmatrix} \end{aligned} \tag{3.202}$$

Thus, $\tilde{N}_3 = 4 \cdot \tilde{N} \otimes \tilde{N}$ can be calculated with 10 additions, 2 multiplications and 2 shift operations. Figure 3.31 shows the flowgraph of $\tilde{N}_3 = 4 \cdot \tilde{N} \otimes \tilde{N}$ with C_4 defined according to equation (3.38). If we combine the flowgraphs in figures 3.29, 3.30, and 3.31 for the matrices $\tilde{N}_1$, $\tilde{N}_2$ and $\tilde{N}_3$, we get the flowgraph for matrix $\tilde{M}_3 = 2\tilde{N} \otimes M_I$ in figure 3.32. The constants used in figure 3.32 are the ones from equations (3.38), (3.144) and (3.148).

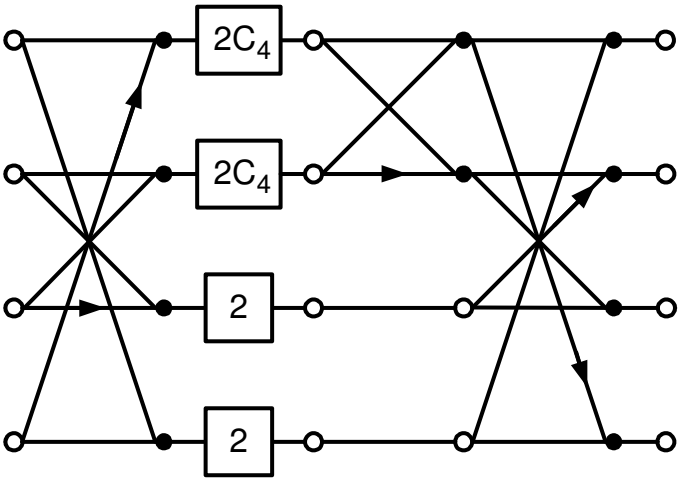

Figure 3.31: Flowgraph for matrix $\tilde{N}_3 = 4 \cdot \tilde{N} \otimes \tilde{N}$

Now that we have all individual flowgraphs for $A_I \otimes A_I$, $B_I \otimes B_I$, $\tilde{M}_1$, $\tilde{M}_2$ and $\tilde{M}_3$, we can finally derive a flowgraph for the calculation of $(K'_8 \otimes K'_8)^{-1}$ without the initial permutation by $P_I \otimes P_I$ in figure 3.33. If we now count the operations required to calculate Feig's 2D IDCT, it will be as in the case of the forward DCT, 54 multiplications, 462 additions and 6 shifts.

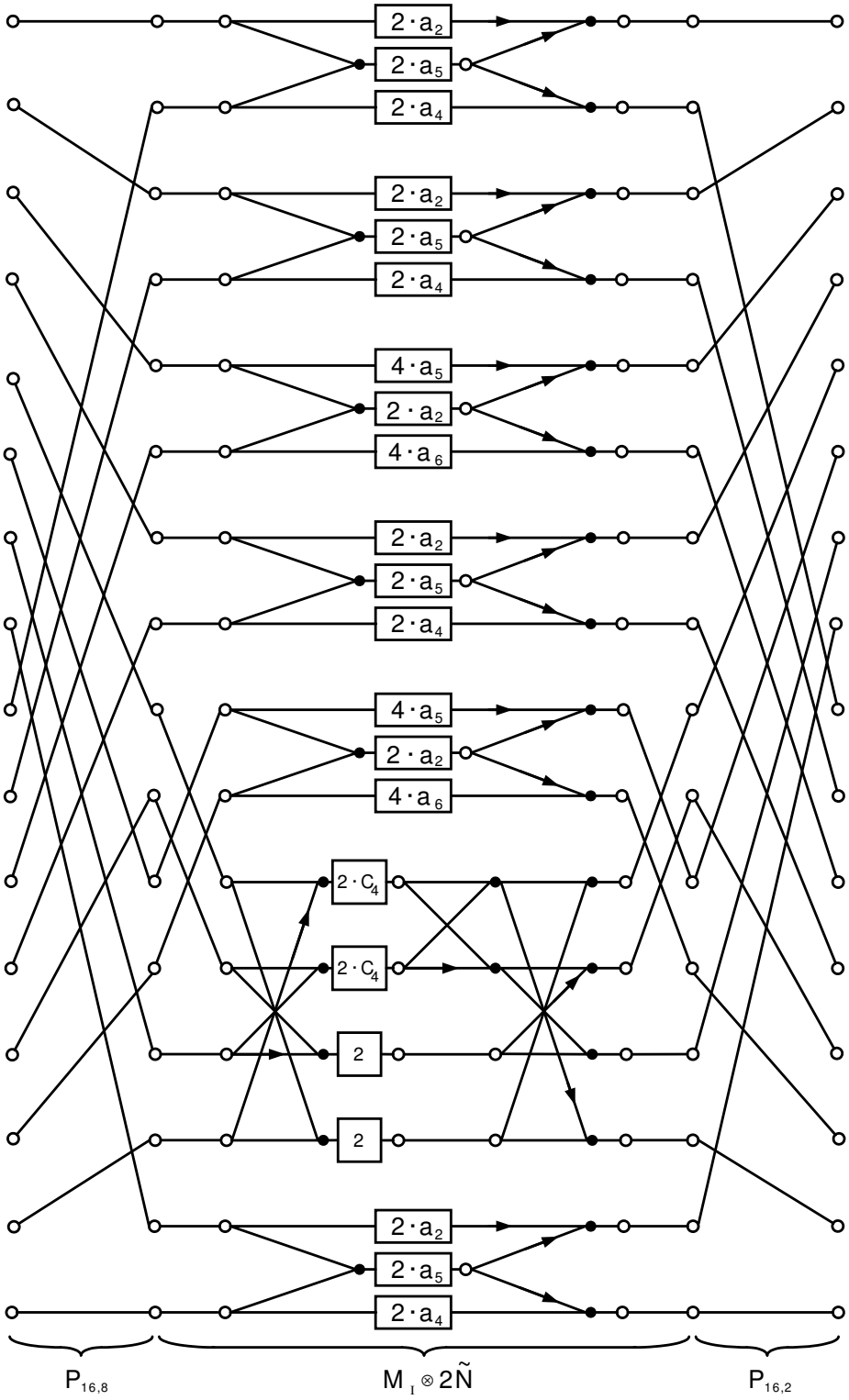

Figure 3.32: Flowgraph for matrix $\tilde{M}_3 = 2\tilde{N} \otimes M_I$

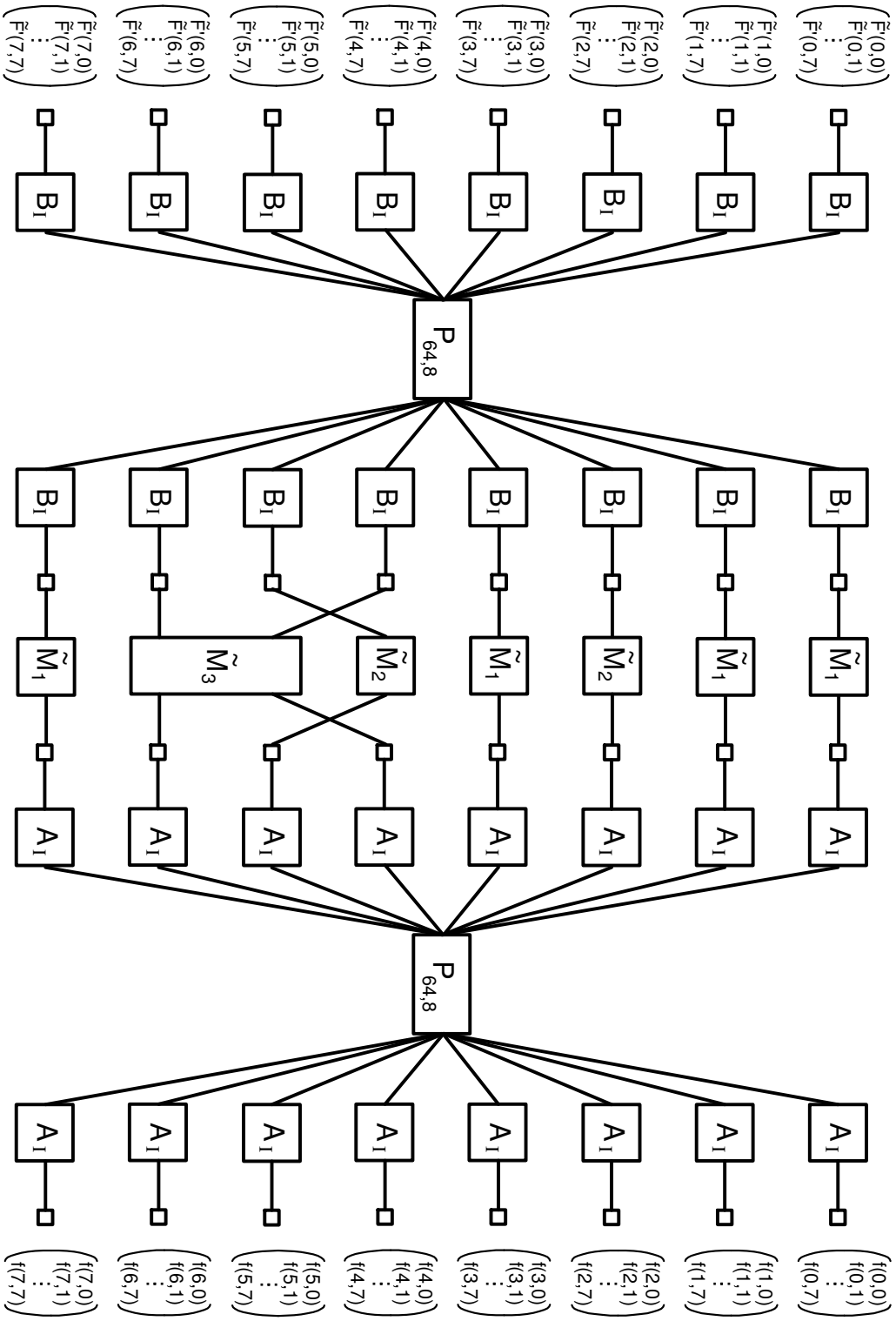

Figure 3.33: Flowgraph of $(K'_8 \otimes K'_8)^{-1}$ without the initial permutation by $(P_I \otimes P_I)$

Chapter 4

Fast Image Scaling in the Context of JPEG Decoding

Everything should be as simple as it is, but not simpler.

– Albert Einstein

WHEN a continuous tone still image has to be rendered on an output device such as a computer monitor, it may become necessary to stretch or compact it to fit the dimensions of the output device. It is very obvious that compacting can be done in the spatial domain by taking some weighted sum of the values of adjacent pixels and that this is especially easy if the old size of the image is some power of 2 times the new size. But even in this case the whole picture has to be decoded first and then resized. Performing image rescaling in the spatial domain will therefore always lead to an additional computational complexity and higher memory requirements than if we already try to incorporate the downscaling process into the IDCT decoding process[1]. Therefore we are investigating possibilities to rescale the image during the 8-point IDCT process, thereby simplifying the decoding process itself. In the following a very fast and computationally inexpensive possibility to do image scaling in the spatial domain will be presented first, along with recommendations of how more sophisticated and accurate solutions could look like.

4.1 Image Scaling in the Spatial Domain

Image scaling in the spatial domain means that first the complete image is decoded and afterwards this "source image" is scaled to a "destination image" with a new resolution that could be finer (upsampling) or coarser (downsampling). There are quite a few techniques for this that differ mostly in computational complexity and the achievable accuracy. Common to all techniques in the spatial domain is an initial loop over all destination pixels in one dimension, resulting in an image that is scaled in this dimension, followed by a loop over the

[1] Again, we will strictly focus on IDCTs with 8-point input vectors, but the results and techniques can probably also be applied to IDCTs of different input vector sizes.

destination pixels in the other dimension, which finally yields the complete scaled image. The difference between the various techniques is the method of obtaining destination pixel values from source pixel values. The most simple technique consists of mapping destination pixels to source pixels and is described in [25]. This method simply determines at each destination pixel location the single nearest source pixel location that lends this destination pixel its value. This way only pixel values are copied from one source pixel location to one or more destination pixel locations. It could also be possible in the case of downsampling that some source pixel values are completely disregarded[2]. As an optimization to the pseudo-code in [25], prior to the actual mapping process, a mapping buffer g_nMapping was used for the project used in this thesis. This buffer contains at the indices corresponding to the destination pixels the source pixel index. The source pixels are determined only once for each dimension so a lot of computational effort is saved in comparison to the pseudo-code in [25]. In order for this code to support both upsampling and downsampling, a line buffer (g_nLineBuf) is used for intermediate storage of the row or the column of the source pixels. The actual source code for the function that scales a bitmap (ScaleBitmap) is the following[3], one pixel is assumed to be coded in two bytes (data type short):

```
#define HORZPIXELS 800

unsigned short g_nLineBuf[HORZPIXELS];

/* We assume, that we always have more pixels horizontally than vertically.
We allocate this globally, so we don't need to increase the stack size and a maximum of
data can be held in XRAM/IRAM. */

unsigned short g_nMapping[HORZPIXELS];


/* This array contains the mapping of destination index to source index.
It can be calculated once before stretching a row and can then be reused
for all subsequent rows. */


void ScaleBitmap(unsigned short  GDI_HUGE *pic /*picture data*/,
    int nXSrcSize /*horizontal width of source picture*/,
    int nYSrcSize /*vertical width of source picture*/,
    int nXDestSize/*horizontal width of dest. buffer*/,
    int nYDestSize/*vertical width of dest. buffer*/)
{
    unsigned short  GDI_HUGE *workpic = pic;
    unsigned long dwRowCol, dwPixel;
    float fXSrcSize = (float)nXSrcSize;
    float fYSrcSize = (float)nYSrcSize;
    float fXDestSize = (float)nXDestSize;
    float fYDestSize = (float)nYDestSize;
```

[2]This leads to the so-called Moiré-Effect.

[3]It is assumed that the experienced reader will be able to understand this code without further explanations, comparing it with the pseudocode in [25] will be of additional help. The code shown here is guaranteed from the context where it was taken to not exceed the boundaries of the buffers in use, so safety checks can not only be omitted for the sake of better readability. This code actually preserves the aspect ratio of the image so at least one destination image dimension is fully covered.

```
    /* account for the aspect ratio: */
    if ((fXSrcSize/fYSrcSize) > (fXDestSize/fYDestSize))
    {
        nYDestSize = (int)(fXDestSize * fYSrcSize/fXSrcSize);
        fYDestSize = (float)nYDestSize;
    }
    else
    {
        nXDestSize = (int)(fYDestSize * fXSrcSize/fYSrcSize);
        fXDestSize = (float)nXDestSize;
    }

    /* calculate the horizontal pixel mapping: */
    for (dwPixel = 0;dwPixel<nXDestSize;dwPixel++)
        g_nMapping[dwPixel] = (int)(((float)dwPixel)* fXSrcSize/fXDestSize + 0.5);
        // we add 0.5 to get the nearest int value instead of the integer part

    /* now scale horizontally: */
    for (dwRowCol=0;dwRowCol<nYDestSize;dwRowCol++)
    {
        memcphfw (g_nLineBuf,workpic,HORZPIXELS);
        for (dwPixel = 0;dwPixel<nXDestSize;dwPixel++)
            workpic[dwPixel] = g_nLineBuf[g_nMapping[dwPixel]];
        workpic = &workpic[HORZPIXELS];
    }

    /* calculate the vertical pixel mapping: */
    for (dwPixel = 0;dwPixel<nYDestSize;dwPixel++)
        g_nMapping[dwPixel] = (int)(((float)dwPixel)* fYSrcSize/fYDestSize + 0.5);
        // we add 0.5 to get the nearest int value instead of the integer part

    workpic = pic;

    /* now scale vertically: */
    for (dwRowCol=0;dwRowCol<nXDestSize;dwRowCol++)
    {
        for (dwPixel = 0;dwPixel<VERTPIXELS;dwPixel++)
            g_nLineBuf[dwPixel] = workpic[dwPixel*HORZPIXELS];

        for (dwPixel = 0;dwPixel<nYDestSize;dwPixel++)
            workpic[dwPixel*HORZPIXELS] = g_nLineBuf[g_nMapping[dwPixel]];
        workpic = &workpic[1];
    }
}
```

It should be clear that the above code only yields visually appealing results for scaling with factors that are greater than 0.5 and smaller than 2. For scaling factors that are smaller than 0.5, Moiré-Effects will become more and more apparent. For scaling factors greater than 2, one source pixel will lend its value to 2 or more adjacent destination pixels per dimension which will make the image look rather "blocky". If better accuracy is required or Moiré-Effects should be circumvented, more sophisticated methods than the simple pixel mapping shown above must be used. For the special case of downsampling to a half or by some other power of 2, simply the arithmetic mean of two adjacent source pixels must be calculated in each direction to form the destination pixel value. Generally, a destination pixel value can be determined by a weighted sum of source pixel values. In theory, for arbitrary scaling each

source pixel is represented by a weighted sinc-Function[4] and destination pixel values that lie inbetween source pixel values are determined by summing up the values of all weighted sinc-Functions at this point. In practice however, very often only the weighted sums of the two neighbouring source pixels of a new pixel are used, since the sinc-Function slowly approaches to zero for all arguments whose absolute value is greater than 1. The interested reader might find useful information and C source code to accomplish this in [21], [26] and especially [30], which contains a discussion of various filters and their properties in the frequency domain.

[4]The sinc function is the Inverse Fourier Transform of the ideal bandlimiting low-pass filter, see [30].

4.2 Image Scaling in the IDCT Process

In this section we will revisit the inverse Loeffler-Ligtenberg-Moschytz DCT from section 3.3.4. We will first try to simplify the algorithm from Loeffler, Ligtenberg and Moschytz for a downscaling by a factor of 2. We will then try to simplify matters by downscaling to a fourth of the original image's size.

4.2.1 Image scaling in the IDCT process to half of the original size

If we want to take the average of two adjacent values of the result vector in equation (3.96), the leftmost matrix on the right side of equation (3.96) must be reduced to a matrix with four rows instead of eight. The new rows have to be the sums of two successive rows of the former matrix. This matrix then still has a symmetrical form and can be simplified to have only 4 columns by adding each two successive rows of the matrix that follows to form a newer matrix. If we continue compacting the matrices from equation (3.96) this way, the final result will be the following:

$$
\begin{aligned}
\frac{1}{2}\begin{pmatrix} f(0)+f(1) \\ f(2)+f(3) \\ f(4)+f(5) \\ f(6)+f(7) \end{pmatrix} &= \frac{1}{4\sqrt{2}} \begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \\ 0 & 1 & -1 & 0 \\ 1 & 0 & 0 & -1 \end{pmatrix} \\
&\cdot \begin{pmatrix} 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & -1 & 0 & 0 & 0 & 0 \\ 0 & 0 & \sqrt{2}(C_3-C_1) & \sqrt{2}(-C_1-C_5) & \sqrt{2}(C_3+C_7) & \sqrt{2}(C_7+C_5) \\ 0 & 0 & \sqrt{2}(C_7-C_5) & \sqrt{2}(C_3-C_7) & \sqrt{2}(C_5-C_1) & \sqrt{2}(C_3+C_1) \end{pmatrix} \\
&\cdot \begin{pmatrix} 2 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & \sqrt{2}(C_6+C_2) & \sqrt{2}(C_6-C_2) & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \end{pmatrix} \\
&\cdot \begin{pmatrix} 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 \\ 0 & 1 & 0 & 0 & 0 & 0 & 0 \end{pmatrix} \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(1) \\ \tilde{F}(2) \\ \tilde{F}(3) \\ \tilde{F}(5) \\ \tilde{F}(6) \\ \tilde{F}(7) \end{pmatrix}
\end{aligned}
\tag{4.1}
$$

Figure 4.1 shows the flowgraph that is associated with equation (4.1). Note that for scaling to half of the original size in equation (4.1) and figure (4.1), $F(4)$ is not required anymore. The multiplication by 2 can easily be done by a left shift and 5 of the constant factors can be absorbed into the quantization values. In the two-dimensional case, the final scaling by $\sqrt{2}$ (and the division by 4 to get the average value) can be reduced to a shift operation. This leads to a total complexity of 5 multiplications, one left shift and nine additions, which is better than an Arai-Agui-Nakajima DCT and taking the average of the values of adjacent pixels.

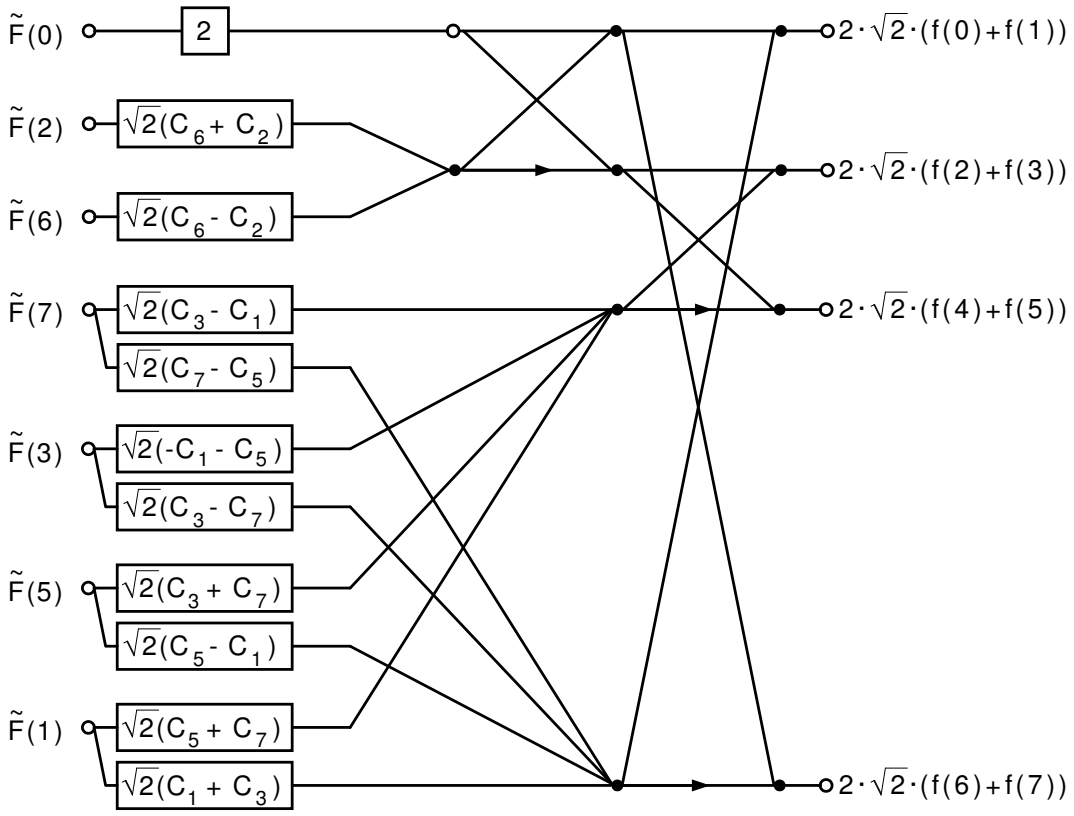

Figure 4.1: Flowgraph for the IDCT resizing to half of the original size, based on the Loeffler-Ligtenberg-Moschytz Fast DCT

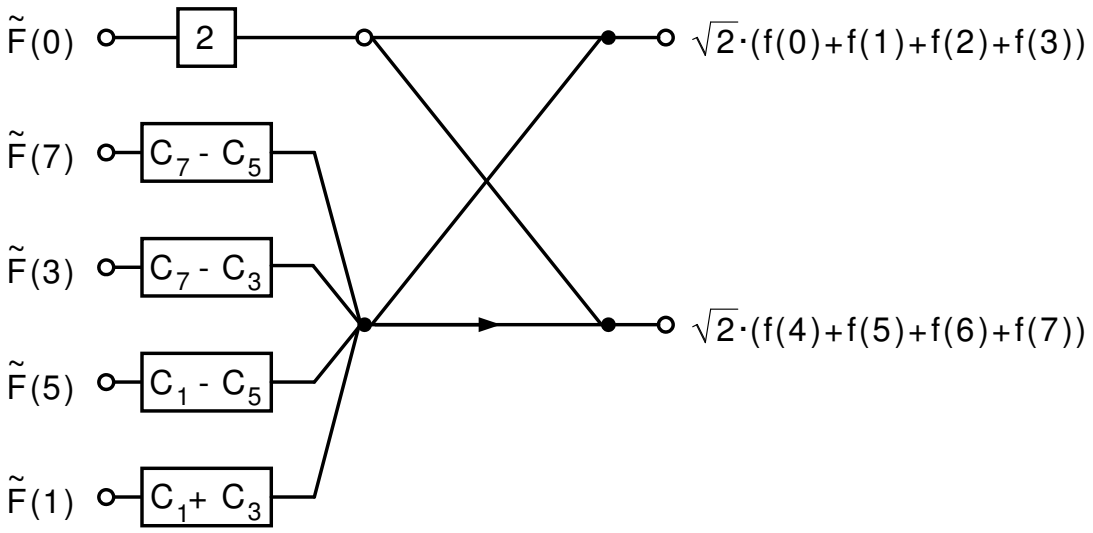

Figure 4.2: Flowgraph for the IDCT resizing to a fourth of the original size, based on the Loeffler-Ligtenberg-Moschytz Fast DCT

4.2.2 Image scaling in the IDCT process to a fourth of the original size

If we use the same technique as in section 4.2.1 to reduce the size of the IDCT process to a fourth of the original size, this time to equation (4.1), we obtain the following matrix vector product:

$$\frac{1}{4}\begin{pmatrix} f(0)+f(1)+f(2)+f(3) \\ f(4)+f(5)+f(6)+f(7) \end{pmatrix} = \frac{1}{4\sqrt{2}} \cdot \begin{pmatrix} 1 & 1 \\ 1 & -1 \end{pmatrix} \cdot \begin{pmatrix} 2 & 0 & 0 & 0 & 0 \\ 0 & C_7 - C_5 & C_7 - C_3 & C_1 - C_5 & C_1 + C_3 \end{pmatrix} \cdot \begin{pmatrix} \tilde{F}(0) \\ \tilde{F}(7) \\ \tilde{F}(3) \\ \tilde{F}(5) \\ \tilde{F}(1) \end{pmatrix} \tag{4.2}$$

Figure 4.2 shows the flowgraph that is associated with equation (4.2). Note that for scaling to the fourth of the original size in equation (4.2) and figure (4.2), $F(2)$ and $F(6)$ are not required anymore in addition to $F(4)$. Again, in the two-dimensional case, the final scaling by $4\sqrt{2}$ (and the division by 16 to get the average value) can be reduced to a shift operation. All 5 multiplicative constants can be completely absorbed into the quantization factors resulting in a computational complexity of a mere 5 additions (the scaled values of $F(7)$, $F(3)$, $F(5)$ and $F(1)$ can not be added within a single addition, as the single filled dot suggests, but rather with 3 additions).

4.2.3 Image scaling in the IDCT process to an eighth of the original size

Applying the same technique as in sections 4.2.1 and 4.2.2 we can derive the trivial operation of scaling to an eighth of the original image size during the decoding process. If we want to

take the average of 8 adjacent pixels we simply have to add the two lines of the first matrix in equation (4.2):

$$\begin{aligned}\frac{1}{8}\big(f(0)+f(1)+f(2)+f(3)+f(4)+f(5)+f(6)+f(7)\big) &= \frac{1}{8\sqrt{2}}\cdot\begin{pmatrix}2 & 0\end{pmatrix}\\ \cdot\begin{pmatrix}2 & 0 & 0 & 0 & 0\\ 0 & C_7-C_5 & C_7-C_3 & C_1-C_5 & C_1+C_3\end{pmatrix}&\cdot\begin{pmatrix}\tilde{F}(0)\\ \tilde{F}(7)\\ \tilde{F}(3)\\ \tilde{F}(5)\\ \tilde{F}(1)\end{pmatrix} = \frac{1}{2\sqrt{2}}\cdot\tilde{F}(0)\end{aligned} \tag{4.3}$$

In the two-dimensional case, for an 8×8-block of pixels, this simply means dividing the DC-value by 8.

Chapter 5

The JPEGLib

Life's most urgent question is: What are you doing for others?

– **Martin Luther King**

I never see what has been done; I only see what remains to be done.

– **Madame Curie**

The JPEGLib is a software library that is written and maintained by the *Independent JPEG Group* (IJG). The Independent JPEG Group describes itself on its website (http://www.ijg.org) as "an informal group that writes and distributes a widely used free library for JPEG image compression". The Independent JPEG Group has no real legal status or form of organization, nor is the group affiliated in any way with the ISO committee that is responsible for the standardization of JPEG. Maybe it can best be described as a group of individuals that are loosely connected via an internet mailing list and their official website, who voluntarily devote part of their time to the development and maintenance of the JPEGLib library.

In the following we will first give some historic retrospect and the design goals of the library. This will be followed by a short description of the capabilities of the library that will show what the user of the library can expect from it. We will continue with a description of the package content and a description of the processes involved in building the library. This will include building the library for an already supported platform as well as porting the library to a new platform as was the case in the project of this thesis. After that the typical usage of the library and some architectural issues will be discussed, followed by a summary of the author's experiences with the library which will conclude this chapter.

This chapter is not at all meant to serve as a replacement for the documentation that comes with the library package. The reader may consider this chapter as additional information that could help clarify things while reading along in the official documentation and more or less reflects the author's experiences from working mainly with the decompression part of the library. As an anticipation of the summary at the end of this chapter, the author could not find any serious flaws in the library, although there are some minor issues that could help improve performance and integratability of the library.

5.1 Goals, Motivation and History

The IJG started their work on a library for JPEG compression and decompression at the end of 1990, at least this is what the earliest mailing list archive (http://www.ijg.org/archives) file suggests[1]. The initial goals of the group can be summarized by quoting from one of the earliest postings of Thomas G. Lane of Carnegie-Mellon University, moderator and organizer of the group:

> "... I see the main technical issues as:
> 1. Making it work (getting the algorithms right);
> 2. Making it portable (avoiding system dependencies);
> 3. Making it fast."

or more seriously, by quoting from the current version's readme file:

> "The emphasis in designing this software has been on achieving portability and flexibility, while also making it fast enough to be useful. In particular, the software is not intended to be read as a tutorial on JPEG. [...] Rather, it is intended to be reliable, portable, industrial-strength code."

Another quote from Lane, this time from a usenet newsgroup posting in comp.graphics, alt.graphics.pixutils and other newsgroups from January 23rd of 1991, is sort of a "mission statement" and puts things a bit more into the historical context of 1991:

> "... We recognize that the net is not going to adopt any new posting format unless free or very inexpensive software is available for a wide variety of machines. We hope that we can cover enough machines to let JPEG reach "critical mass" and become a de-facto standard on Usenet. Once that happens, people with unusual hardware can adapt our source code to run on their machines. Another reason for our work is to encourage compatibility. The draft JPEG standard is an extremely loose document: a large number of decisions are left up to the individual implementor. People working in isolation are likely to produce implementations that can't actually exchange JPEG-format files. [...] By making and giving away a high-quality implementation, we hope to establish a de-facto standard for JPEG file format and to provide a painless path for people to adhere to that particular interpretation of the JPEG standard."

Lane's hope of the JPEGLib to "become a de-facto standard on Usenet" did not only come true, more important, with the Web revolution in the mid of the nineties, the JPEGLib became the de-facto standard for the World-Wide-Web. All major browser vendors up to now use the code from the IJG for rendering JPEG images, including Netscape, the Mozilla Project and Microsoft. Many other software packages such as Ghostscript, the libtiff library and numerous file conversion utilities use the IJG's JPEGLib as well. One reason for this, besides from the exceptional quality, stability and performance of the code, are the very modest licensing requirements which only require that the accompanying documentation must state that "this software is based in part on the work of the Independent JPEG Group" if only executable code is distributed that makes use of the JPEGLib. Traditionally, the

[1] Note that this is actually more than one year prior to JPEG becoming an international standard, see section 2.1.

Version	Date	Description
1	10/07/1991	First version, only JPEG baseline and extended-sequential compression processes supported, no progressive mode.
2	12/13/1991	Inclusion of config.c (later renamed to ckconfig.c), see section 5.4, bugfixes and improved portability.
3	03/17/1992	"Swapping" memory manager, hooks for application UI reflecting progress, two-pass color quantization for the decoding process, miscellaneous small speedups and bugfixes.
4	12/10/1992	Changes for improved accuracy, precision, speed and robustness, bugfixes and name changes.
4a	01/18/1993	Bugfixes and speedups.
5	09/24/1994	Almost complete redesign and rewrite: Image scaling by a factor of 1/2, 1/4, or 1/8 during the decoding process, complete change of the application programmer's interface to the library.
5a	12/07/1994	Better dithering and rounding, bugfixes.
5b	03/15/1995	Bugfixes, better documentation.
6	08/02/1995	Progressive mode support. New application "jpegtran" (see section 5.3), small changes for application programmers.
6a	02/07/1996	Robustness (versioning) and notation improved, small changes in internal structures.
6b	03/27/1998	Robustness improved, support for custom data in the compression and decompression objects, markers can be preserved in memory.

Table 5.1: Revision History of the JPEGLib

copyright owner of the IJG's code is Thomas G. Lane, because the group worried that a statement like "copyright IJG" in the comments of the source code, with the IJG having no real legal status, might be considered meaningless in a court of law if the IJG ever had to try to enforce the copyright[2].

Table 5.1 contains a short overview over the release history of the IJG's JPEGLib, along with information about important changes.

5.2 Capabilities of the JPEGLib

The JPEGLib is a library written entirely in C that implements JPEG compression (encoding) and decompression (decoding)[3]. The supported JPEG modes of operation (see section 2.2.3) are the DCT-based sequential mode, in particular the baseline process, and the progressive mode of operation with incremental decoding for on-the-fly display of pro-

[2]Today, with open-source copyrights pretty well understood, a lot of those early worries of the IJG seem pointless in retrospect.

[3]Additionally, there are some assembler files available, either to make the library work on some platforms (MS-DOS) or in order to speed up critical operations.

gressive images. The lossless and hierarchical modes of operation are not supported by the JPEGLib. As described in section 2.2, the JPEG standard does not fully specify an interchangeable file format, therefore the JPEGLib follows the JFIF file format ([11]) which was introduced in section 2.3. The JPEGLib also does not support arithmetic coding, since this is subject to several patents. Instead, Huffman encoding and decoding is implemented in a very close way to the coding propositions of the JPEG standards document ([5]). There are three different DCT implementations for both the FDCT and IDCT that are runtime options: a slow but accurate floating-point variant using the Arai-Agui-Nakajima DCT (see sections 3.3.6 and 3.3.7), a faster yet slightly less accurate fixed-point variant using the Loeffler-Ligtenberg-Moschytz DCT (see sections 3.3.3 and 3.3.4), and a fast but much less accurate fixed-point variant using the Arai-Agui-Nakajima DCT. Additionally, specialized fixed-point algorithms based on the Loeffler-Ligtenberg-Moschytz DCT allow downscaling to the half, a fourth or to an eighth during decoding as described in sections 4.2.1, 4.2.2 and 4.2.3. The JPEGLib is both able to support 8 bit and the rarely used 12 bit sample data precision, but not simultaneously, since this is a compile-time decision. In addition to decoding into a 24-bit color format, the JPEGLib also contains a color decoding backend that can create dithered output, using ordered dithering or Floyd-Steinberg dithering (see also [10], [12], [16], [24] and [28]).

5.3 The JPEGLib package content

The JPEGLib source code and documentation comes as a compressed file that can be downloaded from http://www.ijg.org/files/. The most current version at the time of writing is version 6b, dating from March 27^{th} of 1998, and can be found at the aforementioned URL as the file jpegsrc.v6b.tar.gz. If this archive is unpacked, the files created can be roughly categorized into the following 3 groups (see also the file filelist.doc that comes with the library): JPEGLib core, sample applications and documentation. A detailed description of every file's purpose can be found at the beginning of each source file. A brief description of all files can be found in the file filelist.doc. As a simple file naming convention, every source and header file of the JPEGLib core starts with a "j" and the character following it in most cases denotes the part to which this module belongs: If it is a "c", the module belongs to the compression/encoding part, if it is a "d", the module belongs to the decompression/decoding part of the library. Before a successful build of the library can be made, the library has to be adapted to the specific environment where it is to be used. See section 5.4 and the JPEGLib's file install.doc for more information on this topic.

When compiled successfully, a file named libjpeg.lib (the name may differ for different platforms or custom makefiles) and 3 sample applications are created whose purpose is described as follows:

- cjpeg: A command-line utility to create JPEG files from different bitmap source formats.
- djpeg: A command-line utility to create different bitmap destination formats from a JPEG source file.
- jpegtran: A command-line utility for lossless transcoding from JPEG source files into JPEG destination files of different formats.

5.4 Adapting the JPEGLib to different platforms and compilers

As laid out before, the JPEGLib was written with portability across different environments in mind. This includes different computer architectures, operating systems and compilers. To account for these different environments, all compile-time relevant settings are collected in one header file (jconfig.h) and are also reflected in the proper use of a suitable makefile. Proper use means here: The correct makefile has to be identified (or adapted from existing ones) and edited to choose the correct memory manager for the environment in question.

5.4.1 Determining the correct jconfig.h file and the correct makefile

For popular non-Unix environments, preconfigured makefiles and header files are already supplied, so making the JPEGLib ready to be successfully compiled takes only two steps: Renaming the right variant of jconfig.* to jconfig.h and renaming the right version of makefile.* to makefile (e.g. on a Win32 based system for the Microsoft C compiler, jconfig.vc is to be renamed to jconfig.h and makefile.vc is to be renamed to makefile). After that step, the JPEGLib is ready to be compiled by a make tool (i.e. the "nmake" tool in the aforementioned case of a Win32 based system and a Microsoft C compiler). This way, the most important (and also some historical) non-Unix environments are covered, such as Amiga, Apple Macintosh, Atari ST, Digital VMS, Borland C on MS-DOS and OS/2, GNU C on MS-DOS, Microsoft C for MS-DOS and Win32 platforms, Watcom C on MS-DOS, Win32 platforms and OS/2.

In the case of a Unix-like environment, the GNU C compiler is required and the two files are created by a shell script (configure).

For the not so popular environments like the one covered in this thesis, a tool can be built that creates the jconfig.h file. For this purpose, the file named ckconfig.c needs to be successfully compiled into an executable. Running the executable then creates the jconfig.h file. For the creation of jconfig.h, ckconfig.c works with some heuristics to determine the correct values for the jconfig.h file to be created. If the compilation of ckconfig.c fails, fixing the cause of this error in ckconfig.c finally yields the correct result in the jconfig.h file to be created. This way, a successful compilation of ckconfig.c finally creates a jconfig.h file for the JPEGLib that is suitable for the environment under which ckconfig.c is compiled, and thus correct compile-time constants can be used for making the JPEGLib library. The compile-time constants mostly include properties of the compiler and Operating System (OS) in use, such as:

- Does the compiler or the OS use a flat memory model?
- Does the compiler use function prototypes?
- Does the compiler have the "boolean" data type?
- Does the compiler have unsigned variants of the char and short data type?
- Does the compiler use an unsigned right shift?

Since ckconfig.c creates a file, its usage is a bit problematic for embedded systems, where there typically is no file system available. In the case of this thesis, the code of ckconfig.c

File name	Description
jmemansi.c	Swaps out data to unnamed temporary files using the ANSI standard library routine tmpfile().
jmemname.c	Swaps out data to named temporary files.
jmemnobs.c	Doesn't swap out data to files("nobs" in jmemnobs stands for "No backing store"), relies on enough main or virtual memory. Should compile on all platforms.
jmemdos.c	Customized memory manager for MS-DOS.
jmemdos.c	Customized memory manager for the Apple Macintosh.

Table 5.2: Memory manager implementations supplied by the JPEGLib

was adapted to transmit the file content over the serial line, where it could be captured by a program running on the development system.

In very rare cases, such as the embedded system being used throughout this thesis, a second header file, jmorecfg.h, needs to be adapted as well. In this file it is determined, among other things, what the preprocessor will evaluate for the pseudo-keyword FAR, that is used within the JPEGLib for segmented architectures, like MS-DOS. Using this pseudo-keyword, different pointer arithmetics can be done on such platforms.

For a suitable makefile for the not so popular environments, two generic makefiles (makefile.ansi and makefile.unix) are provided as starting points for ANSI compliant compilers and compilers without function prototype support. In the case of this thesis, makefile.ansi was choosen and adapted for the usage with the Cygnus port of the GNU make utility for Win32.

5.4.2 Choosing the right memory manager

The makefile for the JPEGLib always contains a variable named SYSDEPMEM which is assigned a value of one object file that contains the implementation of a memory manager. A memory manager for the JPEGLib assures that even on platforms with segmentation, such as MS-DOS or on systems with little memory, JPEG files can be encoded or decoded successfully, even if the images are much bigger in size than the available memory of the system.

There are 5 different memory manager implementations provided with the JPEGLib which are implemented in the files given in table 5.2. If none of these memory managers fits for a particular purpose, a custom one can be written for a particular environment or an existing one can be adapted, as was the case with this thesis. Also, the SYSDEPMEM variable in the makefile needs to be changed such that it points to the object file associated with the desired memory manager implementation.

Reasons for employing a memory manager scheme instead of the the naïve use of the standard C-Runtime function malloc at all places where memory needs to be allocated dynamically, can be the following:

- On some systems, particularly segmented architectures such as MS-DOS or the embedded system used throughout this thesis, malloc simply sometimes cannot allocate

large enough memory buffers[4] and therefore cannot always satisfy the requirements of the caller.

- On some systems, OS system calls or calls into heap managers, e.g. when used in multithreaded and SMP environments on more than one CPU, are much more efficient than the standard C-runtime allocator malloc.

As a result, the JPEGLib's memory manager uses two different allocator/deallocator function pairs throughout its own code. The first one is jpeg_get_small and jpeg_free_small, which are used for small blocks of memory. For bigger blocks of memory, the jpeg_get_large and jpeg_free_large functions are used[5]. Another advantage of the JPEGLib's particular implementation is, that all dynamic memory allocation that happens during encoding or decoding of one image is tracked in so-called "memory pools" that are associated with the compression or decompression object[6] and thus per-image data can be freed automatically in one fell swoop after this process by invoking one single cleanup function (jpeg_finish_compress and jpeg_finish_decompress). Also, all dynamic memory allocation that happens during encoding or decoding per one compression or decompression object can be freed automatically by invoking one single cleanup function (jpeg_destroy_compress and jpeg_destroy_decompress). This way, the programmer does not need to bother about potential memory leaks inside the library that could happen by not tracking all dynamically allocated memory.

5.5 Usage and Architectural Issues

This section will deal with a few design issues of the JPEGLib. The JPEGLib, though written in C, follows a few object-oriented principles that are well worth being explained. In particular this is inheritance, encapsulation via the usage of two fundamental composite data types and polymorphism through function pointers. There are two reasons why the developers of the JPEGLib preferred C over C++ for the implementation of their library: First, in the beginning of the nineties of the 20^{th} century, using C was the only feasible choice for portable code and C++ was not yet well standardized. Secondly, the available free C++ compilers were neither very good[7] nor widely portable.

Another reason could be to have chosen C as the least common denominator, because virtually all platforms have at least a more or less decent C compiler. This way also code written in other languages can use the IJG's code either directly (e.g. C++ via the extern "C" keyword) or indirectly via dynamically linked libraries.

In the following we will first show what the typical code sequences are for encoding and decoding and will then proceed to the encoder and decoder objects and show how inheritance and polymorphism are achieved.

[4] The environment used throughout this thesis allows malloc to allocate only blocks up to a size of $2^{14} - 1$ bytes, generally malloc can only allocate $2^{\text{sizeof(size\_t)}} - 1$ bytes, because the formal parameter of malloc is of type size_t. For MS-DOS or the Win16 programming environment this is a mere 65535 bytes.

[5] A block allocated by jpeg_get_large also is a FAR pointer. Whether or not FAR is a special keyword and thus actually implies a special way of doing pointer arithmetics or whether it is simply expanded by the C preprocessor to nothing, depends on settings made in jconfig.h and jmorecfg.h.

[6] Compression and decompression in the JPEGLib involves a compression/decompression object that can be reused for subsequent encoding or decoding processes of different images, see section 5.5.3.

[7] Even at the time of writing, C++ compiler implementations seem to be inherently flawed, whereas writing good and efficient C compilers is a well researched topic for more more than 30 years.

5.5.1 Typical code sequences for encoding

This section will show how typical code sequences will look like for the application developer who wants to use the encoding part of the JPEGLib from within his code. We will adapt the pseudocode-like approach that is used in the library's libjpeg.doc file and we will comment on the involved steps.

For the compression of images, the outline of the involved operations is as follows:

- Allocate and initialize a JPEG compression object.
- Specify the destination for the compressed data (e.g. a file).
- Set parameters for compression, including image size and colorspace.
- Call jpeg_start_compress(...);
- Do the following loop:
  while (scan lines remain to be written)
  jpeg_write_scanlines(...);
- Call jpeg_finish_compress(...);
- Release the JPEG compression object.

The first step, allocating a JPEG compression object, is usually done by allocating it on the stack, as an automatic variable. The size of such an object of type jpeg_compress_struct on a 32-bit system is typically 360 bytes[8], therefore on systems where the stack is a scarce resource because it has to fit into a single address segment of the microprocessor[9], allocating it on the stack could be a problem[10]. The initialization of the compression object is done by allocating an error manager object (jpeg_error_mgr) on the stack, assigning its address to a member of the compression object and by calling the macro jpeg_create_compress. The error object[11] specifies the amount of diagnostic output during encoding via a so-called "Trace Level". It also contains function pointers as structure members that point to routines for emitting and formatting error and diagnostic messages[12].

In case a JPEG file should be created, specifying the destination for the compressed data is especially easy: Simply jpeg_stdio_dest needs to be called with a FILE* that was opened for writing. In case the JPEG data should have a different destination, e.g. should be transmitted over a serial line or written into memory or a memory-mapped-file, a so-called "destination manager" needs to be written. The interested reader will find more information about this in the JPEGLib's libjpeg.doc file.

Setting the parameters for compression consists merely of assigning values to members of the compression object and calling the JPEGLib's core function jpeg_set_defaults, which

[8] 284 bytes on the embedded system in use throughout this thesis.

[9] This is a problem for operating environments such as MS-DOS, Win16 or the system used throughout this thesis.

[10] In such cases the object must be allocated dynamically, i.e. via malloc. In this case, the programmer is also responsible for releasing the object to the heap after usage.

[11] The error object can be tailored to one's own needs since a prototype is defined in the header file jpeglib.h. This way the default function pointers can be overridden with pointers to own functions.

[12] These messages usually come from a hardcoded message table, see jerror.h.

sets all members of the compression object to reasonable default values. Input colorspace is set via the functions jpeg_default_colorspace or jpeg_set_colorspace[13].

After calling jpeg_start_compress, a loop over all input scanlines must be performed, each time calling jpeg_write_scanlines with a pointer to a buffer of one or more lines of the input image as the parameter.

A call to jpeg_finish_compress will now clean up all per-image data of the compression object and a call to jpeg_destroy_compress will finally clean up all per-object data of the compression object. If the compression object was allocated on the stack, it will be automatically released by leaving the current stack frame.

5.5.2 Typical code sequences for decoding

This section will show how typical code sequences will look like for the application developer who wants to use the decoding part of the JPEGLib from within his code. Again, we will adapt the pseudocode-like approach from the library's libjpeg.doc file and we will comment on the involved steps. For the decompression of images, a similar scheme to that of compression is used. The outline of the involved operations is as follows:

- Allocate and initialize a JPEG decompression object.
- Specify the source of the compressed data (e.g. a file).
- Call jpeg_read_header() to obtain image information.
- Set parameters for decompression.
- Call jpeg_start_decompress(...);
- Do the following loop:
  while (scan lines remain to be read)
  jpeg_read_scanlines(...);
- Call jpeg_finish_decompress(...);
- Release the JPEG decompression object.

The first step, allocating a JPEG decompression object, is usually done by allocating it on the stack, like the compression object in section 5.5.1. The size of such an object of type jpeg_compress_struct on a 32-bit system is typically 432 bytes[14], so again, allocating it on the stack can be a problem in some environments. Similar to the steps involved in compression, initialization of the decompression object is done by allocating an error manager object on the stack, assigning its address to a member of the decompression object and by calling the macro jpeg_create_decompress.

In case the source of compressed data is a file, the second step is quite easy: All that needs to be done is to supply a FILE* to a file that was opened for reading to the function jpeg_stdio_src. In case JPEG data is in memory like it was the case in the project of this thesis or is transmitted in pieces via the serial line, a so-called data source manager has to be written for this purpose (see also the documentation in the JPEGLib's file libjpeg.doc).

[13] Possible input color spaces are grayscale, YCbCr, RGB, CMYK, YCCK.
[14] 338 bytes on the embedded system in use throughout this thesis.

The next step, calling jpeg_read_header, obtains useful information such as image resolution and color space. This is particularly useful if memory is limited as it was in the project of this thesis. At this point the decision was made whether to downscale the picture in the decoding process to half, to a fourth or to an eighth. In any case, knowing the horizontal resolution of the image allows the allocation of big-enough buffers for the line-by-line processing in the while-loop that follows.

Setting parameters for decompression involves specifying which IDCT variant should be used or whether dithering should be used for color-quantized output.

After the call to jpeg_start_decompress, the while-loop is started and one or more line buffers are passed with each call to jpeg_read_scanlines for decoding the image line by line or several lines at once per call to jpeg_read_scanlines.

Like in the encoding process, a call to jpeg_finish_decompress will now clean up all per-image data of the decompression object and a call to jpeg_destroy_decompress will finally clean up all per-object data of the compression object. If the decompression object was allocated on the stack, it will be automatically released by leaving the current stack frame.

5.5.3 The encoder and decoder objects

The encoder and decoder objects are composite data types that are defined in the header file jpeglib.h. A composite data type in the C programming language is called a "struct". A struct is similar to a class in C++ in that it supports some sort of encapsulation, but it does not support access specifiers like private, protected or public as in C++ or Java. A struct in C also has no notion of member functions[15], let alone polymorphism, which is achieved in C++ through virtual member functions and is the default in Java. Also, C does not provide a builtin way to derive one struct from another to achieve inheritance. We will see in the following, how the designers and implementors of the JPEGLib achieved these fundamental principles of object-oriented design in C nevertheless:

The encoder and decoder objects both share a set of identical members which are at the beginning of the binary layout of each object. This allows a pointer to a compression or decompression object to be cast into a pointer to a struct jpeg_common_struct, which contains only those common members. This is an effective way of introducing inheritance in C and allows quite a lot of functions to operate on both encoder and decoder objects. If this were not the case, all the functions that take a pointer to a jpeg_common_struct would need to be duplicated: one variant for the compression object and another for the decompression object. This way the structs for the encoder and decoder objects, jpeg_compress_struct and jpeg_decompress_struct, can be considered to be derived from the struct jpeg_common_struct.

Polymorphism in the JPEGLib is achieved in a very similar way to the way C++ achieves this with virtual functions. In C++, each object that is of a class that has virtual functions or that is derived from a class with virtual functions, has a hidden member variable that points to a table with function pointers, the so-called "virtual function table". The JPEGLib's compression and decompression objects contain pointers to "managers" or "subobjects" that are responsible for different tasks during encoding or decoding, such as performing the DCT, performing entropy encoding or decoding, color-conversion, upsampling or downsampling, etc. These "managers" in turn are again composite data types,

[15] Member functions are also often called "methods".

but most of them (see the header file jpegint.h) contain only function pointers and thus have the same functionality as the aforementioned virtual function tables[16]. By assigning these function pointers different values, the functionality of the managers can be changed without changing the binary layout of the compression and decompression objects. For instance, selection of one of the three possible DCT variants per object consists of assigning one out of three functions with identical signature to the function pointer struct member of the subobject that is responsible for doing the discrete cosine transform. Also, complete managers can be replaced by custom managers, as long as the binary layout of the standard manager is a subset of the custom manager, i.e. if the standard manager is of size n bytes, the custom manager must be at least of size n bytes and the first n bytes of both managers must have the same meaning. This again, is nothing else than the concept of inheritance.

5.6 Summary

As a summary, the JPEGLib can be considered a modular and object-oriented library that uses very elegant ways to achieve the fundamental principles of object-orientation despite the fact that the C programming language has no intrinsic language support for these. In addition, by using subobjects within the two main types of objects for compression and decompression, there is no deep hierarchy of inheritance, which circumvents the problem of "fragile base classes" as it is often encountered in deeply nested C++ class hierarchies with overuse of polymorphism. Since the focus of this thesis was on the decoding part, this summary also only covers the author's experiences with this part of the library.

It turned out that the JPEGLib was the right choice for the purpose of decoding JPEG files on the embedded system used throughout the project of this thesis. The fact that the library was written with a high degree of portability in mind made porting much easier. Especially the carefully written code parts for MS-DOS that via the pseudo-keyword FAR allow different pointer arithmetics where this is necessary, made it possible to overcome the limitations imposed by the segmentation of the microcontroller's address space[17]. Also the modular architecture of the library made it possible to adapt for the requirements of this thesis' project by using a custom data source manager that reads JPEG data from main memory while accounting for the segmented architecture of the controller's address space. Little effort was required to exchange other parts of the library by assigning new functions to the function pointers of various subobjects of the decompression object or even by replacing complete subobjects with custom implementations.

Nevertheless a few observations could be made about potential improvements of the library:

- The algorithms for the downscaled IDCTs to half and especially to a fourth of the original image's size (see sections 4.2.1 and 4.2.2) can be improved drastically by absorbing the multiplicative factors for the dequantized DCT coefficients into the quantization tables. This way, e.g. for downscaling to a fourth, only additions are required and 5 multiplications per one row or column IDCT can be saved. Maybe the code of the JPEGLib uses these multiplications deliberately, because this way

[16] This concept is also known as the concept of *Abstract data types*. The parallels to "interfaces" as they are used in CORBA, COM or Java is obvious.

[17] The microcontroller used throughout this thesis has a memory architecture that partitions the address space into so-called "pages" of size 2^{14} bytes.

better accuracy is achieved. But maybe it was just an oversight that the multiplicative constants like in the JPEGLib's implementations of the Arai-Agui-Nakajima DCT can be absorbed into the quantization table. At least for a few test images, no noticeable difference could be found when using this optimization.

- The JPEGLib employs the row- and columnwise approach for all variants of the DCT. The following comment from the source code appears in a similar form in all modules that deal with the calculation of the DCT:

  > "A 2-D DCT can be done by 1-D DCT on each row followed by 1-D DCT on each column. Direct algorithms are also available, but they are much more complex and seem not to be any faster when reduced to code."

  The author considers this statement to be very questionable, since e.g. Feig's algorithms for the scaled 2D FDCT and IDCT as described in sections 3.4.3 and 3.4.4 require significantly less computational effort. A potential obstacle however for using Feig's algorithms in the JPEGLib could arise from patent issues: There is an indication in Feig's original article ([8]), that these algorithms are included in a U.S. patent application. Patent recherche showed that only one patent (patent number 5293434) was granted to Feig (and Pennebaker, one of the authors of [22]), that explicitly deals with the 2D DCT, and it is unclear to the author whether it actually uses Feig's 2D DCT as described in sections 3.4.3 and 3.4.4. Also: This patent was filed in 1991, Feig's original article ([8]) dates back to 1990 and states that the patent was already filed way back in 1989.

- For an embedded system it makes no sense to have diagnostic messages or error messages during decoding an image, because there is no command prompt or other means for displaying such output. There are only two alternatives: either decoding succeeds or it fails, in which case it is sufficient to show to the user that there was a problem, but not which problem. Therefore it would have been a good idea if the implementors of the JPEGLib had made diagnostic and error output a compile-time option. In the case of this thesis, some measurable improvement could be made by simply expanding the macros WARNMS,TRACEMS, ... in the header file jerror.h to nothing. Further improvement could probably be made by incorporating both the condition check for a diagnostic message and the ID for the diagnostic message into a macro which simply expands to nothing if this compile-time option is not used. This way those applications that use diagnostic output could continue to use it, while for others both the condition check and the diagnostic output could be completely omitted, resulting in higher performance. It could be argued, that users should write their own error handling code, but including this into the default error handler would not hurt and would relieve users from tinkering with an own error handler.

- Even in the case that errors and warnings and other diagnostic output is desired, the suggestion in libjpeg.doc under "Error handling" to override some of the method pointers in the jpeg_error_mgr struct after calling jpeg_std_error() is not really helpful as to internationalization and localization. Even in this case, error messages still come from the file jerror.h as ANSI strings in plain English. But not all the world is English and in the age of UNICODE, ANSI strings more and more appear to be some sort of an anachronism. Again it could be argued, that users should write their own error

handler, but few if any users will probably do this. Most users will probably stick with the hardcoded error messages from jerror.h in ANSI and plain English, therefore the author's view is, that the default error manager should be made more flexible and extensible to account for the aforementioned issues.

- Overuse of macros: During the phase of porting the JPEGLib to the microcontroller used during this thesis, an occasional error occurred during the decoding process in the macro HUFF_DECODE in the source file jdhuff.c. The macro HUFF_DECODE turned out to be the following monster (see header file jdhuff.h):

```
#define HUFF_DECODE(result,state,htbl,failaction,slowlabel) \
{ register int nb, look; \
  if (bits_left < HUFF_LOOKAHEAD) { \
    if (! jpeg_fill_bit_buffer(&state,get_buffer,bits_left, 0)) {failaction;} \
    get_buffer = state.get_buffer; bits_left = state.bits_left; \
    if (bits_left < HUFF_LOOKAHEAD) { \
      nb = 1; goto slowlabel; \
    } \
  } \
  look = PEEK_BITS(HUFF_LOOKAHEAD); \
  if ((nb = htbl->look_nbits[look]) != 0) { \
    DROP_BITS(nb); \
    result = htbl->look_sym[look]; \
  } else { \
    nb = HUFF_LOOKAHEAD+1; \
slowlabel: \
    if ((result=jpeg_huff_decode(&state,get_buffer,bits_left,htbl,nb)) < 0) \
{ failaction; } \
    get_buffer = state.get_buffer; bits_left = state.bits_left; \
  } \
}
```

Note that the author does not object against the coding style of this snippet or the usage of "goto"[18], it simply is impossible to debug something like that[19]. In addition, PEEK_BITS and DROP_BITS used in this macro are again macros. This code had to be expanded manually by the author of this thesis in order to find and fix the error.

It can be considered a matter of taste or performance whether this code should rather be a function or a macro. If performance is the reason for implementing it as a macro, the author of this thesis would probably have written this as normal code instead of a macro, given the fact that this macro is invoked only 6 times.

[18] Despite the doctrine "goto considered harmful" ([7]), goto is the right choice when it comes up to performance as in this library or in C-runtime libraries.

[19] The C preprocessor expands the whole macro to a single line.

Chapter 6

JPEG Decoding on the Micronas SDA 6000 Controller

Don't you know you fool, you never can win?
Use your mentality, wake up to reality.

– From the song "I've got you under my skin" by Cole Porter (from the movie "Born To Dance", 1936)

This chapter contains the results of the project pursued throughout this thesis. For the maintainer of the code base that was developed, it contains a description of the projects used for decoding on the SDA 6000 controller as well as a detailed description of the changes to the JPEGLib, that were required to make the JPEGLib run on this hardware. The aim of these sections is to give the maintainer of the source code explanations to the code changes and they can be probably best understood while using a tool that shows the differences between the standard JPEGLib, as distributed by the IJG, and the code written throughout this thesis. Following this will be a section that explains the changes to the JPEGLib for a modified version of the IDCT algorithm that scales the image to a fourth, as suggested in section 4.2.2. The sections following this are descriptions of the custom managers that were written to replace parts of the JPEGLib. and finally an overview of the actual performance that is possible with the SDA 6000 controller.

6.1 Project descriptions

The controller being used throughout this thesis is the SDA 6000 from Micronas, Munich. The controller's code name during its development was "M2" and this name has been retained until the time of writing, even in official documentation. Therefore and for the sake of brevity, we will use the controller's code name in the following.

For the decoding of JPEG files on the M2, two projects exist that can both be used for further development. Both projects compile cleanly without any warnings or errors in the highest warning level, besides from third-party code. Both require the toolchain "TASKING C/C++ for C166/ST10 v7.0 r1" from Tasking Inc. and reside under the corporate-wide prescribed directory d:\work in separate directories:

- Under d:\work\ae7 a project can be found that initially served as the starting point in that it contained routines for drawing on the TV screen with the primitives from the Graphics Device Interface (GDI) of the M2. This project was mainly used during the phase of porting the JPEGLib to the M2. In this project, the JPEG decoder part is running concurrently with other processes under the real-time operating system OSE. The associated project file can be found under d:\work\ae7\as the file AE7.psp. The binary that is created by a successful build of this project is the file ae7.out in the same directory. After the need was encountered to abandon the operating system for performance reasons, this project was superseded by the second project.
- The second project is located under d:\work\mini and emerged from an OS-less project, named "warmup" that was provided courtesy of Micronas, the manufacturer of the M2. This project allows the JPEG decoder to run at maximum performance as a single task on the M2 without any underlying OS. Additionally, it can be compiled to use either the standard GDI for the M2 in 4-4-4 mode or the experimental "MINI GDI"[1] that switches the M2 into 5-6-5 mode. The associated project file can be found under d:\work\mini\target\warmup\Tasking as the file Warmup.pjt. The binary that is created by a successful build of this project is the file warmup.out in the same directory.

Both projects are built via the Integrated Development Environment that ships with the toolchain.

6.2 Changes to the JPEGLib

Both projects use the same copy of the JPEGLib which was slightly modified during the porting phase to adapt to this previously unsupported platform. This variant of the JPEGLib can be found in the following directory:
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeglib.
The binary image of the library itself is the file libjpeg.lib. In order to successfully build the library a command console must be started and the current directory must be changed to this directory, followed by invoking "make". The suggested make utility comes from the Cygnus port of the GNU compiler for Win32. A custom jconfig.h was created from ckconfig.h and the library's makefile.ansi was modified and used as the library's makefile as described in section 5.4. The JPEGLib had to be modified at surprisingly few places in order to run on the M2. In order to get it compiled, actually no changes were required. However, it turned out that the standard memory manger (memnobs.c) could not allocate large enough blocks, so it was modified accordingly. The following changes to the memory manager were required:

- Two global function pointers were introduced that have identical function signatures to the standard allocator/deallocator pair malloc and free in jconfig.h:

```
#include <stdio.h> //for definition of data type size_t
typedef void* (*memalloc)(size_t);
extern memalloc g_fnMemAlloc;
```

[1] The MINI GDI library was supplied halfway through during this thesis from Tara Systems, the vendor for the official GDI. For information on how to work with it see [17].

```
typedef void (*memfree)(void *);
extern memfree g_fnMemFree;
```

The implementation and initialization of these global function pointers can be found in the modified variant of jmemnobs.c:

```
memalloc g_fnMemAlloc = NULL;
memfree g_fnMemFree = NULL;
```

- A custom implementation of jpeg_get_large and jpeg_free_large (see section 5.4.2) was made in jmemnobs.c:

```
GLOBAL(void FAR *)
jpeg_get_large (j_common_ptr cinfo, size_t sizeofobject)
{
  if (g_fnMemAlloc)
    return (void FAR *) g_fnMemAlloc(sizeofobject);
  else
    return NULL;
}

GLOBAL(void)
jpeg_free_large (j_common_ptr cinfo, void FAR * object, size_t sizeofobject)
{
  if (g_fnMemFree)
    g_fnMemFree(object);
}
```

By using these function pointers, the actually used allocator could be set from outside. This happens in the files GraphicsDemo.cpp under d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src and in the file startup.c under d:\work\mini\target\warmup\Source where these function pointers are assigned the addresses of the functions MemoryAlloc and MemoryFree whose implementation can be found in the file memalloc.c in the directory d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src.

The implementations of MemoryAlloc and MemoryFree are as follows:

```
void * MemoryAlloc (size_t st)
{
    XHandle h = GdiAllocMemory(st+sizeof(XHandle), 0x80, GDI_ALLOC_USE_GAPS);
    if (GDI_NULL_HANDLE != h)
    {
        unsigned char huge *lp = (unsigned char *)GdiLockMemory(h);
        memcpy((void *)lp, (void *)&h, sizeof(XHandle));
        lp = &lp[sizeof(XHandle)];
        return (void *)lp;
    }
    else
        return NULL;
}
```

```
void  MemoryFree (void *p)
{
    XHandle x;
    unsigned char huge *lp = (unsigned char *)p;
    lp -= sizeof(XHandle);
    x = *((XHandle *)lp);
    GdiUnlockMemory(x);
    GdiFreeMemory(x);
}
```

MemoryAlloc allocates with the help of the GDI routines (which can allocate much more memory than the standard allocator malloc) memory that is exactly the size of an XHandle larger than required. The memory handle that is returned from the library is then locked and the value of the memory handle is written at the beginning of this piece of locked memory. The address that starts after this location is then returned to the caller and is exactly the size the caller requested. MemoryFree first determines the address where the value of the associated handle can be found by decrementing the supplied address by sizeof(XHandle), stores this value in a temporary variable and unlocks and releases the memory via this variable. With this scheme via function pointers, the GDI library can lend its capability to allocate big amounts of memory to the JPEGLib without the necessity to merge these libraries or make them too dependent on each other. In order for the JPEGLib to account for the segmented architecture of the M2, jmorecfg.h had to be changed as well so the pseudo-keyword FAR is replaced by the C preprocessor with "huge":

```
#ifdef NEED_FAR_POINTERS
#define FAR  huge
#else
#define FAR
#endif
```

Of course, for this to work, NEED_FAR_POINTERS in jconfig.h has to be defined as well:

```
#define NEED_FAR_POINTERS
```

Additionally, the JPEGLib needs a primitive for copying data blocks, taking into account the segmented architecture of the M2. This is done by specifying the macro FMEMCOPY in jconfig.h:

```
#define FMEMCOPY(dest,src,size) memcphhb((void huge *)(dest), \
                                         (const huge FAR *)(src), (size_t)(size))
```

The function memcphhb is a variant for the standard C-runtime function memcpy that takes "huge Pointers" as arguments, thus it does 32-bit pointer arithmetics instead of the M2's standard 14-bit pointer arithmetics.

The real difficulty in getting the JPEGLib to work was to provide a custom data source manager. Section 6.5 will elaborate on this topic. For now we will continue with the changes made to the JPEGLib.

The JPEGLib employs a function (jzerofar) in jutils.c that initializes memory with zeroes. This can be either defined via a macro (FMEMZERO) as in the case of FMEMCOPY, otherwise via a slow method that initializes the memory byte-by-byte with slow FAR pointer arithmetics. For the M2 there is no runtime primitive for that purpose, a faster variant could be found by separating the memory in pages and using faster standard pointer arithmetics:

```
GLOBAL(void)
jzero_far (void FAR * target, size_t bytestozero)
/* Zero out a chunk of FAR memory. */
/* This might be sample-array data, block-array data, or alloc_large data. */
{
#ifdef FMEMZERO
  FMEMZERO(target, bytestozero);
#else

/* // Standard implementation. Slow, uses FAR ptrs:
  register char FAR * ptr = (char FAR *) target;
  register size_t count;

  for (count = bytestozero; count > 0; count--) {
    *ptr++ = 0;
*/


    // We first determine the number of bytes that are left till the beginning
    // of the next page in variable iRemainder. Then we zero out either bytestozero
    // bytes (if bytestozero<iRemainder) and return, or we zero out the rest of the
    // current page.
    // At that point we set ptr to the beginning of the next page and decrement
    // bytestozero by the number of bytes we already zeroed out. From this moment on
    // iRemainder is always the size of one page (0x4000). Then we'll start all over
    // again the whole thing.

    register char * ptr = (char *) target;
    register unsigned int iRemainder = 0x4000 - ((unsigned int)ptr)%0x4000;
    register size_t count;

    for (;;)
    {
        if (bytestozero<iRemainder)
        {
            for (count = bytestozero; count > 0; count--)
                *ptr++ = 0;
            return;
        }
        else
        {
            for (count = iRemainder; count > 0; count--)
                *ptr++ = 0;
        }
        target =   (void FAR *)((char FAR *)target+iRemainder);
        ptr = (char *) target;
        // now ptr is exactly on the start of a new page.
        bytestozero -= iRemainder;
        iRemainder = 0x4000;
    }

#endif
}
```

As the comment points out, this implementation first calculates the number of bytes until the next page starts and zeroes out these or the maximum number of bytes. This can be done with a normal pointer (as opposed to a huge pointer), because no page boundary is

crossed. Then the pointer is assigned the address of the next page and again the number of bytes until the next page or the maximum number of bytes are zeroed out until no more bytes to zero are left.

In order to keep the binary as small as possible (and so compile and link cycles), several options in the file jmorecfg.h were turned off[2]:

```
#undef DCT_FLOAT_SUPPORTED /* floating-point: accurate, fast on fast HW */
#undef C_PROGRESSIVE_SUPPORTED     /* Progressive JPEG? (Requires MULTISCAN)*/
#undef D_PROGRESSIVE_SUPPORTED     /* Progressive JPEG? (Requires MULTISCAN)*/
#undef SAVE_MARKERS_SUPPORTED     /* jpeg_save_markers() needed? */
#undef BLOCK_SMOOTHING_SUPPORTED   /* Block smoothing? (Progressive only) */
#undef QUANT_1PASS_SUPPORTED     /* 1-pass color quantization? */
#undef QUANT_2PASS_SUPPORTED     /* 2-pass color quantization? */
```

In the case of QUANT_1PASS_SUPPORTED things are a bit complicated: For dithering, this macro must be defined. For undithered output it is not required and has a negative effect on performance. Enabling dithering is a compile-time option (see section). The last optimization to be done in jmorecfg.h is to define a data type for fast multiplication:

```
#ifndef MULTIPLIER
#define MULTIPLIER  short /* type for fastest integer multiply */
#endif
```

For performance reasons, as described in section 5.6, diagnostic output was made a compile time decision by introducing a macro _DEBUG. In case _DEBUG is defined, the JPEGLib does normal diagnostic output, in case it is not defined, the macros WARNMS, WARNMS1, WARNMS2, TRACEMS, TRACEMS1, ... TRACEMS8 and TRACEMSS simply expand to nothing:

```
#define WARNMS(cinfo,code)
#define WARNMS1(cinfo,code,p1)
#define WARNMS2(cinfo,code,p1,p2)
#define TRACEMS(cinfo,lvl,code)
#define TRACEMS1(cinfo,lvl,code,p1)
#define TRACEMS2(cinfo,lvl,code,p1,p2)
#define TRACEMS3(cinfo,lvl,code,p1,p2,p3)
#define TRACEMS4(cinfo,lvl,code,p1,p2,p3,p4)
#define TRACEMS5(cinfo,lvl,code,p1,p2,p3,p4,p5)
#define TRACEMS8(cinfo,lvl,code,p1,p2,p3,p4,p5,p6,p7,p8)
#define TRACEMSS(cinfo,lvl,code,str)
```

In order to get diagnostic output during debugging over the serial line of the M2 board, jerror.c was modified. Again a global function pointer (jprintffunc), this time with the function signature of the well-known function printf was implemented as well as a function for diagnostic output (jprintf). This function could either be called separately or from the error managers output_message function, which was modified to call jprintf:

[2]Besides from reducing the makefile to the decode part only.

```
#include <stdio.h>
#include <stdarg.h>

#ifdef JPRINTF
pprintf jprintffunc= NULL;
char szBuf [1024];

int    jprintf   ( const char * format, ... )
{
        va_list arglist;
        va_start(arglist, format);
        vsprintf(szBuf, format, arglist);
        if (jprintffunc)
            return jprintffunc(szBuf);
        else
            return 0;
}
#endif
METHODDEF(void)
output_message (j_common_ptr cinfo)
{
#ifdef _DEBUG
  char buffer[JMSG_LENGTH_MAX];

  /* Create the message */
  (*cinfo->err->format_message) (cinfo, buffer);

#ifdef USE_WINDOWS_MESSAGEBOX
  /* Display it in a message dialog box */
  MessageBox(GetActiveWindow(), buffer, "JPEG Library Error",
     MB_OK | MB_ICONERROR);
#else
  /* Send it to stderr, adding a newline */
#ifdef JPRINTF // we do a custom printf across the serial line for the M2
  jprintf("%s\n", buffer);
#endif
#endif
#endif
}
```

Of course, for this to work, JPRINTF must be defined in jconfig.h and pprintf typedef'ed;

```
#define JPRINTF     // jprintf function defined or not?

#ifdef JPRINTF //defined in jconfig.h
typedef int (*pprintf)(const char *);
extern pprintf jprintffunc;
int    jprintf   ( const char *, ... );
#endif
```

To make this diagnostic functionality actually work, the function pointer jprintffunc now needs to be assigned a value of a function with a signature that resembles printf's signature. This is done in the file jdecode.c in the function JpegDecode. The actual implementation of this function calls code that transmits the diagnostic output via the serial line of the M2 board to a monitoring program on the development computer via third-party code.

A slight performance boost came from using the M2's IRAM data for tables that are used during Huffman decoding. By qualifying these tables with the keyword _iram the linker/locator puts them into IRAM:

```
int _iram extend_test[16] =   /* entry n is 2**(n-1) */
  { 0, 0x0001, 0x0002, 0x0004, 0x0008, 0x0010, 0x0020, 0x0040, 0x0080,
    0x0100, 0x0200, 0x0400, 0x0800, 0x1000, 0x2000, 0x4000 };

int _iram extend_offset[16] = /* entry n is (-1 << n) + 1 */
  { 0, ((-1)<<1) + 1, ((-1)<<2) + 1, ((-1)<<3) + 1, ((-1)<<4) + 1,
    ((-1)<<5) + 1, ((-1)<<6) + 1, ((-1)<<7) + 1, ((-1)<<8) + 1,
    ((-1)<<9) + 1, ((-1)<<10) + 1, ((-1)<<11) + 1, ((-1)<<12) + 1,
    ((-1)<<13) + 1, ((-1)<<14) + 1, ((-1)<<15) + 1 };
```

The qualifiers "static" and "const" were removed, in order to verify that these tables actually were created in IRAM address space via the map file. For experimental purposes in this file also memory allocation in jpeg_make_d_derived_tbl uses different allocators depending on the value of the macro HUFF_LOOKAHEAD. It turned out that a higher value of HUFF_LOOKAHEAD than its default (8) can accelerate Huffman decoding, but also for some images allocation requests could not be satisfied, so the default value is probably the safest way to go. In order to "hook" into Huffman decoding with own functions (see section), function start_pass_huff_decoder had to be made publicly available, so the METHODDEF macro (which expands to "static") had to be removed for this function.

In order to improve the Arai-Agui-Nakajima IDCT the keyword "const" was removed from a local dequantization table (aanscales) in jddctmgr.c in the function start_pass. If const is applied to a variable, this variable is placed into the ROM and is then first copied into RAM when it is used. By removing const, this step can be omitted, resulting in higher performance.

6.3 Changes for faster downscaling to a fourth

As explained in section 4.2.2, the algorithm to downscale to a fourth can make use of absorbing the constant multiplication factors into the dequantization tables. The actual implementation of this turned out to be dependent on the target platform, and maybe this is why the authors of the JPEGLib did not favor this improvement. Different multiplication tables for the dequantization tables are required for 16-bit and 32-bit platforms and especially for 16-bit platforms such as the M2, this leads to reduced accuracy. In order to turn on the new functionality, the macro USE_FASTER_2x2_IDCT must be defined in jconfig.h:

```
#define USE_FASTER_2x2_IDCT
```

In jddctmgr.c now the two tables are defined with the macros SLOPPY16BITTABLE and ACCURATETABLE:

```
/*
Table for decoding to the fourth on 16-bit platforms
with very unaccurate results:
*/

#define SLOPPY16BITTABLE    32, 29, 16, 10, 16, 7, 16, 6,\
                            29, 26, 14, 9, 14, 6, 14, 5,\
                            16, 14, 8, 5, 8, 3, 8, 3,\
                            10, 9, 5, 3, 5, 2, 5, 2,\
                            16, 14, 8, 5, 8, 3, 8, 3,\
                            7, 6, 3, 2, 3, 1, 3, 1,\
                            16, 14, 8, 5, 8, 3, 8, 3,\
                            6, 5, 3, 2, 3, 1, 3, 1\

/*
Table for decoding to the fourth on all other platforms
with accurate results:
*/

#define ACCURATETABLE 2097152, 1900287, 1048576, 667292, 1048576, 445870, 1048576, 377991,\
                      1900287, 1721902, 950143, 604652, 950143, 404015, 950143, 342508,\
                      1048576, 950143, 524288, 333646, 524288, 222935, 524288, 188995,\
                      667292, 604652, 333646, 212325, 333646, 141871, 333646, 120273,\
                      1048576, 950143, 524288, 333646, 524288, 222935, 524288, 188995,\
                      445870, 404015, 222935, 141871, 222935, 94795, 222935, 80364,\
                      1048576, 950143, 524288, 333646, 524288, 222935, 524288, 188995,\
                      377991, 342508, 188995, 120273, 188995, 80364, 188995, 68129
```

These two tables contain the constant factors for the algorithm of downscaling to a fourth from section 4.2.2 for fixed point arithmetics. SLOPPY16BITTABLE uses 3-bit fixed point arithmetics, which means that the constant factors are scaled with $2^3 = 8$ whereas ACCURATETABLE uses 19-bit fixed point arithmetics, which means that the constant factors are scaled with $2^19 = 524288$. It can easily be shown that these two values lead to the maximum accuracy for 16-bit and 32-bit platforms without leading to an overflow. The floating point values that represent the factors from the algorithm in section 4.2.2 which lead to these two tables are the following:

```
4.000000 ,3.624510 ,2.000000 ,1.272759 ,2.000000 ,0.850430 ,2.000000 ,0.720960 ,
3.624510 ,3.284268 ,1.812255 ,1.153281 ,1.812255 ,0.770598 ,1.812255 ,0.653281 ,
2.000000 ,1.812255 ,1.000000 ,0.636379 ,1.000000 ,0.425215 ,1.000000 ,0.360480 ,
1.272759 ,1.153281 ,0.636379 ,0.404979 ,0.636379 ,0.270598 ,0.636379 ,0.229402 ,
2.000000 ,1.812255 ,1.000000 ,0.636379 ,1.000000 ,0.425215 ,1.000000 ,0.360480 ,
0.850430 ,0.770598 ,0.425215 ,0.270598 ,0.425215 ,0.180808 ,0.425215 ,0.153281 ,
2.000000 ,1.812255 ,1.000000 ,0.636379 ,1.000000 ,0.425215 ,1.000000 ,0.360480 ,
0.720960 ,0.653281 ,0.360480 ,0.229402 ,0.360480 ,0.153281 ,0.360480 ,0.129946
```

In order to do the actual implementation of absorbing these tables into the dequantization values the following code must be added to the function start_pass in jddctmgr.c:

```
#ifdef PROVIDE_ISLOW_TABLES
    case JDCT_ISLOW:
    {
    /* For LL&M IDCT method, multipliers are equal to raw quantization
    * coefficients, but are stored as ints to ensure access efficiency.
    */
    ISLOW_MULT_TYPE * ismtbl = (ISLOW_MULT_TYPE *) compptr->dct_table;

        if (method_ptr != jpeg_idct_2x2)
        {
        for (i = 0; i < DCTSIZE2; i++)
            {
          ismtbl[i] = (ISLOW_MULT_TYPE) qtbl->quantval[i];
        }
        }
        else
        {

#ifdef USE_FASTER_2x2_IDCT
            static const INT32 iTwoByTwoScales[DCTSIZE2] =
            {
#if (INT_MAX==32767)
            SLOPPY16BITTABLE
#else
            ACCURATETABLE
#endif
            };
            for (i=0;i<DCTSIZE2;i++)
                ismtbl[i] = (ISLOW_MULT_TYPE)((INT32) qtbl->quantval[i] *
                                              (INT32) iTwoByTwoScales[i]);
#else   //USE_FASTER_2x2_IDCT
        for (i = 0; i < DCTSIZE2; i++)
            {
          ismtbl[i] = (ISLOW_MULT_TYPE) qtbl->quantval[i];
        }
#endif
        }
    }
      break;

#endif
```

This piece of code checks whether the downscaling should be downscaling to a fourth (if (method_ptr != jpeg_idct_2x2)...). If so, the dequantization tables are multiplied with either SLOPPY16BITTABLE or ACCURATETABLE. The detection of the platform is done with checking the value of the standard C macro INT_MAX. If INT_MAX evaluates to 32767, it is a 16-bit platform. For this to work, the standard header limits.h must be included at the beginning of the file:

```
#include <limits.h>  /*for INT_MAX*/
```

This header file must be included in the same way in jidctred.c because in this file the alternative variant of the function jpeg_idct_2x2 is implemented. Depending on the value of the macro USE_FASTER_2x2_IDCT the preprocessor uses the appropriate version in jidctred.c like in the following:

```
#ifndef USE_FASTER_2x2_IDCT
GLOBAL(void)
jpeg_idct_2x2 (j_decompress_ptr cinfo, jpeg_component_info * compptr,
       JCOEFPTR coef_block,
       JSAMPARRAY output_buf, JDIMENSION output_col)
{
    /* standard implementation that comes with the JPEGLib */
    .....
}
#else
GLOBAL(void)
jpeg_idct_2x2 (j_decompress_ptr cinfo, jpeg_component_info * compptr,
       JCOEFPTR coef_block,
       JSAMPARRAY output_buf, JDIMENSION output_col)
{
    /* new, faster implementation with constants absorbed in dequantization table */
    .....
}
#endif
```

The complete implementation of the faster variant of jpeg_idct_2x2 is the following.

```
GLOBAL(void)
jpeg_idct_2x2 (j_decompress_ptr cinfo, jpeg_component_info * compptr,
       JCOEFPTR coef_block,
       JSAMPARRAY output_buf, JDIMENSION output_col)
{
  int tmp0, tmp10, z1;
  JCOEFPTR inptr;
  ISLOW_MULT_TYPE * quantptr;
  int * wsptr;
  JSAMPROW outptr;
  JSAMPLE *range_limit = IDCT_range_limit(cinfo);
  int ctr;
  int workspace[DCTSIZE*2]; /* buffers data between passes */
  SHIFT_TEMPS

  /* Pass 1: process columns from input, store into work array. */

  inptr = coef_block;
  quantptr = (ISLOW_MULT_TYPE *) compptr->dct_table;
  wsptr = workspace;
  for (ctr = DCTSIZE; ctr > 0; inptr++, quantptr++, wsptr++, ctr--) {
    /* Don't bother to process columns 2,4,6 */
    if (ctr == DCTSIZE-2 || ctr == DCTSIZE-4 || ctr == DCTSIZE-6)
      continue;

    if (inptr[DCTSIZE*1] == 0 && inptr[DCTSIZE*3] == 0 &&
inptr[DCTSIZE*5] == 0 && inptr[DCTSIZE*7] == 0) {
      /* AC terms all zero; we need not examine terms 2,4,6 for 2x2 output */
      int dcval = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
      wsptr[DCTSIZE*0] = dcval;
      wsptr[DCTSIZE*1] = dcval;
      continue;
    }
```

```
    /* Even part */
    z1 = DEQUANTIZE(inptr[DCTSIZE*0], quantptr[DCTSIZE*0]);
    tmp10 = z1;

    /* Odd part */

    z1 = DEQUANTIZE(inptr[DCTSIZE*1], quantptr[DCTSIZE*1]);
    tmp0 = z1;

    z1 = DEQUANTIZE(inptr[DCTSIZE*5], quantptr[DCTSIZE*5]);
    tmp0 += z1;

    z1 = DEQUANTIZE(inptr[DCTSIZE*3], quantptr[DCTSIZE*3]);
    tmp0 -= z1;

    z1 = DEQUANTIZE(inptr[DCTSIZE*7], quantptr[DCTSIZE*7]);
    tmp0 -= z1;

    /* Final output stage */
    wsptr[DCTSIZE*0] = (int) ((tmp10 + tmp0));
    wsptr[DCTSIZE*1] = (int) ((tmp10 - tmp0));

  }

  /* Pass 2: process 2 rows from work array, store into output array. */

  wsptr = workspace;
  for (ctr = 0; ctr < 2; ctr++) {
    outptr = output_buf[ctr] + output_col;
    /* It's not clear whether a zero row test is worthwhile here ... */

#ifndef NO_ZERO_ROW_TEST
    if (wsptr[1] == 0 && wsptr[3] == 0 && wsptr[5] == 0 && wsptr[7] == 0) {
      /* AC terms all zero */
#if (INT_MAX==32767)
      JSAMPLE dcval = range_limit[(int) DESCALE((INT32) wsptr[0],
                      CONST_BITS+PASS1_BITS+3-10) & RANGE_MASK];
#else
      JSAMPLE dcval = range_limit[(int) DESCALE((INT32) wsptr[0],
                      CONST_BITS+PASS1_BITS+3+6) & RANGE_MASK];
#endif
      outptr[0] = dcval;
      outptr[1] = dcval;
      wsptr += DCTSIZE; /* advance pointer to next row */
      continue;
    }
#endif

    /* Even part */
    tmp10 = (INT32) (wsptr[0]);

    /* Odd part */
    z1 = wsptr[1];
    tmp0 = z1;

    z1 = wsptr[5];
    tmp0 += z1;
```

```
    z1 = wsptr[3];
    tmp0 -= z1;

    z1 = wsptr[7];
    tmp0 -= z1;

    /* Final output stage */
#if (INT_MAX==32767)
    outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp0,
  CONST_BITS+PASS1_BITS+3-10) & RANGE_MASK];
    outptr[1] = range_limit[(int) DESCALE(tmp10 - tmp0,
  CONST_BITS+PASS1_BITS+3-10) & RANGE_MASK];
#else
    outptr[0] = range_limit[(int) DESCALE(tmp10 + tmp0,
  CONST_BITS+PASS1_BITS+3+6) & RANGE_MASK];
    outptr[1] = range_limit[(int) DESCALE(tmp10 - tmp0,
  CONST_BITS+PASS1_BITS+3+6) & RANGE_MASK];
#endif


    wsptr += DCTSIZE; /* advance pointer to next row */
  }
}
```

Note that the automatic variables tmp0, tmp10 and z1 are now int variables, not the typedef'ed data type INT32 as before, which also leads to faster code, at least on 16-bit platforms. Despite the 3-bit fixed point arithmetics for 16-bit platforms, a surprising visual fidelity can be achieved. Table 6.1 shows the deviations of the new implementation of jpeg_idct_2x2 for decoding the test file testimg.jpg (227×149 pixels resolution) that comes with the JPEGLib.

	3-bit table	**19-bit table**
Peak Error:	1	1
Mean Square Error:	0.026316	0.009541
Mean Error:	-1	8
Different Pixel Components:	171	62

Table 6.1: Deviations of the new implementation from the standard implementation

Table 6.1 shows that no single component per pixel has a deviation of more than 1 (peak error), but that the 19-bit table (based on the macro ACCURATETABLE) is much closer to the reference made with the JPEGLib's standard implementation than the 3-bit table (based on the macro SLOPPY16BITTABLE), which can be seen from the values for the mean square error[3] and the number of different pixel components[4]. The mean error[5] shows that many errors with the 3-bit table cancel themselves out.

[3]This is the mean per-pixel deviation.

[4]These is the sum of pixel components (R,G and B) that differ from the reference image.

[5]This is the signed sum of deviations from the reference image over the whole image.

In order to measure the performance of this modified algorithm on M2, a JPEG image of file size $\sim$ 160 kByte with a spatial resolution of 2048×1536 pixels resolution and 4:4:4 chroma subsampling was decoded. Decoding time for this image was reduced from 53.10 s to 50.94 s.

6.4 Important compiler optimization settings

In order to get the maximum performance out of the JPEGLib, quite some time was devoted to find the combination of compiler optimization settings that delivers best performance. The settings used were -Oa ("relax alias checking": Registers are not cleared after a write to an indirect address), -Of ("produce fast code": Execution speed is favoured above code density [6]), -Ol ("enable fast loops": Duplicates a loop condition at the end of a loop to save the unconditional jump to the start of the loop), -Os ("generate jump tables for switch statements": Generates a jump table for switch statements instead of a slower jump chain), -Ot ("turn tentative declarations into defining occurrences": declarations of global variables that do not define the variables value are turned into defining occurrences, allowing more data to be optimized).

Furthermore, one compiler switch that is not among the compiler optimizations turned out to improve performance drastically: With the switch -S all automatic variables that cannot be allocated in a register are not allocated on the stack but are treated instead as if they were static variables. This way very expensive extra register move operations can be saved but the functions that use this approach cannot be used recursively.

6.5 The custom data source manager for M2

In order to read JPEG data from memory instead of a file, a custom data source manager object has to be supplied to the decompression object. Typically this happens by allocating a jpeg_source_mgr struct on the stack, along with the decompression object. The address of the data source manager object has to be supplied to the decompression object by assigning it to the src member of the decompression object. Finally, the data source manager object needs to implement four functions, which are only implemented as function pointers of the data source manager object and need to be assigned values of real functions. The implemented functions are init_source, fill_input_buffer, skip_input_data and term_source which are all assigned to identical named members of the data source manager object. For the resync_to_restart member of the data source manager object, the default implementation that comes with the JPEGLib was used. The purpose of these four functions is as follows:

- init_source: This function initializes the source manager and is called by jpeg_read_header before any data is actually read. The members of the data source manager object that specify the memory location where to start to read, and how many bytes from this address on are valid, are to be specified in the implementation of this function.

- fill_input_buffer: This function is called whenever the JPEGLib needs new data, because all data that previously was specified to be valid, either in init_source or the

[6] Code size is not a problem since the M2 does not use an OS that pages and where excessive page misses could occur because of code size.

last call to fill_input_buffer, has already been consumed. Again, the members of the data source manager object that specify the memory location from where to read from now on, and how many bytes from this address on are valid, are to be specified in the implementation of this function.

- skip_input_data: This function is called whenever uninteresting data, such as APPn markers (see section 2.3) need to be skipped. The number of "uninteresting bytes" is passed to the function, and the implementation must calculate the new address of "interesting data" and must calculate the number of valid bytes that are left from this address on.

- term_source: Called by jpeg_finish_decompress. This function allows cleanup of per-source-manager data, e.g. if memory is allocated dynamically in init_source, it can be released here. Often, as in the case of the project for this thesis, this function consists of an empty function body.

The actual implementation of these functions is very similar to the implementation of the function that zeros memory (jzero_far in jutils.c) in section JPEGLibChanges. In init_source simply the start of the array that contains the image is specified as the start of data to read. The number of bytes up to the next page is specified as the initial number of bytes to read. Whenever fill_input_buffer is now called, a page boundary has been reached and the new address to read from is the start of the page and the number of bytes to read is either the complete page or the rest of the input file if this is less than a page. The function skip_input_data simply skips over the "uninteresting" data by using a temporary huge pointer that is incremented by the number of bytes to skip, doing huge pointer arithmetics. The resulting address of the temporary pointer is then the address where "interesting" data begins again. The complete implementation of the data source manager can be found in the file jdecode.c in the directory d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src.

6.6 Other implementations of custom functionality

This section contains the descriptions of various custom "managers" or functionality for faster or more accurate performance of JPEG decoding on the M2. The majority of these implementations consists out of copied versions of functions and data types that originate from the JPEGLib and that were adapted to the needs of this project. Quite some custom implementations could only be made with overuse of the "extern" keyword, which is normally considered to be a sign of bad coding and design practices. The same is true for the redefinition of data types that are already used and defined in the JPEGLib.

In this project, "extern" and the reliance on custom but actually redefined data types with identical binary layout to those in the JPEGLib had to be used because a lot of functionality in the JPEGLib is not meant to be replaced by custom code, though the design as a "subobject" or "manager" suggests this. Consequently, the designers of the JPEGLib deliberately did not prototype these functions and data types in the associated header files.

Doing these customizations in a rather "expedient" fashion served the goal not to make these changes directly to the JPEGLib in order to be able to see afterwards which real changes as documented in section 6.2 are necessary for M2 using the JPEGLib. Furthermore

this way the build cycles were much easier, because only one project needed to be built after a change and the JPEGLib could remain relatively untouched throughout the whole project

A more elegant way would have been just to prototype these functions and data types as was done in some rare cases, or even implement the changes inside the library, but the classification of this project as a feasibility study probably justifies sacrificing some aspects of good coding and design practices. Also, such an approach would have been useful if a new version of the JPEGLib would have been released during this project. In this case, only the few real necessary changes as described in section 6.2 would have to be applied and the redefined data types and functions would have to be checked for compatibility with the new version. Reimplementing instead all custom functionality in a new version of the JPEGLib could easily have become a nightmare.

In the following we will sometimes look at these customizations from a slightly more abstract point of view than from the actual code level, because a lot of code consists out of copied code snippets from the JPEGLib. We will rather point out which functionality was "borrowed" from the JPEGLib and what the essential modifications to the original code were. It probably makes no sense to read the following subsections without access to the source code of the JPEGLib and the custom code for the M2.

All source files mentioned in the following subsections can be found in directory src and all header files in directory include under the directory
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg
except for those from the JPEGLib which can be found in the directory
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\Jpeglib.

6.6.1 "Hooking" into Huffman decoding

In order to put the Huffman tables into the IRAM, the struct my_input_controller as defined in the JPEGLib's file jdinput.c was redefined in jdecode.c. The decompresssion object has a pointer to this struct as a member and this objects member of type jpeg_input_controller (pub) has as a member to a function pointer (start_input_pass) that normally points to the function start_input_pass in the JPEGLib's file jdinput.c but is replaced in function JpegDecode in file jdecode.c with my_start_input_pass from the file huffhook.c. The function start_input_pass calls the decompression object's entropy decoder subobject's function pointer named start_pass which normally points to the function start_pass_huff_decoder in the JPEGLib's file jdhuff.c. In order for start_pass_huff_decoder to use tables that are in IRAM, the standard tables first have to be copied over to the tables in IRAM and the pointers to these tables in the decompression object must modified to point to the tables in IRAM. Both these steps happen in my_start_pass_huff_decoder in file huffhook.c which after performing these steps calls start_pass_huff_decoder. In order for my_start_pass_huff_decoder to be executed, function my_start_input_pass replaces the function pointer start_pass of the decompression object's entropy decoder subobject and finally calls the JPEGLib's start_input_pass function. This way, my_start_input_pass and my_start_pass_huff_decoder work as wrappers for start_input_pass and start_pass_huff_decoder, respectively, that set up the Huffman tables in IRAM and direct the JPEGLib to use these instead of the default ones that have been created before on the heap and have been filled with the correct values from the JPEG input stream.

6.6.2 Using XRAM for the range-limit table

In order to limit the output of certain operations to the range [0...255], a table is used in the JPEGLib, and the decompression object has a pointer (sample_range_limit) that points at index MAXJSAMPLE+1 of this table. In order to use the start of the XRAM address range[7] for this array, a global pointer (unsigned char *g_lpXRAM) to the start of the XRAM is used and set up like the following in function JpegDecode in file jdecode.c:

```
#if RANGELIMIT_IN_XRAM
    /* now copy over the range limit table into the XRAM: */
    if (!g_lpXRAM)
    {
        /* create our range-limit table in XRAM only one time,
           it will be reused for each image */
        g_lpXRAM = (unsigned char *)0xE000;
        memcpy(g_lpXRAM, &cinfo.sample_range_limit[-MAXJSAMPLE-1], RANGE_LIMIT_TABLE_SIZE);
    }
    /* now do this in any case, because after each picture the same range-limit
       table is newly created since it came from the heap: */
    cinfo.sample_range_limit = &g_lpXRAM[MAXJSAMPLE+1];
#endif
```

This way for each newly decoded image, the sample_range_limit is properly set to point at index MAXJSAMPLE+1 in the XRAM, where an exact copy of the range-limiting table lies.

6.6.3 Modifying the color conversion and upsampling subobjects

The JPEGLib's standard approach for decoding one or more lines requires prior allocation of a suitable buffer that after a call to jpeg_read_scanlines contains RGB tupels as 24 bits per pixel and 8 bits per color component. This means for an architecture like the M2 with the 4-4-4 or the 5-6-5 mode, that each line in the destination buffer must be filled by looping over the buffer with 24 bits per pixel, extracting the RGB components, quantizing the components by a right shift and storing this values in the appropriate format in the destination buffer. An alternative for this slow approach is the implementation of custom functionality that does the quantization to the 4-4-4 or 5-6-5 mode already in the color conversion and upsampling stages[8], which is described in the following.

In order to adapt the JPEGLib to store RGB tupels not as 24-bit tupels but instead as 4-4-4 or 5-6-5 tupels in one word, the decompression object's color conversion subobject (for 4:4:4 chroma subsampling) and the upsampling subobject need to be modified to have their function pointers point to custom functions that store the RGB tupels in the desired format.

For this to be accomplished, first the struct my_decomp_master as defined in the JPEGLib's file jdmaster.c needs to be redefined in jdecode.c. After this step, the binary layout of the struct my_decomp_master is known to the application and in function JpegDecode in file jdecode.c the master subobject of the decompression object can safely be accessed.

[7] XRAM starts at address 0xE000 and like IRAM is 2 kBytes in size.

[8] Furthermore, by feeding jpeg_read_scanlines with pointers to the start of lines in the frame buffer, no copy operation is required after jpeg_read_scanlines, because the color conversion and upsampling subobjects work directly on the frame buffer.

Via its member using_merged_upsample it can now be determined, whether the currently decoded file uses 4:4:4 chroma subsampling (i.e. no subsampling) or whether it uses some sort of subsampling. If no subsampling is used, depending on the output color space (color or grayscale), the decompression object's color conversion object's file pointer color_convert is replaced by an own variant (my_ycc_rgb_convert or my_grayscale_convert in files yccrgb.c and grayconv.c, respectively).

For images with chroma subsampling, first the struct my_upsampler from the JPEGLib's file jdmerge.c is redefined in file cstupsmp.h. After that, the binary layout of this upsampling subobject of the decompression object is known to the application and its upmethod function pointer can be assigned a new value, depending on the type of upsampling that is required. For 4:2:2 chroma subsampling, the address of function my_h2v2_merged_upsample in file h2v2mups.c is assigned and for 4:2:0 chroma subsampling the address of function my_h2v1_merged_upsample in file h2v1mups.c is assigned.

The functions my_ycc_rgb_convert, my_grayscale_convert, my_h2v2_merged_upsample and my_h2v1_merged_upsample are nothing else than copies of the JPEGLib's functions ycc_rgb_convert, grayscale_convert[9], h2v2_merged_upsample and h2v1_merged_upsample[10] that were modified to store RGB tupels in the 4-4-4 mode or the 5-6-5 mode, depending on the value of the macro USE_MIN_GDI (see section 6.7), instead of 8-bits per color component, as the JPEGLib's version does.

6.6.4 Modifying the dithering subobjects

For the same reason as in section 6.6.3, the dithering subobjects need to be modified to use custom functionality in order for the JPEGLib to work directly on the frame buffer.

In order to turn on dithering, after calling the JPEGLib's function jpeg_read_header, the quantize_colors member of the decompression object must be assigned the value TRUE and the dither_mode member must be assigned one of the values JDITHER_FS (for Floyd-Steinberg dithering) or JDITHER_ORDERED[11] (for ordered dithering) as is done in the function JpegDecode in file jdecode.c. In order to be able to access the quantizer subobject (cquantize) of the decompression object, the struct my_cquantizer as defined in the JPEGLib's file jquant1.c must be redefined, which is done in the file custdith.h. After this step, the binary layout of this data type is known and the function pointers of this subobject can be safely accessed and assigned new values.

If Floyd-Steinberg dithering is chosen, the color_quantize member of the subobject is assigned the address of the function quantize_fs_dither from file dithfsd.c.

If odered dithering is chosen, the color_quantize member of the subobject is either assigned the address of the function quantize3_ord_dither (for color output) or quantize_ord_dither (for grayscale output). Both these functions reside in file dithord.c.

The functions quantize_fs_dither, quantize3_ord_dither and quantize_ord_dither are all modified copies of functions with the same name that reside in the JPEGLib's file jquant1.c. The modifications that were made include not only functionality for storing RGB tupels in the 4-4-4 mode or the 5-6-5 mode. Additionally, for Floyd-Steinberg dithering, some quality

[9] The functions ycc_rgb_convert and grayscale_convert can be found in the JPEGLib's file jdcolor.c.

[10] The functions h2v2_merged_upsample and h2v1_merged_upsample can be found in the JPEGLib's file jdmerge.c.

[11] We simply disregard here the value JDITHER_NONE which cannot be made effective in this project's source code.

improvements were made over the standard implementation. The standard implementation in the JPEGLib uses only a maximum of 256 equally distributed colors for the display of dithered images. Also, for the error diffusion involved in Floyd-Steinberg dithering (see [10] and [16]), only these 256 colors which reside in a table are used. The custom implementation for the M2 however takes advantage of the 4-4-4 mode or 5-6-5 mode of the M2 in that for error diffusion the nearest colors that the M2 can display in the respective mode is used for error diffusion and display. This results in much better quality of the image and also higher performance because calculating a pixel value and the error it introduces involves only shift operations whereas the standard implementation's approach uses table-lookups which are much slower than shift operations on the M2.

6.7 Customizing the behaviour of the Software

The software that was written for the M2 can be customized at three central points with the help of macros. For both projects, the file jdecode.c in the directory
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src
contains the definitions of the macros ENABLE_FLOYD_STEINBERG_DITHER and ENABLE_ORDERED_DITHER. By assigning these macros different values (1 or 0), usage of the JPEGLib can be customized.

ENABLE_FLOYD_STEINBERG_DITHER and ENABLE_ORDERED_DITHER turn on and off Floyd-Steinberg dithering and ordered dithering, respectively. Deliberate preprocessor errors with the help of the #error directive assure that these two macros are used in a mutually exclusive way. Similarly, if dithering is used, it is also assured that the JPEGLib's macro QUANT_1PASS_SUPPORTED is turned on:

```
#if (ENABLE_FLOYD_STEINBERG_DITHER && ENABLE_ORDERED_DITHER)
#error FS-Dither and ordered dither are mutual exclusive   //deliberate bail-out
#endif

#if (ENABLE_FLOYD_STEINBERG_DITHER || ENABLE_ORDERED_DITHER)
#ifndef QUANT_1PASS_SUPPORTED
#error You have to define QUANT_1PASS_SUPPORTED in jmorecfg.h for dithering, note that
#error this affects performance negatively for undithered images   //deliberate bail-out
#endif
#endif
```

The second location for customization can be found in the file startup.c in the directory d:\work\mini\target\warmup\Source and is effective only for the second project. The macros that can be used for customization are USE_SERIAL_UPLOAD, USE_PROGRESS_MONITOR, DEMO_SHOW, TV_SET, DO_CODE_REDIRECTION, DO_DELAY, DISSOLVE, DELAYDURATION, SCALE and STRIPEHEIGHT.

Setting the value of USE_SERIAL_UPLOAD to 1 allows decoding of even very large files that need to be transmitted onto the M2 development board via the serial line. Further information on this can be found in [18].

Setting USE_PROGRESS_MONITOR to 1 allows for visual feedback of the decoding and scaling process. If the MINI GDI library is used for 5-6-5 mode of the M2, every single decoded line of the image can immediately be seen. When using the standard GDI library (4-4-4 mode), stripes of the image of height STRIPEHEIGHT are output to the screen as soon as these lines are decoded. Setting USE_PROGRESS_MONITOR to 0 has a slight

advantage in performance, since the processor does not need to care about simultaneous read and write accesses to the frame buffer in the case of 5-6-5 mode and the additional overhead of creating the bitmaps for the image stripes in the case of 4-4-4 mode is not necessary.

If the macro DEMO_SHOW evaluates to 1, two different alternating images are decoded that come from the files jsample.c and jsample800x600.c in directory
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src.

If DO_DELAY evaluates to 1, a delay between the alternating decoding processes is introduced whose duration is the value of macro DELAYDURATION in ms.

If DISSOLVE evaluates to 1, a fancy dissolving effect ([19]) after decoding one image and after the delay is applied to the screen.

If DO_CODE_REDIRECTION evaluates to 1, the executable code is copied from ROM into the DRAM upon startup. It is generally not advisable to set this value to 0 for other purposes than to measure the impact of this to the performance.

If TV_SET is assigned the value 1, the M2's timing parameters are changed to those that are suitable for a PAL TV set, otherwise 800×600 pixels resolution with SVGA timing for a standard computer monitor is assumed.

Finally with setting SCALE to 1, the decoded image is scaled to 800×600 pixels with preserved aspect ratio as described in section 4.1.

The third location for customization are the second project's preprocessor settings which have to be set in the Integrated Development Environment. These settings include macros that are of global scope for the whole project. The macros that are specified here are USE_MIN_GDI, RANGELIMIT_IN_XRAM and ENABLE_HUFFMAN_HOOKING.

If USE_MIN_GDI evaluates to 1, the project uses the MINI GDI and images are rendered on the screen in 5-6-5 mode. If USE_MIN_GDI evaluates to 0, the standard GDI library is used and images are rendered on the screen in 4-4-4 mode.

If ENABLE_HUFFMAN_HOOKING evaluates to 1, the tables for the Huffman decoding part are replaced with buffers in M2's IRAM with the help of the functions my_start_pass_huff_decoder and my_start_input_pass from file huffhook.c in directory
d:\work\ae7\V601_EvalBoard\FunctionLibrary\Functions\jpeg\Src.
These two functions replace the JPEGLib's start_pass_huff_decoder and start_input_pass functions.

If RANGELIMIT_IN_XRAM evaluates to 1, an often used table in the JPEGLib (the sample_range_limit member of the decompression object) is replaced by an identical table that is allocated in M2's XRAM.

6.8 Results

The next few subsections will deal with the actual results of this thesis and how the potential obstacles, as identified prior to the start of this thesis (see section 1.1), could be overcome.

6.8.1 Memory limitations and high-resolution JPEG files

One of the potential obstacles that were identified beforehand (see section 1.1) for real usability of the JPEG decoder on the M2 was limited memory. The controller can be used with a maximum of 8 MBytes of DRAM, and since it is a classical "Von-Neumann-Architecture", all the code and data, including the frame buffer that holds the decoded

image must fit into the same address space. Furthermore, in order to improve performance, one of the first steps in the program code is its own replication from ROM to RAM[12], so effectively the complete code exists twice in the address space of the controller. Therefore the initial expectations were that only images up to a certain resolution can be decoded successfully and that typical images from digital still cameras are simply too large to fit into memory, albeit the fact that the M2 can only display images of a maximum resolution of 800×600 pixels anyway. Therefore the need to downscale to this resolution is necessary, but is impossible, due to memory limitations, if first the complete image needs to be decoded and afterwards downscaled in the spatial domain to 800×600 pixels. Fortunately, the JPEGLib offers the algorithms for downscaling to half, the fourth and to the eighth of the original image's size as decribed in sections 4.2 and 5.2. These options were used to scale images down to a resolution of 800×600 pixels or less, if necessary, and made the potential problem of memory requirements a non-issue. This approach should work for images up to a resolution of 6400×4800 pixels[13], which is 8 times the size of the M2's frame buffer. For higher resolutions, additional measures will have to be applied to make an image fit into the 800×600 pixels frame buffer of the M2. Fortunately, such big images are not yet in general use at the time of writing, at least not for digital still cameras.

However, it turned out that decoding progressive mode JPEG files is not possible due to their massive memory requirements already during the decoding process. But since progressive mode JPEG files are not as much in use as sequential mode JPEG files and since the Exif standard for digital still cameras did not adopt progressive mode JPEG files, this mode can safely be disregarded.

6.8.2 4-4-4 mode versus 5-6-5 mode

JPEG Performance One of the standard modes of the M2 for displaying images is the 4-4-4 mode in which one word (two bytes) holds one RGB tuple with 4 bits per color component[14]. The image quality of this mode turned out to be completely insufficient. When dithered with ordered dithering, image quality was very much improved but showed the typical patterns of ordered dithering. When applying Floyd-Steinberg Dithering, image quality seemed to be quite acceptable, but Floyd-Steinberg dithering comes at unacceptable additional costs of computation. Fortunately, when halfway through this thesis, the vendor of the standard GDI library[15], provided an additional experimental library (the "MINI GDI" library) that turns the M2 into 5-6-5 mode. In this mode one word holds RGB tupels in 5 bits for the red and blue component and 6 bits for the green component. The quality of images in 5-6-5 mode turned out to be completely sufficient and the library seemed to be useful and stable. For more information on this library see [17].

6.8.3 JPEG Performance

In order to measure the performance of JPEG decoding on the M2, the decoding software was used in the mode that accepts a file which can be transmitted via the serial line into the controller's memory. Now from one and the same image, JPEG files were created with

[12]This step is called "code redirection". The reason for code redirection is that RAM can be accessed much faster than ROM.

[13]The JPEG standard allows a maximum resolution of 65535×65535 pixels.

[14]The remaining bits are used for transparency and blinking modes.

[15]The standard GDI library is only capable of the 4-4-4 mode.

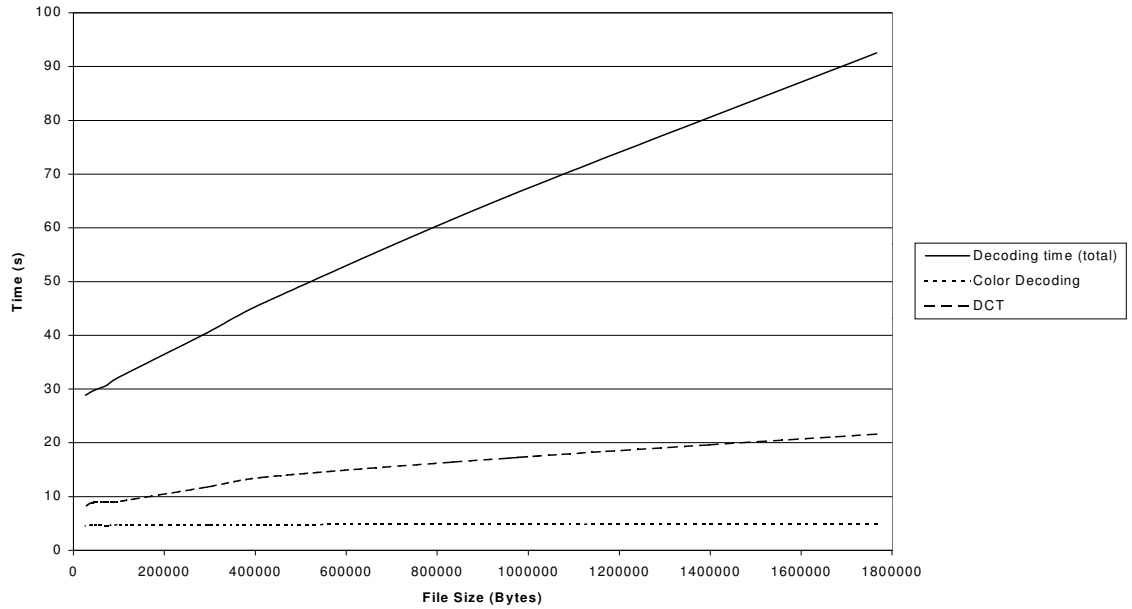

Figure 6.1: Decoding times for a series of images with 2048 × 1536 pixels resolution (4:2:0 mode, downsampled to a fourth)

different quality settings but identical chroma subsampling. These files were transmitted via the serial line onto the M2 board and the total time was measured without a subsequent scaling in the spatial domain. For the same set of files additionally the time for the IDCT and the color decoding step were measured. Figures 6.1 and 6.2 show the result for an image with a resolution of 2048 × 1536 pixels, first with 4:2:0 mode then with 4:4:4 mode, both downsampled to a fourth via the algorithm from section 4.2.2. These graphs show the performance as decoding time versus file size (quality). Figure 6.3 shows the performance for an image with 800 × 600 pixels resolution and 4-2-2 chroma subsampling in different qualities. The time that is not spend in the IDCT and the color decoding parts can be roughly considered the time spent in Huffman decoding. What can be found surprising in figures 6.1, 6.2 and 6.3 is the fact, that the bigger the files size and thus the higher the quality of the picture gets, the time spent in Huffman decoding more and more outweighs the time spent in IDCT and color decoding. At first glance it seems that the complete decoding process has a computational complexity of $O(N)$ and that both Huffman decoding and the IDCT have a computational complexity of $O(N)$ whereas color decoding has a computational complexity of $O(1)$. However, the JPEGLib's IDCT parts contain checks whether e.g. a complete row of DCT coefficients, besides from the DC coefficient, have all zeroes as their values. In this case all coefficients get the DC coefficient's value and the necessity to calculate the coefficients is obsolete. The lower the quality of the picture, the more complete rows can be calculated with this adaptive IDCT. The higher the quality of the image, the fewer such optimizations can be performed and the check for zeroes negatively

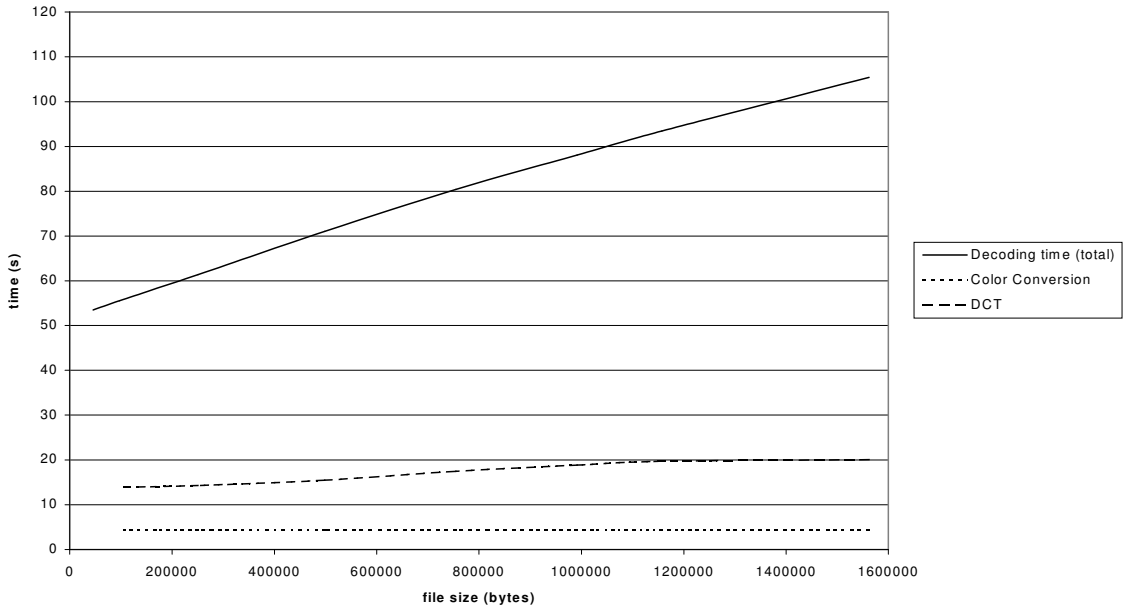

Figure 6.2: Decoding times for a series of images with 2048×1536 pixels resolution (4:4:4 mode, downsampled to a fourth)

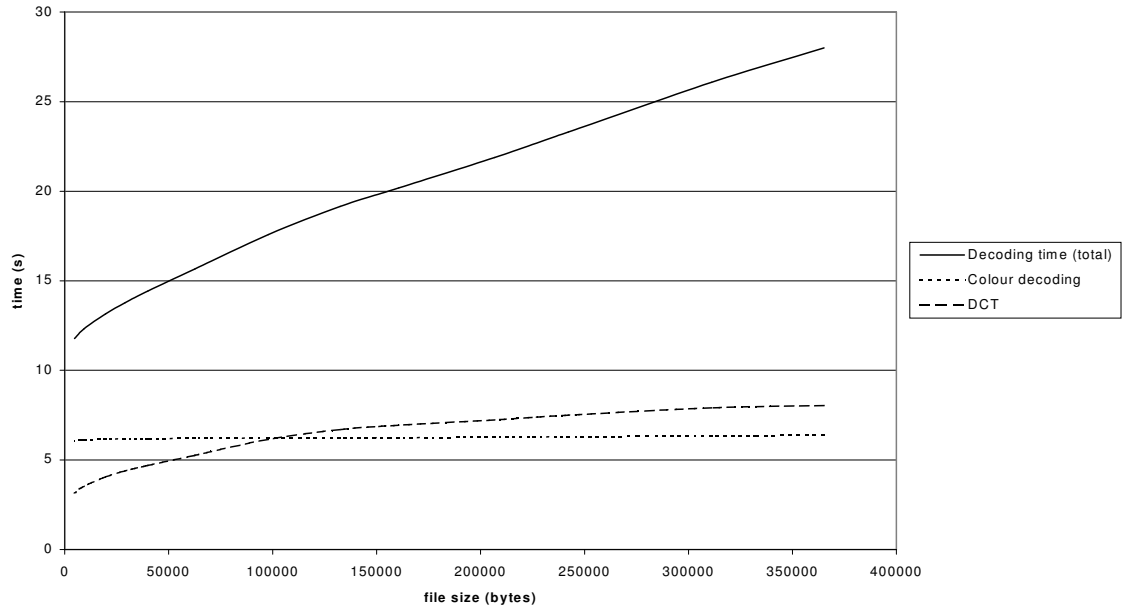

Figure 6.3: Decoding times for a series of images with 800×600 pixels resolution (4:2:2 mode, no downsampling)

affects performance. Since the number of operations required to calculate the IDCT for a certain number of points has an upper bound, if chroma subsampling and final image resolution is constant, we can therefore safely assume a computational complexity of $O(1)$ for the IDCT as well as for the color decoding part.

Steps to improve the performance of the Huffman decoding part included the usage of fast on-chip memory areas (IRAM and XRAM), that were used for the tables that are used during Huffman decoding (see also section 6.2). A slight performance improvement with this could be measured, for instance the complete decoding time for an image with 2048×1536 pixels resolution and 4-2-0 chroma subsampling and a file size of about 1.35 MByte decreased from roughly 70 s to 66 s.

6.8.4 Image scaling Performance

Image scaling performance was measured using the algorithm from 4.1 that simply maps source pixels to destination pixels. Generally, the size of the JPEG source image was used to determine whether a downscaling algorithm from section 4.2 should be used. If the JPEG image to be decoded had a spatial resoulution that exceeded 800 pixels in the horizontal direction or 600 pixels in the vertical direction a proper downscaling algorithm was chosen so that these criteria were met. After the decoding step, images were now 800×600 pixels in size or smaller and could now be upsampled to the maximum resolution of 800×600 pixels while preserving their aspect ratio. The time required for this upsampling step to 800×600 pixels was exactly 11.3 s, independent from the source image size.

6.8.5 Debugging versus "Free-Run" Performance

The decoding times above were all measured with the remote debugger running while decoding. With the debugger running, exact numbers can be measured by a variable in the code that is incremented by a microcontroller timer. This way, even slight improvements in performance can be accurately measured. In a "free run", the executable binary that is executed on the microcontroller gets loaded from an EPROM, and measuring performance is subject to human reaction times, since performance can only be measured using a stopwatch. But performance in this case is actually much faster and also reflects the real usage conditions of the microcontroller.

Table 6.2 contains shows the decoding performance on the M2 for the Debugging Run and the "Free Run" for different configurations of the software. These figures are valid for the aforementioned file (2048×1536 pixels resolution, 4-2-0 chroma subsampling), which has the typical file size ($\sim$ 1.35 MBytes) and quality of images taken with a digital still camera. For a successful commercial product it would be desirable to decode such an image in less than $5 \sim 10$ s, so the processing power needed to achieve this should roughly be tenfold and is definitely not in the realm of the capabilities of the M2.

	Debugging Run (s)	**"Free Run" (s)**
No dithering, no scaling	66	49
Scaling to 800 × 600 pixels, no dithering	77	57
Floyd-Steinberg dithering, no scaling	76	59
Ordered dithering, no scaling	68	53

Table 6.2: Debugging versus "Free Run" Performance

Chapter 7

Summary and Future Outlook

Nothing's impossible I have found,
for when my chin is on the ground.
I pick myself up, dust myself off, start all over again.

Don't lose your confidence if you slip,
be grateful for a pleasant trip.
And pick yourself up, dust yourself off and start all over again

Work like a soul inspired till the battle of the day is won,
you may be sick and tired but you'll be a man my son.
Don't you remember the famous men who had to fall to rise again?
They picked themselves up, dust themselves off and started all over again.

– From the song "Pick Yourself Up" by Jerome Kern and Dorothy Field (from the movie "Swing Time", 1936)

MARKET research has shown that the range of available software packages for JPEG decoding is very limited. The most mature package seemed to be the Independent JPEG Group's JPEGLib, which allows both JPEG encoding and decoding. In addition, this software library, written entirely in C, can be used under very modest licensing requirements, so this software was further examined. It turned out, that the complete library was written with high portability in mind for all imaginable computer architectures, and it took only about two weeks to port the decoding part of the library to the SDA-6000 microcontroller.

The standard functionality of the decoding part of the JPEGLib is to decode the JPEG file into image buffers consisting of RGB tupels with a color depth of 24 bit (or 8 bits per color component). Because the SDA-6000 microcontroller's standard software package is only capable of 4 bits per color component, parts of the library (the color conversion and upsampling backends) were replaced by custom versions with less color depth, that, in order to avoid the need of unnecessary block copy operations, were modified to operate directly on the frame buffer of the graphics accelerator unit of the SDA-6000. Similar modifications

were done for the dithering backends of the JPEGLib in order to determine the performance penalty and image quality improvement when using these backends instead of the modified standard color conversion and upsampling backends.

The display limitations could be overcome with an experimental software library from the standard software library vendor for the graphics accelerator unit of the microcontroller. This experimental library switched the controller into the 5-6-5 mode and turned out to be completely sufficient for the purpose of rendering JPEG decoded images. Also, the image quality seemed to be quite acceptable.

As to memory limitations, it turned out, that for decoding progressive mode JPEG images, memory requirements are too high, at least for 3.3 Megapixel images. But this also is not that much of a problem, since the Exif format specification for digital still cameras specifies only the baseline process to be of any relevance.

The JPEG files to be decoded were first compiled into the executable binary as huge arrays in C source files, but due to compiler limitations, the maximum size of JPEG files that could be decoded this way was about 180 kBytes. In order to be able to decode typical images from digital still cameras, that are well in the range of $1 \sim 2$ MBytes, a tool had to be written, that allows the upload of such big JPEG files via the serial interface onto the microcontroller board. Besides from the tool, that was implemented as a Win32 console mode application, software for the receiving part on the microcontroller had to be written as well.

A result of decoding various sized JPEG files of identical resolution was, that the computational complexity of JPEG decoding is $O(N)$, with N being the JPEG file's size, at least this is what the measurement results suggest. The entropy decoding (Huffman decoding) part of the JPEG decoding process was determined to have this computational complexity of $O(N)$ that is responsible for this behaviour. The color conversion part in the backends and the IDCT parts both show a computational complexity of $O(1)$ if chroma subsampling and final image resolution are identical. However, the surprising result of the comparison of decoding times of JPEG files of different sizes but identical chroma subsampling and resolution was, that for typical JPEG files from digital cameras, the amount of time spent in entropy decoding (Huffman decoding) far outweighs the time spent in the IDCT or color conversion stages. Since the Huffman decoding stage must be completed entirely without any omission before the IDCT process can be performed, ways were sought to improve this process. The Huffman decoding part of the JPEGLib turned out to be very efficiently coded and written according to the guidelines laid out in the JPEG standards document. Because of the extensive use of tables in this process, the attempt was made to use as the memory locations for these tables special on-chip memory areas that can be accessed by the controller without waitstates. It turned out, that this way only a marginal performance improvement was possible.

As to the performance of the IDCT process in the JPEGLib, several observations could be made that could lead to improved performance, such as using a true two-dimensional IDCT instead of a row-columnwise approach, and moving constant multiplication factors into the quantization tables in the case of downscaled decoding. Also for scaling to arbitrary sizes in the spatial domain, much more sophisticated aproaches than the simple but fast pixel mapping algorithm should be pursued, but would require substantial changes to the complete color decoding, upsampling and dithering backends with much higher memory requirements.

As a summary, it turned out, that the SDA-6000 microcontroller is well capable of

decoding JPEG files of common resolutions, but that the performance for this is clearly unacceptable. In order to be competitive with already existing but far more expensive solutions, a microcontroller like the SDA-6000 needs at least tenfold processing power or speed.

Index

Bibliography

[1] Nasir Ahmed. The DCT - an algorithm that impacts the world of digital audio and video. *QUANTUM - Research and Scholarship at the University of New Mexico*, 15(1), 1998.

[2] J.F. Blinn. What's the Deal with the DCT? *IEEE Computer Graphics and Applications*, 13:78–83, 1993.

[3] E. O. Brigham. *The Fast Fourier Transform and its Applications*. Prentice Hall, 1988.

[4] A. Ligtenberg C. Loeffler and G. S. Moschytz. Practical fast 1-D DCT algorithms with 11 multiplications. *ICASSP 1989*, 2:988 – 991, 1989.

[5] CCITT. *Information Technology - Digital Compression and coding of continuous-tone still images - requirements and guidelines*. International Telecommunication Union, Sep 1992. The JPEG Standard: CCITT Recommendation T.81 and ISO/IEC International Standard 10918-1.

[6] J. W. Cooley and J. W. Tukey. An algorithm for the machine computation of complex Fourier series. *Mathematics of Computation*, 19:297 – 301, 1965.

[7] E. Dijkstra. Goto statement considered harmful. *Communications of the ACM*, 11(3):147–148, Mar 1968.

[8] E. Feig. A fast scaled DCT algorithm. *Image Processing Algorithms and Techniques*, 1244:2 – 13, 1990.

[9] E. Feig and S. Winograd. Fast Algorithms for the Discrete Cosine Transform. *IEEE Transactions on Signal Processing*, 40(9):2174 – 2193, Sep 1992.

[10] Robert W. Floyd and Louis Steinberg. An adaptive algorithm for spatial greyscale. *Proceedings of the Society for Information Display*, 17(2):75–77, 1976.

[11] Eric Hamilton. *JPEG File Interchange Format*, Sep 1992. JPEG File Interchange Format Version 1.02.

[12] Stephen Hawley. Ordered dithering. *Graphics Gems I*, I:176 – 178, 1990.

[13] R. Tolimieri J. Granata, M.Conner. The Tensor Product: A Mathematical Programming Language for FFTs and other Fast DSP Operations. *IEEE Signal Processing Magazine*, pages 40 – 48, Jan 1992.

[14] JEIDA. *Digital Still Camera Image File Format Standard (Excnangeable image file format for Digital Still cameras: Exif.* Japan Electronic Industry Development Association (JEIDA), Jun 1998. Exif Standard Version 2.1.

[15] T. Kientzle. Implementing fast DCTs. *Dr. Dobb's Journal*, 24:115 – 119, Mar 1998.

[16] Donald E. Knuth. Digital halftones by dot diffusion. *ACM Transactions on Graphics*, 6(4):245–273, 1987.

[17] S. Kuhr. Introduction to the MINIGDI Library. *Sony Engineering Report, VPE-STG-Report-No. 0083-0*, December 2001.

[18] S. Kuhr. Uploading Files onto M2 via the Serial Interface. *Sony Engineering Report, VPE-STG-Report-No. 0088-0*, December 2001.

[19] Mike Morton. A digital "dissolve" effect. *Graphics Gems I*, I:221 – 232, 1990.

[20] T. Natarajan N. Ahmed and K. R. Rao. Discrete cosine transform. *IEEE Transactions on Computers*, Vol. C-23:90 – 93, Jan 1974.

[21] Mark J. Pavicic. Convenient anti-aliasing filters that minimize "bumpy" sampling. *Graphics Gems I*, I:144 – 146, 1990.

[22] William B. Pennebaker and Joan L. Mitchell. *JPEG still image data compression standard.* van Nostrand Reinhold, 1993.

[23] K. R. Rao and P. Yip. *Discrete Cosine Transform - Algorithms, Advantages and Applications.* Academic Press, 1990.

[24] Dale A. Schumacher. A comparison of digital halftoning techniques. *Graphics Gems II*, II:57 – 71, 1991.

[25] Dale A. Schumacher. Fast anamorphic scaling. *Graphics Gems II*, II:78 – 79, 1991.

[26] Dale A. Schumacher. General filtered image rescaling. *Graphics Gems III*, III:8 – 16, 1992.

[27] H.F. Silverman. An Introduction to Programming the Winograd Fourier Transform Algorithm. *IEEE Transactions on Acoustics, Speech and Signal Processing*, ASSP-25(2):152 – 65, Apr 1977.

[28] Spencer W. Thomas and Rod G. Bogart. Color dithering. *Graphics Gems II*, II:72 – 77, 1991.

[29] B. D. Tseng and W. C. Miller. On computing the discrete cosine transform. *IEEE Transactions on Computers*, C-27:966 – 968, Oct 1978.

[30] Ken Turkowski. Filters for common resampling tasks. *Graphics Gems I*, I:147 – 165, 1990.

[31] M. Vetterli and A. Ligtenberg. A Discrete Fourier-Cosine Transform Chip. *IEEE Journal on Selected Areas in Communication*, Vol. SAC-4:49 – 61, Jan 1986.

[32] M. Vetterli and H.J. Nussbaumer. Simple FFT and DCT algorithms with reduced number of operations. *Signal Processing*, 6:267 – 78, 1984.

[33] Gregory K. Wallace. The JPEG Still Compression Standard. *IEEE Transactions on Consumer Electronics*, Apr 1991.

[34] S. Winograd. On Computing the Discrete Fourier Transform. *Mathematics of Computation*, 32:175 –199, Jan 1978.

[35] T. Agui Y. Arai and M. Nakajima. A fast DCT-SQ Scheme for Images. *Transactions of the IEICE*, E 71:1095 – 1097, Nov 1988.

Declaration

Hereby I explain the fact that I have written the available work independently and used none other than the indicated aids. References to the work of others are clearly documented.

Full name: Stefan Kuhr

Signature: .....

Stuttgart, 31^{st} of January 2002.