DS-CDMA Procedures with the Cell Broadband Engine

9TH SEMESTER PROJECT, AAU
APPLIED SIGNAL PROCESSING
AND IMPLEMENTATION (ASPI)

Group 942
Peter August Simonsen
Jes Toft Kristensen

AALBORG UNIVERSITET

Institute for Electronic Systems
Fredrik Bajers Vej 7B
Telefon 96 35 98 36
Fax 98 15 36 62
http://www.esn.aau.dk

Title:
DS-CDMA Procedures
with the Cell Broadband Engine

Theme:
Non-linear DSP Methods and Real-time Architectures

Project period:
P9, fall semester 2007

Project group:
ASPI 07gr942

Members:
Peter August Simonsen
peter@augusts.dk
Jes Toft Kristensen
jes@buskefjomp.dk

Supervisors:
Yannick Le Moullec (AAU)
Andreas Popp (AAU)

Kristian Sørensen
(Rohde & Schwartz
Technology Center A/S)

Copies: 5

Pages in report: 86

Appendices: 1 CD-ROM

Printed December 20, 2007

Abstract:

This 9th semester project of the "Applied Signal Processing and Implementation" specialization at Aalborg University is an investigation of the challenges in efficient programming of a DS-CDMA descrambling and despreading application for the Cell Broadband Engine (CBE) architecture, aimed at functioning as a wireless communications base station.
Initially a signal model is established and the CBE architecture is analyzed. The CBE architecture is of special interest as it is a heterogeneous multicore processor, which offers concurrent processing power in the form of 6 to 8 offload processors and the capability of vector processing of data (SIMD).
The communication method is examined and a signal model which is exploitable by the CBE is proposed.
To measure the potentials of the CBE platform, two experiments are conducted prior to the implementation of the demodulation. These experiments concern utilization of the internal bus of the CBE, used for transfers between processors, and a test of the CBE's performance in multiplication of data. Based on the signal model further partitioning and extraction of parallelism is conducted, which is later implemented on the CBE for testing.
The tests shows that the implementation is able to demodulate a 10 ms communication burst in 84.2 ms with a utilization of 10.3% of the maximal 76.8 GFLOPS which is the theoretical performance of the CBE. The low utilization is due to the initial problem partitioning which fails to fully exploit the CBE offload processor pipeline. A different partition is proposed but not examined further.

AALBORG UNIVERSITET

Institut for Elektroniske Systemer
Fredrik Bajers Vej 7B
Telefon 96 35 98 36
Fax 98 15 36 62
http://www.esn.aau.dk

Titel:
DS-CDMA Procedures
with the Cell Broadband Engine

Tema:
Non-linear DSP Methods and Real-time Architectures

Projekt periode:
P9, efterårs semester 2007

Projekt gruppe:
ASPI 07gr942

Medlemmer:
Peter August Simonsen
peter@augusts.dk
Jes Toft Kristensen
jes@buskefjomp.dk

Vejledere:
Yannick Le Moullec (AAU)
Andreas Popp (AAU)

Kristian Sørensen
(Rohde & Schwartz
Technology Center A/S)

Kopier: 5

Sider i rapport: 86

Antal bilag: 1 CD-ROM

Printet December 20, 2007

Synopsis:

Dette 9. semester projekt fra specialiseringen i "Anvendt Signalbehandling og Implementering" ved Aalborg Universitet omhandler undersøgelse af udfordringer ved effektiv programmering af en DS-CDMA applikation (descrambling of despreading) til Cell Broadband Engine (CBE) arkitekturen.
Første del af projektet omhandler opstilling af en signalmodel for applikationen og en analyse af CBE arkitekturen. CBE er interessant, fordi den er en heterogen multicore arkitektur, der giver mulighed for at foretage parallelle beregninger på 6-8 offload processorer og beregninger foretaget ved brug af vektordatatyper (SIMD). Signalmodellen opstilles, så denne giver mulighed for udnyttelse af CBE arkitekturens muligheder.
For at evaluere arkitekturens potentielle regnekraft udføres to forsøg, hvor den interne databus til overførsel af data mellem processorer testes, og hvor beregningshastigheden ved multiplikation af vektorer undersøges. Derefter foretages en partitionering af demodulations algoritmen, som implementeres på CBE platformen for test.
Disse test viser, at den foreslåede implementering kan demodulere en 10 ms kommunikationssekvens på 84,2 ms og at udnyttelsen af de mulige 76,8 GFLOPS er 10,3%. Den lave udnyttelsesgrad skyldes, at partitioneringen af algoritmen ikke udnytter CBE offload processorernes pipelines tilfredsstillende. En alternativ partitionering foreslås, men undersøges ikke nærmere.

Preface

This report is documentation for the 9th semester Applied Signal Processing and Implementation (ASPI) project concerning "DS-CDMA Procedures with the Cell Broadband Engine" at the Institute of Electronic Systems at Aalborg University (AAU). The report is prepared by group 07gr942 and spans from September 3rd, 2007 to December 20th, 2007. The project is conducted in collaboration with Rohde & Schwarz Technology Center A/S who offered the original project proposal "Multi-core processing with the Cell processor". The project is supervised by Yannick Le Moullec and Andreas Popp from AAU and Kristian Sørensen from Rohde & Schwartz Technology Center A/S.

The report contains 3 parts and no appendices. Development of the project follows the A^3 model which is further described in the Design Methodology, page 7. The following pages contains an list of content, lists of figures and tables and a nomenclature list for selected abbreviations.

The bibliography is found on page xii with references to the bibliography in square brackets as in [1]. The cited source [1] is the project web-page which contains the report, source code and documented code for the project. All code and materials are also available on the accompanying CD attached to the inside of the back cover of this report.

Peter August Simonsen

Jes Toft Kristensen

Contents

List of Figures

List of Tables

Nomenclature

AWGN Additive White Gaussian Noise, page 17

BER Bit Error Rate, page 22

BS Base Station, page 3

CDMA Code Division Multiple Access, page 4

DMA Direct Memory Access, page 32

DS-CDMA Direct Sequence CDMA, page 4

EA Effective Address, page 65

EIB Element Interconnect Bus, page 31

FLOPS Is for this project defined as floating point multiplications per second, page 76

IPC InterProcess Communication, page 65

LS Local Storage, page 32

MAC multiply and accumulate, page 57

MFC Memory Flow Controller, page 32

MS Mobile Station, page 3

OVSF Orthogonal Variable Spread Factor, page 20

PN Pseudorandom Noise, page 4

PPU Power Processing Unit, page 31

pthread POSIX thread, page 37

QPSK Quadrature Phase-Shift Keying, page 14

SIMD Single Instruction Multiple Data, page 33

SPU Synergistic Processing Unit (general), page 31

Notation

The notation used throughout this report is documented below.

Symbol	Association
s	Mathematical variables in italics
$\bar{\bar{\mathbf{A}}}$	The matrix A
$\bar{\mathbf{b}}$	The vector B
$\bar{\mathbf{a}} \odot \bar{\mathbf{b}}$	The element wise product of vectors a and b
$x\%y$	x modulus y
$\bar{\mathbf{a}}^H$	The Hermitian transpose of vector a
$\bar{\mathbf{r}}(\{s+0, s+1, s+2, s+3\})$	Four element vector: consists of elements {s...s+3} of $\bar{\mathbf{r}}$
$\lfloor a \rfloor$	The expression of a floored
[1, p. 42]	Bibliographic reference to index [1] page 42

Bibliography

[1] The Project Group 07gr942, December 2007. The documented code for the project can be found on the accompanying CD or at http://kom.aau.dk/group/07gr942/.

[2] 3GPP. *Technical Specification Group Radio Access Network: Spreading and modulation (FDD)*. 3rd Generation Partnership Project, v7.2.0 edition, 2007. http://www.3gpp.org/ftp/Specs/html-info/25213.htm.

[3] Bo Bjerrum, Jes Toft Kristensen, and Klaus Dahl Kristiansen. *Noise Reduction for Hands-free Car Phone*. AAU, 1st edition, 2007. Report available at: http://kom.aau.dk/group/07gr840/turnin/.

[4] Alfredo Buttari, Piotr Luszczek, Jakub Kurzak, Jack Dongarra, and George Bosilca. *SCOP3 - A Rough Guide To Scientific Computing on the Playstation 3*. Innovative Computing Laboratory, University of Tennessee Knoxville, 2007. Get from http://www.netlib.org/utk/people/JackDongarra/PAPERS/scop3.pdf.

[5] Daniel Hackenberg. *Fast Matrix Multiplication on CELL (SMP) Systems*. TU-Dresden, 2007. http://tu-dresden.de/die_tu_dresden/zentrale_einrichtungen/zih/forschung/architektur_und_leistungsanalyse_von_hochleistungsrechnern/cell/index_html.

[6] Simon Haykin. *Adaptive Filter Theory*. Prentice Hall, 4th edition, 2002. ISBN 0-13-090126-1.

[7] Simon Haykin. *Communication Systems*. Wiley and Sons, 4th edition, 2001. ISBN 0-471-17869-1.

[8] IBM. *SPE Runtime Management Library*. IBM Systems and Technology Group, version 2.7 edition, 2007. Get from ibm.com.

[9] IBM. *Cell Broadband Engine Programming Handbook*. IBM Systems and Technology Group, version 1.1 edition, 2007. Get from ibm.com.

[10] IBM. *IBM Alphaworks XL compiler*. IBM, v9.0 edition, 2007. http://www-306.ibm.com/software/awdtools/xlcpp/library/?S_TACT=105AGX16&S_CMP=LP.

[11] Axel Jantsch, Shashi Kumar, and Ahmed Hemani. *The Rugby Meta Model*. Royal Institute of Technology, Sweden, 2000. Get from: http://www.ele.kth.se/ESD/doc/ar00/Axel/main-v3.fr.pdf.

[12] J. A. Kahle, M. N. Day, H. P. Hofstee, C. R. Johns, T. R. Maeurer, and D. Shippy. *Introduction to the Cell Multiprocessor*. IBM, 2005. Get from http://www.research.ibm.com/journal/rd/494/kahle.pdf.

[13] Persa Kyritsi. *Short Term Fading (wideband) Physical Description*. AAU, 1 edition, 2007. http://kom.aau.dk/~persa/Semester8/2007/mm7_wb_physical/mm7_lecture.pdf.

[14] LLNL. *POSIX Threads Programming*. Lawrence Livermore National Laboratory, July 2007. Get from: http://www.llnl.gov/computing/tutorials/pthreads/.

[15] Quirin N. Meyer. *Programming the SPEs*. MB-JASS 2006, 2006. http://www5-alt.informatik.uni-erlangen.de/Lehre/WS0506/MB-JASS06/slides/p-2-2.pdf?language=de.

[16] Yannick Le Moullec. *DSP Design Methodology*. AAU, 2007. Lecture notes for mm1 of course in DSP Design Methodology, ASPI8-4 http://kom.aau.dk/~ylm/aspi8-4/aspi8-4-part1-2007.pdf.

[17] Lars K. Rasmussen, Paul D. Alexander, and Teng J. Lim. *A Linear Model for CDMA Signals Received with Multiple Antennas over Multipath Fading Channels*. CDMA Techniquess for 3rd Generation Mobile Systems, chap. 2, Kluwer Academic Publisher, 1999.

Part I

Analysis

The analysis part contains the introduction to the project application of a DS-CDMA up-link and development platform for the Cell Broadband Engine. Next, a design methodology for the project is presented to provide a framework for the presented work. The final chapters of the analysis are concerned with deriving and validating a signal model for the application and an analysis of the Cell Broadband Engine platform implemented on a PlayStation 3.

Contents

Chapter 1

Introduction

This introduction aims to give a general overview of the project scope and presents first the application of a CDMA uplink and secondly the Cell Broadband Engine architecture platform to be investigated in the project. Finally, the project problem specification and delimitations are stated.

1.1 The CDMA Up-link Application

The application considered in this project is the process of receiving data from a set of mobile stations (MSs) transmitted to a base station (BS) in a system with multiple users accessing the same BS through a wireless communication channel and with potential interference from users neighboring BS cells as depicted in figure 1.1. This section gives an informal introduction to the application which is discussed in detail in chapter 3.

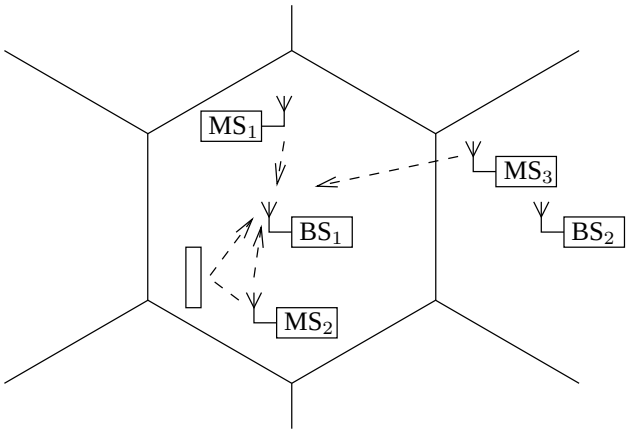


Figure 1.1: *A CDMA system cell, with several mobile stations ($MS_{1,2}$) connected to a base station (BS_1), and crosstalk from neighboring BS cells (MS_3 signal received at BS_1).*

Code Division Multiple Access (CDMA) is a framework for communication channel access which is based on spread spectrum techniques. Where Time Division Multiple Access (TDMA) technologies divide the communication channel into time slots and Frequency Division Multiple Access (FDMA) divides the channel into frequency bands, which may be assigned to each user, CDMA technologies assign codes to distinguish between users.

In Direct Sequence CDMA (DS-CDMA) , these codes are used to spread the signal transmitted from MS_i to BS_j [6, sec. 7.4]. When spreading each MS_i data symbol is multiplied with a user-unique sequence. This operation effectively upsamples the signal from MS_i by a factor equal to the length of the spreading sequence. The principal effects on the signal power spectrum is shown in figure 1.2, and illustrates the achieved spread spectrum characteristic, which effectively allows for transmission of data within the noise floor of the communication channel.

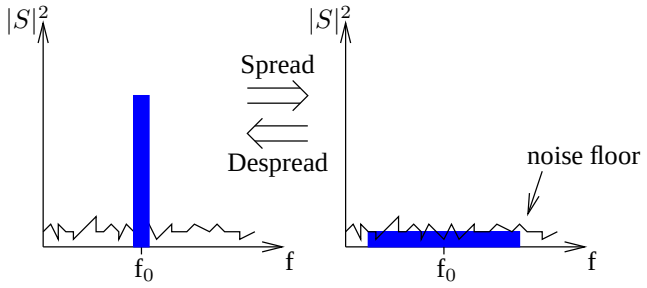


Figure 1.2: *The power spectrum (blue area) of a signal before and after spreading. Spreading allows for transmission of signals within the channel noise floor. The signal may be reconstructed by multiplication of the received signal with the unique spreading code.*

In a system with multiple MSs and noise contributions, the original signal from a specific MS_i may be recovered from a despreading operation, where the spreading sequence for MS_i is multiplied onto the received signal. For perfect recovery the spreading sequences must be orthogonal ensuring that the contributions from each MS is statistically independent.

Besides the application of separating users within a single BS cell, the system must also be able to handle the effects of crosstalk between cells, that is signals received at BS_j originating from MSs belonging to the neighboring cell of BS_k.

In order to allow for reuse of spreading sequences and to handle crosstalk from neighboring BS cells, scrambling is used to randomize the transmitted signals further. This is done by multiplying each sample of the spread signal with a sample from a Pseudorandom Noise (PN) generator. With each BS cell employing individual and uncorrelated scrambling sequences neighboring cells will handle the crosstalk contributions as a noise contribution even though identical spreading sequences are assigned to users within each cell. The scrambled signal may be recovered at the designated BS from the assigned PN sequence in the same way as for recovering a spread signal.

A more detailed examination of the despreading and descrambling operations at a BS as well as handling the effects of passing the transmitted signals through the communication channel are

presented in the signal model in chapter 3.

1.2 Cell Broadband Engine

The purpose of this project is to examine the challenges of implementing algorithms for the DS-CDMA application just presented on a platform based on the Cell Broadband Engine architecture. The CBE is, from the viewpoint of this project, constructed to remedy the following two problems [12, p. 590]:

- Memory access latency and memory transfer bottlenecks
- Diminishing returns from increased processor clock frequency and pipeline length

With the clock frequencies of processors increasing, the latency of accessing RAM becomes more crucial and more of a bottleneck. Furthermore, an increase in processor frequency and pipeline length does not yield as much performance gain as earlier, due to higher chance of wait states caused by inter-data dependencies.

As a means to circumvent these problems the CBE is constructed as a heterogeneous processor architecture, with multiple executions and memory transfers active at the same time.

This yields a processor on a single die which contains a standard PowerPC unit (PPU) and eight simpler "offload" processors, Synergistic Processing Units (SPUs), which are designed to do calculations like DSPs, while the PowerPC performs control, data management and scheduling of operations.

The offload processors are constructed with a short pipeline and depends on the programmer for scheduling of instructions. Furthermore the instruction and execution word length is wide, allowing calculation on multiple data streams in the same instruction by application of Single Instruction Multiple Data (SIMD) intrinsics. This produces a processor optimized for calculations and with very limited branch-prediction.

1.3 Problem Specification

The project problem specification is:

Which factors are important in utilizing the CBE and how does this apply to a CDMA demodulation implementation?

1.4 Evaluation Parameters

The project is evaluated by comparing the achieved time performance with that which is required for a base station in a similar scenario. Furthermore the efficiency in functional unit utilization will be examined for the implementation and compared to the theoretical maximum.

1.5 Project Delimitations

The CDMA demodulation problem is not the focus of this project. It is merely a means to test the CBE on an actual problem. Thus the performance of the CDMA will only be examined to verify the correctness of MatLAB simulations and later that the implementation performs the same operations as the simulation.

As the project aims to investigate the CBE with regards to the problem, no complete solution to the problem is provided, but merely a delimited implementation which only demodulate one communication burst.

In the analysis, mapping and implementation the constants in table 1.1 are used and discussed further in section 3.9 on page 23. These values are determined based on the selected communication burst length and are assumed representative of a working base station.

Symbol	Value	Interpretation
T_c	10	Communication burst length in ms
M	16	Multipaths
K	128	Users
S	128	Spread factor in chips per symbol
N	300	Symbols transmitted per burst
T_d	127	Maximum delay in multipaths in samples

Table 1.1: *Defined constants used in the project. These are set by the project group and are assumed representative for a working base station. Throughout the report indexes m, k, and n are used to index multipaths, users and symbols. The values are in the range of {0..M-1}, {0..K-1}, and {0..N-1}, respectively.*

Chapter 2

Design Methodology

For this project, a custom design methodology is used. This methodology has sought inspiration in the A^3 (A cube) model used at AAU [16] and the Rugby meta-model [11]. The A^3 model is further demonstrated by this report [3, p. 5 and general structure]. This chapter describes the used design methodology.

Purpose of the Design Methodology

As the main purpose of the project is that of extracting maximum utilization of the CBE the design methodology must reflect this. Efficiency in this regard is defined as maximum utilization of all functional units on the CBE, functional units being further defined as contained in the CBE offload processors (SPUs).

With the possibility of many concurrent calculations, the purpose thus becomes that of calculating as many results as possible in parallel, and not necessarily one result as fast as possible, thus emphasizing concurrency rather than serial throughput. This must be contained in the design methodology and be weighted in the design.

Differences in Design Methodology with the Cell Broadband Engine

The use of the CBE forces different considerations onto the design methodology than would be found in methodologies where the architecture is customizable. This mainly consists of:

- The hardware is predefined
- The memory architecture makes special demands

this leads to the following constraints to gain maximum utilization:

- The data must be processed with a SIMD method to exploit the functional units of the SPUs
- The tasks must be parallelized and run concurrently to exploit all of the SPUs

- The memory architecture must be exploited to avoid program transfer latency

This will be further discussed in the platform analysis, page 31.

The design methodology must be adapted to reflect this, which is chosen to be done in two ways:

- The design model will deal only with software design
- The special requirements for parallelization and SIMD usage for the CBE is taken into consideration

Adaptation of the A^3 and Rugby meta-model Design Methodologies

A diagram of the A^3 model combined with the design representations of the Rugby meta-model is proposed in figure 2.1. Here the development is divided into the three domains: Application, Algorithm and Architecture which are the abstraction levels which will be used in this project. Thus changing the focus domain involves changing the level of abstraction. Furthermore, the design representations: data, communications, time and computations from the Rugby meta-model are used to further define the focus areas for each abstraction level.

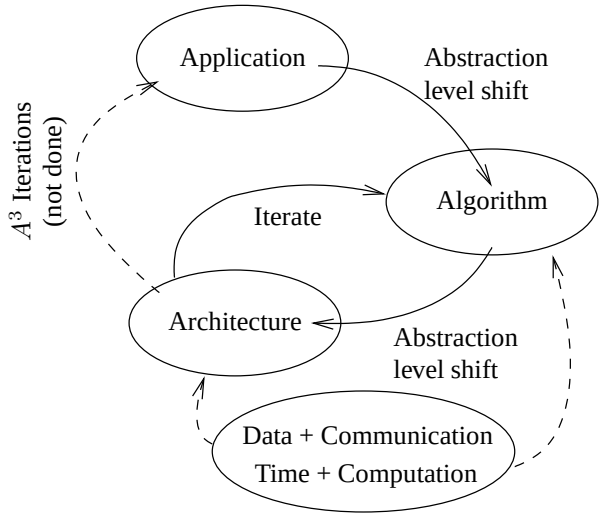


Figure 2.1: *The A^3 design methodology divides the product development into 3 abstraction levels, here also shown with the design representations of the Rugby meta-model*

To utilize the A^3 model at its best, several iterations with measurements of performance and other design criteria must be performed. This project will deviate from this in that the hardware is predefined, and thus the design method and actions will remind more of fitting the algorithm to the platform. As the focus for this project is the actual implementation on the CBE, the the project group will not perform iterations across all abstraction levels, but instead a single

iteration through the application and algorithmic levels and try a single architectural fitting at the architectural abstraction level.

As a consequence of the predefined hardware, the domains from the Rugby meta-model is also only associated with time and computations for the algorithmic abstraction level and data and communication for the architectural level.

Each abstraction level will be discussed in depth in the next section, but is briefly described here:

- Application - Moves the problem solution/algorithm from the idea domain to a working solution, in this project the signal model is verified in MatLAB and the hardware platform is analyzed.
- Algorithm - For this project the algorithmic level is seen as the level for mathematically modifying and fitting the solution found in the application phase to that of the CBE.
- Architecture - At this abstraction level, the algorithm is fitted to the platform, scheduling and binding is performed and memory transfers are defined.

Detailed Design Methodology

Each of the abstraction levels will be defined further here and can be seen in the design trajectory of figure 2.2 on the next page.

Application

The purpose of the application-level is to gain understanding of the usage and general functionality of the problem and the platform. This is performed in the signal model and platform analysis on pages 13 and 31; and includes:

- The signal model is simulated in MatLAB
- A rough identification of blocks and concurrency
- The critical path is defined
- Examination of architecture functional units
- Memory architecture
- Optimal usage of architecture

Algorithm

The algorithmic level incorporates the design representations of time and computation from the Rugby meta-model. The algorithmic abstraction level thus deals with fitting the algorithm to the given platform, the manipulations being performed on algebraic expressions.

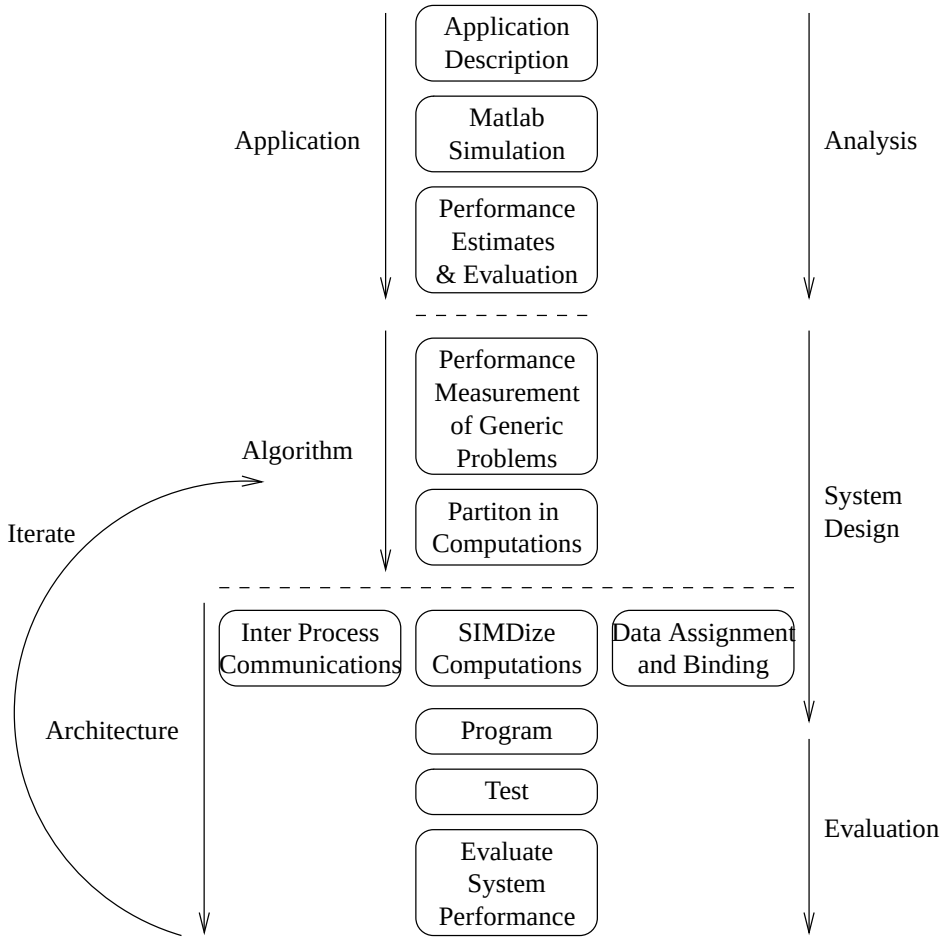


Figure 2.2: *Design trajectory for the project. Labels on the left indicate the abstraction level of each task, where labels on the right refers to the report subdivision into parts.*

- Data and operation transforms
  - Transforming calculations
  - Consider optimized data representations
  - Parallelization of calculations
- Partition into tasks
- Contemplate concurrency

Architecture

The architectural abstraction level moves the algorithm from the mathematical manipulations done at the algorithmic level to an actual implementation in C fitting the CBE.

The architectural levels deals with the following topics:

- Program Synthesis, including:
  - Program structure
  - Task Scheduling
  - Inter-process communication
  - Memory and operations assignment (binding)
  - SIMD transform of calculations
- System test
- Performance evaluation

Chapter 3

Signal Model

The CDMA up-link scenario described in the introduction is here projected onto a signal model which contains all operations needed for the base station to descramble and despread the received signals from the user terminals. The model described here is a delimited model based on [17].

The principle in the project scenario is seen in figure 3.1. The upper most part of the figure, representing the operations at the Mobile Station (MS), is duplicated to model multiple system users. Thus the received signal, $\bar{\mathbf{r}}$, will consist of contributions from several users. The modelling of this recieved signal is done in order to present a method for the descrambling and despreading operations for estimation of the original data symbols, $\hat{\mathbf{d}}_{k,n}$.

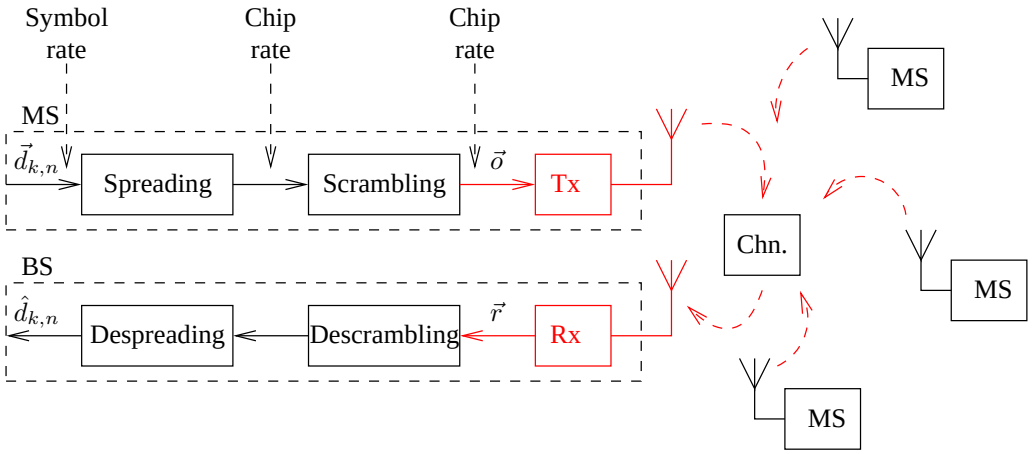


Figure 3.1: *The project signal model. Red areas are not taken into account for this project. One BS receives signals from multiple MSs, each contribution to be descrambled and despread for estimation of the transmitted data symbol n for user k, $\hat{\mathbf{d}}_{k,n}$.*

This signal model description presents the operation and representation of spreading data symbols, scrambling the spread data stream from each user, and passing the signal through a

channel to the receiving base station.

The principles for these operations are first presented for a synchronous CDMA system, where the transmissions from the MSs are assumed synchronized in time at the BS. After presenting these signal processing principles the modifications of the system signal model to model the effects of asynchronous communication and multiple paths are described.

Next, the operations for estimating the original data symbols at the base station are explained from the presented model of the received signal. These base station operations are the points of interest for implementation on the CBE architecture in this project. In order to simulate signals for test in the implementation phase, examples of spread and scramble code generation from [2, p. 18 and p. 21-22] are used and described in connection with MatLAB simulations of the signal model.

Finally an introductory analysis of the algorithm for the descrambling and despreading operations is presented to examine the critical path and the inherent concurrency of the algorithm.

3.1 Model Delimitations

The base station operation in focus in the project is the descrambling and despreading of received CDMA up-link signals at base band. This delimitation in the project focus leads to the following fundamental delimitations in the signal model:

- The model of a transmitted signal from a mobile station is sampled at chiprate (samplerate after spreading).

Since no pulseshaping for the RF part of the system is included, the received signal at the base station need only be sampled at chiprate as well. This introduces a decrease in the amount of calculations needed for the spreading and scrambling procedures which in a real system would have to be included. This is a delimitation performed in order to maintain focus on the platform.

- The attention to effects of passing the transmitted signals through a channel is limited to modelling of the structural consequences to the received signal.

Any procedures for estimation of channel delay and gain coefficient values are left out. The estimation of these channel parameters would require additional modelling of signalling procedures such as midambles or pilot channels.

3.2 System Input

Initially each user generates N symbols which are to be transmitted. These symbols are coherent Quadrature PhaseShift Keying (QPSK) modulated representations of a binary data stream, i.e. $d_k(i) \in \{\pm 1 \pm j\}$. With four possible values of a QPSK symbols, this symbols will represent two bits of the original data stream. A vector representation of the symbol sequences of length

N for K active users, will then be the KN long vector $\bar{\mathbf{d}}$:

$$\bar{\mathbf{d}} = \begin{bmatrix} d_1(0) \\ d_2(0) \\ \vdots \\ d_{K-1}(0) \\ d_1(1) \\ \vdots \\ d_{K-1}(N-1) \end{bmatrix} \tag{3.1}$$

3.3 Spreading

Since each of the K users are transmitting at the same time (i), each symbol to be sent, $d_k(i)$, is modulated by multiplication with a spreading sequence vector, $\bar{\mathbf{s}}_k$:

$$\bar{\mathbf{s}}_k = \begin{bmatrix} s_k(0) \\ s_k(1) \\ \vdots \\ s_k(S-1) \end{bmatrix} \tag{3.2}$$

where the length of the spreading sequence, S, is the ratio between the chip rate after spreading, $1/T_{cr}$, and the data symbol rate, $1/T_{dr}$. The data symbol vector, $\bar{\mathbf{d}}$, defined in equation (3.1), can be spread by left multiplication with a $SN \times NK$ block diagonal matrix $\bar{\bar{\mathbf{S}}}$ to form the SN vector, $\bar{\mathbf{o}}$, containing the sum of the spread data symbols:

$$\bar{\mathbf{o}} = \bar{\bar{\mathbf{S}}} \cdot \bar{\mathbf{d}} \tag{3.3}$$

where $\bar{\bar{\mathbf{S}}}$ has the form:

$$\bar{\bar{\mathbf{S}}} = \begin{bmatrix} \begin{bmatrix} \bar{\mathbf{s}}_0 & \bar{\mathbf{s}}_1 & \dots & \bar{\mathbf{s}}_{K-1} \end{bmatrix} & & 0 & & \dots & & 0 & \\ & 0 & & \begin{bmatrix} \bar{\mathbf{s}}_0 & \bar{\mathbf{s}}_1 & \dots & \bar{\mathbf{s}}_{K-1} \end{bmatrix} & & & \vdots & \\ & \vdots & & & & \ddots & & 0 & \\ & 0 & & & \dots & & 0 & \begin{bmatrix} \bar{\mathbf{s}}_0 & \bar{\mathbf{s}}_1 & \dots & \bar{\mathbf{s}}_{K-1} \end{bmatrix} \end{bmatrix} \tag{3.4}$$

The spreading sequences, $\bar{\mathbf{s}}_k$, assigned to the users are mutually orthogonal. This ensures that the data symbol stream from each user, can be recovered at the receiver, from the knowledge of these spreading codes. It may also be shown from the fact that, since each of the vectors $\bar{\mathbf{s}}_k$ are orthogonal, the transformation matrix, $\bar{\bar{\mathbf{S}}}$, is an orthogonal matrix. Thus $\bar{\mathbf{d}}$ can be retrieved from $\bar{\mathbf{o}}$ as:

$$d = \bar{\bar{\mathbf{S}}}^{\mathbf{H}}\bar{\mathbf{o}} = \bar{\bar{\mathbf{S}}}^{\mathbf{H}}\bar{\bar{\mathbf{S}}}\bar{\mathbf{d}} = \bar{\bar{\mathbf{I}}}\bar{\mathbf{d}} \tag{3.5}$$

3.4 Scrambling

The spread signal is next scrambled in order to separate the cells covered by each base station as shown in figure 1.1. Scrambling is done with a sample-by-sample multiplication of the spread signal with a PN sequence $p(i)$, unique to each BS. Thus the scrambling does not change the chip rate of the signal.

To include scrambling in the signal model of equation (3.3), the scrambling code $\bar{\mathbf{p}}$ is aligned with the first transmission symbol of each user. Thus, the transformation matrix $\bar{\bar{\mathbf{O}}}$, that both spreads and scrambles $\bar{\mathbf{d}}$ is now:

$$\bar{\bar{\mathbf{O}}} = \begin{bmatrix} \begin{bmatrix} \bar{\mathbf{p}}_0 \odot \bar{\mathbf{s}}_0 & \bar{\mathbf{p}}_0 \odot \bar{\mathbf{s}}_1 & \cdots & \bar{\mathbf{p}}_0 \odot \bar{\mathbf{s}}_{K-1} \end{bmatrix} & \begin{matrix} 0 \\ \\ \end{matrix} & \cdots \\ \begin{matrix} 0 \\ \vdots \end{matrix} & \begin{bmatrix} \bar{\mathbf{p}}_1 \odot \bar{\mathbf{s}}_0 & \bar{\mathbf{p}}_1 \odot \bar{\mathbf{s}}_1 & \cdots & \bar{\mathbf{p}}_1 \odot \bar{\mathbf{s}}_{K-1} \end{bmatrix} & \\ & & \ddots \end{bmatrix} \tag{3.6}$$

where $\bar{\mathbf{p}}_n$ is a segment of $\bar{\mathbf{p}}$, defined as:

$$\bar{\mathbf{p}}_n = \begin{bmatrix} p(n \cdot S) \\ \vdots \\ p((n+1) \cdot S - 1) \end{bmatrix} \tag{3.7}$$

In the synchronous CDMA example it is clear that the scramble sequence samples are multiplied on each row of $\bar{\bar{\mathbf{S}}}$. Thus we may write:

$$\bar{\bar{\mathbf{O}}} = \bar{\bar{\mathbf{D}}}(\bar{\mathbf{p}})\bar{\bar{\mathbf{S}}} \tag{3.8}$$

where $\bar{\bar{\mathbf{D}}}(\bar{\mathbf{p}})$ is a $SN \times SN$ diagonal matrix with the scrambling sequence on the main diagonal. The signal model output, $\bar{\mathbf{o}}$, which is the sum of outputs from each mobile station, is now:

$$\bar{\mathbf{o}} = \bar{\bar{\mathbf{O}}}\bar{\mathbf{d}} \tag{3.9}$$

And $\bar{\mathbf{d}}$ may still be recovered from $\bar{\mathbf{o}}$ by left multiplication with $\bar{\bar{\mathbf{O}}}^{\mathbf{H}}$.

3.5 Channel Effects

The spread and scrambled signals is what each mobile station sends to the base station. Although, this project is not concerned with the RF part of the CDMA up-link, the model must take into account some effects of passing the transmitted signals through a channel, which has influence on the received signal $\bar{\mathbf{r}}$.

3.5.1 Channel Gain Coefficients

To model the effects of transmitting from MS to BS, the attenuation of the transmitted signal passing trough the channel must be accounted for. This attenuation is different from each mobile station to the base station, and varies over time, since it depends on channel parameters such as distance from mobile to base station and reflections.

The attenuation is, however, slow varying, and may be assumed constant for a single symbol interval, T_d. Therefore, the channel attenuation may be modeled by multiplying each of the columns of $\bar{\bar{\mathbf{O}}}$ with a channel gain coefficient, $c_{k,n}$, which depends on the user, k, and symbol number, n (source and time). By arranging the channel coefficients in a diagonal matrix, $\bar{\bar{\mathbf{C}}}$:

$$\bar{\bar{\mathbf{C}}} = \begin{bmatrix} c_{0,0} & 0 & \dots & 0 \\ 0 & c_{1,0} & \ddots & \\ \vdots & \ddots & \ddots & 0 \\ 0 & & 0 & c_{K-1,N-1} \end{bmatrix} \tag{3.10}$$

the channel attenuation is included in the synchronous CDMA model by right multiplication of $\bar{\bar{\mathbf{O}}}$ with $\bar{\bar{\mathbf{C}}}$, that is:

$$\bar{\mathbf{y}} = \bar{\bar{\mathbf{O}}}\bar{\bar{\mathbf{C}}}\bar{\mathbf{d}} \tag{3.11}$$

3.5.2 Channel Noise

Apart from the channel fading effects, the received signal will feature a noise contribution from background noise and interference from other base station cells. This noise is modeled as Additive White Gaussian Noise (AWGN) , and is modeled as the SN long vector, $\bar{\mathbf{n}}$ which is added to the matrix product, so the final model of the received signal $\bar{\mathbf{r}}$ at the base station for a synchronous CDMA system becomes:

$$\bar{\mathbf{r}} = \bar{\bar{\mathbf{O}}}\bar{\bar{\mathbf{C}}}\bar{\mathbf{d}} + \bar{\mathbf{n}} \tag{3.12}$$

and the estimate of the original data, $\hat{\mathbf{d}}$, is then:

$$\hat{\mathbf{d}} = \bar{\bar{\mathbf{C}}}^{\mathbf{H}}\bar{\bar{\mathbf{O}}}^{\mathbf{H}}\bar{\mathbf{r}} = \bar{\bar{\mathbf{C}}}^{\mathbf{H}}\bar{\bar{\mathbf{O}}}^{\mathbf{H}}\left(\bar{\bar{\mathbf{O}}}\bar{\bar{\mathbf{C}}}\bar{\mathbf{d}} + \bar{\mathbf{n}}\right) = \mathbf{d} + \bar{\bar{\mathbf{C}}}^{\mathbf{H}}\bar{\bar{\mathbf{O}}}^{\mathbf{H}}\bar{\mathbf{n}} \tag{3.13}$$

As shown in equation (3.13) the estimate of the transmitted symbols $\hat{\mathbf{d}}$ consists of the actual symbol d and a noise contribution of AWGN coloured by $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ and $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$.

3.6 Asynchronous CDMA

So far, the developed signal model of the received signal, $\bar{\mathbf{r}}$, has been assuming perfect synchronization between several users at the base station point of reception. In a real system this feature would be difficult to achieve, when mobile stations are not fixed in position and a global synchronization is not present.

As a result of this the beginning of the reception of a spread and scrambled symbol from user k will be offset compared to the beginning of the symbols from other users. If τ_k is the offset of user k in samples, the spreading and scrambling transformation matrix for a asynchronous CDMA system, $\bar{\bar{\mathbf{O}}}_{\mathbf{AS}}$, which is no longer block diagonal, will have the form shown in figure 3.2.

$$\bar{\bar{0}}_{AS} = \begin{bmatrix} \begin{bmatrix} \bar{p}_0 \odot \bar{s}_0 \end{bmatrix} \begin{bmatrix} \bar{0}_{\tau_1} \\ \bar{p}_0 \odot \bar{s}_1 \end{bmatrix} \cdots \begin{bmatrix} \bar{0}_{\tau_{K-1}} \\ \bar{p}_0 \odot \bar{s}_{K-1} \end{bmatrix} \begin{bmatrix} \bar{0}_S \\ \bar{p}_1 \odot \bar{s}_0 \end{bmatrix} \begin{bmatrix} \bar{0}_{\tau_1} \\ \bar{p}_1 \odot \bar{s}_1 \end{bmatrix} \ddots \end{bmatrix}$$

Figure 3.2: *Structure of $\bar{\bar{\mathbf{O}}}_{\mathbf{AS}}$ for asynchronous CDMA. $\bar{\mathbf{0}}_{\tau_k}$ is a zero vector of length τ_k, and $\bar{\mathbf{0}}_S$ is a zero vector of length S.*

3.6.1 Multiple Path Effects

Finally the effects of reflections of the signal transmitted from each mobile station being received at different times at the base station is included by yet another manipulation of $\bar{\bar{\mathbf{O}}}_{\mathbf{AS}}$, which now will have one column for each received path. The form of the transformation matrix, when including multiple paths, $\bar{\bar{\mathbf{O}}}_{\mathbf{MP}}$, is then as shown in figure 3.3.

$$\bar{\bar{0}}_{MP} = \begin{bmatrix} \begin{bmatrix} \bar{p}_0 \odot \bar{s}_0 \end{bmatrix} \begin{bmatrix} \bar{0}_{\tau_{0,1}} \\ \bar{p}_0 \odot \bar{s}_0 \end{bmatrix} \cdots \begin{bmatrix} \bar{0}_{\tau_{0,M-1}} \\ \bar{p}_0 \odot \bar{s}_0 \end{bmatrix} \begin{bmatrix} \bar{0}_{\tau_{1,0}} \\ \bar{p}_0 \odot \bar{s}_1 \end{bmatrix} \cdots \begin{bmatrix} \bar{0}_{\tau_{K-1,M-1}} \\ \bar{p}_0 \odot \bar{s}_{K-1} \end{bmatrix} \begin{bmatrix} \bar{0}_S \\ \bar{p}_1 \odot \bar{s}_0 \end{bmatrix} \ddots \end{bmatrix}$$

Figure 3.3: *Structure of $\bar{\bar{\mathbf{O}}}_{\mathbf{MP}}$ for asynchronous CDMA with multipath fading. $\tau_{k,m}$ is the offset of the m'th path of the k'th user.*

Each received path experiences an individual channel gain. Therefore, the channel gain matrix must also be modified, and is now a $MK \times K$ block diagonal matrix $\bar{\bar{\mathbf{C}}}_{\mathbf{MP}}$, which has the

form:

$$\bar{\bar{\mathbf{C}}}_{\mathbf{MP}} = \begin{bmatrix} \begin{bmatrix} c_{0,0,0} \\ c_{0,0,0} \\ \vdots \\ c_{0,0,M-1} \end{bmatrix} & 0 & \dots & 0 \\ 0 & \begin{bmatrix} c_{0,1,0} \\ c_{0,1,1} \\ \vdots \\ c_{0,1,M-1} \end{bmatrix} & & \vdots \\ \vdots & & \ddots & 0 \\ 0 & \dots & 0 & \begin{bmatrix} c_{N-1,K-1,0} \\ c_{N-1,K-1,1} \\ \vdots \\ c_{N-1,K-1,M-1} \end{bmatrix} \end{bmatrix} \tag{3.14}$$

where $c_{n,m,k}$ scales the m'th path of symbol n from user k.

3.7 Demodulation

The final model for the received signal in a asynchronous CDMA up-link with multipath propagation, $\bar{\mathbf{r}}_{MP}$ is then:

$$\bar{\mathbf{r}}_{MP} = \bar{\bar{\mathbf{O}}}_{\mathbf{MP}} \bar{\bar{\mathbf{C}}}_{\mathbf{MP}} \bar{\mathbf{d}} + \bar{\mathbf{n}} \tag{3.15}$$

For an estimate of the data signal vector, $\hat{\mathbf{d}}$, the transformation and gain matrices of equation (3.13) are replaced with the new representations of these to get:

$$\begin{aligned} \hat{\mathbf{d}} &= \bar{\bar{\mathbf{C}}}^{\mathbf{H}}_{\mathbf{MP}} \bar{\bar{\mathbf{O}}}^{\mathbf{H}}_{\mathbf{MP}} \bar{\mathbf{r}}_{\mathbf{MP}} & (3.16) \\ &= \bar{\bar{\mathbf{C}}}^{\mathbf{H}}_{\mathbf{MP}} \bar{\bar{\mathbf{O}}}^{\mathbf{H}}_{\mathbf{MP}} \left(\bar{\bar{\mathbf{O}}}_{\mathbf{MP}} \bar{\bar{\mathbf{C}}}_{\mathbf{MP}} \bar{\mathbf{d}} + \bar{\mathbf{n}} \right) = \mathbf{d} + \bar{\bar{\mathbf{C}}}^{\mathbf{H}}_{\mathbf{MP}} \bar{\bar{\mathbf{O}}}^{\mathbf{H}}_{\mathbf{MP}} \bar{\mathbf{n}} & (3.17) \end{aligned}$$

The application to be implemented in the project is thus the matrix product of the matrices $\bar{\bar{\mathbf{C}}}_{\mathbf{MP}}$ and $\bar{\bar{\mathbf{O}}}_{\mathbf{MP}}$ multiplied with a simulated received signal vector $\bar{\mathbf{r}}_{MP}$ as stated in equation (3.16), thus estimating $\hat{\mathbf{d}}$.

3.8 Signal Model Verification

In order to evaluate the functional performance of the implementation of descrambling and despreading on the PlayStation 3, a set of simulated received signals ($\bar{\mathbf{r}}$) have been generated, encoded, and decoded in MatLAB using the described signal model.

3.8.1 Method

Based on the stepwise sophistication of the model described in the chapter introduction on page 13, the following scenarios of a CDMA up-link system have been used in the simulations:

- Synchronous CDMA:
  - Spreading only
  - Spreading and scrambling
- Asynchronous CDMA:
  - Spreading only
  - Spreading and scrambling
  - Multiple paths with individual delays and gains

3.8.2 Spreading Sequences

The orthogonal spreading sequences is generated as Orthogonal Variable Spreading Factor (OVSF) codes codes defined in [2, p. 18]. The principle for generating OVSF codes is shown in figure 3.4.

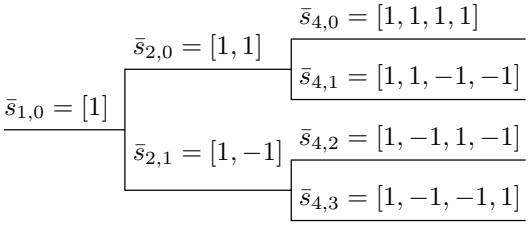


Figure 3.4: *OVSF codes for spread factors $S = \{1, 2, 4\}$. From [2, p. 18]*

.

The individual spreading codes are uniquely identified from the spread factor, S, and a code number, y and may be recursively defined as the rows in Hadamard matrix, $\bar{\bar{\mathbf{H}}}$:

$$\bar{\bar{\mathbf{H}}}_{\mathbf{2S}} = \begin{bmatrix} \bar{\bar{\mathbf{H}}}_{\mathbf{S}} & \bar{\bar{\mathbf{H}}}_{\mathbf{S}} \\ \bar{\bar{\mathbf{H}}}_{\mathbf{S}} & -\bar{\bar{\mathbf{H}}}_{\mathbf{S}} \end{bmatrix} = \begin{bmatrix} \bar{\mathbf{s}}'_{2S,0} \\ \bar{\mathbf{s}}'_{2S,1} \\ \vdots \\ \bar{\mathbf{s}}'_{2S,y} \\ \vdots \\ \bar{\mathbf{s}}'_{2S,S-1} \end{bmatrix} \tag{3.18}$$

where $\bar{\bar{\mathbf{H}}}_{\mathbf{1}} = \mathbf{1}$. As seen from this definition, the maximum number of orthogonal spreading codes, and thus the number of simultaneous users, is limited by the spread factor, $S \in 2^i | i = \{0, 1, 2...\}$.

To avoid any power gain when spreading the data signals, the spreading sequences are normalized, with the normalizing factor, f_{sp} so that:

$$\sum_{i=1}^{S} |f_{sp} \cdot \bar{\mathbf{s}}_{S,s}(i)|^2 = 1 \Leftrightarrow f_{sp} = \frac{1}{\sqrt{S}} \tag{3.19}$$

since $|\bar{\mathbf{s}}_{S,s}(i)|^2 = 1$ for any i.

3.8.3 Scrambling Sequences

The scrambling sequences are generated using a short scramble sequence generator defined in [2, p. 21-22]. The system generates a PN sequence, $p(i)$, from modulo 2 and 4 additions of three recursive generator polynomials, a(i), b(i), and d(i), as shown in figure 3.5.

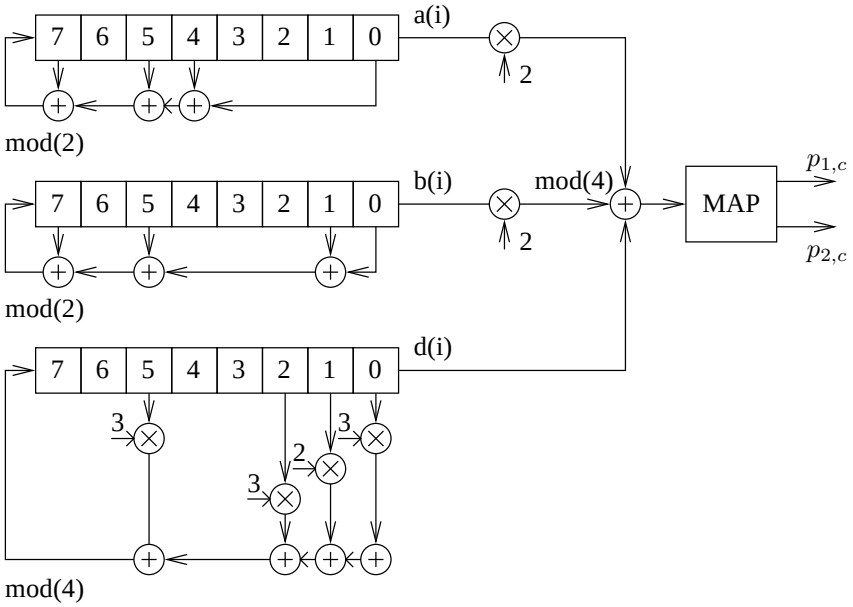


Figure 3.5: *Principle of generating PN sequences $p_{1,c}$ and $p_{2,c}$ for scramble sequence calculation [2].*

The three 8bit shift registers are initialized with the 24bit binary representation of a base station cell number c, where $0 \leq c \leq 2^{24} - 1$. The first 255 samples of the scramble sequence generator is used and a 256' sample is appended, where $p(255) = p(0)$, where $p(i) \in \{0, 1, 2, 3\}$. This sequence is mapped to two binary sequences $p_{1,c}(i) \in \{-1, 1\}$ and $p_{2,c}(i) \in \{-1, 1\}$, which are used to calculate the complex valued scrambling sequence, $p_c(i)$:

$$p_c(i) = p_{1,c}(i \,\%\, 256) \cdot \left(1 + j(-1)^i \cdot p_{2,c}\left(2\left\lfloor \frac{i \,\%\, 256}{2} \right\rfloor\right)\right) \tag{3.20}$$

As for the spreading sequences, the scrambling sequences must be normalized to avoid any gain from this operation. Since the signal chip rate is not changed from the scrambling operation the normalizing factor for scrambling, f_{sc}, becomes:

$$f_{sc} = \frac{1}{|p_c(i)|} = \frac{1}{\sqrt{2}} \tag{3.21}$$

for any i.

3.8.4 Channel Effects

Since no procedures for estimating the channel are implemented, the channel characteristics are modelled with random entries. Channel gain coefficients are complex numbers, where the real and complex parts are generated independently using a normal distribution. This results in gain coefficients with a Rayleigh distributed amplitude, which models the distribution of these coefficients in an urban environment [13, p. 18]. Delays are generated using a random number generator, where the maximum delay is set to S samples. In the simulation scenario with multiple channels the channel coefficients are normalized the squared sum of these coefficients for each user becomes:

$$\sum_{m=1}^{M} |c_{k,m}|^2 = 1 \tag{3.22}$$

This so that no signal power gain is experienced when passing the signal through the channel. The channel noise is modeled as complex AWGN with parameters $\{\mu, \sigma^2\} = \{0, 1\}$.

3.8.5 Error Probabilities

To evaluate the performance of the decoding method, stated in equation (3.17) the achieved results are compared to the theoretical Bit Error Rate (BER) for a coherent QPSK system [7, p. 358]:

$$BER_{QPSK} = \frac{1}{2} e\left(\sqrt{\frac{E_b}{N_o}}\right) \tag{3.23}$$

where E_b is the bit signal energy, N_o is the noise energy, and $e()$ is the complementary error function [7, p. 255]:

$$e(u) = \frac{2}{\sqrt{\pi}} \int_u^\infty exp\left(-z^2\right) dz \tag{3.24}$$

For the system simulated the bit energy is found from the symbol energy E, where two bits are combined into one symbol, thus $E = 2E_b$, and the symbol energy is the squared symbol amplitude:

$$E = \sqrt{(\pm 1)^2 + (\pm 1)^2}^2 = 2 \Leftrightarrow E_b = \frac{1}{2} E = 1 \tag{3.25}$$

The noise energy, N_o, for AWGN$\{0,1\}$ equals the noise variance and thus $N_o = 1$. By equation (3.23) the expected BER for the simulated system is then:

$$BER_{QPSK} = \frac{1}{2}e\left(\sqrt{\frac{1}{1}}\right) \tag{3.26}$$

$$= 0.0786 \tag{3.27}$$

This corresponds to the BER_{QPSK} at a signal to noise ratio of:

$$SNR = 10 \cdot log_{10}\left(\frac{E_b}{N_o}\right) = 0dB \tag{3.28}$$

3.8.6 Simulation Results

Using a spreading factor, S_{sim}, of 16, figures 3.6 and 3.7 show the average correct classification rates, $(1 - BER)$, of the transmitted binary data from each user, k, when the number of users, K_{sim}, is varied from 1 to S_{sim}. Each user sends a total of $N_{sim} = 1000$ QPSK symbols. Finally, in the scenario of multipath propagation the number of received paths, M_{sim}, is 4.

Discussion

For the simulations with synchronous CDMA the achieved rate of correct classification of the transmitted signals is approximately 0.92 as seen in figure 3.6. This corresponds with the value for BER_{QPSK} found in equation (3.27). It is also noticed that the synchronous CDMA systems simulated are indifferent to the number of users, which indicates orthogonality, as expected, between the spread and scrambled signals.

This system indifference to the number of users is the main difference in results when examining the asynchronous CDMA simulations in figure 3.7. Due to the individual time delays introduced between each user and received path the matrix for spreading and scrambling is no longer orthogonal due to random displacement of the columns of $\bar{\bar{\mathbf{O}}}$. This means that the more users introduced to the system, the more interference there will be between them. Solutions to avoid this interference exist, such as spreading sequences generated from low correlation PN sequences analogous to the generation of $\bar{\mathbf{p}}_c$ in section 3.8.3, but will not be investigated further in this project, since the main focus is on the actual demodulation algorithm.

3.9 Critical Path, Storage Size and Concurrency

To estimate the application complexity and computational requirements, the critical path for the demodulation is determined. This is shown with a graphical representation which leads to determination of the approximate amount of calculations needed for calculating a single symbol for one user. For the purpose of this analysis the constants, defined in table 1.1 on page 6, are applied. The communication burst length (T_c) as been chosen to achieve an example chiprate of:

$$f_{chip} = \frac{N \cdot S}{T_c} = \frac{300 \cdot 128}{0.001} = 3.840 \cdot 10^6 \left[\frac{chips}{second}\right] \tag{3.29}$$

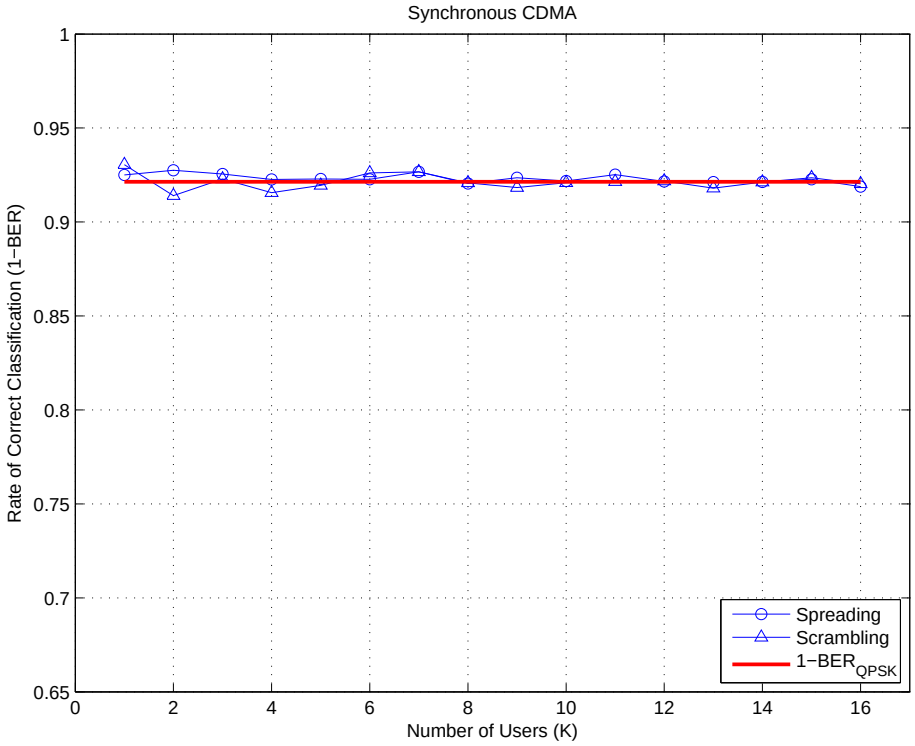


Figure 3.6: *Average of correct classification rate for possible number of users in synchronous CDMA. Red line denotes theoretical value for* $1 - BER_{QPSK}$*, blue lines denote achieved values in simulations.*

which is used in UMTS systems, as are the chosen algorithms for spreading and scrambling codes generation [2].

3.9.1 Graphical Presentation

A graphical representation of the demodulation is seen in figure 3.8 on page 26. The two matrices $\bar{\bar{\mathbf{C}}}^{\mathbf{H}}$ and $\bar{\bar{\mathbf{O}}}^{\mathbf{H}}$ contains the information about multipath fading and the spreading and scrambling respectively. The received samples are placed in $\bar{\mathbf{r}}$ as described in section 3.7.

Multiplication of the row-vector $\bar{\mathbf{C}}_{1,1,M}$ and the columns of $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ produces the row vector in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$. The newfound row vector is multiplied with the samples of $\bar{\mathbf{r}}$ to produce a single symbol in $\hat{d}$.

3.9.2 Critical Path

Determining the critical path is done from figure 3.8 on page 26 which is redrawn as figure 3.9 on page 27 where a single symbol for a single user is seen. Traversal of the critical path will be explained in the next section.

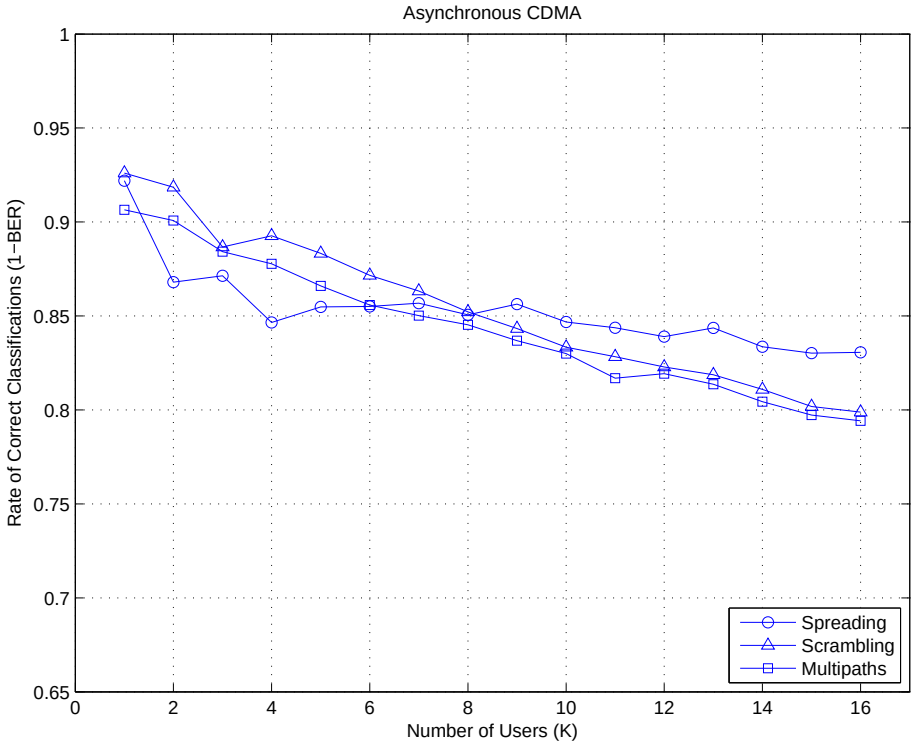


Figure 3.7: *Average of correct classification rate for possible number of users in asynchronous CDMA.*

3.9.3 Size of Calculations

With the graphical representation of figures 3.8 and 3.9 it is straightforward to determine the amount of multiplications needed to compute one symbol.

In the estimation the number of multiplications and additions will be represented by μ and α respectively. All numbers are complex, which will be included at the calculation conclusion.

The first operation is to generate the row vector in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}_{N,K}$ which is done by vector multiplication of $\bar{\mathbf{C}}_{1,1,M}$ and the columns in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$. This requires M multiplications and $M-1$ additions performed between the $\bar{\mathbf{C}}_{1,1,M}$ and the columns in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ performed $S+T_d$ times, in a worst case scenario. This is represented as

$$O_1 = (M \cdot \mu + (M-1) \cdot \alpha) \cdot (S + T_d) \tag{3.30}$$

where O_1 is the number of additions and multiplications.

The second operation is that of multiplying the samples in $\bar{\mathbf{r}}$ with the newfound row. This

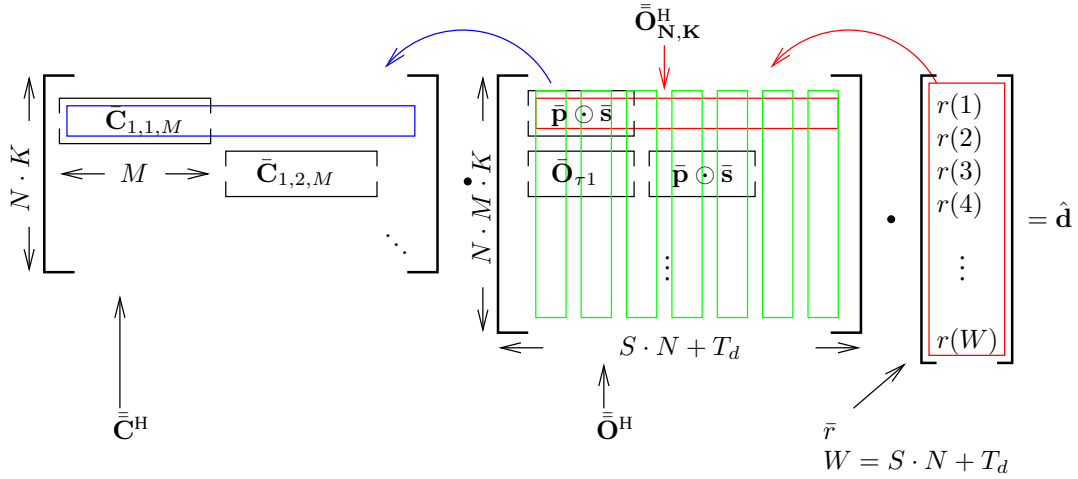


Figure 3.8: *Graphical representation of the demodulation,* $\bar{\bar{\mathbf{C}}}^{\mathbf{H}} \cdot \bar{\bar{\mathbf{O}}}^{\mathbf{H}} \cdot \bar{\mathbf{r}} = \hat{\mathbf{d}}_{\mathbf{k}}(\mathbf{t})$. *The size definitions are for the complete problem.*

requires:

$$
\begin{aligned}
O_2 &= (S+T_d)\mu + \alpha(S+T_d-1) && (3.31)\\
O_{total,real} = O_1 + O_2 &= \mu \cdot (M(S+T_d) + S + T_d) + \dots \\
&\dots \quad \alpha \cdot ((M-1)(S+T_d) + S + T_d - 1) && (3.32)\\
O_{total,real} &= \mu \cdot 4335 + \alpha \cdot 4079 && (3.33)\\
O_{total} &= \mu \cdot 4335 \cdot 4 + \alpha \left(\cdot 4079 \cdot 2 + 4335 \cdot 2\right) && (3.34)\\
O_{total} &= \mu \cdot 17340 + \alpha \cdot 16828 && (3.35)
\end{aligned}
$$

where defined constants have been exchanged and O_{total} is found for the calculations done in the complex domain.

It must be noted that this is a worst case estimate, where it is assumed that T_d is maximum. In fact this will vary wherein the multiply-by-zeros in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ will be fewer. Many of these calculations can be remedied at the cost of more control structure in the setup and calculation of vector multiplications.

Another topic is that of generating $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}$ which is not accounted for in the estimate. In essence this generation will only have to be done once for user and symbol, resulting ind the addition of $\mu \cdot S$ more operations.

3.9.4 Storage Size

Examining the storage sizes for the matrices when handling the entire problem is interesting as the SPUs each have their own local storage of limited size. The CBE architecture is discussed further in section 4.4 on page 32.

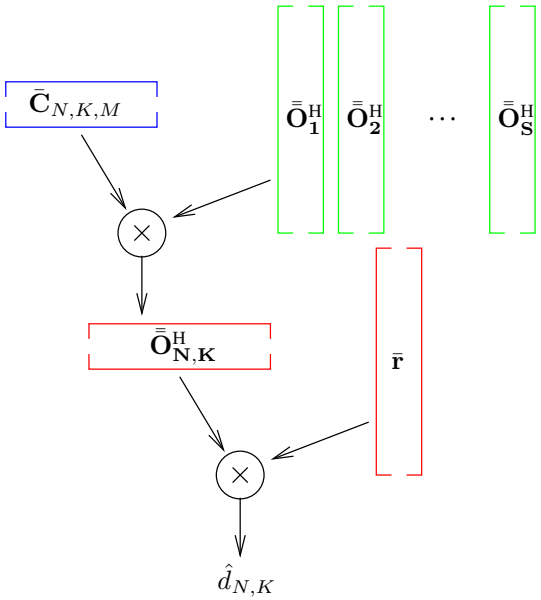


Figure 3.9: *Critical path for one symbol. Extracted from figure 3.8 on the facing page with same colour scheme.*

Each matrix with full representation will have the size of

$$s_{full} = s_{\bar{\bar{\mathbf{C}}}^{\mathrm{H}}} + s_{\bar{\bar{\mathbf{O}}}^{\mathrm{H}}} + s_{\bar{\mathbf{r}}} \tag{3.36}$$

$$s_{\bar{\bar{\mathbf{C}}}^{\mathrm{H}}} = 2 \cdot 4 \cdot (N \cdot K) \cdot (M \cdot N) = 1.47 \text{ GB} \tag{3.37}$$

$$s_{\bar{\bar{\mathbf{O}}}^{\mathrm{H}}} = 2 \cdot 4 \cdot (N \cdot M \cdot K) \cdot (S \cdot N + T_d) = 189.37 \text{ GB} \tag{3.38}$$

$$s_{\bar{\mathbf{r}}} = 2 \cdot 4 \cdot (S \cdot N + T_d) = 308.2 \text{ KB} \tag{3.39}$$

$$s_{full} = 2 \cdot (1.47 \text{ GB} + 189.37 \text{ GB} + 308.2 \text{ KB}) = 190.8 \text{ GB} \tag{3.40}$$

where the factor of $2 \cdot 4$ is multiplied as a single precision complex floating point number takes $2 \cdot 4$ bytes of 8 bit storage.

To reduce the size, the matrix could be stored as a sparse matrix, wherein the matrix coordinate (x,y) of the number and the number itself is stored. Assuming a 16 bit integer for storage of position, the size calculations will be

$$s_{\bar{\bar{\mathbf{C}}}^{\mathrm{H}}} = (2 \cdot 2 + 2 \cdot 4) \cdot (M \cdot N \cdot K) = 7.37 \text{ MB} \tag{3.41}$$

$$s_{\bar{\bar{\mathbf{O}}}^{\mathrm{H}}} = (2 \cdot 2 + 2 \cdot 4) \cdot (N \cdot M \cdot K) \cdot (S) = 943.72 \text{ MB} \tag{3.42}$$

$$s_{\bar{\mathbf{r}}} = 2 \cdot 4 \cdot (S \cdot N + T_d) = 308.2 \text{ KB} \tag{3.43}$$

$$s_{full} = 7.37 \text{ MB} + 943.72 \text{ MB} + 308.2 \text{ KB} = 951.39 \text{ MB} \tag{3.44}$$

where $\bar{\mathbf{r}}$ has been stored as before, since it would not yield less storage space required to store $\bar{\mathbf{r}}$ as a sparse matrix.

A further reduction can be achieved by exploiting the matrix structure which shows that the vectors are of known length with an individual offset. Again assuming a 2 byte storage of the offset, the sizes would be[1]

$$s_{\bar{\bar{\mathbf{C}}}^{\mathrm{H}}} = 2 \cdot (N \cdot K) + 2 \cdot 4 \cdot (M \cdot N \cdot K) = 4.99 \text{ MB} \quad (3.45)$$

$$s_{\bar{\bar{\mathbf{O}}}^{\mathrm{H}}} = 2 \cdot (N \cdot M \cdot K) + 2 \cdot 4 \cdot (N \cdot M \cdot K) \cdot (S) = 630.37 \text{ MB} \quad (3.46)$$

$$s_{\bar{\mathbf{r}}} = 4 \cdot (S \cdot N + T_d) = 308.2 \text{ KB} \quad (3.47)$$

$$s_{full} = 4.99 \text{ MB} + 630.37 \text{ MB} + 308.2 \text{ KB} = 635.67 \text{ MB} \quad (3.48)$$

which shows that large amounts of storage is needed to solve the problem directly. These amounts of storage are not available on the CBE, so a partitioning of the problem is needed, This partitioning is performed in section 6.1 on page 53.

3.9.5 Concurrency

The final topic is that of concurrency. Examining figures 3.8 on page 26 and 3.9 on the preceding page, it is seen that there are no successive data dependencies, except from those defined by the two successive matrix multiplications. As such concurrency can be introduced almost at wish.

One method of partitioning is seen in figure 3.10, where the uppermost column of $\bar{\mathbf{r}}$ is multiplied onto the rows of $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ and another concurrent process performs the same operation for the other set of columns and rows. This partition can be repeated until the need for concurrency is satisfied, while lowering the demands for storage. In the subdivision process one could also take SIMD-ing of data into account which is one of the key mechanisms in achieving high performance on the CBE. This process is discussed further in the program partitioning, section 6.1.2 on page 54.

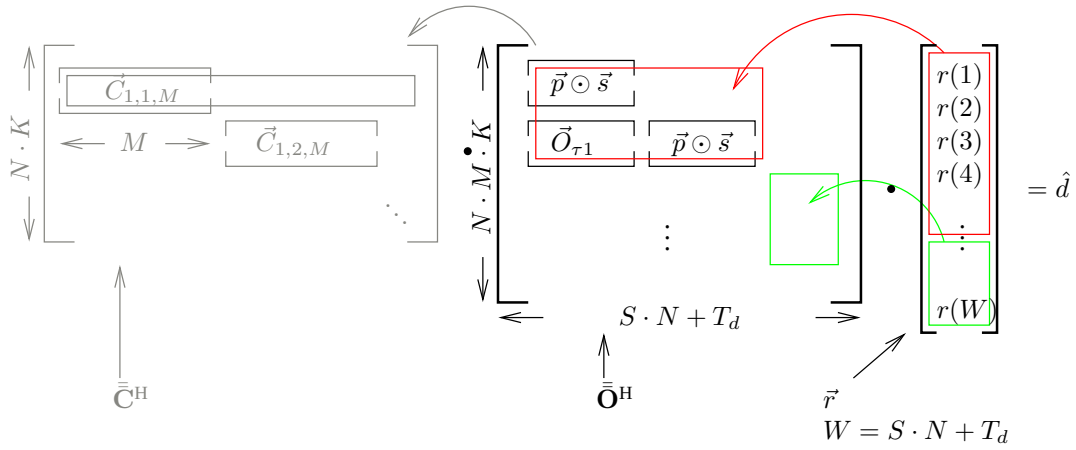


Figure 3.10: *Possible partition of matrix multiplication*

[1] As special cases every M'th entry will have an offset of zero, or a predictable integer multiple of M for $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$. These have been ignored for the calculations.

The subdivision can be performed likewise for the matrix multiplication of $\bar{\bar{\mathbf{C}}}^{\mathrm{H}} \cdot \bar{\bar{\mathbf{O}}}^{\mathrm{H}}$, yielding the same possibilities.

Chapter 4

Architecture Analysis

This chapter contains a architecture analysis of the Cell Broadband Engine in general and relevant specifics of the PlayStation 3 implementation of the CBE. Most of this chapter builds on the Cell Broadband Engines Programmers Handbook [9], a guide to scientific computing on the PlayStation 3[4], and the platform development introduction [12].

Initially the purpose is stated along with a general architecture overview. Afterwards the PowerPC unit and the synergistic processing units are examined. Lastly the memory and access architecture is explored. This leads to a short discussion of the strong and weak sides of the platform.

This analysis is very condensed as the above sources give much information. For further insight into the ideas behind the platform [12] is recommended while [4] gives useful insights into the usage of the processor.

4.1 Purpose

The primary goal of this chapter is to uncover information with regards to efficient utilization of the Synergistic Processing Units (SPUs) and how this is achieved with the PPU.

4.2 Architecture Overview

The Cell Broadband Engine (CBE) is a heterogeneous multicore processor and consists of a PowerPC processing Unit (PPU) and 8 Synergistic Processing Units (SPU) . The PPU is a standard PowerPC with two cores. It accesses the RAM memory via an Element Interconnect Bus (EIB) , which also functions as the SPUs memory access. An overview of the CBE architecture is presented in figure 4.1 on the next page.

It must be noted that one SPU is used for the PlayStation 3 hypervisor[1] and another is disabled

[1] Also known as the “Game OS”, a layer built by Sony which provides access to the hardware [4, p. 5].

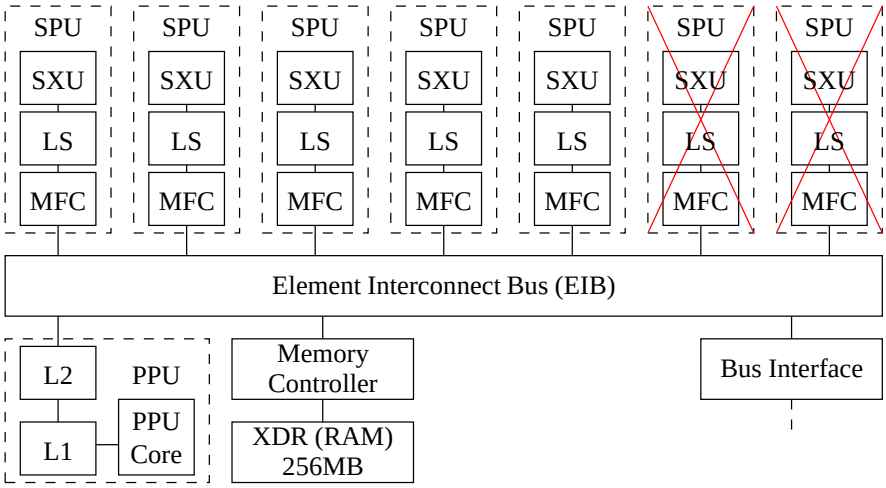


Figure 4.1: *Architecture overview of the Cell Broadband Processor. The Element Interconnect Bus serves as a connection between the Power Processing Unit (PPU), the memory controller and thus the RAM and 8 Synergistic Processing Units (SPUs). The SPUs each consist of a Memory Flow Controller (MFC), Local Storage (LS) and a Synergistic eXecuting Unit (SXU). Two SPUs are disabled on the PlayStation 3. Adopted from [4, p. 4] and [12, p. 592].*

in the PlayStation 3, thus only 6 SPUs are available, also only 6 when running Linux, as shown in figure 4.1 [4, p. 5].

4.3 PowerPC Unit

As the main controller for the system the PPU figures as a standard 64bit PowerPC architecture. This includes two cores which run at a clock frequency of 3.2 GHz and provides a common entry point for programmers. The PlayStation 3 is capable of functioning solely by running the operating system on the PPU. Usage of the SPUs is initiated and controlled from the PPU via specific libraries.

For this project the PPU is regarded as a control-unit and will not be used for computations. It is assumed that the PPU is fast enough to perform the task of controlling the SPUs.

4.4 Synergistic Processing Unit

The SPUs contains a RISC processor with 256 KB of Local Storage (LS) RAM and a 128 entry 128 bit wide register file for processor registers. The LS is filled by Direct Memory Acces (DMA) transfers controlled by the Memory Flow Controller (MFC) which is connected to the EIB. The SPUs are intended to run their own programs from LS.

The functional units of the SPUs are divided into an even- and odd pipeline (pipeline 0 and 1 respectively), as shown in figure 4.2 on the following page.

Examining the SXU of figure 4.2 it is seen that SFS, SLS, SCN and SSC combines to a data moving entity while SFX and SFP provides the data processing units. The even fixed point unit SFX can perform arithmetic- and logic instructions, shifts and rotates and floating point compares, reciprocal and reciprocal square root estimates. The floating point unit SFP can perform fully pipelined single precision (32 bit) floating point instructions and partially pipelined double (64 bit) precision instructions.

The SPU is running qith a clock frequency of 3.2 GHz and each pipeline can execute an instruction each cycle. The datapath of the arithmetic functional units SFX, SFP, SFS are 128 bits wide, resulting in the capability of using Single Instruction Multiple Data (SIMD) instructions, thus $4 \cdot 32$bit multiplications can be issued each instruction.

With one pipeline dedicated to data movement and the other performing single precision floating point multiplications each SPU should give a theoretical output of $4 \cdot 3.2$ GFLOPS , this yields a total of 76.8 GFLOPS for 6 active SPUs. A further discussion on FLOPS as a benchmark measurement is found in the test definition, section 9.1.2 on page 75.

It must be noted that single precision floating point operations does not conform completely to IEEE 754 as only truncation is used in rounding, denormal numbers[2] are forced to zero and NaN[3] are treated as a number [9, p. 68]. The double precision floating point operations do not suffer from this limitation.

Another element which must be considered for SIMD instructions is that of data-alignment. As the SPU is only able to load words from LS which are aligned to a 128bit memory boundary, the data must be constrained to these boundaries for correct operation.

[2]Denormal numbers are numbers smaller than the smallest number in floating point

[3]Floating point with exponent of all ones

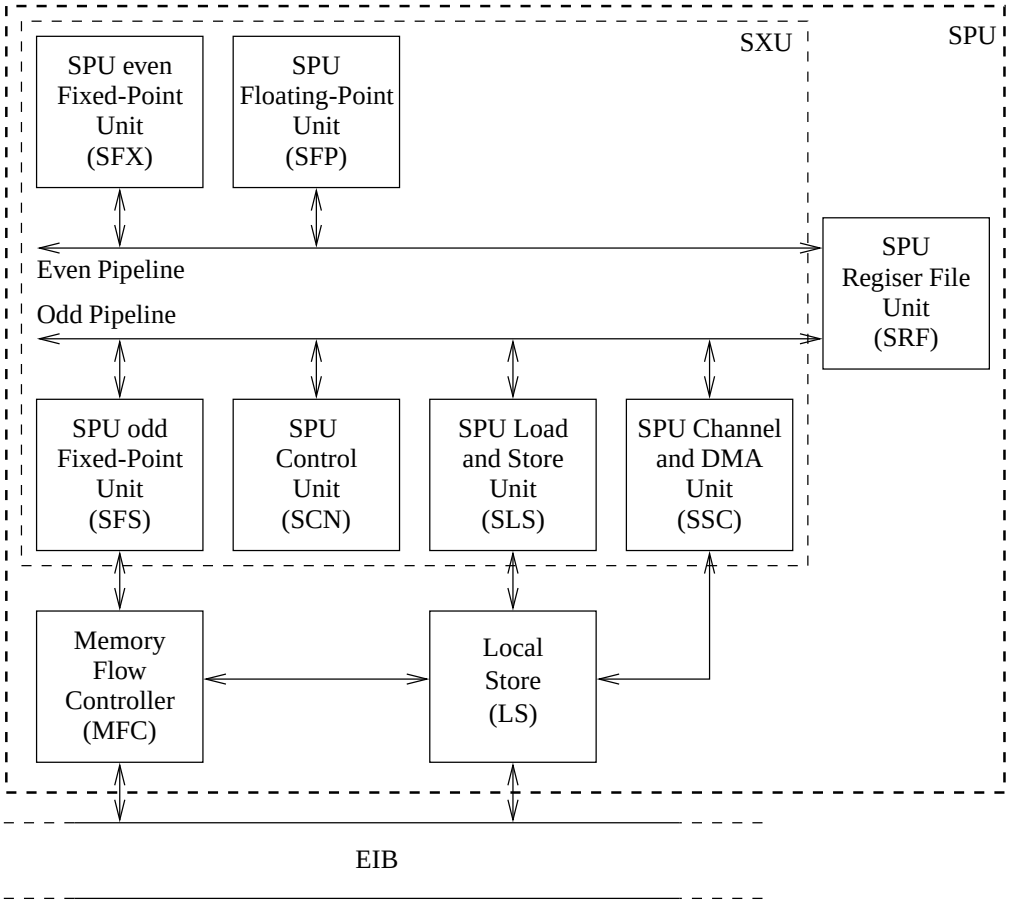


Figure 4.2: *Overview of a synergistic processing unit and its context. The local storage is filled by DMA transfers from the EIB via the MFC and later accessed via the odd pipeline. The SXU contains two fixed point units and a single-precision floating point unit. The graphics is adopted from [9, p. 64]*

4.5 Memory Architecture and Communication

One of the design parameters of the CBE is that of accessing memory efficiently. This is achieved by usage of the EIB which has a bandwidth of 25.6 GB/s (96 bytes per clock cycle) and enabling multiple concurrent data transfers [9, p. 42]. The system memory is 256 MB of dual channel Rambus RAM, which is used to run the operating system and initially contains the SPU programs. The RAM is accessed from the SPU via the EIB and moved to the respective SPU local storage via DMA transfers, with the MFC of the SPU acting as DMA controller.

Each MFC has a queue of DMA transfers, where each transfer can be a maximum of 16 KB. Furthermore the MFC has a system of mailboxes and signals which can parse messages to and from the PPU and SPUs. The DMA transfers are intended to be used as data transfers while the mailboxes and signals are intended for inter-process communication [9, cha. 19].

4.6 Optimized Utilization of the SPUs

The three main challenges in extracting maximum performance from the CBE are those of

1. Identifying tasks which can be parallelized
2. Avoiding memory latencies in data and program transfers to/from the SPUs
3. Serializing data to allow SIMD usage

each of which will be discussed in the following.

The strong point of the CBE is the ability to execute tasks concurrently on each SPU. The challenge then lies in identifying concurrent tasks and ensuring that each task has a sufficient duration to allow concurrent transfer of the next task. If the task duration is shorter than the transfer time for the next task, the SPU will be forced to wait thus loosing efficiency.

Exploiting the architectural possibilities, namely the SIMD instructions, is another topic which must be addressed. This calls for a design exploration which includes alternative solutions relying more on brute force approaches and less on control/evaluation-approaches. Especially the control and evaluation must be avoided with regards to the SPUs, as these contain little branch prediction and no cache. The relative short pipeline seems to compensate for this, but a pipe shunt and subsequent clear wastes a potential of $4 \cdot 18 = 72$ single precision floating point multiplies, where 18 is the pipeline latency for a single precision floating point operation [9, p. 691].

As a smaller problem the software must be partitioned to fit in LS of the SPUs (256 KB). This renders it unlikely that the entire data can be placed in directly accessible memory for all SPUs, thus a data and program partition must be developed with lower bound of the program and data transfer time and upper bound by the program and data fitting in LS.

4.7 Programming Environment and Intrinsics

In order to promote structured software development for the CBE in the following system design part, this section presents the project development platform and a basic structure for software design and implementation.

4.7.1 Development Platform

As mentioned the platform for programming applications for the CBE in this project is a Sony PlayStation 3. Besides the console itself, the equipment includes a monitor, keyboard, mouse and LAN connection for remote access.

The PlayStation 3 for this project is installed with a Linux operating system and a set of development tools:

- Fedora 8 Linux kernel 2.6.23.1-42.fc8
- IBM SDK3.0 for the CBE architecture, including:
  - gcc compiler toolchain for the CBE (*ppu-gcc* and *spu-gcc* ver. 4.1.1)
  - *libspe2* - SPE runtime management library ver. 2.2 [8]
  - Custom makefile

4.7.2 Basic Programming for the CBE Architecture

The programming for the CBE is divided into two principal tasks, namely the development of program for the PPU-element, and programs to be run on the SPUs. In this section we present the basic programming mechanisms for developing an application which enables concurrent execution of SPU-programs.

SPU Programming using Contexts

Programs for SPU execution are developed and compiled separately. As described in section 4.7 references to the SPU programs are embedded into the main program during compilation.

To run a SPU program, the main application, run on the PPU, creates a SPU context, loads the embedded program into this context. Next program requests the execution of the context on the first available SPU. This flow of creating, loading and running a single SPU-context is shown in figure 4.3. When the context finishes on the SPU, the context is destroyed to free the memory resources used for the context. The specific functions for SPU context management are described in [8]

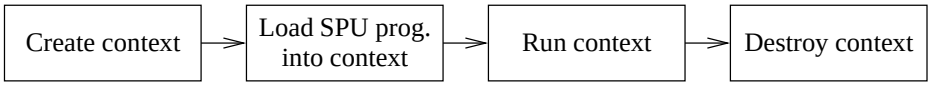


Figure 4.3: *The flow of running a single SPU program context.*

Parallel Programming by POSIX Threads

When running a SPU context, the calling program will lock until execution of the context finishes. Therefore, only one context may run in a single process. To enable simultaneous execution, and thereby execution on multiple SPUs, the programming environment supports the use of POSIX threads (pthreads) [14]. The principal flow of multithreaded context execution is shown in figure 4.4.

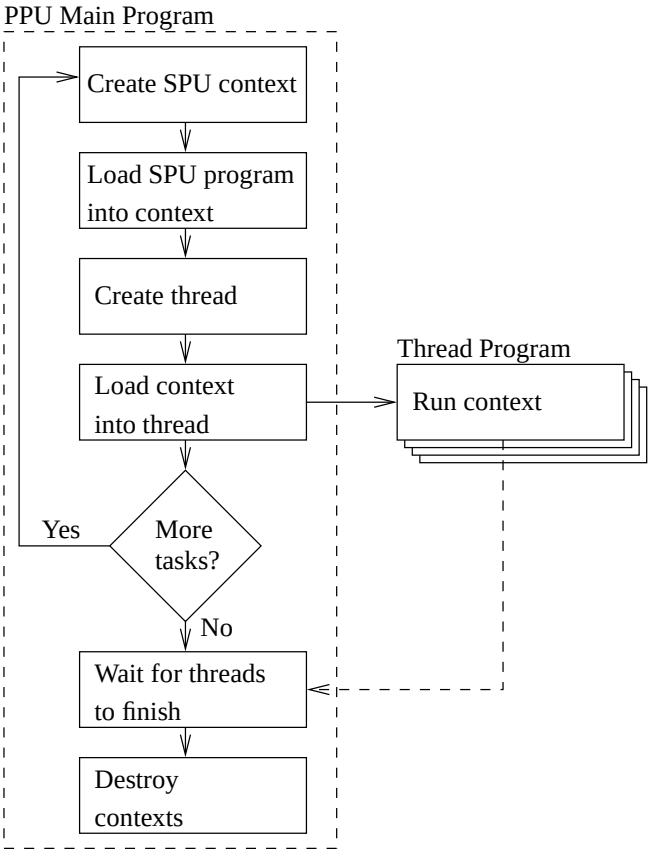


Figure 4.4: *Principle of executing SPU contexts concurrently in threads.*

Each context that may be run simultaneously is loaded into a thread. The thread program issues a request for the context to be run, and is then locked until the context finishes execution. The locking of a thread does not influence execution of the main program, and thus as many threads as needed may be created. When all SPUs are busy, threads will queue up and be executed in the same order as they were created.

Finally when threads finish and return, the main program must handle any return arguments and destroy the no longer needed SPU contexts.

4.7.3 Source Code Structure

In order to maintain a homogeneous program structure, the source code is arranged as shown in figure 4.5. As described in section 4.7.2 separate programs need to be developed for PPU and SPU execution. To comply with this basic structure, the code structuring of the developed programs is divided into PPU and SPU programs as well. Furthermore a folder for SPU tools, generic SPU subfunctions e.g. timing tools, is allocated.

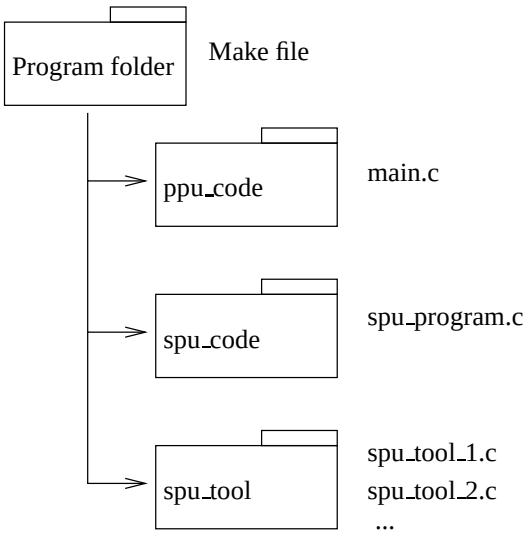


Figure 4.5: *Main structure of CBE program code. A program folder consists of 3 main folders containing PPU program code (ppu_code), SPU program code (spu_code), and tools for inclusion in SPU programs (spu_tool).*

4.7.4 Program Compilation

Translating the developed C code for execution on the CBE consists of several steps as shown in figure 4.6 based on the code structure presented in section 4.7.3.

First all .c files are compiled using *ppu-gcc* for PPU programs and *spu-gcc* for SPU programs and tools. Next SPU tools are included in the SPU programs to create SPU executables using *spu-gcc*. These executables are embedded into the PPU programs by first creating embedded PPU images of the SPU executables (using *ppu-embedspu*), next creating PPU libraries (using *ar*), and finally compiling the PPU programs again with the generated libraries to obtain the CBE program (using *ppu-gcc*).

To preserve this framework for program compilation a Makefile has been written which compiles the developed C code into a CBE program when the program structure obeys the structure presented in section 4.7.3.

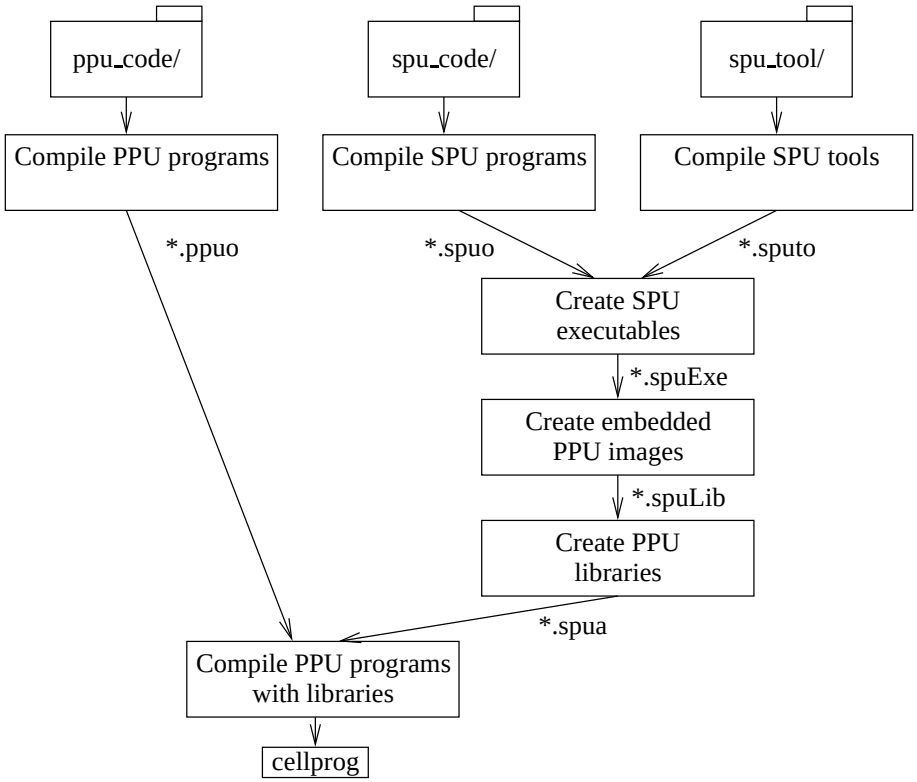


Figure 4.6: *Flow for CBE program compilation. First .c source files for PPU programs, SPU programs, and SPU tools are compiled using* ppu-gcc *and* spu-gcc*, respectively. SPU tools are integrated into SPU programs to create SPU executables, which are compiled through embedded PPU images into PPU libraries and finally compiled into the PPU program.*

Part II

System Design

The objective of this part is the synthesis of the algorithm from MatLAB to the Cell Broadband Engine. Looking at the design trajectory from the previous part, this corresponds to the algorithmic and architectural abstraction levels. This divides the development in intermediate steps between the theoretic MatLAB model of the previous part and an actual implementation.

The part starts with a set of principal experiments to gain understanding of the principles for programming efficiently for the CBE. Next a partition of the algorithm into tasks and design of PPU and SPU program is conducted. Lastly a mapping concerning specialized aspects of the SPUs are performed.

Contents

Chapter 5

CBE Programming Experiments

To gain initial experience and insight with the CBE architecture, two experiments is performed prior to the algorithm mapping and implementation. The first experiment concerns DMA transfers, with special regards to the performance and tradeoffs of double buffering. The second experiment concerns the performance when doing multiplications, with and without SIMD instructions and predefined loop counts.

5.1 DMA transfers

One method to ensure efficient utilization of the SPUs are to ensure concurrent DMA data transfers and computation in the SPU programs. This section present an experiment to evaluate the performance improvements which may be achieved from making use of the concurrency offered by the SPUs.

5.1.1 Double Buffering on the CBE

When transferring data by use of the SPU MFC unit for DMA transfers, the SPU data processing units may process data concurrently with the transfer, as the data are transferred by switching between two buffers during the transfer and allowing processing of the buffer which is not being filled. This double buffering principle is shown in figure 5.1.

Double buffering is achieved by use of MFC Tag Groups [9, p. 513]. Each MFC (DMA) transfer request may be assigned to a specified tag group regardless of the order in which the transfers are requested. By assigning every transfer request to B_0 to tag group 0, and transfers to B_1 to group 1 it is possible to wait for the transfer to a specific buffer to finish by polling the tag group status. This ensures that the processing of a buffer does not begin before the transfer to the buffer is complete. An optimal implementation ensures that data for the SPUs is ready when needed and not before, otherwise the SPU must wait until the data is ready before continuing the calculations.

As mentioned double buffering allows for data transfers and calculations to be performed concurrently. Figure 5.2 shows the principal benefit of implementing double buffering, where

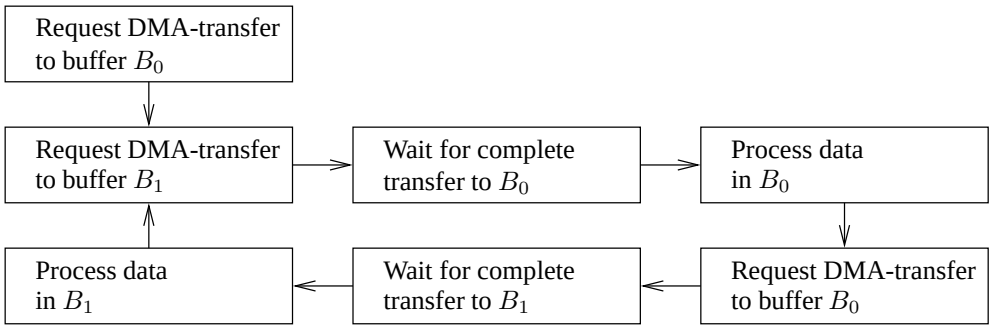


Figure 5.1: *Principle of double buffering. Graphics from [9, p. 685]. The goal is to perform the calculations in small steps concurrent with the transfer of the next data to be processed. In this way no delay in waiting for the data transfer to complete is incurred.*

the achieved concurrency results in faster execution (t_{Tm}) of the application as compared to waiting for all data to be transferred before executing calculations (t_{T1}).

It is clear, that the smaller the buffers are, and thus the DMA transfers, the earlier the first buffer will fill and calculations using these data may start. However, for each switch of buffers time is spent on requesting a new DMA transfer (t_{ds}) and starting up the calculation algorithm (t_{cs}) which will counteract the time saving feature of implementing a double buffering technique. The optimal buffer size is therefore highly dependent on the characteristic times for each problem and implementation of solution.

5.1.2 Experiments with Double Buffering

The experiments performed for evaluating the double buffering principle, tests the application of calculating a vector dot product while transferring data using double buffering. The purpose is firstly to examine the effects of double buffering versus single buffering. Secondly the experiment aims to find the optimal buffer size for this vector dot product algorithm, which is significant for the descrambling and despreading application of the project worked example. The source code for the experiments is found in [1, (C2), *spu_code/vecprod.c*].

5.1.3 Results

Figure 5.3 shows the measured execution times using the SPU decrementer register [9, p. 386] for a set of different DMA transfer sizes. In all experiments the total amount of data to be transferred is 16kB, or 4096 single precision floating point values. The test case with a transfer size of 4096 floats thus corresponds to single buffer execution of the program. The green line shows the total time spent on calculating the vector product, the red line is time spent waiting for DMA transfers to finish and the blue line is the total program execution times.

Figure 5.4 show the same experiment conducted at a total data length of $38 \cdot 4096$ floats, a vector length comparable to the length of $\bar{\mathbf{r}}'$. This experiment is conducted with both 1 and 6 active SPUs to test if the activation of multiple processor units affects the results due to increased

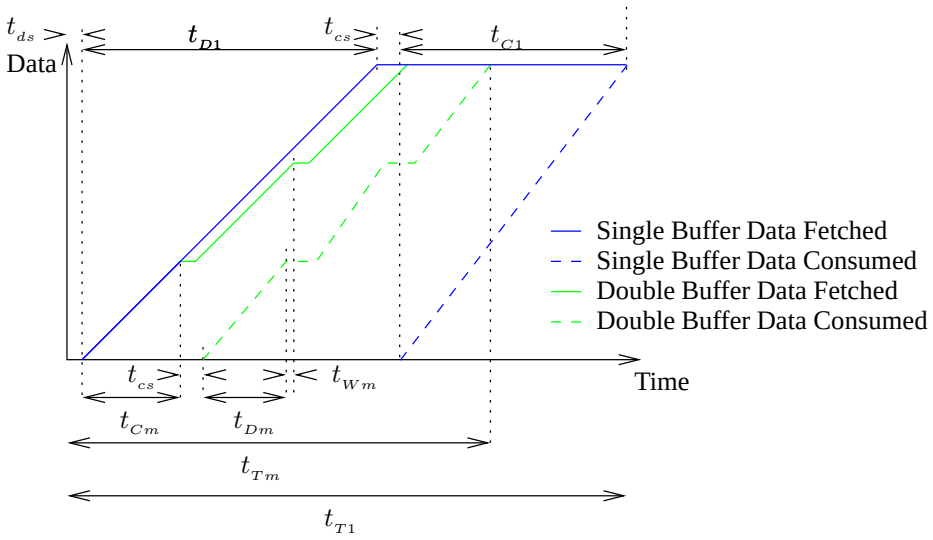


Figure 5.2: *Principal comparison of execution time spent when implementing single- and double-buffering techniques. It is seen that the dashed line of double buffering (t_{Tm}) completes faster than the dashed line of single buffering (t_{T1}), thus achieving an overall faster execution time. The time it takes to issue a new DMA request is t_{ds} and calculation algorithm startup is t_{cs}. The time t_{Wm} signifies the difference if the calculation or the transfer must wait for the other. The total transfer time of the problem is t_{D1} and the time it takes to calculate the entire problem in one iteration is t_{C1}.*

usage of the EIB. For the multiple SPU scenario each SPU calculates a full length vector product and the time measured is the mean time across the active SPUs.

5.1.4 Discussion

From the results presented in figure 5.3, it is clear that partitioning the data transfers into several buffers has positive effects on the total execution time of a vector product. Compared to the single buffer scenario with an total execution time of 15.560 clock cycles, an optimum is achieved with a transfer size of 1024 floats and an execution time of 10.880 cycles yielding a improvement of 30%. From this point the time used for setting up calculation loops and data transfers increase fast with the decreasing transfer size.

The second experiment with longer vectors multiplied show first that running multiple SPUs does not affect the mean execution time of the SPU program, indicating that DMA transfers are conducted fast enough to prevent SPU starvation, which would lead to a increase execution times. With 6 SPUs running the mean total execution time is minimized when using the maximum buffer size of 16kB (4096 floats), where this time is measured to 305.240 cycles. The conclusion of these principal experiments is thus that double buffering is definitely beneficial for the total

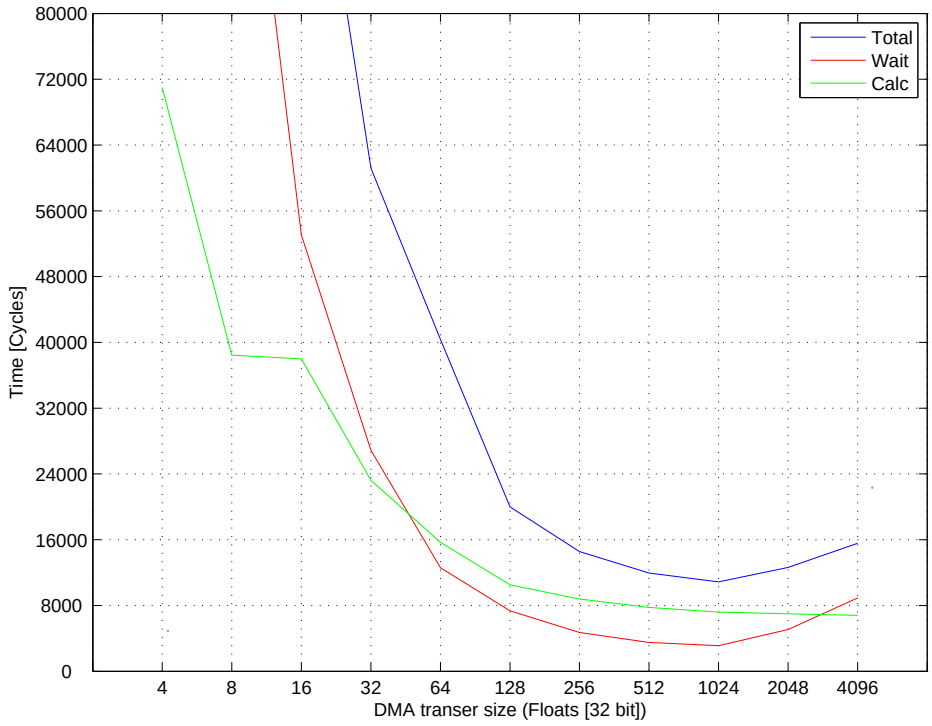


Figure 5.3: *Simulation results of time spent waiting for buffers to fill, calculating vector products and total execution time, when using different buffer sizes and double buffering*

execution time of an algorithm, but the transfer sizes should be kept relatively large. The results of the test with multiplying a longer set of vector show that when processing larger amounts of data, the DMA transfer size should be kept as large as possible.

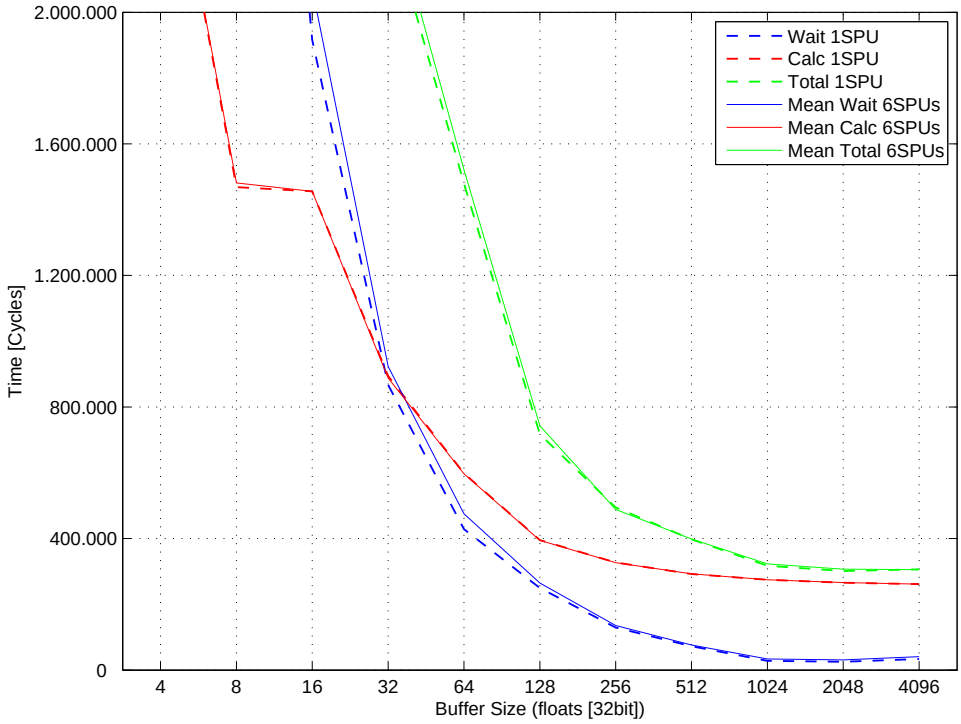


Figure 5.4: *Simulation results of time spent waiting for buffers to fill, calculating vector product and total execution time for calculating long vector dot product with different buffer sizes and 1 or 6 active SPUs.*

5.2 Scalar and SIMD Multiplication

This experiment tests the SPUs with regards to performance and capabilities in performing multiplications. This test is performed to give an initial measurement of the SPU performance and determine, if there exists an upper bound for the number of calculations which can be performed in a loop, with regards to storage and register usage.

The multiplications are performed on single precision floating point (32 bit), as opposed to integers as this yields an easier implementation with regards to simulation comparison etc. Furthermore the CBE does only support pipelined integer multiplies of 16 bit [9, p. 708].

5.2.1 Theoretical Limit

An experiment with multiplications can be designed in many ways and the output will vary accordingly. The author of [5] has demonstrated an implementation close to one SIMD multiplication per instruction, which is the theoretical maximum. This is done in hand coded assembler working on matrix-multiplications. This implementation heavily exploits data reuse and is possible because the matrix is divided into 64×64 tiles and processed in parallel on each SPU.

To determine the maximum performance for a SPU, it is seen that the SPU accesses the local store via pipeline 1, as shown in figure 4.2 on page 34. If multiplying two independent vectors from LS and saving the result back to LS incurs 3 instructions in pipeline 1, two quadword loads and a quadword store. The multiplication is done in pipeline 0 and in parallel with the operations in pipeline 1. The result for this scenario is a theoretical performance of 3 instructions per SIMD multiplication assuming perfect loop unrolling, scheduling and no prolog and epilogue overhead.

The rate of multiplications is thus highly dependent on the problem and the partition of the problem. For this test two independent vectors are assumed.

5.2.2 Test setup

The test is performed on both linear- and SIMD-code, with and without compile-time definition of loop trip count. A variable number of multiplications between two independent number sequences is performed on a single SPU.

Random numbers are transferred to the SPU LS and multiplied after the transfers are complete. Code for the linear multiplication is seen in figure 5.5 on the next page. Code for the SIMD multiplication is found in figure 5.6 on page 50.

The code is compiled with the optimization flag -O3, causing the compiler to perform all optimizations. Time is measured using the SPU decrementer register via the *prof_{clear, start, stop}* function calls [9, p. 386].

See [1, (C1), *spu_code/mult.c*] for a complete view of the used code.

5.2.3 Results

A graph of the results is seen in figure 5.7 on page 51. On average the linear implementation yields approximately 22 cycles per multiplication, while the SIMD implementation yields 3.5 cycles per single multiplication. Timing information for the assembler output can be found at [1, (C3) and (C4)] and for SIMD as a listing in figure 5.8 on page 52.

Examining a single sample where 20480 mutliplications has been performed in 73040 cycles, it is seen that this yields a throughput of 897 MFLOPS for a single SPU.

5.2.4 Discussion

It is seen that the SIMD capabilities should be used whenever possible and that a constant trip-count does not have any impact in these experiments. The experiment also reveals that the default compiler output does not yield optimal scheduling. In the SIMD case cycles are spent waiting for needed operands to propagate through the pipeline. The achieved SIMD result is approximately 4 times slower than the theoretical result discussed in section 5.2.1 on the previous page. The main problem is the pipeline stalls, which occur in 65% of the cycles, which could be remedied by manual unrolling and scheduling rearrangement as in [15, p. 15].

To further improve the results the problem must be examined for possible data reuse, which can lower the amount of loads and stores which are needed in pipeline 1.

```
1   /* Common for both cases */
2   float * input1, * input2, * output;
3   int i, count;
4
5   /* With compile time trip count */
6   prof_clear();
7   prof_start();
8   for(i = 0; i < DATALEN*OVERLOAD; i++){
9     *(output+i) =
10          (*(input1 + i)) * (*(input2 + i ));
11  }
12  prof_stop();
13
14  /* Without compile time trip count */
15  prof_clear();
16  prof_start();
17  for(i = 0; i < count; i++){
18    *(output+i) =
19          (*(input1 + i)) * (*(input2 + i ));
20  }
21  prof_stop();
```

Figure 5.5: *C code for linear multiply with and without compile time trip count executed on the SPU. The prof_ { start,stop,clear } commands measure the time in cycles executed on the SPU. In the latter case without compile time trip count, the "count" is transferred as part of a DMA command from the PPU.*

With the experiments concluded the algorithm is partitioned in the next chapter and further software design and architecture mapping is performed.

```
1   /* Common for both cases */
2   vector float temp0, temp1, temp2, temp3;
3   vector float in1a, in1b, in1c, in1d, in2a, in2b, in2c, in2d;
4   vector float * varray1 __attribute__ ((aligned (16)));
5   vector float * varray2 __attribute__ ((aligned (16)));
6   vector float * vout __attribute__ ((aligned (16)));
7   varray1 = (vector float *) (input1);
8   varray2 = (vector float *) (input2);
9   vout    = (vector float *) (output);
10
11  /* With compile time trip count */
12  prof_clear();
13  prof_start();
14  for(i = 0; i < DATALEN*CHUNKS/4; i+=4){
15        vout[i] = spu_mul(varray1[i], varray2[i]);
16  }
17  prof_stop();
18
19  /* Without compile time trip count */
20  prof_clear();
21  prof_start();
22  ii = (arg.datalen * arg.dataChunks)/4;
23  for(i = 0; i < ii; i++){
24        vout[i] = spu_mul(varray1[i], varray2[i]);
25  }
26  prof_stop();
```

Figure 5.6: *C code for SIMD multiply with and without compile time trip count executed on the SPU. The* prof_ start,stop,clear *commands measure the time in cycles executed on the SPU. In the latter case without compile time trip count, the* arg*-structure is transferred as part of a DMA command from the PPU. The* spu_ mul *command is an intrinsic which maps directly to the assembler instruction* fm*, but does not otherwise confer any optimizations, it only forces the compiler to use the* fm *instruction.*

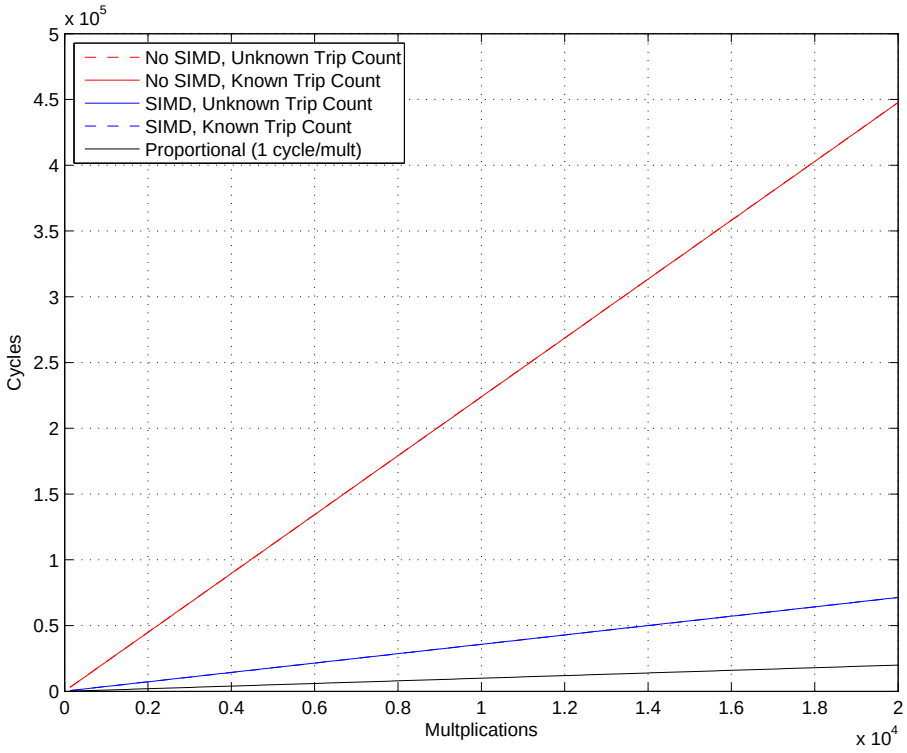


Figure 5.7: *Results of multiplications test. The presence of compile time information about loop count has no effect on the performance, while SIMD instruction usage has a noticeable impact as expected. Both cases have overlapped graphs for known/unknown trip count.*

```
1  000154 0D      45                                                    ai
2  000154 1D      456789                                                lqx
3  000155 0D       56                                                   ai
4  000155 1D       5                                                    hbrp
5  000156 1         678901                                             lqx
6  000157 0         78                                                 ai
7  000158 0          89                                                ai
8  000159 0           90                                               ai
9  000160 0            01                                              ai
10 000161 0              12                                            ai
11 000162 0               234567                                       fm
12 000163 0                34                                        ai
13 000164 0d                4                                        nop
14 000168 1d                ----890123                               stqx
15 000169 0D                    90                                   ai
16 000169 1D                    901234                               lqx
17 000170 1                      012345                              lqx
18 000171 1                       1                                  hbrp
19 000176 0d                       ----678901                        fm
20 000182 1d                           ------234567                  stqx
21 000183 1                                345678                   lqx
22 000184 1                                 456789                  lqx
23 000190 0d                                 -----012345             fm
24 000196 1d 01                                    ------6789        stqx
25 000197 1  012                                         789         lqx
26 000198 1  0123                                         89         lqx
27 000204 0d ----456789                                    -         fm
28 000210 1d      ------012345                                       stqx
29 000211 1            123456                                      lqx
30 000212 1             234567                                     lqx
31 000218 0d             -----890123                               fm
32 000224 1d                  ------456789                         stqx
33 000225 1                         567890                        lqx
34 000226 1                          678901                       lqx
35 000232 0d                          -----234567                  fm
36 000238 1d                                ------890123           stqx
37 000239 1                                       901234           lqx
38 000240 1                                        012345          lqx
39 000246 0d 01                                     -----6789       fm
40 000252 1d --234567                                      ----     stqx
41 000253 1    345678                                              lqx
42 000254 1     456789                                             lqx
43 000260 0d     -----012345                                       fm
44 000266 1d          ------678901                                  stqx
45 000267 0D                7                                     nop
```

Figure 5.8: *Timing overview of unrolled SIMD multiplication loop kernel from figure 5.6 on page 50 or [1, (C4)]. In line 1 the first number is the cycle count assuming straight execution, the next is pipeline number followed by dual issue (D) or not dual issued because of dependency stall (d). Afterwards is a pipeline visualization where '-' is a pipeline stall due to dependency. It is seen that this code has pipeline stalls in approximately* 65% *of the cycles, which could be remedied by manual instruction reordering as in [15, p. 15].*

Chapter 6

Algorithm Partitioning

In this chapter, the partitioning of the descrambling and despreading calculations, derived in section 3.9 on page 23, into program tasks is discussed to derive a structure for the PPU and SPU programs. Later the results of the experiments of the previous chapters are used in determinatin of transfer buffer sizes and multiplication setup.

6.1 Calculations Partitioning

As examined in the signal model, section 3.7 on page 19, the demodulation of the received signal $\bar{\mathbf{r}}$ is performed as:

$$\hat{\mathbf{d}} = \bar{\bar{\mathbf{C}}}^{\mathrm{H}}\bar{\bar{\mathbf{O}}}^{\mathrm{H}}\bar{\mathbf{r}} \tag{6.1}$$

where $\bar{\mathbf{r}}$ is the received signal, $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ contains the spreading and scrambling sequences and $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ contains the channel effects and multipath information.

6.1.1 Left or Right Matrix Multiplication

Calculation of (6.1) can be done either left to right, or right to left, which will produce different intermediate products. The efficiency of the calculation will be highly dependent on these products, so an evaluation is required.

Examining the left to right multiplication where $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ is multiplied with $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ shown in figure 6.1 on the following page, it is seen that this will produce a scaling channel-filtering of the spreading and scrambling sequences. This matrix will contain a band with zeroes outside the band. The width of the band is given by the asynchronous multipath delays D in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ and thus data dependent. This introduces uncertainty in the calculation of $\bar{\bar{\mathbf{O}}}' \cdot \bar{\mathbf{r}}$ as multiply by zeroes can occur, which is a waste of processing power.

Performing the multiplication as right to left and thus multiplying $\bar{\mathbf{r}}$ onto $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ will produce a column vector. This intermediate vector $\bar{\mathbf{r}}'$ is then multiplied onto $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$. This approach is very predictive as the structure of $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ and $\bar{\mathbf{r}}'$ is well known.

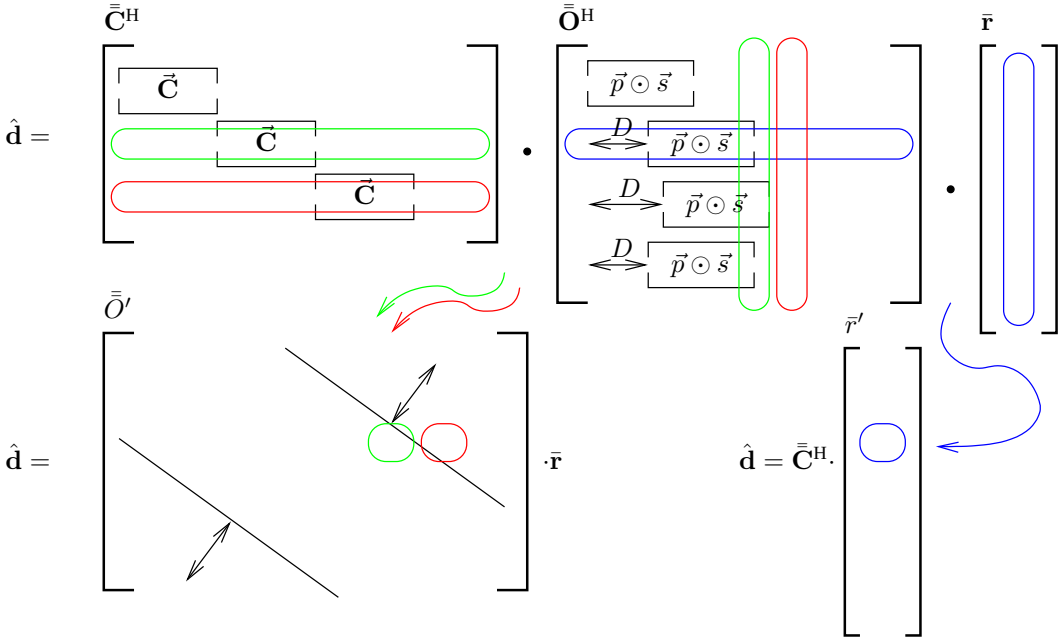


Figure 6.1: *Example of left to right and right to left multiplication with excerpts from the matrices in (6.1). It is seen that the left to right multiplication will produce a variable-band matrix, depending on the delays D. The right to left multiplication produces column vector.*

The intermediate product with the highest efficiency potential is the right to left multiplication which produces the single column vector $\bar{\mathbf{r}}'$. This column vector will have a precalculable length and is thus much more predictable than the left to right multiplication which produces a matrix with a data-dependent number of off-band zeros. Thus the right to left multiplication is selected due to ease of addressing (predictability) which will lead to less multiplications by zero or data look-ups for delays.

6.1.2 Partitioning in Time or Users

Given the limited size of the SPU local store, it is not possible to keep the entire problem in the LS of the SPUs, see section 3.9.4 on page 26. It is therefore necessary to partition the algorithm further. The two partition types which are examined is that of partitioning with regards to time, thus the received signal $\bar{\mathbf{r}}$, or assume all data is known and divide the problem into K partitions, one for each user.

Figure 6.2 on the facing page shows the computation of (6.1) with received symbols $\bar{\mathbf{r}}$ (time) as a variable parameter. Calculating a given symbol for a given user is essentially the same for partition in time or users, the difference lies in the number of context-switches between users each

SPU must perform. If partitioning in time is used, jobs will be issued to the SPUs for each time a new data-segment is received, as shown in figure 6.3 on the next page part (a), assuming that SPUs are issued different users for each iteration. This approach differs from the user partition, part (b) in figure 6.3, in that each time a new user is assigned to a SPU, the spreading and scrambling vectors must be calculated or transferred to the SPU.

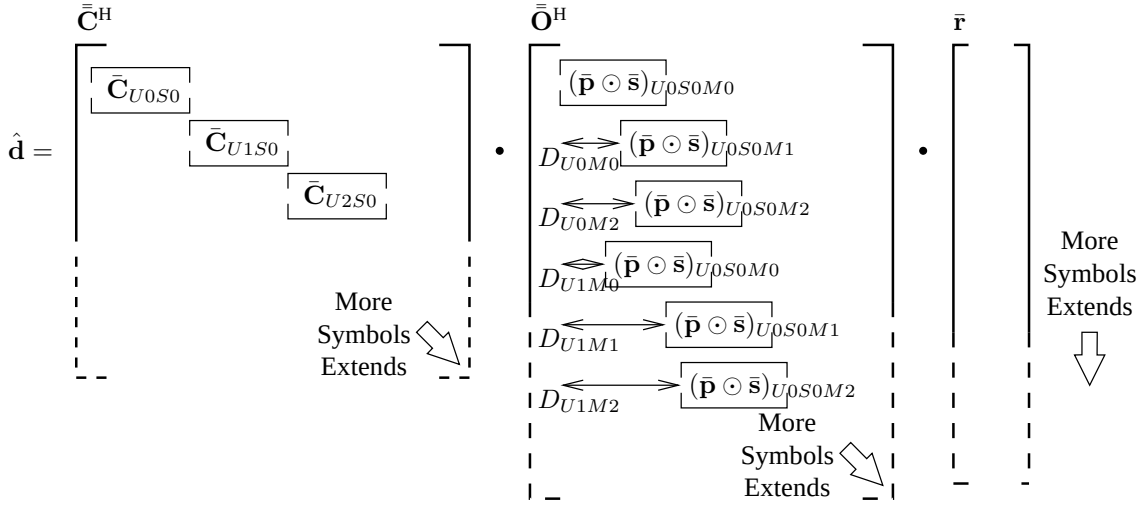


Figure 6.2: *Depiction of* $\hat{\mathbf{d}}$ *estimation with variable number of received symbols in* $\bar{\mathbf{r}}$.

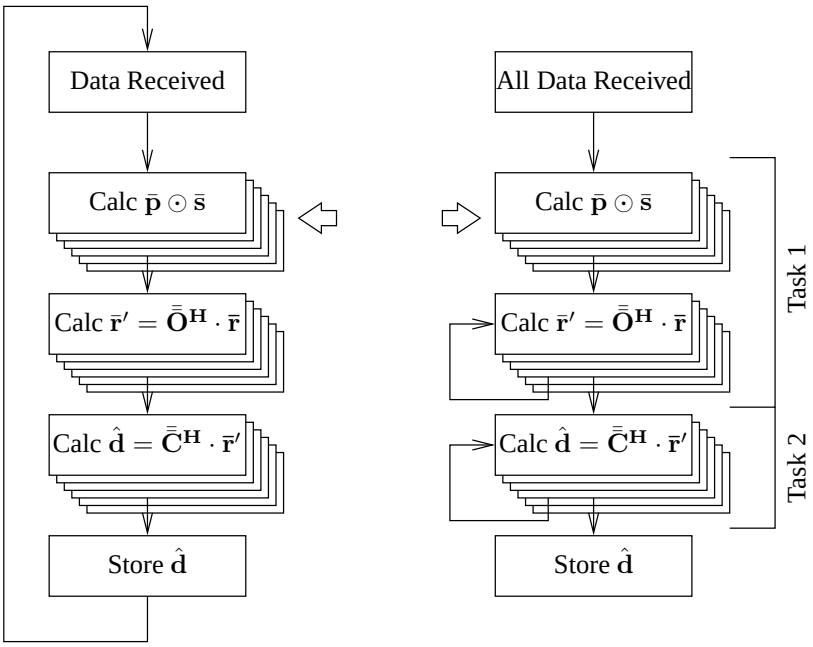
Selecting a partitioning scheme thus depends on how the spreading and scrambling generation are performed and the priority with regards to the hard-time requirements. Ideally these limited and well defined tasks [2, p. 18 and p. 22] would be performed in dedicated hardware, but this is not in the scope of this project. With this in mind, and to avoid unnecessary calculations on the SPUs, the partition is selected to be on a per user basis. All data is thus assumed available for each 10 ms transmission burst.

This is not ideal for speech-transmission, which mobile phone tends to do, as it introduces a constant delay of 10 ms in the base station. But the partition in users is still selected as it furthermore will yield a simpler implementation and still be usable for non-time critical transfers (HTTP data etc.).

The algorithm is divided into independent tasks as shown in figure 6.3. Task 1 consists of calculating spreading and scrambling sequences for a specific user on the SPU, and later using this calculated sequence in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ to determine $\bar{\mathbf{r}}'$. Task 2 is that of calculating the estimated received symbols $\hat{\mathbf{d}}$ from the channel coefficients (given as data) and $\bar{\mathbf{r}}'$.

6.2 Buffer Size Estimates

To solve the demodulation problem for a user at a time, while still achieving efficient usage of each SPU, double buffering must be employed as shown in section 5.1 on page 43. The same



(a) Time Division (b) User Division

Figure 6.3: *Differences between time division (a) and user division (b). The broad arrows show the difference, namely that the spreading and scrambling vectors must be calculated for each iteration in time division (a), whereas it can be reused for the entire execution in user division (b). The algorithm is furthermore divided into task 1 and task 2 as shown, where the thin arrows signifies fetch of new data. Superpositioned boxes signifies parallel execution on the SPUs.*

experiment shows that the DMA transfer size can have a significant impact on the completion time. Thus an optimal transfer size must be determined.

In this regard the project group used the potential erroneous approach of fitting the transfer sizes to an easily implementable value. Thus the maximum transfer size of 16 KB per DMA issue was used, which will be described in the two following sections. To achieve optimal performance the transfer sizes should be evaluated when designing the system. A transfer size which should yield better results is estimated at the end of this section.

6.2.1 Task 1

To compute 1 output (1 complex value in $\bar{\mathbf{r}}'$) inputs of 1 delay per multipath (16 delays of 32 bit integers), 128 values of $\bar{\mathbf{r}}$ (32 bit complex floats) and 128 values in the spreading and scrambling sequence are needed (128 32 bit complex floats). When calculating task 1 for one user, the delays can be reused and the transfer size thus depends on how many symbols can be executed per transfer. As a delay can have a maximum of S (128) samples, according to model delimitations

on 3.8.3 on page 21, the transferred number of symbols (X) must be $(X+1) \cdot S \cdot 4 < 16000 \Leftrightarrow X < \frac{16000}{S \cdot 4} - 1 \Leftrightarrow X < 30.25$ thus a value of 30 symbols in $\bar{\mathbf{r}}'$ is selected. With 300 symbols per user, this must be performed 10 times per user.

With complex numbers in $\bar{\mathbf{r}}$ two transfers of $(30+1) \cdot S$ symbols are initialized to the SPU, while 30 values in $\bar{\mathbf{r}}'$ is transferred back to the PPU per iteration. The values in $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}$ are calculated once for each user on the SPU, and reused across the calculations of all symbols in $\bar{\mathbf{r}}'$ for the user. No transfer of this is thus needed.

6.2.2 Task 2

The transfer size for each iteration in task 2 is selected to make 3 iterations and calculate 100 symbols in $\hat{\mathbf{d}}$ for each iteration. Thus one execution of task 2 calculates all $\hat{\mathbf{d}}$ for one user. To perform this calculation the channel gain coefficients $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ and values of $\bar{\mathbf{r}}'$ must be transferred to the SPU.

This yields transfers of 100 times the multipaths (M) complex 32bit floats for $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ and likewise for $\bar{\mathbf{r}}'$. Thus 4 transfers of 6400 bytes are issued for each iteration.

6.2.3 Optimal Buffer Size Estimate

Estimating the optimal transfer size can be a complex task, and should ultimately be tested for each specific task. Something that the project group did not perform. A rather light approach for estimation is used here, based on assumptions and the results of the double buffer experiments in section 5.1 on page 43.

The double buffer size depends on the following factors:

- Number of active SPUs and thus the bandwidth available for each SPU
- The characteristics of the task with regards to consumption and production of data on the SPU

Assuming 6 active SPUs the bandwidth becomes $\frac{96 \frac{\text{byte}}{\text{cycle}}}{6} = 16 \frac{\text{byte}}{\text{cycle}}$ for each SPU, with a EIB bandwidth of $96 \frac{\text{byte}}{\text{cycle}}$ according to section 4.2 on page 31. The theoretical maximum rate for a multiply and accumulate (MAC) operation is 2 instructions per MAC operation in SIMD mode. The transfer bandwidth, discussed in section 4.5 on page 35, is thus theoretically big enough to keep a MAC operation running with 2 different operands and still transfer extra data.

According to section 5.1 on page 43 a buffer size between 1024 and 4096 transferred real floats is optimal. See figure 5.4 on page 47. The lowest value of 1024 floats removes some of the risk of SPU starvation, but lowers the loop trip count which can have an adverse effect on the performance. The larger buffer size of 4096 transferred floats improves the trip count but imposes the risk of SPU starvation.

This buffer size is much lower than the one used in task 1 and 2, which issues several 16 KB transfers per iteration. This could yield SPU starvation, but is not tested.

The buffer size estimation is thus very complex and changes characteristics with both the problem and with different implementations of the solution. Testing with different buffer sizes to find the optimal performance has not been performed.

Chapter 7

Software Design

This chapter explains the design of the PPU and SPU software implementation of the CDMA descrambling and -spreading operations partitioned into Task 1 and Task 2 in the previous chapter.

All program synthesis is implemented in C with no use of programming framework other than what is presented in the platform analysis, section 4.7.3 on page 38.

7.1 PPU Program Design

The PPU program acts as a controller of tasks for the SPUs. The proposed design and structure of this controller is presented in the following.

7.1.1 PPU Program Structure

The role of the PPU is to:

- Present needed data to SPUs
- Issue new tasks to SPUs which finishes
- Maintain synchronization in algorithm execution

This functionality is implemented in a state machine which is illustrated in figure 7.1 on the following page. The main parts of the PPU program are found in [1, (B1), *ppu_code/main.c* and *ppu_code/controller.c*]. Each of the states are detailed here with function names and state name for identification in the source code:

- Init: This state initializes all the SPUs and loads all data from file storage to the memory of the PPU. (*main()*)
- Task 1: Available SPUs are assigned a user k and calculates $\bar{\mathbf{r}}'_k = \bar{\bar{\mathbf{O}}}_k^{\mathrm{H}} \cdot \bar{\mathbf{r}}$. Tasks are scheduled using a dynamic, work requester scheme, where the SPUs are assigned a job (user) upon request by sending a ready signal to the PPU. The communication between

PPU and SPU processes necessary to implement this scheduling scheme is elaborated in section 8.1 on page 65. (*stateMachine()*, case *TASK_OR_PPUSTATE*)

- Synchronize 1: Wait until all SPUs are finished with their issued Task 1, to avoid data corruption as the data for Task 2 is dependent on $\bar{\mathbf{r}}'_k$ from Task 1. (*stateMachine()*, case *AWAIT_OR_COMPLETION_STATE*)
- Task 2: Issue Task 2 to available SPUs. Available SPUs are assigned a user k and calculates $\hat{\mathbf{d}}_k = \bar{\bar{\mathbf{C}}}_k^{\mathrm{H}} \cdot \bar{\mathbf{r}}'_k$. Tasks are assigned using the same scheme as for task 1. (*stateMacine()*, case *TASKCR_PPU_STATE*)
- Synchronize 2: Wait until all SPUs are finished with their issued Task 2. When this synchronization is done the calculation of $\hat{\mathbf{d}} = \bar{\bar{\mathbf{C}}}^{\mathrm{H}}\bar{\bar{\mathbf{O}}}^{\mathrm{H}}\bar{\mathbf{r}}$ is complete. (*stateMachine()*, case *TASK_DONE_PPU_STATE*)
- Verify Data: Verify that the calculated output of the algorithm is similar to the reference output from MatLAB. (*controllerInit()*)

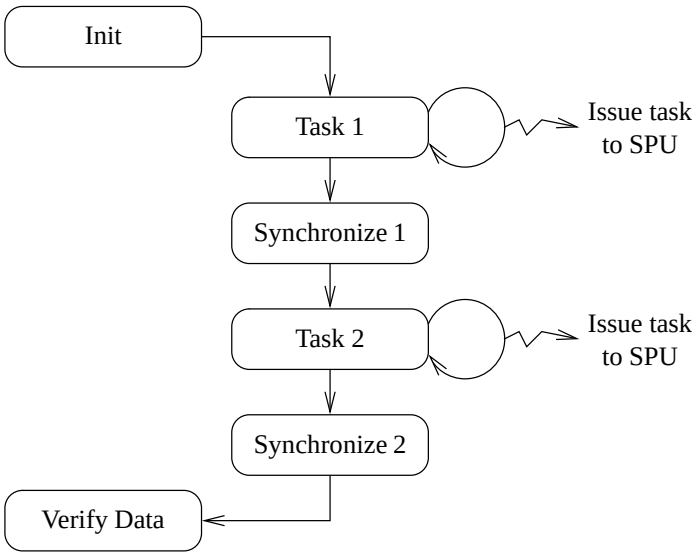


Figure 7.1: *State machine for the PPU program*

7.2 SPU Program Design

As explained in section 6.1.2 on page 54 any SPU may be assigned one of two tasks, either calculating the intermediate result $\bar{\mathbf{r}}'_k = \bar{\bar{\mathbf{O}}}_{\mathbf{k}}^{\mathbf{H}} \cdot \bar{\mathbf{r}}$ (task 1) or the data symbol estimate $\hat{\mathbf{d}}_k = \bar{\bar{\mathbf{C}}}_{\mathbf{k}}^{\mathbf{H}} \cdot \bar{\mathbf{r}}'_{\mathbf{k}}$ (task 2). This section presents the proposed structure of the SPU program found in [1, (B1), *spu_code/spuDecode.c*].

7.2.1 SPU Program Structure

The overall structure of the SPU program is shown in figure 7.2.

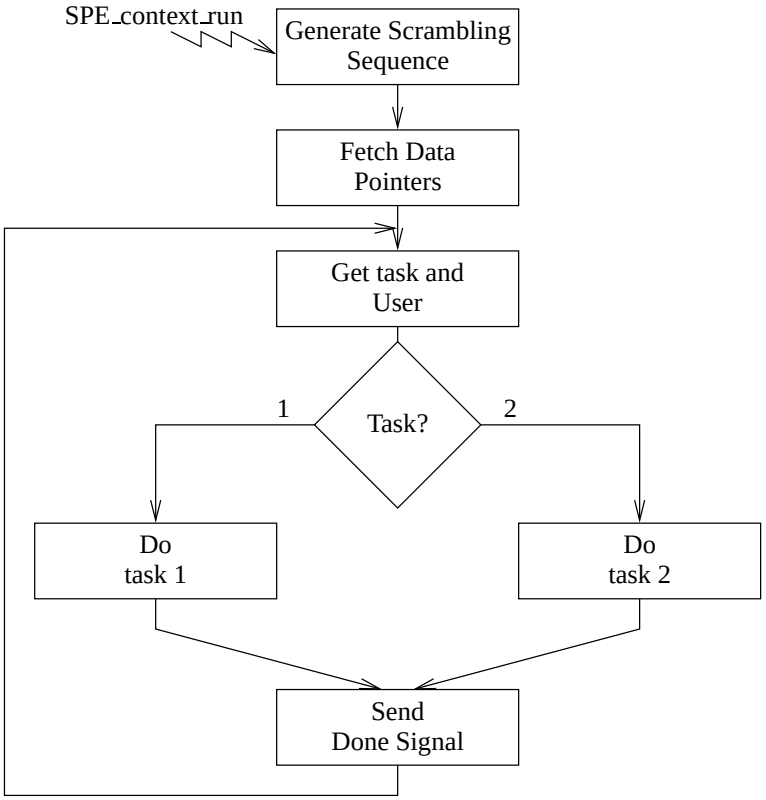


Figure 7.2: *SPU Program structure. Individual task flows are elaborated in figure 7.3.*

When the SPU context is launched a local copy of the BS dependent scrambling sequence, $\bar{\mathbf{p}}$, is generated using the short scramble sequence generator, presented in section 3.8.3. Next, the SPU sends a ready signal to the PPU and waits to be assigned an ID and to get information on the memory locations from where to get $\bar{\mathbf{r}}$, $\bar{\bar{\mathbf{C}}}$ and put $\bar{\mathbf{r}}'$ and $\hat{\mathbf{d}}$. Next, the SPU enters a main loop awaiting incoming task assignments. A switch is made on the task type to execute the appropriate task with a pointer to the a structure containing data memory locations, and the received user number as arguments. Upon completion of a task the SPU returns to waiting for the next task and user.

The structure of each task is based on the double buffering principle explained in section 5.1 to allow for concurrent data transfers and task execution. The structures of the two tasks are elaborated in figure 7.3.

Task 1, *int calcOr()*, is that of calculating the intermediate product $\bar{\mathbf{r}}'_k$ for a user, k. For initialization, the task issues a DMA request for the delays of each multipath for the specific user along with the request for the first batch of $\bar{\mathbf{r}}$. Before entering the double buffering loop, the

user specific spreading code is determined using the OVSF spread sequence scheme presented in section 3.8.2 and the $\bar{\mathbf{p}}_{\{0,1\}} \odot \bar{\mathbf{s}}_k$ sequences are calculated, where $\bar{\mathbf{p}}_0$ is the first S samples of the 256 long complex vector, $\bar{\mathbf{p}}$, and $\bar{\mathbf{p}}_1$ is the last S samples of $\bar{\mathbf{p}}$.

In the task loop DMA requests are issued to the inactive $\bar{\mathbf{r}}$ buffers before processing the active buffers. When processing is done, a DMA transfer to the PPU is requested for the active output buffers containing $\bar{\mathbf{r}}'_k$. Before each iteration a switch is is made between active and inactive buffers. When ending the task loop iteration in which the last DMA transfer of $\bar{\mathbf{r}}$ is requested, the loop terminated and the last set of buffers are processed and results returned before returning to the main SPU program.

Task 2, *int calcCr()*, has the same structure as task 1. However, all data to be processed in $\bar{\mathbf{r}}_k$ and $\bar{\mathbf{c}}_k$ are located and transferred from external memory, so no sequences need to be initialized before entering the double buffering routine of task 2.

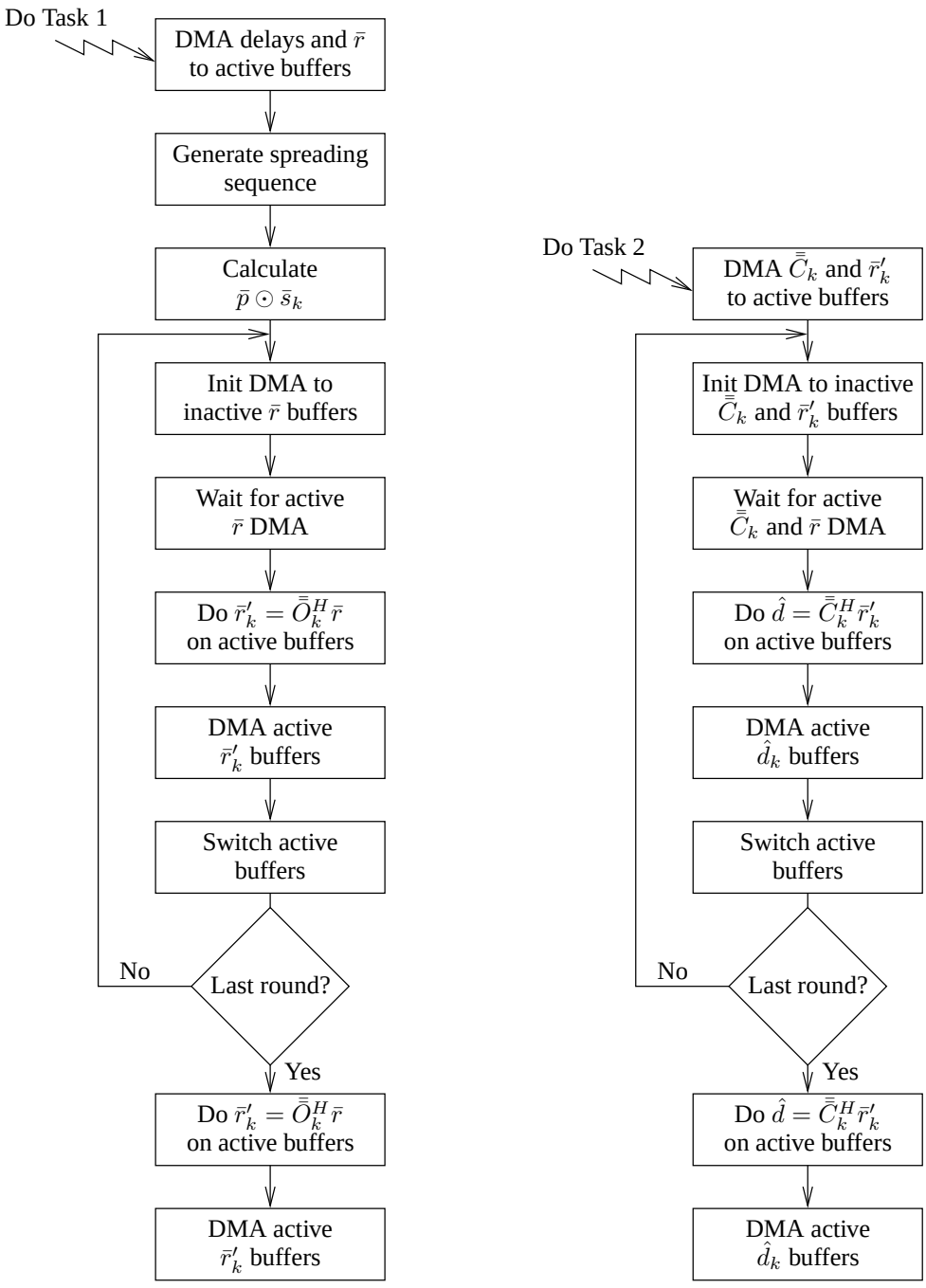


Figure 7.3: *Flow of SPU tasks for calculating* $\bar{\mathbf{r}}'_k$ *(task 1) and* $\hat{\mathbf{d}}_k$ *(task 2).*

Chapter 8

Architecture Mapping

This chapter covers the major tasks of mapping the PPU and SPU programs designed in the previous chapters to the CBE architecture. Relevant issues are the elaboration of the interprocess communication between the PPU and SPU programs, the SIMDization of the vector product algorithms, and handling the requirement for data alignment in memory to ensure correct functionality of DMA transfers and SIMD operations.

8.1 Interprocess Communication

The InterProcess Communication (IPC) of task assignments and status signals, described in the software design chapter 7, is implemented by message exchange through the four entry SPU Read Incoming Mailbox (SPU inbound), and the one entry SPU Write Outbound [9, p. 533]. The protocol for IPC between the PPU controller process and a single SPU process is shown in figure 8.1.

The first part of the IPC is an initialization phase, where the PPU assigns a identifier to a ready SPU and an Effective Address (EA) of a structure containing pointers to data spaces for $\bar{\mathbf{r}}$, $\bar{\mathbf{r}}'$, $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$, and $\hat{\mathbf{d}}$. When the SPU has transferred needed data, a ready signal is sent to the PPU to indicate ready for task assignment. The task assignment consists of two messages containing first the task type; DOOR_MSG or DOCR_MSG, for calculation of $\bar{\mathbf{r}}' = \bar{\bar{\mathbf{O}}}^{\mathrm{H}}\bar{\mathbf{r}}$ or $\hat{\mathbf{d}} = \bar{\bar{\mathbf{C}}}^{\mathrm{H}}\bar{\mathbf{r}}'$, respectively. The second message contain the user number to be processed and thus identifies which spread code and elements of $\bar{\bar{\mathbf{C}}}^{\mathrm{H}}$ to use for the issued task. To minimize the SPU wait time for receiving a new task upon completion, the PPU issues two tasks in the first round, thus filling the four entry SPU Read Inbound Mailbox. When a SPU signals task completion, ORDONE_MSG or CRDONE_MSG, the PPU may write the messages for the next task to the SPU mailbox.

As described in the PPU software design in section 7.1.1 on page 59 the PPU process performs a synchronization, idling unneeded SPUs, after issuing task 1 for the last user, to ensure all data for $\bar{\mathbf{r}}'$ is ready for processing before issuing any task 2. The subsequent IPC for task 2 follows the same protocol as shown for task 1 in figure 8.1 on the following page.

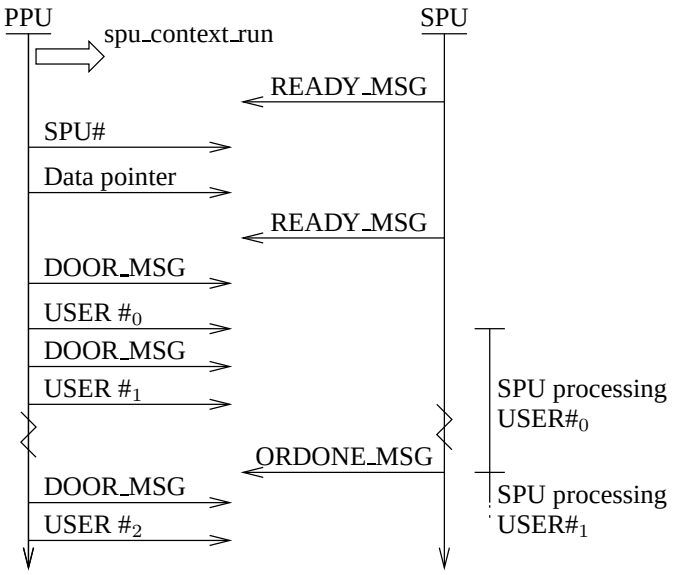


Figure 8.1: *PPU-SPU interprocess communication using mailboxes for message interchange. SPU initiated messages are transferred using the SPU Write Outbound Mailbox. PPU initiated messages are transferred through the SPU Read Inbound Mailbox.*

8.2 SIMD Mapping for Task 1

To enable the use of SIMD instructions when calculating the intermediate matrix product $\bar{\mathbf{r}}'$ (task 1) the algorithm needs to be modified to exploit the CBE vector data type intrinsic. The vector data type used, is a four element vector of single precision floating point values, thus covering 128 bit. When a vector is aligned to a 128bit memory boundary (a multiple of 10_h) an entire vector may be loaded in one instruction. Vectorized data arrays are achieved by assigning a vector pointer to the beginning of the source data array. Since data are already stored in floating point types the vectorizing is only a switch of reference and no data transformations are needed.

Early time measurements has shown that Task 1 takes the most time, it is thus the target of this optimization. Task 1 is the only task which is mapped to SIMD instructions for this project.

The expression to be vectorized is:

$$\bar{\mathbf{r}}'_{k,n,m} = \sum_{s=0}^{S-1} \left[\bar{\mathbf{p}}_{n\%2} \odot \bar{\mathbf{s}}_k\right](s) \cdot \bar{\mathbf{r}}(n \cdot S + \tau_{k,m} + s) \tag{8.1}$$

Rewritten to a four element vector form, equation (8.1) equals:

$$\begin{aligned} \bar{\mathbf{r}}'_{k,n,m} = \sum_{i=0}^{3} \Bigg(\sum_{s=0}^{S/4-1} & \left[\bar{\mathbf{p}}_{n\%2} \odot \bar{\mathbf{s}}_k\right]\{4s+0, 4s+1, 4s+2, 4s+3\} \ldots \\ \ldots \; & \odot \bar{\mathbf{r}}(n \cdot S + \tau_{k,m} + \{4s+0, 4s+1, 4s+2, 4s+3\}) \Bigg)\{i\} \end{aligned} \tag{8.2}$$

where the outer sum over i is across the result of the element wise vector product to end up with a single result.

The random start index of $\bar{\mathbf{r}}$ caused by the user and path specific delay, $\tau_{k,m}$, described in detail in section 3.6 on page 17, gives rise to a misalignment of the data to be processed, since this delay cannot be assumed to be a multiple of 4. Since moving data in memory is not desirable, four copies of the complex 256 entry $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}_k$ are stored in the SPU local storage, each with a shift of $i = \{0, 1, 2, 3\}$. A graphic representation of the four $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}_k$ vectors are shown in figure 8.2.

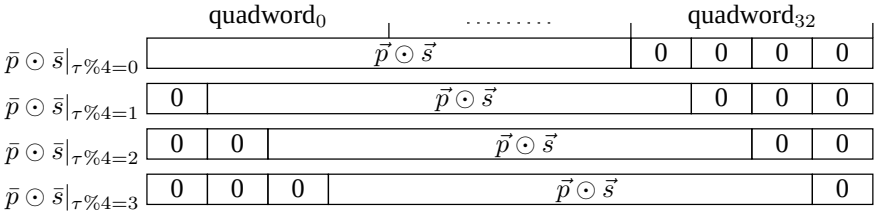


Figure 8.2: *Placement of four copies of $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}$ in memory for alignment with $\bar{\mathbf{r}}$ containing signals with random delays, $\tau_{k.m}$.*

To handle the extended $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}_k(\tau_{k,m})$ the expression in equation (8.2) is then rewritten to:

$$\begin{aligned} \bar{\mathbf{r}}'_{k,n,m} &= \sum_{i=0}^{3} \Big(\sum_{s=0}^{S/4} [\bar{\mathbf{p}}_{n\%2} \odot \bar{\mathbf{s}}_k] \,|_{\tau_{k,m}\%4} \{4s+0, 4s+1, 4s+2, 4s+3\} \ldots \\ &\ldots \quad \odot \bar{\mathbf{r}}(n \cdot S + \tau_{k,m} - (\tau_{k,m}\%4) + \{4s+0, 4s+1, 4s+2, 4s+3\}) \Big) \{i\} \end{aligned} \tag{8.3}$$

As a result of the 4 added elements to each of the $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}_k$ vectors, the expression in equation (8.3) requires 4 more multiply-accumulate operations for each of the 4 product-summations of the complex vector product, compared to the expression in equation (8.2). However, the result of the expression is maintained since the added operations have zeroes as results. The implementation of the SIMDized complex vector product to calculate a single element of $\bar{\mathbf{r}}'$ is shown in figure 8.3.

Implementations of task 1 using both the linear multiplication method of equation (8.1) and the SIMDized method is found in *calcOr()* in [1, (B1), *spu_code/spuDecode.c*].

8.3 Memory Assignment and Binding

To exploit SIMD operations features and enable DMA transfers between PPU and SPUs, data need to be aligned to 128bit (quadword) memory address boundaries. One way to ensure this is to statically define global arrays with an *aligned* attribute, which will align the first array element to an address boundary. However, this method for memory allocation results in inflexible and ineffective memory usage when an SPU need to perform different tasks as is the case for the designed SPU program.

To enable flexible allocation with aligned variables, a memory allocation algorithm is designed based on the standard *malloc()* function. The principle of this function, *SPU_malloc_align(size_t size, int n, void * origPointer)*, is shown in figure 8.4. The source code is shown in figure 8.5 and found in [1, (B1),*spu_tool/misc.c*]. A similar mechanism is used on the PPU side, but not discussed here. The PPU version is found in [1, (B1)*ppu_code/tools.c.*].

The *SPU_malloc_align()* function allocates the amount of memory given by the multiplication of the function arguments *size* and *n* plus one quadword (16 bytes) which is the maximum displacement of the original allocated memory space. Next the pointer address is floored to 128 bit alignment by zeroing the four least significant bits of the pointer. Since this address may be allocated for other variables the pointer is finally moved to the next 128bit (16 bytes) slot by adding 10_h to the address.

By use of this function for allocating memory for program variables it is ensured, that variables are aligned correctly for use of SIMD functions and DMA transfer origins. Although this is at the cost of one quadword for each allocated memory area, it is possible to free the allocated memory, using the *origPointer*, for use in e.g. different SPU tasks, which is not possible with the same degree of flexibility when using static global variables.

```
1  /*Define vector data pointers*/
2  Rre_vec   = (vector float *) (Rre[active] + 4*times + i*S);
3  Rim_vec   = (vector float *) (Rim[active] + 4*times + i*S);
4  POSre_vec = (vector float *) (POSreFound);
5  POSim_vec = (vector float *) (POSimFound);
6  /* Reset accumulator */
7  out1re = spu_splats(0.0f);
8  out1im = spu_splats(0.0f);
9
10 /*Find complex vector product (loop kernel)*/
11 for(iii = 0; iii < S/4 + 1; iii++){
12       in1re = POSre_vec[iii];
13       in1im = POSim_vec[iii];
14       in2re = Rre_vec[iii];
15       in2im = Rim_vec[iii];
16       out1re = spu_madd(in1re, in2re, out1re);
17       out1re -= spu_mul(in1im, in2im);
18       out1im = spu_madd(in1re, in2im, out1im);
19       out1im = spu_madd(in1im, in2re, out1im);
20 }
21 /* Assign back */
22 AccRe = 0;
23 AccIm = 0;
24 AccRe = spu_extract(out1re, 0);        /*Sum real part*/
25 AccRe = spu_extract(out1re, 1) + AccRe;
26 AccRe = spu_extract(out1re, 2) + AccRe;
27 AccRe = spu_extract(out1re, 3) + AccRe;
28 AccIm = spu_extract(out1im, 0); /*Sum imaginary part*/
29 AccIm = spu_extract(out1im, 1) + AccIm;
30 AccIm = spu_extract(out1im, 2) + AccIm;
31 AccIm = spu_extract(out1im, 3) + AccIm;
32
33 *(Rmre[active] + i*M + ii) = AccRe; /*Store in r'*/
34 *(Rmim[active] + i*M + ii) = AccIm;
```

Figure 8.3: *C code for SIMDized calculation of a single element of $\bar{\mathbf{r}}'$. Vector pointers are initiated to $\bar{\mathbf{r}}$ and the appropriate $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}_k$, and the complex vector products are accumulated in vectors* out1re *and* out1im*. Finally the elements of the accumulator vectors are summed up and stored in the appropriate location of $\bar{\mathbf{r}}'$ (*Rmre *and* Rmim*).*

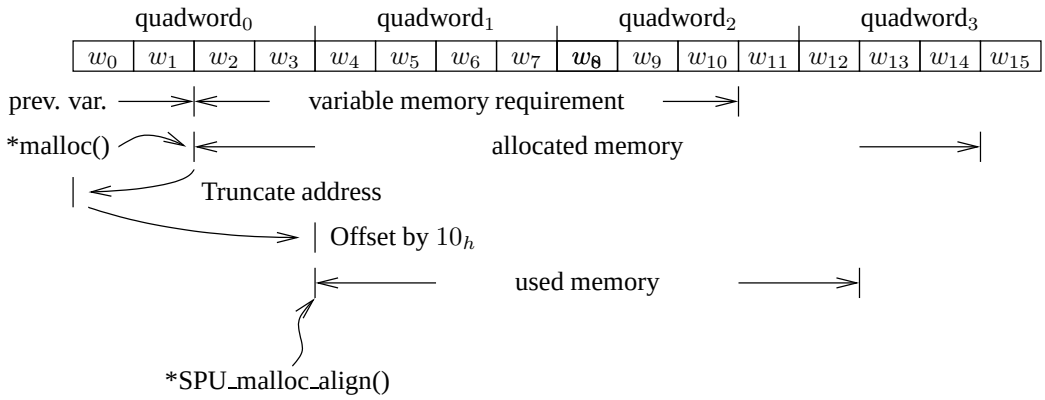


Figure 8.4: *Aligned memory allocation. A graphical interpretation of SPU_malloc_align(), where standard memory allocation is used to allocate a memory space large enough to accommodate that the variable is shifted up to achieve 128bit alignment.*

```
1  void * SPU_malloc_align(size_t size, int n, void * origPointer){
2    void * ape;
3    void * * weasel;
4    weasel = (void * *)origPointer;
5    *weasel = (void *) malloc( n*size + 16 );
6    /* 16 bytes = 128bit word */
7    /* zero four lowermost bits, add 16 to get next quadword aligned ↘
        address*/
8    ape = (void *) (((uint32_t)(*weasel) & 0xFFFFFFF0) + 0x10);
9    return ape;
10 }
```

Figure 8.5: *C code for SPU_malloc_align(). The pointer returned from* malloc() *is kept in* origPointer *for later freeing. Next the address of the subsequent 128bit aligned memory slot to* origPointer *is calculated and returned in pointer* ape*. A variable of the type void * * cannot be passed as a function argument, thus the special treatment by the "weasel void * *".*

Part III

Evaluation

This part serves as evaluation of the project. It contains definitions of the tests and details about the execution of these. The results of the test are also contained herein, and the discussion of these results leads to the conclusion and the further work this project has suggested.

Initially the tests are defined and executed. These results are then evaluated and discussed. This suggests focus areas for further iterations in the development model. Afterwards the project is concluded.

Contents

Chapter 9

Test Definition and Execution

9.1 Test Definition

The two test parameters described in the project introduction concerns the total executing time for demodulation of a single communication burst and how efficiently the CBE is utilized for the demodulation. The measurements are connected, meaning that the determination of computation time gives the utilization. Furthermore, to ensure that the calculations performed by the SPUs are correct, they are compared to reference output from the MatLAB simulation on the PPU side, after all calculations are finished. This measurement is not a performance measurement used as evaluation, but simply a verification of correct operation.

Each of the measurements are detailed in the following, starting with the time utilization and ending with the precision measurement description.

9.1.1 Time Measure

The time measurement itself is based on code from [5] and uses the *gettimeofday()* API call from time.h, encapsulated in a function called *mod_gettimeofday()* placed in *ppu_ code/tools.c* in [1, (B1)]. No specific precision for *gettimeofday* system call on a PowerPC architecture could be found, but it is assumed in the μs range and thus precise enough to measure values in the ms range.

The time is measured on the PPU side, from entering the controlling state machine until task 1 is reported done and again until all calculations of task 2 is done. This introduces a small overhead on the PPU side, but is necessary to differentiate the execution time of task 1 and 2. See the project source code for specific implementation in [1, (B1), *ppu_code/controller.c*]. The test is run 100 times for each number of active SPUs to achieve statistical confidence.

9.1.2 Efficiency Measure

The measurement of efficiency is defined for this project as the time spent by the functional units in the SPUs, on the execution of the algorithm described in the signal model section 3.7 on

page 19. The efficiency measurement thus penalizes time spent in address computation, stalls on awaiting memory and DMA transfers and everything else which does not directly contribute to the output. Not that the penalized time is wasted, it just serves to show how the CBE is utilized with the selected algorithm partition and implementation.

This definition can be ambiguous as one could move all non-MAC operations to the PPU and achieve a high efficiency, furthermore the signal model does not calculate the number of operations used on the generation of spreading and scrambling sequences. But coupled with the time requirements and close inspection of the algorithm in question, the project group believes that the efficiency measure yields insight into the utilization of the CBE.

The efficiency is determined by calculating the number of operations needed by the problem as defined in the algorithm, and divide it by the time spent on the problem. This yields a number of floating point operations per second.

In section 3.9.2 on page 24, equation (3.35), the number of floating point operations needed to estimate one symbol value of $\hat{\mathbf{d}}$ was found to be:

$$O_{total} = \mu \cdot 17340 + \alpha \cdot 16828 \tag{9.1}$$

The total number of symbols in the test communication burst is $N \cdot K$, thus the total amount of operations for this test scenario is:

$$\begin{aligned} O_{NK} &= N \cdot K \left(\mu \cdot 17340 + \alpha \cdot 16828\right) && (9.2) \\ &= \mu \cdot 665,856,000 + \alpha \cdot 646,195,200 && (9.3) \end{aligned}$$

where μ denotes multiplications and α denotes additions or subtractions. The theoretical estimate of possible Floating Point Operations per Second (FLOPS) on an SPU found in section 4.4 on page 32 is based only on the number of multiplications. Since utilization of multiply and add (MAC) operations results in one free addition per multiplication, the achieved efficiency should also be found from only the number of necessary multiplications divided by the execution time:

$$n_{FLOPS} = \frac{665,856,000}{t_{total}} \tag{9.4}$$

9.1.3 Precision Measure

In order to verify the functionality of the program implementation and the effects of the SPU truncation instead of rounding, the calculated estimates of $\hat{\mathbf{d}}$ on the CBE platform are compared to the estimates found using the MatLABsimulations, explained in section 3.8 on page 19. The MatLAB estimates are then referred to as true estimates of $\hat{\mathbf{d}}$.

In the implementation on the CBE, the output of the calculations will be compared to the true estimates of $\hat{\mathbf{d}}$. A threshold for the difference in real and imaginary is set, as seen in figure 9.1 on the next page. This is used to determine at which minimum decimal precision the algorithm can be expected to perform exactly like the MatLAB simulation. This will be used in later calculations of a minimum SNR for the algorithm.

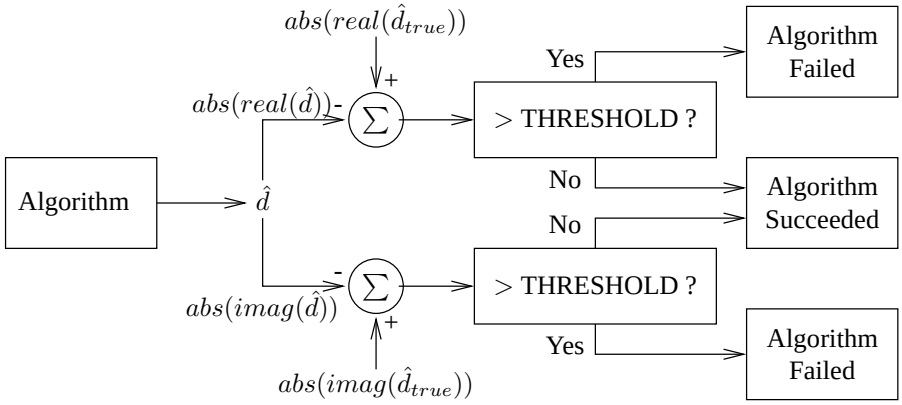


Figure 9.1: *Precisio measurement. The THRESHOLD is compared to the differences between each estimated and true (MatLAB) values of* $\hat{\mathbf{d}}$.

9.1.4 Test Scenarios

The tests of time consumtion and SPU utilization efficiency are performed on versions running with:

- Linear multiplications
- SIMD on $\bar{\bar{\mathbf{O}}}^{\mathbf{H}}\bar{\mathbf{r}}$ and linear $\bar{\bar{\mathbf{C}}}^{\mathbf{H}}\bar{\mathbf{r}}'$

For each of the above scenarios the number of active SPUs is varied from 1 to 6 to show how well added parallel SPUs are utilized.

9.2 Test Results

Figure 9.2 shows the time measures of the program execution when demodulating the test communication burst with different number of active SPUs. The mean demodulation time with 6 SPUs is 84.2 ms where 77.4 ms is spent on Task 1, while 6.8 ms is spent on Task 2. The variance of the total time is $935.7 \cdot 10^{-6}$ ms which approaches zero. This yields 7.9 GFLOPS or a utilization of approximately 10.3%.

In figure 9.3 and table 9.1 the time measures have been converted into the efficiency measure defined in equation (9.4). Furthermore, table 9.1 includes a relative measure of the achieved efficiency compared to running the program on 1 SPU, to indicate the program ability to exploit the addition of concurrent processing power when activating multiple SPUs. Timing information for the SIMD loop kernel has been extracted and placed in figure 9.4 on page 82.

The maximum deviation from the MatLAB simulation was found to be 10^{-4}, which means that the implementation is always accurate to 4 decimal places.

The precision for the calculations is determined for the real and imaginary parts independently, with the mean of the absolute values in $\hat{\mathbf{d}}$ of the MatLAB simulation to be: 0.9928 for the

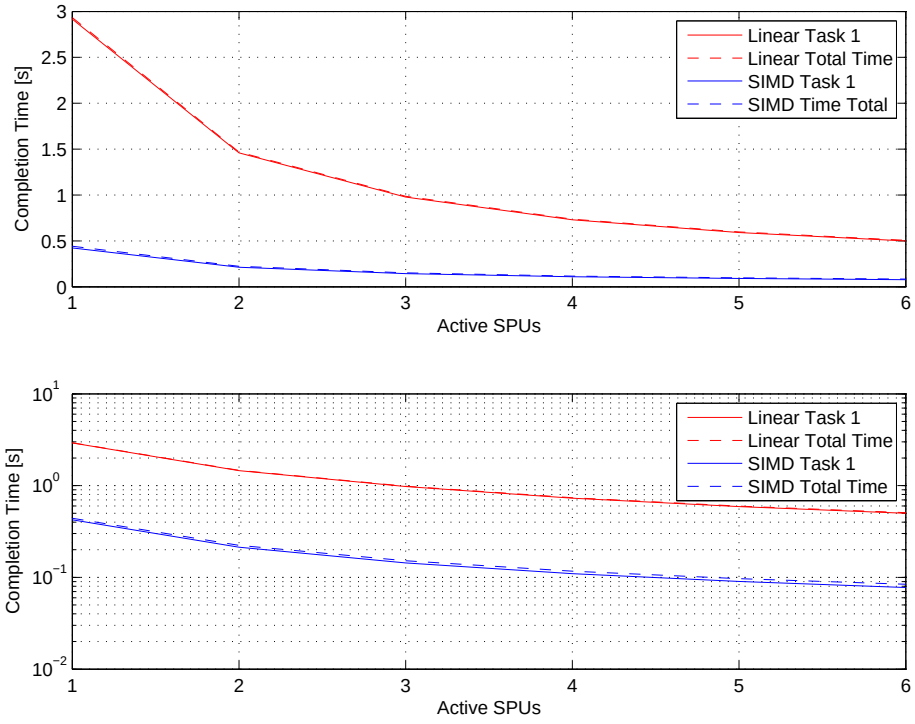


Figure 9.2: *Graphs of completion times for task 1 and the total time in linear and SIMD implementation. The same data is plotted in a semi logarithmic system to show that the usage of more SPUs does not yield a linear decrease in execution power.*

real part and 0.9922 for the imaginary part. The values of $\hat{\mathbf{d}}$ has variances of 0.5617 for the real and 0.5652 for the imaginary parts.

This yields a worst case SNR ratios of:

$$SNR_{real} = 20 \cdot \log_{10}\left(\frac{0.9928}{10^{-4}}\right) \approx 80 \text{ dB} \tag{9.5}$$

$$SNR_{imag} = 20 \cdot \log_{10}\left(\frac{0.9922}{10^{-4}}\right) \approx 80 \text{ dB} \tag{9.6}$$

This is a worst case SNR as the implementation measures all calculated numbers and no error from outliers are larger than 10^{-4}.

9.3 Discussion

This section contains a discussion of the achieved results. Initially the precision will be discussed, followed by an evaluation of the linear and SIMD implementation and the compiler output with specific focus on the loop calculating $\bar{\mathbf{r}}' = \bar{\bar{\mathbf{O}}}^{\mathrm{H}} \cdot \bar{\mathbf{r}}$. Afterwards the topics of SPU starvation and utilization are covered.

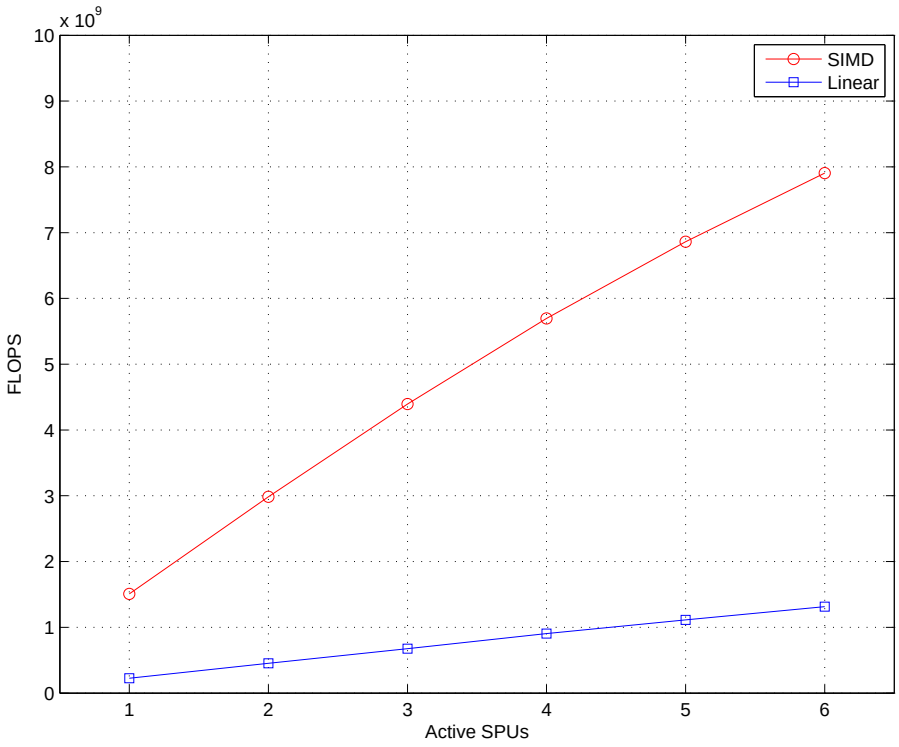


Figure 9.3: *Graph of achieved floating point operations for test communication burst.*

9.3.1 Precision

Measurements show that the worst case precision of the calculations are approximately 80 dB measured on the real and imaginary axes. This is the error introduced in the rounding mechanism on the SPUs, as the SPUs only performs truncation and not true rounding.

In association with the CDMA scenario the CBE introduced error has a SNR of

$$20 \cdot \log_{10}\left(\frac{\sqrt{0.5617}}{10^{-4}}\right) \approx 77 \text{ dB} \tag{9.7}$$

when compared to the noise variance for the real part which is 0 dB when compared to the signal. Thus the truncation noise contribution is still negligible when compared to the transmission noise in the system.

9.3.2 Linear vs. SIMD

As in the multiplication experiment on page 47 the SIMD implementation has a faster execution than the linear version. For 1 active SPU the SIMD implementation is actually more than 4 times faster than the linear code. This could be caused by the compiler being more optimized for SIMD

Active SPUs	1	2	3	4	5	6
Linear Multiply	0.2270	0.4534	0.6757	0.9039	1.1129	1.3132
Relative	1	1.9975	2.9767	3.9822	4.9032	5.7855
SIMD Multiply	1.5090	2.9837	4.3950	5.6947	6.8610	7.9047
Relative	1	1.9772	2.9125	3.7738	4.5467	5.2383

Table 9.1: *Number of achieved* GFLOPS *(10^9FLOPS) for linear and SIMD multiply test scenarios for varying number of active SPUs in figure 9.3 on the preceding page. The relative measure shows the FLOPS ratio to 1 active SPU to emphasize the performance gain incurred from computation on multiple SPUs.*

scheduling and that the CBE must use mask operations in conjunction with load/store when not loading a full quadword.

With more active SPUs the SIMD implementation suffers a slowdown, as the execution time or FLOPS does not scale with the added SPUs. This is seen in table 9.1 where the linear code achieves almost unity in relative performance, which the SIMD code only achieves for 3 or fewer SPUs. This is also seen in figure 9.3 on the preceding page where the SIMD implementation has a falloff in FLOPS with more added SPUs.

This could be caused by the SPUs becoming starved as they are waiting for their data transfers to complete. The risk of SPU starvation logically increases with more SPUs. The difference from the linear code is that the SIMD implementation consumes data at more than 4 times the rate of the linear code which causes buffer sizes and data partition to be more of an issue than in the linear code.

9.3.3 Compiler Output

An example of the compiler generated SIMD code with timing, for the loop kernel in figure 8.3 on page 69, is shown in figure 9.4 on page 82. It is seen that the loop kernel is 53 cycles long, where 23 of these cycles are spent on pipeline stalls. This is taken directly from the assembler output generated by the *spu-gcc* compiler.

Comparison with the multiplication experiment on page 47 reveals that the loop unrolling and scheduling is better, achieving 43% pipeline stalls instead of 65%, with the implementation achieving 1.5 GFLOPS in table 9.1 compared to the 897 MFLOPS in the experiment. This is partly caused by the use of temporary variables in figure 8.3 as compared to the direct use of memory access in figure 5.5 of the experiment. Secondly the timing code for the implementation shows that it performs 16 SIMD multiplications in the same loop iteration where the experiment performs 8 SIMD multiplications.

The 43% wasted cycles could be remedied by manual loop unrolling in figure 8.3 and reordering to take the pipeline into account. As a last resort the assembler instructions could be written by hand and checked with the timing tool until a better usage is achieved.

There exists another compiler (XLC compiler from IBM) tailored to multiprocessor architecuteres, but this were not tested [10].

9.3.4 SPU Utilization

The SIMD implementation achieves approximately 10.3% of the maximum theoretical throughput of 76.8 GFLOPS as found in section 4.4 on page 32. Performance close to this limit is seen demonstrated in [5] for matrix multiplication, so the limit is achievable. With the selected partition and with better loop unrolling and scheduling a utilization just below 30% is expectable. This is derived from 2 quadword loads and one quadword store, in parallel with the multiplications. Approximately 1 SIMD multiplication per 3 SPU cycles could then be performed, compared to the optimum of 1 SIMD multiplication per SPU cycle.

Part of the missing utilization is caused by the suboptimal loop unrolling shown in figure 9.4 on the following page and general problem-management which is also performed on the SPUs (spreading sequence generation, message passing).

Another cause could be SPU starvation as discussed earlier, but the gravity of this is not thoroughly tested to conclude upon. Many enhancements could be done to improve the utilization, which will be discussed in the chapter about further work, see page 83.

```
1  002395  0D                                                                  56      ai
2  002395  1D  0                                                               56789   lqx
3  002396  0D                                                                   67     ai
4  002396  1D                                                                   6      hbrp
5  002397  0D                                                                    7     nop
6  002397  1D  012                                                              789     lqx
7  002398  0D                                                                    89     ai
8  002398  1D  0123                                                             89     lqx
9  002399  0D                                                                     9     nop
10 002399  1D  01234                                                            9      lqx
11 002400  0D  01                                                                      ai
12 002400  1D  012345                                                                  lqx
13 002401  0D   1                                                                     nop
14 002401  1D   123456                                                              lqx
15 002402  0D    23                                                                 ceq
16 002402  1D    234567                                                             lqx
17 002403  0D     345678                                                            fma
18 002403  1D     345678                                                            lqx
19 002404  0D      456789                                                           fma
20 002404  1D      4                                                                hbrp
21 002405  1         567890                                                        lqx
22 002406  1          678901                                                       lqx
23 002407  1           789012                                                      lqx
24 002408  1            890123                                                     lqx
25 002409  0D            901234                                                    fnms
26 002409  1D            901234                                                    lqx
27 002410  0D             012345                                                   fma
28 002410  1D             012345                                                   lqx
29 002411  1               123456                                                 lqx
30 002412  1                234567                                                lqx
31 002415  0                 --567890                                             fma
32 002416  0                    678901                                            fma
33 002421  0                     ----123456                                       fnms
34 002422  0                         234567                                       fma
35 002427  0                          ----789012                                  fma
36 002428  0                              890123                                  fma
37 002433  0                               ----345678                             fnms
38 002434  0                                   456789                             fma
39 002439  0                                    ----901234                        fma
40 002440  0                                        012345                        fma
41 002446  0   01                                    -----6789                     fma
42 002447  0   012                                       789                      fnms
43 002448  0D                                             8     nop
44 002448  1D  01                                          89    brz
```

Figure 9.4: *Timing information for unrolled SIMD loop kernel, seen in figure 8.3 on page 69. It is seen that the loop is unrolled 4 times and 23 of the 53 cycles in the loop kernel is spent on pipeline stalls. Complete timing information is found at [1, (B2)]*

Chapter 10

Further Iterations

The development model discussed in chapter 2 on page 7 prescribes several iterations across the partitioning and mapping of the problem. The areas which could be of specific interest is described in this chapter. The discussion is split into the part of the changes which stem from the results discussion of section 9.3 on page 78 and the general changes found during the project.

10.1 Focus Areas from Results

From the results discussion the main change needed is the program which will have to be repartitioned. Examining the theoretical limit of the CBE and the required FLOPS of 665'856'000 in 10 ms it is seen that a utilization of 86.7% is needed to solve the problem in the required 10 ms. Thus a repartitioning which fully exploits the SPU architecture is needed to attain a better utilization than the 10.3% achieved.

The main deficiency in the selected partitioning is found in task 1 where $\bar{\mathbf{r}}' = \bar{\bar{\mathbf{O}}}^{\mathrm{H}}\bar{\mathbf{r}}$ is performed with multiplication of new 4-number vectors for each instruction. This imposes 3 actions (two quadword loads and a store) in pipeline 1 and thus a bottleneck. Instead the partition should facilitate reuse of a part of $\bar{\mathbf{r}}$ and utilize MAC operations, thus iterating across several users instead. With this pipeline 1 only needs to load a quadword every instruction. With correct loop unrolling a ratio of 1 instruction per multiplication could be achieved. This technique is demonstrated in [5], but were discovered only after the partitioning and design were performed. The scenario used in [5] with matrices of size 64×64 which are multiplied is not directly applicable to the DS-CDMA problem since the delays imposed by asynchronous communication offsets the data in $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ as described in section 8.2 on page 66. A workaround could impose redundancy in the storage of $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$, but this tradeoff seems favorable in light of the possible performance achievements.

Another topic is that of loop unrolling and scheduling as the direct output from the compiler performs sub-optimally. Significantly better performance can be achieved by manual programming. Not necessarily from assembler, but by use of the provided intrinsics which map directly to assembler instructions. This technique is shown in [15].

Thirdly, the utilization of the EIB and possible data-stalls by the SPUs should be measured to determine if the bus is a bottleneck. This information should be used in conjunction with determination of the buffer-sizes to improve the overall throughput and avoid SPU stalls.

10.2 General Focus Areas

Another part of the partitioning which could be reevaluated is that of determining $\bar{\mathbf{p}} \odot \bar{\mathbf{s}}$ for $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$. Currently this is calculated on the SPUs and not counted in the utilization measure of section 9.1.2 on page 75. Instead the calculation of $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ could be performed by the PPU and data moved to the SPUs as needed. This would utilize the PPU more but free the SPUs for direct calculations. Alternatively $\bar{\bar{\mathbf{O}}}^{\mathrm{H}}$ could be precalculated by the SPUs, transferred to the PPU storage and sent back to the SPUs as needed.

To offload the PPU/SPUs further, dedicated hardware for the generation of spreading and scrambling sequences could be implemented. The two tasks are deterministic and well suited for hardware implementation. Ideally this would be connected to the EIB via the bus interface in figure 4.1 on page 32, but this would require hardware which is compatible with the EIB.

Chapter 11

Conclusion

The purpose of this project was to investigate which factors are key to develop programs which efficiently utilizes the processing power of the Cell Broadband Engine, and how these programming methods may be applied to procedures related to a DS-CDMA up-link base station application.

The main concern with regard to efficient utilization of the CBE is to keep the offload processors (SPUs) busy. Two main factors arise in this context; to employ Single Instruction Multiple Data (SIMD) intrinsics for SPU functional unit utilization and efficient PPU-SPU data transfers due to the small memory capacities of the SPUs.

In order to evaluate the importance of the two programming challenges a set of principal experiments were conducted investigating the effects of employing a double buffering scheme to allow for concurrent data transfers and vector product calculations as compared to performing transfers and calculations in series, and the effects of implementing a vector product using scalar multiplications versus SIMD multiplications. These experiments show unambiguous performance gains of implementing these principles in programming for the SPUs.

Next, the partition of the DS-CDMA descrambling and despreading application was considered. The two matrix multiplications constituting the application were split into separate tasks, where the size of the intermediate product decides the sequence of task execution. Furthermore a partition of the problem where several data symbols are demodulated for each user was chosen as opposed to a single symbol at a time. This gave rise to a reduction of time spent generating user specific demodulation codes at the cost of introducing longer delays for data symbol estimates to be ready.

Job assignment and overall problem solution is assigned to the Power PC (PPU) processor which communicates with the SPUs via the CBE's inbuilt mailbox system. The scheduling is performed in a work-requester model, where each SPU requests new jobs from the PPU once the current job is completed. The PPU also presents the needed data to the SPUs which transfers the data from the main storage connected to the PPU via the Element Interface Bus to the local storage on the SPUs for use in calculations. To utilize SIMD capabilities of the SPUs the data must be specially aligned in memory, which presents a problem with regards to asynchronous communications in CDMA. A solution with redundant store of spreading and scrambling sequences

were implemented thus enabling SIMD computations and the resulting performance enhancement.

The product is a CBE program which descrambles and despreads a communication burst of 300 symbols for each of the 128 users who transmits through a fading multipath channel. Two versions of the SPU program were tested, the difference being the matrix multiplications being implemented with linear and SIMD instructions respectively. The performance of each program was measured as program execution time which was translated into an achieved utilization measure. This utilization measure shows the SIMD implementation to achieve up to 7.9 GFLOPS, with all 6 SPUs active, compared to a 76.8 GFLOPS theoretical limit. Other studies show that the theoretical limit is close to achievable for problems which are partitioned to cater for data reuse in multiplications.

Another subject for improvements is the actual programming of the calculations to improve performance of the loop kernels, as the direct output of the compiler it suboptimal with regards to instruction scheduling and loop unrolling. Instead manual loop unrolling and instruction reordering should be applied, where it should be possible to avoid pipeline stalls of 40% of the time.

The outcome of the project is an implementation that works, but the hard real-time requirement of 10 ms is not met, the total demodulation time for 6 SPUs is measured to 84.2 ms. To improve on this several more iterations across the development model is needed. These iterations are discussed in chapter 10 on page 83. The most significant changes is that of repartitioning the problem and solution for more data-reuse and thus higher throughput, and make a manual scheduling and instruction reordering of the loop kernels to exploit the functional units to a higher degree. Secondly the utilization of the Element Interface Bus must be measured and it must be determined if this is a bottleneck.

A change in the SPU architecture, which could enable a higher and easier attainable utilization for more problems, is the inclusion of an extra SPU Load and Store Unit (SLS on figure 4.2 on page 34). This will significantly improve the speed at which to independant vectors could be multiplied, such as the problem tested in section 5.2 on page 47 where the movement of data to and from registers becomes the bottleneck and idles the floating point unit (SFX) for 2/3 of all instructions. This could be achieved by segmenting the LS of the SPUs, which would enable multiple reads for each instruction.