Vectorization of fast Fourier transform and digital audio broadcast reception

Gerritsen, D.

Award date:
2004

TECHNISCHE UNIVERSITEIT EINDHOVEN

Department of Mathematics and Computer Science

MASTER'S THESIS

Vectorization of Fast Fourier Transform and Digital Audio Broadcast reception

by

D. Gerritsen

Supervisor: prof. dr. C.H. van Berkel

Advisor: dr. N. Engin (Philips Research)

Eindhoven, November 2003

Preface

This report is my Master's thesis, resulting from my graduation project for my studies in Computer Science at the Eindhoven University of Technology. The graduation project was performed at the Embedded Systems Architectures on Silicon department of Philips Research Laboratories in Eindhoven.

This report describes the research I have done on mapping Fast Fourier Transform algorithms on a co-vectorprocessor architecture called CVP. Besides a proposal for an implementation of such algorithms on CVP, also suggestions have been made for ISA extensions that would improve FFT performance on CVP.

Furthermore I have looked into relevant applications using FFT, one of which is Digital Audio Broadcasting. For several kernels involved in DAB reception an analysis has been made on the feasibility of mapping those kernels to CVP.

I would like to express my gratitude to Nur Engin for her supervision at Philips Research, Kees van Berkel for his role as my university supervisor, Srinivasan Balakrishnan and Patrick Meuwissen for their help with the CVP compiler and assembler/simulator respectively and finally Albert van der Werf for allowing me to perform my graduation project in his group.

Abstract

CVP (Co-Vector Processor) is a co-processor being developed in the SW-Modem project, combining data and instruction level parallellism in order to obtain a low-cost, power-efficient solution for the signal processing involved in third generation mobile communication.

This document presents a mapping of the Fast Fourier Transform algorithm as well as the kernels involved in baseband processing for Digital Audio Broadcast (DAB) reception to this architecture. Based on schedules for the inner loops of these algorithms cycle count estimates are calculated. Furthermore, bottlenecks in mapping the algorithms to the current architecture are pointed out and possible solutions are suggested. Based on the suggested architectural extensions, estimates are presented for the possible performance on an extended version of CVP.

Finally, a feasibility analysis is presented on the implementation of a DAB receiver on both the current CVP architecture and the extended version, in this document referred to as CVP0 and CVP0x respectively.

Contents

List of Figures

List of Tables

Glossary

ACU	Address Computation Unit
ALU	Arithmetic Logic Unit
AMU	ALU-MAC Unit
ASIC	Application Specific Integrated Circuit
CDMA	Code Division Multiple Access
CIF	Common Interleaved Frame
COFDM	Coded OFDM
CVP	Co-Vector Processor
DAB	Digital Audio Broadcasting
DD	Differential Demodulation
FFT	Fast Fourier Transform
FIC	Fast Information Channel
FIR	Finite Impulse Response
ISA	Instruction Set Architecture
LAN	Local Area Network
LF	Logical Frame
MAC	Multiply-Accumulate
MIPS	Mega-Instructions Per Second
MPEG	Moving Pictures Expert Group
MSC	Main Service Channel
OFDM	Orthogonal Frequency Division Multiplex
QPSK	Quadrature Phase Shift Keying
SFU	Shuffle Unit
SIMD	Single Instruciton Multiple Data
SLU	Shift Left Unit
SRU	Shift Right Unit
VMU	Vector Memory Unit
VLIW	Very Long Instruction Word

1 Introduction

1.1 General introduction

In the software-modem research project, conducted in the Embedded Systems Architectures on Silicon group at Philips Research, a co-vector processor is being developed, which is intended to serve mainly as a low-cost, low-power, programmable solution for the baseband processing involved in various third generation wireless communication standards, such as UMTS, TD-SCDMA and IS-2000.

In order to meet the computational requirements while at the same time dissipating as little power as possible, a high degree of parallellism is used. This involves data as well as instruction level parallellism.

Since this coprocessor is intended to be used in third generation mobile handsets, it is interesting to explore on other possible mobile aplications, such as wireless LAN, digital audio and video reception, etcetera.

Since fast fourier transforms are used in many of those applications, this will be the first kernel to be discussed. This part is covered in section 2. Section 3 covers one of the applications that involves the use of FFT, namely Digital Audio Broadcasting. It contains a feasibility analysis for implementing various kernels involved in the baseband signal processing for DAB reception. The final section contains the conclusions and suggestions for future study on the subject.

1.2 The Co-Vector Processor

To achieve the level of parallellism needed to meet the computational requirements, CVP combines elements of so-called VLIW and SIMD processors. Like a VLIW processor, CVP has multiple different functional units, each of which can typically perform one operation every clock cycle. Furthermore, every sub-instruction processed by a functional unit effects multiple data elements at once, a feature CVP has in common with so-called SIMD processors.

CVP sub-instructions operate on 256 bit vectors, which can be organized in several ways, ranging from 32 8-bit word elements up until 8 32-bit elements. Complex vectors exist in two forms, containing 16 elements of word sized real and imaginary components, or 8 elements of double word sized complex values.

The CVP consists of seven functional units. These are the Instruction Distribution Unit, Vector Memory Unit, Code Generation Unit, ALU-MAC Unit, Shuffle Unit, Shift Left Unit and the Shift Right Unit. Since there is no central register file, these units are connected by a full interconnect network in order to facilitate for communication between functional units. Communication between the functional units is performed by means of send and receive instructions. Every functional unit can send values (either explicit or implicit) that can be received by one or more other units. Every unit can send data to and receive from any other unit during each cycle. The exception to this is the ALU-

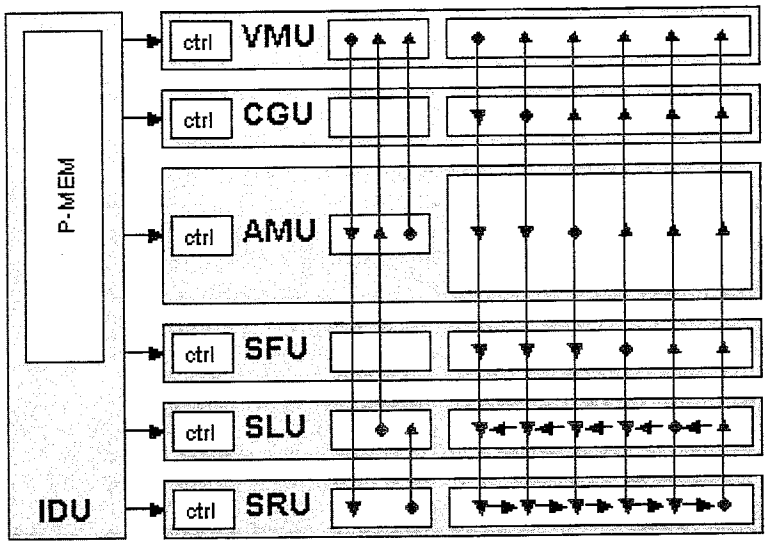

Figure 1: The CVP framework

MAC unit, which can receive data from two units every cycle. The CVP framework is graphically depicted in figure 1.

In short, the purpose of the several functional units is as follows.

- The Instruction Distribution Unit (IDU) contains the program memory and handles the distribution of the sub-instructions to the functional units. Allowed control statements involve subroutine calls and zero-overhead looping.
- The Vector Memory Unit (VMU) contains the vector memory and handles all memory read and write accesses. Memory accesses can pertain to vectors as well as scalar values. Indexing the memory locations is done using one of the address computation units (ACUs).
- The Code Generation Unit (CGU) is used to generate CDMA code chip vectors, the code type required for the UMTS/FDD standard. Future extensions could be to include codes for other standards as well.
- The ALU-MAC Unit (AMU) performs all arithmetic and logic functions. These include operations where arithmetic is performed elementwise on multiple vectors, as well as intra-vector operations, where the arithmetic is performed on elements within a single vector.
- The Shuffle Unit (SFU) is used to rearrange elements within a vector. At the moment only half shuffles are supported, which can shuffle only the odd or even indexed half of a vector in a single instruction.
- The Shift Left Unit (SLU) can shift elements in a vector to the left on a word, double word or quad word basis. The value shifted in at the right side of the vector can be either zero or a value presented to the scalar register of the SLU.

Functional unit	Instruction	Meaning
VMU	SEND	Send a scalar value
	SENDL	Send a memory line
	SENDV	Send a vector
	RCVL	Receive a memory line
AMU	RCV	Receive a vector
	SND	Send a vector
	MUL	Multiplication
	ADD	Addition
	SUB	Subtraction
	ASR	Arithmetic shift right (division by 2^n)
	ASRA	Arithmetic shift right of an accumulator register
SFU	RCV	Receive a vector
	CONF	Receive a configuration vector to one the configuration registers
	FULL	Perform a full shuffle on the vector elements

Table 1: Used CVP assembly instructions

- The Shift Right Unit (SRU) is similar to the SLU, bit it shifts in the opposite direction. In addition it can merge sparse vectors resulting from consecutive intra-vector operations.

For a more detailed description of the instruction set architecture, see [1, 2].

For the current CVP tools exist for assembly, simulation and debugging. Although pipelining and caching effects inside functional units are hidden from the programmer, assembly level programming is still a complex task, because it involves error-prone tasks such as scheduling of functional unit operations, and register allocation. In order to relieve the programmer of these complex tasks, a higher level programming language called CVP-C exists. It is however still up to the programmer to transform the original algorithm operating on scalar values into a vectorized version. Since the CVP-C compiler became available during the course of this graduate work and is not yet fully reliable, most programming was performed in assembly language.

In this document we shall present schedules for some parts of the algorithms we will look into. These schedules will only include the functional units that are actually used for the particular (part of the) algorithm. In these schedules we use the assembly instruction names to denote the operations. The meaning of these operations can be found in table 1. This table contains the 'basic' instruction names only. In an assembly program the AMU arithmetic instructions are appended with a suffix the denotes the precision to be used. These suffixes and their meaning can be found in table 2.

Besides the current CVP architecture we shall also look into possible architectural extensions that could improve the performance for the algorithms under investigation. In order to distinguish between the current and extended version, we shall refer to these as CVP0 and CVP0x.

Instruction suffix	Short for	Indicated precision
i	int	8 bit integer number
di	double int	16 bit integer number
qi	quad int	32 bit integer number
ci	complex int	$2 * 8$ bit complex number
cdi	complex double int	$2 * 16$ bit complex number

Table 2: Assembly instruction suffixes

1.3 Fast Fourier Transform

In a wide variety of applications, signal analysis in the frequency domain is performed. A common way to transform a signal from time to frequency domain is by means of the fourier transform. When this transform is is computed over a sampled signal, it is called the discrete fourier transform, in short DFT. The DFT of a finite sequence of N samples is defined by

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \qquad k = 0, 1, \ldots, N-1,$$

where $W_N = e^{-j(2\pi/N)}$.

Algorithms that exploit periodicity and symmetry properties of W_N^{kn} are collectively known as Fast Fourier Transform (FFT) algorithms. Applications involving the use of FFT include spectrum analysis, digital audio and video broadcasting and wireless LAN. Because of the importance of the algorithm many attempts have been undertaken to efficiently implement the algorithm on current Digital Signal Processing (DSP) architectures.

1.4 Digital Audio Broadcasting

Digital Audio Broadcasting (DAB) is a system designed to become the successor to the present-day analogue radio. Current AM and FM radio signals are subject to numerous kinds of interference on the way from transmitter to the receiver. These interferences are caused by obstacles such as mountains and high-rise buildings as well as weather conditions. The DAB system is far less sensitive for these kinds of interferences because of the application of forward error correction coding, interleaving, etcetera. This results in a distortion-free reception of CD quality sound. Aside from a higher quality of the received audio, DAB offers further advantages, as it can not only carry radio, but also data such as text, pictures and even video.

In contrast to conventional analogue radio signals carrying one service per frequency, DAB signals can carry multiple services, both audio and data. Such a DAB transmission signal is commonly referred to as a DAB ensemble, and carries a multiplex of audio and/or multimedia services. The number of services, also called sub-channels, depends on the (audio or data) bitrate and the level of error correction coding that is applied.

Figure 2: DAB world coverage map (January 2003)

DAB is currently being implemented in many parts of the world. See figure 2 for the DAB coverage as of January 2003. In the Netherlands, the nation-wide service broadcaster NOS put on air an ensemble containing the radio stations Radio 1, Radio 2, 3FM, Radio 4, 747AM and 'De Concertzender', broadcasted through the transmitter located in Lopik. Nation-wide DAB coverage can be expected within a few years.

1.5 Problem statement and objectives

Since the CVP is intended to be used in third generation mobile handsets, it is desirable that it can also be used in other mobile aplications, such as wireless LAN, digital audio and video reception, etcetera. Since FFT is a common kernel in many of those applications, good FFT performance on CVP is required. Furthermore, an analysis of the other kernels involved in those applications is desirable.

The first goal of this graduate project is to study several FFT algorithms and transform these into a vectorized version, intended to be implemented on CVP. Because vector processing in itself is not a new subject, vectorization attempts have been made for other vector processors, such as VIRAM and OnDSP. The novelty lies in the fact that CVP combines this data parallellism with instruction level (VLIW) parallellism. The challenge is therefore to exploit this parallellism in order to make full use of CVP's processing capabilities.

Based on these vectorizations, bottlenecks in both the algorithms itself and the current CVP architecture should be pointed out. A second goal is therefore to come up with possible extensions to the current CVP architecture that would improve the performance for FFT. Estimates on the FFT performance for a hypothetical extended version, which we will call CVP0x, will be made. For both CVP0 and CVP0x the FFT performance will

be compared to that of other DSP architectures.

Besides the mapping of FFT, another goal is to analyse the feasibility of implementing a DAB receiver on CVP. For the various DAB reception kernels too the bottleneck with respect to the current architecture should be pointed out, as well as architectural extensions that would solve these problems.

2 Fast Fourier Transform

2.1 Introduction

In order to come up with a good vectorization, we shall first take a look at several existing FFT algorithms. We hope that by implementating a first naive vectorization we will be able to point out the performance bottlenecks. Possible solutions, both algorithmic changes and CVP architecture extensions will then be suggested to address these bottlenecks, as well as the performance gain that could be achieved by these extensions. The resulting performance figures will be compared to benchmarks of existing architectures.

The DFT of a finite sequence of N samples is defined by

$$X[k] = \sum_{n=0}^{N-1} x[n] W_N^{kn}, \qquad k = 0, 1, \ldots, N-1, \tag{1}$$

where $W_N = e^{-j(2\pi/N)}$.

Since $x[n]$ may be complex, direct evaluation of the DFT according to Eq. 1 requires N complex multiplications and $N-1$ complex additions. To compute all N values would therefore require N^2 complex multiplications and $N(N-1)$ complex additions.

By exploiting symmetry and periodicity properties of W_N^{kn}, computation of a DFT can be done much more efficiently, especially when N is a product of two or more integer numbers. This allows for decomposition of the DFT into successively smaller DFTs, which all together require less computation.

2.2 Overview of FFT algorithms

The most commonly known FFT algorithms reduce the required computation by decomposing the DFT into successively smaller DFTs. For this approach to work, the length of the DFT, N, should be a composite number, i.e. a product of two or more integer factors. These algorithms differ in how the decomposition is performed.

2.2.1 Fixed radix

If the decomposition of the DFT is done by using the same factor for every successive decimation, we call the algorithm fixed radix. For the decomposition we have two options. Algorithms that decompose the input sequence $x[n]$ into successively smaller subsequences are called *decimation-in-time algorithms. Decimation-in-frequency* algorithms perform the decomposition on the output sequence $X[k]$.

How this decomposition speeds up computation is most easily illustrated by considering the case that N is a power of 2. The N-point DFT is then split into two $(N/2)$ point DFTs, one consisting of the even numbered points in $x[n]$ and the other one consisting of

the odd numbered points. This way we get

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_N^{2rk} + \sum_{r=0}^{(N/2)-1} x[2r+1] W_N^{(2r+1)k} \tag{2}$$

Since $W_N^2 = W_{(N/2)}$, Eq. 2 can be rewritten as

$$X[k] = \sum_{r=0}^{(N/2)-1} x[2r] W_{(N/2)}^{rk} + W_N^k \sum_{r=0}^{(N/2)-1} x[2r+1] W_{(N/2)}^{rk} \tag{3}$$

In Eq. 3, both sums can be seen as an $(N/2)$-point DFT. We can continue this decompostion until we end up with 2-point DFTs only, which do not involve complex multiplications. Therefore the only remaining complex multiplications are the ones with the factors W_N^k in Eq. 3, the so-called *twiddle factors*, reducing the computation complexity from $\mathcal{O}(N^2)$ to $\mathcal{O}(N \log_2 N)$.

The entire radix 2 decimation-in-time algorithm for a 16-point FFT is graphically depicted in figure 3. In this flow graph, the branch transmittances correspond to multiplication coefficients. The value for a node in the graph is calculated by summing all branches entering the node, multiplied by their branch transmittances. For every branch where the transmittance is omitted, it is assumed to be 1. As can be seen in the flow graph, the input to the algorithm is not in sequential, but in so-called bit-reversed order. For more information and a more detailed derivation of the algorithm, see [3].

Other fixed radix algorithms are similar to the radix 2 algorithm. For instance in a radix 4 algorithm, the input (for decimation-in-time) or the output sequence (decimation-in-frequency) is decomposed iteratively into 4-point DFTs.

2.2.2 Other algorithms

In the previous section, we have discussed the case that N is an integer power of some radix r. If we can decompose the DFT using the same factors every decimation step, we get a fixed radix algorithm as discussed in the previous section, but other algorithms exist. We can use different decimation factors during consecutive stages, for instance using as many factors of 4 as possible and finishing with a factor 2. These algorithms are called mixed radix algorithms. Furthermore, FFT algorithms exist which exploit the fact that N is a product of two or more relatively prime factors.

For a more detailed description of these other types of FFT algorithms, see [3].

2.3 A naive implementation

This section describes an implementation of a radix 2 decimation-in-time FFT algorithm for CVP. It also contains figures on the performance of this implementation and an analysis of the encountered performance bottlenecks.

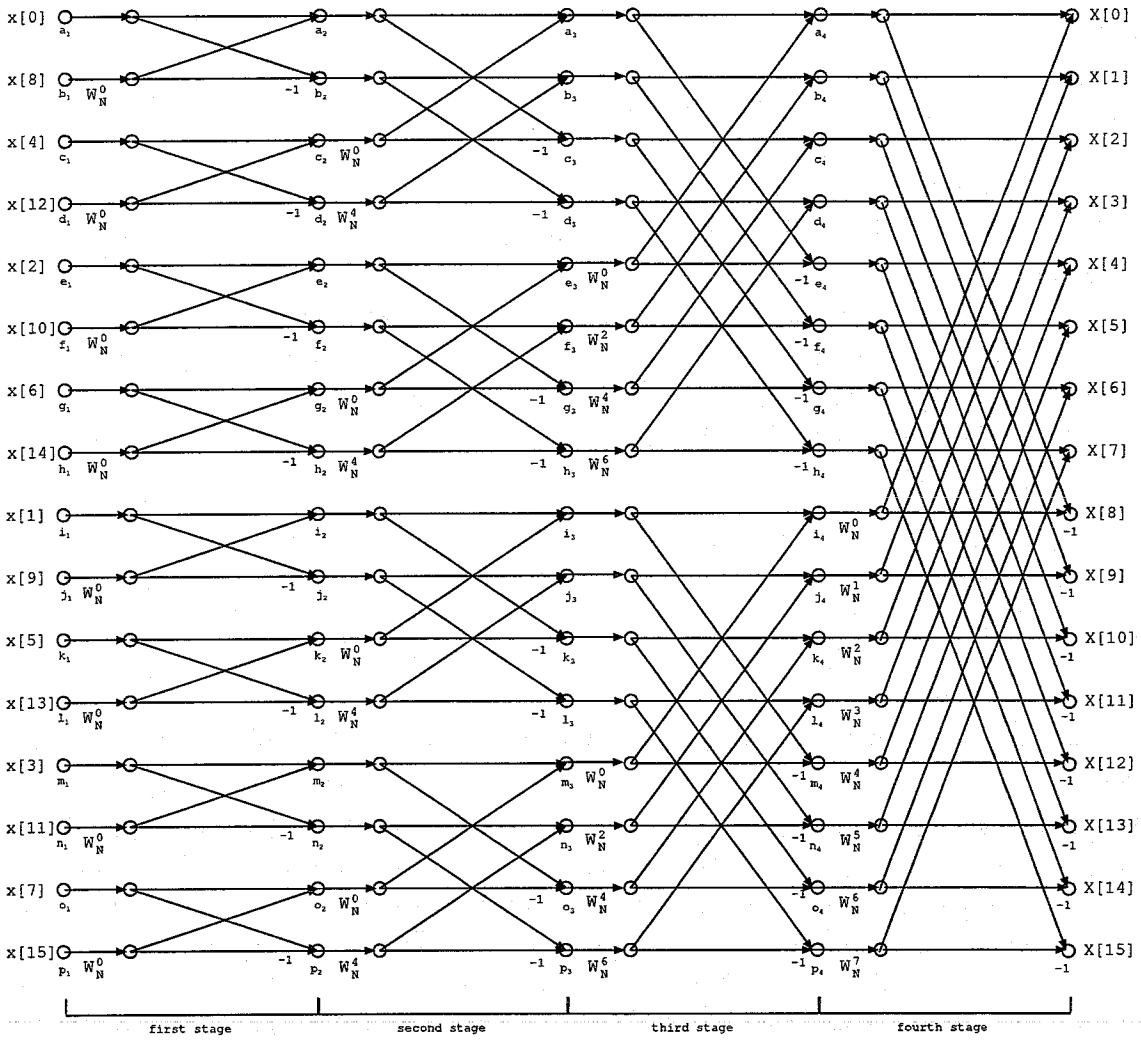

Figure 3: Flow graph for a 16-point decimation-in-time FFT

2.3.1 Brief description of the algorithm

The implementation is basically a vectorization of the radix 2 decimation-in-time algorithm described in section 2.2.1. According to [1, 2], a vector on the CVP can contain 8 or 16 complex values, depending on the precision. We shall define the vector size as

$$N_v = \frac{128}{P},$$

where N_v is the number of elements in a vector and P is the precision in bits of the real as well as the imaginary part of a complex vector element.

As can be seen in figure 3, for the first $\log_2 N_v$ computation stages the values that have to be summed reside in a single vector, whereas for the remaining $\log_2 N - \log_2 N_v$ stages, these values reside in different vectors. We shall refer to these stages as the *intra-vector* and *inter-vector* stages respectively.

Vectorizing the algorithm is straightforward for the inter-vector stages. If we assume $N_v = 8$, all computation for the fourth stage of figure 3 can be done with only a few vector operations. Having a vector X_0 consisting of a_4 through h_4, X_1 containing i_4 through p_4 and W holding the twiddle factors W_N^0 through W_N^7, we could compute the new values for X_0 and X_1 as follows

$$\begin{aligned} &tmp := W \times X_1 \\ &; X_0' := X_0 + tmp \\ &; X_1' := X_0 - tmp \end{aligned}$$

where $X \times Y$ is the vector resulting from the element-wise multiplication of the vectors X and Y.

For the intra-vector stages, things are a bit more complicated. Since the values that have to be summed together reside in a single vector, we cannot implement the butterflies in the same manner as we did for the inter-vector stages. In order to be able to use the same basic computation, we will first have to make sure that the values that should be added together are distributed over separate vectors. A means that CVP offers for this is the shuffle unit. It allows us to specify a shuffle pattern, that describes which element of the input vector should end up at which position in the output vector. The current CVP architecture only supports half shuffles, meaning that we can perform a shuffle on only the odd or even indexed vector elements in a single instruction. Since the shuffle result is stored in an implicit target register in the shuffle unit, we can achieve a full shuffle by sequentially performing an odd and even shuffle.

By performing the correct shuffles after each of the intra-vector stages, we can rearrange the vector elements in such a way that we can perform the same basic butterfly operation as we did for the inter-vector stages. For a length N and vector size N_v, we have a total of $M = N/N_v$ vectors, X_0 through X_{M-1}. Shuffling is performed between every two consecutive vectors, X_{2i} and X_{2i+1} for $0 \leq i < (M/2) - 1$. The shuffle pattern to be used

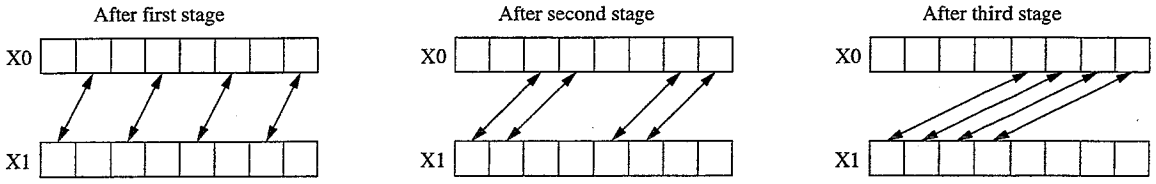

Figure 4: The shuffle pattern for $N_v = 8$

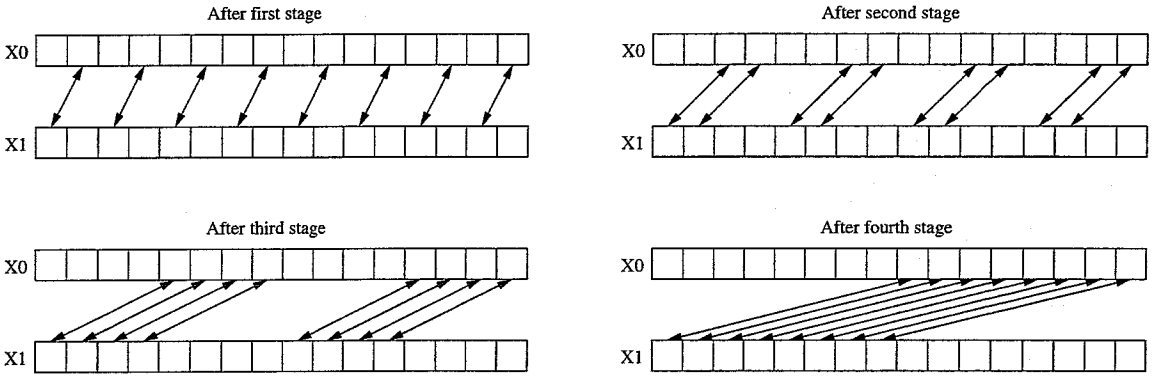

Figure 5: The shuffle pattern for $N_v = 16$

is depicted in figure 4 for $N_v = 8$ and figure 5 for $N_v = 16$. This shuffling mechanism was suggested in [4].

As can be seen in figure 3, we have to supply the input to the first computation stage in a very specific order, which we will call 'bit-reversed'. This terminology becomes obvious when we look at the indices of the inputs written as binary digits. The order in which we supply the input is regularly increasing, but with the bits reversed. In order to avoid intra-vector operations, we have to interleave these values over two consecutive vectors. As we shall see later, it is this bit-reversed ordering which has to be done in advance that can drastically decrease performance.

2.3.2 The actual algorithm

Following is the pseudo-code for the naive algorithm described in the previous section. We assume that all twiddle values to be used reside in memory, grouped in correctly ordered vectors. This makes computation much more efficient, since we do not have to shuffle the twiddle values in order to get the correct values into a vector, at the expense of some extra memory usage. Furthermore, we assume that the FFT length N is at least twice the vector size N_v, so we have at least two vectors of input data.

```
|[
  doBitreverse()
 ;stage := 0
 ;do stage < log2 Nv →
```

```
    butterfly := 0
   ;do butterfly < N/(2N_v) →
        computeButterly()
      ;butterfly := butterfly + 1
    od
   ;shuffle()
   ;stage := stage + 1
 od
;do stage < log_2 N →
    butterfly := 1
   ;do butterfly < N/(2N_v) →
        computeButterly()
      ;butterfly := butterfly + 1
    od
   ;stage := stage + 1
 od
]|
```

The function computeButterfly is given by

```
|[
  X_0 := M[&X_0]
 ;X_1 := M[&X_1]
 ;W := M[&W]
 ;tmp := W × X_1
 ;M[&X_0] := X_0 + tmp
 ;M[&X_1] := X_0 − tmp
]|
```

$M[i]$ denotes the vector residing in the vector memory at location i. The calculation of the correct memory addresses $\&X_0$ and $\&X_1$ is done using the address calculation units present in the vector memory unit. The bit-reversed ordering prior to the actual butterfly calculations is performed by means of the shuffle unit. As we shall see later, time tends to grow to a large number of clock cycles for large values of N.

The schedule for the inner loop that performs the actual butterfly computations is illustrated in figure 6.

2.3.3 Performance figures and bottlenecks

As can be seen in figure 6, performing the actual butterfly computations costs five clock cycles per vector pair, per computation stage. This adds up to $(\log_2 N) * \frac{N}{2N_v} * 5$ cycles.

cycle number	VMU vopcode	VMU aopcode	AMU receive1	AMU receive2	AMU vopcode
n	RCVL	INCR			MULcdi
$n+1$	SENDL	INCR			
$n+2$	SENDL	INCR	RCV(VMU)		ASRAcdi
$n+3$	SENDL	INCR	RCV(VMU)		ADDcdi
$n+4$	RCVL	INCR	RCV(VMU)		SUBcdi

Figure 6: Scheduling for the inner loop calculating the actual butterflies

In practice, due to overhead for setting up the ACUs, the number of clock cycles will be higher.

Furthermore, we have to add some cycles for the shuffling after each of the intra-vector stages. For every vector pair we have to perform four full shuffles. Since for every two consecutive shuffles, we have to use a different shuffle pattern, this will cost three cycles per full shuffle for the configuration, odd and even shuffle instructions, adding up to a total of $(\log_2 N_v) * \frac{N}{2N_v} * 12$ clock cycles, only a very small part of which can be executed in parallel with the butterfly computations.

Finally, we have to take into account the clock cycles needed to perform bit reversal prior to the actual computation. Again, like for the shuffles after the intra-vector stages, every full shuffle costs three cycles. So in order to derive the cycle count for the necessary bit reversal, we need to know how many shuffles bit reversal takes for an FFT of length N.

To address this problem, let us define the exact order in which we need to supply the input data. Consider the input data to be an array X of length N. We can divide this array into $\frac{N}{N_v}$ vectors X_i of length N_v, each vector X_i containing the values $X[i * N_v]$ through $X[N_v - 1 + i * N_v]$. We define the function b as follows

$$b((i_0, i_1, \ldots, i_{\log_2 N-1})_2) = (i_{\log_2 N-1}, i_{\log_2 N-2}, \ldots, i_0)_2$$

where $(i_0, i_1, \ldots, i_{\log_2 N-1})_2$ is the binary representation of the number i, using $\log_2 N$ bits. Having the input data array X, we will call the array Y its bit-reversed order if

$$Y[b(i)] = X[i]$$

Similarly we could have defined the array Y by

$$Y[i] = X[b(i)].$$

As mentioned before in section 2.3.1, the order we want prior to computation is slightly different than regular bit-reversed. We provide the input to the first computation stage as vectors Y_i, where every vector $Y_{(2j)}$ contains the values $Y[2jN_v]$, $Y[2 + 2jN_v]$, ..., $Y[2N_v - 2 + 2jN_v]$ and every vector $Y_{(2j+1)}$ contains $Y[1 + 2jN_v]$, $Y[3 + 2jN_v]$, ..., $Y[2N_v - 1 + 2jN_v]$. We now want to derive the number of shuffles needed for this reordering.

N	N_v	Number of full shuffles
16	8	2
32	8	8
64	8	32
≥ 128	8	N
32	16	2
64	16	8
128	16	32
256	16	128
≥ 512	16	N

Table 3: The required number of full shuffles for bit reversal

We observe that the indices of the values that initially reside within the same vector X_i, that is iN_v through $N_v - 1 + iN_v$, written as $\log_2 N$ bit numbers, share the same $\log_2 N - \log_2 N_v$ most significant bits. If we now calculate the bit-reversed order, then these values share the same $\log_2 N - \log_2 N_v$ least significant bits. We can therefore conclude that the corresponding values in array Y will be $k\frac{N}{N_v}$ apart for $k \neq 0$.

In our so-called 'interleaved' bit-reversed order, the values originating from one single vector X_i will all end up in different vectors Y_j if the corresponding values in array Y are separated by at least $2N_v$. Since we know that the values originating from a single vector X_i will be multiple of $\frac{N}{N_v}$ apart in the array Y, we conclude that all elements from one single vector have to end up in all different vectors if $\frac{N}{N_v} \geq 2N_v$ or $N \geq 2N_v^2$. This means that for each of the $\frac{N}{N_v}$ vectors we have to perform a full shuffle per vector element. In this case, the bit-reversed ordering will take a total of N full shuffles. For smaller values of N the number of required shuffled can easily be deduced as well. The number of shuffles required for bit reversal for different values of N and N_v can be found in table 3.[1]

Combining the numbers for bit reversal, inter-stage shuffling and the actual butterfly computations, we can derive the estimated cycle counts for computing FFTs for several different sizes. These estimates can be found in table 4. In calculating these numbers, for the inter-vector stages six cycles per vector pair per stage were assumed, whereas this could ideally be five cycles. This was done to compensate for the overhead caused by reconfiguration of the ACUs and loop initiation. Especially for the larger sized FFTs, this gives rise to somewhat pessimistic estimates. For the smaller sized FFTs the estimates are quite accurate. Implementations of 32 and 64 point FFTs have cycle counts within a few percent range of the estimated numbers. These implementations however followed a slightly less naive scheme, hiding the bit reversal behind the actual computations. This has been accounted for by adding three times the required number of shuffles for bit reversal to the cycle count of the simulated cycle counts.

[1] As mentioned earlier, we only take into account the cases where $N \geq 2N_v$.

N	N_v	Estimated cycle count
16	8	60
32	8	140
64	8	350
128	8	935
256	8	1955
512	8	4100
1024	8	8580
32	16	75
64	16	165
128	16	400
256	16	1035
512	16	2935
1024	16	6055

Table 4: Cycle count estimates for FFT, naive implementation

2.4 Improving FFT performance on CVP

The implementation presented in the previous section is far from optimal. Therfore we will present some measures for enhancing the FFT performance in this section.

2.4.1 An alternative implementation

In the naive implementation of the radix 2 algorithm presented earlier, a disproportionately large part of the total cycle count is consumed by the intra-vector stages. This is due to the necessary bit reversal and the inter-stage shuffles. In this section we will make an effort to come up with a more efficient implementation that reduces the cycle count for those first few computation stages.

Looking at the flowgraph in figure 3, we can see that in order to compute the values resulting after the intra-vector stages for two vectors, we do not need any other values than the ones that reside within these two vectors before the first stage. Therefore we can compute all intra-vector stages for one vector pair before moving on the next pair, thereby eleminating the need to store and load new vectors in between these stages. Another advantage of this approach is that performing these $\mathrm{Log}_2 N_v$ intra-vector stages for a single pair can be combined with the bit-reversed ordering of the next vector pair. That way the bit reversal for all vectors except the first pair can be 'hidden' behind the butterfly computations and inter-stage shuffles during the intra-vector stages.

However, during the intra-vector stages the shuffle unit is already utilized most of the clock cycles. It is therefore not desirable to use the shuffle unit for the reordering of the next vector pair, since it would then become inpossible to hide the bit reversal completely. For the current CVP architecture however, the shuffle unit is the only means available for such reordering. Suggestions for enhancements to the architecture and instruction set in

order to cope with this problem, are given in the next sections.

2.4.2 Bitreversed addressing acu mode

A possible solution that would make it possible to perform bit reversal without utilizing the shuffle unit, would be to extend the instruction set architecture to support bitreversed addressing. One could think of an extra 1-bit register per ACU, or probably only for one of them, indicating whether or not to use bitreversed addressing and an extra ACU suboperation to set the value for this register. Another option is to only include an extra ACU suboperation for a bitreversed increment of the offset register of that particular ACU.

In both cases, performing a 'bit-reversed increment' would involve three consecutive steps: bitreversing the current value of the offset register, adding the value of the increment register to the bit-reversed offset value and finally bitreversing the incremented offset value again.

In presence of a bit-reversed addressing mode we can use scalar memory read operations to read the values for the next vector pair to be processed. Using the shift right unit, we can combine these scalar values to form the actual vectors. Since we do not have to use the already heavily utilized shuffle unit, we can perform bit reversal for the next vector pair in parallel with the intra-vector stages for the current pair.

This approach was implemented using the current CVP assembler/simulator toolset. Since the current architecture and toolset do not support this bit-reversed addressing this was simulated by providing the input data in bit-reversed order and reading the inputs on a scalar level using regular ACU addressing. In this implementation the computation of the intra-vector stages for a single vector pair took 57 clock cycles for single precision and 44 cycles for double precision data.[2] For single precision we therefore have 57 clock cycles in which to schedule 32 scalar memory accesses in order to read the values for the next vector pair. For double precision we have to schedule 16 accesses in 44 clock cycles. It turns out that this is indeed possible without having to schedule a scalar and a vector memory access within the cycle, which could cause a memory stall cycle. It is therefore possible to almost completely eliminate the extra clock cycles spent on bit reversal, as we had with the naive algorithm. The only cycles we have to spend on bit reversal are those we need to reorder the first vector pair.

2.4.3 Scatter-gather memory accessing

Another architectural enhancement that is currently under investigation, is to divide the vector memory into sixteen separate memory banks, which can be accessed in parallel. The programmer can specify per bank which element to fetch, thereby making it possible to fetch all desired elements in just one memory access. After the fetch it is still required to reorder the vector elements themselves in the right order, but that will then only take one full shuffle per vector.

[2]For single precision data $N_v = 16$ and we have $\mathrm{Log}_2\ 16 = 4$ intra-vector stages, whereas for double precision ($N_v = 8$) the number of intra-vector stages is 3, hence the larger cycle count for single precision.

If we have such a so-called scatter-gather addressing method, we can perform the reordering of a vector pair in parallel with the butterfly computations of the previous pair, thereby eliminating the cycle count for bit reversal except for the cycles needed to reorder the first vector pair. For this to work the programmer has to make sure that the elements to be fetched for each vector all reside in different memory banks, in order to prevent bank conflicts. This requires a little extra effort by the programmer, but greatly reduces the number of cycles needed to reorder a vector of input data. If we can then reorder more than one vector pair during the intra-vector stages of a previous pair, it may even be possible to introduce some extra software pipelining by performing the inter-stage shuffles for one pair in parallel with the butterfly computations of another pair. This will be explored in section 2.4.5.

2.4.4 Single cycle full shuffle instruction

Another improvement would be if it were possible to perform a full shuffle in a single cycle instead of having to combine two half shuffles. Because there is large amount of shuffles that has to be performed during the intra-vector stages, this would result in a reasonable reduction of the cycle count for these stages.

We will take the cycle counts for the intra-vector stages given in section 2.4.2 as a starting point. For double precision data, since the intra-vector stages require a total of 12 full shuffles per vector pair, this would reduce the cycle count per vector pair by 12 (one cycle per shuffle), resulting in 32 instead of 44 cycles per pair. Since we have a $N/16$ of these pairs, this would result in a total saving of $\frac{3}{4}N$.

For the 8-bit case, things are somewhat more complicated. In this case the intra-vector stages require 16 full shuffles per vector pair. Therefore one would think that the cycle count for the intra-vector stages per vector pair would be reduced by 16, that is from 57 to 41. Having $\frac{N}{32}$ vector pairs for a length N FFT, this amounts to a total saving of $\frac{1}{2}N$ cycles.

However, in this case memory stalls can become the bottleneck. During the intra-vector stages we have to perform some line memory accesses to read the necessary shuffle patterns, as well as some to store the vector pair after having calculated the intra-vector stages. The number of line accesses needed adds up to 18 (16 read accesses for the shuffle patterns and two write accesses for the two resulting vectors after the intra-vector stages). Would we use the bit-reversed addressing method to reorder the next vector pair, this would cost another 32 (scalar) memory accesses. We would therefore have to do 50 memory accesses. Although the current architecture does allow a line and a scalar access to be issued in a single instruction, this would generally cause a stall cycle due to a cache miss. See [1] for more information. This would mean that for the single precision case, the full shuffle would only reduce the cycle count for the intra-vector stages from 57 to 50, which would be a total saving of $\frac{9}{32}N$ instead of $\frac{1}{2}N$.

If we could use the scatter-gather memory accessing discussed in the previous section, we would only need 3 memory accesses in order to read the next vector pair, two for the actual input data and one for the shuffle pattern needed, which happens to be the same for

both input vectors. This would mean that having this banked memory architecture, we can reduce the cycle count for the intra-vector stages to 41 for the single precision case by introducing a full shuffle, because we have fewer memory accesses and therefore do not encounter stall cycles due to cache misses.

2.4.5 Software pipelining

If we assume some of the proposed modifications to exist in the current CVP architecture, such as the scatter-gather addressing or some other efficient means to read input vectors in a bit-reversed manner withouth the use of the shuffle unit, and we also assume the single-cycle full shuffle and a sufficient number of SFU configurations registers, we can improve the FFT performance by introducing software pipelining. Software pipelining is the interleaving of instructions that operate on independant sets of data, so that operations that are dependant for one data set can be scheduled in parallel with those for the other data set, thereby avoiding the data dependencies.

For the intra-vector stages we can pipeline the arithmetic operations on one vector pair with the inter-stage shuffles of another pair, thereby interleaving two vectorized butterflies which together cost less clock cycles than by computing them sequentially. Because we assume an efficient means to read the input in a bit-reversed order, we will not take into account the bit reversal in the schedule. A schedule for this pipeline is depicted in figure 7. In this schedule operations on two independant sets of data are pipelined. For the sake of readability these operations are denoted using different colors. Besides the use of color, also a number is appended to the instruction indicating the data set it operates on.

The inter-vector stages can be optimized further by introducing a pipelined multiply-scale instruction. This instruction would perform a regular multiplication, followed by a scale by the desired amount. It should furthermore offer the possibility to initiate another ALU or MAC instruction in the next cycle. This is currently not allowed, but from a hardware point of view (in particular the number of read and write ports to and from the AMU register file) there is no compelling reason not to allow this. By having such an instruction we could pipeline the butterflies for the inter-vector stages in such a way that an entire vector butterfly (two vectors input, two vectors output) can be performed in only 3 cycles. See figure 8 for the resulting schedule.

However, for larger FFT sizes it is impossible to maintain all intermediate results in the AMU register file. It is therefore necessary to store these intermediate results in the vector memory. This adds up to five memory accesses per vector butterfly (two read accesses for the two input vectors, one for the vector of twiddle factors and two write accesses for the output vectors). This degrades the possible performance from 3 to 5 cycles for a vectorized butterfly. The FFT lengths where this extra pipelining for the inter-vector stages is beneficial is $N \leq 64$ for $N_v = 8$ and $N \leq 128$ for $N_v = 16$.

VMU	AMU			SFU		
send/receive	receive1	receive2	vopcode	receive	config	shuffle
SENDL1						
SENDL1	RCV1(VMU)					
	RCV1(VMU)					
			SUB1			
SENDL2			ADD1	RCV1(AMU)		
SENDL2	RCV2(VMU)		SUB1	RCV1(AMU)		FULL1
	RCV2(VMU)		ADD1	RCV1(AMU)		FULL1
	RCV1(SFU)		SUB2	RCV1(AMU)		FULL1
			ADD2	RCV2(AMU)		FULL1
	RCV1(SFU)		SUB2	RCV2(AMU)		FULL2
			ADD2	RCV2(AMU)		FULL2
	RCV2(SFU)		MUL1	RCV2(AMU)		FULL2
						FULL2
	RCV2(SFU)		SUB1			
			ADD1	RCV1(AMU)		
			SUB1	RCV1(AMU)		FULL1
			ADD1	RCV1(AMU)		FULL1
	RCV1(SFU)		MUL2	RCV1(AMU)		FULL1
						FULL1
	RCV1(SFU)		SUB2			
			ADD2	RCV2(AMU)		
			SUB2	RCV2(AMU)		FULL2
			ADD2	RCV2(AMU)		FULL2
	RCV2(SFU)		MUL1	RCV2(AMU)		FULL2
						FULL2
	RCV2(SFU)		ASRA1			
			SUB1			
			ADD1	RCV1(AMU)		
			SUB1	RCV1(AMU)		FULL1
			ADD1	RCV1(AMU)		FULL1
RCVL1(SFU)			MUL2	RCV1(AMU)		FULL1
						FULL1
RCVL1(SFU)			ASRA2			
			SUB2			
			ADD2	RCV2(AMU)		
			SUB2	RCV2(AMU)		FULL2
			ADD2	RCV2(AMU)		FULL2
RCVL2(SFU)				RCV2(AMU)		FULL2
						FULL2
RCVL2(SFU)						

Figure 7: Schedule for the radix 2 intra-vector stages

AMU		
receive	ALU	MAC
		MULcdi1
		MULcdi2
RCV1(AMU)	ADDcdi1	
RCV1(AMU)	SUBcdi1	
		MULcdi3
RCV2(AMU)	ADDcdi2	

Figure 8: Schedule for the radix 2 inter-vector stages

2.4.6 Radix 4 FFT

As illustrated in the previous section, we can introduce extensive software pipelining, that would allow us for the inter-vector stages to compute an entire vectorized butterfly in only 3 clock cycles, but memory overhead prohibits this for larger FFT sizes. In order to cope with this problem we will look into radix 4 FFT algorithms. In these algorithms the number of butterflies that are computed is less, but the computation of the butterflies themselves is more complex. It is therefore reasonable to assume that the memory accesses will not form the bottleneck and that we might obtain a higher utilization of the available computational power offered by the CVP.

In a radix 4 FFT algorithm the input (for decimation in time algorithms) or output sequence (decimation in frequency) is decomposed into successively smaller sequences by iteratively deviding the sequence into four subsequences until we end up with $N/4$ four-point DFT. Therefore the length of the input sequence has to be an integer power of four.

The radix 4 decomposition is discussed in [3], section 9.6. For $N = 16$ the flow graph corresponding to the radix 4 decimation in time algorithm is depicted in figure 9. The branch multipliers within the butterflies have been omited for the sake of readability. For the same reason, only one butterfly was drawn for the second computation stage. The multipliers for any radix 4 butterfly can be found in figure 10.

A naive vectorization of the Radix 4 algorithm A first vectorization attempt of the radix 4 algorithm is similar to that of the radix 2 version. Since the vectorized algorithm processes butterflies consisting of four vectors at once, we assume $N \geq 4N_v$. Since N also has to be a power of 4, N has to be at least 64. As was the case with radix 2 decomposition, we now too distinguish between intra- and inter-vector stages. We will now also have to shuffle the vectors after each of the intra-vector stages before moving on to the next computation stage. The number of intra-vector stages depends on the vector size. For a vector size N_v the number of intra-vector stages is equal to $\lceil \mathrm{Log}_4\ N_v \rceil$. For $N_v = 8$ the shuffle pattern to be used is depicted in figure 11.

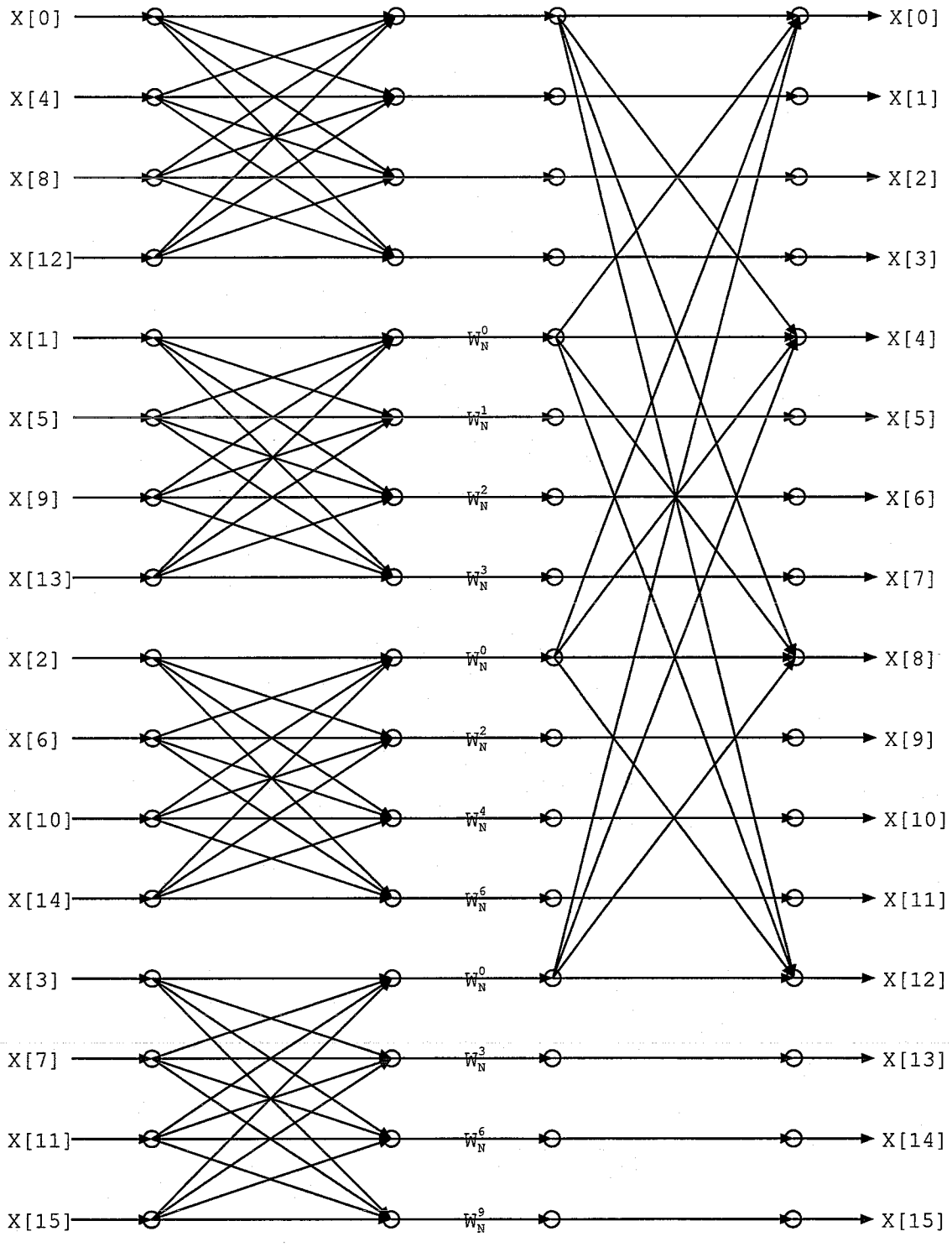

Figure 9: Flow graph for a 16 point radix 4 FFT algorithm

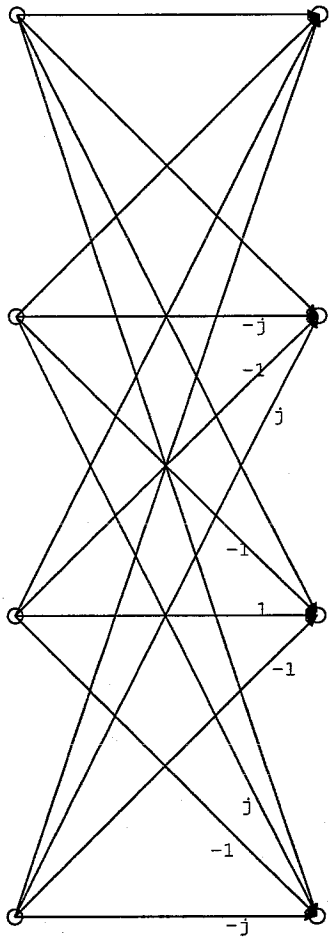

Figure 10: Butterfly for a 4-point DFT

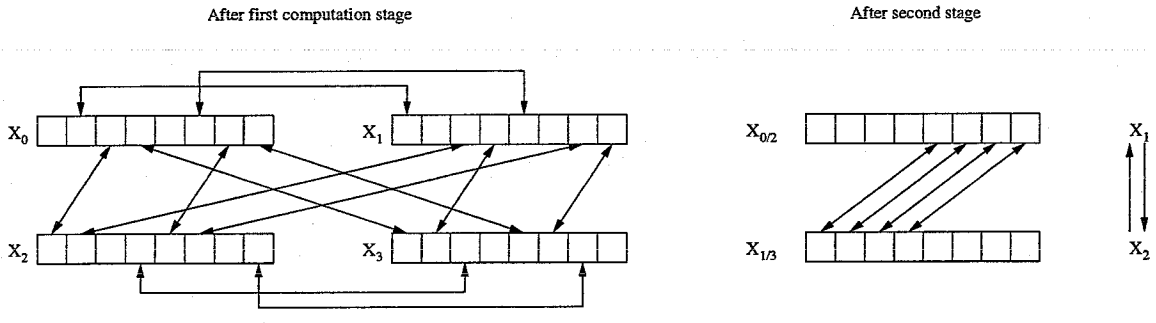

Figure 11: The shuffle pattern for radix 4 FFT

Besides the shuffling after each of the intra-vector stages, we also have to shuffle the input data in the correct order prior to computation. For radix 2 implementation, this order was bit-reversed, for radix 4 it is base-four reversed.

For the actual algorithm we make the same assumptions as for the radix 2 implementation with respect to the twiddle values. They are assumed to reside in memory, correctly ordered in vectors, prior to computation. The pseudo-code for the radix 4 algorithm is as follows.

```
[[
  doBase4Reverse()
 ;stage := 1
 ;do stage ≤ ⌈Log4 Nv⌉ →
      butterfly := 1
     ;do butterfly ≤ N/(4Nv) →
          computeButterfly()
         ;butterfly := butterfly + 1
      od
     ;shuffle()
     ;stage := stage + 1
  od
 ;do stage ≤ Log4 N →
      butterfly := 1
     ;do butterfly ≤ N/(4Nv) →
          computeButterfly()
         ;butterfly := butterfly + 1
      od
     ;stage := stage + 1
  od
]]
```

The function computeButterfly is given by

```
[[
  X0 := M[&X0]
 ;X1 := M[&X1]
 ;X2 := M[&X2]
 ;X3 := M[&X3]
 ;W1 := M[&W1]
 ;W2 := M[&W2]
 ;W3 := M[&W3]
 ;tmp1 := W1 × X1
```

$;tmp2 := W_2 \times X_2$
$;tmp3 := W_3 \times X_3$
$;tmp4 := X_1 \times [j, j, j, j, j, j, j, j]$
$;tmp5 := X_3 \times [j, j, j, j, j, j, j, j]$
$;tmp6 := X_0 + tmp2$
$;tmp7 := X_0 - tmp2$
$;M[\&X_0] := tmp6 + tmp1 + tmp3$
$;M[\&X_1] := tmp7 - tmp4 + tmp5$
$;M[\&X_2] := tmp6 - tmp1 - tmp3$
$;M[\&X_3] := tmp7 + tmp4 - tmp5$
]|

The calculation of the memory addresses $\&X_0$ through $\&X_3$ is done using the ACUs. W_0 through W_3 are the vectors containing the correct twiddle values for this stage. If we look at this implementation of the butterfly computations roughly, we see that each vectorized butterfly takes five multiplications, three of which (those with the twiddle factors) need to be scaled afterwards, five additions and five subtractions. The memory accesses (seven read and four write accesses) can be performed in parallel. This amounts to a total of 18 cycles per four-vector butterfly per stage for single precision data and 23 cycles for double precision.[3]

In the radix 2 algorithm, the total number of butterflies to be calculated is four times the number of butterflies for the radix 4 algorithm. Since the number of cycles per vectorized radix 4 butterfly is roughly four times as high as for the radix 2 algorithm, this does not seem to be a big improvement. However, if we implement the computation of the radix 4 butterfly in a different way and introduce some software pipelining, we can improve the radix 4 algorithm such that it outperforms the radix 2 version.

A more efficient implementation The radix 4 butterfly depicted in figure 10 can be computed in a different way, as depicted in figure 12. This reduces the number of multiplications by $-j$. These multiplications can also be eliminated by using the shuffle unit for this and compensating for the negated imaginary part in a later addition or subtraction.[4]

As mentioned earlier, for the inter-vector stages we will take four input vectors at once to produce four output vectors. We shall call this a vectorized butterfly, so in fact a vectorized butterfly involves N_v scalar butterflies. The pseudo code for computing a single vectorized butterfly is as follows.

|[

[3]For the radix 2 case we did not distinguish between single and double precision, because there the memory accesses were the bottleneck in computing a single butterfly.

[4]$(a + bj) * (-j) = b - aj$

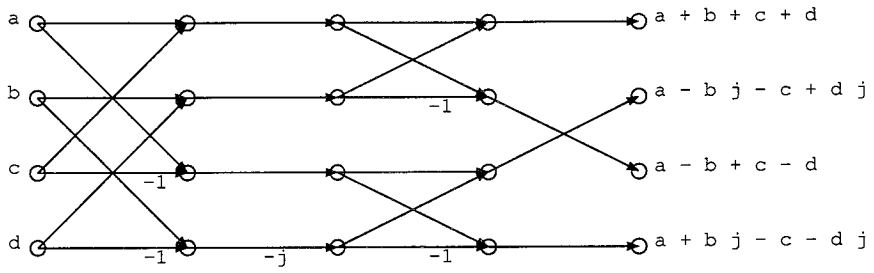

Figure 12: Alternative flow graph for the radix 4 butterfly

$$
\begin{aligned}
& X_0 := M[\&X_0] \\
& ; X_1 := M[\&X_1] \\
& ; X_2 := M[\&X_2] \\
& ; X_3 := M[\&X_3] \\
& ; W_1 := M[\&W_1] \\
& ; W_2 := M[\&W_2] \\
& ; W_3 := M[\&W_3] \\
& ; X_1 := X_1 \times W_1 \\
& ; X_2 := X_1 \times W_2 \\
& ; X_3 := X_1 \times W_3 \\
& ; tmp0 := X_1 + X_3 \\
& ; tmp1 := X_1 - X_3 \\
& ; tmp2 := X_0 + X_2 \\
& ; tmp3 := X_0 - X_2 \\
& ; tmp4 := \text{shuffle}(tmp1, conf) \\
& ; X_0 := tmp2 + tmp0 \\
& ; X_1 := tmp2 - tmp0 \\
& ; X_2 := \text{MACdi}(tmp3, tmp4, sign) \\
& ; X_3 := \text{MACdi}(tmp3, tmp4, -sign) \\
& ; M[\&X_0] := X_0 \\
& ; M[\&X_1] := X_1 \\
& ; M[\&X_2] := X_2 \\
& ; M[\&X_3] := X_3 \\
&]\!]
\end{aligned}
$$

For the scheduling of the code for the inter-vector stages we make the same assumption as in the previous section. We assume that there exists a multiply-scale instruction and that it is allowed to initiate a new ALU or MAC instruction in the next cycle. We again assume the existence of a single-cycle full shuffle, but now the current number of two configuration registers suffices. Based on these assumptions it is possible to schedule butterfly computations for the inter-vector stages such that we can compute an entire vector butterfly (four vectors input, four vectors output) in 14 clock cycles. See figure 13 for the pipelined schedule.

VMU	AMU				SFU		
send/receive	receive1	receive2	ALU	MAC	receive	config	shuffle
SENDL1							
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)			MULcdi1			
SENDL2	RCV1(VMU)			MULcdi1			
SENDL2		RCV2(VMU)		MULcdi1			
SENDL2	RCV1(AMU)	RCV2(VMU)	ADDcdi1				
SENDL2	RCV1(AMU)	RCV2(VMU)	SUBcdi1				
SENDL2	RCV1(AMU)	RCV2(VMU)	SUBcdi1		RCV1(AMU)		
SENDL2	RCV1(AMU)	RCV2(VMU)	ADDcdi1				FULL1
SENDL2	RCV1(SFU)	RCV2(VMU)	ADDcdi1				
RCVL1(AMU)		RCV2(VMU)	SUBcdi1				
RCVL1(AMU)				MACdi1			
				MACdi1			
RCVL1(AMU)				MULcdi2			
RCVL1(AMU)				MULcdi2			

Figure 13: Schedule for the radix 4 inter-vector stages

Furthermore we have two options regarding the computation of the intra-vector stages. The first approach is the one also used for the radix 2 algorithm and in the naive radix 4 implementation. In this approach we shuffle the values in such a way that we can use the same computation as for the inter-vector stages in order to compute a vectorized butterfly using four vectors as input and generating four output vectors. If we assume that the shuffling before the first stage is combined with the necessary digitreversal, we only need to perform some additional shuffling after each of the intra-vector stages. The shuffling pattern used is depicted in figure 11.

The schedule for the first stage is depicted in figure 14. From this schedule we can conclude that the first (intra-vector) stage will cost 22 cycles per four vectors. For the second stage, this approach only costs two full shuffles per vector. The schedule for the second stage is depicted in figure 15. From this schedule we can see that the second stage will cost 14 cycles per four vectors. In both schedules for the intra-vector stages, we assume that we have a sufficient number of SFU configuration registers.

We could also use a different shuffling approach for the intra-vector stages in order to reduce the required number of shuffles. For the first stage, assuming $N_v = 8$, a single vector contains the values needed to compute 2 (scalar) butterflies. Instead of shuffling values in such a way that we can compute an entire vectorized butterfly using 4 input vectors, we will now shuffle a single vector into two separate vectors in order to compute two scalar butterflies. For the inter-vector stages, we will still compute an entire vectorized butterfly at once, using four input vectors to produce four output vectors.

For this approach the algorithm for the intra-vector stages is as follows. For the first stage this code will compute 2 scalar butterflies at once. We will use the MAC instruction in order to do the necessary additions and subtractions. Since the multiplications involved in the MAC are all by 1 and -1, we can suffice with a real instead of a complex MAC.

VMU	AMU				SFU		
send/receive	receive1	receive2	ALU	MAC	receive	config	shuffle
SENDL1							
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)						
	RCV1(VMU)						
	RCV1(AMU)		ADDcdi1				
	RCV1(AMU)		SUBcdi1				
	RCV1(AMU)		SUBcdi1				
	RCV1(AMU)		ADDcdi1		RCV1(AMU)		
	RCV1(AMU)		ADDcdi1				FULL1
SENDL2	RCV1(SFU)	RCV1(AMU)	SUBcdi1		RCV1(AMU)		
SENDL2	RCV2(VMU)			MACdi1	RCV1(AMU)		FULL1
SENDL2	RCV2(VMU)			MACdi1			FULL1
SENDL2	RCV2(VMU)		SND1				
	RCV2(VMU)		SND1		RCV1(AMU)		
			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
RCVL1(SFU)			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
RCVL1(SFU)			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
RCVL1(SFU)			SND1		RCV1(AMU)		FULL1
			SND1		RCV1(AMU)		FULL1
		RCV2(AMU)	ADDcdi2		RCV1(AMU)		FULL1
		RCV2(AMU)	SUBcdi2				FULL1
RCVL1(SFU)		RCV2(AMU)	SUBcdi2				

Figure 14: Schedule for the first radix 4 intra-vector stage, first intra-vector approach

VMU	AMU				SFU		
send/receive	receive1	receive2	ALU	MAC	receive	config	shuffle
SENDL1							
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)						
SENDL1	RCV1(VMU)			MULcdi1			
SENDL2	RCV1(VMU)			MULcdi1			
SENDL2		RCV2(VMU)		MULcdi1			
SENDL2	RCV1(AMU)	RCV2(VMU)	ADDcdi1				
SENDL2	RCV1(AMU)	RCV2(VMU)	SUBcdi1				
SENDL2	RCV1(AMU)	RCV2(VMU)	SUBcdi1				
SENDL2	RCV1(AMU)	RCV2(VMU)	ADDcdi1		RCV1(AMU)		
	RCV1(AMU)	RCV2(VMU)	ADDcdi1				FULL1
SENDL2	RCV1(SFU)	RCV1(AMU)	SUBcdi1		RCV1(AMU)		
		RCV2(VMU)		MACdi1	RCV1(AMU)		FULL1
				MACdi1			FULL1
RCVL1(SFU)			SND1				
			SND1		RCV1(AMU)		
			SND1		RCV1(AMU)		FULL1
			SND1	MULcdi2	RCV1(AMU)		FULL1
RCVL1(SFU)			SND1	MULcdi2	RCV1(AMU)		FULL1
			SND1	MULcdi2	RCV1(AMU)		FULL1
RCVL1(SFU)		RCV2(AMU)	ADDcdi2		RCV1(AMU)		FULL1
		RCV2(AMU)	SUBcdi2				FULL1
RCVL1(SFU)		RCV2(AMU)	SUBcdi2				

Figure 15: Schedule for the second radix 4 intra-vector stage, first intra-vector approach

Note that the twiddle multiplications do not apply to the first stage.

$$
\begin{aligned}
&[\![\\
&\quad X := M[\&X] \\
&\;; W := M[\&W] \\
&\;; tmp0 = X \times W \\
&\;; sh\_tmp0 := \text{shuffle}(tmp, conf0) \\
&\;; tmp1 := \text{MACdi}(sh\_tmp0, tmp0, sign0) \\
&\;; sh\_tmp1 := \text{shuffle}(tmp1, conf1) \\
&\;; sh\_tmp2 := \text{shuffle}(tmp1, conf2) \\
&\;; X := \text{MACdi}(sh\_tmp1, sh\_tmp2, sign1) \\
&\;; M[\&X] := X \\
&]\!]
\end{aligned}
$$

In the algorithm presented above, the first and third shuffle take care of reordering the values such that they can be combined to form the output vector. The second shuffle is used to perform part of the mulitiplication by $-i$, since it does the swap of real and imaginary parts. The negation of the imaginary part is taken care of by the last MAC instruction.

If we assume a full shuffle is possible in a single cycle and that we have 3 instead of 2 configuration registers for the shuffle unit, we can pipeline these instructions such that it takes 10 cycles per vector. Should the MAC produce an implicit send of the result (instead of only writing back this result to the AMU register file), this could be reduced to 8 cycles per vector. See figure 16 for the pipelined schedule.

Comparing the cycle counts for the intra-vector stages, we can conclude that the first approach is superior. This is however under the assumption that we have a sufficiently large number of SFU configuration registers.

Radix 2 versus Radix 4 Based on the forementioned assumptions we can compare the cycle count estimates for radix 2 with those for radix 4. Since these assumptions do not hold for the current architecture and are therefore not supported by the current assembler and simulator it is not possible to simulate these programs in order to get exact numbers for the cycle count. In order to make a reasonable comparison we only take into account FFT sizes N that satisfy the preconditions for both radix 2 and radix 4, so N must be an integer power of 4 and $N \geq 4N_v$. We will not take into account extra overhead for ACU configuration and loop initiation, but since we omit this extra overhead for both algorithms this should do no harm to the radix 2 versus radix 4 comparison.

For the intra-vector stages, the radix 2 algorithm takes 17 cycles per vector pair for all 3 (for $N_v = 8$) intra-vector stages. For the inter-vector stages, we need 3 cycles per vector pair per inter-vector stage if N is sufficiently small. That is, if we can keep all intermediate results in the AMU register file. This holds for $N \leq 64$. For $N > 64$ we will therefore have to store the intermediate results in the vector memory, and each inter-vector stage

VMU	AMU				SFU		
send/receive	receive1	receive2	ALU	MAC	receive	config	shuffle
SENDL1							
	RCV1(VMU)						
				MULcdi1			
SENDL2							
	RCV2(VMU)		SND1				
				MULcdi2	RCV1(AMU)		
							FULL1
	RCV1(SFU)		SND2				
				MACdi1	RCV2(AMU)		
							FULL2
	RCV2(SFU)		SND1				
				MACdi2	RCV1(AMU)		
							FULL1
	RCV1(SFU)		SND2				FULL1
	RCV1(SFU)				RCV2(AMU)		
				MACdi1			FULL2
	RCV2(SFU)						FULL2
	RCV2(SFU)		SND1				
RCVL1(AMU)				MACdi2			
SENDL3							
	RCV3(VMU)		SND2				
RCVL2(AMU)				MULcdi3			

Figure 16: Schedule for the radix 4 intra-vector stages, second intra-vector approach

N	N_v	cycle count Radix 2	cycle count Radix 4
64	8	104	94
256	8	672	464
1024	8	3328	2208
4096	8	15872	10240
64	16	58	55
256	16	344	264
1024	16	1696	1232
4096	16	8064	5632

Table 5: Comparison between the cycle counts for radix 2 and radix 4 on CVP0x, not accounting extra loop initiation and acu configuration overhead cycles.

will therefore cost 5 cycles per vector pair. Adding it all together we get for radix 2 a total of $(N/16) * 17 = 68$ cycles for the intra-vector stages and $(N/16) * 3 * (\log_2 N - 3)$ cycles for the inter-vector stages for $N \leq 64$. For $N > 64$ the number of cycles for the inter-vector stages will be $(N/16) * 5 * (\log_2 N - 3)$.

For the radix 4 algorithm, we need 22 cycles per four vectors for the first intra-vector stage, whereas the second intra-vector stage costs 14 cycles per four vectors. The inter-vector stages cost 11 cycles per vectorized butterfly (four vectors in- and output). This results in $(N/32)*(22+14)$ cycles for the intra-vector stages and $(N/32)*11*(\log_4 N-2)$ cycles for the inter-vector stages. For the estimated cycle counts for several values of N, see table 5.

For $N_v = 16$, things are only slightly different. For radix 2 the number of intra-vector stages is four instead of three, so the intra-vector stages now take 6 extra cycles per two vectors and the number of cycles per vector pair per inter-vector pair remains the same. For the radix 4 algorithm, the second intra-vector stage now needs four instead of two full shuffles per vector, so this stage now costs 8 cycles extra. For the radix 4 algorithm the number of cycles per four vectors per inter-vector stage does not change either. The cycle counts for single precision data for several FFT sizes can also be found in table 5. A disadvantage of the radix 4 algorithm is the necessity that N be an integer power of 4, which limits the usability of the algorithm as compared to the radix 2 version.

From this table we can conclude that the radix 4 algorithm outperforms the radix 2 version. This is however only true if we implement the intra-vector stages using the second approach, which is only possible if we have a sufficient number of SFU configuration registers at our disposal.

2.5 Conclusions

In this section we have investigated the mapping of FFT to CVP. It turned out that by making some modifications to the current architecture, the cycle count for FFT can be reduced by a significant number. One of the major issues is the reduction of the cycle

	CVP0	CVP0x
Shuffle type	half	full
SFU configuration registers	2	16
Scatter-gather memory accessing	no	yes
Pipelined multiply-scale	no	yes

Table 6: CVP0x architecture extensions

count needed to perform the necessary bit reversal prior to the actual computation. Two means to achieve this have been adressed, the latter of which – scatter gather memory addressing – seems to be the most promising. As we shall see in the next section, this will also be a beneficial enhancement for some other applications.

Other architectural changes such as a single cycle full shuffle and a pipelined multiply-scale instruction with a one-cycle initiation interval improve the FFT performance even more. Introduction of more extensive software pipelining led to a high memory pressure for the radix 2 algorithms. Because of the more complex butterfly layout, radix 4 algorithms do not have this disadvantage. Radix 4 implementations are therefore more efficient, but only if we assume that the shuffle unit has a sufficient number of configuration registers, thereby reducing the number of cycles needed for reconfiguration.

We shall refer to the 'future version' of CVP with the forementioned extensions as CVP0x, as opposed to CVP0 for the current CVP architecture. The differences between CVP0 and CVP0x can be found in table 6. With these modifications the AMU is utilized for 100% during the both the intra- and the inter-vector stages, but still some performance gain can be achieved by allowing parallel operation of the ALU and MAC parts of the AMU. The effect of such a modification remains to be investigated.

The possible cycle counts for CVP with these modifications are presented in table 5. These numbers are the minimum number of clock cycles needed to perform the entire FFT on such an enhanced version of CVP. There is however a trade-off between code size and performance. Reducing the code size by introduction of loops requires some extra clock cycles for loop initiation and ACU reconfiguration, adding about 10-15 percent to the presented cycle counts. Accounting for this extra overhead, we get the cycle count comparison between CVP0 and CVP0x presented in table 7. For CVP0 all figures refer to radix 2 implementations, since this gives the best performance in the presence of only 2 SFU configuration registers. For CVP0x the figures for $N = 256$ and $N = 1024$ refer to radix 4 implementations, since for CVP0x the radix 4 algorithm outperforms the radix 2 version.

Table 8 contains a comparison of the performance for the 1024 point double precision FFT to that of other DSP-like architectures. As we can see, the current CVP0 is outperformed by the other architectures if we take into account the used vector sizes. The M3-DSP architecture also outperforms CVP0x, as does OnDSP is we account for the vector size, which is half that of CVP. A cause for this is the parallellism between ALU and MAC units that these architectures allow.

FFT size	cycle count CVP0	cycle count CVP0x
64	350	110
256	2000	600
512	4100	1600*
1024	8600	2500*
2048	18000	7500

Table 7: FFT cycle count estimates for CVP0 and CVP0x, 16-bit precison, radix 2 implementation, except those marked with $*$

Architecture	Clock frequency (MHz)	Cycle Count	Vector size	Reference
CVP0	200	8300	8	This document
CVP0x	200	2500	8	This document
M3-DSP	40	1024	8	[13]
VIRAM	200	7400	8	[14]
TigerSHARC	600	10800	4	[15]
OnDSP	120	4260	4	[16]

Table 8: Performance figures for a 1024 point 16 bit complex FFT

3 Digital Audio Broadcasting

Digital Audio Broadcasting (DAB) is a system designed to become a successor to the current FM radio system and is intended for audio as well as data transmission. The current FM system is among others very sensitive to so-called multipath interference. This effect occurs when an antenna picks up the direct signal along with several replicas of it that were delayed because of reflections from obstacles. Unless all these signals are 'in phase', these multiple signals reduce the strength and quality of the signal, and therefore the quality of the received audio.

In order to make the DAB system less sensitive to multipath interference, it uses Orthogonal Frequency Division Multiplex (OFDM) as its modulation technique. In OFDM the data is transmitted at a low rate on multiple carriers at the same time. The mapping of the data to the carriers is performed by an inverse FFT. For a detailed explanation on OFDM and why OFDM reduces the negative effect of multipath interference, see [5].

Although OFDM makes DAB less sensitive to multipath interference, some interference can still occur because of fading or doppler effects. These interferences usually effect only a certain set of adjacent carriers or occur in a short period of time before modulation. Therefore the data-bits are interleaved in frequency as well as time, in order to reduce sensitivity for these interferences. Furthermore, forward error correction coding is used and finally the data is transmitted using differential modulation between two consecutive OFDM symbols. The resulting system is called Coded OFDM (COFDM) and is used for DAB transmission.

For the transmitter side the DAB standard is exactly defined. The implementation of a DAB receiver is left up to the manufacturer. As CVP is intended to be used in mobile handsets, it is of course not DAB transmission, but reception we are interested in. In order to get a good understanding of the kernels involved in DAB reception we will therefore have to look into the DAB transmission standard first. DAB reception is basically a walk along the reversed path. For a feasibility analysis of DAB reception on CVP, we will take an existing implementation as a starting point and map the various kernels to CVP. Bottlenecks in implementing these kernel on the current CVP will be pointed out, as well as possible solutions.

3.1 DAB system description

In this section we shall give a brief overview of the DAB standard as defined by the European Telecomunication Standards Institute (ETSI). For more information on the standard, see [6]. The exact standard is defined in [7].

The transmitted DAB signal is built up around a transmission frame structure which is a sequential multiplex of the so-called synchonization channel, the fast information channel (FIC) and the main service channel (MSC) (see figure 17). The latter takes up the largest part of the transmission frame and it is this part that contains the actual audio data. The FIC contains control information necessary to interpret the configuration of the MSC. The synchronization channel consists of a known pattern of OFDM symbols which the

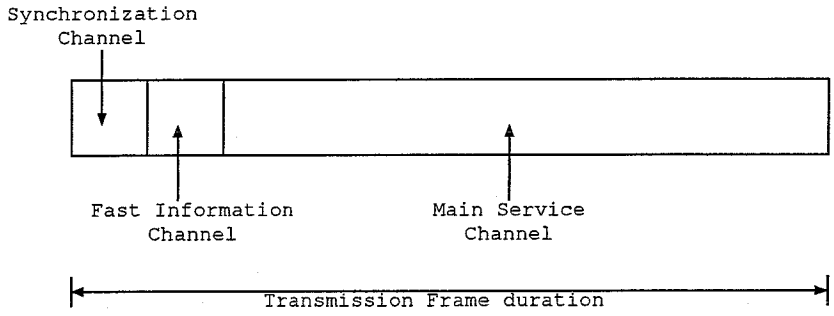

Figure 17: Structure of the DAB transmission frame

Parameter	Mode I	Mode II	Mode III	Mode IV
OFDM symbol count	76	76	153	76
number of carriers	1536	384	192	768
(I)FFT size	2048	512	256	1024
frame duration	96 ms	24 ms	24 ms	48 ms

Table 9: Parameters for the transmission modes I, II, III and IV

receiver can use for synchonization purposes. The duration and composition of the transmission frame depends on the transmission mode. Four transmission modes are defined, each having its own particular set of parameters. The transmission mode used depends on the transmission medium – satellite, cable or terrestrial – as well as the intended coverage area – local, regional or nation-wide. The parameters for each of the transmission modes can be found in table 9.

The way the actual audio data is coded and multiplexed into these transmission frames is depicted in figure 18. First of all, the audio data for each sub-channel in the DAB ensemble is divided into blocks corresponding to 24 ms of audio. Such a block is called a Logical Frame.

Convolutional Coding and Puncturing Each of the logical frames is convolutionally encoded in order to provide some redundancy that can be used for error correction during the decoding process. The logical frame data is therefore fed to a convolutional encoder as depicted in figure 19. After this convolutional coding several bits on predefined positions in the codeword are not transmitted. This is called puncturing. The standard defines several puncturing schemes, called Error Protection Levels. For higher puncturing rates the level of error protection is of course less.

Time interleaving After convolutional encoding and puncturing, the resulting codewords of each logical frame are interleaved in time. The convolutional codedword resulting from logical frame with index r shall be denoted as B_r. The output of the time

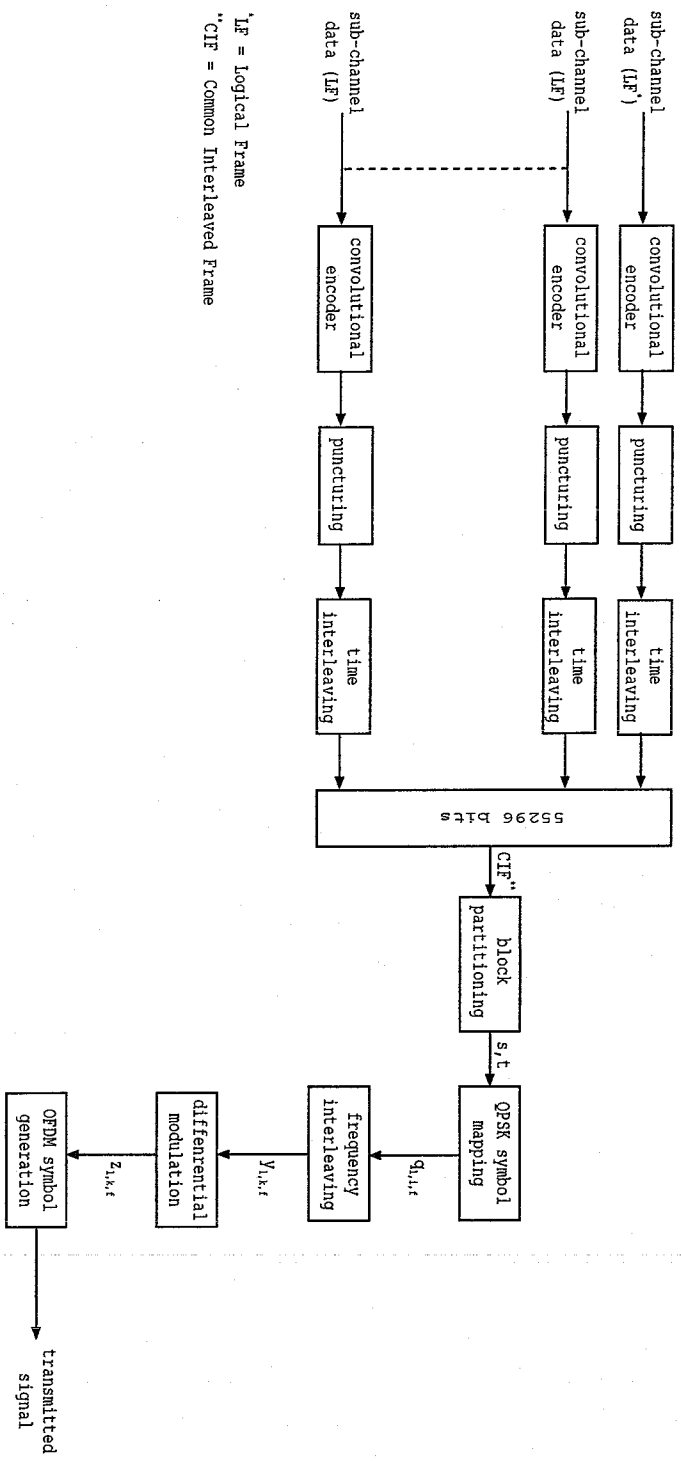

Figure 18: DAB transmitter block diagram

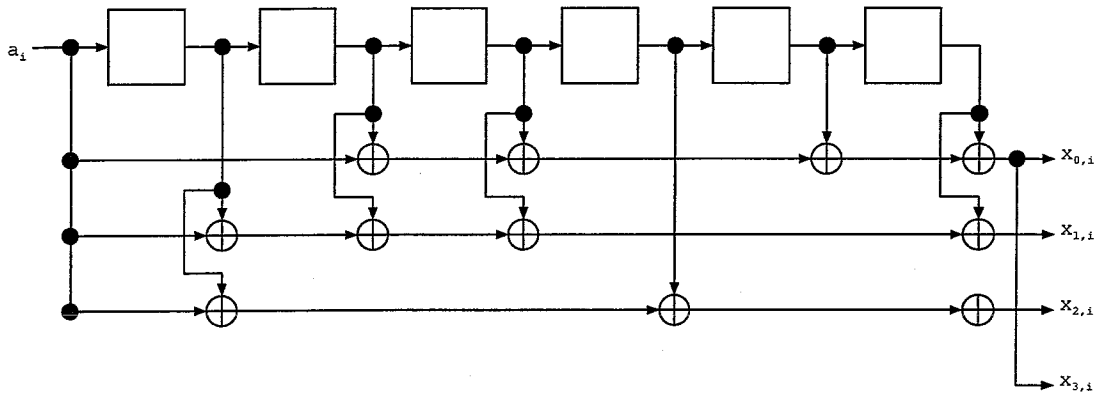

Figure 19: The convolutional encoder

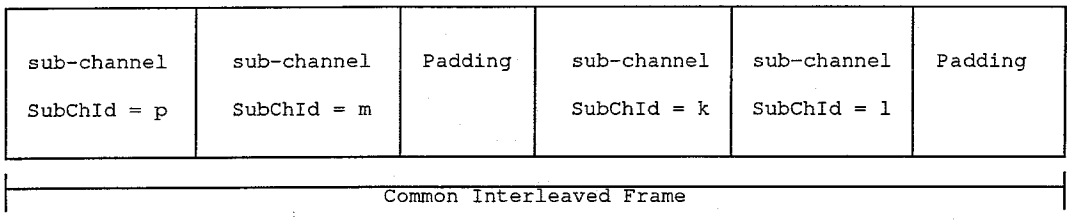

Figure 20: Example of a Common Interleaved Frame

interleaver will be a vector C_r, where B_r and C_r are related as follows

$$c_{r,i} = b_{r',i},$$

where $c_{r,i}$ denotes the element i of vector C_r.

The relation between r and r' is specified in table 10, where $R(i/16)$ denotes the remainder of the division of i by 16.

Multiplexing the subchannels After time-interleaving the various subchannel blocks are multiplexed sequentially into a so-called Common Interleaved Frame (CIF) of length 55296 bits. This is depicted in figure 20. The exact structure of the CIF depends on the number of sub-channels and the error protection scheme used per sub-channel. This information can be found in the Fast Information Channel.

Block partitioning After multiplexing the encoded and time-interleaved logical frames into common interleaved frames, these CIFs are partitioned into blocks corresponding to separate OFDM symbols. For transmission mode I four CIFs of 55296 bits each are grouped into the Main Service Channel of a transmission frame. The four CIFs are divided into 18 sub-frames of 3072 bits. These sub-frames each correspond to a single OFDM symbol in the MSC (1536 carriers per OFDM symbol, 2 bits per carrier). The first 1536 bits correspond to the bits $s^i_{l,f}$ ($0 \leq i < 1536$), the last 1536 to the bits $t^i_{l,f}$. The block

$R(i/16)$	$r'(r,i)$
0	r
1	$r-8$
2	$r-4$
3	$r-12$
4	$r-2$
5	$r-10$
6	$r-6$
7	$r-14$
8	$r-1$
9	$r-9$
10	$r-5$
11	$r-13$
12	$r-3$
13	$r-11$
14	$r-7$
15	$r-15$

Table 10: Time interleaving order

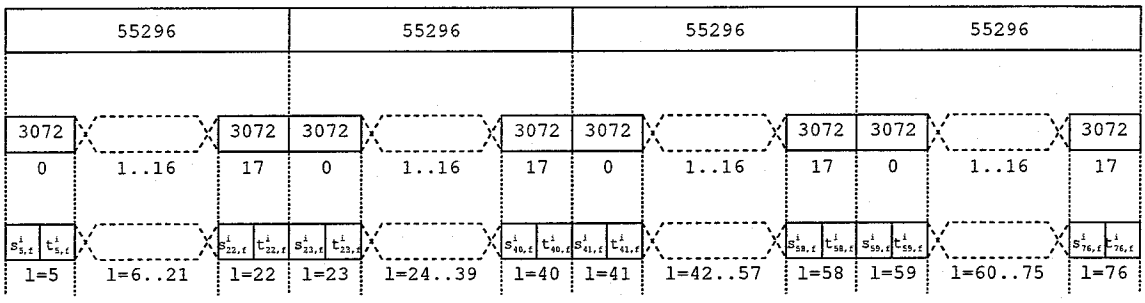

Figure 21: Block partitioning for transmission mode I

partitioning this transmission mode is depicted in figure 21. In this picture l represents the OFDM symbol index within transmission frame f.[5]

For transmission modes II, III and IV block partitioning is quite similar, except for the number of CIFs that make up a tansmission frame, which is one for modes II and III, and two for mode IV. For the block partitioning for these modes, see [7].

QPSK symbol mapping The bits $s^i_{l,f}$ and $t^i_{l,f}$ obtained from block partitioning are modulated onto symbol $q_{l,i,f}$, using quadrature phase shift keying (QPSK) according to the following relation:

$$q_{l,i,f} = \frac{1}{\sqrt{2}}(1-2s^i_{l,f}) + j(1-2t^i_{l,f})$$

[5]OFDM symbols 0 through 4 are used for the synchronization and fast information channels

Frequency interleaving Frequency interleaving is the mapping of QPSK symbol $q_{l,i,f}$ to the right carrier k in OFDM symbol $y_{l,f}$. The QPSK symbols shall therefore be re-ordered according to the following relation:

$$y_{l,k,f} = q_{l,i,f},$$

where $k = F(i)$.

For transmission mode I function F is defined as follows. Let $\Pi(i)$ be a permutation in the set of integers $i = 0, 1, \ldots, 2047$ obtained from the following congruential relation:

$$\Pi(i) = 13\Pi(i-1) + 511 \pmod{2048} \text{ and } \Pi(0) = 0$$

for $i = 1, 2, \ldots, 2047$.

Let K be the set $K = \{-768, -767, \ldots, 768\}\backslash\{0\}$. We then define K' as follows:

$$K' = \{\Pi(i) - 1024, 0 \leq i < 2048\} \cap K$$

Then $k = F(i)$ is the $(i \pmod{1536} + 1)$-th number in the set K'.

Differential modulation Differential modulation is applied to the QPSK symbols on each carrier. The value of the QPSK symbol for carrier k, to be transmitted in OFDM sysmbol l in frame f is multiplied to the value of the QPSK symbol for the same carrier in the previous OFDM symbol. In other words:

$$z_{l,k,f} = z_{l-1,k,f} \cdot y_{l,k,f}$$
$$\text{for } l = 2, 3, \ldots, L,$$

where L is the number of OFDM symbols in a transmission frame (excluding the NULL symbol), which depends on the transmission mode. The values for $l = 0$ and $l = 1$ are predefined as these are the NULL and phase reference symbol.

OFDM symbol generation Once all QPSK symbols for an OFDM symbol have been calculated, they should be combined to form the actual OFDM symbol. This is generally done by calculating an inverse FFT over the QPSK symbols for the carriers in the OFDM symbol. The thus calculated OFDM symbols are multiplexed sequentially to form the transmission frame depicted in figure 17.

3.2 DAB reception and CVP

For a feasibility analysis of DAB reception on CVP we take as a starting point the Philips SAA 3500 ASIC implementation [8]. The main part of this channel decoder implementation is depicted in figure 22. We will step through each block and see how well the kernels can be vectorized and implemented on CVP. Where necessary we will point out how changes to the current architecture could improve the possibility of implementation on CVP.

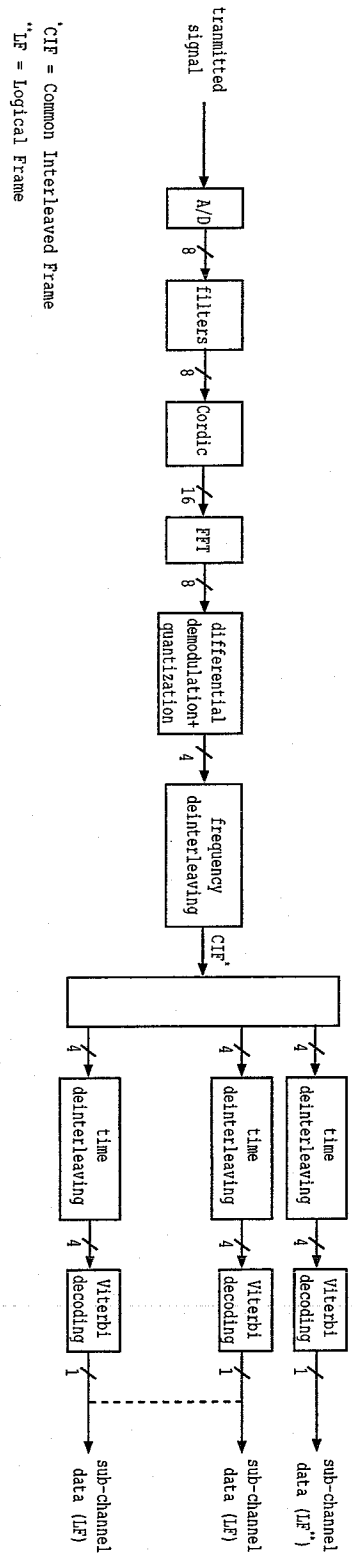

Figure 22: DAB receiver block diagram

3.2.1 Digital filters

In the SAA 3500 implementation, the transmitted signal is passed to several filters after A/D conversion. The first filter is a Complex Input Filter, which converts the sampled real signal to a complex signal, by calculating a phase-quadrature part to serve as the imaginary part for each real input sample. For more information on complex sinusoids and the conversion of real to complex sinusoids, see [9]. The second filter is a 34th order decimation filter. For more information on the exact filter specifications, see [8].

Studies have been performed on implementation of several filters – FIR, decimation and adaptive filters – on CVP. According to [10] the complex input filter can be implemented on CVP such that it can produce a vector of 16 outputs in about 20 clock cycles. The decimation filter would need about 40 cycles in order to produce a vector containing 16 outputs.

Knowing the output frequencies for both filters, 4.096 MHz for the complex input filter and 2.048 MHz for the decimation filter, we can calculate the processing load for both filters. The load for the complex input filter is 5.1 MIPS, as is the load for the decimation filter.

3.2.2 Cordic

Given an input coordinate (X, Y) and an angle θ, the Cordic algorithm efficiently computes the values (X', Y') resulting from rotating (X, Y) over the angle θ, as depicted in figure 23. By splitting up the rotation over the desired angle θ into multiple rotations over smaller well defined angles, the desired rotation can be calculated by using only shifts and additions. For more information, see for instance [11]. In the SAA 3500 the cordic algorithm is used to convert the signal to baseband. See [8] for more information. The algorithm is as follows.

$[\![$
$n := 0$

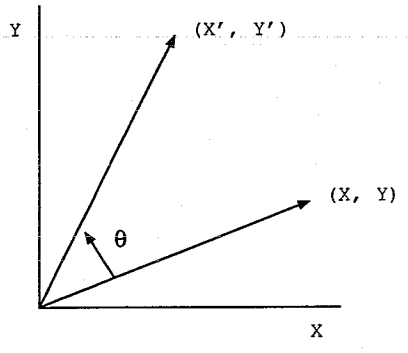

Figure 23: The rotation performed by the Cordic algorithm

```
;do |Z(n)| < ε →
    if (Z(n) ≥ 0) →
      X(n + 1) := X(n) − (Y(n)/2^n)
     ;Y(n + 1) := Y(n) + (X(n)/2^n)
     ;Z(n + 1) := Z(n) − arctan(1/2^n)
    [] (Z(n) < 0) →
      X(n + 1) := X(n) + (Y(n)/2^n)
     ;Y(n + 1) := Y(n) − (X(n)/2^n)
     ;Z(n + 1) := Z(n) + arctan(1/2^n)
    fi
   ;n := n + 1
 od
]|
```

For input values $X_0 = x$, $Y_0 = y$ and $Z = \theta$, the algorithm converges to $X_\infty = P_c(x \cos\theta - y \sin\theta)$, $Y_\infty = P_c(y \cos\theta + x \sin\theta)$ and $Z_\infty = 0$, where $P_c \approx 1.64677$.

In the algorithm described above, the number of loop iterations depends on the value of $Z(n)$. In practice however, a fixed number of iterations is used. The cordic mixer in the SAA 3500 performs 11 of these iterations. Since division by 2^n can be implemented using a right shift and the values for $\arctan\frac{1}{2^n}$ can be precomputed and stored in a lookup table, the algorithm only needs to perform additions (or subtractions) and shifts.

In the CVP implementation of the algorithm we perform the same computation on multiple X, Y and Z inputs at once. Since we will be using 16 bit integer vectors for X, Y and Z we can compute 16 values in parallel. The problem with converting the algorithm into a CVP implementation, is that the CVP ISA does not support guarded statements. Therefore we do not have any means to implement an 'if' statement. To eliminate this 'if' statement the algorithm can be rewritten as follows.

```
|[
  n := 0
 ;do |Z(n)| < ε →
     X(n + 1) := X(n) − sgn(Z(n)) ∗ (Y(n)/2^n)
    ;Y(n + 1) := Y(n) + sgn(Z(n)) ∗ (X(n)/2^n)
    ;Z(n + 1) := Z(n) − sgn(Z(n)) ∗ ArcTan(1/2^n)
  od
]|
```

In this algorithm the function sgn(x) equals 1 for $x \geq 0$ and -1 for $x < 0$. Currently the CVP does not support such a sgn instruction, so this is implemented using an arithmetic shift right to obtain the sign bit followed by a bitwise or with the proper bitmask

The algorithm was implemented in CVP-C. It consists of an outer loop which iterates over the number of input vectors (that is the number of input values divided by 16). Within this loop the output is calculated for one X, Y and Z input vector triple. The CVP-C algorithm contains sequential program instructions only. Scheduling of code for the different functional units is taken care of by the compiler. Furthermore, the C program contains regular variables instead of registers, as the compiler is responsible for the register allocation. At the current point in the development of the compiler however, every CVP-C instruction is still a function call directly corresponding to an assembly instruction for a specific functional unit. The meaning of the AMU instructions used in this algorithm can therefore be found in table 1.

As mentioned before the algorithm performs done 11 iterations per input triple X, Y and Z. However, since the value by which the vectors X and Y need to be shifted depends on the loop counter, this loop had to be unrolled since there is no means to use this loop counter in an arithmetic shift right instruction. The code for one of those iterations is given below. In this code, where it says `amuv_ASRdi_<<n>>`, the `<<n>>` is to be replaced by the iteration number.

```
rindex_1:   Cvp_acuAddMod(r_addr, r, arctan, r,
                                  dword, line);
rd1:  read_dw(arctan, atan, r_addr);

amuv_ASRdi_15(sgnBit, Z);
amuv_ORi(sign, sgnBit, bitmask);
amuv_NEGdi(negSign, sign);

amuv_ASRdi_<<n>>(Xshift, X);

amuv_ASRdi_<<n>>(Yshift, Y);

amuv_MACdi_0(temp0, X, negSign, Yshift);

amuv_MACdi_1(temp0, Y, sign, Xshift);

amuv_MACdi_2(temp0, Z, negSign, atan);
```

All but the first few memory accesses can be scheduled in parallel with the AMU operations and the `amuv_SND` instructions are nothing more than a copy from one AMU register to another and can therefore be omitted in the actual assembly code. One iteration for one set of input vectors X, Y and Z, will therefore cost 8 clock cycles. For each triple of input vectors we perform 11 of these iterations plus 3 memory accesses at the loop boundaries. Thus the cycle count for an N point cordic adds up to $91 * (N/16)$ plus a few cycles for initialization.

The cycle count could be improved if the ISA supported some guarded addition or subtraction instruction. This instruction would take three arguments, the first two being the

values to be added or subtracted and a third one indicating whether to perform an addition or subtraction. Using the same notation as in [1], the instruction would look like this: $\forall_{p:0\leq p<P}\{snd[p] = (src3[p] \geq 0)\ ?\ src1[p] + src2[p]\ :\ src1[p] - src2[p]\}$.

Using this instruction we would have the following implementation for one iteration step.

```
rindex_1:   Cvp_acuAddMod(r_addr, r, arctan, r,
                                    dword, line);
rd1:  read_dw(arctan, atan, r_addr);

amuv_ASRdi_<<n>>(Xshift, X);
amuv_ASRdi_<<n>>(Yshift, Y);

amuv_NEGdi(negYshift, Yshift);
amuv_NEGdi(negAtan, atan);

amuv_CondAddSub(X, X, negYshift, Z);
amuv_CondAddSub(Y, Y, Xshift, Z);
amuv_CondAddSub(Z, Z, negAtan, Z);
```

Since the memory accesses can be scheduled in parallel with the AMU instructions, this implementation will cost 7 cycles per iteration step. If the conditional add/subtract instruction worked exactly the other way around, that is, if it would perform an addition if $src3[p] < 0$ and a subtraction if $src3[p] \geq 0$, then we could save another cycle because we would then need only one instead of two NEG instructions. Having both these alternatives would eliminate both NEG instructions and would therefore result in 5 cycles per iteration step.

The input frequency of 4.096 MHz, combined with the calculated cycle counts results in a processor load of 23.2 MIPS for the current CVP. With both conditional add/subtract alternatives present, the processor load would drop to 14.8 MIPS.

3.2.3 FFT

After conversion to baseband a Fast Fourier Transform is applied to every OFDM symbol in order to retrieve the values for each carrier. The number of carriers and therefore also the FFT size N depends on the used transmission mode.

Cycle count estimates for FFT on the current CVP architecture as well as for possible future CVP0x have been given in the previous chapter. Table 11 gives estimates for the cycle counts for FFT for the various transmission modes for both CVP0 and CVP0x. Since the SAA 3500 uses 12 bit precision for the intermediate results, the estimates are based on double precision complex data ($N_v = 8$).

Based on the derived cycle count estimates and the OFDM symbol durations which are defined by the standard [7], we can compute the processing load for FFT for both CVP0 and CVP0x. These load figures can be found in table 12.

Transmission mode	FFT size	cycle count CVP0	cycle count CVP0x
I	2048	18000	7500
II	512	4000	1600
III	256	2000	600
IV	1024	8300	2500

Table 11: FFT cycle count estimates for the various DAB transmission modes

Transmission mode	FFT load CVP0 (MIPS)	FFT load CVP0x (MIPS)
I	10.4	6.0
II	12.8	5.1
III	12.8	3.8
IV	13.3	4.0

Table 12: FFT processing load for the various DAB transmission modes

3.2.4 Differential demodulation

After applying an FFT to each of the OFDM symbols to obtain the individual carriers, differential demodulation is applied to each carrier in order to obtain the QPSK symbol holding the data for that carrier, by multiplying the FFT result with the complex conjugate of the previous FFT result. On the current CVP architecture this will cost 2 clock cycles per vector.

Since we are only interested in the values for the $K = \frac{3}{4}N$ transmitted carriers, we do not need to compute all N values. This gives rise to a processing load of 0.15 MIPS for both CVP0 and CVP0x.

3.2.5 Quantization

Once we have obtained the QPSK symbol for a specific carrier, we want to distill the two databits it conveys. In order to facilitate for soft-decision Viterbi decoding we want to pack each databit in a 4-bit symbol. This symbol has a value ranging from 0000 to 1111, where 0000 indicates the highest probability that the databit was a 0 and 1111 indicates that the databit was most likely a 1. The value for the first symbol is to be extracted from the 8-bit real part of the complex QPSK symbol, the second symbol will be extracted from the imaginary part.

Figure 24 graphically depicts the mapping from the complex QPSK symbol to the two data symbols. Since for the QPSK symbol both the real and imaginary part have a 8-bit precision, the values for both parts lie between -128 and 127. The values for the 4-bit symbols lie between 0 and 15. The mapping is fairly straightforward. First we shift both the real and the imaginary part of the QPSK symbol to the right by 4 bits and then we subtract the result from 7. That way -128 is mapped to 1111 and 127 to 0000 which is exactly what we want.

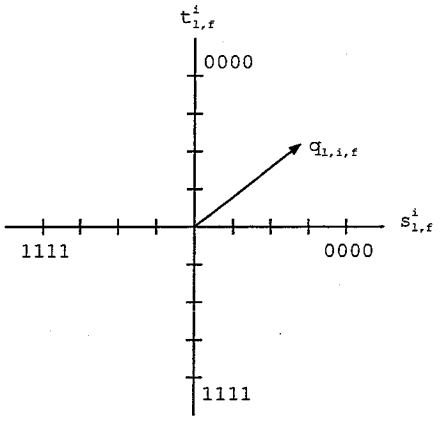

Figure 24: Quantization of the complex QPSK symbol $q_{l,i,f}$ to two 4-bit symbols $s^i_{l,f}$ and $t^i_{l,f}$

This quantization costs 2 clock cycles per vector. For K transmitted carriers per OFDM symbol we have $\frac{K}{16}$ vectors. Since we have to finish quantization within the OFDM symbol time, this gives rise to a processor load of 0.15 MIPS.

3.2.6 Frequency deinterleaving

After quantization we have all the data symbols that are conveyed in a single OFDM symbol during transmission. What we have to do now is reorganize these symbols in the correct order to compensate for block partitioning and frequency interleaving that was applied before transmission. The frequency interleaving rules were discussed in the previous section.

Since we need to deinterleave 4-bit symbols, this is only possible on the current CVP architecture if those symbols are stored in 8-bit memory locations, so that we can deinterleave on an 8-bit word basis. Calculation of the correct memory location for the next symbol to be read is not possible, since it not only involves arithmetic operations, but there is also the need for a selection on the computed value. Since CVP does not offer instructions for this selection, the memory addresses have to be precomputed and stored in memory. The number of memory locations needed to store these addresses is twice the number of carriers per OFDM symbol for the used transmission mode.

If we wish to read a value from a memory location, for which the address is stored in memory itself, we need to have a means to read the address from memory and put that value into the offset register of an ACU. The current ISA however only supports configuration of the offset registers by means of immediate values. The only means available to configure an ACU using values located in memory is by using the LDACUS instruction, which reconfigures all registers for two ACUs at once, based on a configuration vector located in memory. This would mean that for every symbol we need to read the correct offset from memory, write that value to the correct location in the ACU configuration

vector, use that vector to reconfigure the ACU and then finally read the correct symbol to be deinterleaved and store it again. This would add up to 5 instructions for deinterleaving one symbol. If the ISA would allow for the configuration of an ACU offset register with a value in memory, pointed to by another ACU, only 3 instructions would be needed per symbol.

For frequency deinterleaving scatter-gather memory addressing, as suggested in section 2.4.3 is no solution, since the addressing pattern, defined by the intersection of Π with the set of carriers K (see page 39) is very irregular. Typically there will therefore be multiple bank conflicts per line access. However, if we do wish to use scatter-gather for optimizing frequency deinterleaving, a possible approach would be to use the function Π for deinterleaving instead the intersection of Π and K, because in that case scatter-gather accessing can be used without any bank conflicts. This way the desired data symbols are rearranged in the correct order, but we read a total of N data symbols instead of the desired $\frac{3}{4}N$. What remains to be done is 'filtering out' the unwanted symbols. Solutions to this problem remain to be investigated, for now we will not explore on this any further.

The number of data symbols to be deinterleaved is twice the number of carriers per OFDM symbol. These data symbols need to be deinterleaved within the OFDM symbol duration. For the current architecture the processing load will be 12.3 MIPS. Would the ISA support configuration of an ACU offset register with a value located in memory, the load would be 7.4 MIPS.

3.2.7 Time deinterleaving

Time interleaving is performed on the basis of a so-called logical frame. As mentioned in the previous section, this is the data burst contributing to a sub-channel for the duration of 24 ms. Multiple logical frames are multiplexed in a Common Interleaved Frame. Since there is no interleaving between separate sub-channels in the same CIF, we do not need to deinterleave all sub-channels. We can select the sub-channel we are intereseted in and perform time deinterleaving on that sub-channel only. Of course this is more efficient in both processing load and memory usage. The only disadvantage is that instantanious switching to another sub-channel by the user is not possible, since the deinterleaver memory has to be filled for the new sub-channel before time deinterleaving can start.

As opposed to frequency deinterleaving, time deinterleaving is much more regular in nature. All sixteen symbols that have to end up subsequently, are interleaved over 16 logical frames, but reside at subsequent offsets in those frames. As a result, when using scatter-gather memory accessing, there will be no bank conflicts if only we make sure that for every memory line, there will be only one symbol per bank initially (every 4-bit symbol will then take up 16 bits of memory space) and make sure that every subsequent logical frame starts at a new memory line. Generating a new offset vector from the current one is achieved by one single addition, which can be performed in parallel with the two memory accesses per vector (one read and one write access). This way we can deinterleave 16 data symbols in only 2 clock cycles.

For the current CVP, we have to apply the same scheme as for frequency deinterleaving,

	CVP0	CVP0x
Conditional ADD/SUB	no	yes
Indirect ACU offset load	no	yes
Specific Viterbi instructions [12]	no	yes

Table 13: Extra CVP0x extensions for DAB reception kernels

which adds up to 5 clock cylces per data symbol. To calculate the processing load we need to know the number of data symbols that make up one logical frame. Since the standard defines several audio bitrates and error protection schemes, this number can vary. The maximum number of data symbols per logical frame is 26616. This results in a maximum processing load for time deinterleaving of 5.5 MIPS for CVP0 and 0.14 MIPS for CVP0x.

3.2.8 Viterbi decoding

After time deinterleaving all data symbols are in the correct order. What remains to be done is retrieving the original data bit stream from the stream of data symbols. The data symbols are decoded using the Viterbi algorithm. Studies have been performed on vectorizing this algorithm and implementing this on CVP. For more information on the algorithm itself and the mapping to CVP, see [12].

According to [12], Viterbi decoding for the specified convolutional encoding scheme will cost about 10 clock cycles per data symbol. This does however assume some extensions to the current CVP architecture, so these numbers apply to CVP0x only. Again the number of symbols to be processed per second depends on the audio bitrate and error protection scheme, but for the maximum number of data symbols per logical frame (26616), the processing load for Viterbi decoding is 11.1 MIPS.

3.3 Overall DAB performance on CVP

3.3.1 Processing load

In this chapter we have covered the main kernels involved in the implementation of a DAB receiver on CVP. For every kernel we have computed estimates for the processor load on CVP0 and the extended version CVP0x.

Table 6 contains the architectural differences between CVP0 and CVP0x that were suggested in order to improve FFT performance. In order to improve the performance for the other DAB reception kernels as well, some additional modifications were suggested. These can be found in table 13.

The load figures for each kernel can be found in table 14. For the digital filter kernels the figures were based on studies performed by others. These do not include figures for CVP0x. Since it is beyond the scope of this document to try and compute possible figures for CVP0x, the number for CVP0 were used for these kernels. Since the figures for Viterbi

Kernel	processor load CVP0 (MIPS)	Processor load CVP0x (MIPS)
Digital filters	10.2	10.2
Cordic	23.2	14.8
FFT	12.3	4.7
Differential demodulation	0.15	0.15
Quantization	0.15	0.15
Frequency deinterleaving	12.3	7.4
Time deinterleaving	5.5	0.14
Viterbi decoding	-	11.1
Total	63.7 (excl. Viterbi)	48.5 (incl. Viterbi)

Table 14: Processing load for the various DAB reception kernels

as derived from [12] assume some extensions to the current CVP, no figures are available for CVP0. For FFT, the avarage of the processor loads for the various transmission modes was used. For frequency deinterleaving, the possibility to load a value from the AMU or VMU to an ACU offset register was also assumed to be a CVP0x extension.

For both time deinterleaving and Viterbi decoding the processing load depends on the number of symbols per logical frame. The figures in table 14 pertain to the maximum number and should therefore be considered as an upper bound.

Since the CVP is intended to operate at a clock frequency of 300 MHz, DAB reception should be feasible on the current architecture (except for Viterbi, which makes use of some specialized instructions that are not available in CVP0). The extensions to the current architecture resulting in the CVP0x architecture, greatly improve the performance of some of the DAB reception kernels, and include the possibility to run Viterbi on CVP as well.

3.3.2 Memory usage

When analysing the feasibility of DAB reception on CVP, processing load is not the only thing we need to worry about. The amount of memory needed to perform the several kernels is equally important.

The first few kernels – from the digital filters until frequency deinterleaving – can be performed using the same block size. Since the FFT size for transmission mode I is 2048, we take the block size to be 2048 samples (except for the input to the digital filters, which is twice the size of its output). For the other transmission modes, block sizes of 256, 512 or 1024 samples suffice, and these will therefore require less memory. In order to calculate the maximum memory requirement, we will consider transmission mode I in detail. Since some of the kernels can be performed in place, these can use the same memory block for both input and output. For these first few kernels the code combining them would be as follows.

```
[[
  var B0 : Block(4096)
  var B1, B2, B3, old_FFT_result : Block(2048)
  do for every block →
      B0 := Buffer from A/D convertor
    ; B1 := Filters(B0)
    ; B1 := Cordic(B1)
    ; B2 := FFT(B1)
    ; B3 := Differential Demodulation(B2, old_FFT_result)
    ; old_FFT_result := B2
    ; B3 := Quantization(B3)
    ; B4 := Frequency Deinterleaving(B3)
  od
]]
```

The above program fragment indicates that we will need four 2048 sample blocks and one 4096 sample block for storing inputs and outputs for these kernels. This amounts to a total of $(5 * 2048 + 4096)/8 = 1792$ memory lines. Furthermore the kernels need some additonal memory for storage of intermediate results and program constants. The digital filters need about 10 lines for storage of the multiplication factor constants. The cordic algorithm needs two extra memory lines for the ArcTan constants. FFT needs an additional 128 memory lines for the storage of the twiddle factors. Finally the interleaving pattern for frequency interleaving has to be stored in memory, which costs an additional 256 lines of memory. The total memory usage for these kernels thus amounts to 2188 lines.

The next kernel to do its processing is time interleaving. Time deinterleaving requires a vast amount of memory. In order to deinterleave one logical frame, we have to buffer 16 time interleaved frames. A single local frame can consist of a maximum number of 26616 4-bit data symbols. On CVP this would correspond to 6654 memory lines. In order to be able to use scatter-gather for time deinterleaving a memory line can only contain 16 4-bit symbols. In that case the required memory would be 26616 memory lines.

The memory needed to perform Viterbi decoding equals the memory needed to store both the input and output buffer plus an additional 70 memory lines for internal computation. The block size for Viterbi decoding is equal to a logical frame. The input buffer should therefore have a size that can contain 26616 4-bit symbols. The size of the output buffer can be four times less, since every 4-bit input symbol corresponds to only one output bit. The total memory needed for Viterbi decoding then amounts to 590 memory lines.

For transmission mode I, the total memory that is needed for all kernels except for time deinterleaving equals 2778 lines, which is well within the current memory size of 4096 lines. The memory usage for the other transmission modes (excluding time interleaving) can be found in table 15. In order to be able to perform time deinterleaving on CVP as well, the amount of memory has to be increased by a significant amount.

Transmission mode	Required memory lines
I	2778
II	1146
III	874
IV	1690

Table 15: Memory usage for DAB reception kernels, excluding time deinterleaving

4 Conclusions

4.1 Achievements

We derived several vectorized FFT algorithms and implemented a radix 2 algorithm for both 32 and 64 input points using the current CVP toolset. Based on these implementations we have pointed out some bottlenecks in fitting FFT to the current CVP architecture. An inefficiency that has major impact on FFT performance is that the current architecture lacks a means to efficiently implement the bit reversal that has to be performed before the actual FFT computation. Currently the only means for the bit-reversed reordering is the shuffle unit and using the SFU for bit reversal turns out to very inefficient.

Suggestions have been made to incorporate a bit-reversed ACU addressing mode or scatter-gather memory accessing in order to reduce the number of clock cycles spended on bit reversal. Both alternatives make it possible to perform bit reversal almost completely in parallell with the actual FFT computation.

Another performance bottleneck is caused by the shuffles that are required in between the so-called intra-vector stages. The number of clock cycles needed to perform these shuffles can be reduced by a significant amount by introducing a full shuffle instruction, thereby eliminating the need to spend two half shuffles in order to achieve the desired shuffle result. Furthermore, additional SFU configuration registers would reduce the required number of reconfigurations and thus the number of cycles required.

The actual FFT butterfly computations can be sped up by introduction of a pipelined multiply-scale instruction and allowing the initiation of another ALU operation during the clock cycle immediately following the multiply-scale initiation. This would allow for more extensive software pipelining. Unfortunately this extra pipelining leads to a high memory pressure for the radix 2 algorithm. Radix 4 algorithms lack this disadvantage because of the more complex butterfly structure. A radix 4 algorithm on the other hand requires a more complex shuffling scheme inbetween the intra-vector stages, and the radix 4 algorithm is therefore more efficient only in the presence of a sufficient number of SFU configuration registers. For the current CVP architecture which provides for two of these configuration registers, the radix 2 algorithm outperforms the radix 4 version.

Besides FFT, we have also analysed the feasibility of implementing the other kernels involved in DAB baseband processing on CVP. For these kernels we have estimated the processing load for the current CVP architecture, as well as the possible performance on an extended CVP version. Except for frequency interleaving, all of the investigated kernels are well vectorizable. With respect to processing load, baseband processing for DAB reception seems feasible on CVP, the total processing load being roughly 64 MIPS. Memory usage however seems to be a problem, since especially time deinterleaving requires much more memory than is available in the current architecture.

4.2 Suggestions for future study

As indicated earlier, because of the more complex butterfly structure for radix 4 FFT algorithms, these algorithms give rise to a lower memory bandwidth. Therefore more extensive pipelining can be applied to achieve a higher performance on the actual butterfly computations than for the radix 2 algorithms. Unfortunately these radix 4 algorithms require a more complex shuffling pattern inbetween the intra-vector stages. It would therefore be worthwile to investigate mixed radix algorithms, for example using radix 2 for the intra-vector stages and radix 4 for the latter inter-vector stages.

An architectural extension that is currently under investigation in the SW-Modem project is the incorporation of an ALU select instruction that takes two vectors as input and produces one output vector. The output is an interleaving of the two input vectors at a partly configurable segment granularity. This instruction could very well be used for the vector rearrangements inbetween the intra-vector stages. The impact of such a select statement on the overall FFT performance is worth investigating.

Several VLIW based architectures allow for parallel operation of an ALU and MAC unit. For CVP these are combined into the AMU functional unit. Allowing these two parts of the AMU to operate in parallel would enhance the FFT performance. The performance gain achieved by this modification is another subject for future study.

With respect to DAB it would be interesting to investigate the possibility to use scatter-gather memory accessing in order to vectorize frequency deinterleaving, as mentioned in section 3.2.6. Another issue that has not been addressed with respect to DAB reception is the communication between CVP and the host processor.

References

[1] K. van Berkel, P. Meuwissen, S. Weijs, R. Wubben, N. Engin; *Obelix: a Co Vector-Processor for 3rd Generation Moblie Communication*, Nat.Lab. Technical Note 2001/031, Philips Research Laboratories, September 2002.

[2] K. van Berkel, P. Meuwissen, N. Engin, S. Balakrishnan; *CVP: A programmable co vector processor for 3G mobile baseband processing*, to appear in the proceedings of the World Wireless Congress 2003.

[3] A.V. Oppenheim, R.W. Shafer; *Discrete-time signal processing*, Prentice-Hall International Inc. 1989, pp. 581-622.

[4] R. Thomas; *An architectural performance study of the fast fourier transform on Vector IRAM*, Report No. UCB/CSD-00-1106, University of California, Berkeley, June 2000

[5] R. van Nee, R. Prasad; *OFDM for wireless multimedia communications*, Artech House, 2000.

[6] P.W.F. Gruijters; *Architecture study for a DAB channel decoder*, Design Automation Section, Eindhoven University of Technology, Digital VLSI Group, Philips Research Laboratories, August 1995.

[7] European Telecomunication Standards Institute; *Radio systems; Digital Audio Broadcasting (DAB) to mobile, portable and fixed receivers*, ETSI standard EN 300 401 v1.3.3, May 2001.

[8] F. van der Laar; *DAB channel decoder IC specification documents*, Philips ASA lab, December 1996.

[9] J.O. Smith; *Complex sinusoids*, Part of online publication *The mathematics of the discrete fourier transform*, http://www-ccrma.stanford.edu/~jos/mdft/Complex_Sinusoids.html

[10] B.L. Dang; *Vectorization of Digital Filters*, Master's thesis, Faculty of Electrical Engineering, Mathematics and Computer Science, Delft University of Technology, 2003.

[11] R. Herveille; *Cordic Core Specification*, Rev. 0.4, December 18, 2001, http://www.opencores.org/cgi-bin/cvsget.cgi/cordic/documentation/cordic.pdf

[12] N. Engin; *Vectorization of Viterbi Algorithm*, Nat.Lab Technical Note PR-TN-2003/00596, Philips Research Laboratories, August 2003.

[13] T. Richter, W. Drescher, F. Engel, S. Kobayashi, V. Nikolajevic, M. Weiss, G. Fettweis; *A platform-based highly parallel digital signal processor*, Proc. CICC'2001, San Diego, 2001.

[14] R. Thomas, K. Yelick; *Efficient FFTs on IRAM*, Computer Science Division, University of California, Berkeley.

[15] Analog Devices; *TigerSHARC Processor Benchmarks*, http://www.analog.com/processors/processors/tigersharc/benchmarks.html

[16] J. Kneip, M. Weiss, W. Drescher, V. Aue, J. Strobel, T. Oberthür, M. Bolle, G. Fettweis; *Single chip programmable baseband ASSP for 5 GHz wireless LAN applications*, IEICE Trans. Electron., Vol. E85-C, No. 2, pp. 359-367, February 2002.