



IST-2001-34561

MUMOR

D3.3

Hardware/Software Partitioning

Document Number: IST-2001-34561/NOKIA/WP3/R/PU/011

Contractual Date of Delivery to the CEC:	July 2003
Actual Date of Delivery to the CEC:	04.08.2003
Author(s):	NOKIA
Participant(s):	NOKIA, UNIS, LETI
Workpackage:	WP3
Est. person months:	21
Security:	Public
Nature:	Report
Version:	v11
Total number of pages:	84

Abstract:

The main purpose of the present document is to describe the hardware/software partitioning of the investigated multi-mode digital baseband design. There are several techniques to do this partitioning. One of them is based on actual profiling information of the design, i.e. figures about the computational complexity of the design. The partitioning of the design will be done for each functional entity or unit. Therefore the computational expense of each of the components will be presented. After introducing the principles of the partitioning process and the related algorithms the partitioning of the investigated system will be presented together with the implementation architecture.

Keyword list: UMTS, HSDPA, FDD, TDD, Complexity, Partitioning, Implementation, Architecture, Soft-configurable

Abbreviations / Terminology

Note: Some of the abbreviations, terminology and vocabulary, which are frequently used in the 3GPP specification are considered as "well known" and is used in the present document without being explained. Please refer to the 3GPP "Vocabulary" document (3G TR 25.990) in this case.

ACS	Add Compare Select
AFC	Automatic Frequency Control
ASIP	Algorithm Specific Instruction Processing
BMU	Branch Metric Unit
CCTrCH	Coded Composite Transport Channel
DCCH	Dedicated Control Channel
DTCH	Dedicated Traffic Channel
FLP	Floating point
FXP	Fixed point
H-ARQ	Hybrid Automatic Repeat Request
HDL	Hardware Description Language
HS-	High Speed (HSDPA channels and block names)
LUT	Look-up Table
MAI	Multiple Access Interference
MAP	Maximum A Posteriori
MIPS	Million Instructions Per Second
MMSE	Minimum Mean Squared Error
MOPS	Million Operations Per Second
NRE	Non-recurring Engineering
OVSF	Orthogonal Variable Spreading Factor
PE	Processing Element
PSC	Primary Synchronisation Code
RM	Rate Matcher
RSC	Recursive Systematic Codes
SoC	System on Chip
SRRC	Square Root Raised Cosine
TrCH	Transport Channel
ZF-BLE	Zero-Forcing Block Linear Equalization

Table of contents

1	INTRODUCTION	8
2	EVALUATION OF MEASURES	9
2.1	RX PULSE SHAPING FILTER	12
2.2	TIME CONTROLLER.....	13
2.3	CHANNEL ESTIMATOR & MULTI-PATH SEARCHER	13
2.4	RAKE COMBINER (FDD/HSDPA MODE ONLY)	15
2.5	DESPREADER & SOFT BIT DEMAPPER (FDD/HS-DPA MODE ONLY)	16
2.6	DESCRAMBLER (FDD/HS-DPA MODE ONLY)	16
2.7	SOFT BIT DECODER (TDD MODE ONLY)	17
2.8	FREQUENCY ESTIMATOR	19
2.9	FEEDBACK FREQUENCY SYNCHRONIZER	20
2.10	FORWARD FREQUENCY SYNCHRONIZER	20
2.11	DL RX CCTRCH AND TRCH DECODING.....	21
2.11.1	<i>PhChannelDeMap (FDD and TDD)</i>	21
2.11.2	<i>Second De-Interleaver (FDD and TDD)</i>	21
2.11.3	<i>Second DTX Remove (FDD)</i>	21
2.11.4	<i>TrChDeMux (FDD and TDD)</i>	21
2.11.5	<i>First De-Interleaver (FDD and TDD)</i>	22
2.11.6	<i>First DTX Remove (FDD)</i>	22
2.11.7	<i>RateDeMatching (FDD and TDD)</i>	22
2.11.8	<i>Bit-De-Scrambling (TDD)</i>	22
2.11.9	<i>RadioFrameDeEqualization (TDD)</i>	22
2.11.10	<i>HS-Physical Channel DeMapping</i>	22
2.11.11	<i>HS-Rx-Constellation Re-Arrangement</i>	22
2.11.12	<i>HS-DeInterleaving</i>	23
2.11.13	<i>HS-RateDeMatching with H-ARQ</i>	23
2.11.14	<i>Viterbi decoder (FDD and TDD)</i>	24
2.11.15	<i>Turbo decoder (FDD, TDD and HSDPA)</i>	24
2.11.16	<i>CRC Check (FDD, TDD and HSDPA)</i>	24
2.12	CELL SEARCHER	25
3	APPLIED OPTIMISATION AND OPTIMIZATION GAIN	29
3.1	MULTI-MODE RX PULSE SHAPING FILTER	30
3.2	MULTI-MODE TIME CONTROLLER.....	31
3.3	MULTI-MODE CHANNEL ESTIMATOR & MULTI-PATH SEARCHER	31
3.4	MULTI-MODE RAKE COMBINER	32
3.5	MULTI-MODE DESPREADER AND SOFT BIT DEMAPPER	33
3.6	MULTI-MODE DESCRAMBLER.....	34
3.7	MULTI-MODE FREQUENCY ESTIMATOR	35
3.8	MULTI-MODE FEEDBACK FREQUENCY SYNCHRONIZER	36
3.9	MULTI-MODE FORWARD FREQUENCY SYNCHRONIZER	36
3.10	MULTI-MODE DL RX CCTRCH DECODING	36
3.10.1	<i>PhChannelDeMapping</i>	37
3.10.2	<i>Second De-Interleaver</i>	37
3.10.3	<i>FDD Physical Channel Desegmentation</i>	37
3.10.4	<i>TDD Physical Channel Desegmentation and Bit Descrambling</i>	37
3.10.5	<i>FDD Second DTX Removal</i>	37
3.10.6	<i>TrChDemux</i>	37
3.11	MULTI-MODE DL RX TRCH DECODING	38
3.11.1	<i>Radio Frame Desegmentation</i>	38

3.11.2	<i>First DeInterleaver</i>	38
3.11.3	<i>FDD 1st DTX Removal</i>	39
3.11.4	<i>Rate Dematching</i>	39
3.11.5	<i>TDD Radio Frame DeEqualization</i>	39
3.11.6	<i>Channel Decoding</i>	39
3.12	MULTI-MODE CELL SEARCHER	40
3.12.1	<i>P-SCH</i>	40
3.12.2	<i>S-SCH</i>	43
3.12.3	<i>RAM</i>	44
3.12.4	<i>Cell Search Summary</i>	45
3.13	MULTI-MODE UPLINK TX CCTRCH AND TRCH ENCODING	45
3.14	MULTI-MODE UPLINK BIT MAPPER AND SPREADER.....	46
3.15	MULTI-MODE UPLINK SCRAMBLER & MIDAMBLE INSERTION (TDD).....	47
3.16	MULTI-MODE UPLINK TX PULSE SHAPING FILTER	48
4	IMPLEMENTATION DEFINITION	49
4.1	HARDWARE/SOFTWARE PARTITIONING	49
4.1.1	<i>Basic Principles</i>	49
4.1.2	<i>Design Flow</i>	51
4.2	PARTITIONING OF THE SYSTEM UNDER INVESTIGATION	54
4.2.1	<i>Target Implementation Components</i>	54
4.2.2	<i>Partitioning onto HW and SW</i>	55
4.2.3	<i>Partitioning with Soft-Configurable Technologies</i>	60
5	IMPLEMENTATION ARCHITECTURE	64
5.1	ARCHITECTURE CONSIDERATIONS	64
5.2	ARCHITECTURE DESCRIPTION	64
5.2.1	<i>Pulse Shaping</i>	64
5.2.2	<i>Channel estimation</i>	65
5.2.3	<i>Rake receiver</i>	67
5.2.4	<i>The communication between nodes (The HUNT API)</i>	68
5.2.5	<i>HUNT API description</i>	68
5.2.6	<i>Bit rate performances</i>	76
5.2.7	<i>Exemple of API using</i>	80
6	SUMMARY AND CONCLUSION	83
	REFERENCES	84

List of figures

Figure 2.1 : Joint Detection Block Diagram	17
Figure 2.2 : Joint Detection Matrix.....	18
Figure 2.3: General Turbo decoder structure.....	24
Figure 3.4: Multi-mode TDD and FDD Coded Composite Transport Channel block diagram.....	36
Figure 3.5: Multi-mode TDD and FDD Transport Channel TrCH (downlink) block diagram	38
Figure 3.6: Multi-mode TDD and FDD Cell Search block (downlink) diagram.....	40
Figure 3.7: Block diagram of matched filter a sub block of P-SCH multi-mode block.....	41
Figure 3.8: Block diagram of Combined Matched Filter of PSCH Cell Search	41
Figure 3.9: Block diagram of P-SCH Matched Filter	41
Figure 4.1: Hardware/Software Partitioning; Simplified trade-off: Performance vs. Flexibility.....	49
Figure 4.2: Structure of the plain HW/SW partitioning design flow	51
Figure 4.3: Iterative HW/SW partitioning design flow.....	53
Figure 4.4: Filter structured component with bit-level operations.....	61
Figure 4.5: Turbo Encoder Structure	62
Figure 4.6: Bit Scrambler and Descrambler Structure	62
Figure 4.7: Bit-level access for Bit-Arithmetic Units	63
Figure 5.1 : Receiver demonstrator blocks	65
Figure 5.2 : Correlator block structure.....	66
Figure 5.3 : Correlation block (1 sample per chip).....	66
Figure 5.4 : Channel estimator overview	67
Figure 5.5 : soft data decoding.....	68
Figure 5.6: The HEART.	69
Figure 5.7: The node access point.....	70
Figure 5.8: Time slot concept.	71
Figure 5.9: Simple connection from the node A toward the node B.....	72
Figure 5.10: Simple connection from the node A toward the node B using two time-slot.....	72
Figure 5.11: Two separate connections from the node A toward the node B.....	73
Figure 5.12: Two separate connections from the node A toward the node B, from node B towards node D re-using the same time-slot.....	74
Figure 5.13: Multi-cast connection from node A toward both nodes B and D.....	74
Figure 5.14: FIFO connection.....	76
Figure 5.15: HERON FPGA4 diagram.....	76

List of tables

Table 2.1: FDD DL reference measurement channel, physical parameters (384 kbps).....	10
Table 2.2: FDD DL reference measurement channel, transport channel parameters (384 kbps)	10
Table 2.3: TDD DL reference measurement channel, 384 kbps (chip rate processing)	11
Table 2.4: TDD DL reference measurement channel, 384 kbps (symbol and bit rate processing).....	11
Table 2.5: HSDPA Fixed Reference Channel Definition H-Set 3 (16QAM, 2332 kbps).....	12
Table 2.6: Computational complexity of Rx Pulse Shaping Filter in FDD-mode specific design for 1 frame.....	12
Table 2.7: Computational complexity of Rx Pulse Shaping Filter in TDD-mode specific design for 1 slot.....	12
Table 2.8: Computational complexity of Time Controller in FDD-mode specific design for 1 frame.....	13
Table 2.9: Computational complexity of Channel Estimator & Multi-path Searcher in FDD-mode specific design for 1 frame.....	13
Table 2.10: Computational complexity of Channel Estimator & Multi-path Searcher in HSDPA-mode specific design for 1 frame	14
Table 2.11: Computational complexity of Channel Estimator in TDD-mode specific design for 1 slot.....	15
Table 2.12: Computational complexity of Rake Combiner in FDD-mode specific design for 1 frame.....	15
Table 2.13: Computational complexity of Rake Combiner in HSDPA-mode specific design for 1 frame	15
Table 2.14: Computational complexity of Despreader & Soft Bit Demapper in FDD-mode specific design for 1 frame and 1 channel	16
Table 2.15: Computational complexity of Channel Estimator & Multi-path Searcher in HSDPA-mode specific design for 1 frame	16
Table 2.16: Computational complexity of Descrambler in FDD-mode specific design for 1 frame	16
Table 2.17 : Joint Detection complexity	19
Table 2.18 : Joint Detection complexity - example	19
Table 2.19: Computational complexity of Frequency Estimator in FDD-mode specific design for 1 frame	20
Table 2.20: Computational complexity of Feedback Frequency Synchronizer in FDD-mode specific design for 1 frame	20
Table 2.21: Computational complexity of Forward Frequency Synchronizer in FDD-mode specific design for 1 frame	20
Table 2.22: Computational complexity of HSDPA DL Rx HARQ without certain counters.....	23
Table 2.23: Computational complexity of FDD DL Rx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A3.4.....	26
Table 2.24: Computational complexity of TDD DL Rx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.102 Chapter A2.5	27
Table 2.25: Computational complexity of HSDPA DL Rx blocks (TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A7.1.3.....	28
Table 3.1: FDD UL reference measurement channel, physical parameters (384 kbps).....	29
Table 3.2: FDD UL reference measurement channel, transport channel parameters (384 kbps)	30
Table 3.3: TDD UL reference measurement channel, 12.2 kbps (chip rate processing)	30
Table 3.4: Computational complexity of Downlink Rx Pulse Shaping Filter in FDD-mode of multi-mode design for 1 frame.....	30
Table 3.5: Computational complexity of Downlink Rx Pulse Shaping Filter in TDD-mode of multi-mode design for 1 slot	31

Table 3.6: Computational complexity of Channel Estimator & Multi-path Searcher in FDD-mode of multi-mode design for 1 frame	31
Table 3.7: Computational complexity of Channel Estimator in HSDPA-mode of multi-mode design for 1 frame	32
Table 3.8: Computational complexity of Channel Estimator in TDD-mode of multi-mode design for 1 slot.....	32
Table 3.9: Computational complexity of Rake Combiner in FDD-mode of multi-mode design for 1 frame	32
Table 3.10: Computational complexity of Rake Combiner in HSDPA-mode of multi-mode design for 1 frame	33
Table 3.11: Computational complexity of Rake Combiner in TDD-mode of multi-mode design for 1 slot	33
Table 3.12: Computational complexity of Despreader and Soft Bit Demapper in FDD-mode of multi-mode design for 1 frame	33
Table 3.13: Computational complexity of Despreader and Soft Bit Demapper in HSDPA-mode of multi-mode design for 1 frame and 1 physical channel.....	34
Table 3.14: Computational complexity of Despreader and Soft Bit Demapper in TDD-mode of multi-mode design for 1 frame and 1 physical channel.....	34
Table 3.15: Computational complexity of Descrambler in FDD-mode of multi-mode design for 1 frame.....	35
Table 3.16: Computational complexity of Descrambler in TDD-mode of multi-mode design for 1 slot	35
Table 3.17: Computational complexity of Frequency Estimator in FDD-mode of multi-mode design for 1 frame	35
Table 3.18: Computational complexity of Frequency Estimator in TDD-mode of multi-mode design for 1 slot	36
Table 3.19: Computational complexity of FDD UL Tx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A2.4.....	45
Table 3.20: Computational complexity of TDD UL Tx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.102 Chapter A2.1	46
Table 3.21: Computational complexity of Uplink Bit Mapper and Spreader in FDD-mode of multi-mode design for 1 frame.....	46
Table 3.22: Computational complexity of Uplink Bit Mapper and Spreader in TDD-mode of multi-mode design for 1 slot	47
Table 3.23: Computational complexity of Uplink Scrambler in FDD-mode of multi-mode design for 1 frame..	47
Table 3.24: Computational complexity of Uplink Scrambler in FDD-mode of multi-mode design for 1 slot	47
Table 3.25: Computational complexity of Uplink Tx Pulse Shaping Filter in FDD-mode of multi-mode design for 1 frame.....	48
Table 3.26: Computational complexity of Uplink Tx Pulse Shaping Filter in TDD-mode of multi-mode design for 1 slot	48
Table 4.1: Functional units exceeding the defined complexity threshold.....	57
Table 4.2: Functional units with bit-level operations.....	58
Table 4.3: Mapping of the remaining functional units.....	59
Table 4.4: Soft-configurable technology vs. HW and SW	60
Table 5.1: Peak rate.....	77
Table 5.2: Rate using float DMA.....	78
Table 5.3: Rate using dedicated DMA.....	78
Table 5.4: Rate using input and output FIFO.....	79
Table 5.5: PCI host bandwidth.....	80

1 Introduction

Embedded systems for reactive (real-time) applications are implemented as mixed software-hardware systems, utilising microprocessors, microcontroller and/or Digital Signal Processors. Generally, software is used for features and flexibility, while hardware is used to achieve the required performance or for power saving reasons. Re-configurable macros and architectures are going to be introduced for embedded systems to increase flexibility on a higher performance level compared to general-purpose processors. Typically the HW/SW partitioning task is simply seen as mapping the defined functionality from e.g. an executable specification onto given resources.

However, when designing an embedded system for a heterogeneous implementation (i.e. hardware and software on embedded processors), more complex activities have to be undertaken as given in the following list:

- System behavioural description – giving the executable specification of what the system is supposed to do
- Hardware/software partitioning – deciding which parts of the system behaviour should be realised by what parts of the hardware architecture.
- Task scheduling – controlling how the different computational resources of the hardware architecture are shared between the tasks of the behavioural description
- Hardware architecture selection – describing what hardware components should be used and how they are connected

The main purpose of the present document is to describe the hardware/software partitioning of the investigated multi-mode digital baseband design. There are several techniques to do this partitioning. One of them is based on actual profiling information of the design, i.e. figures about the computational complexity of the design. The partitioning of the design will be done for each functional entity or “(group of) block(s)”. Therefore the computational expense of each of the components must be available.

These complexity figures are provided in chapter 2 of this document. For each functional unit these figures will be provided. These figures are based on selected test cases as defined by 3GPP and are used for two purposes. First they are used to show the optimisation gain achieved during the multi-mode optimisation to come from the reference design to the true multi-mode operating design. Though no figures like gate count or DSP code length or execution time can be provided because the non-optimised design has not been and will not be coded for DSP or synthesized, with these figures the optimisation gain can be given quantitatively in shape of computational complexity expenses. This is done in chapter 3 together with introducing the means by which the optimisation has been achieved. It is also explained what approaches have been applied to optimise the reference design, i.e. by changing entire algorithms, improving their implementation, changing the control flow, reorganising the data flow, etc.

Chapter 4 contains the actual hardware/software partitioning decisions. In its first part general issues about the partitioning process will be introduced. Term definitions are done and basic principles, approaches, and techniques will be presented. In the second part the actual partitioning of the investigated system will be presented. First partitioning approach is restricted to pure HW and SW solutions. In an additional chapter selected components are discussed for their feasibility to implement them more optimised with a soft-configurable technique.

In chapter 5 the resulting implementation architecture will be derived together with some means concerning the task scheduling onto the (shared) resources. It will be shown how the components of the implementation will be connected together.

2 Evaluation of Measures

In this chapter the complexity figures of the three single-mode designs, i.e. mode specific, are given. If nothing else is mentioned the figures in the tables are based on the actual operations to be executed during the simulation of the respective blocks rather than just on theoretical calculations. This approach has been preferred to consider also the control flow beside the pure data flow. The control flow includes address calculations for memory operations and counters in loops, which are e.g. used to separate different data blocks. The control flow can have great impact on the total amount of executed operations. Some algorithms, e.g. Interleaver, mainly consist of address calculation.

However there are some critical issues to be considered when going for this approach:

- Some operations are only included in a specific way of writing the model. Such operation must be completely eliminated before the calculation.
- There may also be models written with already one implementation in mind, i.e. rather SW-like or in a HW oriented manner. If the operation count figures should be used for portioning they should be implementation independent. If this kind of figure forms a major portion of the total count for that block it has to be mentioned explicitly because it is typical that the final operation count is dependent on the implementation.
- The results are tied to the testcase and are not scalable. In contrast the theoretical approach has certain parameters like input block length, etc. which can be varied to get results for different use cases. However the test case based approach can provide only numbers for the simulated test cases.

Nevertheless to be sure that the control is included in the figures this approach has been selected to get a better impression of the capacity usage of a DSP and not to overload it because of neglected control operations. For this reason the figures need certain explanations to increase their confidence for the partitioning process.

It has to be noted that here no distinction between FXP and FLP calculations has been made as the transition to FXP simulation and implementation will be done after the partitioning. Additionally there is no weighting of the different kinds of operations in the sum (“total”) fields of the tables beyond. All operations are calculated with a weight of 1. But to get information about the usage of different operations the following types have been distinguished during the complexity analysis:

- Addition / Subtraction
- Multiplication
- Division
- Modulo.

Only arithmetic operations are counted here. Logical operations are neglected, because they are basic cells in HW and thus much simpler than adder or multiplier. Though it should be checked if a block uses logical operations to great extend, because mainly these operation are done on bit-level. If such a block will be implemented in SW on a DSP it has to be noticed that these operations use the same amount of resources as the registered arithmetic operations, because of their inefficient execution.

Beside the *types* of operation it also has to be considered, which *operands* appear. The difference in implementation complexity especially for multiplication and division may be in orders of magnitude for the different operands. Distinguished are generally constants and varying operands, constants are further distinguished so that there are in all:

- 1 (constant)
- 2^n , $n \in \mathbb{N}$, $n \neq 0$ (constant)
- other constants

- varying operand (any non-constant).

The varying operands are not further distinguished. The distinction between constants and non-constants has been made especially to get precise figures for HW implementations. While in SW there is not much difference at all if constants or varying numbers are processed, the HW complexity may drastically decrease.

For division and multiplication, a varying number, which is in any case equal to 2^n ($n \in \mathbb{N}$, $n \neq 0$), has been calculated as being a constant of that type though it is slightly more complex in HW but it is a lot simpler to implement than a multiplication or division with an arbitrary number. So the error made in the calculation, when applying this deviation is much smaller.

As mentioned above the chosen approach requires the definition of dedicated test case for which the complexity figures will be retrieved. The following test cases have been selected from those defined in the 3GPP specification documents [3GPP25.101][3GPP25.102]:

FDD

For the FDD complexity calculation the test case “DL reference measurement channel (384 kbps)” according to chapter A.3.4 of [3GPP25.101] has been selected. The details of this test case are given here:

Parameter	Unit	Level
Information bit rate	kbps	384
DPCH	ksps	480
Slot Format # I	-	15
TFCI		On
Power offsets PO1, PO2 and PO3	dB	0
Puncturing	%	22

Table 2.1: FDD DL reference measurement channel, physical parameters (384 kbps)

Parameter	DTCH	DCCH
Transport Channel Number	1	2
Transport Block Size	3840	100
Transport Block Set Size	3840	100
Transmission Time Interval	10 ms	40 ms
Type of Error Protection	Turbo Coding	Convolution Coding
Coding Rate	1/3	1/3
Rate Matching attribute	256	256
Size of CRC	16	12
Position of TrCH in radio frame	fixed	Fixed

Table 2.2: FDD DL reference measurement channel, transport channel parameters (384 kbps)

TDD

For the TDD complexity calculation two different test cases have been used for different blocks. The chip level processing complexity has been calculated with the test case “DL reference measurement channel (2 Mbps), 3.84 Mcps TDD Option” according to chapter A.2.8.1 of [3GPP25.102] to test TDD with a peak data rate (this is the reference channel with highest data rate). The symbol and bit level processing has been done with test case “DL reference measurement channel (384 kbps), 3.84 Mcps TDD Option” according to chapter A.2.5.1 of [3GPP25.102] to have the same data rate as with the FDD test case to ease the comparison of the results.

The Parameters for the chip rate processing are given in Table 2.3.

Parameter	Value
Information data rate	2048 kbps
RU's allocated	16*12TS = 192RU
Midamble	256 chips
Interleaving	10 ms
Power control	0 Bit/user
TFCI	16 Bit/user
Inband signalling DCCH	2 kbps
Puncturing level at Code rate 1/3 : DCH / DCCH	13.9% / 0%

Table 2.3: TDD DL reference measurement channel, 384 kbps (chip rate processing)

The Parameters for the symbol and bit rate processing are given in Table 2.4.

Parameter	Value
Information data rate	384 kbps
RU's allocated	8*3TS = 24RU
Midamble	256 chips
Interleaving	20 ms
Power control	0 Bit/user
TFCI	16 Bit/user
Inband signalling DCCH	2 kbps
Puncturing level at Code rate : 1/3 DCH / 1/2 DCCH	43.4% / 15.3%

Table 2.4: TDD DL reference measurement channel, 384 kbps (symbol and bit rate processing)

HSDPA

For the HSDPA complexity calculation the test case “Fixed Reference Channel Definition H-Set 3” according to chapter A.7.1.3 of [3GPP25.101] has been selected. This test case is defined for both QPSK and 16QAM modulation. For the complexity calculation the more complex 16QAM case has been selected. The details of this test case are given here:

Parameter	Unit	Level
Nominal Avg. Inf. Bit Rate	kbps	2332
Inter-TTI Distance	TTI's	1
Number of HARQ Processes	Processes	6
Information Bit Payload	Bits	4664
Number Code Blocks	Blocks	1
Binary Channel Bits Per TTI	Bits	7680
Total Available SML's,in UE	SML's	57600
Number of SML's per HARQ Proc.	SML's	9600
Coding Rate		0.61
Number of Physical Channel Codes	Codes	4
Modulation	kbps	16QAM

Table 2.5: HSDPA Fixed Reference Channel Definition H-Set 3 (16QAM, 2332 kbps)

For further details concerning the test cases refer to the 3GPP specification.

2.1 Rx Pulse Shaping Filter

Computational complexity of Rx Pulse Shaping Filter in FDD-mode specific design which is normalised to 1 frame period is in Table 2.6.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	19,753,024	4,938,272	9,876,504	0	0	
Const (1,-1)	20,215,978	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	39,814,656	9,876,504	20,061,648	0	0	
Σ	79,783,658	14,814,776	29,938,152	0	0	124,536,586

Table 2.6: Computational complexity of Rx Pulse Shaping Filter in FDD-mode specific design for 1 frame

The complexities of Rx Pulse Shaping Filter in FDD and HSDPA -specific modes are exactly the same.

Computational complexity of Rx Pulse Shaping Filter in TDD-mode specific design for 1 slot period is in Table 2.7.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	0	0	3,317,760	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	3,317,760	0	0	0	0	
Σ	3,317,760	0	3,317,760	0	0	6,635,520

Table 2.7: Computational complexity of Rx Pulse Shaping Filter in TDD-mode specific design for 1 slot

Note that the total complexity of Rx Pulse Shaping Filter in TDD mode specific design over 1 frame is almost the same as in FDD mode.

The complexities above are from a multiplication of input signal with coefficients of the PSF and an accumulation. So the figures are all related to algorithm itself, not to modelling purpose.

2.2 Time Controller

Computational complexity of Time Controller in FDD-mode specific design, which is normalised, to 1 frame period is in Table 2.8.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	0	0	0	0	0	
Const (1,-1)	1,736,101	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	0	0	0	0	0	
Σ	1,736,101	0	0	0	0	1,736,101

Table 2.8: Computational complexity of Time Controller in FDD-mode specific design for 1 frame

Timing Controller is to adjust the searching window position with the peak path position centered. It may have the functionality of time tracking of each path but we don't need it because Channel Estimator and Rake Combiner work at chipx8 oversampling rate every slot as fast as time tracking does. So most of the complexity required for Time Controller is caused by increment operation.

Time Controller is exactly the same for FDD and HSDPA modes, but we don't have it for TDD mode.

2.3 Channel Estimator & Multi-path Searcher

Computational complexity of Channel Estimator & Multi-path Searcher in FDD-mode specific design which is normalised to 1 frame period is in Table 2.9.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	37,637	35,520	282,724	0	0	
Const (1,-1)	73,691,548	0	0	0	0	
Const (any)	0	0	71,280	0	0	
Any	220,619,009	146,596,576	293,688,859	1,059	0	
Σ	294,348,194	146,632,096	294,042,863	1,059	0	735,024,212

Table 2.9: Computational complexity of Channel Estimator & Multi-path Searcher in FDD-mode specific design for 1 frame

In Table 2.9, figures are for algorithm itself for correlating input signal in the searching window with training sequence. Other complexity required for modeling purpose such as starting control has not been included.

Computational complexity of Channel Estimator & Multi-path Searcher in HSDPA-mode specific design which is normalised to 1 frame period is in Table 2.10.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	37,638	35,520	282,719	0	0	
Const (1,-1)	73,690,058	0	0	0	0	
Const (any)	0	0	71,280	0	0	
Any	220,614,742	146,593,606	293,683,517	1,667	0	
Σ	294,342,438	146,629,126	294,037,516	1,667	0	735,010,747

Table 2.10: Computational complexity of Channel Estimator & Multi-path Searcher in HSDPA-mode specific design for 1 frame

In Table 2.10, figures are for algorithm itself for correlating input signal in the searching window with training sequence. Other complexity required for modeling purpose such as starting control has not been included.

Note that complexities of Channel Estimator & Multi-path Searcher in FDD and HSDPA-mode specific design are almost the same.

Concerning TDD single mode, the channel estimation is based on the Steiner algorithm. It is processed on the midamble data (G matrix). The received midamble data is:

$$\mathbf{e} = \mathbf{G} \cdot \mathbf{h} + \mathbf{n}$$

where \mathbf{G} is a matrix [P×P] built thanks to the midamble knowledge.

We have to find the estimated vector \mathbf{h}' of the impulse response of the channel \mathbf{h} (FIR filter).

\mathbf{G} is a circular matrix. This property enables to write:

$$\mathbf{h}' = \frac{1}{\sqrt{P}} \text{FFT}^{-1} [\text{FFT}(\mathbf{e}) ./ \text{FFT}(\mathbf{G}_0)]$$

where :

\mathbf{G}_0 is the first column of \mathbf{G} .

The operation $./$ is a term by term division.

\mathbf{h}' contains the coefficients of a FIR filter.

We have to process these functions:

FFT of the e vector,

FFT of the first column of G,

Vector division,

reverse FFT.

The (I)FFT processes 192 taps and each is divided into three 64 taps (I)FFT. The complexity of an (I)FFT is $2N \log_2(N)$ multiplications and $2N \log_2(N)$ additions. Considering that $\text{FFT}(\mathbf{G}_0)$ is processed once, the computational complexity of Channel Estimator & Multi-path Searcher in TDD-mode specific design for 1 time slot period is in Table 2.11.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	192	0	
Any	1,536	0	1,536	0	0	
Σ	1,536	0	1,536	192	0	3,264

Table 2.11: Computational complexity of Channel Estimator in TDD-mode specific design for 1 slot

2.4 Rake Combiner (FDD/HSDPA mode only)

The computational complexity of Rake Combiner in FDD-mode specific design, which is normalised, to 1 frame period is in Table 2.12.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	903,680	38,400	0	0	0	
Const (1,-1)	1,674,211	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	2,842,800	942,080	3,614,720	0	0	
Σ	5,420,691	980,480	3,614,720	0	0	10,015,891

Table 2.12: Computational complexity of Rake Combiner in FDD-mode specific design for 1 frame

In Table 2.12, figures are for algorithm itself for combining all the signals from all the valid paths after compensating for channel distortion with channel estimates. Other complexity required for modeling purpose such as taps buffering and timing sequence checking has not been included.

Computational complexity of Rake Combiner in HSDPA-mode specific design which is normalised to 1 frame period is in Table 2.13.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	1,422,080	38,400	0	0	0	
Const (1,-1)	2,193,016	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	4,291,050	1,422,080	5,688,320	0	0	
Σ	7,906,146	1,460,480	5,688,320	0	0	15,054,946

Table 2.13: Computational complexity of Rake Combiner in HSDPA-mode specific design for 1 frame

In Table 2.13, figures are for algorithm itself for combining all the signals from all the valid paths after compensating for channel distortion with channel estimates. Other complexity required for modeling purpose such as taps buffering and timing sequence checking has not been included.

The difference of complexities between FDD and HSDPA is from more valid (non-zero) channel estimate taps of HSDPA than those of FDD because test channel environment (VEHICULAR A) of HSDPA has 6 paths, but 4 (CASE 6) for FDD.

2.5 Despredator & Soft Bit Demapper (FDD/HS-DPA mode only)

Computational complexity of Despredator & Soft Bit Demapper in FDD-mode specific design which is normalised to 1 frame period is in Table 2.14.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	43,200	9,600	0	0	0	
Const (any)	0	0	0	0	0	
Any	81,598	0	76,798	9,600	0	
Σ	124,798	9,600	76,798	9,600	0	220,796

Table 2.14: Computational complexity of Despredator & Soft Bit Demapper in FDD-mode specific design for 1 frame and 1 channel

In Table 2.12, figures are for algorithm itself for despredating input signal with OVFS codes and demapping soft bits. Other complexity required for modeling purpose such as “outLast” Control and Buffer Full Detection has not been included.

Computational complexity of Despredator & Soft Bit Demapper in HSDPA-mode specific design which is normalised to 1 frame period is in Table 2.15.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	52,800*4	9,600*4	0	0	0	
Const (any)	0	0	0	0	0	
Any	82,758*4	0	81,598*4	4,800*4	0	
Σ	542,232	38,400	326,392	19,200	0	926,224

Table 2.15: Computational complexity of Channel Estimator & Multi-path Searcher in HSDPA-mode specific design for 1 frame

In Table 2.15, note that the figures are multiplied by 4, the number of codes, and so the complexities of Despredator & Soft Bit Demapper in FDD and HSDPA-mode specific design are almost the same after normalized to 1 code.

2.6 Descrambler (FDD/HS-DPA mode only)

Computational complexity of Descrambler in FDD-mode specific design which is normalised to 1 frame period is in Table 2.16.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	38,400	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	76,800	38,400	153,600	0	0	
Σ	76,800	38,400	192,000	0	0	307,200

Table 2.16: Computational complexity of Descrambler in FDD-mode specific design for 1 frame

In Table 2.16, all the figures are for algorithm itself for descrambling input signal with a scrambling code.

Computational complexity of Descrambler in HSDPA-mode specific design is exactly the same as in FDD.

2.7 Soft Bit Decoder (TDD mode only)

Concerning TDD mode, the FDD blocks described in paragraph 2.4, 2.5 and 2.6 (only related to FDD/HSDPA) are all included in the Joint Detection algorithm. This is a complex matrix computation, and we only provide the algorithm complexity. Before adding the complexity, let's introduce the step process.

The algorithm is based on a ZF-BLE criteria. This equalizer cancels the MAI. But another algorithm based on a MMSE criteria could be processed: no difference between noise and MAI is made. The MMSE equalizer needed the noise power estimation.

The received signal \mathbf{e} could be written $\mathbf{e} = \mathbf{A}\mathbf{d} + \mathbf{n}$, where:

- \mathbf{d} is the transmitted signal
- \mathbf{A} is a Matrix processed by the channel estimator, its size is $[(N \times Q + W - 1) \times J] \times [N \times K]$ where N is the number of symbols in the block, K the number of spreading code and Q the spreading factor, W the delay of the estimated impulse response, and J the sampling rate.
- \mathbf{n} the additive white gaussian noise.

The algorithm consists of finding the estimated vector \mathbf{d}' of \mathbf{d} thanks to the impulse channel response. The block diagram of joint detection algorithm is given in Figure 2.1.

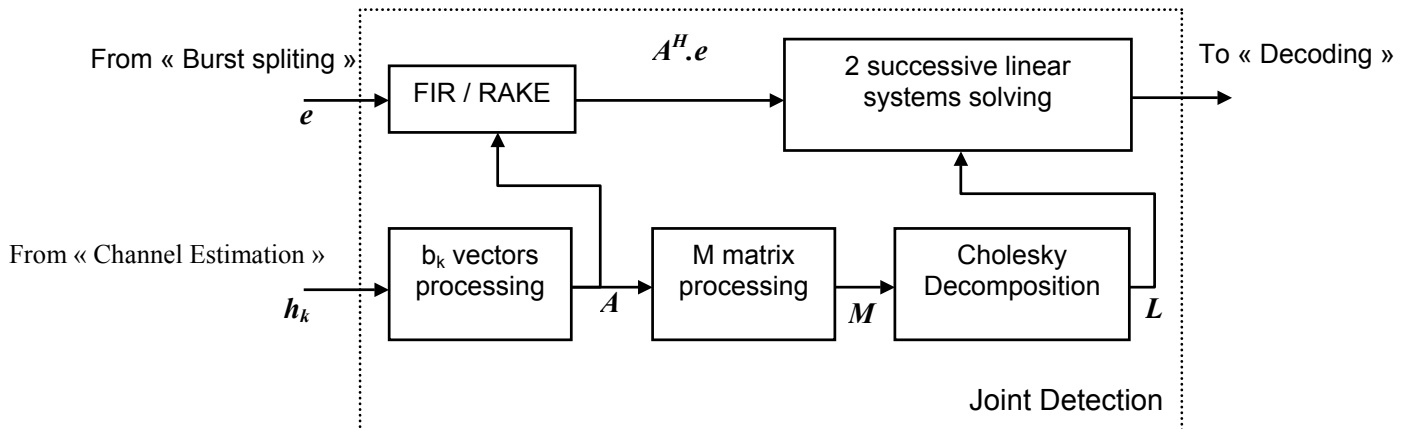


Figure 2.1 : Joint Detection Block Diagram

For that purpose, we compute the M matrix: $M = \mathbf{A}^H \mathbf{A}$ (the MMSE criteria leads to $M = \mathbf{A}^H \mathbf{A} + \sigma^2$). This matrix is represented on the Figure 2.2. The complex vectors $\mathbf{b}^{(k)}$ are processed by the convolution of the spreading code, the scrambler code, and the estimated impulse response of the channel.

Thus, calculation of $\mathbf{A}^H \mathbf{e}$ (see Figure 2.1) is despreader, descrambling and equalization in the same time.

We could assume that $\mathbf{d}' = \mathbf{M}^{-1} \cdot \mathbf{A}^H \cdot \mathbf{e}$. We state $\mathbf{z}_1 = \mathbf{A}^H \cdot \mathbf{e}$ (twice processing on the two data blocks), the vector \mathbf{z} is the output of the "rake" block. So, $\mathbf{z} = \mathbf{M} \cdot \mathbf{d}'$.

To get \mathbf{d}' , we have to process:

- Cholesky decomposition of $\mathbf{M} = \mathbf{U}^H \mathbf{U}$: $\mathbf{z} = \mathbf{M} \cdot \mathbf{d}' = \mathbf{U}^H \mathbf{U} \cdot \mathbf{d}'$. We only calculate the N_0 first blocks to simplify the algorithm. The last block is repeated until the end of the matrix. The results are not affected by this simplification.

Let's have $\mathbf{y} = \mathbf{U} \cdot \mathbf{d}'$, so that $\mathbf{z} = \mathbf{U}^H \cdot \mathbf{y}$. We solve the linear systems to obtain \mathbf{y} , then \mathbf{d}' .

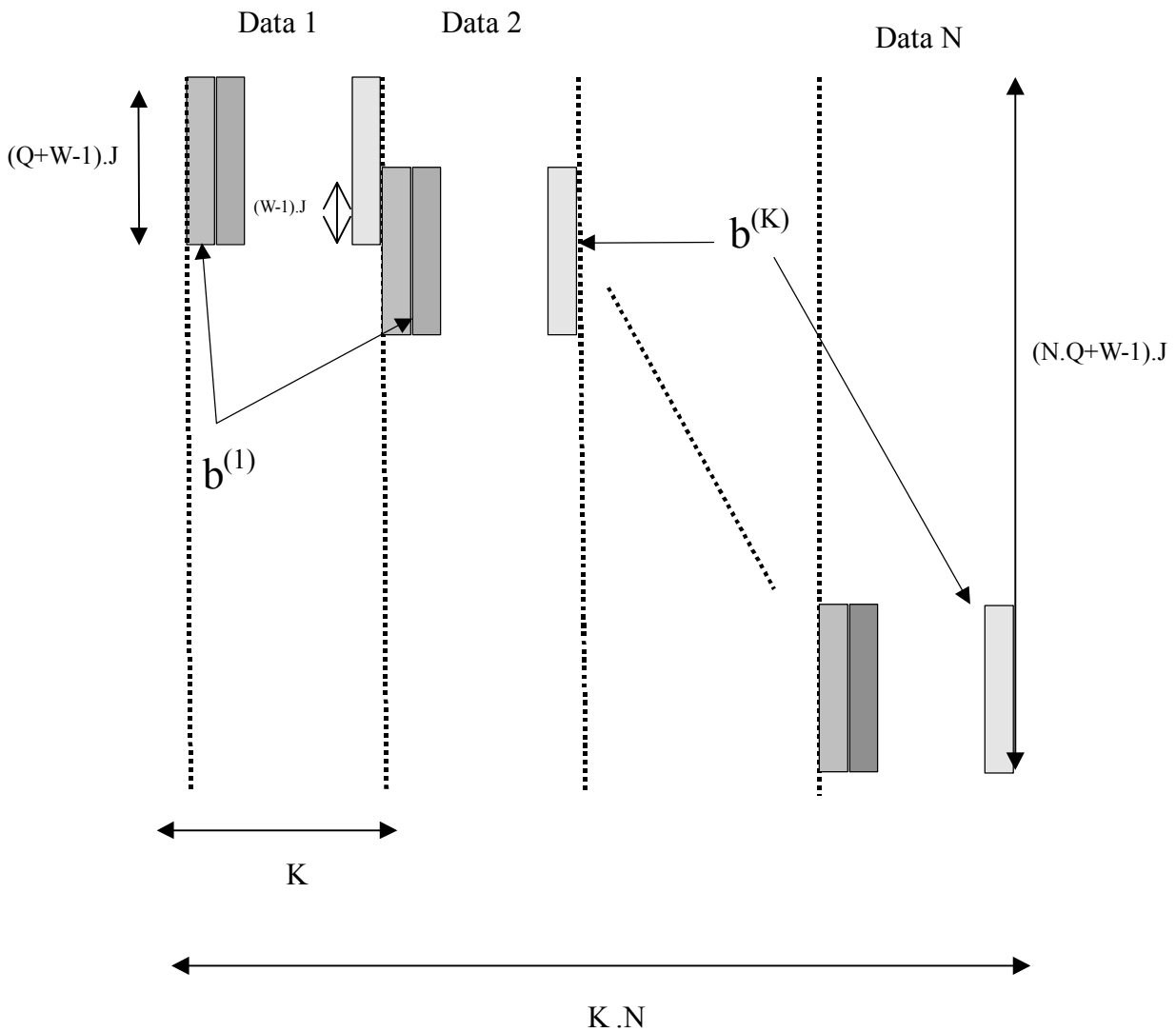


Figure 2.2 : Joint Detection Matrix

The parameters are (for burst type 2):

- Q** spreading factor (16)
- N** transmitted symbols (69)
- W** channel impulse response (57 chips)
- J** samples per chip (8)
- K** number of burst (8)

The computational complexity is calculated as follows:

$$\text{Let's have } P = \left\lceil \frac{Q+W-1}{Q} \right\rceil = 4.$$

State of the algorithm	Arithmetic operations	Square root	division
$M = A^h A$	$8.J.K^2(P+1)(Q+W-1 - PQ/2) = 819,200$	0	0
$z_1 = A^{*T} e_1$	$8.J.K.N(Q+W-1) = 2,543,616$	0	0
Cholesky Decomposition	$8 \left[\frac{(P+1)K(BK-1)BK}{2} - \frac{(BK-1)BK(2BK-1)}{6} + \frac{(B-P-1)}{2} \left(K(P^2K^2 - PK) + (2PK-1) \frac{K(K+1)}{2} + \frac{(K+1)K(2K+1)}{6} \right) + \frac{K}{2} \left(K^2 \frac{(P+1)P(2P+1)}{6} - K \frac{P(P+1)}{2} \right) \right]$ $= 124,976$	KB = 64	$BK \left(KP + \frac{(K-1)}{2} \right) = 2,272$
$\begin{cases} Ly = z \\ L^h \hat{d} = y \end{cases}$	$8 \left(2NK \left[K(P+1) - \frac{(K+1)}{2} \right] - (P+1)PK^2 \right)$ $= 303,296$	0	2KN = 1,104

Table 2.17 : Joint Detection complexity

Accordinging that the factor 8 for arithmetic operations is used to decompose a MAC in addition and multiplication (1 MAC = 4 additions and 4 multiplications), we can fulfil the following table:

	Add	Multiplic.	Division	sqrt	TOTAL
Any	1,895,544	1,895,544	3,376	64	
Σ	1,895,544	1,895,544	3,376	64	3,794,528

Table 2.18 : Joint Detection complexity - example

This complexity is given for one time slot. The computation resources are higher than a simple rake receiver, but this algorithm assumes a better multi-code (or multi-user detection).

2.8 Frequency Estimator

Computational complexity of Frequency Estimator in FDD-mode specific design which is normalised to 1 frame period is in Table 2.19. In this table, all the figures are for algorithm itself for estimating frequency and phase offsets using correlation and accumulation of training sequence.

Computational complexity of Frequency Estimator in HSDPA-mode specific design is exactly the same as in FDD, but we don't have it for TDD mode.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	49,681	0	0	0	0	
Const (1,-1)	246,778	0	0	0	0	
Const (any)	30	30	165	0	0	
Any	523,155	92,610	553,560	30	0	
Σ	819,644	92,640	553,725	30	0	1,466,039

Table 2.19: Computational complexity of Frequency Estimator in FDD-mode specific design for 1 frame

2.9 Feedback Frequency Synchronizer

Computational complexity of Feedback Frequency Synchronizer in FDD-mode specific design which is normalised to 1 frame period is in Table 2.20.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	308,641	154,321	617,282	0	0	
Σ	308,641	154,321	617,282	0	0	1,080,244

Table 2.20: Computational complexity of Feedback Frequency Synchronizer in FDD-mode specific design for 1 frame

In Table 2.20, all the figures are for algorithm itself for sample rate frequency offset compensation according to the estimated frequency offset.

Computational complexity of Feedback Frequency Synchronizer in HSDPA-mode specific design is exactly the same as in FDD, but we don't have it for TDD mode.

2.10 Forward Frequency Synchronizer

Computational complexity of Forward Frequency Synchronizer in FDD-mode specific design which is normalised to 1 frame period is in Table 2.21.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	76,800	38,400	192,000	0	0	
Σ	76,800	38,400	192,000	0	0	307,200

Table 2.21: Computational complexity of Forward Frequency Synchronizer in FDD-mode specific design for 1 frame

In Table 2.21, all the figures are for algorithm itself for symbol rate frequency offset compensation according to the estimated frequency and phase offsets.

Computational complexity of Forward Frequency Synchronizer in HSDPA-mode specific design is exactly the same as in FDD, but we don't have it for TDD mode.

2.11 DL Rx CCTrCH and TrCH Decoding

The figures in the tables below show the complexity values for the CCTrCH and TrCH decoding for the FDD (Table 2.23), the TDD (Table 2.24) and the HSDPA operation mode (Table 2.25). The figures are discussed in the subsequent chapters.

2.11.1 PhChannelDeMap (FDD and TDD)

The Physical Channel De-Mapper does not manipulate the actual data of the channels, but demultiplexes it and extracts information like TFCI, TPC and other control bits. Therefore this implementation includes some counters being mainly responsible for the increment (Inc) operations. This is caused by the streamed nature of the data processing in the model, creating data streams for the different bit streams. A memory-based structure would only have to generate start and end addresses of the respective data blocks in memory and provide them to the blocks using this data, reading directly from the memory. This has to be considered during the partitioning.

The TDD PhChannelDeMap is slightly more complex due to the fact that more than one physical channel (code) can be used for the same TrCH within the same time slot. The implementation of the mapping scheme includes some operations, which are not required in FDD case.

2.11.2 Second De-Interleaver (FDD and TDD)

The Second De-Interleaver is quite similar in FDD and TDD. The interleaving scheme is the same, though the data base may differ: Interleaving may either be done on frame or on slot basis in TDD, while in FDD there is only frame basis possible. But as this choice does not influence the total amount of Interleaved data (still *all* received data will be interleaved sometime), the number of operations does not heavily change and the types of required operations do not change at all. The operations are required for creating the interleaver metrics, i.e. the address for the data. This calculation is part of the algorithm and independent from the implementation.

The difference between the number of operations for FDD and TDD in the current test cases though the information bit rate is the same (384 kbps) is caused by a different amount of data to be interleaved due to different number of redundant bits (different puncturing). Thus the amount of data to be interleaved here is roughly, i.e. excluding CRC bits, turbo coder termination bits and DCCH processing:

$$2^{\text{nd}} \text{ IL data rate FDD: } 384\text{kbps} * 3 \text{ (TurboCode)} * 0.78 \text{ (Puncturing)} = 898.56\text{kbps}$$

$$2^{\text{nd}} \text{ IL data rate TDD: } 384\text{kbps} * 3 \text{ (TurboCode)} * 0.566 \text{ (Puncturing)} = 652.032\text{kbps}$$

Thus there is data rate relation between FDD:TDD of about 1.38:1 or TDD has about 27% less operations in the Second De-Interleaver. This difference is also valid for other blocks as we will see.

2.11.3 Second DTX Remove (FDD)

The operations (only increments) are used in a simple bit counter, which is implementation dependent and might be replaced by other means, e.g. address calculation.

2.11.4 TrChDeMux (FDD and TDD)

The Transport Channel De-Multiplexing is responsible for separating the different transport channels out of the Coded Composite Transport Channel (CCTrCH). The data bits themselves are not changed. The operations are required for bit counters in the streamed implementation. Immediate address calculation could replace some of those operations.

The different puncturing causes the difference in the figures for FDD and TDD. See chapter 2.11.2 for details.

2.11.5 First De-Interleaver (FDD and TDD)

For the First De-Interleaver the same is valid as for the Second De-Interleaver. There is again a difference in the data rate relation of FDD and TDD, because in FDD this First De-Interleaving is done with the still punctured data (before RateDematcher), in TDD the data has already been processed by the RateDeMatcher, when it reaches this block. Thus in the respective test cases (using puncturing in the RateMatcher for both FDD and TDD rather than repetition) in TDD mode the amount of data to be processed for each information data block is slightly bigger in TDD mode.

2.11.6 First DTX Remove (FDD)

The operations (nearly all are increments) are used in a simple bit counter, which is implementation dependent and might be replaced by other means, e.g. address calculation.

2.11.7 RateDeMatching (FDD and TDD)

Though the RateDeMatching algorithms is in principle the same for both modes, there are some differences in the calculation of the rate (de) matching parameters. Therefore in the TDD also bit shift operations appears, which are not counted in FDD case.

Also here the streamed nature of the model causes increment operations, which could be replaced by other means like address calculation in an architecture exchanging the information data bits via memory bank.

2.11.8 Bit-De-Scrambling (TDD)

The Bit-De-Scrambling algorithm, which is exclusively included in the TDD mode, has FIR structure and thus is based on bit shifting operations (Figure 4.6). This explains the number of divisions and multiplications with constants $c=2^n$, $n \in \mathbb{N}$, $n \neq 0$, which of course are bit shifting operations, either left or right shifts.

From complexity and data rate point of view this block can be easily implemented in SW, however as all bit-level operations this solution is quite inefficient. This problem will be discussed in more detail in chapter 4.

2.11.9 RadioFrameDeEqualization (TDD)

The operations of this block are caused by a bit counter. There are no modifications to the data stream.

2.11.10 HS-Physical Channel DeMapping

The High-Speed-Physical Channel De-Mapper does not manipulate the actual data of the channels, but de-multiplexes it. This implementation includes some counters being mainly responsible for the increment (Inc) operations. This is caused by the streamed nature of the data processing in the model, creating data streams for the different bit streams. A memory-based structure would only have to generate start and end addresses of the respective data blocks in memory and provide them to the blocks using this data, reading directly from the memory. This has to be considered during the partitioning, when some operations could be saved then.

2.11.11 HS-Rx-Constellation Re-Arrangement

The Constellation Re-Arrangement is transparent in QPSK and does also perform no operations for a certain parameter setting in 16QAM, which is selected in the present test case (constellation version parameter $b = 0$). However even with the other possible parameters settings, it is just a re-ordering of input steam and/or inversion of bits. As the swapping is only based on the four bits of one 16QAM constellation the required buffer is not big, i.e. in DSP realisation processor registers would usually be sufficient and in HW synthesized registers would be used, thus no memory cells as for the Interleavers would be required. This means that even in the case of other values of the constellation version

parameter b no address calculations would be required. There is also no complex operation done on the bits, thus the only kind of operation, which is counted here, is the increment of several loops in the model.

2.11.12 HS-DeInterleaving

The De-Interleaving is split into two subblocks in Table 2.25, the actual De-Interleaver and a multiplexer. In fact in 16QAM mode the input data stream is split in two, thus there are two interleavers required. After the interleaving the results are combined to form again one single data stream. This is done by the multiplexer. The interleaving itself is identical to the scheme used in the Second De-Interleaver for FDD. The required operations scale with the data rate or more exactly with the number of blocks to be interleaved.

The major portion of the DeMux operations is caused by a counter in a function for memory organisation (about 3 million increments). These operations are required for a stream based processing of the data, thus the appearance of these operations is implementation dependent. They are not considered in chapter 4 during the HW/SW partitioning. There the figures are reduced by the modelling overhead in terms of operations.

2.11.13 HS-RateDeMatching with H-ARQ

The H-ARQ functionality is tightly coupled with the Rate-DeMatching, which is implemented in two stages in HS mode. A soft bit buffer in-between decouples these two stages. Additional components of the H-ARQ implementation on the Rx side are the HARQ Bit DeCollection and HARQ Bit DeSeparation.

Similar to the HS-DeInterleaving the stream based processing of the data causes increment operations. Table 2.22 shows the complexity figures without these about 6 million increments and decrements in the First- and Second-DeRateMatcher.

Blockname	TOTAL	Additions and Subtractions				Others (Mult., Div., Mod.)		
		Inc/Dec	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Other (not const)
HS-HARQ (Total)	1,421,550	813,416	14,960	14,955	578,089	40	15	75
<i>HS-BitDeCollection</i>	<i>343,640</i>	<i>165,390</i>	<i>14,945</i>	<i>14,955</i>	<i>148,295</i>	<i>25</i>	<i>15</i>	<i>15</i>
<i>HS-Second-DeRM</i>	<i>292,910</i>	<i>173,388</i>	<i>15</i>	<i>0</i>	<i>119,457</i>	<i>0</i>	<i>0</i>	<i>50</i>
<i>HS-Rx-SoftbitBuffer</i>	<i>72,575</i>	<i>48,030</i>	<i>0</i>	<i>0</i>	<i>24,545</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>HS-First-DeRM</i>	<i>313,545</i>	<i>168,548</i>	<i>0</i>	<i>0</i>	<i>144,972</i>	<i>15</i>	<i>0</i>	<i>10</i>
<i>HS-BitDeSeperation</i>	<i>398,880</i>	<i>258,060</i>	<i>0</i>	<i>0</i>	<i>140,820</i>	<i>0</i>	<i>0</i>	<i>0</i>

Table 2.22: Computational complexity of HSDPA DL Rx HARQ

The HS-BitDeCollection is a (de)interleaver distinguishing the systematic bits and the parity1 and parity2 bits. This includes the calculation of the parameters for the interleaver and the interleaving itself (address calculation). Beside these activities this block also does the calculation of the RM parameters for the following blocks.

The Second-DeRM uses the redundancy version (RV) parameters to determine its functionality. The operations arise from the calculation of the bits to be punctured (in the selected testcase there is no repetition used). The Rx-SoftbitBuffer takes care of the address calculation (counting) for data buffering.

The First-DeRM is from functionality and parameter point of view similar to the FDD RM for turbo coded data. Some specific parameter values and limitation are applied here.

The HS-BitDeSeparation does merge the data streams made of systematic bits and parity bits to provide them to the Turbo Decoder. Address calculations and counters for the different data streams cause a number of operations here.

2.11.14 Viterbi decoder (FDD and TDD)

The Viterbi decoder is used in the present test cases for channel decoding of the dedicated control channel (DCCH). Different complexity values in FDD and TDD mode are caused by different test case parameters for code rate and puncturing.

FDD: Coding Rate = 1/3; Puncturing = 22%

TDD: Coding Rate = 1/2; Puncturing = 15.3%

In total the Viterbi decoder requires a lot of operations. About half of them are bit shift operations, the other half additions and increments. Those are caused by the algorithm itself for branch metric calculation and the ACS units.

2.11.15 Turbo decoder (FDD, TDD and HSDPA)

The Turbo decoder can be optionally used for the channel decoding of the actual data path in all three modes. In HSDPA there is only turbo (de)coding possible, there is no alternative convolutional coder like in FDD and TDD. The operations of the turbo decoder are similar to those of the Viterbi, beside a great amount of additions, subtractions and increments there are a lot of bit-level shift operations. In terms of operations per data unit this block is in the order of two magnitudes bigger than the other blocks in the CCTrCH and TrCH decoding.

The number of operation scales well with the data rate (same in FDD and TDD; about factor 6 compared to HSDPA). The major part of the counted operations belongs to the actual algorithm for branch metric calculation, deinterleaving and ACS unit. Figure 2.3 shows the general structure of the Turbo decoder. Each SISO-Decoder is in MUMOR case based on log-MAP algorithm, i.e. a Maximum A Posteriori decoder implemented in the log domain for practical implementation reasons.

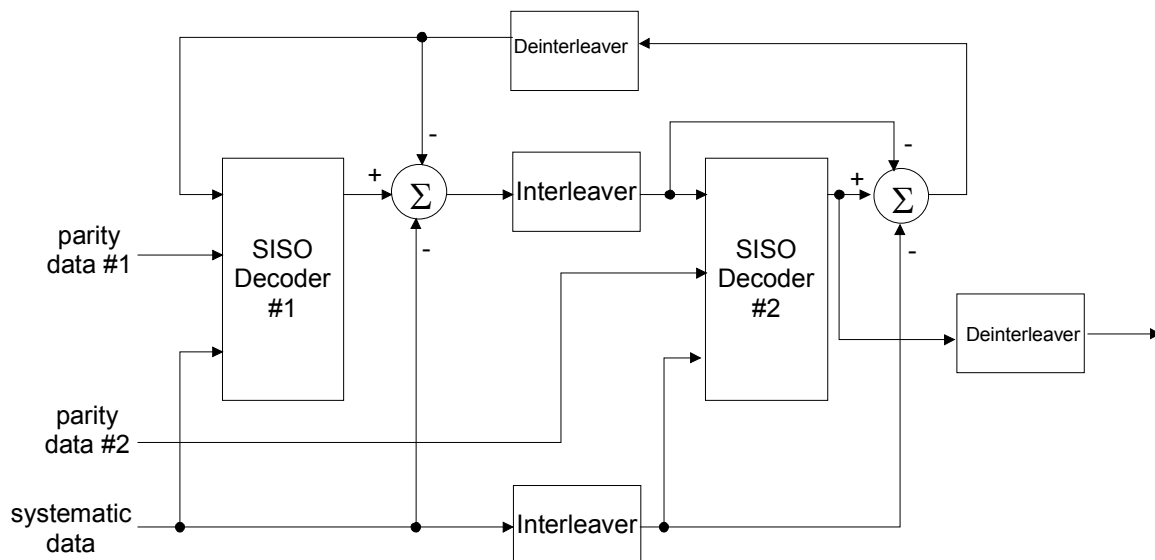


Figure 2.3: General Turbo decoder structure

2.11.16 CRC Check (FDD, TDD and HSDPA)

The CRC check is also used in all three modes. The operations of the CRC check are mainly bit-level shift and logical operations. Thus though the block does not require much operations compared to

others it might be more efficient (especially in terms of power consumption) implemented in HW than in SW. Find a more detailed discussion about these problems in chapter 4.

The number of operation of the CRC Check scales well with the data rate (same in FDD and TDD; about factor 6 compared to HSDPA). The major part of the counted operations belongs to the actual algorithm, there is not much overhead from modelling included.

2.12 Cell Searcher

The cell search block has not been included in the simulation for retrieving complexity figures. The reason is that this block is asynchronous compared to the data stream, i.e. after the initial detection of the cell code and group number it changes the mode. Thus a calculation of the complexity values has not been done. However there is an optimization of this block in terms of multi-mode combination of FDD and TDD described in chapter 3. For the HW/SW partitioning it can be estimated that depending on the implementation the cell search block requires at least, i.e. only for correlation more than 1000 operations per chip, resulting in a lower limit of 38.400.000 operations per frame. This figure can be used for HW/SW partitioning purposes.

Blockname	TOTAL	Additions and Subtractions				Multiplication			Division		Modulo
		Inc/Dec	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Other (not const)	Const (2 ⁿ)
PhChannelDeMap	48,144	48,099	0	45	0	0	0	0	0	0	0
Second De-Interleaver	63,903	45,661	0	18,240	1	0	1	0	0	0	0
Second DTX Remove	9,120	9,120	0	0	0	0	0	0	0	0	0
TrChDeMux	27,377	27,369	0	0	2	0	0	2	4	0	0
First De-Interleaver	63,793	54,635	0	0	9,156	0	1	0	1	0	0
First DTX Remove	9,122	9,120	0	0	1	0	1	0	0	0	0
RateDeMatching	54,529	44,169	0	0	10,360	0	0	0	0	0	0
Viterbi decoder (DCCH)	253,777	53,946	30	30	92,190	15,391	0	0	92,190	0	0
Turbo decoder (DTCH)	12,985,888	5,576,579	0	493,953	4,198,612	1,728,840	0	0	987,904	0	0
CRC check	38,712	3,907	0	0	1	3,865	0	0	30,939	0	0
TOTAL	13,554,364	5,872,604	30	512,268	4,310,323	1,748,096	3	2	1,111,038	0	0

Table 2.23: Computational complexity of FDD DL Rx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A3.4

Blockname	TOTAL	Additions and Subtractions				Multiplication			Division	Modulo	
		Inc/Dec	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Const (2 ⁿ)	Const (any)
PhChannelDeMap	81,033	54,108	0	0	13,562	0	0	0	147	13,216	0
Second De-Interleaver	46,407	33,145	0	13,260	1	0	1	0	0	0	0
TDD Bit-De-Scrambling	39,648	0	0	0	6,608	6,608	0	0	26,432	0	0
TrChDeMux	19,845	19,835	0	0	6	2	2	0	0	0	0
RateDeMatching	79,003	62,611	0	0	16,385	2	0	0	3	0	2
First De-Interleaver	69,861	58,188	0	0	11,670	0	2	0	1	0	0
RadioFrameDeEqualisation	12,678	12,678	0	0	0	0	0	0	0	0	0
Viterbi decoder (DCCH)	192,337	46,266	30	30	69,150	15,391	0	0	61,470	0	0
Turbo decoder (DTCH)	12,985,888	5,576,579	0	493,953	4,198,612	1,728,840	0	0	987,904	0	0
CRC check	38,712	3,907	0	0	1	3,865	0	0	30,939	0	0
TOTAL	13,565,411	5,867,316	30	507,243	4315996	1,754,708	5	0	1,106,896	13,216	2

Table 2.24: Computational complexity of TDD DL Rx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.102 Chapter A2.5

Blockname	TOTAL	Additions and Subtractions				Multiplication			Division		Modulo
		Inc/Dec	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Other (not const)	Const (2 ⁿ)	Other (not const)	Const (2 ⁿ)
HS-Physical Channel DeMapping	115,240	115,240	0	0	0	0	0	0	0	0	0
HS-Rx Constellation Re-Arrangement (16QAM)	38,400	38,400	0	0	0	0	0	0	0	0	0
HS-DeInterleaving	3,606,000	3,492,373	0	76,800	36,780	0	40	0	0	0	0
<i>Subblocks</i> <i>HS-DeInterleaver</i> <i>(2 instances used in 16QAM mode)</i>	<i>232,920</i>	<i>156,040</i>	<i>0</i>	<i>76,800</i>	<i>40</i>	<i>0</i>	<i>40</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>HS-SecDeMux</i>	<i>3,373,080</i>	<i>3,336,333</i>	<i>0</i>	<i>0</i>	<i>36,747</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
HS-HARQ	7,879,710	7,271,576	14,960	14,955	578,089	5	15	50	25	25	10
<i>Subblocks</i> <i>HS-BitDeCollection</i>	<i>343,640</i>	<i>165,390</i>	<i>14,945</i>	<i>14,955</i>	<i>148,295</i>	<i>5</i>	<i>15</i>	<i>5</i>	<i>15</i>	<i>10</i>	<i>5</i>
<i>HS-Second-DeRM</i>	<i>3,521,990</i>	<i>3,402,468</i>	<i>15</i>	<i>0</i>	<i>119,457</i>	<i>0</i>	<i>0</i>	<i>35</i>	<i>0</i>	<i>15</i>	<i>0</i>
<i>HS-Rx-SoftbitBuffer</i>	<i>72,575</i>	<i>48,030</i>	<i>0</i>	<i>0</i>	<i>24,545</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
<i>HS-First-DeRM</i>	<i>3,542,625</i>	<i>3,397,628</i>	<i>0</i>	<i>0</i>	<i>144,972</i>	<i>0</i>	<i>0</i>	<i>10</i>	<i>10</i>	<i>0</i>	<i>5</i>
<i>HS-BitDeSeperation</i>	<i>398,880</i>	<i>258,060</i>	<i>0</i>	<i>0</i>	<i>140,820</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>	<i>0</i>
Turbo decoder	78,927,840	33,894,095	0	3,002,245	25,519,140	10,507,880	0	0	6,004,480	0	0
CRC Check	233,580	23,575	0	0	5	23,320	0	0	186,680	0	0
TOTAL	90,800,770	44,835,260	14,960	3055600	26,172,420	10,531,205	55	50	6,191,185	25	10

Table 2.25: Computational complexity of HSDPA DL Rx blocks (TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A7.1.3

3 Applied Optimisation and Optimization Gain

In this chapter the kinds of optimisation, which have been applied to transform the mode-specific models into optimised multi-mode models will be described. Different approaches have already been introduced in [M_D3.2], here it is explained how they are best applied to the different components of the downlink receiver.

For the blocks, which will have reduced complexity due to a change in their implementation, e.g. by optimising on algorithm level, the complexity figures are given here accompanied by a description about the reason for this reduction. The given complexity figures for the optimised downlink blocks of course are based on the same test cases as in the previous chapter.

For some blocks or functional group of blocks the multi-mode optimisation will rather be a structural one, i.e. basically a sharing of resources. This means that the actual block will stay the same for a distinct mode or only slight modifications like multiplexers in the data line will be done. However this block will be re-used in different modes by sharing resources. In this discussion the HW/SW partitioning will partly be anticipated. This is required to allow at least a rough estimation about the gain achieved by this optimisation, because this gain depends very much on the implementation.

At the end of the chapter also complexity figures for the multi-mode optimised *uplink* transmitter will be given. These blocks will not be implemented during this project (on the demonstrator), thus the discussion is not so detailed as for the downlink components. There will also be no complete HW/SW partitioning for these blocks in this document. However some resources for blocks of the uplink chain can be shared with blocks in the downlink. Therefore also uplink figures are included here.

The complexity calculations for the uplink are calculated in a similar way as the downlink figures, i.e. based on a testcase. The testcases used for FDD and TDD in the uplink are described here (the HSDPA uplink control channel is not considered at all due to its low encoding complexity):

FDD

For the FDD complexity calculation the test case “UL reference measurement channel (384 kbps)” according to chapter A.2.4 of [3GPP25.101] has been selected. The details of this test case are given here:

Parameter	Unit	Level
Information bit rate	kbps	384
DPDCH	kbps	960
DPCCH	kbps	15
DPCCH Slot Format #1	-	0
DPCCH/DPDCH power ratio	dB	-11.48
TFCI	-	On
Puncturing	%	18

Table 3.1: FDD UL reference measurement channel, physical parameters (384 kbps)

Parameter	DTCH	DCCH
Transport Channel Number	1	2
Transport Block Size	3840	100
Transport Block Set Size	3840	100
Transmission Time Interval	10 ms	40 ms
Type of Error Protection	Turbo Coding	Convolution Coding
Coding Rate	1/3	1/3
Rate Matching attribute	256	256
Size of CRC	16	12

Table 3.2: FDD UL reference measurement channel, transport channel parameters (384 kbps)

TDD

For the TDD complexity calculation the test case “UL reference measurement channel (12.2 kbps)” according to chapter A.2.4 of [3GPP25.101] has been selected. The details of this test case are given here:

Parameter	Value
Information data rate	12.2 kbps
RU's allocated	2 RU
Midamble	512 chips
Interleaving	20 ms
Power control	2 Bit/user
TFCI	16 Bit/user
Inband signalling DCCH	2 kbps
Puncturing level at Code rate 1/3 : DCH / DCCH	10% / 0%

Table 3.3: TDD UL reference measurement channel, 12.2 kbps (chip rate processing)

For further details concerning the test cases refer to the 3GPP specification.

3.1 Multi-mode Rx Pulse Shaping Filter

Computational complexity of Downlink Rx Pulse Shaping Filter in FDD-mode of multi-mode design is in Table 3.4.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	19,752,960	4,938,272	9,876,448	0	0	
Const (1,-1)	20,215,887	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	39,814,428	9,876,448	20,061,532	0	0	
Σ	79,783,275	14,814,720	29,937,980	0	0	124,535,975

Table 3.4: Computational complexity of Downlink Rx Pulse Shaping Filter in FDD-mode of multi-mode design for 1 frame

Computational complexities of Downlink Rx Pulse Shaping Filter in FDD and HSDPA -mode of multi-mode design are exactly the same.

Computational complexity of Downlink Rx Pulse Shaping Filter in TDD-mode of multi-mode design is in Table 3.5.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	1,316,864	329,218	647,507	0	0	
Const (1,-1)	1,336,462	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	2,609,922	647,507	1,314,907	0	0	
Σ	5,263,248	976,725	1,962,414	0	0	8,202,387

Table 3.5: Computational complexity of Downlink Rx Pulse Shaping Filter in TDD-mode of multi-mode design for 1 slot

Note that the complexity of Downlink Rx Pulse Shaping Filter in FDD-mode (for 1 frame) of multi-mode design is almost 15 times of that in TDD-mode (for 1 slot) as we expected.

Because no further optimisation has been required for multi-mode Rx Pulse Shaping Filter apart from sharing it between modes, the complexity figures for FDD and HSDPA of multi-mode are almost the same as of single mode.

3.2 Multi-mode Time Controller

Multi-mode Time Controller is exactly the same for FDD, HSDPA and TDD modes because no further optimisation has been required for multi-mode Time Controller apart from sharing it between modes.

3.3 Multi-mode Channel Estimator & Multi-path Searcher

Computational complexity of Channel Estimator in FDD-mode of multi-mode design is in Table 3.6.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	37,772	35,640	282,850	0	0	
Const (1,-1)	73,709,774	0	0	0	0	
Const (any)	0	0	71,280	0	0	
Any	220,701,035	147,426,986	293,755,165	1,137	0	
Σ	294,448,581	147,462,626	294,109,295	1,137	0	736,021,639

Table 3.6: Computational complexity of Channel Estimator & Multi-path Searcher in FDD-mode of multi-mode design for 1 frame

Computational complexity of Channel Estimator in HSDPA-mode of multi-mode design is in Table 3.7.

Note that the complexity figures for FDD and HSDPA of multi-mode have been slightly increased compared to those for single mode because some overhead signal control regarding reconfigurability control is required to share the module in an appropriate way between modes and no further optimization have been required in algorithm point of view.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	37,891	35,520	282,719	0	0	
Const (1,-1)	73,699,058	0	0	0	0	
Const (any)	0	0	71,280	0	0	
Any	220,614,742	146,593,606	293,683,517	1,263	0	
Σ	294,351,691	146,629,126	294,037,516	1,263	0	735,019,596

Table 3.7: Computational complexity of Channel Estimator in HSDPA-mode of multi-mode design for 1 frame

Computational complexity of Channel Estimator in TDD-mode of multi-mode design is in Table 3.8.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	135,187	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	525,569	0	525,569	2,064	0	
Σ	660,756	0	525,569	2,064	0	1,188,389

Table 3.8: Computational complexity of Channel Estimator in TDD-mode of multi-mode design for 1 slot

Note that the complexity figures of Channel Estimator in TDD-mode of multi-mode design have been increased compared to those in single mode TDD because Steiner method of Channel Estimator in single mode requires less complexity, but we changed TDD Channel Estimation structure in multi-mode to make it similar to that of FDD/HSDPA for sharing common parts between modes by reconfigurability control and for getting gain in implementation point of view.

3.4 Multi-mode Rake Combiner

Computational complexity of Rake Combiner in FDD-mode of multi-mode design is in Table 3.9.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	966,644	38,284	0	0	0	
Const (1,-1)	1,736,986	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	2,894,391	966,644	3,866,576	0	0	
Σ	5,598,021	1,004,928	3,866,576	0	0	10,469,525

Table 3.9: Computational complexity of Rake Combiner in FDD-mode of multi-mode design for 1 frame

Computational complexity of Rake Combiner in HSDPA-mode of multi-mode design is in Table 3.10.

Note that the complexity figures for FDD and HSDPA of multi-mode have been slightly increased compared to those for single mode because some overhead signal control regarding reconfigurability is required to share the module in an appropriate way between modes and no further optimization have been required in algorithm point of view.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	1,432,080	39,460	0	0	0	
Const (1,-1)	2,203,016	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	4,331,050	1,422,080	5,688,320	0	0	
Σ	7,966,146	1,461,540	5,688,320	0	0	15,116,006

Table 3.10: Computational complexity of Rake Combiner in HSDPA-mode of multi-mode design for 1 frame

Computational complexity of Rake Combiner in TDD-mode of multi-mode design is in Table 3.11.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	265,696	6,623	2,327	0	0	
Const (1,-1)	312,835	0	0	0	0	
Const (any)	120	0	0	0	0	
Any	562,964	265,696	1,062,784	0	0	
Σ	1,141,615	272,319	1,065,111	0	0	2,479,045

Table 3.11: Computational complexity of Rake Combiner in TDD-mode of multi-mode design for 1 slot

Note that as mentioned in single mode case we changed the Joint Detection in single mode TDD into Rake Receiver in multi-mode TDD to make TDD Rx structure similar to that of FDD/HSDPA for sharing common parts between modes by reconfigurability control.

3.5 Multi-mode Despreader and Soft Bit Demapper

Computational complexity of Despreader and Soft Bit Demapper in FDD-mode of multi-mode design is in Table 3.12.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	43,055	9,568	0	0	0	
Const (any)	0	0	0	0	0	
Any	81,335	0	76,551	9,568	0	
Σ	124,390	9,568	76,551	9,568	0	220,077

Table 3.12: Computational complexity of Despreader and Soft Bit Demapper in FDD-mode of multi-mode design for 1 frame

Computational complexity of Despreader and Soft Bit Demapper in HSDPA-mode of multi-mode design is in Table 3.13.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	52,997*4	9,563*4	0	0	0	
Const (any)	0	0	0	0	0	
Any	82,879*4	0	81,865*4	4,987*4	0	
Σ	543,504	38,252	327,460	19,948	0	929,164

Table 3.13: Computational complexity of Despreader and Soft Bit Demapper in HSDPA-mode of multi-mode design for 1 frame and 1 physical channel

Note that the complexity figures for FDD and HSDPA of multi-mode have been slightly increased compared to those for single mode because some overhead signal control regarding reconfigurability control is required to share the module in an appropriate way between modes and no further optimization have been required in algorithm point of view.

Note that “*4” in the table means multiplication by the number of codes.

Computational complexity of Despreader and Soft Bit Demapper in TDD-mode of multi-mode design is in Table 3.14. Note that “*16” in the table means multiplication by the number of codes.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	1*16	
Const (1,-1)	1,243*16	276*16	0	0	0	
Const (any)	0	0	0	0	0	
Any	4,554*16	0	4,416*16	276*16	0	
Σ	92,752	4416	70,656	4416	16	172,256

Table 3.14: Computational complexity of Despreader and Soft Bit Demapper in TDD-mode of multi-mode design for 1 frame and 1 physical channel

Note that the complexity figures are after changing the algorithm of Joint Detection in single mode into Rake Receiver in multi-mode for TDD.

The main reasons for changing the algorithm are:

- to be closer to FDD mode in order to get higher level of reconfigurability.
- To enable hardware implementation. Actually, joint detection needs a specific study to be implemented in hardware. Cholesky decomposition is very sensitive to fixed point variable.

3.6 Multi-mode Descrambler

Computational complexity of Descrambler in FDD-mode of multi-mode design is in Table 3.15.

Computational complexities of Downlink Descrambler in FDD and HSDPA -mode of multi-mode design are the same.

Note that the complexity figures for FDD and HSDPA of multi-mode are exactly the same as those for single mode because no further optimization have been required in algorithm point of view.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	38,400	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	76,800	38,400	153,600	0	0	
Σ	76,800	38,400	192,000	0	0	307,200

Table 3.15: Computational complexity of Descrambler in FDD-mode of multi-mode design for 1 frame

Computational complexity of Descrambler in TDD-mode of multi-mode design is in Table 3.16.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	2,208	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	2,208	2,208	8,832	0	0	
Σ	2,208	2,208	8,832	0	2,208	15,456

Table 3.16: Computational complexity of Descrambler in TDD-mode of multi-mode design for 1 slot

3.7 Multi-mode Frequency Estimator

Computational complexity of Frequency Estimator in FDD-mode of multi-mode design which is normalised to 1 frame period is in Table 3.17.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	50,043	0	0	0	0	
Const (1,-1)	246,961	0	0	0	0	
Const (any)	30	30	165	0	0	
Any	524,155	92,893	553,889	30	0	
Σ	821,189	92,923	554,054	30	0	1,468,196

Table 3.17: Computational complexity of Frequency Estimator in FDD-mode of multi-mode design for 1 frame

Computational complexity of Frequency Estimator in HSDPA-mode specific design is exactly the same as in FDD. Note that the complexity figures for FDD and HSDPA of multi-mode have been slightly increased compared to those for single mode because some overhead signal control regarding reconfigurability control is required to share the module in an appropriate way between modes and no further optimization have been required in algorithm point of view.

Computational complexity of Descrambler in TDD-mode of multi-mode design is in Table 3.18.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	3,312	0	0	0	0	
Const (1,-1)	2,108	0	0	0	0	
Const (any)	2	2	12	0	0	
Any	4,416	542	4,648	2	0	
Σ	9,838	544	4,660	2	0	15,044

Table 3.18: Computational complexity of Frequency Estimator in TDD-mode of multi-mode design for 1 slot

3.8 Multi-mode Feedback Frequency Synchronizer

Multi-mode Feedback Frequency Synchronizer in FDD, HSDPA and TDD modes is exactly the same as those of single mode because no further optimisation has been required for multi-mode Feedback Frequency Synchronizer apart from sharing it among different modes.

3.9 Multi-mode Forward Frequency Synchronizer

Multi-mode Feedback Frequency Synchronizer among FDD, HSDPA and TDD modes is exactly the same as those of single mode because no further optimisation has been required for multi-mode Feedback Frequency Synchronizer apart from sharing it among different modes.

3.10 Multi-Mode DL Rx CCTrCH Decoding

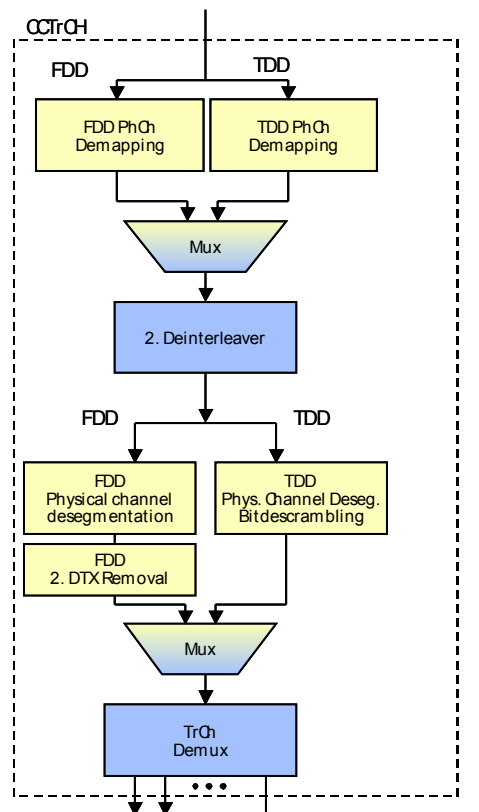


Figure 3.4: Multi-mode TDD and FDD Coded Composite Transport Channel block diagram

The CCTrCH block optimisation is based on the idea to reuse blocks and included functionality of TDD mode as far as possible in FDD mode and vice versa. Starting point of the analysis has been the fact that both signal paths for TDD and FDD mode are implemented in parallel. For this reason each

reduction of software and hardware by implementing multi-mode blocks reduces the signal path complexity. Comparison and evaluation of both signal paths led to the in Figure 3.4 resulting structure of CCTrCH block. Analysing block by block the functionality and parameters lead to the decision to extend the block capability to multi-mode functionality or to design one mode depended signal path by multiplexing.

3.10.1 PhChannelDeMapping

The TDD/FDD PhChannelDemapping blocks are not joined because the functionality of these blocks is too different. Thus to reuse parts of one design in a way to save one block by reusing most of the functionality of the other seems not possible. The computational complexity numbers for this block show that the used operations are different for TDD and FDD mode (see Table 2.23 and Table 2.24). In TDD mode increment, decrement, additions, divisions and modulo operators are used whereas in FDD mode increment and additions are used mainly. Also this gives the advice to less opportunity to optimise this block by implementing one multi-mode block. Finally the small overall complexity does not offer much potential for optimisation at all.

3.10.2 Second De-Interleaver

The 2nd DeInterleaver block used for TDD and FDD mode comprises mainly the same functionality. In TDD mode this block is using certain additional inputs and outputs (Frame Start and Frame End). The enhancement of this block to multi-mode functionality has been quite simple and does not extend significantly the hardware effort. From this follows the complete software necessary for one mode can be saved. The savings in SW have an impact on both the code size in the implementation and the development and especially verification time of the code. In case the block is implemented in hardware the necessary hardware of one block can be saved.

3.10.3 FDD Physical Channel Desegmentation

After the 2nd DeInterleaver follows the Physical Channel Desegmentation. Because of low block complexity and the fact that for TDD mode Physical Channel Desegmentation and BitDescrambling are combined in one block this block is not implemented as multi mode block. The computational complexity table does not include the results of this block because of low number of operations for all listed operations.

3.10.4 TDD Physical Channel Desegmentation and Bit Descrambling

This block is not a candidate combining FDD and TDD mode functionality in one multi-mode block. One reason is the Bit Descrambling is combined with the Physical Channel Desegmentation functionality as mentioned before. The other reason is the Bit Descrambling block is not used in FDD mode and so this combined block cannot be used in FDD mode.

3.10.5 FDD Second DTX Removal

This block is used in FDD mode only, because discontinuous transmission indication bits are not used in signal processing of TDD mode. From this follows no optimization is possible because no multi-mode functionality is necessary.

3.10.6 TrChDemux

The Transport Channel Demultiplexer TrChDemux block works identical in TDD and FDD mode. Mode dependent the output signals are multiplexed to the input of this block. Also for this block one time the hardware or software can be saved.

3.11 Multi-mode DL Rx TrCH Decoding

The TrCH (downlink) block optimisation has been done in two steps. First observe reusable functionality of the sub blocks and second combine the signal path for multi-mode purpose via multiplexer to get the wanted sub block combination, because TDD and FDD mode requires the multi-mode block functionality in different orders. Figure 3.5 shows the multi-mode TrCH block diagram. Only two sub blocks of TrCH block could not be adapted to multi-mode functionality. These blocks are the FDD 1st DTX Removal and the TDD Radio Frame De-equalization. All other blocks have been redesigned to join TDD and FDD functionality or have not been changed, because their functionality is the same for both transmission modes anyway.

Three multiplexers have been added to connect the sub blocks in mode dependent signal path order. The blocks that are not used in a specific mode are disabled.

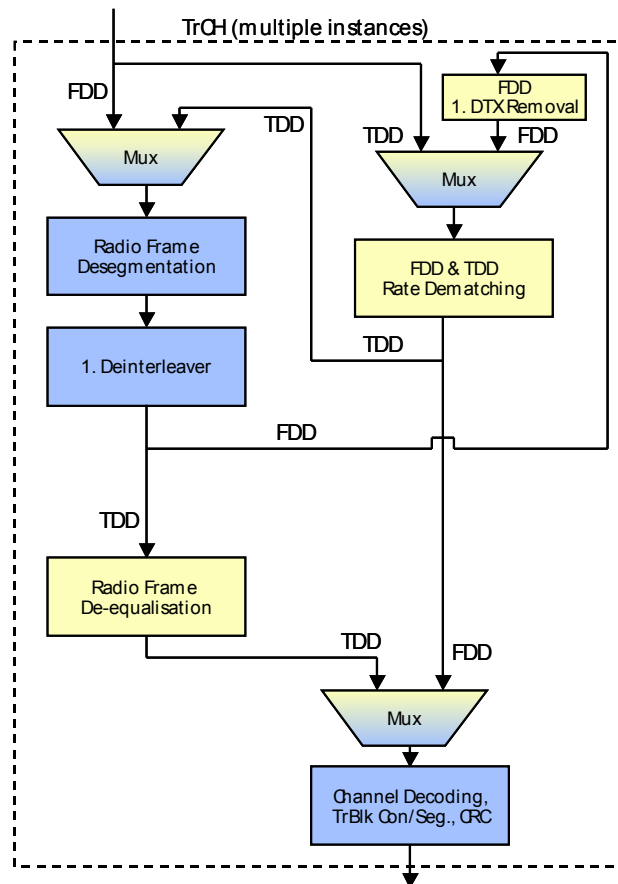


Figure 3.5: Multi-mode TDD and FDD Transport Channel TrCH (downlink) block diagram

3.11.1 Radio Frame Desegmentation

The block functionality of the Radio Frame Desegmentation is multi-mode capable. No changes have been done to adapt this block to both modes. The necessary differences are realized by parameters. From this follows the software for one mode is saved. In case this block will be implemented in hardware the combination of signal paths by multiplexing will save the complete hardware effort for one mode.

3.11.2 First DeInterleaver

The block functionality of the 1st DeInterleaver is multi-mode capable. No changes have been done to adapt this block to both modes. From this follows the software for one mode is saved complete. In case this block will be implemented in hardware the combination of signal paths by multiplexing will save the hardware effort for one mode

3.11.3 FDD 1st DTX Removal

1st DTX Removal block is responsible for the insertion of discontinuous transmission indication bits. This functionality is used in FDD mode only. For this reason there is no possibility to use this block in both modes thus no optimization is possible.

3.11.4 Rate Dematching

Rate Dematching is used in TDD and FDD mode. This block can be separated into two functionalities:

- Dynamic internal parameter calculation out of mode dependent parameter: This part is implemented direction dependent for FDD and TDD mode. The parameter calculation of TDD uplink and downlink direction is the same as for FDD uplink. In case of FDD down link direction it deviates from the other modes. The parameter calculation is different, not the parameter itself. For this reason the parameter calculation is extended by this implementation part to realize multi mode block capability. Consequently the parameter calculation is mode dependent in the multi-mode Rate Dematching block. The parameters only have to be calculated, when there is a change in the configuration of the transmitted channels, otherwise these parameters stay constant.
- Puncturing and repetition of chips: These parts of FDD and TDD Rate Dematching block are the same, because the functionality depends only on parameters calculated by the dynamic internal parameter calculation. From this follows FDD and TDD functionality can be joined in one multi mode Rate Dematching block.

The block optimisation of this block is realized by reducing in case of hardware implementation the hardware for puncturing and repetition parts in case of software the code for this parts for one mode, reducing design time and device cost.

3.11.5 TDD Radio Frame DeEqualization

The Radio Frame DeEqualization is used to remove the added fill data inserted by the Radio Frame Equalizer. In case of not certain integer multiple number of input data samples matches into certain period length fill data will be added. This block is not used in FDD mode thus no multi-mode optimisation is possible.

3.11.6 Channel Decoding

This block combines Channel Decoding, transport block concatenation, code block segmentation and CRC displacement. The Channel Decoding multi-mode block (viterbi and turbo decoding) is for TDD and FDD mode identical.

The computational complexity number of down link transmission (see chapter 2.1) of TDD and FDD mode shows the Turbo Decoder is the block with highest operation numbers. For this reason this block is one of the best candidates to observe for hardware / software optimisations.

The observation of operation numbers shows, nearly 42.94% (5,576,579) of all Turbo Decoder operations are “increment / decrement” operations with less computational effort compared to other listed operations. For this kind of operation the optimisation gain is not high.

Whereas the next frequently executed operations are the “add / subtract of not constant values”, with 32.33% of all operation numbers. One possible optimisation is the sharing of components like add /subtract blocks. The Turbo Decoder is using data on symbol rate. The clock rate of the hardware is “chip rate * over sampling rate”, therefore it must be possible to serialize operations which are done in parallel. The higher the factor of clock rate over symbol rate the more components can be saved.

Multiplication (13.31%) and division (7.60%) appear only as “const 2ⁿ” operation. These operations can be implemented on DSP as n times shift left for multiplication and n-times shift right for division. IN hardware this will rather be a FIR structure with delay line. The operations with lowest execution number are the “add / subtract any constant value”.

The complete hard or software of one signal path of Channel Decoding block can be saved because of multi-mode capability.

3.12 Multi-mode Cell Searcher

The Cell Search (CS) block optimisation can be done on P-SCH matched filter, S-SCH correlator and the RAM sub blocks (Figure 3.6). The optimisation is done in two steps. First the functionalities used in FDD and TDD mode will be observed and second the multi-mode possibilities checked. Above-mentioned blocks are good candidates for sharing resources, because most of the functionality of TDD and FDD mode is similar. The last part of Cell Search block, the scrambling code search, is not included in this evaluation because it is not used in this project. It is assumed that the scrambling code is given so the determination is not necessary. Further the following description is partitioned to P-SCH and S-SCH related evaluations. In this chapter the hardware effort is calculated for the correlation blocks because these show the highest complexity. The necessary memory cells for storage of the correlation results will be calculated and in addition to this the hardware effort will be compared for TDD and FDD mode.

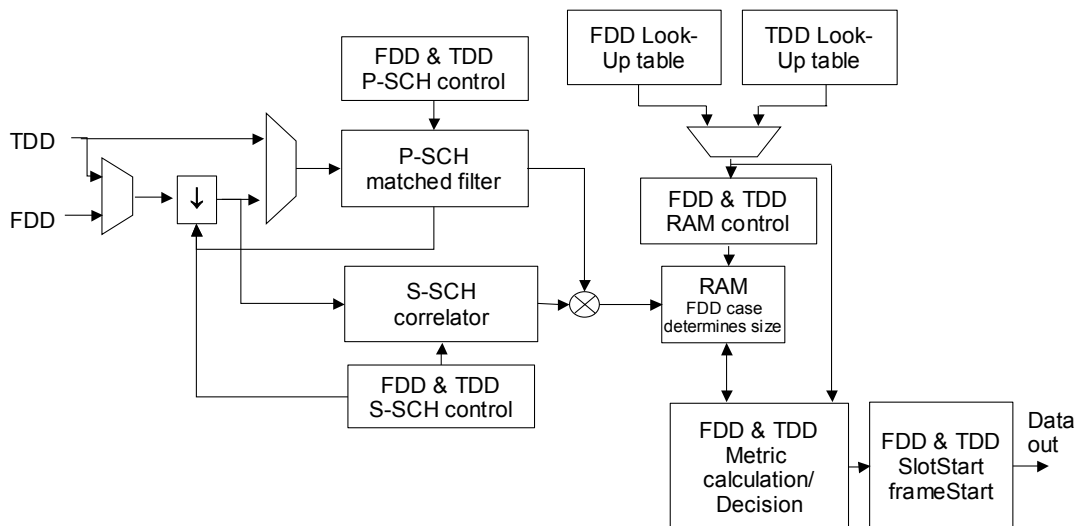


Figure 3.6: Multi-mode TDD and FDD Cell Search block (downlink) diagram

3.12.1 P-SCH

To combine TDD and FDD mode functionality into one multimode Primary Synchronization Channel P-SCH block the functionality is separated into P-SCH matched filter block and additional sub routines.

3.12.1.1 P-SCH Matched Filter

The P-SCH block observes the received signal to find the PSC n -times in certain period by matched filtering. These results are stored in a memory of certain length.

The block diagrams in Figure 3.7, Figure 3.8, and Figure 3.9 show the architecture of the P-SCH Matched Filter multi-mode block. The PSC is pre calculated to get sign sequence (Pg_0, \dots, Pg_{255}) used in Matched Filter block (Figure 3.7). In this figure the coefficients are numbered up to 63 only because the filter is divided into 4 blocks.

In case the I/Q data samples are over sampled the matched filter structure has to be adapted to this. Between two taps over sampling rate minus one delay blocks are included.

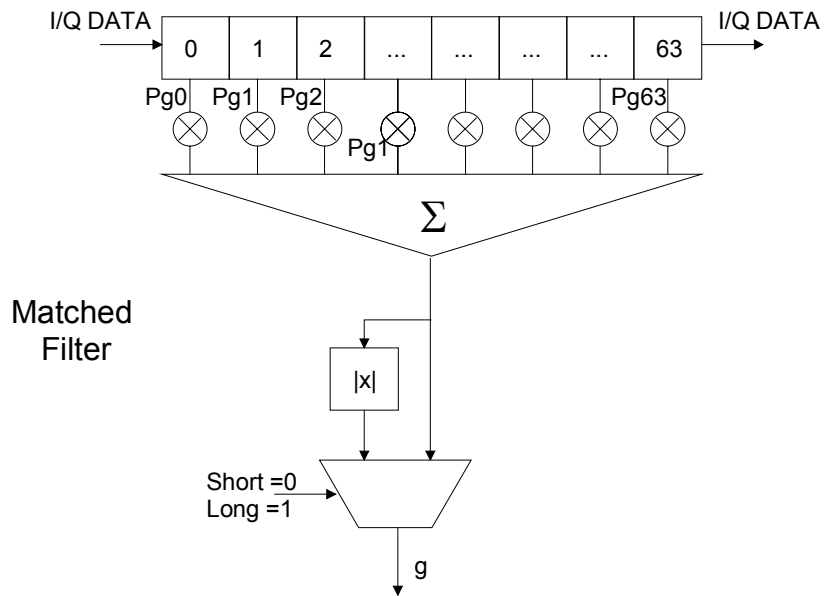


Figure 3.7: Block diagram of matched filter a sub block of P-SCH multi-mode block

A combined filter structure as shown in Figure 3.8 is chosen because in case of higher carrier frequency error the correlation result will be disturbed dramatically. To avoid the filter is divided in 4 sub sequences. Thus the correlation period is shortened by factor 4 so four times higher carrier frequency error is tolerable. The results $g_0 \dots g_3$ will be summed in Combined Matched filter block (see Figure 3.8).

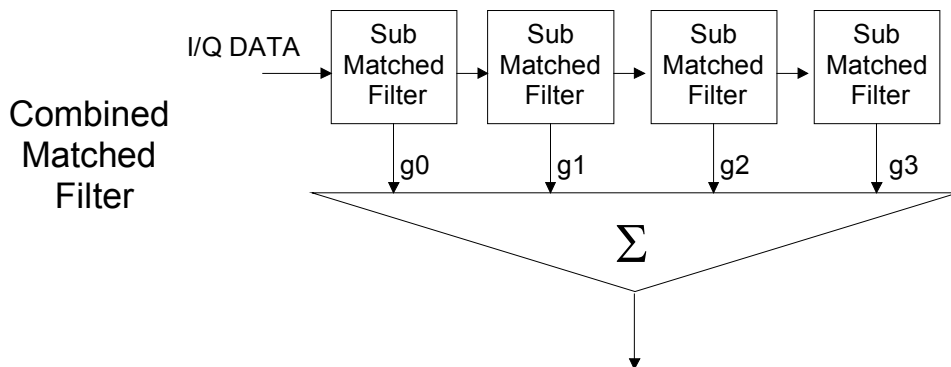


Figure 3.8: Block diagram of Combined Matched Filter of PSCH Cell Search

Figure 3.9 shows the block diagram of complete P-SCH Matched Filter block. In this part the I and Q separated correlation results are added and the absolute value is calculated. To get averaged correlation results a shift register of search period length N is included. The number of iterations to average the results before the control part of P-SCH enables the S-SCH correlator can be chosen by factor I .

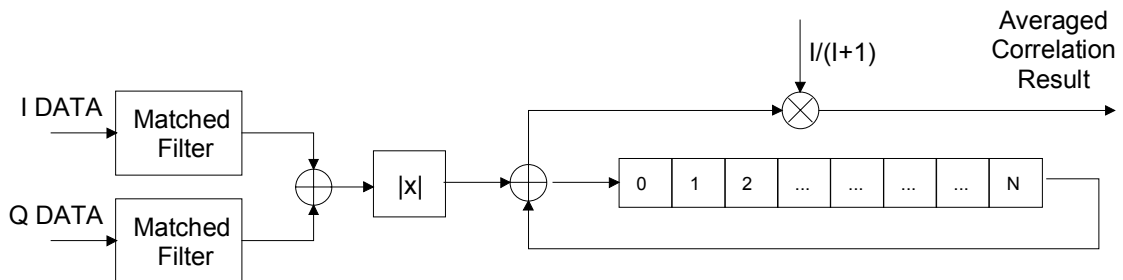


Figure 3.9: Block diagram of P-SCH Matched Filter

Differences and complexity of TDD and FDD mode:

- FDD: The number of correlation results to store in a memory is “ $2560 * \text{over sampling rate } K$ ”, because the PSC will be found in each time slot thus the search period is of this length. The hardware effort of the correlation is calculated to:

$$\begin{aligned} \text{matched filter} &= 2 * (4 * 64 * \text{adder} + 4 \text{ adder}) + 3 \text{ adder} + 1 \text{ multiplier} + 1 \text{ divider} \\ &= 523 \text{ adder} + 1 \text{ multiplier} + 1 \text{ divider} \end{aligned}$$

The number of used memory cells depends on the search period length and the over sampling rate: $2560 * K + 256 * N$
- In TDD mode the number of correlation results to store in a memory is “ $2560 * 15 * \text{over sampling rate } K$ ”, the PSC appears once or twice in one frame, thus the search period is one frame length. Because of less expected maximums in one frame compared to FDD mode the number of possible maximum candidates is increased to guaranty better results. This is done to increase the probability of correct detection. The hardware effort is:

$$\begin{aligned} \text{matched filter} &= 2 * (4 * 64 * \text{adder} + 4 \text{ adder}) + 3 \text{ adder} + 1 \text{ multiplier} + 1 \text{ divider} \\ &= 523 \text{ adder} \end{aligned}$$

The number of used memory cells depends on the search period length and the over sampling rate: $2560 * 15 * K + 256 * N$

The functionality of the P-SCH Matched filter in TDD and FDD mode is the same and following the hardware effort. The number of memory cells needed to store correlation results of FDD mode can be seen as subset of TDD mode. For TDD mode the number of necessary memory cells will be extended proportionally with the over sampling rate K . The more the sample rate is near to chip rate the more the amount of memory cells can be reduced. Disadvantage of this is the performance of the Cell Search block will decrease also. The higher the block performance should be the higher the hardware effort will be.

With this combination of FDD and TDD mode in one multi-mode block the complete FDD mode effort is reduced.

One additional and central optimisation aspect is the resource sharing of the memory cells. The Cell Search block is running in TDD and in FDD mode. In case FDD mode is enabled the allocated memory (in TDD mode 15 times the amount of FDD mode) is not used complete. The remaining P-SCH memory cells can be shared with other blocks, because the TDD mode memory resources are free except those cells required by FDD. Candidates to use these memory cells are all HSDPA mode blocks, which are not multi-mode blocks used in TDD or FDD mode, like the soft bit buffer in the HARQ unit.

3.12.1.2 FDD/TDD P-SCH Control

The P-SCH control block consists of a maximum search routine, a sort algorithm and additional control logic.

The maximum search routine observes in range of search period length a certain number of averaged correlation results for peaks (maximums).

Differences of FDD and TDD mode:

- FDD: The PSC is located in each time slot at the beginning. From this follows the maximum is located once in one time slot with a distance of exactly the time slot duration. One maximum per time slot period is expected. Hence the search routine has to find per time slot one absolute maximum
- TDD: As mentioned before the PSC is located once or twice in a frame. Therefore the search period length is one frame length. The position of PSC can differ with respect to time slot start position. The search routine is used to find several maximum candidates, because the number of expected maximums L in one search period is not known at this stage. Therefore the search

routine is implemented to find local maximum instead of absolute maximum like in FDD mode. An additional sort routine is used to sort the local maximum candidates of one search period per maximum value and index.

The search routines of TDD mode can be used for FDD mode, too. Via certain switches like search period length and the number L of maximum candidates the FDD mode functionality can be seen as a special case of TDD. Thus the hardware or software of the FDD mode for this block can be saved.

The state machine and control block:

The state machine enables in dependence to the number of to be observed correlation periods the S-SCH block at certain chip positions. TDD and FDD mode differences are:

- FDD: The PSC is located once in one time slot. Hence 15 times per frame a maximum is expected. For this reason the number of frames to average differ in comparison to TDD mode. In addition to this the over sampling rate can be reduced because of the higher number of expected correlation peaks in one frame.
- TDD: The PSC is located once or twice (optional) in one frame. The enabling is done as it is done for FDD mode after a certain number of received search periods. The difference to FDD mode is instead of one time the enable as start point for S-SCH L times at maximum candidate position the S-SCH is started. The over sampling rate will differ in comparison to FDD mode because it is not known if all the maximum candidate positions are in over sampling rate distance located or not.

The different requirements of these routines can be joined to one multi-mode P-SCH. In this the FDD functionalities can be seen as a subset of TDD's. From this follows the hardware for FDD mode can be saved nearly complete.

3.12.2 S-SCH

The Secondary Synchronization Channel S-SCH Correlator block used for FDD and TDD mode has to be divided in sub routines to allow optimisations and implementation of one multi-mode block. These routines can be reused by parameter settings and mode switch. The common subroutines are received data down sampling combined with additional controlling, the combined correlation to find the S-SCH and the carrier frequency error correction. The complexity evaluation is done by means of hardware effort of operands and memory cells.

3.12.2.1 FDD and TDD Controlling

This block is used to mark the best fitting data symbols out of the over sampled data. Because of timing frequency error the correct data samples have to be observed and down sampled to feed S-SCH correlation with best fitting data symbols. Therefore P-SCH Matched filter correlation result is used to find maximum peak as candidate for correct best sample as start point for S-SCH correlation. In addition to this a state machine is necessary to mark the correct sample point of time out of one symbol time period. Choosing the maximum peak out of correlation results does this and thus the point of time that fits best. As mentioned before because of different accuracy of averaged correlation results in both modes the over sampling rate will differ. TDD and FDD mode differences are:

- FDD: One maximum candidate per timeslot is expected. The down sampling rate will be higher as in TDD mode to come to chip rate directly. In addition to this the sample point taken after down sampling can be shifted to get the best fitting correlation results.
- TDD: The number of enables per frame is as high as the number L of maximum candidates, which have to be taken into account. The down sampling rate can be adjusted as in FDD mode but it is expected that P-SCH matched filter has to run on sample rate and for this reason the S-SCH correlator has to run on sample rate also. It will be enabled at the start point given by P-SCH control. From this follows that each sample is given to S-SCH correlator. The S-SCH correlator is running on chip rate realized by certain enabling.

From this follows the down sampling can be used for both modes in case the down sampling rate is adjustable so the FDD mode is seen as sub functionality of TDD mode. Further controlling differs in number of enable of S-SCH and the correlation period length.

The S-SCH correlator in TDD mode has to be instantiated L times because the distance between possible maximums thus S-SCH start is not known and can be shorter as necessary (see correlation period). For this reason the S-SCH correlator has to be placed L times in parallel.

3.12.2.2 S-SCH Correlator

The S-SCH correlator is implemented as a three-stage correlation. It is used to find S-SCH by correlation of SSC with received data and de-rotation of S-SCH correlation results by found frequency error, which is seen as phase shift on the P-SCH correlation results. Differences and complexity numbers of FDD and TDD mode:

- FDD: The S-SCH correlator is used once per time slot. The correlation period length is of SSC length = 256. The expected hardware effort is:

Stage 1: 1 adder
 Stage 2: 2 * 16 adder
 Stage 3: 16 * (2 adder + 2 Multiplier)
 correlation = 65 adder + 32 Multipliers
 memory cells = 65

Store and calculate average of correlation results:
 15 time slots with S-SCH, 16 SSC's, 240 correlation results per frame (search period)
 average = 16 adder + 16 multiplier + 16 divider
 each result in one memory cell for 15 time slots so
 memory cells = 240

- TDD: L maximum candidates have to be taken into account in this mode. From this follows L enablings for 2 expected S-SCH in one frame. The correlator has to be instantiated L times.

Stage 1: L * 1 adder
 Stage 2: L * 14 adder
 Stage 3: L * 14 (2 adder + 2 multiplier)
 correlation = L * (43 adder + 28 multipliers)
 memory cells = L * 14 * 2

Store and calculate average of correlation results:
 2 time slots with S-SCH, L * 2 * 14 SSC correlation results per frame (search period)
 correlation = 14 * adder + 14 multiplier + 14 divider
 each result in one memory cell for 2 time slots so
 memory cells = L * 2 * 14

General functionality of the correlation is the same so the outcome of this in FDD and TDD mode can be implemented as one multimode block. The hardware or software of one mode can be saved. In dependence of the number L of maximum candidates in TDD one mode can be seen as subset of the other. Differences are only in number of codes to correlate and in number of possible candidates where the S-SCH correlation is started and the correlation results have to be saved.

3.12.3 RAM

The RAM block is used to store the calculated S-SCH correlation results. The RAM has to have the highest necessary number of memory cells depending on which mode uses higher numbers. The other mode is using only a subset of this area.

3.12.4 Cell Search Summary

All remaining parts of the Cell Search block like FDD/TDD look up table, RAM control, Metric calculation and Decision and Slot / FrameStart should be implemented separately because multi-mode optimization is not possible without high implementation effort. The different strategies of TDD and FDD mode to include the SSC into signal stream and to combine SSC to built code groups leads to extremely different algorithms. From this follows the effort to adapt the blocks for both modes seemed to be not meaningful.

The major optimization of this block is to use functionality of one mode as a subset of the other. The P-SCH Matched Filter block gives the possibility to share memory resources with other blocks as mentioned before. The control structures for the P-SCH part can be multi-mode used but for the S-SCH part this is valid only for smaller functionalities.

3.13 Multi-mode Uplink Tx CCTrCH and TrCH Encoding

Table 3.19 depicts the complexity of the FDD uplink transmitter blocks of the CCTrCH and TrCH encoding, i.e. the bit and symbol rate encoding. Comparing the figures with those of the terminal receiver, I can be found that there are two kinds of functionalities included: Some are symmetrical on Tx and Rx and some are very unsymmetrical in terms of their complexity. Though it must be noted that of course the specified coding in UL Tx and DL Tx are not identical, so this comparison must be made with care. Nevertheless here some examples, which show symmetrical behaviour, thus with also the same algorithm applied here the implementation may be shared in the equipment for DL and UL as long as the total performance of the used components is sufficient for this quasi-simultaneous use, i.e. in time multiplex.

Blockname	TOTAL	Additions and Subtractions			Multiplication		Division and Modulo	
		Inc/Dec	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Const (2 ⁿ)	Const (any)
CRC Encoder	38,712	3,907	0	1	3,865	0	30,939	0
Turbo encoder (DTCH)	61,737	15,447	0	0	7,712	0	38,578	0
Convolutional Encoder	665	245	0	0	30	0	390	0
Radio Frame Equalisation	11,673	11,670	0	0	1	0	1	0
First Interleaver	58,428	46,710	0	11,715	0	1	1	0
RadioFrameSegmentation	11,671	11,670	0	0	0	0	1	0
RateMatching	87,277	48,575	7,720	30,971	2	2	2	6
TrChMux	115,218	115,216	0	2	0	0	0	0
Ph Channel Segmentation	9,600	9,600	0	0	0	0	0	0
Second Interleaver	67,264	48,062	19,200	1	0	1	0	0
PhChannelMap	96,926	96,864	0	45	1	15	0	1
TOTAL	559,169	407,964	26,920	42,735	11,611	19	69,913	7

Table 3.19: Computational complexity of FDD UL Tx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.101 Chapter A2.4

The CRC Encoder is fully symmetrical on UL-Tx and DL-Rx. The same polynomials are applied and the same parameters and parameter ranges are valid. Furthermore the complexity figures show that the 384kbps downlink requires the same number and kind of computations as the 384kbps uplink. This offers potential for resource sharing of the CRC implementation for UL and DL as long as the

implementation can provide sufficient execution speed and other resources like data buffers to handle both directions. At least this sounds like a reasonable option to be considered in the implementation.

The same option can also be applied for the first and second interleaver, which are also symmetrical for uplink and downlink. The most valuable resource to be shared here is not the algorithm implementation, which anyway might be done in SW, but the required memory to perform the interleaving. Unless it is not used for other purposes like data decoupling of SW and HW implemented components the required memory can be shared between the link directions.

The complexity figures for the TDD uplink are given in Table 3.20. It is very important to notice that the selected test case only has an information data rate of 12.2kbps instead of 384kbps as used in most other test cases. The reason for this is that there is no comparable test case (reference measurement channel) defined for TDD UL. Also TDD mode CRC and both interleaver can be used in DL and UL in time division resource sharing mode.

Blockname	TOTAL	Additions and Subtractions			Multiplication		Division and Modulo	
		Inc/Dec	Const (any)	Other (not const)	Const (2 ⁿ)	Const (any)	Const (2 ⁿ)	Const (any)
CRC Encoder	1,506	171	0	1	147	0	1,187	0
Convolutional Encoder	3,610	1,318	0	0	160	0	2,132	0
Radio Frame Equalisation	494	492	0	0	1	0	1	0
First Interleaver	2,538	1,998	0	537	0	2	1	0
RadioFrameSegmentation	493	492	0	0	0	0	1	0
RateMatching	2,929	2,431	0	492	2	0	2	3
TrChMux	5,442	5,440	0	2	0	0	0	0
TDD Bit Scrambling	20,104	15,976	0	1,868	452	0	1,808	0
Second Interleaver	3,396	2,434	960	1	0	1	0	0
PhChannelMap	4,152	2,995	0	235	452	0	470	0
TOTAL	44,663	33,746	960	3,136	1,214	3	5,601	3

Table 3.20: Computational complexity of TDD UL Tx blocks (CCTrCH and TrCH decoding), 1 Frame, Testcase according TS25.102 Chapter A2.1

3.14 Multi-mode Uplink Bit Mapper and Spreader

Computational complexity of Uplink Bit Mapper and Spreader in FDD-mode of multi-mode design is in Table 3.21.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	0	0	0	0	0	
Const (1,-1)	67,248	9,648	0	0	0	
Const (any)	0	0	0	0	0	
Any	0	0	96,482	0	0	
Σ	67,248	9,648	96,482	0	0	173,378

Table 3.21: Computational complexity of Uplink Bit Mapper and Spreader in FDD-mode of multi-mode design for 1 frame

Computational complexity of Uplink Bit Mapper and Spreader in TDD-mode of multi-mode design is in Table 3.22.

	Add	Sub	..	Division	Mod	TOTAL
Const (2^n)	0	0	0	0	0	
Const (1,-1)	966	138	0	0	0	
Const (any)	0	0	0	0	0	
Any	0	0	4,692	0	0	
Σ	966	138	4,692	0	0	5,796

Table 3.22: Computational complexity of Uplink Bit Mapper and Spreader in TDD-mode of multi-mode design for 1 slot

Computational complexity of Bit Mapper and Spreader in FDD and TDD-mode of multi-mode design is almost the same as in single mode because some overhead signal control regarding reconfigurability control will be required to share the module in an appropriate way between modes and no further optimization will not be required in algorithm point of view, but by sharing Spreader apart from Bit Mapper we can reduce hardware size, e.g. in FPGA, or code size, e.g. in DSP.

3.15 Multi-mode Uplink Scrambler & Midamble Insertion (TDD)

Computational complexity of Uplink Scrambler in FDD-mode of multi-mode design is in Table 3.23.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	0	0	38,580	0	0	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	77,160	38,580	154,320	0	0	
Σ	77,160	38,580	192,900	0	0	308,640

Table 3.23: Computational complexity of Uplink Scrambler in FDD-mode of multi-mode design for 1 frame

Computational complexity of Uplink Scrambler and Midamble Insertion in TDD-mode of multi-mode design is in Table 3.24.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2^n)	0	0	0	0	2,572	
Const (1,-1)	0	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	8,768	7,136	9,344	0	0	
Σ	8,768	7,136	9,344	0	2,572	27,820

Table 3.24: Computational complexity of Uplink Scrambler in TDD-mode of multi-mode design for 1 slot

Computational complexity of Bit Mapper and Spreader in FDD and TDD-mode of multi-mode design is almost the same as in single mode because some overhead signal control regarding reconfigurability control will be required to share the module in an appropriate way between modes and no further optimization will not be required in algorithm point of view, but by sharing Scrambler apart from Midamble Insertion we can reduce hardware size, e.g. in FPGA, or code size, e.g. in DSP.

3.16 Multi-mode Uplink Tx Pulse Shaping Filter

Computational complexity of Uplink Tx Pulse Shaping Filter in FDD-mode of multi-mode design is in Table 3.25.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	19,752,320	0	4,938,112	0	0	
Const (1,-1)	15,740,187	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	20,061,080	4,938,112	10,184,856	0	0	
Σ	55,553,587	4,938,112	15,122,968	0	0	75,614,667

Table 3.25: Computational complexity of Uplink Tx Pulse Shaping Filter in FDD-mode of multi-mode design for 1 frame

Computational complexity of Uplink Tx Pulse Shaping Filter in TDD-mode of multi-mode design is in Table 3.26.

	Add	Sub	Multiplic.	Division	Mod	TOTAL
Const (2 ⁿ)	1,322,240	0	330,592	0	0	
Const (1,-1)	1,053,526	0	0	0	0	
Const (any)	0	0	0	0	0	
Any	1,343,030	330,592	681,846	0	0	
Σ	3,718,796	330,592	1,012,438	0	0	5,061,826

Table 3.26: Computational complexity of Uplink Tx Pulse Shaping Filter in TDD-mode of multi-mode design for 1 slot

Note that the complexity of Uplink Tx Pulse Shaping Filter in FDD-mode of multi-mode design is almost 15 times of that in TDD-mode as we expected.

No optimisation can be applied apart from using the same PSF for both modes.

4 Implementation Definition

Typically current implementations of complex systems are neither implemented completely as hardware solution nor purely in software being executed on one or more processors. The most feasible solution offering the best trade-off between the different design constraints and objectives is the implementation with an application specific embedded system, which consists of hardware and software tailored for a particular task. As an effect a special step has to be included into the system design flow to define, which parts of the system will be implemented in hardware and which parts will be done in software. Beside a pure mixture of HW and SW also some hybrid implementation approaches exist, like re-configurable HW or HW accelerators of processors. Then the partitioning can be sketched as in Figure 4.1. This chapter gives an introduction to this step of the design flow (section 4.1) and also presents the results of the partitioning of the investigated multi-mode system (section 4.2).

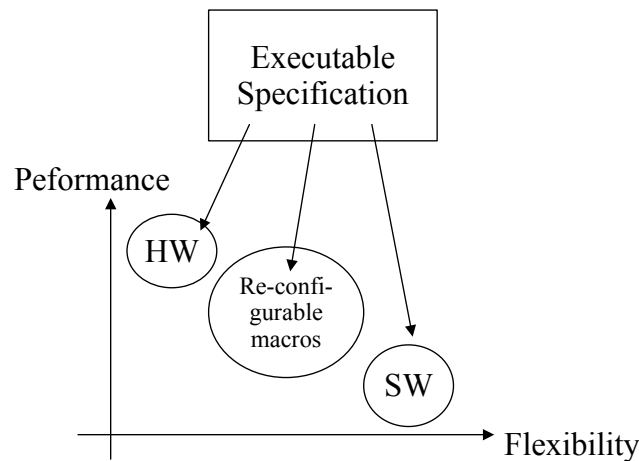


Figure 4.1: Hardware/Software Partitioning; Simplified trade-off: Performance vs. Flexibility

4.1 Hardware/Software Partitioning

During the introduction to the partitioning process given in section 4.1 only two targets are considered for implementation: Either hardware or software. Soft-configurable approaches as mentioned above are not explicitly covered here because they usually are a mixture of HW and SW implementation themselves and thus their properties and behaviour are also a mixture of the HW and SW properties presented here. Therefore it is not possible to give general rules concerning the design trade-offs as it is possible for pure HW/SW comparison. If such a hybrid approach should be considered it is required to know the specific implementation of the soft-configurable technology. Nevertheless in section 4.2.3 also hybrid technologies as introduced in [M_D3.2] will be considered for the actual mapping of the investigated system.

4.1.1 Basic Principles

The HW/SW partitioning can be defined as the process of deciding, if the required functionality is better implemented in HW or SW. The system functionality is described as a collection of indivisible functional objects (or components or subsystems) of which each one will be either implemented in HW or SW. Thus the implementation of the entire system will be a mixture of both. A formal definition of the HW/SW partitioning is done in [RASSP] using two sets H and S , where

$$H \subset O, S \subset O, H \cup S = O, H \cap S = \emptyset.$$

Figure 4.1 shows the partitioning process pretty much simplified as two-dimensional optimisation between “performance” and “flexibility”. However, in fact it is a multivariate optimisation problem considering design targets and technological constraints, which have to be applied for selected

subsystems and/or the entire system. These constraints can be understood as the *metrics* for the partitioning. The following ones are important to be mentioned:

- Performance (e.g. error rates at certain channel conditions that a receiver can provide)
- Speed issues (e.g. latency, processed data unit over time unit)
- Power consumption
- Reliability
- Flexibility of components to be used for different operation modes and purposes as well as for later updates
- Form factor
- Manufacturing cost of the final implementation

Beside the metrics capturing directly the capabilities of the final implementation there are also some factors defining the feasibility of the design process. To these factors belong:

- Availability of embedded processors for the target technology
- Availability of tools for co-design and co-verification to model the system during design process
- Development and design time (engineering, simulation, debugging, etc.)
- Development and design cost (personnel, development tools, emulators, trial systems, etc.)
- Capability of handling late changes in the system specification, i.e. flexibility during design time

The partitioning into HW and SW influences the overall system in terms of all these factors. Usually the hardware implementation provides higher performance via hardware speeds and parallel execution of operations and incurs additional expense of fabricating ASICs. On the other hand software implementation may run on high-performance processors at low cost (due to high-volume production), but incurs high cost of developing and maintaining (complex) software [RASSP].

The partitioning process can be initiated based on two perspectives:

- (1) SW oriented: All functionality will be done in SW by default and selected parts are moved into HW. These could be time- and power-critical ones.
- (2) HW oriented: By default all functionality will be done in HW and selected parts will be done in SW. These could be parts that are subject to change or are considered to be updated later, e.g. after first samples of the entire system are available.

The selection of the appropriate approach may be done based on the first experiences with the system complexity and the design constraints in mind. These first ideas may lead to an estimation about the implementation of some components or subsystems. For systems that are assumed to be mainly implemented in SW the first approach will be used and for other systems the HW oriented approach would be easier. Often it also may be useful to first separate the design into a small amount of subsystems and do the partitioning of each subsystem based on different approaches.

Beside the differentiation into HW and SW oriented approach, several techniques to actually do the partitioning decisions about each block or subsystem. Mainly three approaches can be distinguished [RASSP]:

- (1) Deterministic estimation: This method can only be applied, when detailed information about the system, the expected conditions it will run at and the components it will be build with are available. I.e. all parameters, which appear in the overall cost function leading to the partitioning results, are well known with sufficient accuracy. For complex designs with a lot

of dependencies between the cost functions for each subsystem this will rarely be the case. Nevertheless this technique leads to pretty reliable partition results if applicable.

- (2) Statistical estimation: In the case when the dependencies between cost functions for each subsystem could not be fully resolved or some parameters of the cost functions are not known in a sufficient accuracy a statistical estimation can be applied. This is based on analysis results or experiences of comparable systems or subsystems.
- (3) Profiling: If no figures of a comparable system are available the control and data flow of the system components or subsystems are profiled concerning their computational complexity. Computational expensive parts will then be implemented rather in a dedicated HW solution than in SW.

4.1.2 Design Flow

From HW developer perspective, hardware/software co-design has received increasing attention during the last years, as systems are getting more and more complex involving also embedded microprocessor and DSP design and implementation including memory and bus architectures. These kinds of systems are subject to many different types of constraints, including performance, size, weight, power consumption, reliability, and cost.

Most HW/SW co-design approaches are based on the fact that a single executable description of a system can nowadays be compiled into either silicon or machine code, and this opens up the possibility to uniformly describe the behaviour of a system. This description is then partitioned, with assistance from more or less automatic tools, into separate hardware and software parts. The timing constraints of some parts of an RT system can sometimes only be met by a hardware implementation whereas less time-critical parts can benefit from the advantages of a software solution, i.e. higher flexibility, faster implementation cycles, etc.

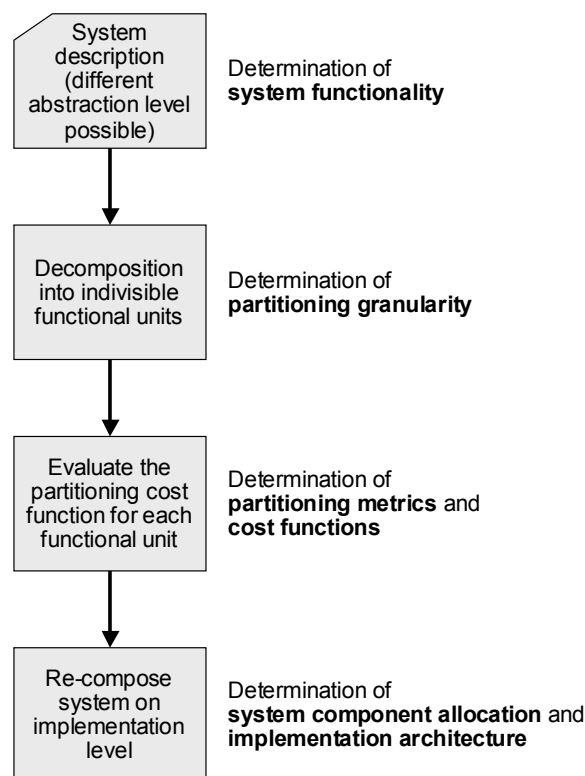


Figure 4.2: Structure of the plain HW/SW partitioning design flow

Figure 4.2 presents the structure of the HW/SW partitioning design flow starting with a system description on a high abstraction level. This system description determines the functionality of the

system to be implemented and also includes information about the targeted performance. In that sense it can be understood as an executable specification. The abstraction level of this executable specification can be different in different systems to be partitioned or even for different functional units within the same partitioning flow. Two contrary properties make the appropriate selection of abstraction difficult: A high level abstraction has the advantage to be really independent of the final implementation thus does not anticipate –neither explicit nor implicit– any partitioning decision, though the target was to postpone as many decisions as possible that place constraints on the design. However it still has to be guaranteed that the executable specification includes enough accuracy to include all properties of the system, which have influence on the partitioning process. If they are not included in this description they have to be borne in mind by some other means. On the other hand a detailed specification on a low abstraction level can more easily be equipped with all relevant information for the partitioning. But then it is difficult to not anticipate the partitioning decision by using some constructs, which will explicitly or implicitly already lead to a partitioning right before this step in the flow has actually begun.

Also the decomposition of the system into indivisible functional units is associated with an inherent trade-off in terms of the selected granularity, i.e. the partitioning effort vs. possible optimisation quality. The granularity of the decomposition is a measure of the size of the specification in each unit. A coarse granularity means that each object contains a large amount of the specification, whereas a fine granularity leads to many more units offering more possible partitions and also to better optimisation possibility. However the quality of the partitioning also depends much on the accuracy of the parameters of the cost functions. As long as these figures are do only have a coarse accuracy or some figures are entirely open the decision about the granularity should be taken favouring a coarser one, because the limiting factor of the optimisation quality then will most likely not be the selected granularity but the vague results of the partitioning cost function evaluation.

The determination of the partitioning metrics and the evaluation of the cost functions is the next step. Metrics are attributes, which describe the properties of a certain realisation of a functional unit in a quantitative way. As already introduced in the beginning of chapter 4.1.1 there are different kinds of metrics covering the design properties itself and metrics rather describing the design process. All these metrics have to be computed, i.e. a formal definition of the influence of each design or design process properties has to be generated. Several approaches can be used, which have an inherent trade-off between required effort and accuracy. First idea is to create a detailed implementation and produce accurate metric values based on the experiences made during the design process and by characterising the achieved implementation. Obviously this approach is impractical due to its time and cost consumption. Alternatively a rough implementation can be created, which includes the major register transfer components, but skips the details such as precise routing or optimised logic to save design time. This determination of metric values is called estimation. When even less effort is spent, it comes towards a “scientific guess”.

The metrics are used inside *objective functions*, which are a combination of metrics to capture the overall quality of a certain partitioning [Kuch02]. The multiple metrics such as power, latency, size, performance, etc are weighted against one another because of their different importance for the implementation. Usually this importance is defined by external constraints. The return value of the objective function is the *cost*.

An example of an objective function looks like this [Kuch02]:

$$ObjFunc = w_1 \cdot \sum_i \left(100 \cdot \frac{violate_area(Cl_i)}{max_area(Cl_i)} \right)^2 + w_2 \cdot \sum_i \left(100 \cdot \frac{violate_exectime(Cl_i)}{max_exectime(Cl_i)} \right)^2 + \dots$$

$$\text{with } violate_area(Cl_i) = \begin{cases} area(Cl_i) - max_area(Cl_i) & \text{if } area(Cl_i) - max_area(Cl_i) > 0 \\ 0 & \text{otherwise} \end{cases}$$

A cost function is a function $Cost(H, S, Cons, I)$, which returns a natural number that summarises the *overall quality* of a given partition [RASSP]. I contains any additional information that is not contained in H or S or $Cons$.

Beside the metrics described above there can be also metrics representing the benefit of grouping any two functional units into the same partition, called *closeness metrics*. These are used in a *closeness function*, which is then based on the *local* view of the system. Here an example for a closeness function expressing the closeness between two functions f_i and f_j [Kuch02].

$$Close(f_i, f_j) = w_1 \cdot \frac{cost(f_i) + cost(f_j) - cost(f_i \cup f_j)}{cost(f_i \cup f_j)} - w_2 \cdot part(f_i, f_j)$$

with $part(f_i, f_j) = \begin{cases} 1 & \text{if } f_i \text{ and } f_j \text{ can be executed in parallel} \\ 0 & \text{otherwise} \end{cases}$

Finally after the cost function has been evaluated the system components must be allocated and then the implementation architecture has to be defined. The first mentioned activity is the choosing of system component types from among those allowed, and selecting a number of each to use. The set of selected components is called an allocation. Usually various allocations can be used to implement a specification, differing in fulfilling the constraint parameters. A partitioning technique must designate the types of system components to which functional units can be mapped.

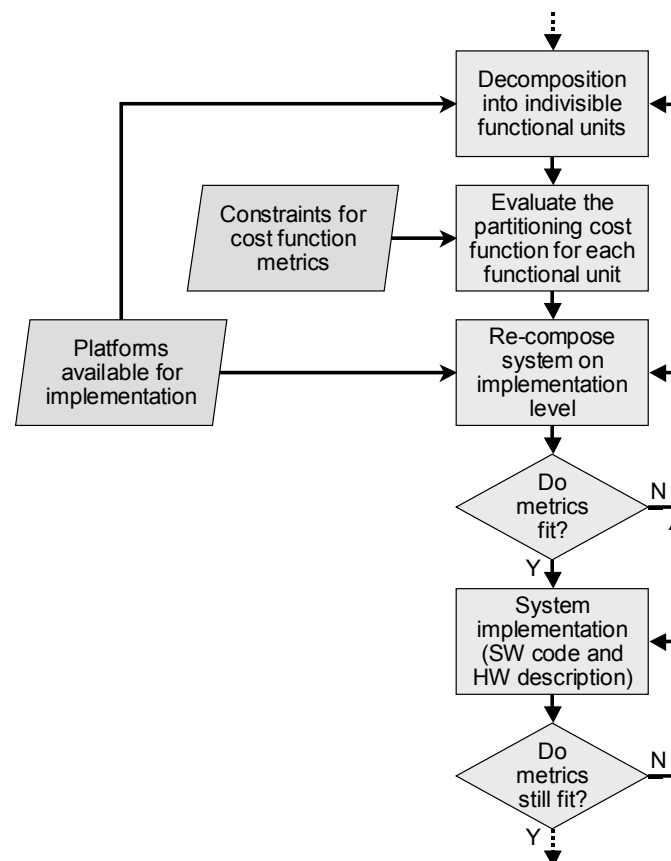


Figure 4.3: Iterative HW/SW partitioning design flow

In Figure 4.2 the flow looks rather straight forward than complex. However in reality the flow will have iterative aspects both locally, i.e. within each step, and globally, i.e. between the decision making steps as shown in Figure 4.3 to minimize the cost function and do the partitioning. Often exist

constraints concerning the final embedded system to be selected for the implementation, e.g. there is a limited set of “platforms” available, of which one has to be chosen in the end. So there is a limited degree of freedom for implementing the system. In these cases it is reasonable to have a look onto the available platforms already during the decomposition into functional units.

All these constraints make the optimal partitioning, i.e. the best way to partition the given set of functional units to the given set of system components, quite complex. Beside simulating all possible solutions, which is impractical because of the inordinate amount of computation and time required there are basically two kinds of systematic approaches to solve this problem: Manually guided and automatic [Kuch02]. The manually guided approach needs strong support from the design environment, like estimation tools and schedulers. During the automatic partitioning the design space has to be explored according to a certain strategy, which converges towards a solution close to one achieving the minimal cost function result. The automatic partitioning approaches are: Constructive Partitioning and Iterative Partitioning.

The constructive algorithms, e.g. hierarchical clustering, are bottom-up approaches in which each unit is grouped together with others to form a cluster. These clusters are initially built based on the closeness metrics and are then gradually merged until the desired partitioning is found. The computation time is spent constructing a small number of partitions. This approach does not require a global view of the system but relies only on local relations between objects.

The iterative algorithms, e.g. transformation based partitioning, are based on a design space exploration, which is guided by an objective function that reflects the global quality of the partitioning. A starting solution is modified iteratively, by passing from one candidate solution to another based on evaluations of an objective function. Typically one yields more accurate evaluations than with closeness functions used by constructive algorithms.

Examples are [Gajs94]:

- The Greedy algorithm moves objects or functions from software to hardware (and vice versa) as long as an improvement of the cost function occurs. This algorithm is simple and fast, but gets stuck in local minima. The cost function must take performance and hardware size into account.
- The Vulcan II algorithm ensures that performance constraints are met in the final partitioning. It starts with an all-hardware partitioning to guarantee this. After a successful move, it tries to move closely related objects next.
- There are various kind of Hill-climbing algorithms available, notably simulated annealing (SA). Usually they accept some negative moves to overcome local minima. In simulated annealing this level of negative moves is gradually decreased.
- The Cosyma approach uses simulated annealing; starting from an all-software solution and moving statements to hardware until constraints are met. The input specification may contain software constructs such as pointers.

In practice, a combination of consecutive and iterative algorithms is often employed.

4.2 Partitioning of the System under Investigation

4.2.1 Target Implementation Components

Each partitioning process maps the given specification, which in our case is an electronic specification in the shape of a SystemC model, onto two or more implementation component alternatives. These alternatives are introduced in this subchapter.

Important to mention right from the beginning is that within MUMOR no SoC will be developed. But there will be a HW demonstrator based on the components introduced in chapter 5. The physical

components used in the demonstrator system certainly differ from those, which would be used in a product. The focus when selecting the components for a demonstrator is on

- (fast) re-programming and re-configuring with big flexibility and
- extensive debug facilities,

whereas the components for (mass) product in the area of mobile communication will be selected mainly because of their

- low price per piece (NRE cost can be high compared to piece cost) and
- minimized power consumption.

The partitioning of this system will be done for the demonstrator components. The following components are the main building blocks for the demonstrator:

- Floating-point DSP: The most flexible solution, which needs fewest modifications of the C/C++ multimode code, when transferring the electronic specification into the design. But because it is unlikely to have such a unit in the mobile terminal and the cost of this component is high, the algorithms will be embedded in a floating point DSP only if it is necessary.
- Fixed-point DSP: To transfer the model code basically only the types of the FLP variables have to be changed, but not the structure of the algorithm. This will be the default for component for those parts of the design, which will be implemented in software.
- FPGA (fixed point): This is probably the best solution to implement blocks which need a lot of memory, or the application has timing requirements too tough for a DSP, or when it is a prototype to develop an ASIC/SoC after. The effort to map the functionality onto the DSP is higher than for the DSP solutions. The architecture has to be studied in detail, and a new code (VHDL) has to be written according to the chosen architecture.

In addition to these commonly known kinds of components there exist some implementations, which are based on a close coupling of processor and FPGA or even can be considered as a hybrid approach. Within MUMOR such an approach has been developed further to especially serve the requirements of a UMTS/HSDPA system. The approach has been introduced in [M_D3.2] in detail:

- Soft-configurable approach: The Algorithm Specific Instruction Processing (ASIP) accelerator includes a microprocessor, DSP, memory, I/O logic, user logic and one or more ASIPs, which represent the ASIP accelerator. The ASIP blocks take over the functionality of (additional) user logic and enhance the whole system in order to stay flexible after production by allowing a wide range of programmability of the algorithms that are executed but are too computation intensive to be solved by the microprocessor or the DSP.

Of course the application of the structure is depending on the available types of ASIP implementations, i.e. processing elements.

4.2.2 Partitioning onto HW and SW

Before going into the details for partitioning the system a preliminary classification will be done here to avoid mislead partitioning simply based on the complexity values. With this preliminary classification the different classic functions can be distinguished, and seen whether it can be embedded on DSP or FPGA. For that purpose, it is considered that the input of each block is in FXP format.

Filter functions

- FIR filter: The partitioning decision depends on the precision loss due to the quantification of the coefficients when leaving the FLP data format. It has to be taken care that the filter remains stable. Previous studies have shown that a 10-bit quantification is good for a SRRC

filter. The structure of a FIR fits easily on an FPGA, thus FIR filters are good candidates for FPGA implementation. The final decision depends on the required data rate.

- Matched filter (or correlator) for data despreading: The coefficients of this filter are only ± 1 , so FPGA implementation is suitable and very fast. The output of the sum must be divided. This operation of course is a critical part.

Complex algorithms

- Arithmetic function (cos, sqrt, $1/x$): These functions are already implemented on floating point DSP. But, if these functions are called in a fixed point module, they have to be rewritten in a fixed point way. Subfunctions in C language have to be written with “int” data type, or in VHDL writing when an FPGA is targeted for speed considerations.
- Matrix algorithms (decomposition, solving linear equation system, etc.): An FPGA implementation requires too much state machine to translate nested “for” loops, or a suitable architecture has to be studied and a solution for controlling the dynamic during each steps has to be found. A floating point DSP is the fastest solution in this case.

Memory functions

In this kind of functions, the overhead for an FPGA implementation consists of computing the read/write addresses, to schedule the process. For the memory access a state machine is often needed.

Operations on bit level

Algorithms doing a cyclic shift of polynomials with logical operations on bit-level could be done on a DSP but are very inefficient because normal DSPs do only allow operation on byte, word, or bigger data structures. Also register and memory accesses do not allow bit-level operations. Thus these kind of operations are preferably done on FPGA.

During the partitioning process done here the software oriented approach will be applied. Based on the functional classification made above and complexity figures retrieved for all functional units those which are very computational expensive have been selected for a hardware implementation. The first partitioning only considers implementation on either DSP or FPGA. In a second step also soft-configurable technologies will be applied where feasible.

The reason for applying the SW based approach is mainly to preserve the best flexibility of the design. This flexibility speeds up the design flow by simplifying the consideration of late design changes, e.g. when the terminal requirements change for example in terms of the supported capability parameters as defined in 3GPP. Even when certain blocks will themselves never be affected, it might be relevant for the neighbouring blocks, which also has an influence over the closeness metrics. The second aspect of using the flexibility of the SW approach, i.e. trying to map as much components onto SW as possible, is the flexibility for later updates and –especially important for multi-mode designs– being capable of augmenting the system towards other standards, including future ones. Especially algorithms, which only require low computational complexity but need much memory, may easily be updated for other systems. Changing the SW code itself usually has to be accompanied by an increased memory (assuming that the data rates will increase in future systems), but still this modification of the design is less complicated than a modification of the HW especially from verification point of view. The additional memory would then also be required in this case.

Another reason to map blocks to SW implementation is in case they include a complex state machine as their main purpose or just as an internal control structure. Such structures can hardly be

implemented in HW because of a complicated verification. High-level SW languages inherently offer more potential to describe and verify such structures.

The threshold C_{HW} used for the HW/SW-partitioning decision has been set to the value of

$$C_{Thw} = 500.000 \text{ OPs per frame, i.e. } C_{Thw} * 100 = 50 \text{ million operations per second (MOPS),}$$

where one operation is counted as defined in chapter 2 (TOTAL number) and the values “per frame” for the frames they are calculated for have been extrapolated with a multiplication by 100, because the duration of 100 data frames as defined by 3GPP is 1 second. For TDD the figures, which are normalized per slot, have been additionally multiplied by a factor of 15 to get comparable peak values.

The granularity of the partitioning follows the same partitioning as the complexity figures are given in chapter 2 unless exceptions are named here. The first exception to mention here is that the Descrambler is in the following considered as part of the Rake receiver.

The functional units reaching or exceeding the defined threshold C_{Thw} are listed in the following table.

Functional Unit	Complexity [operations per frame]		
	FDD	TDD	HSDPA
Rx Pulse Shape Filter	124,535,975	123,035,805	124,535,975
Rake Receiver (Combiner+Descambler)	10,776,725	37,417,515	15,423,206
Channel Estimation / MP Searcher	736,021,639	17,825,835	735,019,596
Despread / Softbit Demap	220,077	2,583,840	929,164
Turbo decoder	12,985,888	12,985,888	78,927,840
AFC (Frq. estimation and correction)	1,468,196	225,660	1,468,196
Cell Search (theoretical values)	> 38,000,000	> 38,000,000	N/A

Table 4.1: Functional units exceeding the defined complexity threshold

The components listed in the table above are no good candidates for SW implementation because of their complexity. The FDD Despreader/SB Demap will for symmetry reasons also implemented in HW, or HW from TDD re-used respectively. That is the reason for having this block included though it does not really exceed the defined threshold. The same applies for AFC in TDD mode. Additionally there is an FIR filter and the correlator, which has been found to be better implemented in HW during the classification. Additionally the turbo decoder includes a huge amount of bit-level shift operations, which have also been found to be better implemented in HW.

The blocks listed in Table 4.2 include the remaining blocks, which do a great amount of bit-level operations per processed data frame. The RadioFrameDeEqualisation does not really belong to this group, but is included in this list for HW components to optimise the closeness metrics. With the current partitioning approach this block would otherwise be one SW block between HW implementations. This is not optimal in terms of implementation, because usually a data buffering on block level and a certain synchronization (handshaking, etc.) has to be implemented for each change between the domains.

Functional Unit	Complexity [operations per frame]		
	FDD	TDD	HSDPA
TDD Bit-De-Scrambling	0	39,648	0
RadioFrameDeEqualisation	0	12,678	0
Viterbi decoder (DCCH)	253,777	192,337	0
CRC check	38,712	38,712	233,580

Table 4.2: Functional units with bit-level operations

The overall computational complexity values per second (= 100 times the complexity per frame) of the *remaining* components, i.e. all that are still not planned for HW implementation so far are:

$$\begin{aligned}
 C_{FDDsw} &= 27.6 \text{ MOPS,} \\
 C_{TDDsw} &= 29.6 \text{ MOPS,} \\
 C_{HSDPAsw} &= 222.8 \text{ MOPS.}
 \end{aligned}$$

If the partitioning process would be finished here this would be the amount of operation, which would have to be implemented in SW running on a DSP or other processors. It has to be noticed that the given MOPS value cannot be directly compared to the MIPS figures given for processors at certain clock speed. For the handling of data there are usually a lot more instructions required than for the pure operations, i.e. fetching data from memory or moving it between the registers of the processor's register file. Furthermore the MIPS values are typically only achievable in ideal cases with no branches taken or at least always predicted correctly and no stalls due to data dependencies may appear. So the processor's MIPS value has to be somewhat higher than the MOPS value given here to allow the execution of the mapped code.

On the other hand some processor offer instructions, which can perform more than one single operation (as we define it here) within one instruction. For example the multiply-accumulate MAC operation executes a multiplication and an addition both with arbitrary operands. So in this case the instruction is mightier than one operation as defined here. This would mean that less MIPS are required than MOPS given. However the MAC instruction is special to some processors and cannot be assumed as available.

Even these few conditions mentioned above give an impression how complicated it is to compare the computational expense of an algorithm to the processing power of processors. Thus making the partitioning process hardly deterministic unless no iterations with HS/SW-in-the-loop are applied. Inaccuracies in the figures have to be coped with by considering a guard margin between the required computational power and the performance claimed by the physical component provider.

Now a limit has to be defined for the operations to be performed in SW, i.e. a limit of operations per second has to be given. Though for the demonstrator powerful DSPs can be selected it has to be taken into account that the MIPS figures given by the provider assume ideal program trace estimations (e.g. no branches) and include also all the load and store (or move) commands, which are not counted here. Thus a very big margin has to be accepted between the number of operations to be mapped on the processor (as counted here) and the provided MIPS figures. Thus a limit for SW operation of

$$C_{LIMITsw} = 50.0 \text{ MOPS}$$

will be used.

This means that FDD and TDD mapping already fit to the proposed partitioning, whereas the HSDPA implementation does exceed the limit by a factor of about 4.5 mainly due to the high data rate. Consequently criteria have to be found to map further blocks onto FPGA implementation.

Two issues must be considered here: First the number of changes between HW and SW in the implementation within the data stream flow must be minimized to reduce the buffering and

handshaking overhead, i.e. there should be no isolated HW blocks between two SW implemented blocks and vice versa. A second issue is that functions, which are mainly based on memory access, can better be implemented in SW.

The remaining HSDPA block requiring the most operations is the HARQ component consisting of the RateDeMatcher, Bit Collection, Bit Separation and a soft bit buffer. Though one part is only the memory buffer handling the entire block should be mapped onto FPGA for closeness reasons.

Then after this second step of the partitioning the remaining HSDPA blocks are: Physical Channel De-Map, Rx Constellation Re-Arrangement and the De-Interleaver, which together require a number of operations of

$$C_{HSDPA_{sw2}} = 62.2 \text{ MOPS.}$$

This value is still over the limiting target value $C_{LIMIT_{sw}}$. The memory, which has is used by the Interleaver can function as the means to transfer the data from the blocks implemented in SW and those done in HW. Thus the part of the DeInterleaving called SecDeMux can be easily done in the following HW component, when reading the data from the memory. Then the 235.867 operations per frame counted for the HS-SecDeMux can be subtracted from $C_{HSDPA_{sw2}}$ leading to a new figure of

$$C_{HSDPA_{sw \text{ final}}} = 38.7 \text{ MOPS ,}$$

which does well fit to the limit $C_{LIMIT_{sw}}$ defined for the operations executed in SW per second.

The partitioning figures for the blocks neither mentioned in Table 4.1 nor Table 4.2 are given in Table 4.3 to give an overview of the complete partitioning on block level. Some minor changes to this partitioning will be made during the architecture development in chapter 5. E.g. some blocks will be further split into functions, which will be partitioned differently to achieve a more optimised design implementation.

Functional Unit	Implementation in HW or SW		
	FDD	TDD	HSDPA
PhChannelDeMap	SW	SW	-
Second De-Interleaver	SW	SW	-
Second DTX Remove	SW	SW	-
TrChDeMux	SW	SW	-
First De-Interleaver	SW	SW	-
First DTX Remove	SW	SW	-
RateDeMatching	SW	SW	-
HS-Physical Channel DeMapping	-	-	SW
HS-Rx Constellation Re-Arrangement	-	-	SW
HS-DeInterleaving:DeInterleaver	-	-	SW
HS-DeInterleaving:DeMux	-	-	HW
HS-HARQ (with all subblocks)	-	-	HW

Table 4.3: Mapping of the remaining functional units

4.2.3 Partitioning with Soft-Configurable Technologies

In the previous parts of this chapter the partitioning only takes into account two solutions:

- Dedicated hardware: Structure of logic gates and registers, which is usually described in an HDL and then synthesized to a netlist. This HW may include signals, which slightly influence the behaviour of the hardware. However, in principle the flexibility of the functionality is quite low. The main advantage is that HW is very efficient in terms of power consumption and –if the utilization is quite high– in terms of silicon area.
- Software implementation: Assembler code being executed on a DSP, i.e. only using the resources of the DSP itself and external components like memories and one or more bus systems to connect the components. The DSP itself offers units for arithmetic and logical operations. These units are fixed and usually can execute basic arithmetic operations. Algorithms have to be described in the SW code and must include the full control flow with all decisions, comparisons, and (conditional) changes in the control flow.

In [M_D3.2] different approaches for soft-reconfigurable HW have been proposed. One approach has been selected here to show exemplary how this technique can be used in the system under investigation, i.e. the advantages not just using pure HW and SW solutions but try to combine the advantages of both. Therefore the potential gain of choosing a soft-configurable implementation for certain components of the design will be discussed here. The selected approach has been further developed in the MUMOR project and is introduced in chapter 2.2 of the above-mentioned document.

First the reasons for transferring a component planned for either HW or SW implementation into a soft-configurable component have to be discussed, before making a selection of components. Table 4.4 shows advantages and drawbacks for changing the way of implementing the components into soft-configurable technology, which are originally planned to reside wither in HW or in SW. A general argument against any kind of re-configurability is that it inherently comes along with an overhead in both design cost and implementation cost. Thus it has only a positive effect if it is exploited by the functionality.

	Advantages when done soft-configurable	Drawbacks when done soft-configurable
Originally planned for HW	<ul style="list-style-type: none"> • The HW component of the soft-configurable design (PE) will be directly controlled and configured by the attached processor and thus have increased flexibility compared to normal HW components • Through the increased flexibility the HW component can be used for similar tasks and thus offers potential for better utilization of the silicon area 	<ul style="list-style-type: none"> • Efficiency can only be exploited, when the data processing is done block-based, with a certain minimum block size (This is only true for processor-coupled solutions; the proposed solution belongs to this category) • Beside the HW development, which has to be done if a well tailored processing element is not available, SW has to be created and both tested together
Originally planned for SW	<ul style="list-style-type: none"> • The execution speed can be increased by using better tailored HW for algorithms solution than a DSP normally has • Especially operations on bit-level, which are frequent in the PHY layer of communication systems, can be implemented much faster and more power efficient 	<ul style="list-style-type: none"> • Utilization of accelerator has to be done explicitly by reserved commands in the SW code • Beside the SW development, HW has to be created (done if a well tailored processing element is not available) and both tested together

Table 4.4: Soft-configurable technology vs. HW and SW

One component, which is well suited for an implementation using a soft-configurable technique are the Turbo- and Viterbi-Decoder in a hybrid component. Papers like [Cava03] and [Bick02] have shown that it is feasible to build a reconfigurable components out of these two functionalities due to its functional overlapping. Both require a branch metric unit (BMU) for calculating the hamming

distances for the path metrics and the Add-Compare-Select (ACS) functionality to pick the most probable branch metric, i.e. to retrieve the data that has been transmitted with the highest likelihood.

The advantages of having these two components, Turbo- and Viterbi-Decoder, implemented in one soft-configurable technology can partly be derived from the complexity figures provided in this document. The selected test cases use Turbo decoding for the DTCH and Viterbi for the DCCH, thus both components are used simultaneously. The amount of data to be Viterbi-decoded is relatively small. This could lead to the result to implement a slow Viterbi, with a speed-area trade-off decided in favour of area. However there are other valid configurations of the TrCH coding, which allow also the DTCH to be convolution coded, thus the Viterbi must provide sufficient speed. On the other hand in these cases the turbo decoder is entirely unused (if not by HSDPA channels). As a result, due to the different possible configurations in 3GPP –using basically either Viterbi or Turbo decoding– it appears to be reasonable mapping the sum of their workload to a dedicated implementation. Thus in addition to the similarity of their basic block functionality, plus the block-based data processing nature of these algorithms enable the implementation on a soft-configurable design, like a HW accelerator. The accelerator requires this kind of data operation to minimize the number of disturbances of the controlling processor to increase the efficiency.

The major part of the computations in both Turbo and Viterbi is –beside increment and decrement operations for control– made of additions and subtractions with varying operands as well as bit shifting operations (division and multiplication with operands of kind 2^n) used for BMU and ACS calculations. A processing element including these components as entities or their basic arithmetic with corresponding configuration means in an optimised HW implementation with DSP-autonomous memory to store and access the intermediate results can help increasing the HW utilization.

This soft-configurable solution for Turbo and Viterbi decoders will be further investigated to specify the optimal functionality and degree of re-configurability of a HW architecture for this purpose.

Another class of components which are well suitable for an accelerator structure are those based on bit pipeline, bit shift registers or filter structures with (typically) XOR operations on different stages of the pipeline as shown in Figure 4.4. There are several minor variations possible, e.g. using the filter output directly as data stream or do further operation with this data stream (Output option 1 in the figure). Another option is to use the filter output for further processing on the actual data stream (Output option 2 in the figure). The “further processing” mentioned in the figure could for example be bit interleaving or other bit level operations.

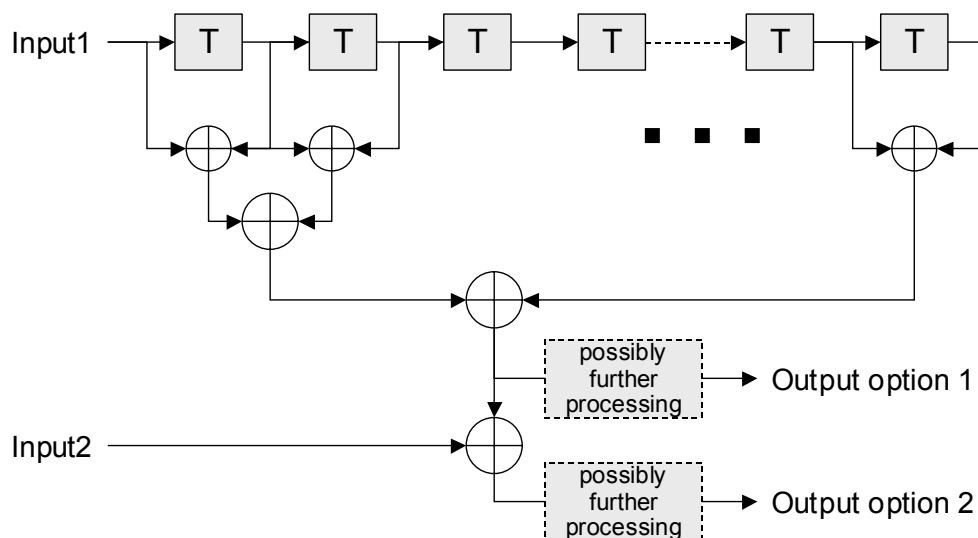


Figure 4.4: Filter structured component with bit-level operations

Such a structure can be for example found in the Recursive Systematic Code (RSC) block, which is part of the Turbo encoder as shown in Figure 4.5. But there are several more blocks in the investigated design on the terminal receiver as well as on the transmitter side, which are based on this principle:

- CRC Encoder (all modes)
- CRC Decoder (all modes)
- Bit Scrambler (TDD mode)
- Convolutional Encoder (FDD and TDD mode)
- Turbo Encoder (all modes)

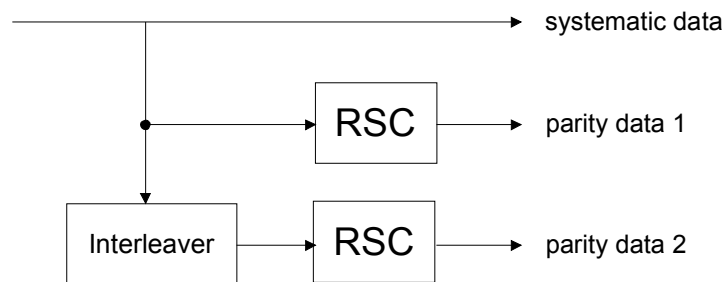


Figure 4.5: Turbo Encoder Structure

It may be even possible to apply this bit level processing for the Bit Descrambler, which is from an algorithmic point of view identical to the Bit Scrambler (Figure 4.7), but this block rather runs on soft bits than on true Boolean bits. Bit Scrambling on the soft bits means that the sign bit has to be altered, when the result of the XORd values of the delay line finally is a “1”, thus the XOR operation on the data stream somehow still is present, but has to be made on the sign bit of the soft bit – unless it is stored in sign-magnitude representation. Nevertheless it will be little more complicated to apply the accelerator processing on bit-level because the mapping of the processed bits to the actual sign bits of the soft bits has to be done. It would have to be checked in more detail taking further aspects, like the number representation and the memory handling into account. In this document we will continue with those cases, which run on bits that are just Boolean values.

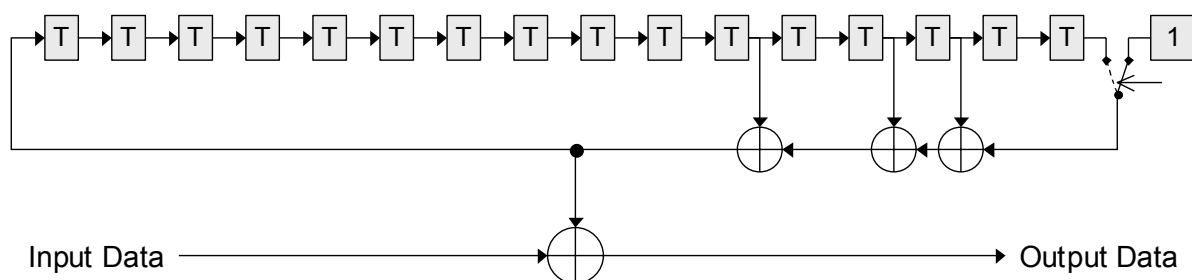


Figure 4.6: Bit Scrambler and Descrambler Structure

Before going into further details of an accelerator implementation of a structure like this – subsequently called “Bit-Arithmetic Unit” – it has to be argued, whether it is really justified to replace the originally intended implementation solution.

It is obvious that a SW implementation has a huge overhead in terms of implemented arithmetics, because ALUs with e.g. 16-bit word length will be used to operate on bit level. In contrast to that it is much more difficult to argue why the implementation should not simply be synthesized in dedicated HW. The few registers and logic gates will not require much area, thus even when the units are not running permanently it would be exaggerated to call this a waste of resources, especially when the components are clock-gated. So there is seemingly not much left that can be eliminated, when an implementation in dedicated HW is proposed, even if several configurations are implemented in

parallel to add flexibility. However there is one aspect left, which provides a benefit to an accelerator structure: The memory access.

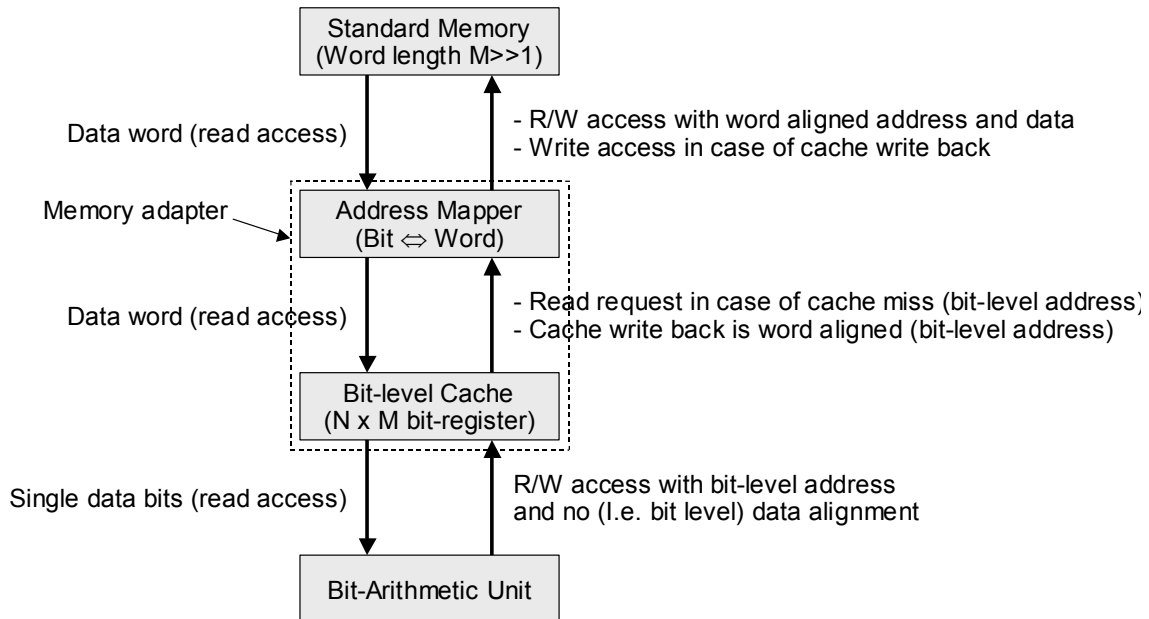


Figure 4.7: Bit-level access for Bit-Arithmetic Units

When each of the bits to be processed is stored in one standard memory cell there is a huge overhead in terms of memory area and a power dissipation due to the frequent memory accesses – one for each bit. The area overhead can partly be reduced by selecting memories with small word length, e.g. 4 is commercially available, but then the memory itself requires an area consumptive address decoding compared to the actual memory cells. What is needed is a “compressing unit” that uses all bits of a memory word by e.g. storing neighbouring bits into one standard memory word. With this approach the required standard memory will be reduced. To also reduce the number of memory accesses a cache is required, that maps the memory words to a bit level memory and includes the typical cache functionality like writing back after modifications and also requires some kind of address decoding. The bit-level cache is being addressed with a bit level address, i.e. the number of a bit. The standard memory access, which the cache has to perform in case of write back (write access) and cache miss (read access) has to be handled by an address mapper, which contains e.g. a look-up table (LUT) to generate the standard memory address out of the bit-level address, which the cache uses. Another solution for the LUT would be an address calculation that also can be applied when the bits are mapped in regular order onto the memory cells. It has to be checked, which solution is more efficient in terms of power consumption and silicon area requirements.

Naturally this bit-level cache with address mapper requires a certain amount of silicon area, which is much bigger than the actual implementation of the actual Bit-Arithmetic unit like the one introduced above. On the other hand it helps to reduce the required memory size and the number of memory accesses dramatically. To maximize this optimisation benefit, which can be achieved by this unit, it should be used by as much Bit-Arithmetic Units as possible. This can be achieved by designing an accelerator processing element including a variable programmable Bit-Arithmetic Unit, which is capable of doing (at least) all the operations required in the UMTS/HSDPA Tx and Rx listed above plus the memory adapter consisting of a bit-level cache and the address mapper.

For the cache optimisation all those strategies, which can be used for normal cache, i.e. those with the same word length as the system memory, can be applied. In this respect prediction strategies for loading new data into the cache early enough to avoid a cache miss (prefetch) are worth mentioning.

5 Implementation Architecture

This chapter describes the implementation architecture, which has been found to be the optimal solution under the given constraints. In chapter 5.1 the main considerations leading to the found solution are presented and chapter 5.2 describes the developed implementation architecture.

5.1 Architecture Considerations

Embedded systems like HW/SW heterogeneous systems, typically have many activities (or tasks) occurring in parallel (concurrent). Concurrent tasks are the natural mode for e.g. many real time applications. Consequently there must be a separation of what each task does from when it does it. Concurrent tasks allow a greater scheduling flexibility since time critical tasks may be given higher priority. However, concurrent multi-tasking introduces complexity into the system, because these tasks have to interact with each other, especially if the different tasks execute asynchronously, i.e. at different speeds. There are three types of interactions between concurrent tasks possible:

- (1) Communication to transfer data between tasks (e.g. a bus)
- (2) Synchronisation to co-ordinate tasks (e.g. interrupts)
- (3) Mutual exclusion to control access to shared resources (e.g. memory)

This task interaction lead to three types of behaviour

- (1) Independent tasks have no interaction with each other
- (2) Co-operating tasks communicate and synchronise to perform some common operation
- (3) Competing tasks communicate and synchronise to obtain access to shared resources.

During the HW/SW partitioning process dedicated resources, which may be an application specific circuit or a microprocessor/DSP, are allocated for a dedicated task. When this allocation is done on a shared resource (e.g. processor, memory or even a shared HW component), the system designer has to take care about what mechanisms are provided to enable the resource to execute one task and to change its activity to another task. Additional topics to be considered, are timing and priority issues to fulfil performance requirements and to avoid deadlock situations respectively.

5.2 Architecture Description

The overall architecture is based on the diagram, which has been introduced in deliverable [M_D3.2] (Figure 5.1).

The aim of this chapter is to propose an in-depth architecture of each block, and according to the previous chapter, to propose a target. So the structure of this chapter will be the same as in [M_D3.2].

5.2.1 Pulse Shaping

We previously say that this module could be included in the simulated channel. But it could be interesting to implement a structure that enables coefficient loading in order to study other filters. In this case, the hardware FPGA target is the best. Previous study has shown that a 10 bit quantification (for coefficients) is enough.

The control structure will read the coefficients from a memory.

Furthermore, the coefficients used for TDD and FDD/HSDPA (even though it is the same “analog” filter) are not the same. The length for FDD pulse shaping filter is 65, and for TDD. The FDD mode over-sampling rate is 4 where as the TDD mode is 8.

Newer FPGA families include dedicated hardware block like multipliers, adders... These blocks are very interesting to use in order to design FIR filter and avoid logic cells consuming.

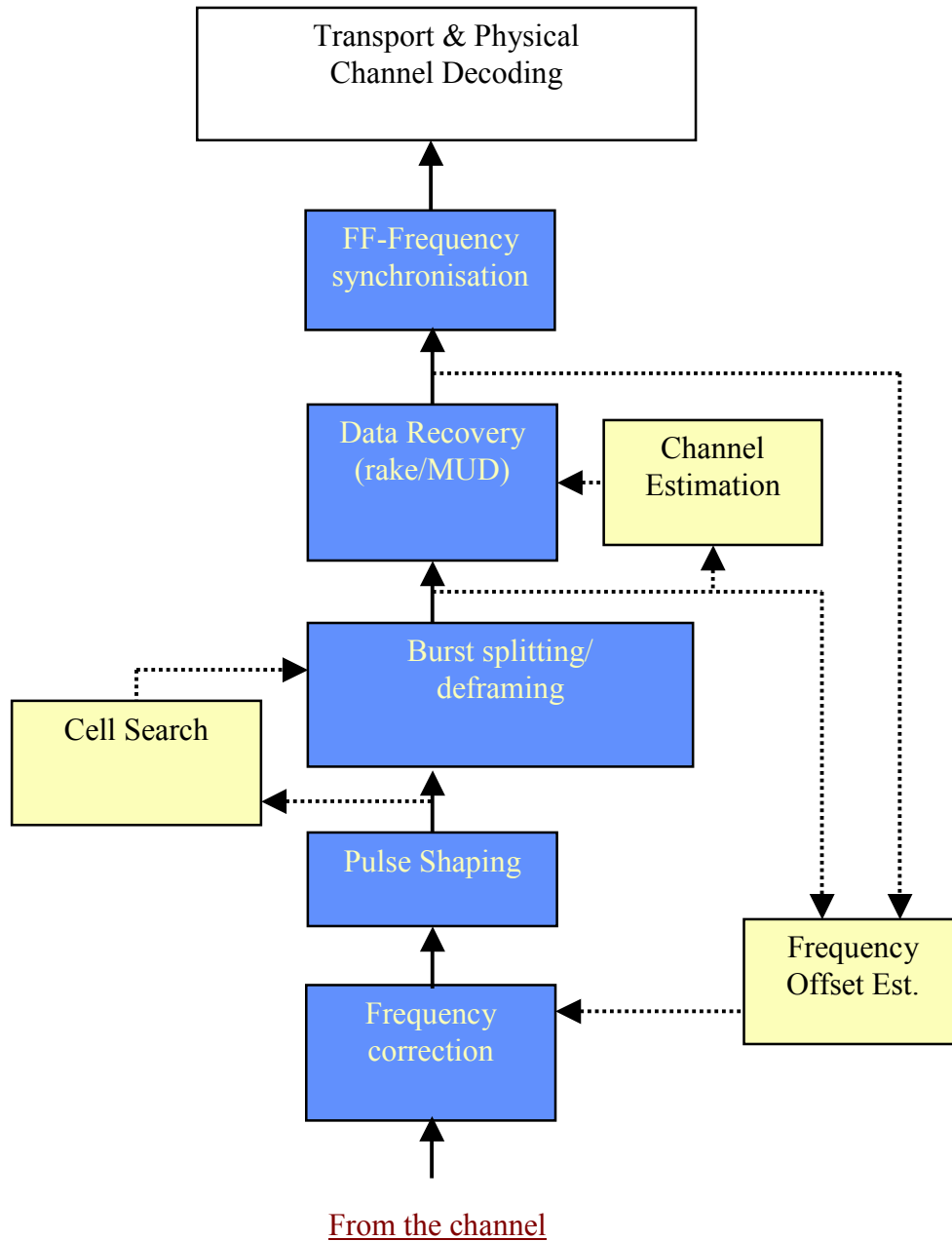


Figure 5.1 : Receiver demonstrator blocks

5.2.2 Channel estimation

For FDD and TDD mode, the channel estimator module mainly uses correlators. The structure of the channel estimator is presented in Figure 5.2. This structure estimates the Amplitude and the Delay of each path:

$H_i(n) = \sum_{[0; P-1]} a_p \delta(n - \tau_p)$, where i is the sampling branch index (in our case $N=8$), a_p the estimated complex amplitude and τ_p the estimated delay.

The result of the power of the correlation is threshold, and enable to recover the amplitude of the correlation peak. The threshold is calculated by a percentage of the sum of the energy of each branch within « max delay length » window.

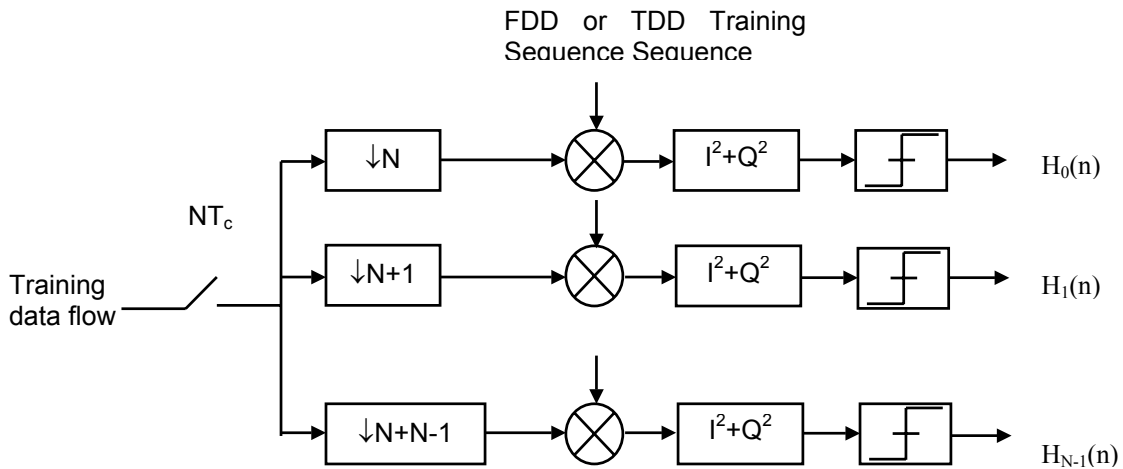


Figure 5.2 : Correlator block structure

The critical part is the correlator. It is based on ± 1 multiplications according to the training sequence. A hardware target is feasible.

In this block, we are supposed to know the beginning of the training data (provided by synchronisation module), so a correlator structure like the one presented in Figure 5.3 could be implemented to despread the training sequence. Obviously, all the synchronisation signals and the shifting of the coefficients have been omitted.

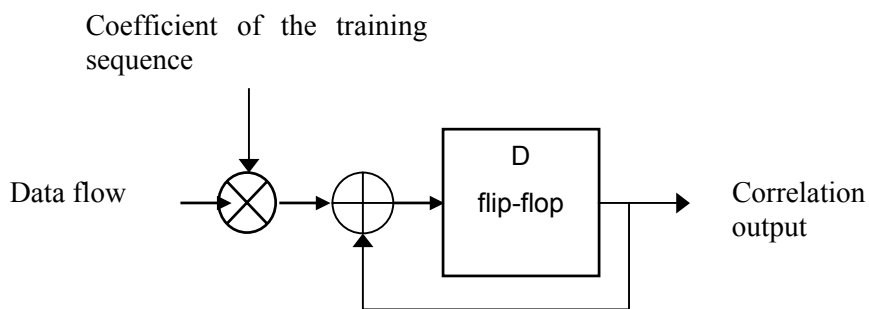


Figure 5.3 : Correlation block (1 sample per chip)

The overall scheme of the channel estimator is proposed Figure 5.4. Each correlator is the one introduced Figure 5.2.

The delay line enables to scan every position within the searching window. It could be embedded in memory or shift register. Obviously, the length of the searching window is different for FDD and TDD mode. But it is very easy to get it programmable without huge hardware overhead.

In this case, the correlators bench will be embedded in hardware contrary to Multipath Searcher (this block recovers the Rake parameters, i.e. amplitude and delay of each path). Actually, the algorithm could be different for FDD/TDD, and includes maximum searcher, a posteriori thresholding processes (several loop over the searching window).

A software implementation will be targeted for Multipath Searcher.

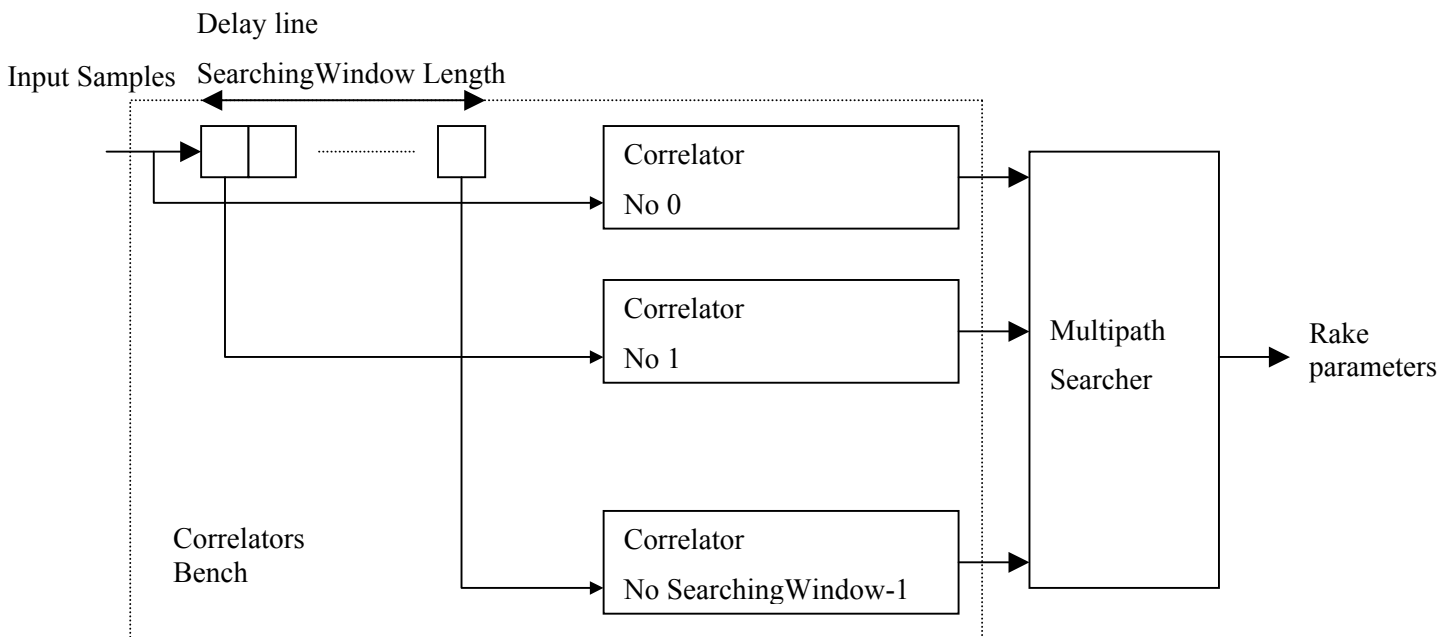


Figure 5.4 : Channel estimator overview

NB: The Figure 5.2 structure could be used, if the despreading of the training sequence is directly processed by a match filter. Due to the correlation length for FDD (2048), the complexity of a match filter would be too high.

So we prefer to introduce the Figure 5.4 structure. We can assume that the complexity decreases: Instead of the correlation length parameter (2048 for FDD), we deal with the searching window size (297 for FDD). Obviously, the MultiPath Searcher algorithm is more complex because we have to wait for the full correlation before processing thresholding.

The hardware complexity calculation is:

- Match filter (for FDD):
 - Correlation length \times sampling rate = 2048×4 registers,
 - Correlation length \times sampling rate = 2048×4 “+/-” multipliers,
 - Correlation length + (Correlation length-1) + ... + 1 = 2096128 adders (match filter adder tree).
- Correlators :
 - Searching Window \times sampling rate = 297×4 registers (delay line),
 - Searching Window \times sampling rate = 297×4 registers (correlators),
 - Searching Window \times sampling rate = 297×4 multipliers,
 - Searching Window \times sampling rate = 297×4 adders.

5.2.3 Rake receiver

Once the channel parameters have been estimated, they can be applied for data recovery. For TDD mode, as the midamble is in the middle of the two data packets, the first packet must be buffered before being processed.

The first part of the algorithm consists of combining the chips according to the estimated path delays, and add them up. The second part is to despread and descramble the data thanks to the knowledge of the sequence. For TDD mode, the despreading is processed for each burst (the burst sequences are different).

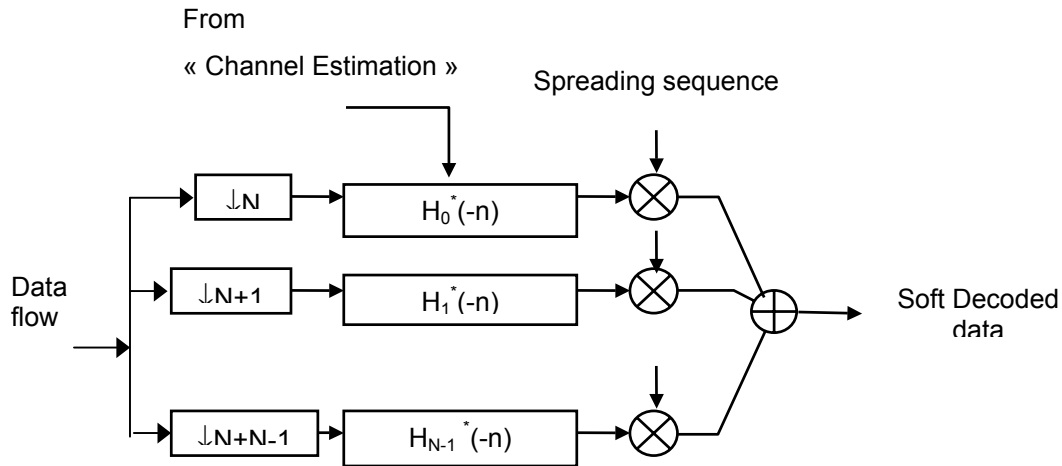


Figure 5.5 : soft data decoding

Like mentioned above, the despreading of the sequence and the peak searcher could be easily embedded in a FPGA. As the multipath has been cancelled, we have only one correlation peak to search, thus we choose the maximum power one.

The combiner will be embedded in DSP. Actually, this structure must contain programmable delay.

If we try to implement this part in FPGA, to get programmable delay, address memory management is needed, and the hardware cost is high. Furthermore, access to RAM may be too slow for the application, and power consuming.

So a fixed point DSP will be a good trade-off between hardware complexity and algorithmic complexity.

5.3 The communication between nodes (The HUNT API)

To demonstrate properly the multimode chain, one platform will be used. The hardware and software modules will be embedded on Hunt board in case of multi-FPGA, multi-DSP, or mixed FPGA/DSP implementation.

5.3.1 HUNT API description

The objective of supplying the HUNT ENGINEERING API is to provide a common interface between software on the host machine and all HUNT ENGINEERING host boards. This interface is also common for the host machine operating systems: Windows 95/98/ME/NT/W2K. HUNT ENGINEERING supports a set of development tools for their products, and users often need to gain efficient access to HUNT ENGINEERING module carriers. The supported way of doing this is the API interface. The API uses a simple asynchronous communications model. First the device must be claimed by performing an HeOpen() on the device. This function takes a board identifier, (to specify which type of HUNT ENGINEERING module carrier is to be opened), a board number, (to specify which of the boards in the system) and a device number (to specify which resource on that board). The function returns a file descriptor if the call is successful, or else an API error code. If the system is to be booted, a reset must be performed using HeReset() on the file descriptor given by the open call. A write to the FIFO can be started using the HeWrite() function on the file descriptor. A read from the

FIFOs can be started using the HeRead() function on the file descriptor. Both the read and write functions will return immediately, with either a successful status, an in-progress status or an error. The in-progress status allows the host side application to continue processing of previous data while the hardware access is taking place. The status of an I/O can be tested at any time using the HeTestforIO() function, or the host program can be blocked until it is complete by using the HeWaitforIO() function. HERON systems are designed to be nodal systems. That is, they are made up of nodes that communicate with each other. The nodes can be processing nodes, I/O nodes or communication nodes. HEART treats all node types in the same way, providing the communications between them. HEART has been designed for boards that have 6 nodes. That is 4 HERON modules plus a node that is the interface to the host computer, plus a node that is used to connect to other boards in the system.

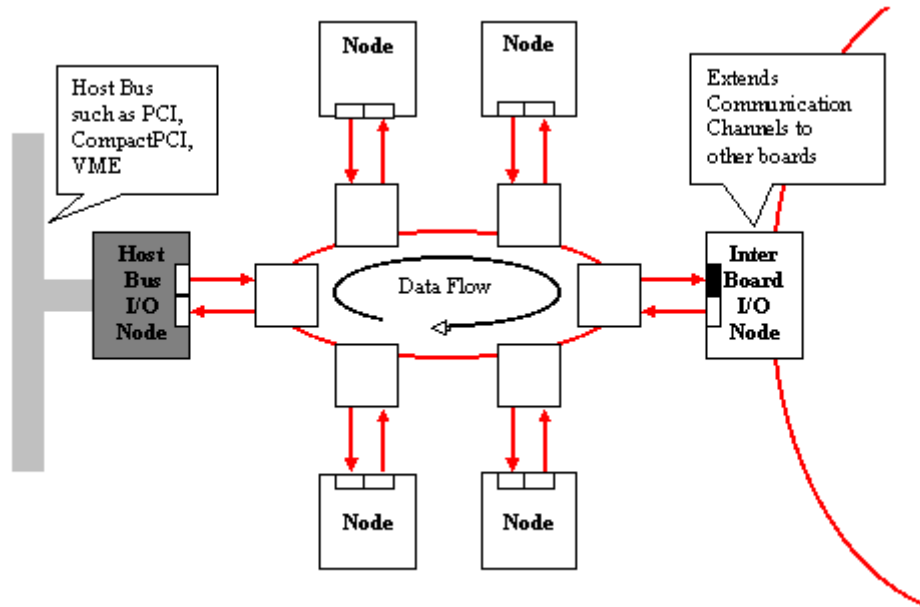


Figure 5.6: The HEART.

HEART is implemented in hardware (designed in FPGAs). Each of the six « node access point » are identical, and connected in a ring.

5.3.1.1 One HEART access point

Each node connected through the HERON interface. That interface defines a 32 bit input path, and a separate 32 bit output path. Each of these paths is like a group of synchronous FIFOs, using a common data path and clock, but separate control signals.

The HEART device on the module carrier board is also split into a read side and a write side. Each provides 6 separate FIFOs that can be accessed at the “far” end by the HERON module.

Data is passed from the read side to the write side, through two registers.

When the six nodes are connected together to form a ring, data is passed from one node to another through a similar pair of registers. As the registers are clocked, the data will pass from node to node, and eventually back to the same register again. This means data can be sent from a node to itself and that data will travel around the whole ring.

The “Ring” is formed out of a 32 bit data path and some control signals. The “Ring” of registers is clocked at 100Mhz meaning that 400Million bytes/second pass between each node.

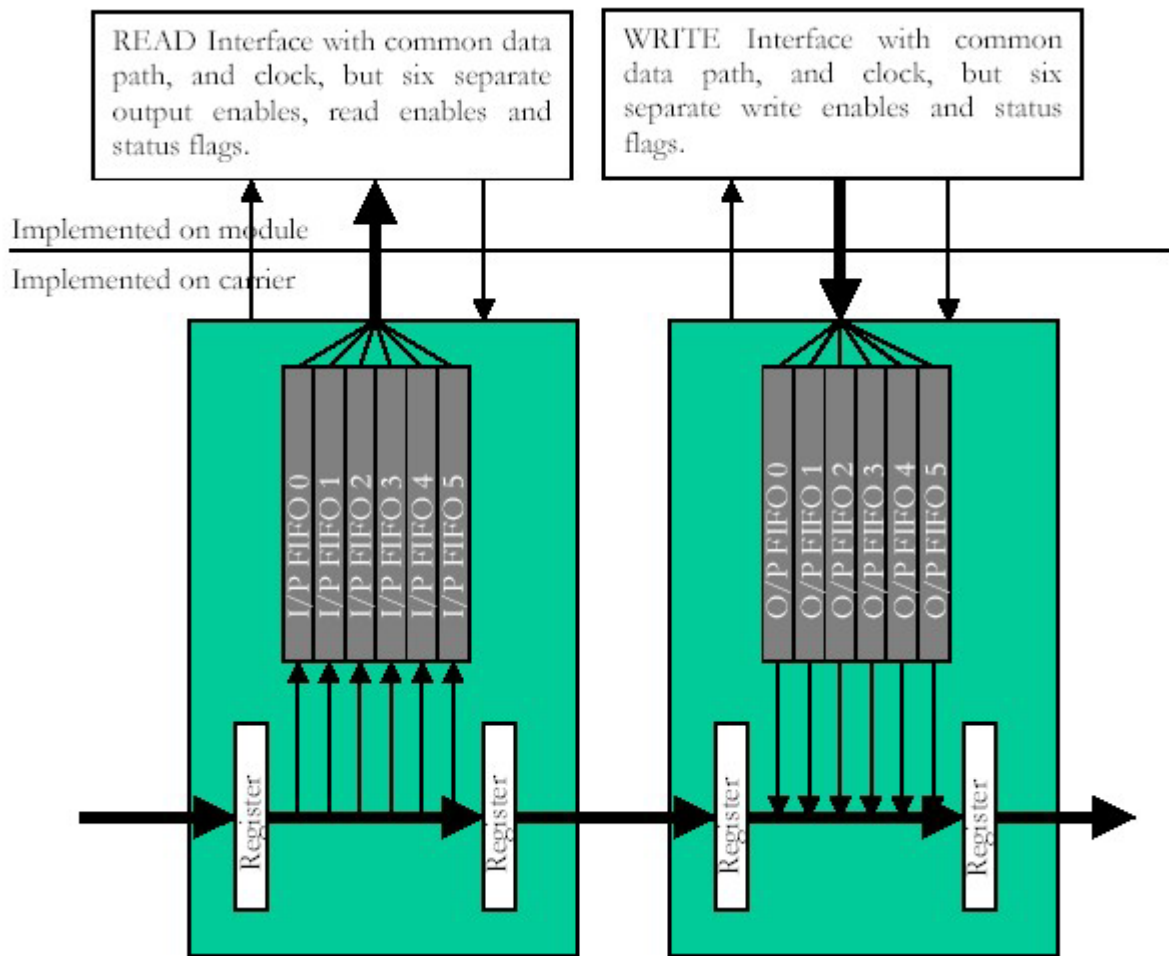


Figure 5.7: The node access point.

5.3.1.2 Time slots

There are six different identifiers and four registers in the ring that contain data that is marked with each identifier. On each clock cycle the data and its identifier moves to the next register position in the ring. As these data items pass around the ring, from register to register, the time slot concept is created. So the 400 Million bytes/sec of total bandwidth is now split into six equal parts of 66.66 Million bytes/sec. Each node in the ring can tell by the control signals on the ring which “time slot” the data in a register belongs to. In fact it can deduce from that which time slot the next piece of data relates to, as the slots are always traveling in numerical order. Hence each node can choose to transmit or receive any time slot. When a data word is placed into a timeslot, another control bit is set to show the data is valid. This allows the time slot to be used for data transfers of less than its full bandwidth. If a transmitter is configured to use a time slot but it does not place data into it, it must invalidate the control signal, to prevent the same data being read again by the receiver. The choice of 6 timeslots is arbitrary, other than the fact it must be an integer division of the number of registers in the ring. The number of time slots does not relate to the number of FIFOs provided by the HERON interface. It can sometimes be easier to consider those time slots as 6 lanes on a highway. Each lane can carry its own traffic only, but traffic cannot switch lanes.

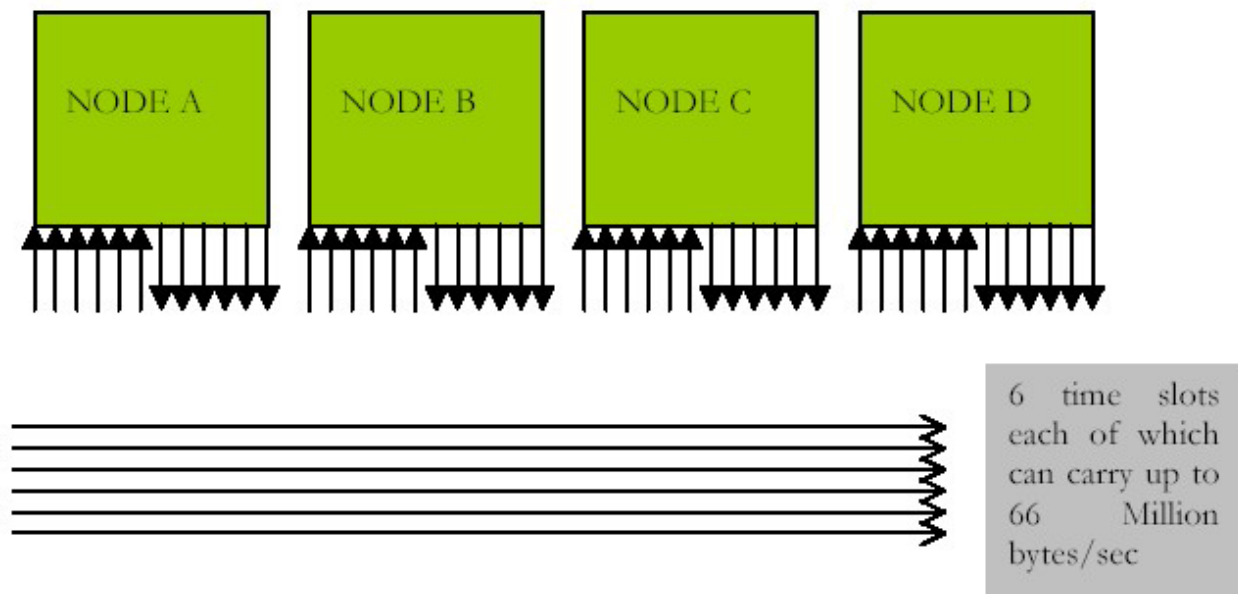


Figure 5.8: Time slot concept.

5.3.1.3 Making a connection

At system reset, each node access point has six FIFOs in each direction, which are not connected to HEART. In fact any connections that were made before the reset are disconnected by the reset. To set up a connection in the system, the node access point of the Transmitting module must “connect” the FIFO that will be used, to a time slot. Now data written into that FIFO by the node will be placed onto the ring, whenever a marker shows that the correct time slot is at this node. If the FIFO is empty when the time slot passes, no data will be placed onto the ring, and the control bit will be set to “invalid”. At this point there is no node set to receive the data, so it will simply travel around the ring until it reaches the transmitting node again. At that point it will be overwritten or invalidated (overwritten with an empty data item). So now a receiver should be configured for the data. The node access point for the receiver needs the FIFO that it will use to be set to receive data from the correct time slot. Now whenever the correct time slot passes the receiving node, the data valid bit will be checked. If the data is valid, the data will be copied into the receiving node’s FIFO. If the data is not valid nothing will be done. Note that the data is not destroyed, or invalidated by the receiving node. Now we have a connection made, where any data written into the transmitting FIFO, will appear in the receiver’s FIFO. Each node can use the flags provided by its local FIFO to control when it can write or read data. To ensure that data is not lost between the FIFOs, another control signal is used in the ring. This signal is used to signal backwards around the ring, when the receiving FIFO is approaching its full state. This signal is used to prevent the sending FIFO from placing any more data items onto the ring. There can still be several data items already on the ring, which will still be received by the receiving FIFO. This is possible because the control signal is *not* the Full flag of the receiving FIFO, but an *almost* full flag. The transmitting node does not pass the blocking signal on around the ring, so it is possible for the receiver to de-assert the signal when it is ready to accept data again.

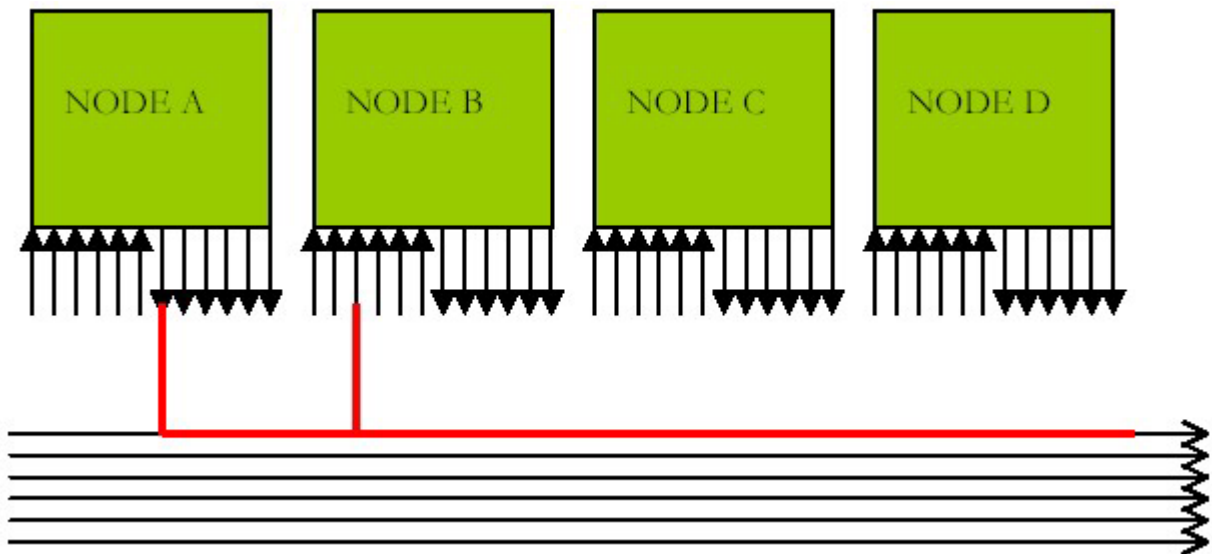


Figure 5.9: Simple connection from the node A toward the node B.

5.3.1.4 Using multiple time slots for a connection

The connection made in the section above uses a single timeslot. That means the maximum number of timeslots passing the node is equivalent to 66.6 Million bytes/sec. The transmitting and receiving nodes will each throttle the transfer to suit themselves, but the transfer cannot exceed the limit of the time slot. If a higher transfer rate is needed, several timeslots can be allocated to the transfer. The same single FIFO can be used at each end of the transfer, but simply allocate multiple time slots for the connection between them. Then the nodes read and write their FIFOs in the same way and the HEART hardware will ensure that the data ordering is preserved. Using multiple time slots for a connection should only be used when the system requires that bandwidth as it consumes system resources, and may make it harder to achieve the topology that you need.

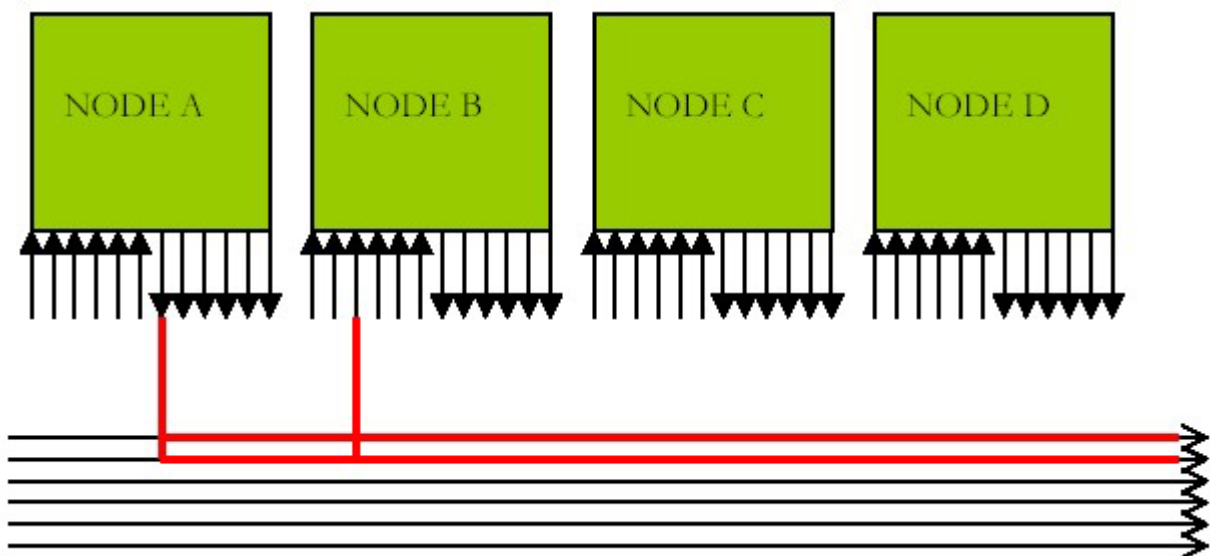


Figure 5.10: Simple connection from the node A toward the node B using two time-slot.

5.3.1.5 Multiple connections between the same nodes

The use of one FIFO and several time slots is the way that you achieve a high bandwidth connection, but there is another way to connect several time slots between the same nodes. That is to use more than one independent connection. In this case you make one virtual connection using one or more time slots, and then make a second connection with its own time slots. In this case the nodes send and receive the data using different FIFO numbers, handling them as different communications. There is no synchronization between the two connections. This can be useful for a number of reasons. For example, the nodes can assign priority to the different streams, and the high priority data can “overtake” the low priority data. Control flow and data flow can separate, sending them over separate virtual connections.

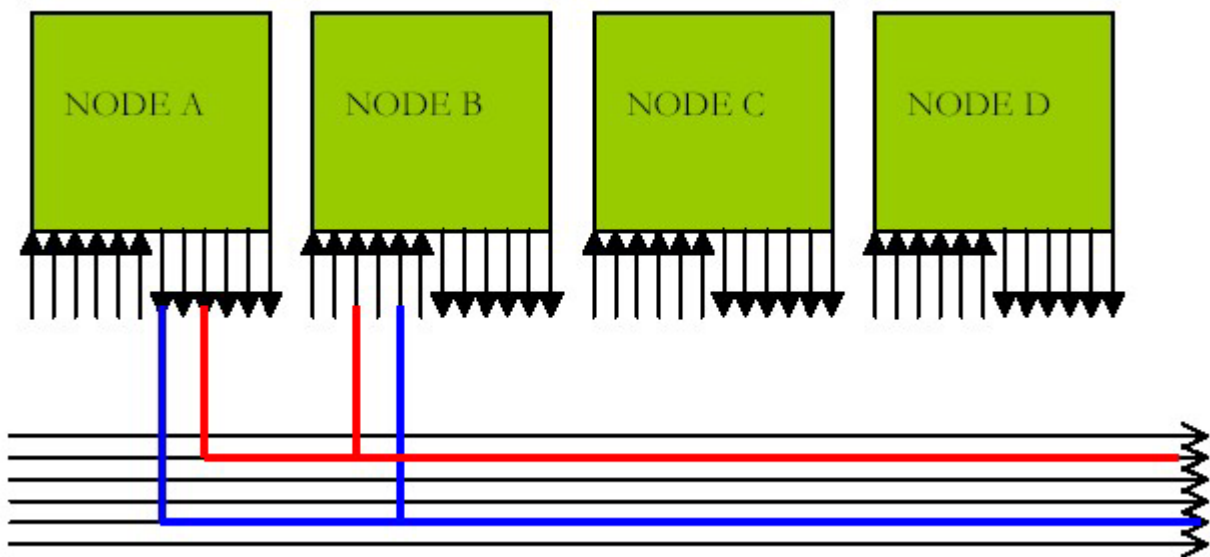


Figure 5.11: Two separate connections from the node A toward the node B.

5.3.1.6 Re-using a time slot

In the sections above we discussed making a connection between two nodes using one or more time slots. We noted that the receiving node did not destroy or invalidate the data as it passed. This means that the data in that time slot will continue around the ring until it is overwritten/invalidated by the original transmitting node. However another node in the system can be configured to transmit into the same time slot. As long as this new transmitter is after the last receiver of the original connection, this is an acceptable use of the system. i.e. as long as two connections do not overlap on the ring, the same time slot can be used by the two connections. In this case each transmitter simply overwrites or invalidates the data from the other transmitter, breaking the timeslot into two separate connections. The fact that each transmitting node does not pass on the blocking signal means that this is also split into separate sections. Thus each virtual connection can “block” its transmitter without affecting the other connections that use the same time slot. The result of allowing re-use of slots in this way is to provide more system resources for making connections. The limit of 400 Million bytes/sec is no longer system wide, but is now applied to each segment of the ring independently.

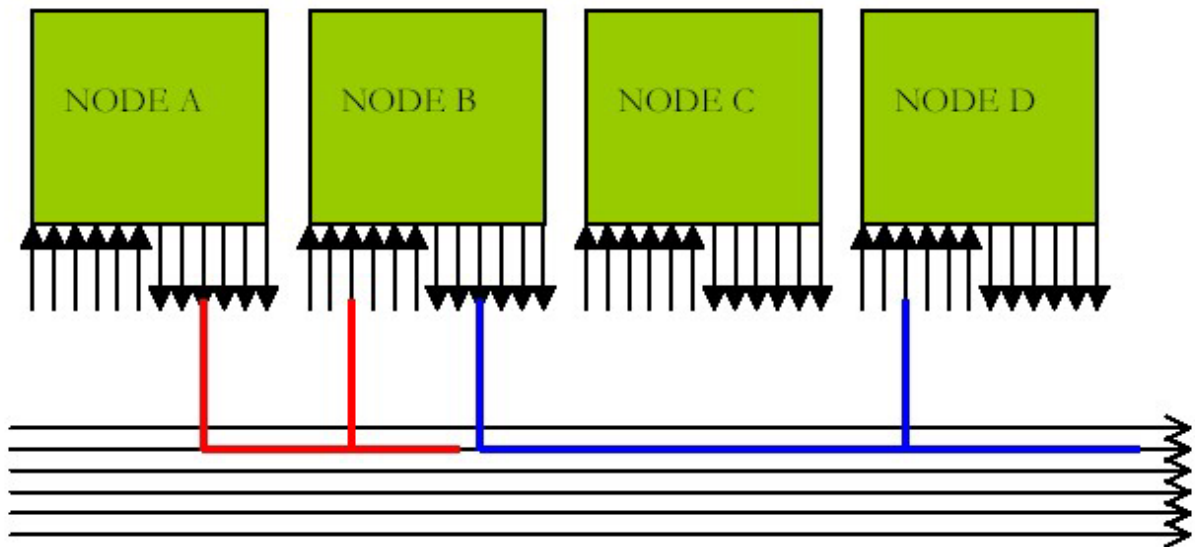


Figure 5.12: Two separate connections from the node A toward the node B, from node B towards node D re-using the same time-slot.

5.3.1.7 Multi-Cast

In the sections above we discussed making a connection between two nodes using one or more time slots and noted that the receiving node did not destroy or invalidate the data as it passed. This means that the data in that time slot will continue around the ring until it is overwritten/invalidated by the original (or another) transmitting node. This allows a node further around the ring to be configured to receive the same time slot, and that node will receive the same data. This is a multi-cast mode, where the same data can be transmitted to multiple receivers. Broadcast is a special case of multi-cast which can be just as easily handled by HEART, where the same data is received by *all* nodes. In a multi-cast connection any node that asserts the blocking signal will cause the transmitter to stop transmitting – effectively blocking the whole connection.

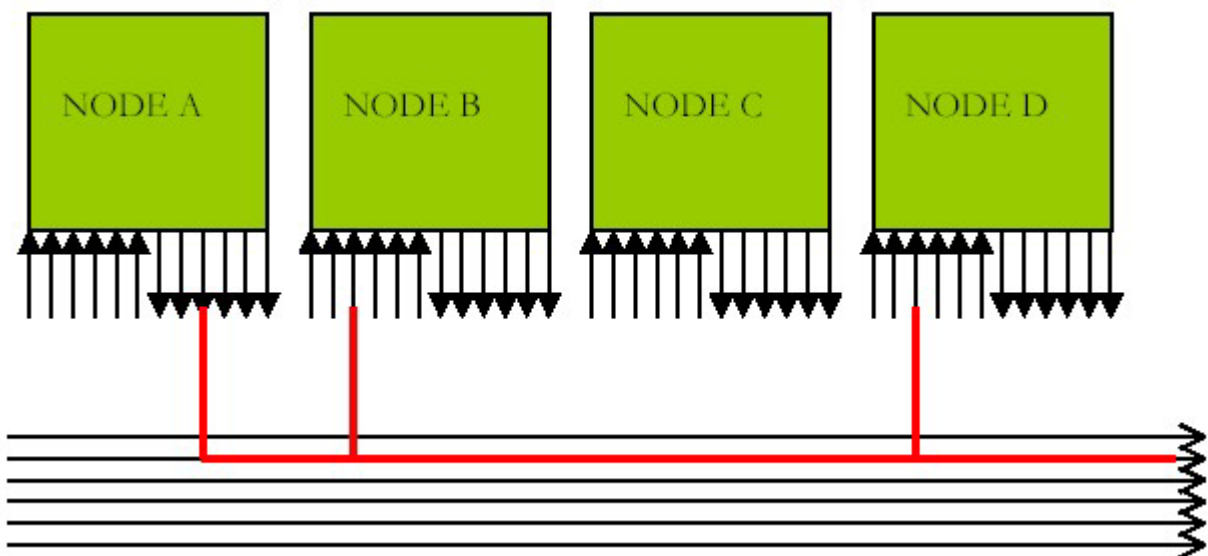


Figure 5.13: Multi-cast connection from node A toward both nodes B and D.

5.3.1.8 Small packets and DSPs

When hardware accesses a HERON FIFO, it will typically respond to the FIFO flag that is at the limit (i.e. the empty or full flags) to determine if data can be transferred. Because of the timing of FIFO flags it can be useful to also use the “almost” flags. Then the following logic can be used. If not “almost” then transfer one data item per clock. If “almost “ is asserted then check the “limit”, and transfer a single word as allowed. Recheck the flag before making each transfer. Hardware can implement this quite efficiently, but a processor cannot. Usually a DMA will be used to make a transfer to or from the FIFO. There is an overhead associated with setting up the DMA, both in terms of processor instructions and hardware pre and post-amble. To minimize these effects it is normal to transfer as large a block of data as possible with each DMA. In fact, there is a trade off in choosing that block size.

If too small buffer is transferred, then there will be lots of small DMAs and the overheads will become significant. Transferring packets that are less than one block cannot use DMA, and must be transferred using less efficient access types.

If too large a block size buffer is transferred, then the size below which transfers are inefficient gets raised. Then small packets will be slowed down.

HEART devices provide a FIFO flag to trigger a DMA block. To provide a good compromise the FLAG is used to indicate that a block of 64 words (32 bits each) can be transferred. However it may not be unusual to transfer a packet that is smaller than this. To help with that problem, HEART devices provide a programmable “almost” flag. This flag is programmable between the limit flag and the block flag, i.e. they can be set to indicate any value between 2 and 63. Then “special” functions on the processor can use that customized “almost” flag to trigger a smaller transfer. This is only useful if a particular connection will always use a small packet size that is constant. It is not intended that these flags be re-programmed between each transfer as that would be just as efficient as transferring the data using the limit flag. The “almost” flags are not conventionally used by processor modules, and may be inaccessible to the processor (such as on the HERON4 module). For this reason HEART devices also allow a UMI pin to be driven by an almost flag. In that case the module can poll the UMI signal to determine if a block of the programmed size can be transferred. These features can be used on C6000 modules by some of the advanced functions in HERON-API.

5.3.1.9 Reconfiguring systems while running

The design of HEART has been made using the assumption that your system connectivity is pre-configured before your application program starts to execute. If this assumption is correct then the system resources will be guaranteed, and the delivery of data by HEART can be relied upon. However there is nothing in the design of HEART that stops you from reconfiguring connections while your system is running. There are several issues that need to be taken care of by your system design if you are to attempt this. This means re-configuring of connections should not be attempted unless it is really necessary and you fully understand the consequences. The fact that there is no arbitration means that reconfiguring a connection will not be “agreed” between nodes in the system. The action of sending the HSB messages to set a new connection will be applied by the hardware immediately. This means that you should be careful about any data that is being transmitted on the connection during the re-configuration. If for example the previous connection is still carrying data when the connection is reconfigured, some data may be lost. The node that was previously receiving the data may hang waiting for the completion of a “packet”, and the receiver of the new connection may receive some data that was not intended for it. If you are to use re-configuration in your system, it is recommended that you stop transmitting data before making the re-configuration. Perhaps HSB messages can be used to achieve this, but it is up to your application program to handle the consequences.

5.3.2 Bit rate performances

In a HERON system using HEART to transport the data between the nodes, there are many things that affect the speed that data can flow in the system.

If we consider a simple transfer, between two nodes using a FIFO connection:



Figure 5.14: FIFO connection.

If the receiver can receive data faster than the transmitter is generating it, then the data transfers are expected without interruption. The Receiver should not read the FIFO when it is empty, so only valid data will be received. This is true as long as the FIFO can support that data rate. With HEART (as on the HEPC9) the FIFO connection is a “Virtual FIFO”. HEART makes this connection using 2 real FIFOs (implemented in FPGAs), connected together using a time-slotted ring. The FIFOs are 32 bits wide, and use a 100Mhz clock, so the module can transfer data to and from the FIFO at a peak rate of 400Mbytes/sec (100Mwords/sec). The timeslots of HEART are 66 Mbytes/sec.

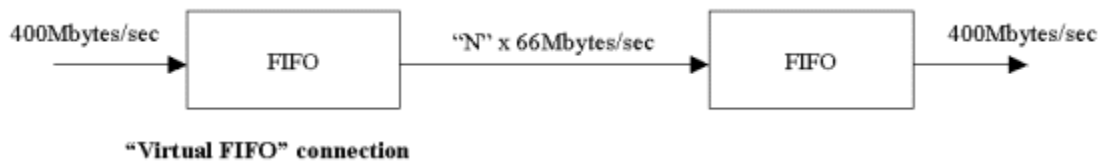


Figure 5.15: HERON FPGA4 diagram.

So the speed that the “Virtual FIFO” connection can support is limited by the number of timeslots selected for it.

If the transmitter generates data faster than the receiver can accepted, the transmitter uses the full flag on the FIFO. In most cases, the transmitter can simply wait for the FIFO to be ready, so the transfer of data is limited by the slowest element.

5.3.2.1 FPGA based modules

In the case of an FPGA based module, the input FIFO connection and the output FIFO connection are connected directly to the FPGA. This means the FPGA has direct control over the transfer of data, and can sustain the maximum transfer rate in and out at the same time, if the FPGA design allows that. Of course, the rates of the transfer data depends on the frequency of the FIFO clock that is generated from the FPGA design. Normally this will be the full 100Mhz, but there can be cases where the FIFO design is simpler if a lower clock rate is chosen. If this is the case, the data rates is reduced accordingly. Normally the HERON FIFOs will be accessed using the hardware interface layer (HIL) VHDL that is supplied by HUNT ENGINEERING.

When accessing a single FIFO (in or out) the HIL allows the full 400Mbytes/sec to be sustained into and out of the FPGA (at the same time).

When several FIFOs are transferring data at the same time, the HIL needs to switch accesses from one FIFO to another. This switching causes a single cycle where data cannot be transferred, reducing the total data rate that is possible. The HIL supports different models of accessing several FIFOs.

The first is to permanently request data from the FIFOs, then the HIL will use a “round robin” method to access the FIFOs. This can cause data to be transferred one word at a time, with one “dead” cycle between each word. This makes the total rate become 200Mbytes/sec.

It is possible to achieve a greater rate by forcing the access before the “dead” cycle. Using this method total rates of above 350Mbytes/sec can easily be achieved.

Peak rate	100Mwords/sec = 400Mbytes/sec in + 400Mbytes/sec out
Sustained rate single FIFO in one direction	400Mbytes/sec
Sustained total rate on several FIFOS “normal method”	200Mbytes/sec
Sustained total rate on several FIFOS “cycling method”	>350Mbytes/sec

Table 5.1: Peak rate.

Rates achieved with an FPGA based module on a HEPC9 (FIFO clock=100Mhz).

5.3.2.2 C6000 based modules

When using a C6000 processor in a real time system, IN/OUT data transfer are needed. The C6000 DSPs have DMA engines that allow the I/O to be handled separately from the processing. Of course there are interactions between processing and I/O. The DMA and processor share internal and external busses, and the DMAs need to be programmed by the processor. The HERON4 clocks the HERON FIFOS using the processor clock halved. This means that a C6701 module produces a 83.5Mhz FIFO clock, and the peak transfer is 334Mbytes/sec. The C6201 version has a peak rate of 400Mbytes/sec because the processor clock is 200Mhz. The HERON2 uses the XBUS of the C6203 to access the FIFOs using a 75Mhz clock. Using this separate bus allows FIFO data to be read and memory to be written in the same clock, thus reducing the conflict of resources.

5.3.2.2.1 HERON-API and FIFO accesses

It is usual to use HERON-API to control the DMAs and interrupts in order to correctly access the HERON FIFOs. This library is linked into your DSP program allowing that program to use simple read write calls to transfer the data. HERON-API has a complex job to do. It must share the four DMA engines of the processor around a possible 12 different data streams. To achieve this it maintains a list of DMA engines that do not have transfers in progress, and another list of transfers ready to be made.

5.3.2.2.2 Floating DMA

The default use of these lists is that the tasks get allocated resources in the order that they became ready, a mode that we talk about as "floating DMAs". In this mode of operation, each FIFO block is queued and allocated resources separately, so during a large transfer the DMAs will be claimed and freed inside the HERON-API many times. HERON-API uses interrupts triggered by the FIFO flags and by DMA completions to manage the queues. Using two HERON4-C6701 modules on an HEPC9 gives us the following results

HERON4-C6701 to HERON4-C6701 one timeslot to external memory	51Mbytes/sec
HERON2-C6203 to HERON2-C6203 one timeslot to external memory	66Mbytes/sec
HERON4-C6701 to HERON4-C6701 two timeslots to external memory	51Mbytes/sec
HERON2-C6203 to HERON2-C6203 two timeslots to external memory	132Mbytes/sec
HERON4-C6701 to HERON4-C6701 two timeslots to internal memory	113Mbytes/sec
HERON4-C6701 to HERON4-C6701 more than 2 timeslots to internal memory	113Mbytes/sec
HERON2-C6203 to HERON2-C6203 more than 2 timeslots to internal memory	132Mbytes/sec

Table 5.2: Rate using float DMA.

So the HERON4 achieves it's maximum if 51Mbytes/sec with 1 or more timeslots when using external memory. Using internal memory increases this limit to 113Mbytes/sec. The HERON2 however can achieve 132Mbytes/sec whether it is using internal or external memory.

5.3.2.2.3 Dedicated DMA

Because the floating DMA technique could cause a transfer to be delayed while there is no DMA resource available, HERON-API has an option to "dedicate" a DMA engine to a particular transfer. Of course using this option for all transfers limits you to a maximum of 4 transfers. Less if the program wishes to claim a DMA engine. Actually this will speed up a HERON4 transfer regardless of how many transfers are happening, because the HERON4 can make the entire transfer using hardware, and only raise one processor interrupt at the end. This is not possible for a non dedicated transfer as the DMA would be returned to the pool of available resources, and re-claimed when this transfer is next on the list.

HERON4-C6701 to HERON4-C6701 one timeslot to external memory	66Mbytes/sec
HERON2-C6203 to HERON2-C6203 one timeslot to external memory	66Mbytes/sec
HERON4-C6701 to HERON4-C6701 two timeslots to external memory	66Mbytes/sec
HERON2-C6203 to HERON2-C6203 two timeslots to external memory	132Mbytes/sec
HERON2-C6203 to HERON2-C6203 four or more timeslots to external memory	210Mbytes/sec
HERON4-C6701 to HERON4-C6701 two timeslots to internal memory	132Mbytes/sec
HERON4-C6701 to HERON4-C6701 three timeslots to internal memory	199Mbytes/sec
HERON4-C6701 to HERON4-C6701 4 or more timeslots to internal memory	232Mbytes/sec
HERON2-C6203 to HERON2-C6203 4 or more timeslots to internal memory	213Mbytes/sec

Table 5.3: Rate using dedicated DMA.

The HERON4 speed improves dramatically using dedicated DMA, but only when you are transferring to internal memory.

The HERON2 is actually slower that the HERON4 when using lots of timeslots and internal memory but achieves almost the same rate to external memory as it does to internal memory.

5.3.2.2.4 Multiple “Blocking” transfers

When the DSP is transferring to or from several FIFOs there will be some interaction between the two flows because the resources of the DSP are being shared between the transfers. Measuring the speeds with one input stream and one output stream gives

HERON4-C6701 1 timeslot read and write non-dedicated DMAs int mem	94Mbytes/sec total (in and out)
HERON2-C6203 1 timeslot read and write non-dedicated DMAs int mem	126Mbytes/sec total (in and out)
HERON4-C6701 1 timeslot read and write dedicated DMAs int mem	127Mbytes/sec total (in and out)
HERON2-C6203 1 timeslot read and write dedicated DMAs int mem	127Mbytes/sec total (in and out)
HERON4-C6701 2 or more timeslot read and write non-dedicated DMAs int mem	121 Mbytes/sec total (in and out)
HERON2-C6203 2 or more timeslot read and write non-dedicated DMAs int mem	178 Mbytes/sec total (in and out)
HERON4-C6701 2 or more timeslot read and write dedicated DMAs int mem	200Mbytes/sec total (in and out)
HERON2-C6203 2 or more timeslot read and write dedicated DMAs int mem	236Mbytes/sec total (in and out)

Table 5.4: Rate using input and output FIFO.

So with the HERON4 each of the transfers is reduced a little but only by around 10% each. The HERON2 has a bigger loss in total because the memory bus is now switching from read to write. It also seems that you can find a limit in the total I/O at around 200Mbytes/sec.

5.3.2.2.5 Single Sample processing (Control loops)

If the samples are received one per one, the application will "poll" for data, perform processing and output as soon as it arrives, and expect to be ready to poll for the next data item before it arrives. In this case, the functions HeronReadWord and HeronWriteWord can be used. These functions are compiled "in-line" for maximum efficiency and will not return until the data is transferred. The number of cycles to transfer a single word using a HERON4 is 52. This means that the processor clock rate is important. A C6701 module runs at 167Mhz for example takes 312 ns to read and another 312 ns to write any output. This makes a loop with no processing take a minimum of 624 ns which is 1.6 Mhz. The HERON2 is forced to use DMA for all transfers. This means it will be inefficient and will have some uncertainty of response when accessing only single words. As the number of samples is increased the overhead of the DMA becomes less significant.

5.3.2.3 PCI HOST bandwidth

Like the C6000 modules, the HEPC9 has been designed so that the peak transfer rate is the maximum allowed by the PCI bus: 132 Mbytes/sec.

One thing that can affect the sustained rates possible is the operating system running on the host PC. The main difference is between OS models that allow the driver to know the physical address of the memory buffer the user has allocated. If the physical address can be obtained, then the Master Mode can be used to transfer the data directly using the buffer allocated by the user program. Win NT uses this model. If however a physical memory address cannot be obtained, the master mode must first transfer the data into a “driver buffer” and then copy it into the user buffer. The copy process takes extra time, reducing the sustained data rate. Win 98/ME uses this model.

The following performances have been measured on several PCs with the following results:

PC type	OS	1 slot read	1 slot write	1 slot Read + 1 slot write total	2 slot read	2 slot write
AMD K6-450	Win98	35	36	50	47	37
AMD Athlon 850	Win98	51	57	60	65	57
PIII-800	Win98	41	45	43	59	46
Dell PIII-800	Win NT	64	65	92	101	65
P-Pro 200	Win98	31	23	28	40	23
P4 1.5G	Win98	56	35	50	62	35
P4 1.5G	Win NT	64	36	56	70	37

Table 5.5: PCI host bandwidth.

There are obviously variations between different machines. Generally, more the machine is faster, more the transfer is faster. Win NT is generally faster than 98. On the P4 1.5Ghz the same machine was tested using NT and 98 for a comparison where the OS is the only difference. The difference is about 40%. This machine shows a big difference between reading and writing speeds. Looking at the PCI signals it seems that this chipset handles reads and writes in different ways. The different chipsets in PCs seems to have a big effect. In general the speed when reading and writing is about 1.5 times the speed of one direction. Measuring speeds with more than 2 HEART timeslots connected does not increase the bandwidths achieved.

5.3.3 Exemple of API using

5.3.3.1 Generalities

The partners involved in the design of the blocks which will be mapped on demonstration board must take care of the connection between module to build the data flow. Futhermore, the added code need to be as light as possible to enable the reuse on an other platform. For that purpose, as introduce before, HUNT developed API (access peripheral interface) for read and write operation.

The partners who develop a module have to know the frontier between algorithm and implementation.

So the goal of this chapter is to explain how to work.

The access to the Heron network is asynchronous. It means that the **read** and **write** operations are asynchronous.

The two functions are **HeRead()** and **HeWrite()**. The API provides the designer with status function **HeTestIo()** and **HeWaitForIo()** to allow to track the progress of the transfer. In fact, the read or write transfer is processing in parallel with the application thanks to the DMA (Direct Access Memory). The **HeRead()** or **HeWrite()** function only requests the parallel thread to perform a read or write transfert. The status functions only check whether the parallel thread has already completed or not.

- **HeTestIo()** function just asks the parallel thread whether is completed. It immediately returns.
- **HeWaitForIo()** function waits for the transfer to complete.

5.3.3.2 Writing operation

The status values for the writing of data are after calling **HeTestIo()** or **HeWrite()**:

- **HE_OK** if the transfer is completed,

- **HE_IoInProgress** if the transfer is still on going
- Error value.

Example using **HeTestIo()** : we have to scan the status variable to determine wheter the transfer is completed or not.

```
Status = HeWrite(hDevice, WriteBuffer, size, WriteIoStatus);
While (status == He_IoInProgress)
{
    custom_instruction(list of variables and parameters)
    status = HeTestIo(hDevice, WriteIoStatus);
}
if (Status != HE_OK) Error_Report();
```

where *hDevice* is the address of the targeted writing module, *WriteBuffer* the array to write, *size* the length of the array.

The status values for the writing of data are after calling **HeWaitForIo()** :

- **HE_OK** if the transfer is completed,
- Error value.

Example using **HeWaitForIo()** : we don't need to scan to scan the status variable to determine wheter the transfer is completed or not.

```
Status = HeWrite(hDevice, WriteBuffer, size, WriteIoStatus);
custom_instruction(list of variables and parameters)
if (status == He_IoInProgress)
    status = HeWaitFotTestIo(hDevice, WriteIoStatus);
if (Status != HE_OK) Error_Report();
```

5.3.3.3 Reading operation

It follows the same philosophy as the writing.

The status values for the reading of data are after calling **HeTestIo()** or **HeWrite()**:

- **HE_OK** if the transfer is completed,
- **HE_IoInProgress** if the transfer is still on going
- Error value.

Example using **HeTestIo()** : we have to scan the status variable to determine wheter the transfer is completed or not.

```
Status = HeRead(hDevice, ReadBuffer, size, ReadIoStatus);
While (status == He_IoInProgress)
{
    custom_instruction(list of variables and parameters)
    status = HeTestIo(hDevice, ReadIoStatus);
}
if (Status != HE_OK) Error_Report();
```

where *hDevice* is the address of the targeted reading module, *ReadBuffer* contains the read array, *size* the length of the array.

The status values for the writing of data are after calling **HeWaitForIo()** :

- **HE_OK** if the transfer is completed,
- Error value.

Example using **HeWaitForIo()** : we don't need to scan to scan the status variable to determine whether the transfer is completed or not.

```
Status = HeRead(hDevice, ReadBuffer, size, ReadIoStatus);  
custom_instruction(list of variables and parameters)  
if (status == He_IoInProgress)  
    status = HeWaitFotTestIo(hDevice, ReadIoStatus);  
if (Status != HE_OK) Error_Report();
```

6 Summary and Conclusion

In the present deliverable complexity figures for the downlink receiver modules of the UMTS FDD, TDD and HSDPA (FDD) system have been presented. The figures are based on different test case, which are defined in the 3GPP specifications. The complexity here is counted as the number of arithmetic operations required fulfilling the system functionality. To enable a distinction of the operations into different complexities, for each block not just a sum of operations, but a detailed list divided in to different operations and also into different kinds of operands has been done.

In a first step this has been done for the single-mode downlink chains. These results are used later to discuss the gain that can or has been achieved during the multi-mode partitioning. In this partitioning basically two approaches have been applied on algorithm level. The inherent difficulty that these optimisations on algorithm level have is that most of the implementation specific factors are unknown, which makes it complicated to make final decisions in terms of optimisation. Nevertheless two approaches for design optimisation have been applied:

- Find algorithms, which are similar in their implementation and for all modes and
- Exploit the re-use of arithmetic components and memory cells across the modes.

The applied optimisation has been described for the components of the receiver and complexity figures have been provided for those components, which have been modified in their functionality. For those components, which can be shared across the modes the complexity itself stays the same as only the implementation will look different. However this change is not directly visible on algorithm level. Beside the downlink Rx complexity figures also the complexity figures for uplink reference channels (test cases) have been provided for the transmitter to allow extending the implementation optimisation also to resource sharing between terminal transmitter and receiver.

For this reason in a separate chapter possible implementation scenarios have been presented. After a comprehensive presentation of the general partitioning principles and the specific practices that have been applied during this work, a suitable partitioning of the system onto HW and SW components has been discussed. During the quantification of the optimisation gain this partitioning partly had to be anticipated, because the effect of SW sharing and HW sharing on the implementation are quite different. To preserve a maximum flexibility of the design a software-oriented partitioning approach has been selected. However it was quite obvious that especially the components operating on chip level are computationally expensive, and not suitable for SW implementation.

In addition to the pure HW and SW mapping it has also been considered, what can be even more improved, if the implementation also uses soft-configurable techniques. Two classes of functions have been found for which an extensive usage of soft-configurable technologies, especially a HW accelerator, would be efficient. In some cases for example adding slight functional flexibility to them can increase the utilization of HW components. This approach helps to reduce the silicon area. In a second step, when certain HW is utilized quite well, it may be reasonable to add a hardware overhead to optimise in terms of other metrics, e.g. power consumption. In this document the basic functionality of a memory adapter have been presented, which allow an efficient usage of memory cells and also reduce the number of power consuming memory accesses. But this unit can only be implemented, when the overhead it produces in terms of area, implementation and verification time and is in a suitable relation to its usability and utilization.

Finally the implementation architecture for critical components has been presented as part of the overall hardware implementation. These architectures will later be implemented for the MUMOR baseband demonstrator.

Next steps will be detailed performance comparison of the multi-mode optimised design with the single-mode reference design. Then transition to fixed-point implementation will be done to create block specification for the Rx components suitable for block implementation in either HW or SW.

References

- [3GPP25.101] 3GPP Specification, TS25.101, “User Equipment (UE) radio transmission and reception (FDD) (Release 5)”, Version 5.7.0
- [3GPP25.102] 3GPP Specification, TS25.101, “UE Radio Transmission and Reception (TDD) (Release 5)”, Version 5.1.0
- [Bick02] M. Bickerstaff, D. Garrett et al., “A unified Turbo/Viterbi Channel Decoder for 3GPP Mobile Wireless in 0.18 μ _m CMOS,”, ISSCC 2002.
- [Cava03] J. R. Cavallaro, M. Vaya: "VITURBO: A reconfigurable architecture for Viterbi and Turbo decoding", 2003 IEEE International Conference on Acoustics, Speech and Signal Processing (ICASSP), April 2003, Hong Kong
- [Gajs94] Daniel D. Gajski, Frank Vahid, Sanjiv Narayan, and Jie Gong: “Specification and Design of Embedded Systems”. Prentice Hall, 1994
- [Kuch02] Krzysztof Kuchcinski: Lecture “Design of embedded Systems”, Lund Institute of Technology, Dept. of Computer Science, 22.04.2002
http://www.cs.lth.se/home/Krzysztof_Kuchcinski/DES/Lectures/Lecture7.pdf
- [M_D3.2] MUMOR D3.2 “Methodology Evaluation Report”,
 Doc Number: IST-2001-34561/NOKIA/WP3/R/RE/006
- [RASSP] RASSP (Rapid Prototyping of Application Specific Signal Processors) Course
 Module 14 “Hardware/Software Codesign Overview”,
<http://www.eda.org/rassp/modules/abstracts.html>