NTNU
Innovation and Creativity

A programmable DSP for low-power, low-complexity baseband processing

Hallvard Næss

Master of Science in Electronics
Submission date: June 2006
Supervisor: Kjetil Svarstad, IET
Co-supervisor: Robin Hoel, Chipcon
Per Torstein Røine, Chipcon

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Problem Description

The concept of Software-Defined Radio (SDR) holds tremendous promise. The basic idea is to get code as close to the antenna as possible, in order to achieve greater flexibility, improve adaptability, and decrease development time compared to implementing dedicated, fixed-function hardware modules for digital radio. The concept is not new, but the development of RF-compatible deep submicron process technology has reached the point where SDR can be implemented costcompetitively and sufficiently high-performance compared to fixed-function digital radio hardware.

In a typical digital single-chip radio transceiver, digital demodulation and modulation is performed in dedicated hardware, MAC and link functionality in a combination of software and hardware, and network and application in software/firmware. By using a high-performance DSP core to implement the demodulation/modulation of the signal, it should be possible to support multiple radio physical layers, and possibly save area compared to a dedicated hardware implementation due to reuse of hardware resources. Furthermore, higher performance may be achievable by fine-tuning the demodulator/modulator algorithms on the actual silicon device or by implementing channel coding or equalization techniques that are too costly or complex for dedicated hardware.

The development of a processor architecture especially suited for implementation of simple to medium complexity radio physical layers will be the task for two student master thesis projects. One student will concentrate on how one or more radio standards can be implemented as SDR on such a processor, while the other student will focus on the actual DSP architecture (this thesis).

The focus for this thesis will be:

- Searching published literature for related baseband processors and concepts relevant for low-power DSP architectures.
- Propose an area-efficient DSP-architecture that within certain documented constraints supports the required primitive operations, and that can implement one or more of the physical layers studied.
- Estimate computational complexity and power consumption for the proposed architecture implementing a specific physical layer.
- Discuss system partitioning for an SDR: some DSP-operations may be more suited for dedicated hardware and some control and some low-bandwidth control loops or data decoding better suited for a general-purpose CPU.

Assignment given: 16. January 2006
Supervisor: Kjetil Svarstad, IET

Abstract

Software defined radio (SDR) is an emerging trend of radio technology. The idea is basically to move software as close to the antenna of a radio system as possible, to improve flexibility, adaptability and time-to-market.

This thesis covers the description of a DSP architecture especially optimized for modulation / demodulation algorithms of low-complexity, low-power radio standards. The DSP allows software processing of these algorithms, making SDR possible. To make the DSP competitive to traditional ASIC modems, tough constraints are given for area and power consumption. Estimates done to indicate the power consumption, area and computational power of the DSP, shows that a software implementation of the studied physical layer should be possible within the given constraints.

Preface

This paper is a Master's thesis in electronics, based on the finishing work of a Master's degree during the spring 2006. It is written by a student at the Department of Electronics and Telecommunication at the Norwegian University of Science and Technology (NTNU). The task of this project has been given by Chipcon, as a study of the feasibility of implementing their future radio transceivers as *Software Defined Radio* (SDR). This work has partially been done in cooperation with one other student, resulting in two different theses as the work has been partitioned between the two students. However, it is assumed that the reader has some knowledge of the work described in the second thesis, as its contents are not fully described in this paper.

I would like to thank my team-mate, Roger M. Koteng, for cooperation during this work. I will also like to thank my supervisors at NTNU and Chipcon; Kjetil Svarstad, Lars Lundheim, Per Torstein Røine and Robin Hoel for valuable counselling and guidance along the way.

Trondheim, 17.06.2006

Hallvard Næss

Contents

Abbreviations

ADC	Analog to digital converter
AGC	Automatic gain control
AGU	Address generation unit
ALU	Arithmetic logic unit
ASIC	Application specific integrated circuit
ASIP	Application specific instruction set processor
CORDIC	Coordinate rotation digital computer
DAC	Digital to analog converter
DMA	Direct memory access
DSP	Digital signal processor *or* processing
DSSS	Direct sequence spread spectrum
FFT	Fast Fourier transform
FIR	Finite impulse response
FPGA	Field programmable gate array
HW	Hardware
IP	Intellectual Property
ISA	Instruction set architecture
LIFO	Last in first out
LQI	Link quality indicator
Lsb	Least significant bit
MAC	Multiply-accumulate *or* Media access control
MIMD	Multiple instructions multiple data
Modem	Modulator-demodulator
Msb	Most significant bit
OFDM	Orthogonal frequency division multiplexing
PC	Program counter
PHY	Physical layer
QPSK	Quadrature phase shift keying
RAM	Random access memory
RISC	Reduced instruction set computer
ROM	Read only memory
RSSI	Received signal strength indicator
RTL	Register transfer level
SDR	Software defined radio
SFD	Start of frame delimiter
SIMD	Single instruction multiple data
SNR	Signal-to-noise ratio
SW	Software
VLIW	Very long instruction word
WLAN	Wireless local area network
WPAN	Wireless personal area network

1 Introduction

1.1 Problem

As wireless communications systems keep evolving at an ever increasing speed, the amount of different radio standards keeps growing. Due to this development, it is a need for devices such as mobile phones, laptops etc to handle a large amount of different radio standards. However the low-cost and low-area requirements for the transceivers in such devises are not becoming smaller. This raises a need for a higher degree of hardware reuse and flexibility in radio systems to make it possible to implement several communication standards on the same piece of hardware. These issues have led to an increasing interest of using reconfigurable rather than fixed-function hardware for realizations of radio systems. The concept of softening the hardware of radio architectures is often referred to as Software Defined Radio (SDR).

In a typical digital single-chip radio transceiver, digital demodulation and modulation is performed in dedicated hardware, MAC and link functionality in a combination of software and hardware, and network and application in software/firmware. To make SDR implementations of such transceivers feasible, it is a need for programmable implementations of the heavy signal processing tasks of the modulator/demodulator (modem).

Chipcon, a leading supplier of low-power low-cost radio chips, are currently evaluating the feasibility of developing a digital signal processor (DSP) to implement existing and future radio physical layer (PHY) protocols. The focus of their interest is mainly at reducing their time-to-market of future radio chips by minimizing the required hardware adjustments when a chip implementing a new standard or update is to be developed. Also, the ability of fine tuning the modem algorithms on the actual silicon device, may improve performance for a programmable solution.

The major challenges of designing such a DSP core are the tough area and power efficiency requirements that must be satisfied to make this solution competitive to an ASIC implementation. In order to satisfy these requirements, the DSP will need extensive optimizations for the most demanding tasks of common radio physical layer algorithms. Additionally, a high degree of flexibility is essential to provide a high throughput across a wide range of algorithms and implementations.

1.2 Scope

The scope of this thesis is to develop and describe a DSP architecture especially optimized for implementation of low-complexity, low-power modems. The development of the architecture will mainly be based on the IEEE 802.15.4 standard. However, it is

also a focus on flexibility to make implementation of other standards feasible by the architecture. The work included in this thesis has been restricted to a system level and a partial register transfer level *description* of the architecture. The actual implementation of the architecture is not included in this work.

The work of this thesis has been done in co-operation with a second parallel master project. In this project suitable algorithms and radio signal-chains are found to identify critical operations for the DSP. It is referred to the resulting thesis [1] of this project for a throughout description of these issues.

1.2.1 Requirements

To study the feasibility of the architecture, it was given requirements on area and power consumption for the DSP. These requirements should be fulfilled in order to make the cost of a DSP solution comparable to a traditional ASIC solution. The following requirements were given:

- The total equivalent gate count of the DSP should not exceed a total of 40000 gates. This gate count includes all necessary memories for the DSP.
- Implementation of the IEEE 802.15.4 physical layer should be possible at a typical current consumption of 5mA and a maximum of 10mA.

1.2.2 Assumptions for RF front-end

The DSP will operate as a part of a larger radio system, a block diagram of the assumed radio system including the RF front-end is shown in figure 1.

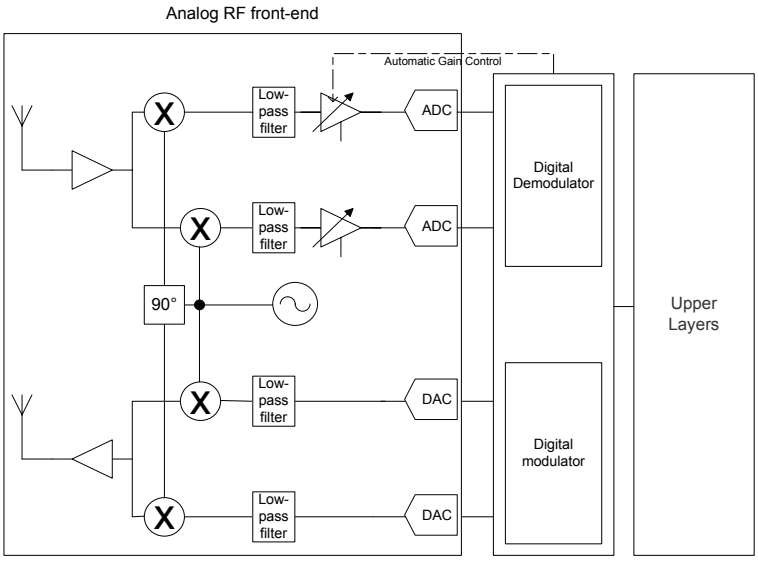

Figure 1 - System overview and analog front-end

The DSP will interface to the RF front-end on one hand, and the upper radio protocol layers on the other. The basic function of the RF front-end is to amplify the signal

received at the antenna and convert it from a carrier frequency down to baseband. This conversion is performed in one step, i.e. direct-conversion. The analog to digital converters (ADC) will convert the real and imaginary values of the received down-converted signal to digital form. The sampling ratio of the ADC's will be between 4 and 8 MSamples/s, and the accuracy will be of 12 bits. The digital to analog converters (DAC) are used to convert the complex values received from the DSP to analog form. The sampling ratio and accuracy of these are assumed to be the same as for the ADC's. For a more throughout description of the RF front-end, it is referred to [1].

1.3 Software defined radio

The idea of SDR is basically that it should be possible, by the use of programmable and/or reconfigurable hardware, to alter much of the radio functionality simply by replacing software. By describing modulation, demodulation, error correction and other baseband processing techniques in software, multiple radio physical layers could be realized on the same, or nearly the same hardware platform. The potential benefits of such implementations are many, especially the time-to-market aspect are a driving force currently leading to an increasing commercial interest of these concepts.

The SDR Forum is an international industry association dedicated to promoting the development and use of SDR for advanced wireless systems. The SDR forum divides the term "Software defined radio" in several tiers based on the capabilities of the SDR [2].

- **Tier 0 – Hardware radio (HR):** The radio is implemented using hardware components only and cannot be modified except through physical intervention.
- **Tier 1 – Software controlled radio (SCR):** Only the control functions of the radio is implemented in software - thus only limited functions are changeable using software. Typically this extends to inter-connects, power levels etc. but not to frequency bands and/or modulation types etc.
- **Tier 2 – Software defined radio (SDR):** The radio provide software control of a variety of modulation techniques, wide-band or narrow-band operation, communications security functions (such as hopping), and waveform requirements of current and evolving standards over a broad frequency range. The frequency bands covered may still be constrained at the front-end requiring a switch in the antenna system.
- **Tier 3 – ideal software radio (ISR):** ISRs provide dramatic improvement over an SDR by eliminating the analog amplification or heterodyne mixing prior to digital-analog conversion. Programmability extends to the entire system with analog conversion only at the antenna, speaker and microphones.

Due to limitations in the RF-front-end, the architecture described in this thesis will not reach a programmability exceeding tier 2.

2 Baseband processing

This chapter will provide a brief overview of the standard the DSP is aimed at and a short description of various hardware blocks relevant for baseband processing.

2.1 IEEE 802.15.4 overview

The DSP architecture proposed in this thesis was mainly directed at implementing the IEEE 802.15.4 [18] physical layer (PHY). The standard is especially aimed at applications requiring very low cost and power consumption at the expense of a low data throughput.

The physical layer is characterized by and responsible of the following:

- To keep the power consumption at a minimum, the physical layer shall be able to activate and deactivate the radio transceiver at request.
- A received signal strength indicator (RSSI) must be included to estimate the activity within the current channel.
- Link quality index (LQI) shall be applied to the received packet as an estimate of the link quality of the reception.
- Clear channel assessment (CCA) shall be provided to investigate if the channel is idle before packet transmission.
- Channel frequency selection.
- The actual transmission and reception of data on the channel, including modulation and demodulation. The modulation technique utilizes *direct sequence spread spectrum* (DSSS) and *offset quadrature phase shift keying* (O-QPSK). The data rate is 250 kb/s. For a throughout description of the modulation and demodulation techniques used, it is referred to [1].

2.2 Building blocks for baseband processing

This section will provide a short description of various hardware modules possibly used for baseband processing [3].

2.2.1 Application-specific integrated circuit

An application-specific integrated circuit (ASIC) is an integrated circuit designed for a single particular use. Since an ASIC is designed with a single application in mind, they usually display the best power efficiency, area and computational power. However, the high performance comes at a cost. An ASIC provide close to zero flexibility, and the chances of being able to use an ASIC for another application than the one it is designed for is minimal. The design time of an ASIC will usually be very long compared to the time of altering programmable hardware, this result in a slow time-to-marked for ASIC's.

An ASIC is unsuited for software radio exceeding *tier 1* or even *tier 0* of the definitions provided by the SDR Forum.

2.2.2 Field programmable gate array

A field programmable gate array (FPGA) is a chip or chip module whose hardware functionality is programmable. This is performed by changing the values of memory elements which determines the functionality of configurable logic blocks that the FPGA is compound of. By making their hardware programmable, an FPGA will allow some flexibility while a relatively high computational power is achieved. However, the many lengthy routing lines between the CLB's of an FPGA cause a high power consumption and area. FPGA's are frequently used in SDR systems when area and power consumption is non-critical. For such systems an FPGA is often used in collaboration with a digital signal processor to accelerate functions unsuitable for software implementation.

2.2.3 General microprocessor

A general microprocessor is, in opposite to an ASIC, especially optimized for flexibility. The general microprocessor usually consists of one or a few functional units, controlled by a simple instruction set with few instructions (RISC). Few optimizations for specific functions and algorithms are included in a general microprocessor; this makes the processor able to operate at a reasonable high speed for a large range of applications. A change of application to execute can be performed very fast in software programmable processors. By performing a simple branch to a new location in the program memory, the processor will start processing a new application. Due to few optimizations for digital signal processing algorithms, general microprocessors are unsuitable for implementations of radio physical layers.

2.2.4 Digital signal processors

When the application domain of a processor is limited, it is suitable to optimize the instruction set and processor architecture for commonly executed operations. A digital signal processor (DSP) is a programmable processor especially optimized for digital signal processing. Digital signal processors are further discussed in chapter 3.

2.2.5 Application specific instruction set processors

An Application specific instruction set processors is a programmable processor especially optimized for running a single application. By such optimizations, the efficiency of the processor can become comparable to that of an ASIC, while some flexibility is preserved by making the architecture programmable.

3 Domain specific digital signal processors

Digital signal processing algorithms are usually very repetitive in nature, in which the same mathematical operation is performed on a large number of input samples [4]. A digital signal processor (DSP) is a microprocessor especially optimized to perform such tasks. This enables a DSP to be very efficient in terms of speed and power efficiency compared to a general microprocessor when such tasks are executed. By applying further optimizations for a certain domain of digital signal processing, the efficiency of the processor can be enhanced even more. A domain specific DSP can be considered as a cross between a DSP and an ASIP, by being especially optimized for a limited domain of digital signal processing. The DSP proposed in this thesis is especially optimized for the domain of low-power, low-complexity baseband processing.

The following summarizes the basic requirements made for general and domain specific DSP's:

- Dedicated units for multiply-and-accumulate (MAC) operations are essential for efficient processing of many DSP algorithms. Digital filtering, correlation and Fast Fourier Transforms (FFT) are examples of DSP operations requiring efficient MAC operations.
- Parallel processing architectures may be deployed to speed up operations on parallel data.
- A large memory bandwidth must be provided in order to allow efficient processing of a large amount of data.
- Dedicated units handling address generation (AGU's) should be supported to avoid additional cycles for calculation of addresses.
- The instruction set and execution units should be especially optimized for the most demanding operations the DSP is supposed to handle.
- The provided addressing modes should be optimized for the algorithms to process.
- Efficient handling of loops should be supported in order to optimize the performance of repetitive operations.
- The chosen word lengths of the datapath and memory should be especially optimized for the applications at hand.

This chapter will provide an overview of the different enhancements that can be made to optimize the efficiency of a DSP.

3.1 Architectures for parallel processing

Employing a parallel architecture will allow the DSP to perform more operations each clock cycle, increasing the speed of the processing without increasing the clock frequency. Additionally, such enchantments will decrease the control overhead for each operation, making the DSP more power efficient.

Another highly relevant issue considering the power efficiency of a parallel architecture is the possibility of using wider memories. Generally, a memory which stores multiple words of data in each memory location will require less power per word that is accessed [5]. Potentially, employing a parallel architecture will thereby greatly reduce the power consumption due to memory accesses compared to a traditional scalar architecture which accesses one word per cycle from each of the memories.

In this section some of the basic principles of parallel processor architectures are discussed.

3.1.1 Pipelining

By inserting registers in the critical path of the control and data path of a processor, the execution of operations can be divided into several stages. By performing the different pipeline stages for following instructions in parallel, the critical path of the system is decreased significantly. The reduction of the critical path makes the system able to run at a higher clock frequency with only minor additional system complexity.

A processor pipeline is usually divided into the following steps [5]:

- **Instruction fetch:** The instruction is fetched from program memory.
- **Operand fetch:** The operands for the operation are accessed.
- **Execute:** One or more steps are used for execution of the instruction by the datapath.
- **Result store:** The resulting operands are stored in memory.

Figure 2 shows the basic principle of a processor pipeline.

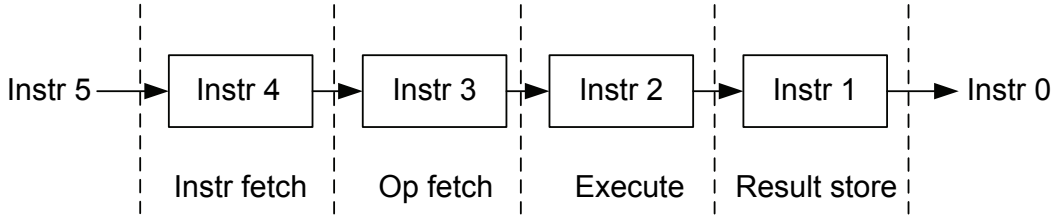

Figure 2 - Processor pipeline

A problem of datapath pipelining arises when executing an instruction dependent on data currently being modified in the pipeline. In a deep pipeline such dependencies may waste a large amount of clock cycles, since the instruction will be unable to execute until the previous instruction has completed. Also program branching may result in a waste of clock cycles since the pipeline must be flushed when the program branches [5].

Since the logical depth of the *execute* stage may vary, some processors utilize a variable pipeline depth by pipelining the execution step of demanding operations over multiple stages. This may cause conflicts when a pipelined instruction is followed by a non-pipelined, since the instructions may need access to the same resources simultaneously. This adds to the complexity of a variable-length pipeline, since such conflicts must be detected and resolved by dedicated hardware.

3.1.2 Single Instruction Stream, Multiple Data Stream

In a typical Single Instruction stream, Multiple Data stream (SIMD) architecture, a single operation stated by the instruction will be executed by multiple execution units operating in parallel [5]. The principle of the SIMD architecture is shown in figure 3.

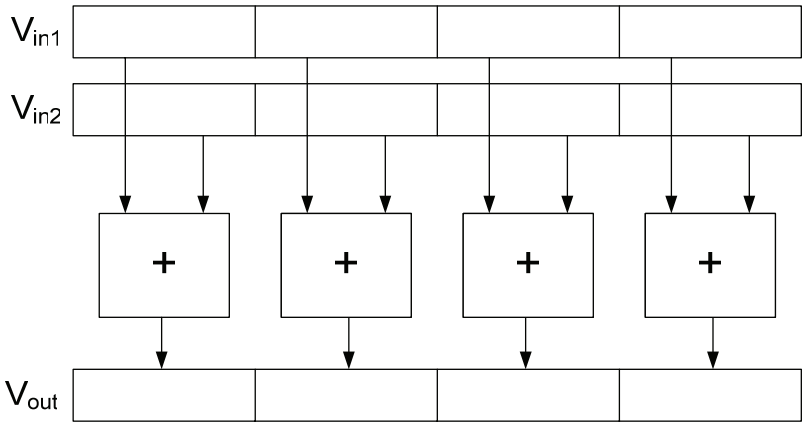

Figure 3 - SIMD architecture

V_{inn1} and V_{inn2} denote vectors which are added in parallel, resulting in a vector V_{out}. The principle of the SIMD architecture is to perform the same operation on multiple words of a vector simultaneously. This exploits the data parallelism that can be found in many types of operations found in the DSP and baseband processing domains, thereby increasing execution speeds. A benefit of the SIMD approach is that the memory architecture may be kept quite simple since each vector of data may be placed in one large register or RAM location.

3.1.3 Very long instruction word (VLIW) processors

A considerable restriction of a SIMD processor is that the same operation has to be performed by each execution unit, resulting in poor efficiency when the rate of data parallelism is low. The VLIW architecture solves this problem by introducing very long instructions consisting of multiple operation codes, each controlling an execution unit [6]. The instruction also needs to address data for all of the execution units if they are to operate efficiently in parallel. Figure 4 shows the structure of a typical VLIW architecture.

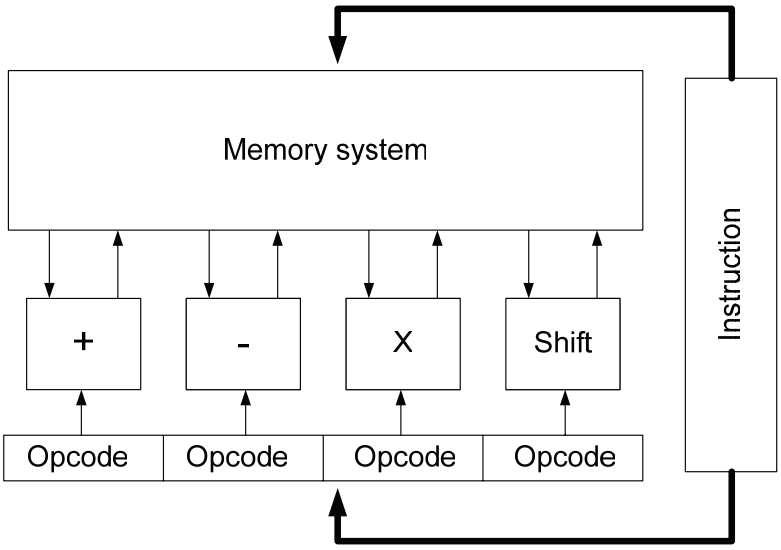

Figure 4 - VLIW architecture

Due to the increased flexibility, the VLIW processor will usually perform efficiently over a wider range of applications than the SIMD architecture. However, to fully utilize the execution units of a VLIW processor the memory architecture might become a bottleneck. For example, if 4 execution units where to obtain single cycle throughput, a memory architecture with eight read ports and four write ports would be required. As each unutilized functional unit will cause the corresponding operation code to be filled with a NOP instruction, VLIW processors usually exhibit a poor code density and an unnecessary large program size.

Both for a VLIW and a SIMD processor, scheduling of operations and execution unit allocation are typically performed by software compilation. This saves costly hardware implementation of these processes, and also makes execution-time deterministic.

3.1.4 Superscalar processors

To improve the low code density of VLIW processors, a superscalar architecture may be deployed. As opposed to the VLIW approach, a superscalar processor is able to fetch multiple sequential instructions per clock cycle. These instructions are then consecutively packaged and dispatched to the execution units. The principal structure of this architecture is shown in figure 5.

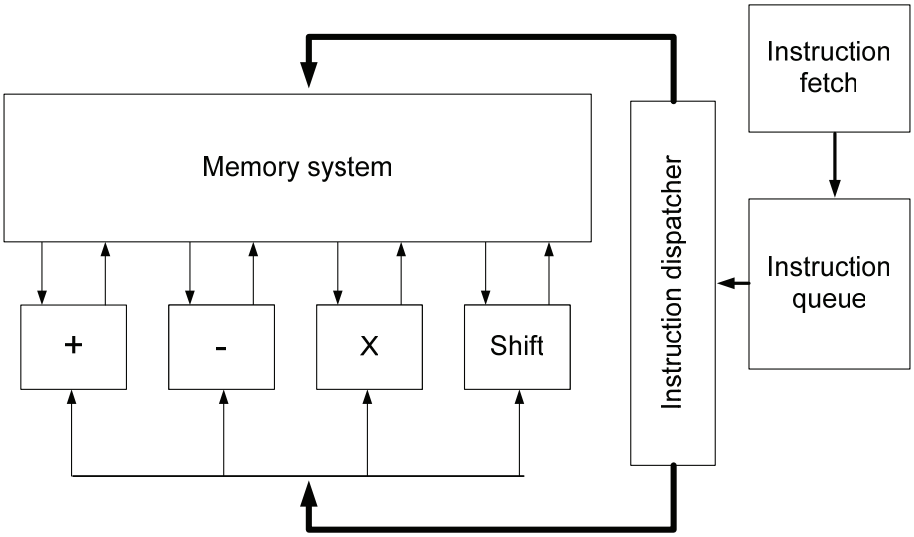

Figure 5 - Superscalar architecture

A key issue to obtain high performance in a superscalar architecture is the instruction dispatcher. Several instructions will be fetched from the program memory simultaneously, and before each instruction can be assigned to an execution unit, some sort of scheduling based on data dependencies between the instructions must be performed. The overall performance of a superscalar processor will in a high degree depend on how well the instruction fetcher and the dispatcher manage to keep the processor cores busy.

The superscalar approach allows higher code density and shorter instruction words than the VLIW approach, and it is also in a higher degree possible to make this architecture code-compatible to other processors. The downside is a more complex control unit which introduces a significant overhead slowing down processing speed. The run-time dispatching of instructions also makes execution time non-deterministic, and with that less favorable for real-time processing.

3.1.5 Multiple Instruction stream, Multiple Data stream

In a Multiple Instruction stream, Multiple Data stream (MIMD) architecture, parallelism is characterized by the concept that each processing element is really a processor operating asynchronously and independently. This allows multiple streams of instructions to be active simultaneously, often referred to as *Multithreading* [5]. A block representation of the basic MIMD principles is shown in figure 6.

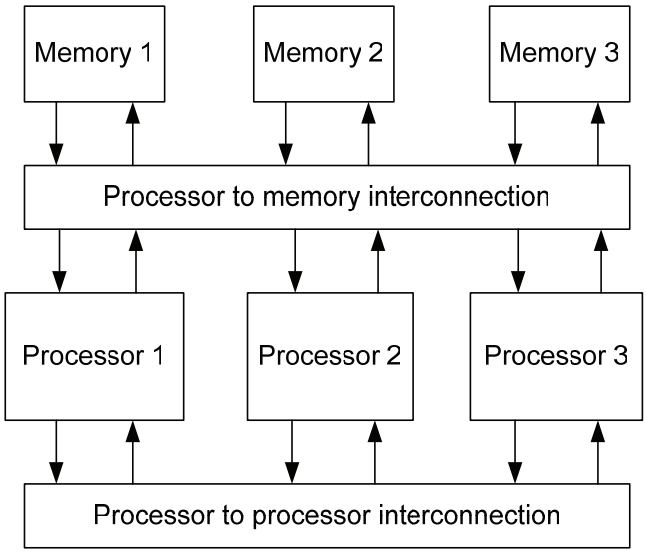

Figure 6 - MIMD architecture

By dedicating each of the processors of a MIMD architecture to execution of a set of related operations, each operation can potentially be executed very efficiently and with a low overhead. A downside of the MIMD approach is that multiple memories will be required in order to make the processors run in parallel. Also the interconnection and synchronization between different processors and memories will introduce expenses in term of area and throughput.

3.1.6 Discussion

The proposed DSP architecture is mainly based on the principles of SIMD and VLIW processing. Especially the possibility of employing a simple memory structure while a high throughput is achieved, favored a SIMD approach. The VLIW capabilities of controlling multiple functional units while the complexity is kept at a minimum were also considered advantageous. The use of a multithreaded architecture was found unsuitable considering the strict area requirements that was given for the DSP. Especially the memory structure was considered too complex to satisfy the given constraints. The superscalar approach was neither followed. This is because compiler-friendliness was not considered a big issue for the DSP, and because the instruction dispatcher was considered too complex.

3.2 Memory architecture

The efficiency, power consumption and area of a DSP are in a large extent depending on the memory architecture. The memory can be divided in two subsystems – the program memory, and data memory. The program memory will store and provide the instructions to be executed by the processor, while the data memory stores and provides data to be altered by the execution units in the DSP datapath. To provide a single cycle throughput,

it is needed to perform one memory access from the program memory and at least one memory-read access and one memory-write access for the data memory each clock cycle. Two main types of memory architecture are defined to provide the needed amount of memory accesses each cycle, the Harvard architecture and the modified von Neuman architecture.

The basic difference between these two architectures is that the Harvard architecture divides the program memory and the data memory to two separate memory spaces with multiple data busses. In a von Neuman architecture only one data bus is provided; this is ran at a higher clock frequency than the computations, allowing multiple memory accesses each cycle.

Since the DSP processor to be designed is aimed at low-throughput radio PHY's, the necessary memory will consist of small on-chip memories making the cost of multiple busses relatively small. Also, running the memories at a high speed will require the use of high speed memories, resulting in lower power efficiency. Thus, some kind of a Harvard memory architecture consisting of multiple busses should be the best choice for the DSP architecture.

3.3 Programming model

The programming model of a processor is the processors interface to the programmer or the compiler. The programming model will basically consist of the instruction set and the available addressing modes of the processor. To make the programming of a processor as easy as possible, it is necessary to provide a well-structured programming model with a well defined instruction set and straightforward use of the available addressing modes.

3.3.1 The instruction set

The choice of instruction set format may have great influence on the system cost of a processor, and trade-offs will have to be made between orthogonality, word length, programmability and throughput.

3.3.1.1 Orthogonality

Orthogonality is a principle by which two variables are independent of each other [7]. In the context of an instruction set, the term usually refers to making the different addressing modes uniformly available for different operations. Also, a complete and regular set of instructions are considered orthogonal [8].

On the machine code level, the instruction word of an orthogonal instruction set should be divided into different subfields, independent of each other. For example, the operation to be performed, the addresses of the data to be read and the location to write the result to may be specified in different subfields of the instruction, making each subfield describe the same type of functionality across a large number of different instructions.

A highly orthogonal instruction set leads to easier programming and simpler decoding logic, but will generally result in a larger instruction word.

3.3.1.2 The instruction word width

Since a large instruction word width will require a wider program memory and bus width, the chosen size of the instruction words will directly affect the area and power dissipation within the processor. The use of an orthogonal instruction set will generally increase the word width of the instructions. All subfields of an orthogonal instruction word will not be used by all instructions, increasing the redundancy of the instructions.

An approach to achieve a smaller instruction word length is to divide the instruction set into groups of similar instructions. To avoid making sacrifices on performance, the instructions of different groups should be unable or unlikely to operate in parallel. An example would be to use two different instruction types for control operations such as branches, and data operations such as additions.

3.4 Addressing modes

In this section, some addressing modes commonly available in DSPs are discussed. The available addressing modes state how a programmer is able to declare the data to be used for an operation. For further reading, see [6].

3.4.1.1 Register addressing

In this addressing mode the data to be accessed is contained in registers. The registers to access and write to is specified by field in the instruction. If the value of register R1 (data1) is added to R2 (data2) and the result are saved in R1, the instruction could look as follows:

ADD R1, R2, R1 ; data1+data2→R1

3.4.1.2 Direct addressing

The address of the data to be accessed is specified by a field in the instruction. If the data contained at the address *data1 should be added by the data stored at the address *data2, the instruction could be look follows:

ADD *data1, *data2, R1 ; data1+data2→R1

3.4.1.3 Indirect addressing

The address is determined by the content of a register, e.g. the address register of an address generation unit (AGU). This addressing mode often supports an altering of the register before or after the operands are fetched. The value of the register can usually

either be added or subtracted by a value given by the instruction, a register or an implied value of 1 [6]. In the following example, the data at the address given by an address register, R_{addr1}, is added to the data at the address of R_{addr2}. If the registers contains *data1 and *data2, the instruction could look as follows:

ADD R_{addr1}, R_{addr2}, R1 ; data1+data2→R1

3.4.1.4 Immediate addressing

In this addressing mode, the data is given in the instruction itself. If data1 contained by R1 should be incremented by a constant, C, the instruction could look as follows:

ADD R1, C, R1 ; data1+C→R1

3.4.1.5 Circular addressing

Circular addressing, or *modulo addressing* is an addressing mode that is especially useful in DSP algorithms. If circular addressing is applied, the address pointer of the address register will automatically start over from the beginning if it reaches a certain value. This mode is useful for implementing circular buffers or when addressing operands of sliding windows (e.g. convolution).

Circular addressing is usually implemented in one of two ways. The start and end address of the circular buffer can be stored, by setting the address to the start address when the end of the buffer is reached a circular buffer is implemented. Otherwise, only the buffer size can be stored. The start of the buffer will then be restricted to an N-word boundary, where N is the smallest power of 2 that is greater than or equal to the buffer size. The first implementation gives a higher flexibility, while the second reduces the complexity.

3.4.1.6 Bit reversed addressing

This is an addressing mode optimized for computation of FFT's (Fast Fourier Transforms). Since the use of FFT was found unbeneficial for low-throughput radio standards [1], these concepts are not further discussed.

3.5 Loop handling

Because of the repetitive nature of digital signal processing algorithms, it is essential for a DSP to provide efficient program looping. This section will describe some of the most commonly used approaches to improve the efficiency of execution of looped instructions.

3.5.1 Address Generation Unit

To perform efficient looping in DSPs, it is a need for dedicated units handling address generation. These units will calculate the addresses of operands and write locations in

parallel with execution of other operations. In this way, these addresses will not have to be given directly in each instruction or be calculated explicitly by separate instructions. The AGU's will typically be controlled by separate fields in the instruction, and should be able to support a variety of address modifications based on the addressing modes provided by the DSP.

3.5.2 Software looping

Looping performed by software instructions is often referred to as *Software looping*. This means that all loop-handling operations are performed by branch instructions given in the program. Consequently, for each iteration of a loop, instructions must be added to modify a loop-count register and perform a check to determine if a branch should occur [9]. Additional cycle delays are introduced in pipelined datapaths, since the datapath must be flushed before each branch operation.

3.5.3 Hardware looping

To avoid the extra cycles introduced in a software loop, a hardware support for looping may be implemented. By letting the hardware loop perform tasks such as loop counting, evaluation of dependencies and branching in parallel with other operations, both throughput and power consumption may improve considerably. Three main types of hardware loops are discussed [9]: Single instruction looping, block of instructions looping and block of nested looping.

3.5.3.1 Single instruction looping

Many program sequences may consist of a single instruction repeated a number of times, performing the same operation on a vector of multiple words. By freezing the value of the program counter until a loop-counter has iterated a specified number of cycles, such instruction loops could be efficiently handled by hardware. In addition to performing such loops with zero cycle overhead, the power consumption due to memory accesses from program memory will also be significantly reduced.

Typically the hardware realization of the loop will consist of a decrementor/incrementor to count the number of iterations, a register to store the number of iterations to be performed and a comparator to determine if the loop has finished. The hardware cost of such an implementation will typically be very low, determined by the maximum number of iterations to be performed.

3.5.3.2 Block of instructions looping

If a sequence of instructions are to be repeated a number of times, a block of instructions loop may be implemented. This implementation will typically be a bit more complex than in the case of instruction repeating [9]. Usually such loops are implemented by saving the end- and start-address of the loop and the number of loop-iterations. When the current

address of the program counter equals the end-address, the loop will start over and a loop counter is decremented. The loop will finish when the loop counter reaches zero.

3.5.3.3 Nested looping

Hardware implementation of nested loops can be achieved in a similar way. However, the registers storing the start-address, end-address and number of iterations of each loop, will have to be stored in a LIFO-stack. The depth of the stack determines the number of nested loops the processor is able to handle.

3.5.3.4 Instruction buffering

For short loops, the power consumption due to program memory accesses may be decreased significantly by employing a small local buffer for storage of looped instructions. By this enhancement, the program memory will only be accessed the first time the loop is executed. For larger loops, the hardware cost of an instruction buffer will become very high, making this approach unsuitable.

3.6 Acceleration techniques

What separates a semi-custom processor like a DSP or an ASIP from a general purpose processor is basically the use of accelerators to enhance execution speeds for certain operations. According to [8], acceleration can be performed on instruction level or function level.

3.6.1 Instruction level acceleration

Instruction level acceleration means that common operations in an application or application domain are given a specific instruction in the processors instruction set. One of the most common examples of instruction level acceleration is the MAC operation used in nearly every DSP instruction set. The amount of changes that has to be done in the processor datapath and instruction decoder to add a new instruction may vary considerably depending on how much the new instruction part from the existing ones. The trade-off between gained efficiency due to saved clock cycles vs. efficiency-loss due to higher complexity has to be considered before adding a new instruction. A highly orthogonal instruction set and a well organized instruction decoder may ease the implementation of future instructions in an existing architecture.

3.6.2 Functional level acceleration

Functional level acceleration means to partition a whole algorithm or subroutine to a fixed-function hardware module. This would typically lead to a higher increase in performance than for instruction level acceleration, but will also imply a larger hardware cost and a lower flexibility. Algorithms that requires high performance or would spend much more power implemented in software (SW) than hardware (HW) should possibly

be executed by a functional level accelerator. The use of accelerators can also potentially increase the parallelism of a system since the accelerator should be able to run in background while the program continues to execute. The possibility of reuse of accelerators is an issue that should be considered. An accelerator that only can be used to implement a specific algorithm only used by one certain standard would be much more expensive in terms of e.g. development cost than one for a general algorithm applicable in a broad range of applications. The process of deciding which functions should be implemented in HW accelerators is often referred to as HW-SW partitioning.

3.7 Low-power considerations

In addition to the architectural enhancements discussed earlier in this chapter, some other issues considering power efficiency should be kept in mind when designing a processor. This section will give a brief description of various techniques that may be applied to keep power consumption at a minimum.

3.7.1 Low-area implementation

A trade off between performance and area/power consumption exist for the implementation of most functional units. The implementation of an adder is used as an example illustrating this. A ripple carry adder will have a considerably lower area, and thereby also power consumption, than a faster implementation such as a carry look ahead adder. By parallel enhancements such as pipelining or a parallel architecture, it should be ensured that low-area implementations of functional units are sufficient to obtain the desired throughput.

3.7.2 Low power memories

Memories supplied from IP-vendors can usually be optimized for low area, high speed or low power [5]. By ensuring that low-power optimized memory blocks are sufficient to reach the area and clock frequency requirements of the system, the power consumption of each memory access will be reduced. Also, keeping the amount of memory accesses as low as possible is essential to obtain power efficient processing.

3.7.3 Clock gating and operand stopping

Usually, all parts of a processor are not used by all instructions. Clock gating and operand stopping are techniques used to turn off unused units, making them consume close to zero current when disabled [10]. When clock gating is applied, the system clock is simply disabled for the unused functional unit. Operand stopping is applied by keeping the input values of unused units stable, avoiding unnecessary switching. On the architectural level, an orthogonal instruction set and a modular architecture should make the implementation of such enhancements simpler.

3.8 Related work

In this section, some related baseband processors used for SDR are briefly discussed.

3.8.1 BaseBand Processor 1 (BBP1)

Eric Tell at the University of Lindköping has proposed an architecture especially designed for software baseband processing [8]. The proposed architecture was optimized for running WLAN applications.

The idea behind the BBP1 is basically to connect two DSP cores to multiple function level accelerators and memories through a configurable network. An overview of the architecture is shown in figure 7.

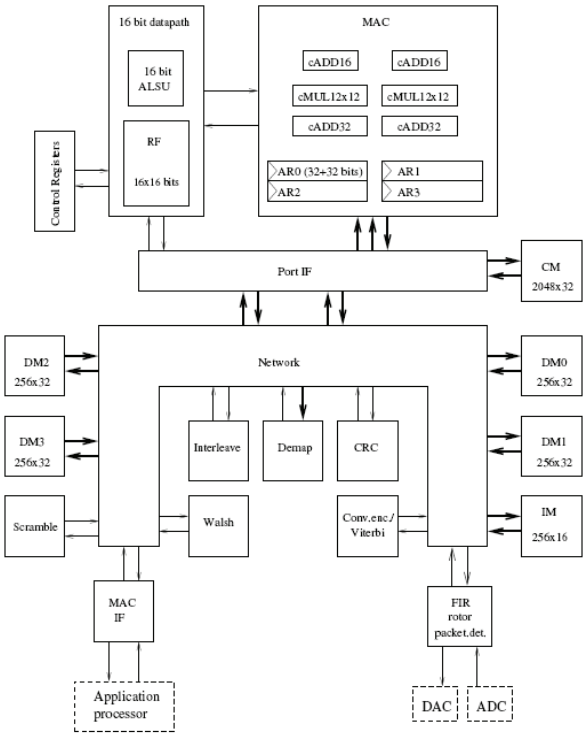

Figure 7 - The BaseBand Processor 1 architecture

The DSP core of the BBP1 architecture consists of two datapaths. A simple 16 bit RISC like processor handles control operations, and a dual complex MAC unit handles complex valued calculations. When operating, each accelerator is given direct access to one of the small memories through the crossbar network, making multiple accelerators able to work in parallel. This ensures a high throughput by parallelism, while the power consumption is kept low by each operation being executed by a nearly optimized unit.

3.8.2 Sandblaster

Sandbridge technologies have developed a multithreaded processor capable of executing DSP operations, embedded control and java code [12]. The processor is optimized for handheld wireless communication.

To obtain a high performance, SIMD capabilities and deep processor pipelines are included in the architecture. To minimize the instruction memory, each fetched instruction may be compound of multiple operations. The architecture utilizes multithreading, by letting up to 8 simultaneous threads control the execution of various execution units in the core.

3.8.3 Montium

The Montium processor is developed at the University of Twente, and is a DSP processor especially optimized for mobile wireless terminals [13]. The Montium employs a *coarse-grained* reconfigurable architecture. This means that a system-on-chip can consist of multiple tile-processors communicating via a reconfigurable network-on-chip. Each tile-processor consists of multiple execution units connected to multiple memories via a reconfigurable crossbar network. The reconfigurability makes the complexity of the processor adaptable to the computational power of the task at hand.

3.8.4 Discussion

All of the discussed baseband processors are optimized for high-throughput standards such as WLAN or 3G applications. Especially the memory structures of these architectures would have to be made much simpler in order to achieve the area requirements of low-throughput standards. Also, complex control and interconnection features such as multithreading and reconfigurability was considered too complex to be adopted by a low-area DSP. The idea of function level acceleration was considered as a possible solution for the most computationally intensive operations the proposed DSP will have to handle.

4 Analysis of operations

In this section, the most computational extensive operations required implementing the signal chains of the IEEE 802.15.4 and related physical layers are analyzed. For further information on the studied algorithms, it is referred to [1]. The following was found as the most critical operations the DSP should be able to compute.

- Support of multiply and accumulate (MAC) operations on both 2 and 8 bit numbers are essential i.e. to implement multiple forms of convolution based operations and to find the signal power of a data stream.
- Optimization for CORDIC rotation and vectoring are necessary for efficient frequency offset estimation and correction.
- Shift operations and logical operations must be supported e.g. to perform scaling and quantization (divide by 2^N). These operations are also useful when packing or unpacking sub-word sized values to or from a register.
- Saturation and rounding should be applied to keep a high precision through the signal chain.
- Arithmetic operations such as additions and subtractions should be included.
- Compare instructions to efficiently find maximum / minimum of a vector of multiple values are needed.
- The DSP should be able to perform Lin/log conversion due to the specification of the RSSI of the IEEE 802.15.4 standard.

4.1 Convolution

A substantial part of the algorithms used in baseband processing are based on convolution. FIR filtering, matched filtering and cross-correlation are examples of operations being highly relevant for modem implementations. It is therefore of great importance to achieve a high throughput while minimizing the power consumption for such operations. Convolution over a shorter sequence, C_j, and longer sequence, D_{i+j}, of real data is defined as [4]:

$$(C * D)_i = \sum_j^N C_j \times D_{i+j}$$

N denotes the length of the shortest sequence and *i* is the sample number.

4.1.1 FIR-filters

Channel filters, matched filters and correlators with unknown timing will basically be implemented as FIR-filters. An N-tap FIR filter can be described as a convolution of a continuous stream of data and N coefficients. The data sequence is shifted by one for

each FIR computation. The first M output samples resulting from the FIR-filtering will be as follows:

$$(C*D)_0 = C_0 \times D_0 + C_1 \times D_1 + C_2 \times D_2 + ... + C_N \times D_N$$
$$(C*D)_1 = C_0 \times D_1 + C_1 \times D_2 + C_2 \times D_3 + ... + C_N \times D_{N+1}$$
$$(C*D)_2 = C_0 \times D_2 + C_1 \times D_3 + C_2 \times D_4 + ... + C_N \times D_{N+2}$$
$$(C*D)_3 = C_0 \times D_3 + C_1 \times D_4 + C_2 \times D_5 + ... + C_N \times D_{N+3}$$

.

.

$$(C*D)_{M-1} = C_0 \times D_{M-1} + C_1 \times D_M + C_2 \times D_{M+1} + ... + C_N \times D_{N+M-1}$$

Two structures applicable of SIMD FIR-filter implementations will be discussed in this section. In the examples shown, each memory locations will hold a vector of 4 words of data or coefficients and 4 word sized multipliers will be provided by the datapath. However, the discussed principles will also be applicable to other vector sizes.

4.1.1.1 Solution 1

A straightforward SIMD implementation of a FIR-filter would be to calculate one output sample at a time. A structure for such operations is described by figure 8.

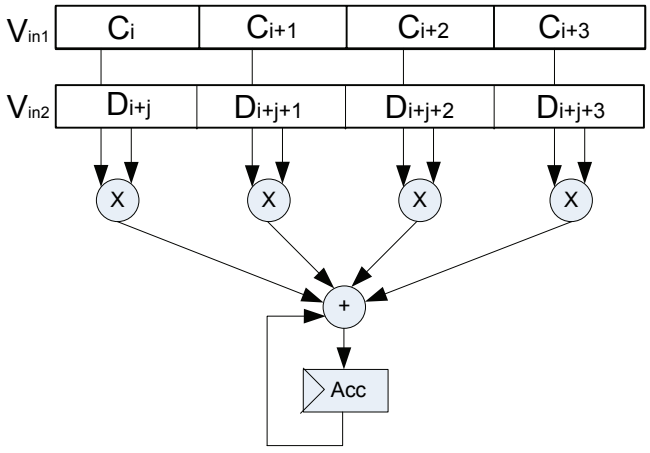

Figure 8 - SIMD fir-filter solution 1

This structure will require 2 memory accesses each cycle, one vector of data samples and one of coefficients. A problem occurs when the start of a data sequence is unaligned with the memory boundaries, i.e. the first data sample of a sequence is not contained in the leftmost location within the memory location. This can be solved by using 4 different coefficient streams where 0, 1, 2 or 3 zero coefficients are inserted at the start of the sequence. Consequently, the necessary amount of coefficients to be stored in data memory will be greatly increased.

4.1.1.2 Solution 2

A way to decrease the amount of memory accesses and the coefficient memory requirements is to compute multiple output samples in parallel [15]. The structure of figure 9 shows how this can be performed.

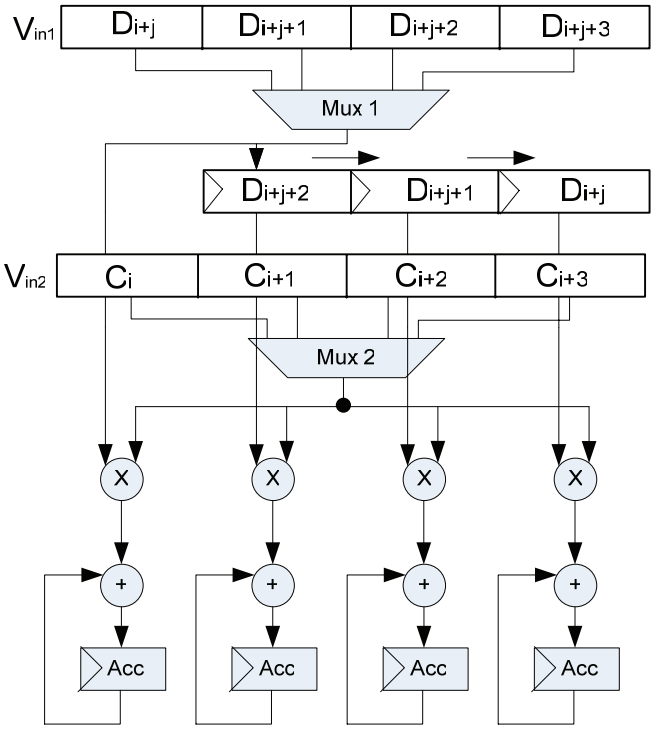

Figure 9 - SIMD fir-filter solution 2'

One output of the vector containing the data samples is multiplexed to a 3 word shift register. The words contained by the registers and the multiplexer output are inserted at the multiplier inputs. One coefficient of the coefficient-vector is multiplexed to the other inputs of the multipliers. By inserting three zero-coefficients at the start of the sequence, the data samples are allowed to propagate through the shift register before the computations starts. Table 1 shows how the computations will be performed each cycle.

Table 1 - Computations per cycle - solution 2

Cycle #	1	2	3	4	5	…	N+2	N+3
Mux 1	0	1	2	3	0	…	2	3
Mux 2	0	1	2	3	0	…	2	3
Acc 1	0*U	+0*U	+0*U	$+C_0*D_0$	$+C_1*D_1$	…	$+C_{N-1}*D_{N-4}$	$+C_N*D_{N-3}$
Acc 2	0*U	+0*U	+0*U	$+C_0*D_1$	$+C_1*D_2$	…	$+C_{N-1}*D_{N-3}$	$+C_N*D_{N-2}$
Acc 3	0*U	+0*U	+0*U	$+C_0*D_2$	$+C_1*D_3$	…	$+C_{N-1}*D_{N-2}$	$+C_N*D_{N-1}$
Acc 4	0*U	+0*U	+0*U	$+C_0*D_3$	$+C_1*D_4$	…	$+C_{N-1}*D_{N-1}$	$+C_N*D_N$

When the first data sample of a sequence is unaligned with the memory boundaries, this can be solved by simply offsetting *Mux 1* with a value of 1, 2 or 3. Each time one of the

multiplexers reaches the last word of a vector, a new vector will be loaded from memory. This means that the amount of memory accesses will be quartered compared to the previously discussed structure. Another benefit of this structure is that the accumulators also can be used for purposes such as element-wise vector additions and subtractions.

A problem of the depicted solution arises when handling FIR-filters with downsampled output. Some hardware optimizations should be applied at the ADC interface to simplify such computations when *solution 2* is employed. The resulting stream of downsampled values from the channel filter will be as follows:

$$(C*D)_0 = C_0 \times D_0 + C_1 \times D_1 + C_2 \times D_2 + ... + C_N \times D_N$$
$$(C*D)_1 = C_0 \times D_2 + C_1 \times D_3 + C_2 \times D_4 + ... + C_N \times D_{N+2}$$
$$(C*D)_2 = C_0 \times D_4 + C_1 \times D_5 + C_2 \times D_6 + ... + C_N \times D_{N+4}$$
$$(C*D)_3 = C_0 \times D_6 + C_1 \times D_7 + C_2 \times D_8 + ... + C_N \times D_{N+6}$$
.
.
$$(C*D)_M = C_0 \times D_{2M} + C_1 \times D_{2M+1} + C_2 \times D_{2M+2} + ... + C_N \times D_{2M+N}$$

This problem should only be of relevance for the channel filter [1]. To avoid half of the calculations for such cases, the optimizations depicted in figure 10 can be performed for the ADC interface.

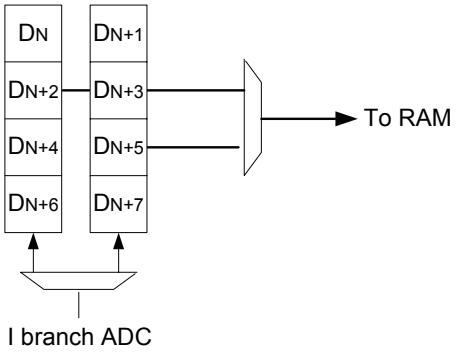

Figure 10 - Modified ADC interface

By letting the ADC interface save the odd and even samples of the incoming data stream in different memory locations, the downsampling channel filter can be implemented in the same manner as an ordinary FIR implementation. By first calculating the multiplications with odd coefficients, the stream of data samples will be shifted by one each iteration. Subsequently, the multiplication by even coefficients can be performed in the same manner.

4.1.1.3 Discussion

The throughput and required amount of memory accesses to compute an N-tap FIR filter is shown for a selection of different architectures in table 2.

Table 2 - Comparison of architectures

Architecture	SIMD factor	Cycles / output sample	Memory accesses / output sample	Required number of coefficients
Scalar MAC	1	N	2N	N
SIMD solution 1	2	N/2+1	N+1	N*2
SIMD solution 2	2	N/2+1/2	N/4+3/2	N+1
SIMD solution 1	4	N/4+1	N/2+1	N*4
SIMD solution 2	4	N/4+3/4	N/8+3/4	N+3
SIMD solution 1	8	N/8+1	N/4+1	N*8
SIMD solution 2	8	N/8+7/8	N/32+3/8	N+7

From the table it can bee seen that the throughput will be about the same for, the two solutions. However, it is a large benefit considering required number of coefficients and number of memory accesses by employing *solution 2.* In spite of the added complexity to support FIR-filtering with down-sampled outputs for *solution 2*, this solution was considered preferable.

4.1.2 Correlation

Correlation is used to find the strength of the relation between two variables. In the domain of baseband processing, correlation is used to compare an incoming symbol with multiple possible symbols defined by a protocol [1]. The correlation values found can then be used to evaluate which symbol the incoming data is most likely to represent. Correlation is also based on the principle of convolution.

The correlation values of multiple data sequences correlated with one coefficient sequence of N samples will be as follows:

$$(C*D)_0 = C_0 \times D_0 + C_1 \times D_4 + C_2 \times D_8 + ... + C_N \times D_{N\times 4}$$
$$(C*D)_1 = C_0 \times D_1 + C_1 \times D_5 + C_2 \times D_9 + ... + C_N \times D_{N\times 4+1}$$
$$(C*D)_2 = C_0 \times D_2 + C_1 \times D_6 + C_2 \times D_{10} + ... + C_N \times D_{N\times 4+2}$$
$$(C*D)_3 = C_0 \times D_3 + C_1 \times D_7 + C_2 \times D_{11} + ... + C_N \times D_{N\times 4+3}$$

By swapping coefficients and data samples in the equations, correlation of one stream of data against multiple streams of coefficients can be calculated in the same manner.

The correlation values can be found efficiently by a similar architecture as *solution 2* previously discussed. This architecture is shown in figure 11 [15].

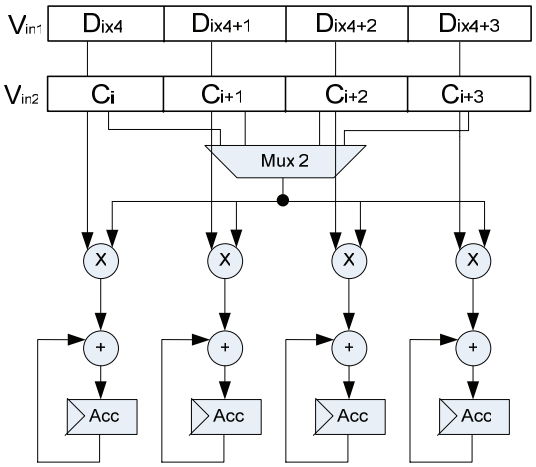

Figure 11 - Architecture for multiple data, single coefficient

The structure utilizes a parallel input of the data samples and a multiplexed input of one coefficient. The *i* denoted in the figure is the currently accessed coefficient. For this example, a SIMD factor of 4 is deployed, but the concept can also be extended to other SIMD factors.

4.1.2.1 Discussion

The throughput and amount of necessary memory accesses for the depicted architecture utilizing a selection of SIMD factors is shown in table 3.

Table 3 - Performance for single stream vs. multiple streams of data

Architecture	SIMD factor	Cycles / output sample	Memory accesses / output sample
Scalar MAC, one data stream, one coefficient stream.	1	N	2N
Two coefficient streams, one data stream.	2	N/2	3N/4+1/2
Two data streams, one coefficient stream.	2	N/2	3N/4+1/2
Four coefficient streams, one data stream.	4	N/4	5N/16+1/4
Four data streams, one coefficient stream.	4	N/4	5N/16+1/4
Eight coefficient streams, one data stream.	8	N/8	9N/64+1/8
Eight data streams, one coefficient stream.	8	N/8	9N/64+1/8

It is a clear advantage both considering throughput and memory accesses as the parallelizing increases. However, this will come at a cost of added area and complexity.

4.1.3 CORDIC

The COordinate Rotation DIgital Computer (CORDIC) algorithm can be used for a wide variety of functions [16]. For the DSP proposed in this thesis, CORDIC optimization was found critical for frequency offset estimation and compensation [1]. This compensation is necessary due to a drift of the phase of the complex samples at the ADC outputs.

Due to a large amount of branching (if/else statements), conditional additions and logical shifts used in the CORDIC algorithm [16], it would require a vast amount of instruction cycles to perform CORDIC operations on an unoptimized ALU. Hence, it is a need for certain hardware optimizations to increase the performance of this operation. Two possible implementations of a CORDIC sub-unit of the DSP have been discussed; a word serial and a word parallel structure. Both of these implementations require a *basic rotation/vectoring unit* [17] as depicted in figure 12.

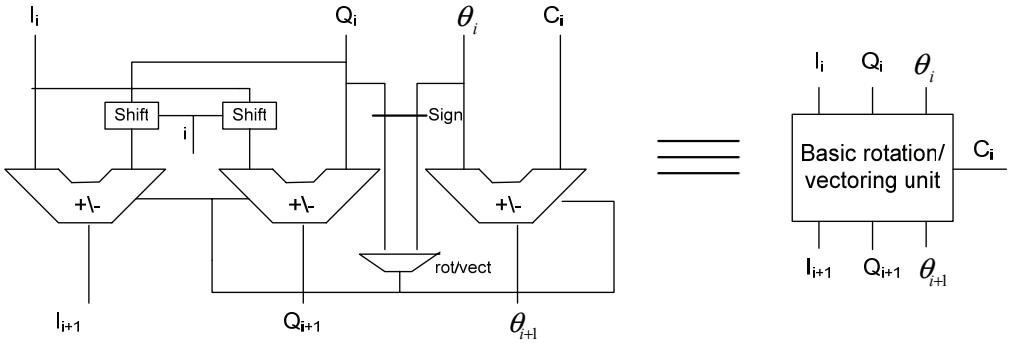

Figure 12 - Basic rotation/vectoring unit

The *rotation/vectoring unit* will have four inputs, two for the complex sample to be rotated or to find the angle of, the angle to rotate and an elementary angle coefficient. The three outputs give the complex sample and angle to be processed during the next iteration.

One bit of precision is obtained per iteration of the *basic rotation/vectoring unit*, i.e. 8 iterations are needed to rotate or vectorize an 8 bit word [17].

4.1.3.1 Word serial implementation

In a word serial implementation, the CORDIC processing is performed in a recursive fashion as shown in figure 13 [17].

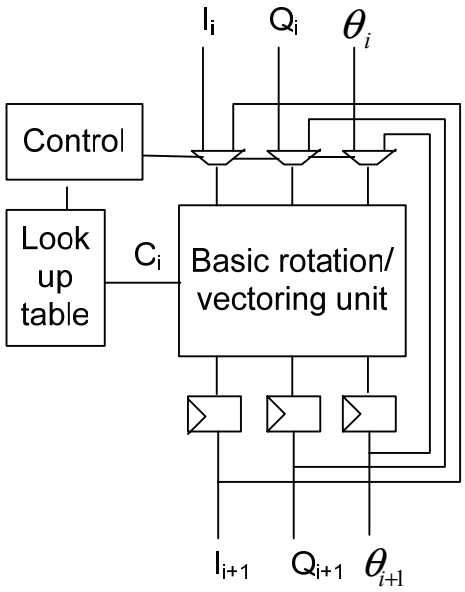

Figure 13 - Word serial implementation

This implementation requires only one rotation/vectoring unit, but will require one clock cycle per bit of obtained precision to execute. It will also require the possibility to perform a table look-up in parallel with the rotation/vectoring operation, and some control overhead for the iteration counter.

The benefits of the described CORDIC implementation are a relatively small logical depth, giving a small area and making it possible to run at high clock frequencies without pipelining. Also it should be possible to utilize this unit to also perform other operations than CORDIC, such as add and shift operations. However, the implementation is relatively slow in terms of required clock cycles to perform a CORDIC operation; one cycle is needed per bit of precision. Also the look-up table may be expensive in terms of power dissipation due to memory accesses, and there will be a need of barrel shifters to perform the shift operations.

4.1.3.2 Word parallel implementation

In a word parallel CORDIC implementation, all iterations of the CORDIC algorithm are performed in one combinatorial sequence. This method allows for single cycle vectoring and rotation by using one dedicated *rotation/vectoring unit* for each of the iterations in the CORDIC loop. The concept is shown in figure 14 [17].

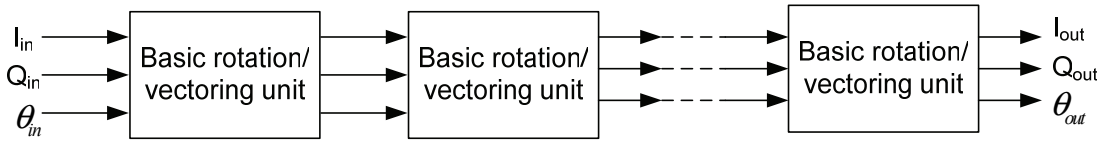

Figure 14 - Word parallel implementation

This implementation will require N rotation-elements to obtain N bits of precision, requiring 3*N adders. A number of pipeline stages may be placed between the rotation/vectoring units to increase performance. By hardwiring the shift-operations it will be no need of barrel shifters when this design is used, also the elementary angle coefficients may be hard-coded since each of the sequential *rotation/vectoring units* will only use *one* coefficient each.

The drawbacks of this implementation is the large logical depth since N adders will have to operate in serial, a number of pipeline stages is probably needed to allow a high clock frequency. Also the area requirements are quite large since a large amount of adders are needed; additionally the pipeline stages are added to the total area cost. However, omitting the barrel shifters and look-up table will save some area compared to the word serial solution.

4.1.3.3 Discussion

The approximated area and performance of the two solutions are shown in table 4.

Table 4 - Comparison of architectures

Chosen architecture	Estimated area	# Cycles/ output sample	#Pipeline stages
Word serial	1250 gates	4	1
Word parallel	4032 gates	½	2

It can be seen that the area will be about three times higher for the word parallel solution. However, this solution will achieve 8 times the performance. In terms of power dissipation, the second solution will most certainly be more efficient since this solution requires only one control cycle per iteration. Also, omitting the barrel shifters and table look-ups will have a positive impact on power consumption.

The resulting samples of a CORDIC rotation will have an error proportional to a scale factor [17]; this will not be an issue for this design since it is the mutual scaling between samples that is of relevance, not the actual value of the rotated samples.

4.1.4 Find max/min

During chip-to-symbol demapping, there is a need of finding the correlation samples with the largest absolute value. A typical way to find the largest of multiple values would be through a *compare* instruction which compares two values and sets a flag according to the result. This approach becomes problematic when the number of values to compare becomes high, since a very large amount of conditional branches would be required. It is also difficult to take advantage of a SIMD architecture when this approach is followed, since only one comparison can be computed at a time. This would lead to slow program execution and a large program size due to the branches, and should therefore be avoided.

To utilize the SIMD architecture for this function, it would be more suitable to give the result of a comparison as the address of the largest value (i.e. the values placement in the memory bank), and store this value in the datapath. These addresses can then be used to calculate the address of the symbol value which the input signal was most correlated to. The use of branches would then be avoided, and multiple values may be compared simultaneously.

4.1.5 Shift operations

Shift operations are useful for a wide variety of microprocessor and DSP tasks. In the domain of baseband processing, shift operations are especially useful for scaling (division by 2^N), quantization and subregister packing/unpacking. Two options have been considered for implementation of shift operations;

- Multipliers could be used by multiplying/dividing the data to be shifted by a proper constant
- Dedicated barrel shifters could be included in the architecture

The complexity of the barrel shifter will basically be $O(n^2)$ where n is the word-size of the data samples. For the word sizes the DSP is operating on, the required area for these units will be quite small. The use of multipliers will greatly increase the power consumption during these operations due to a larger logical depth and additional memory accesses.

4.1.6 Linear/log conversion

The IEEE 802.15.4 standard specifies how RSSI values shall be represented to the MAC layer. The RSSI value shall be given as an 8 bit word, but with a higher dynamic range than what is feasible within 8 bits. Obviously, some kind of linear to logarithmic conversion is needed to satisfy these requirements. Two options where considered; the use of an iterative logarithm for the computation or the use of table look-ups.

In [21] an iterative algorithm for such computations is represented. It was found that a 16 bit barrel shifter would be needed in order to convert a 16 bit value to a smaller logarithmic representation. Barrel shifters tend to get very expensive in terms of area when the word sizes are large. Considering that the log/lin conversion will only need to be computed once for each packet that is received, a 16 bit barrel shifter would probably cause a too high system cost for this DSP.

Instead, it should be possible to perform the conversion by the use of table look-ups. The required resolution for the RSSI value is defined as 4dB over a range of 40dB [18]. The number of coefficients to store for the table look-ups will therefore be 10, which should be considered as a much lower system cost than the barrel shifter.

5 Architecture

In this chapter, the proposed DSP architecture is presented. The chapter will be divided in three parts, by describing the memory structure, the datapath and the control path separately.

5.1 Memory structure

The proposed memory architecture contains two memory blocks, a ROM for storage of instructions and coefficient and a RAM for storage of data. It is stated in [11] that a word size of 8 bit should be sufficient for implementation of the 802.15.4 modem. The word size of memories and datapath are therefore set to 8 bits. A SIMD factor of four is employed to obtain a large throughput, the resulting widths of data and coefficient memories should therefore be 32 bits. A block diagram of the proposed memory structure is shown in figure 15.

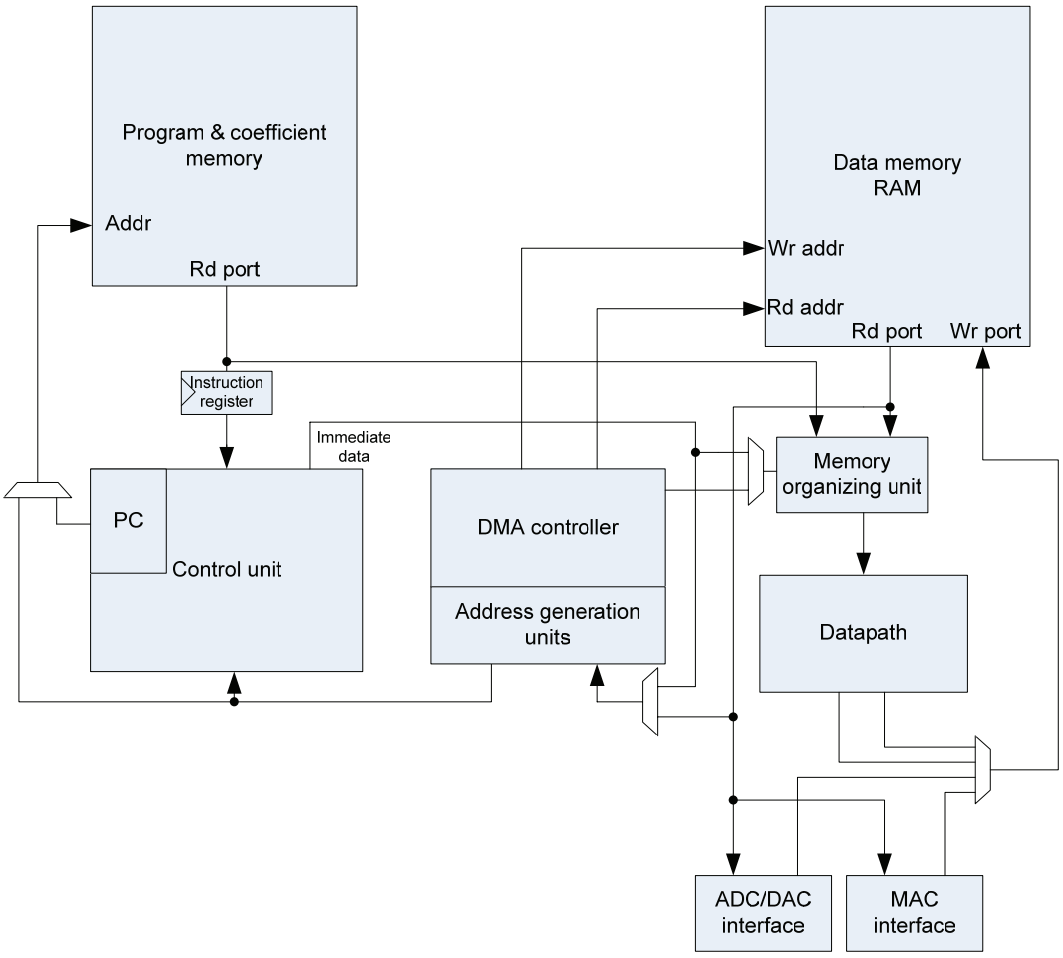

Figure 15 - Memory architecture

For the most computationally intensive operations to be implemented in the core, such as convolution, correlation and CORDIC, a register file would at a large extend only be used

as a temporary buffer for memory read from and to be written to the main memory. Considering this, it was decided to equate the register file with the data memory to reduce the area of the memory structure. The required amount of memory to implement the low-throughput radio standards the proposed DSP core is aimed at can be considered quite small. The cycle time of these small on-chip memories should therefore be considered low enough to allow efficient processing. The use of caches was also considered unnecessary considering the types of memories used.

For volatile data storage, four possible solutions were considered:

- **Single ported RAM**: This use one shared port for both read and write operations. Consequently only one read or write operation is possible each cycle
- **2-port RAM:** Two dedicated ports for read and write operations.
- **Dual ported RAM:** Two shared ports for both read and write operations.
- **3-port RAM**: Two ports for read accesses and one for write operations.

It was found that during most of the operations, the memory required at the inputs of the datapath will consist of a set of constant coefficients and a set of data samples. Therefore, it should in most cases be sufficient to equip the DSP with a dual or 2 port RAM in combination with a coefficient ROM to obtain a single cycle throughput. The area and power savings obtained by this configuration can be considered quite large since a 3-port RAM consists of two 2-port RAM blocks containing the same data. This configuration would double the area and the energy needed to write to memory compared to a 2-port solution. The use of a single ported data memory was considered disadvantageous in spite of a much lower area than the other solutions. When no register file is used, the obtainable throughput would be significantly lower for this solution. Also, the energy efficiency is generally lower for such memory blocks [19]. The use of 2-port memory was preferred above a dual port solution considering gains in form of lower area and higher power efficiency.

Two options was considered for coefficient storage; the use of a separate ROM block or equating the coefficient memory with the program memory. A separate ROM block would produce a large memory overhead since two small memories will have a significantly larger area than one larger memory [5]. Equating coefficient and program memory will on the other hand result in a lower memory bandwidth since both the instruction and coefficients would need to be supplied from one memory port. However, the improved area efficiency was evaluated to outweigh the small sacrifice at execution speed and was chosen as the best solution for this DSP. To maintain a high memory bandwidth and minimize memory accesses for the most demanding operations, the use of a *single* and *dual instructions repeat loop* is proposed. The hardware loop is further described in section 5.3.4, and will allow ROM access to be handed over by the datapath during loop execution. Also, issue of one or four words of immediate data in non-looped instructions is supported to allow coefficient access for such operations.

The considered trade-offs for different memory architectures is summarized in table 5.

Table 5 - Memory architecture trade offs

Storage type	Memory configuration	Advantages / Disadvantages
Data storage	Single ported RAM (1RW)	+ Low area requirements - Low throughput - Low power efficiency
	2 port RAM (1R-1W)	+ Highest power efficiency + Single cycle single read - single write - High area requirements
	Dual ported RAM (2RW)	+ Single cycle two read, two write or one read and one write +/- Medium power efficiency - Higher area requirements
	3 port RAM (2R-1W)	+ Single cycle dual read - single write - Low power efficiency - Highest area requirements
Coefficient storage	One ROM for both instructions and coefficients	+ Low area requirements + Single instruction repeat looping dramatically reduces power consumption due to memory accesses + Requires two memory busses + The use of a single ROM simplifies programming - Lower throughput - Requires a large instruction width when equating to 4 word SIMD coefficient memory
	A separate ROM for coefficient storage	+ Higher throughput + Possibly enabling a smaller instruction width + Reduced power due to smaller memories - High area requirements - Require three memory busses

5.1.1 Memory organizing unit

The memory organizing unit is included to efficiently provide correct data from the memory ports to the datapath inputs. The structure of this unit is shown in figure 16.

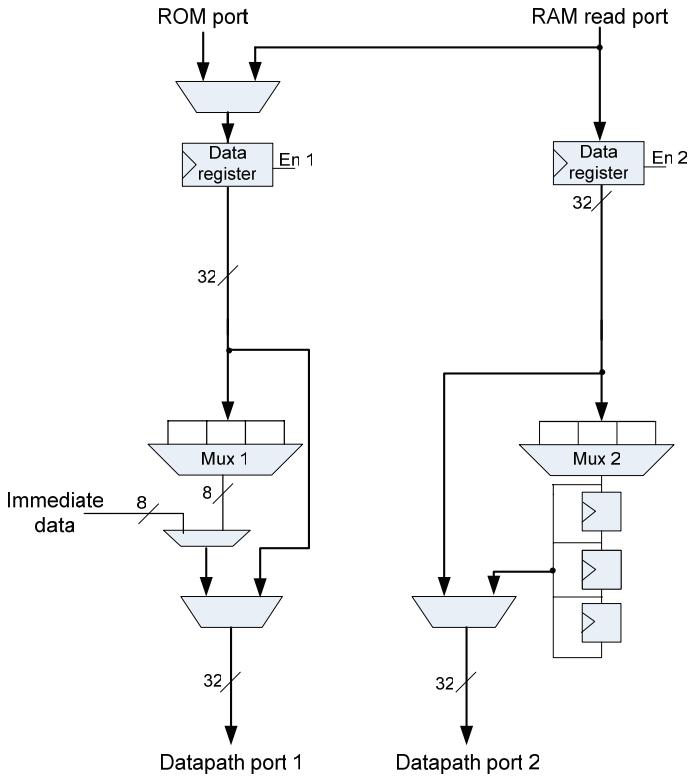

Figure 16 - Memory organizing unit

The data registers are used to store four words given on the RAM or ROM port when a memory request have been given by the DSP. Datapath port 1 will either supply one multiplexed word (the same word repeated four times on the data bus) or four parallel words. The data is given by four words at the ROM port, the RAM port or one word of delayed immediate data given explicitly in the instruction. Datapath 2 will either supply the rightmost four words of an eight word shift register (see section 4.1) or four parallel words. The data is given as a value from the RAM port stored in the data register.

5.1.2 Address generation units

The proposed architecture contains 2 AGUs for the RAM block, one for each memory port. Additionally, 2 AGUs are included for the ROM block, one for coefficients and one for program memory (the program counter).

To avoid insertion of extra delay to the memory read cycle, the read addresses will always be post-modified. I.e. the addresses modified by the currently executed instruction will be available for the next instruction. The write address can be modified by the instruction which makes the access request for the read port. Figure 17 shows a sequence diagram for the pipeline, illustrating the timing of the memory accesses and AGU modifications.

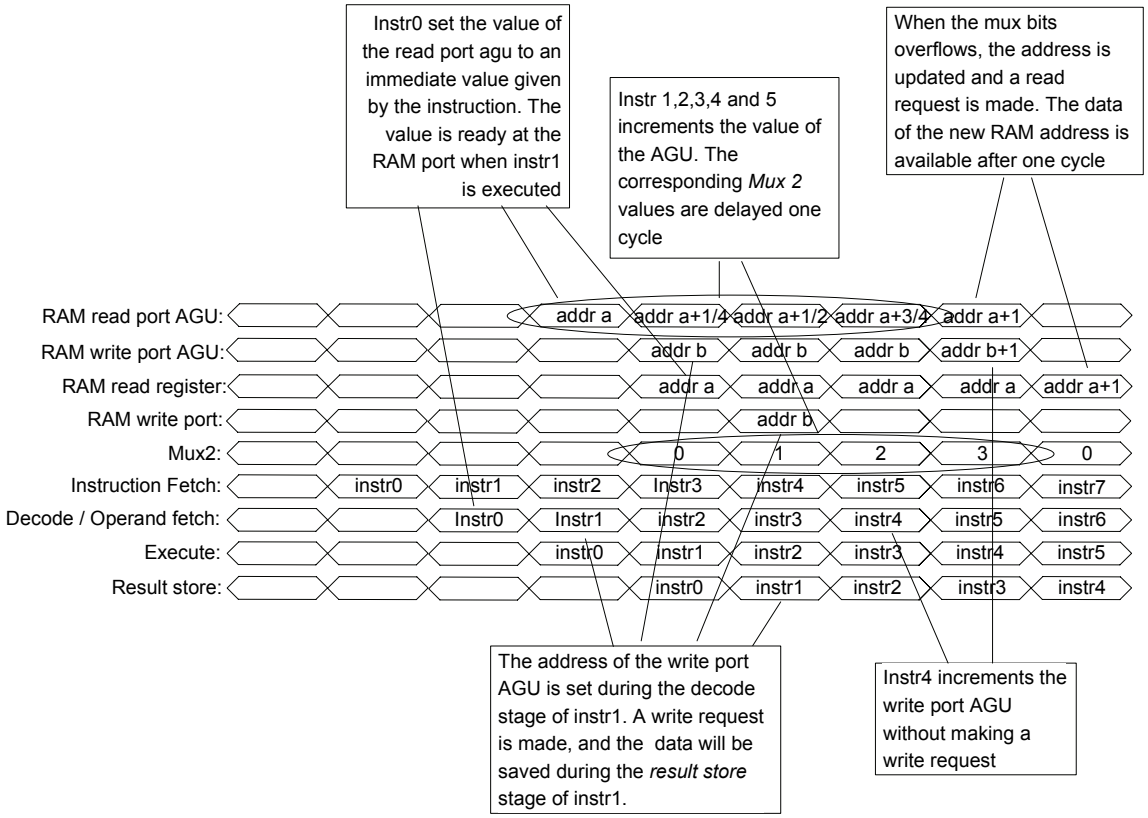

Figure 17 - Timing diagram of memory generation

5.1.2.1 RAM read port AGU

A block diagram of the AGU is shown in figure 18.

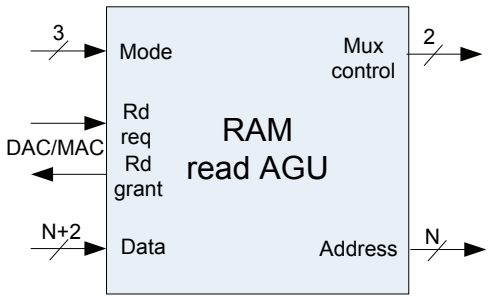

Figure 18 - RAM read port AGU

The AGU will contain two address registers; one for the data to be accessed by the datapath, and one for the MAC/DAC interfaces. The datapath address register will be 2 bit larger than the accessible memory size (N); the two lsb's of the register will be used as a control signal for *Mux 2* of figure 16. This control signal is delayed by one cycle, since the execution of the operands takes place during the pipeline stage following the operand fetch. The MAC/DAC address register is of size N, the multiplexer-control bits are not needed for this register.

A 5 bit increment register is included in the for the DSP address register. The 2 lsb's of the register determines the value to increment the multiplexer control bits of the address when the *mode* input selects an increment. The remaining bits of the register determine the increment of the address when the multiplexer-control bits overflow. This is useful e.g. when I and Q samples are processed separately, by enabling the AGU to skip some of the addresses. If the two lsb's of the increment register is set to zero, the address will always be incremented.

The MAC/DAC address register will be used as output and incremented by one if the *Rd req* bit is set high and the DSP address is not modified. When the request has been granted, the *Rd grant* bit is set high.

The AGU supports a circular addressing mode for both address registers. The registers R_{BS_DP} and $R_{BS\_MAC/ADC}$ are used to hold the size of the circular buffers. The registers, R_{BS_DP} and $R_{BS\_MAC/ADC}$, will store the length of the circular buffer; this can have any value up to 128. The start of the buffer will have an N-word boundary, where N is the smallest power of 2 that is greater than or equal to the buffer size. When an address exceeding the limits of the buffer is calculated, the address will wrap around to the start of the buffer. The multiplexer bits are not included in the buffer size for these registers.

The different modes available for the AGU are summarized in table 6.

Table 6- AGU modes of operation

Mode	Function
Load R_{addr_DP}	Loads the datapath address register with the value given at the *Data* input.
Load R_{inc_DP} and R_{BS_DP}	Loads the datapath increment and buffer-size registers with the value given at the *Data* input.
Load $R_{addr\_MAC/DAC}$ and $R_{BS\_MAC/DAC}$	Loads the MAC/DAC address and buffer-size registers with the value given at the *Data* input.
Add R_{addr_DP}	Adds the value of the datapath address register with the value given at the *Data* input. If the value is negative, a subtraction is performed.
Increment DP	Increment the value of the datapath address register. When the multiplexer-control output generates carry, both increment values are added to the address. If the multiplexer-control field of the increment register is set to zero, the remaining bits of the increment register are always added to the address.
No operation	All registers are left unchanged.

5.1.2.2 ROM coefficient AGU

The ROM coefficient AGU will work in a similar way as the RAM read port AGU. The calculated address will optionally connect to ROM and *Mux 1* of figure 16 during the execution of a HW-loop. Optionally, this AGU will work as a secondary RAM read port

AGU when operations requiring two operands from RAM are executed. To implement circular buffers of coefficients, modulo addressing will be supported for this AGU. In contrast to the RAM read port AGU, the multiplexer fields of the address is included in the specified buffer size to obtain the necessary precision of the circular buffer.

This AGU will be used as a secondary RAM read port AGU if an instruction requires access of two operands from RAM simultaneously.

The following modes will be available for the AGU:

- Load R_{addr}
- Load R_{inc} and R_{BS}
- Increment
- Add R_{addr}
- No operation

5.1.2.3 RAM write port AGU

This AGU will calculate the read address of the RAM. One N+2 sized register is used to store the address used by the datapath, the two lsb's of this register is used as a control signal for the accumulators. A second N bit register holds a write address for the ADC/MAC interface. Circular addressing is supported for the ADC/MAC address register, this mode will be identical as the circular addressing mode for the RAM read port AGU. DMA control of the two addresses is supported in the same manner as for the read port AGU.

The following modes are supported for the AGU:

- Load R_{addr_DP}
- Load $R_{addr\_MAC/ADC}$ and $R_{BS\_MAC/ADC}$
- Increment by 1 or 1/4
- No operation

5.1.3 DMA control

The ADC, DAC and MAC interfaces will occasionally need access to data from the DSP memory. The use of interrupts to handle such routines would lead to expenses in terms of increased complexity and decreased throughput. Also, the execution time would become unpredictable, leading to added complexity of the hardware loops. Instead, it was decided to provide the external interfaces with direct memory access (DMA). The DMA control is handled by the AGU's and a very simple DMA controller. The only purpose of the DMA controller is to issue a dual cycle read request from RAM if the primary and secondary RAM read port address registers changes values during the same cycle.

The different units are given the following priorities when a memory access request is made:

Priority 1: To prevent delays and unpredictable execution-time for the DSP core, a request from this unit will always be granted. If a request is made simultaneously from the primary and secondary RAM read port AGU's, the primary address will be loaded during the first cycle, followed by loading of the secondary address.

Priority 2: The ADC and DAC interfaces will have second priority. Since the ADC and DAC sampling frequency will be in an order much lower than the DSP frequency, the latency requirements for such memory accesses should not be very strict. E.g. if the data converters are sampled at 8 MHz, and the DSP at 80 MHz, a data converter memory request must be granted within 10 cycles. By employing some data buffering at the interfaces, these requirements could be lowered considerably. However, care must be taken when the DSP is programmed to avoid too many sequential memory reads or writes.

Priority 3: It is assumed sufficient for the MAC interface to communicate with the DSP when a packet has been fully received or before a packet is sent. It should thereby not be a conflict between these memory accesses and DSP and DAC/ADC accesses. The MAC interface is therefore given lowest priority.

5.2 Datapath

The datapath architecture of the DSP is shown in figure 19.

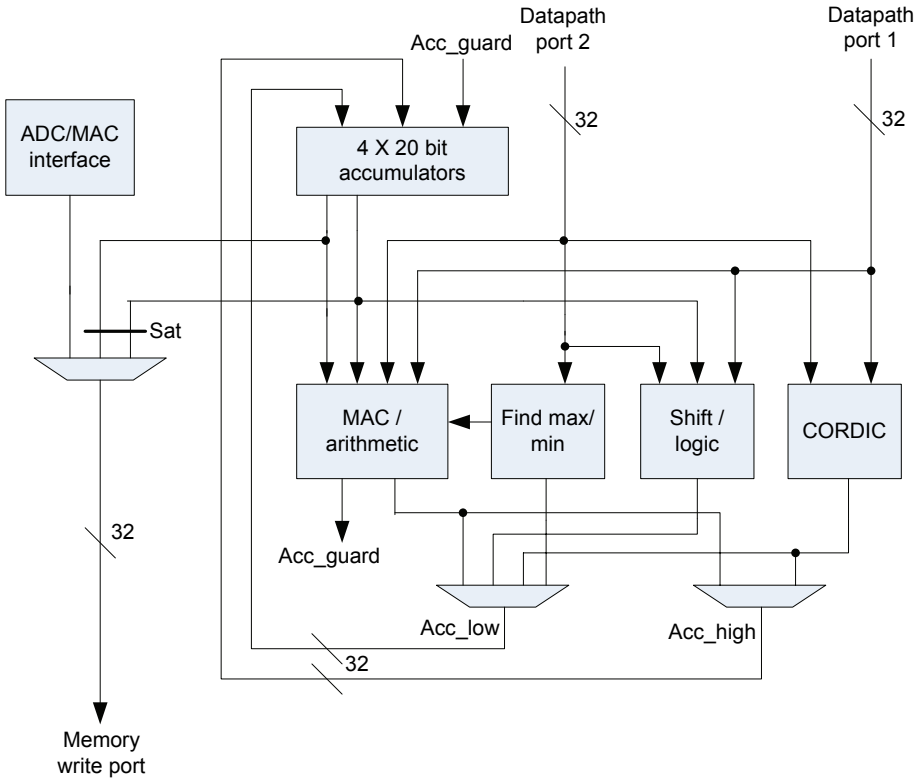

Figure 19- Datapath architecture

Four execution units are included in the datapath of the DSP. All results from an operation performed by any of the execution units will be stored in the accumulators. Depending on the operation, the operands will either be accessed from the accumulator or from the main memory through the datapath ports.

5.2.1 The pipeline

To minimize the logical depth and improve the performance of the DSP, execution will normally be performed over four pipeline stages. Table 7 summarizes the pipeline stages and the operations performed in each of them.

Table 7 - Pipeline stages

Pipeline stage	Operations
Instruction fetch	The instruction is fetched from program memory and loaded to the instruction register.
Decode / operand fetch	The instruction is decoded and the data currently pointed to by the AGU is fetched from RAM. The address pointer is optionally modified.
Execute	The execution stage of the datapath. The actual execution of an instruction is performed by the datapath during this stage. The result of the computations is always stored in the accumulators.
Result store	If specified in the instruction, the current values of the accumulator will be stored in the memory location given by the RAM write AGU.

It was found that a variable pipeline depth would increase the complexity of the DSP since additional hardware would have to be implemented to monitor the execution and prevent pipeline conflicts. Also, throughput may decrease since no-operation instructions would have to be inserted when pipeline conflicts appear. It was therefore focused on obtaining a fixed pipeline length.

From [10] it was found that a multiplier operating on 8 bits and a 20 bits adder would have a quite small critical path considering the small word sizes. When chaining low-area implementations of these units, the corresponding clock frequency would be about 200MHz in a 0.13 micron process. The access time of low-power memory modules will be comparable to these delays [5], making pipelining of the MAC unit unnecessary. The critical path delay of the *Find max/min* unit will be about the same as for the MAC unit, and should not have to be pipelined. The barrel shifters and logical blocks of the *shift and logic* unit will have an even smaller logical depth. The CORDIC unit however, was found too slow to perform non-pipelined execution. If a low area should be obtainable, it was found necessary to pipeline this unit in two stages to align with the critical path of the other units. To avoid added complexity to the DSP control path, such complications can be solved by delaying the write operations one cycle when CORDIC instructions are issued. Five instructions must be given to execute four CORDIC operations.

5.2.2 The accumulators

The accumulators are large registers used as temporary storage of data processed by the execution units. The data resulting from an operation will always be saved in the accumulators at the end of the execution stage(s) of the pipeline. Figure 20 shows how the accumulators are realized.

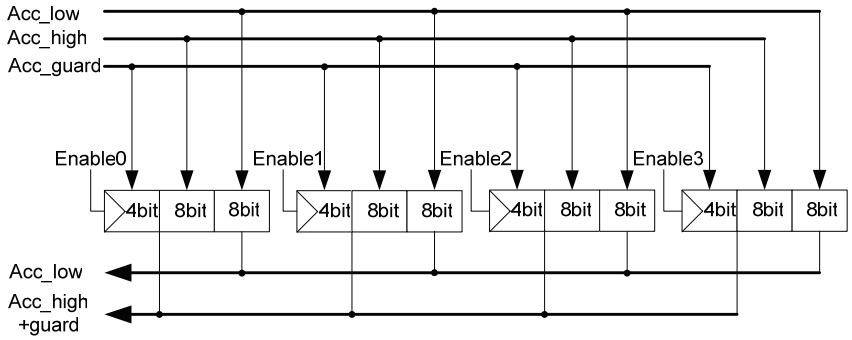

Figure 20 - The accumulators

Each of the four accumulators is 20 bits wide, divided in the *Acc_high* and *Acc_low* registers of 8 bit and the 4 bit *Acc_guard* register. The *Enable* control signals are used to specify whether the accumulators shall be updated or not. These signals will be controlled by the two lsb's of the RAM read port AGU in combination with a field in the instruction which specifies if the accumulators shall be updated or not.

5.2.2.1 Data storage and saturation

Except from the ADC and MAC interfaces, only data stored in the accumulators can be stored directly in the main memory. Depending on the instruction, the 32 bits of either the *Acc_low* or *Acc_high* registers are stored. The values of the *Acc_high* registers can optionally be saturated; the *Acc_guard* registers are then used to determine if an overflow or underflow has occurred. The data at the RAM read port is after saturation set according to table 8.

Table 8 - Saturation dependencies

Register to store	Guard bits	Dependencies	Output
Acc_high	*Acc_guard[0:3]*	If (*Acc_guard[3]=1* and (*Acc_guard[0:2]!=111* or *Acc_high[7]!=1))*	Underflow: 0x80
		If (*Acc_guard[3]=0* and (*Acc_guard[0:2]!=000* or *Acc_high[7]!=0))*	Overflow: 0X7F
		Else	*Acc_high*

5.2.3 The MAC and arithmetic unit

The execution of 2 and 8 bit MAC operations and 8, 16 and 20 bit arithmetic operations are equated in the same execution unit to save area. The unit will employ a SIMD factor of 4 or 16 by parallelizing the execution units. Each of the four MAC and arithmetic units will operate on one word from each of the datapath ports and a 20 bit value from one of the accumulators. The structure of each of the parallel MAC units will be as shown in figure 21.

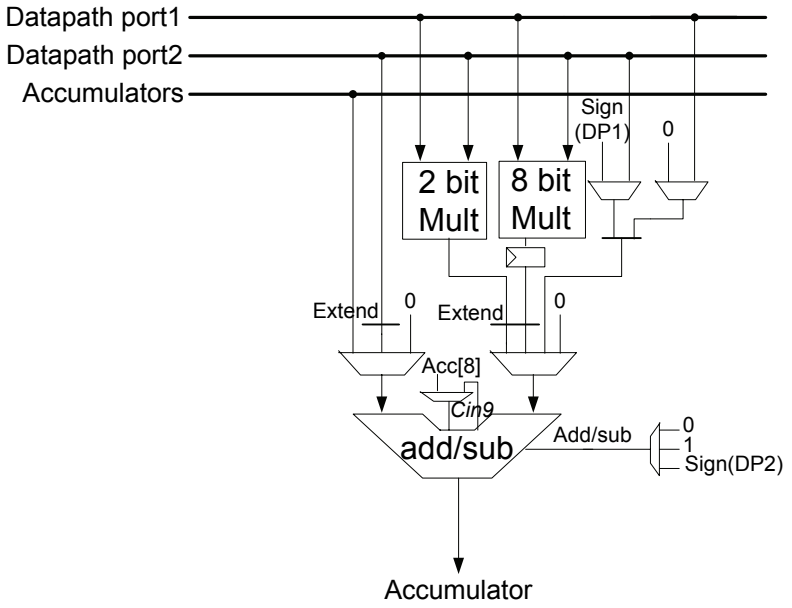

Figure 21 - The MAC and arithmetic unit

5.2.3.1 The multipliers

Each of the multipliers will either operate on two 8 bit values or eight 2 bit values, employing a SIMD factor of 4 or 16 for the MAC operations.

8 bit multiplications

For 8 bit operations, the 16 bit output of each multiplier will be:

$$DP1_i \times DP2_i$$

DP1 denote datapath port1, DP2 denote datapath port2. *i* denote the *i*'th 8 bit word available at the corresponding port.

2 bit multiplications

To minimize the required complexity, the 2 bit multiplications are aligned with each of the four MAC units as shown in figure 22.

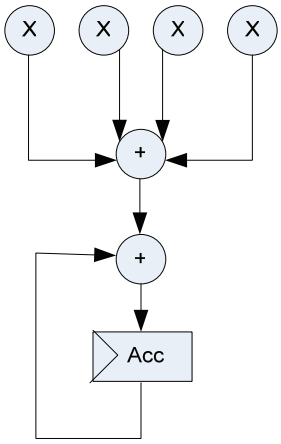

Figure 22 - Integration of 2 bit multipliers

By the use of this scheme, no additional modifications of the memory organizing unit was necessary to realize a SIMD factor of 16 operating on 2 bit numbers. However, the use of four coefficient sets is required due to unalignment within the 8 bit memory boundaries when 2 bit MAC operations are executed.

The resulting 4 bit output value of a 2 bit MAC operation will be as follows for each of the four MAC units:

$$DP1_{i1} \times DP2_{i1} + DP1_{i2} \times DP2_{i2} + DP1_{i3} \times DP2_{i3} + DP1_{i4} \times DP2_{i4}$$

DP1 denote datapath port1, DP2 denote datapath port2. *iN* denote the n'th 2 bit word available at the corresponding 8 bit word.

This concept can easily be extended to also apply for 1 bit MAC operations. The hardware cost of both 1 and 2 bit MAC units can be considered very low since all operations can be performed as additions or simple logic operations.

5.2.3.2 The adders / subtractors

Each of the adders / subtractors is 20 bits wide. They can operate on 20 bit values supplied from the accumulators, 16 bit values from the 8 bit multipliers, 8 bit values from the memory ports and 4 bit values from the 2 bit multipliers. Additionally, the values of both datapath ports can be assembled as one 16 bit value at one of the adder inputs. To provide a large dynamic range for certain additions, the word at datapath port 1 can be given at the lower 8 bits of the adder, allowing additions by the value of the accumulator. All inputs will be sign extended to 20 bits by inserting zeroes or ones (depending on the sign) for the upper bits at the input of the adders. The lower 8 or 12 bits are generally set to zero for 4 and 8 bit additions.

To perform *rounding* of the accumulators, a control signal is used to select the 8'th bit of the accumulator as a carry bit for the upper 12 bit of the adder (*Cin9*) while the

accumulator is added by zero. If this bit is *one*, the value of *Acc_high* will be incremented.

The operational mode of the adders (add/sub) is either given by the instruction or the sign of the sample provided by datapath port 2. The latter is used for computation of absolute value by subtracting the input value from zero if the input is negative.

5.2.4 The shift and logic unit

A block diagram of the shift and logic unit is shown in figure 23. It consists of four 8 bit barrel shifters and four logic blocks. The units will operate in a SIMD fashion – all barrel shifters and logic blocks will always perform the same operation.

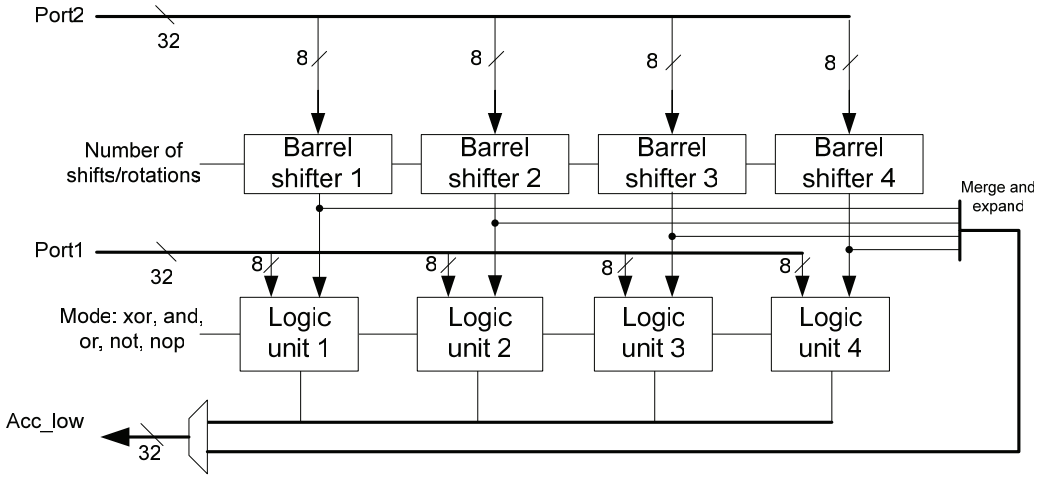

Figure 23 - Shift and logic unit

The barrel shifters will shift or rotate the input samples by a number specified by a field in the instruction. The shift direction is given by the sign of this field. For unshifted logic operations, the barrel shifters are set to shift by zero. The barrel shifters will operate on the data from RAM given at datapath port 2.

The logic blocks will operate in one of five modes, depending on the instruction; xor, and, or, not or nop. The nop mode will pass the signal unchanged to the output. The logic blocks will operate on data given at the outputs of the barrel shifters and the current data of datapath port 1.

The shift and logic unit will also be able to reduce the precision of the values of the barrel shifter inputs to 2 bits, and pack these values to an 8 bit word. By only enabling one of the accumulators, only the packed word will be stored.

5.2.5 The CORDIC unit

A high throughput for CORDIC vectoring and rotation is essential to obtain a high performance of the DSP. The CORDIC unit is therefore based on a word parallel implementation. Figure 24 shows a block representation of the CORDIC unit.

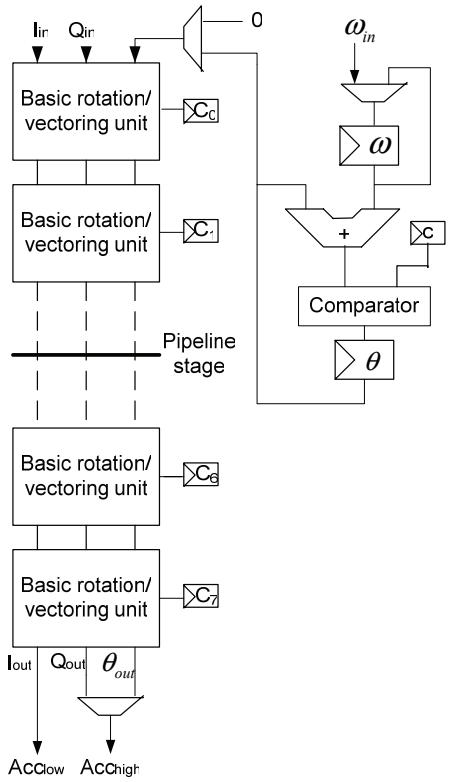

Figure 24 - CORDIC architecture

The CORDIC unit will consist of 8 sequential *basic rotation / vectoring units* (see section 4.1.3), each of them will use dedicated phase constants to avoid table look-ups. To avoid extra cycles due to scaling of the rotation angle, a dedicated adder is employed for these operations. Each time a CORDIC rotation is performed, the rotation angle (θ) will be incremented by the phase velocity (ω). The phase velocity is computed during frequency estimation and saved in the ω register. When the angle reaches 2π, the comparator will make the θ register overflow. To align the critical path of the CORDIC to the cycle delay of the memories, the CORDIC unit is pipelined in two stages.

To avoid making the two-stage execution of CORDIC operations visible during other operations by the DSP, pipeline control will have to be handled by software. E.g. when four complex values are rotated by the CORDIC unit, this can be performed by five instructions. The first four instructions can then specify the desired read operations, while the four last instructions specify the write operations.

Since the CORDIC unit will need both I and Q values to perform a rotation, there will be needed an extra cycle each time new values are loaded from memory (assuming I and Q values are stored in separate memory locations). This will align well with the extra cycle

needed for pipeline control. The three following rotations will have a single cycle throughput since the input data will already be available at the inputs. The high and low values of the accumulators are used to store the resulting I and Q values. This is performed to keep these branches in separate memory locations when the CORDIC operation is performed. The *enable_acc* bits of the write port AGU is used to select in which of the accumulators to store the values.

The CORDIC unit will not utilize a SIMD structure, due to the high cost of this unit in terms of area.

5.2.6 Find max/min unit

To minimize program size and computational overhead for finding the maximum value of a vector, it was found necessary to optimize for such computations. This unit will find the largest value of the four words contained in a RAM location and one word contained in an accumulator. Figure 25 shows how this can be solved.

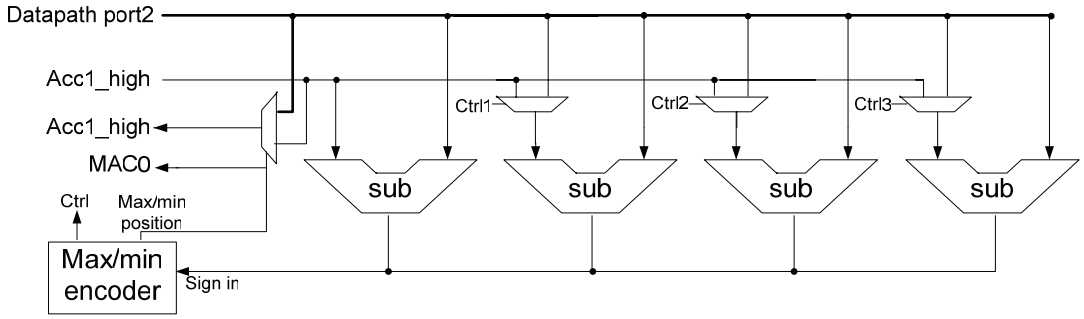

Figure 25 - Find max/min unit

First, the leftmost word in the RAM location will be subtracted from the value currently stored in an accumulator register. The resulting sign of this subtraction will be used by the *Max/min encoder* to select the largest value for the next subtraction. Subsequently, the resulting signs of the two first subtractions will be used to determine the input for the third subtraction and so on. After four subtractions, the max/min encoder is able to determine the largest input value and store this in the accumulator register. The position of this word in the input vector will be added to the current value of an accumulator register. The critical path of this unit will be quite large - four 8bit subtractions and some delay for decoding and multiplexing. The updating of the position in the accumulator should therefore be pipelined by inserting this value to the pipeline register of the corresponding MAC unit. Note that the absolute value of the words contained in the evaluated vector must be found before these calculations can take place.

5.2.7 The status register

A status register is a set of flags needed to make the DSP able to perform conditional testing and branching. The status register will consist of the following flags:

- **Z1:** Zero flag. Set if the lowest 16 bits of accumulator 0 are equal to zero.

- **Z2:** Zero flag. Set if the guard bits of accumulator 0 are equal to zero.
- **N:** Negative flag. Set if the value of accumulator 0 is negative.
- **O:** Overflow flag. Set if an overflow has occurred for accumulator 0.
- **Reserved**

The first four flags are used to evaluate conditions, such as “greater than”, “less than” or “not equal”. Also, the overflow flag can be used to determine if an overflow has occurred for an 8 bit or 16 bit arithmetic operation. These flags are set based on the content of accumulator 0, i.e. the operations to base the branches on must be calculated by the leftmost execution unit of a SIMD unit.

A set of flags are reserved for future enhancements, such as for communication with function level accelerators.

5.2.8 The ADC/DAC interface

This unit supplies an interface between the DSP core and the data converters. During demodulation, the main purpose of this unit is to arrange the incoming samples suitable for the channel filter algorithm before storing the samples in memory. The unit will also supply some data buffering to reduce the delay requirements for the DMA accesses. Figure 26 shows a block diagram of the ADC/DAC interface.

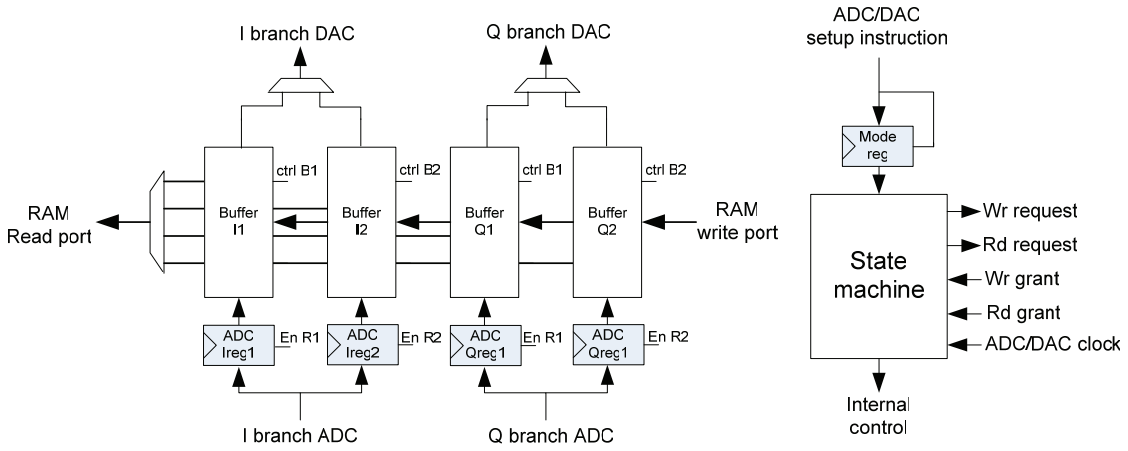

Figure 26 - ADC/DAC interface architecture

The buffers are used to store and organize samples received from the ADC’s and to buffer up samples to be converted by the DAC’s. Each of the buffers will have a depth of 4 samples and is able to receive and dispatch samples in both parallel and serial fashion. The purpose of the state machine is to supply correct control signals and read/write requests based on a mode specified by a setup instruction and grants from the AGU’s. This unit will basically consist of a small adder and some decoding logic to set the outputs based on inputs and current state.

During demodulation, samples will be received in serial form from the ADC's and sent in parallel form from one of the buffers to the RAM read port. A write request is issued when one of the buffers is full. The corresponding buffer will be selected by the read port multiplexer. The next sample from the ADC register will not be clocked to the buffer until a grant has been received from the DMA controller. Since two buffers are supplied for the I and Q branches respectively, the interface is able to store odd and even numbered samples in different memory locations. This optimization was performed to greatly increase the efficiency of channel filters with downsampled output (see section 4.1.1).

During modulation, samples are received in parallel form from the RAM write port and dispatched serially to the DAC's. I and Q samples will be stored in two buffers each. When a buffer is empty, a read request is issued and the corresponding buffer is enabled for parallel reception. When an access grant has been received, the available data on the RAM read port is clocked in and the register is disabled.

The interface will have different latency requirements for DMA access based on the operating mode. These requirements are summarized in table 9.

Table 9 - DMA latency requirements

Mode	Latency requirements
Demodulation – odd and even samples stored separately	Four write operations within $5\times\frac{f_{DSP}}{f_{ADC}}$ clock cycles
Demodulation – odd and even samples stored together	Two write operations within $7\times\frac{f_{DSP}}{f_{ADC}}$ clock cycles
Modulation	One read operation within $5\times\frac{f_{DSP}}{f_{DAC}}$ clock cycles

5.3 Control path architecture

The control path of the DSP basically consists of a program memory for program storage, an instruction decoder and a control unit for hardware loops and branches. A block diagram showing the control path of the proposed DSP architecture is show in figure 27.

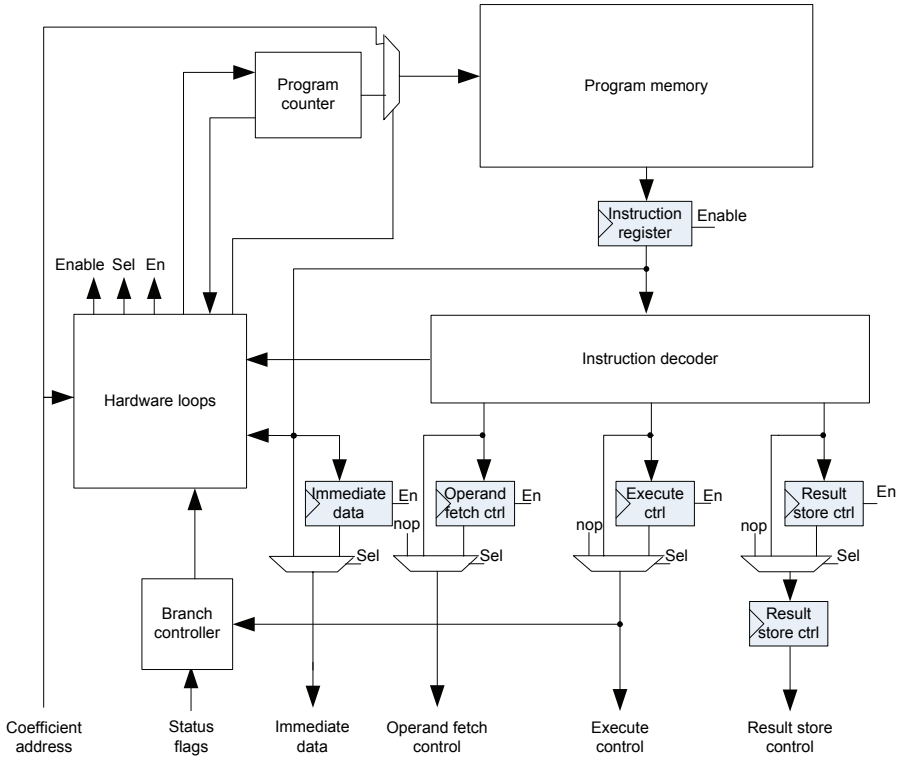

Figure 27 - Control path architecture

The registers shown in the figure are pipeline registers for correct timing of operations performed during the four pipeline stages. During normal execution, the *operand fetch* control signals are given directly from the instruction decoder to be executed during the decode stage. The *execute* control signals are normally delayed one cycle by selecting them from the output of the *Execute ctrl* register. By applying two registers to the path of the *result store* control signals, these will be executed during the *result store* pipeline stage. The multiplexers are used by the hardware loop to control execution of certain loops. This is discussed further in section 5.3.4.

5.3.1 The instruction decoder

The instruction decoder decodes the instruction stored in the instruction register to control signals for the control path, datapath and memory architecture. The implementation of this block is not discussed further in this thesis, but the orthogonal instruction set should make its implementation quite simple.

5.3.2 The program counter

The program counter is used to compute the program memory address. It will be implemented as a simple address register and an incrementer. Each cycle, the address register is either updated by an incrementation of 1 or the value of an address register in the *hardware loop.*

5.3.3 The branch controller

The branch controller is a simple unit which compares a condition given during the *execution stage* of a branch instruction to the current values of the status register. If the condition is met, the branch controller will tell the state machine of the hardware loop to perform a branch. Two instructions following a branch instruction will always be executed, whether the branch is executed or not. This is performed to avoid insertion of no-operation instructions while the dependency is evaluated.

5.3.4 The hardware loops

The purpose of this unit is to handle execution of hardware loops and branches. Figure 28 shows a block representation of how this unit can be implemented.

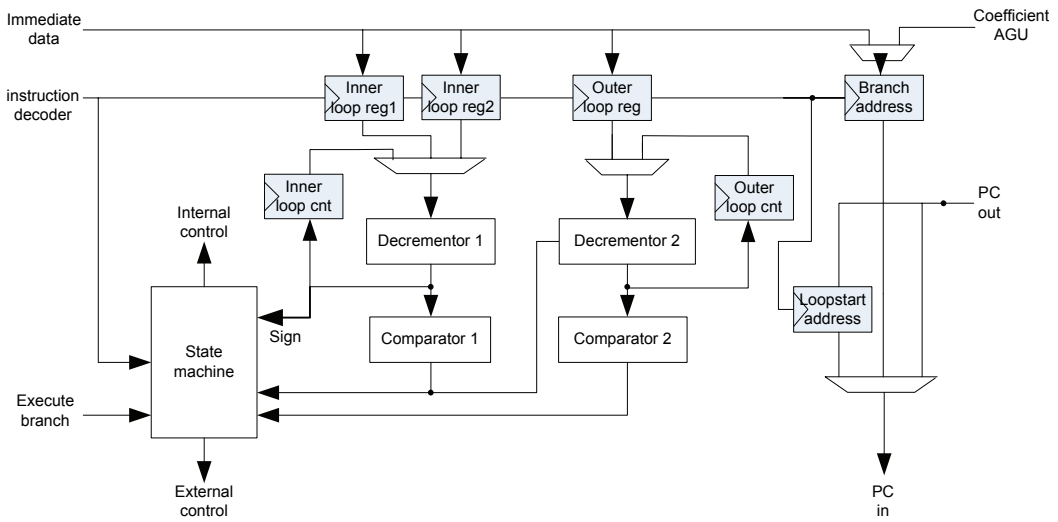

Figure 28 - The hardware loop

As shown in the figure, this unit mainly consists of registers to keep loop values and addresses, decrementors, comparators and a state machine. When a loop instruction is decoded by the instruction decoder, control signals will set the state machine to load a set of mode bits to a register. Based on these bits and the results from the comparators, the state machine will set the necessary internal and external control signals and determine the next state of the loop execution. The different modes the hardware loop is able to operate in is summarized in table 10. How the modes are executed by the state machine is explained further in the following section.

Table 10 - Loop modes

Mode	Description
Idle	No loop is executing. The state machine is to set *PC in* to the address stored in the *Branch address* register if the *Execute branch* bit is set high. If an instruction giving a 32 bit immediate value is executed, a *no-operation* is inserted.

Single/Dual instruction repeat loop	The two instructions following the loop instruction is fetched and decoded once. Then these two instructions are executed by the following loop structure:
	For (i=0; i++; i<(Outer loop reg/2))
	For (j=0; j++; i<Inner loop reg1){
	Execute instruction 1}
	For (j=0; j++; i<Inner loop reg2){
	Execute instruction 2}
	}

Single/Dual instructions repeat loop w/coefficient instr1	Mode set if the first instruction to be executed requires ROM coefficient access. A no-operation instruction will be inserted before the loop is executed.
Single/Dual instructions repeat loop w/coefficient instr2	Mode set if the second instruction to be executed requires ROM coefficient access. A no-operation instruction will be inserted after the loop is executed.
Single/Dual instructions repeat loop w/coefficient both	Mode set if both instructions to be executed require ROM coefficient access. A no-operation instruction will be inserted before and after the loop is executed.
Optimized mode for FIR filtering with down-sampling	By minor modifications, this mode is especially optimized for execution of FIR-filters with downsampled output.

Block of instructions loop	A number of sequential instructions are looped in the following manner:
	For (i=0; i++; i<outer loop reg/2){
	execute instruction 1
	execute instruction 2
	…..
	execute instruction #(inner loop reg)
	}

5.3.4.1 Dual instructions repeat loop

To keep the required amount of memory accesses from the program memory as low as possible, it was essential to describe as much functionality as possible with a minimum of instruction memory accesses. Accessing the program memory may contribute largely to the total power consumption of the DSP. It was found that a ROM configuration of 32 X 256 bits would contribute with a power consumption of over 5mA, assuming that one instruction would have to be fetched every cycle at a clock frequency of 150MHz. When the restrictions given on the total power consumption of the DSP is 5mA, obviously a large effort will have to be done to minimize the amount of instruction fetches.

It was found that many of the most demanding operations the DSP where to perform could be described by two instructions in a nested loop. The control unit of the proposed DSP includes a *Dual instructions repeat loop*, optimized for execution of nested loops containing two instructions. The two instructions are only fetched and decoded once, making the concept very energy efficient when suitable program sequences are executed by the loop. The concept also allows ROM access to be handed over to the datapath during execution of the loop, providing a large memory bandwidth for coefficient fetches.

The hardware loop is able to multiplex between the decoded instruction contained in the pipeline registers and the instruction given at the output of the instruction decoder. This gives a very small area overhead for this implementation, since few additional registers are required for the instruction buffering. However, some multiplexers are needed to multiplex between the two instructions (see figure 27).

The *inner loop cnt* register is used to keep track of how many times one of the instructions has been repeated. The counter will alternate between counting down the values of two inner loop registers, holding the number of repeats for *instruction 1* and *2* respectively. The *outer loop cnt* register will be decremented each time the *inner loop cnt* reaches zero. This is done to provide the state machine with correct timing of the finalization of the loop.

The following explains how the loop will execute.

1. During the *decode* stage of the *loop* instruction, the two inner loop registers and the outer loop register is set according to the values specified by the instruction.
2. *Instruction 2* is fetched; the next address to fetch from ROM can optionally be given by the ROM coefficient AGU. *Instruction 1* is decoded and stored in the pipeline registers. The PC, currently containing the address of *instruction 3*, is stopped.
3. If *instruction* 1 requires access to ROM, a *no operation* instruction is inserted while the coefficients are fetched. In this case, the inner loop counter will not be decremented during this cycle. Otherwise, *instruction 1* will be selected by the multiplexers. The selection of the execution and write operand stages of the instruction will always be delayed by one cycle. *Instruction 2* is decoded and the instruction and pipeline registers are disabled, making the output of the instruction decoder and the pipeline registers stable.

4. *Instruction 1* is executed until the inner loop counter reaches zero. When zero is reached, *instruction 2* is selected by the multiplexers and the number of iterations of *instruction 2* is loaded to the inner loop decrementor. Each time a zero is reached by the inner loop counter, the outer loop counter is decremented.
5. *Instruction 2* is selected to execute until the inner loop counter reaches zero. When zero is reached, the multiplexers select *instruction 1* to execute and the number of iterations of *instruction 1* is loaded to the inner loop counter.
6. When the outer loop counter has reached zero and the value of the inner loop counter is one or zero, the PC is started and the current value of the PC will be fetched during the next cycle. The instruction register is enabled and the program continues its execution. If ROM coefficients are needed by the last instruction to be executed, a *no operation* instruction will be inserted while the next instruction is fetched.

Figure 29 is a sequence diagram illustrating which operations are performed during the execution of a small loop. In this example *instruction 1* is repeated twice, *instruction 2* is executed once before *instruction 1* is repeated two more times. Both instructions need ROM access for coefficients.

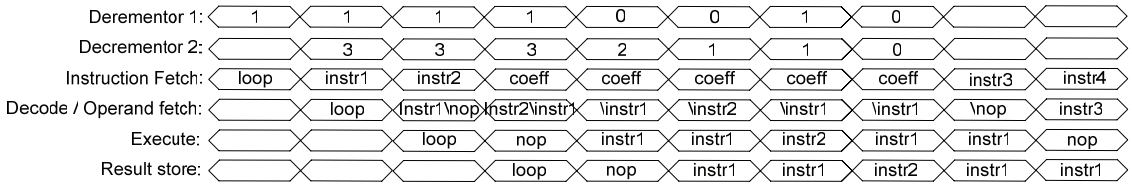

Figure 29 - Timing diagram of loop operations

Note that the outer loop register must be set to a value one higher than the number of inner loops to be executed.

5.3.4.2 Optimized mode for FIR filtering with down-sampling

Since the channel filtering will contribute with a large amount of the total processing of the core, it was found necessary to provide an optimized mode for such computations with downsampled outputs. This mode will basically work as a *dual instruction repeat* loop whereas the first instruction requires coefficient access, but with some modifications. Every second time the inner loop register has counted down for instruction 1, starting at the first countdown, only the operand fetch field of instruction 2 is set to execute. All other fields will be given by instruction 1. The next time this counter reaches zero, instruction 2 will be fully executed. This is useful to be able to control the memory accesses for FIR-filtering when the odd and even samples of the input vector are stored in different memory locations. By this approach, an arbitrary number of such FIR-filter implementations can be executed while only three instruction memory accesses will have to be made.

5.3.4.3 Single instruction repeat loop

Single instruction repeat looping is performed in the same manner as dual instruction repeat looping. By specifying the outer loop register to 1 and the inner loop registers to the number of repeats and zero, a single instruction will be repeated.

5.3.4.4 Block of instructions loop

To increase throughput and power efficiency of loops containing more than two instructions, a general hardware loop for repetition of sequences of multiple instructions should also be implemented. This can be implemented in a similar way as the *dual instruction repeat* loop, by letting the inner loop counter count down a value specifying the number of instructions contained in one iteration of the loop. Since the DSP does not support interrupts and have a predictable execution time, this is possible rather than comparing the PC address to the end address of the loop. The outer loop will count down from the total number of loop iterations.

The following explains how this loop is executed:

1. During the *decode* stage of the *loop* instruction, the *inner loop reg1* and the *outer loop reg* registers are set according to the values specified by the instruction. The current value of the PC is stored in the *loop start* register.
2. One more instruction is executed before the loop begins. This is necessary since it is the address of the next instruction that was stored.
3. The instructions are executed sequentially until the *inner loop cnt* register reaches zero. The value of the outer loop counter is then decremented, if the resulting value is a zero, the loop has finished and the PC will be incremented during the next cycle. Otherwise, the value of the PC is set to the address contained in the *loop start* register, making the loop start over from the beginning.

Figure 30 shows a sequence diagram for the execution of a small loop. A sequence of three instructions is repeated two times in this example.

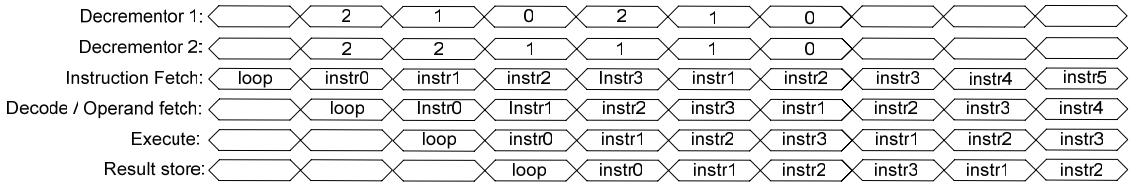

Figure 30 - Timing diagram of block of instructions loop

This concept can also be extended to include support for nested loops. This would require an implementation of stacks to store program addresses and loop count values. The depth of the stacks would give the maximum number of nested loops. However, in most cases a loop depth of one should be sufficient. Since such enhancements would add to the total complexity of the DSP, it is not included in the hardware loop.

6 Programming model

This chapter will give an overview of the programming model of the proposed DSP architecture.

6.1 The instruction set

The proposed instruction set of the DSP will have an instruction width of 32 bits to align with the data vectors to be contained in the same ROM as the instructions. To minimize the required decoding logic and make future adjustments of the instruction set as simple as possible, it was focused on achieving a high degree of orthogonality in the instruction set.

The instructions are arranged in four possible ways as shown in figure 31.

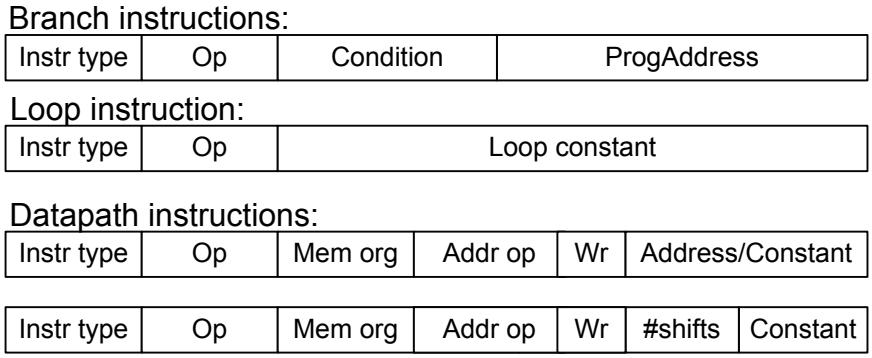

Figure 31 - Instruction set format

Some VLIW features are included in the instruction set to improve orthogonality and decoding complexity. There are used separate operation fields for datapath operations, address manipulation and write operations. Also, operations to be executed in different pipeline stages are stated in different subfields. Control operations and operations performed by each of the execution units are connected to different instruction groups. This is performed to simplify implementation of clock gating and operand stopping, and possibly simplify the instruction decoder. The *instruction type* field is used to specify which group an instruction belongs to. A description of the different subfields is given in table 11.

Table 11 - Instruction subfields

Subfield	#Bits	Description
Instr type	3	Controls a multiplexer which switches between the different instruction types.
Op	5	The operation code states the actual operation of the execution or control unit selected by the *instr type* field. A certain bit specifies whether the operation is pipelined or not.
Mem org	4	This field specifies the operation of the memory organizing unit to supply the desired data at the datapath input ports.
Rd op	5	Specifies the mode of the three AGU's. Depending on the mode, data stated in the *immediate data* field can be used to update or modify one of the AGU registers.
Wr op	3	Specifies whether the high or low values of the accumulators shall be written to RAM and whether the data should be saturated.
Address/Constant	12	This field can either contain an address or modifier for one of the AGU's or one word immediate data to be supplied at Datapath port 1.
Constant	8	Field containing one word of immediate data be supplied at Datapath port 1.
#shifts	4	Specifies the number of shifts to be performed by the barrel shifters. A negative value specifies a left-shift.
Condition	12	Used by conditional branch instructions to specify the condition to be tested.
ProgAddress	12	Specifies the program address to jump to after a *branch* instruction.
LoopConst	24	Specifies the number of iterations to be performed by a hardware loop.

The instruction set could probably have been made smaller. However, a tighter instruction format would give few possibilities for future adjustments of the instruction set. It was also found suitable to align the instruction width with the data bus width to obtain single cycle throughput when the program memory are used for coefficient accesses. The decoding complexity would also increase when shorter instruction words are applied.

6.2 Supported addressing modes

To avoid insertion of additional delay in the memory read cycle, the address registers containing read addresses will always have to be altered by the instruction executing **before** the instruction accessing the operands is executed. Write addresses are modified by the same instruction that makes the write operation.

6.2.1.1 Register addressing

There are only 4 data registers in the DSP; the accumulators. These can be addressed directly of the adders of the MAC unit (e.g. during MAC operations) and the barrel shifters. The MAC units can also access the address registers of the AGUs. This is necessary e.g. to obtain synchronization with the ADC interface and during chip to symbol mapping.

6.2.1.2 Indirect addressing

The addresses of the RAM and ROM are always determined by the values of the AGUs, i.e. indirectly addressed. Which of the address registers that are given priority, is decided by the AGU's and the DMA controller. See section 5.1.2 and 5.1.3 for further details.

6.2.1.3 Immediate addressing

Immediate addressing is supported by setting a *constant* field of the instruction to an 8 or 12 bit value to be accessed by the datapath, the AGU's or the PC. Also, 32 bit data can be given immediate by specifying it in an instruction. The ROM location following the instruction will hold this data.

6.2.1.4 Circular addressing

Circular addressing modes are supported for the AGU's. The start of a circular buffer will have an N word boundary, where N is the smallest power of 2 that is greater than or equal to the buffer size. I.e. if the size of the buffer is 12, the buffer will have to start at address 0, 16, 32, 48 etc. The circular addressing mode will work in a slightly different way for the AGU's. For the RAM AGU's, the buffer will not include the multiplexer fields of the addresses. These fields are included in the buffer size of the ROM coefficient AGU.

7 Estimated complexity

In this chapter, estimates of area, power consumption and required clock frequency for the proposed architecture are presented. Additionally, a description of how an IEEE 802.15.4 demodulator can be implemented by the DSP is provided.

7.1 Implementation of an IEEE 802.15.4 demodulator

The most critical operational mode for the DSP core will be during demodulation of incoming data. A possible signal chain for implementation of the IEEE 802.15.4 demodulator is shown in figure 32. This section will merely describe how the needed algorithms can be implemented by the proposed DSP, for further details regarding the algorithms it is referred to [1].

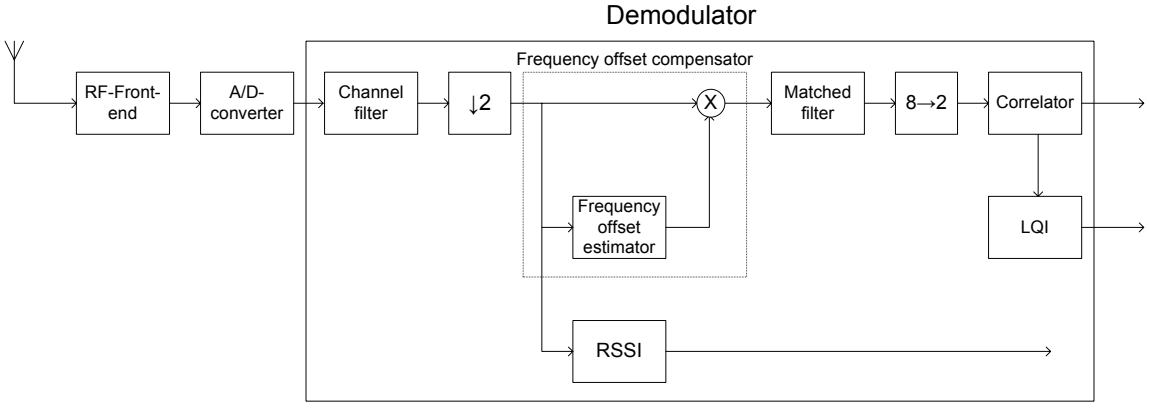

Figure 32 - IEEE802.15.4 receiver chain

The real and imaginary components of the A/D converter output will be processed separately throughout the signal chain (except from the frequency offset compensator). The chain will consist of a channel filter, frequency offset compensator, Received Signal Strength Indicator (RSSI), matched filter, correlator and a Link Quality Indicator (LQI). The following will explain how the different blocks of the signal chain can be computed by the proposed DSP architecture.

7.1.1 Channel filter and downsampling

The channel filter is an 11-tap FIR filter with downsampled outputs. The ADC interface should be set to store odd and even samples of the I and Q branch separately before the channel filter is implemented. This will greatly reduce the amount of cycles for the computation, since the downsampled outputs will not have to be computed. To minimize the amount of memory accesses, the channel filter can be executed by the *dual instructions repeat* HW-loop with shifted input of data samples and circular addressing of coefficients. The optimized mode of the hardware loop for FIR-filtering with downsampled output should be used to obtain maximum power efficiency. This allows

execution of an arbitrary number of FIR-filter implementation while only three instructions will have to be fetched and decoded. The first instruction following the loop instruction will specify MAC operations and increments of the operand address registers. The second instruction will specify a round and save operation with saturation to maximize the precision. The read operation of instruction 2 should specify an addition of the operand address by an immediate value given by one of the instructions.

7.1.2 RSSI

RSSI is computed as an average value of the signal power over 10 symbol periods during a preamble. The power is found by squaring each of the samples included in the calculations. This can be performed by the DSP as parallel MAC operations with the same input signal for both of the multiplier inputs, and should be highly suited for *single instruction repeat* looping. To avoid overflow, the samples will have to be scaled before the RSSI value is computed. This can be performed by the barrel shifters operating in a *single instruction repeat* loop.

The RSSI value must be given in logarithmic form before it is passed on to the MAC-layer. This conversion can be performed as a table look-up by modifying the coefficient AGU based on the computed 16 bit RSSI value. An instruction can then set the program to branch to this address, and the logarithmic value can be given immediate by the following instruction.

7.1.3 Frequency offset compensator

A block diagram of the frequency offset compensator is shown in figure 33.

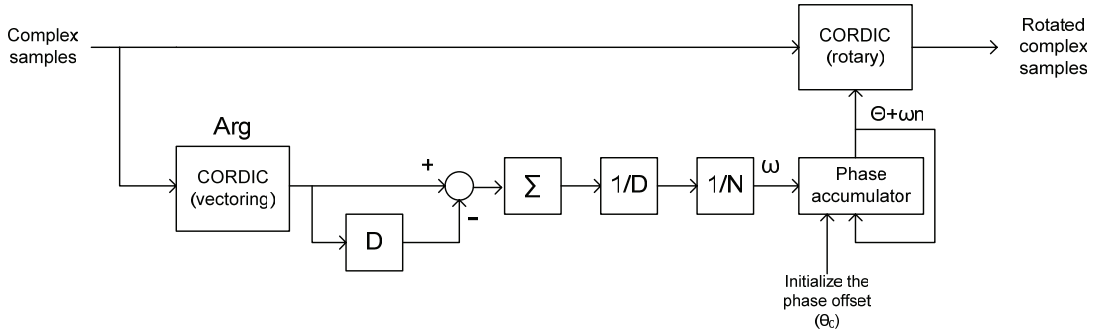

Figure 33 - Frequency offset compensator

First, the frequency offset is calculated by the lower branch of the figure. The obtained estimate is then used to rotate the incoming samples for correction of frequency offset.

7.1.3.1 Frequency offset estimation

The frequency offset estimation is performed by the CORDIC unit operating in vectoring mode during a preamble. It is most efficiently implemented as a *dual instruction repeat* loop where both instructions specify CORDIC vectoring. For every four times one of the

instructions has iterated, the second instruction of the loop is executed once. This scheme can be used to compensate for the necessity of delayed write operation due to pipelining of the CORDIC unit, and will allow for insertions of delays due to dual cycle operand fetches. The loop of these two instructions is repeated a desired number of times, 4 phase angles are found during each loop iteration.

When the desired angles are found, the vector of phase values are subtracted by a delayed version of the same vector. A delay of a factor of 4 is most suitable, due to the SIMD memory structure. The subtractions can be carried out by a *dual instruction repeat* loop by delaying the subtractions every other cycle due to dual cycle operand fetches. The resulting vector is then accumulated and shifted to obtain the average phase value. The resulting phase offset is stored in the phase register of the CORDIC unit.

7.1.4 Frequency offset correction

The frequency offset correction is computed in the same manner as the CORDIC vectoring during frequency offset estimation, but with the CORDIC unit operating in rotation mode. Eight rotated samples are obtained for each five cycles of the hardware loop.

7.1.5 Matched filter

The matched filter is implemented as a FIR-filter of order 4. A desired number of FIR-implementations can be calculated by two *dual instruction repeat* looped instructions. The first instruction will specify the MAC operations and an increment of the read port AGU's. The second instruction may specify a round and save operation and an increment of the write port AGU. Additionally, a subtraction of the data address should be specified by the second instruction to bring the address back to the first input sample of the next FIR-implementation. The coefficients will rotate by the use of the circular addressing mode for the coefficient address.

7.1.6 The correlator

The correlation will have different form during preamble and during reception of the actual packet.

7.1.6.1 SFD detection

To find correct timing of the start of the first symbol of the packet following the preamble, the start of frame delimiter (SFD) at the end of the preamble must be detected.

Quantization
During SFD detection, the incoming data stream will be quantized by packing four subsequent samples into a word with 8 bit boundaries. The quantization can be performed by a *single instruction repeat* loop, quantizing and packing four samples each cycle.

Correlation

During the preamble, correlation is used to determine the start of the first symbol of the packet. Since the start-of-symbol timing is unknown at this point, the detection is implemented as a FIR-filter working on 2 bit samples. The order of the FIR-filter will be 256 when correlation is performed for the two SFD symbols. Four shifted versions of the same coefficient set are correlated to one stream of input data, sixteen 2 bit multiplications are performed in parallel each cycle during this operation. To avoid extra sets of coefficient when the incoming word is unaligned within the four words of the 32 bit memory boundary, the shifted input can be used for the data samples. The *Dual instruction repeat* loop can be used for these calculations in a similar way as for the matched filter.

Each of the computed samples must be compared to a threshold value to determine if a match has been found, this can be performed by a subtraction of a constant and a conditional branch. When a match has been found, the end-address of the detected sequence can be used to calculate the address of the start of the next symbol.

7.1.6.2 Chip to symbol demapping

During chip-to-symbol demapping, the incoming symbol is correlated to multiple possible predefined symbols to find the one it is most likely to represent. At this stage, the timing of the start of each symbol is known; this makes the quantization and correlation take a slightly different form during these calculations.

Quantization

To align the data samples suitable for correlation, every fourth sample should be placed within the same 8 bit memory boundaries. The 8 to 2 bit quantization will require 6 cycles to perform the necessary shifts for quantization and subregister packing of 16 samples. A *block of instructions loop* can be set to handle the packing of 16 samples for each loop iteration.

Correlation

Figure 34 shows a block representation of how the algorithm for finding the most correlated symbol is depicted [1].

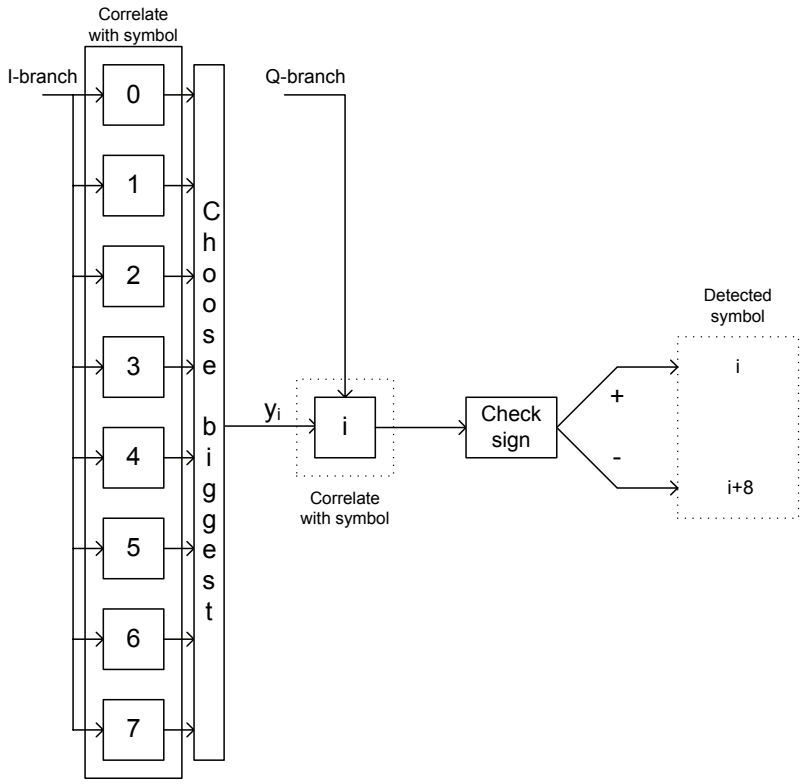

Figure 34 - Chip to symbol correlator

The real values (I branch) of one symbol period will consist of 64 samples outputted from the matched filter. This sample sequence will be correlated with 8 predefined chip sequences representing the I values of each symbol. The correlation can be performed by 2 bit MAC operations with multiplexed and parallel inputs described by a *dual instructions repeat* loop. When all correlations have been performed, the results must be compared to find the one with the highest correlation. This can be performed by first taking the absolute value of each sample before the dedicated Find max/min unit is used to find the largest value. Both these operations can be performed by *single instruction repeat* looping. The corresponding address of the symbol the input stream was most correlated to, will be used to compute the address of the Q values of the symbol. The Q branch of the incoming chip sequence will then be correlated with this value. The data symbol the incoming chip sequence represents is determined by the sign of this value in combination with the symbol found by the I branch.

7.1.7 LQI

The Link quality indicator is a value basically based on the RSSI value and the correlation value. This value should be relatively straightforward to compute.

7.2 Estimation of computational complexity

The estimates of computational complexity of the IEEE 802.15.4 demodulator for the proposed DSP core is summarized in table 12. For a more throughout description of how the estimates are obtained, it is referred to Appendix A.

Table 12 - Computational requirements

Mode	Operation	% of total complexity
Frequency offset correction	Channel filter	63
	RSSI calculation	2
	Frequency offset estimation	22
	Additional control overhead	13
	Minimum clock frequency	**50 MHz**
SFD detection	Channel filter	15
	Frequency offset correction	3
	Matched filter	7
	SFD Correlation	73
	Additional control overhead	2
	Minimum clock frequency	**200 MHz**
Packet reception	Channel filter	41
	Frequency offset correction	7
	Matched filter	19
	Symbol correlation	24
	Additional control overhead	9
	Minimum clock frequency	**75 MHz**

The DSP will operate in three different modes during demodulation; Frequency offset correction, SFD detection and packet reception. The two first mentioned, will take place during the preamble. The computational requirements of each of the blocks of figure 33 and the minimum clock frequency of the DSP during the different modes are summarized in the table. As seen from the table, the requirements during SFD detection will be much higher than during the other modes. The reason for this is the high complexity of the 256-tap FIR filter used for SFD correlation. Even when operating on two bit values with a SIMD factor of 16, the resulting complexity was exhaustive. During the other modes, the computation of the FIR-filter is dominant.

7.3 Estimation of critical path

The critical path of the system will constitute of the 8 bit MAC when the memories are not considered. The critical path of this operation will consist of access of two registers, an 8 bit multiplication, a 20 bit addition and two multiplexers. The resulting critical path is 5,0ns for 0.13 and 8,4ns for a 0.18 micron silicon technology. The resulting maximal clock frequencies are at 200 and 120MHz. These estimates are based on adders and multipliers optimized for lowest possible area, much higher frequencies should be obtainable by optimizing for performance.

The estimates are based on [10], [19] and [20].

7.4 Estimation of current consumption

The current consumption was estimated for the DSP during SFD detection and packet reception of IEEE 802.15.4 demodulation.

The current consumption for the memories and logic is summarized in table 13.

Table 13 - Estimated current consumption

Contribution	SFD detection 0.18um	SFD detection 0.13um	Packet reception 0.18um	Packet reception 0.13um
Logic blocks [mA]	10,8	5,4	7,2	3,6
Memory accesses [mA]	3,3	2,0	1,6	1,0
Total current consumption [mA]	14,1	7,4	8,8	4,6

During SFD detection, a clock frequency of 200MHz and an activity factor of 15% were assumed. During packet reception, it was assumed a clock frequency of 80MHz and an activity factor of 25%. Clock gating and operand stopping are assumed applied for the estimation of activity factor. The activity factor during SFD detection was assumed lower since a very large part of the computations consists of 2 bit multiplications, requiring a very low active gate area. Also, the instruction decoder will have a very low activity during these operations, since the instructions will only have to decoded once each symbol period for the SFD correlation. For further details on the estimation of the amount of memory accesses, it is referred to Appendix B. The estimates are done for both 0.18 and 0.13 micron technologies [20]. A 40% current reduction was assumed for 0.13 micron memories compared to 0.18 micron.

7.5 Area estimation

The total area of the proposed DSP including data and program memory was estimated. The estimated area of the various units of the DSP is summarized in table 14.

Table 14 - Estimated gate area

Part of DSP architecture	Unit	Area
Memory architecture	**256x32 bit RAM**	14500
	512x32 bit ROM	4600
	Memory organizing unit	1264
	Write port multiplexer	200
	Address generation units	1312
Total area: 22066	**DMA controller**	190

Datapath architecture	**MAC and arithmetic unit**	5336
	CORDIC unit	4613
	Accumulators	460
	Find max/min unit	516
	Shift and logic unit	722
Total area: 13233	**ADC/DAC interface**	1736
Control path architecture	**Pipeline registers**	674
	Instruction decoder	1000
	Branch controller	100
Total area: 2902	**Hardware loop**	1128
Total area:		**38351**

For further details on how these estimates are obtained, it is referred to Appendix C.

8 Discussion

8.1 Gate area

The total estimated area was found to be slightly below the 40000 gates given as an area restriction for the DSP.

The area of the memories, MAC and CORDIC unit will contribute by a very large part of the total area. The area of the memories could have been reduced significantly by employing high density memory blocks rather than the low power memories these estimates are based upon. Also, a commercial memory compiler would probably obtain a denser memory than the compiler that was used to obtain these estimates [5]. On the other hand, the area estimation may be a bit optimistic for the logic blocks of the design. Hidden factors such as delay buffers, higher drive strength for certain components and additional logic for clock gating and operand stopping are not included in the estimate. However, a gate count of 40000 should be considered feasible.

The gate count is estimated for low-area implementations of all logical blocks. If a clock frequency higher than the estimated maximal clock frequency is needed to obtain the desired throughput, the resulting logic area will increase.

8.2 Performance

The performance of the proposed DSP architecture has been evaluated for a IEEE 802.15.4 demodulator during its different modes of operation.

8.2.1 SFD detection

The obvious bottleneck considering the computational performance of the DSP is the SFD detection during preamble. The minimal clock frequency during this mode was estimated to 200MHz. For silicon technologies above 0.13 micron, this high frequency will not be obtainable for a low-area implementation of the architecture. A faster design of the functional units and memories will lead to a higher area and power consumption. This should be avoided in order to achieve the given constraints. For a silicon technology of 0.13 micron, the estimated maximal clock frequency will be the same as the minimum frequency when optimizing for low area. A slightly larger area of the MAC unit will be required in order to increase the maximal frequency.

The correlator will contribute by a very large part of the computational complexity during the SFD detection. This operation can either be simplified or further optimized to obtain a

lower minimal clock frequency of the system. The following options have been considered to increase the speed of the correlator:

- By downsampling to 1 bit before the correlation, the calculations can be performed by a SIMD factor of 32. The complexity of the correlator would be decreased by 50 percent, but the accuracy of the detection would also be decreased. Some hardware optimizations would be necessary for the DSP core to make 1 bit correlation possible.
- The SFD detection could be reduced to only check one of the SFD symbols. A reduction of the complexity by 50 percent would be achieved, but the accuracy of the detection would also be reduced.
- A functional level accelerator could be included in the DSP core to speed up the correlation. The hardware cost of such a unit would probably be quite large, and the flexibility would be limited.
- The SIMD factor of the DSP can be made even larger. This would be a very expensive solution, since the memories and datapath units would have to be scaled by the same factor.

If a high accuracy of the detection is required, the best choice will probably be to integrate a hardware accelerator for 2 bit correlation in the core. This would imply very few changes of the DSP architecture, by letting repeated instructions handle the operand fetches of the correlator and state a conditional branch. By letting the accelerated unit set a flag in the status register when a match has been found, the program will branch when this occurs. By specifying the values to correlate with in software, some flexibility can be preserved for this solution.

8.2.2 Frequency offset estimation

The frequency offset estimator implemented for the estimation of computational requirements is too simple for realization of the IEEE 802.15.4 PHY [1]. However, the estimates show that the complexity for computation of the current algorithm is low compared to the other modes that were considered. Based on the available computational resources, it should therefore be possible to raise the complexity considerably during this mode of operation.

8.2.3 Packet reception

The estimates indicate that a clock frequency below 80 MHz should be sufficient for the DSP during packet reception.

8.3 Power consumption

The estimates for power consumption show that the proposed DSP architecture is able to achieve the given constraints for maximal power consumption during its most computational intensive tasks. This requires that a technology of 0.13 micron or below is used for the realization of the processor. For this technology, the current consumption

will be below the constraint of typical current consumption of 5mA during packet reception. Demodulation is dominated by this mode of operation, since the preamble has a duration of only ten symbol periods while the duration of the packet reception will be from 10 to 512 symbol periods depending on the packet size. Both typical and maximum constraints on power consumption should therefore be considered as obtainable. If a hardware accelerator was applied for SFD correlation, the constraints of maximal current consumption should also be achievable for a 0.18 micron process.

From Appendix B it can be seen that a great reduction of instruction fetches have been obtained by the use of *single* and *dual instruction repeat* loops. Nearly all of the most intensive and repetitive operations was found suitable for such looping, resulting in instruction fetches only necessary for between 3 and 21% of the total cycle count. These numbers will depend on the possible amount of buffering achievable for a certain RAM size, resulting in a trade off between a larger RAM and a reduced number of instructions fetches.

The number of memory fetches due to data accesses could have been further reduced if the concept of *solution 2* discussed in section 4.1.1 was fully applied for 2 bit convolution. However, such enhancements would result in a quite large added complexity of the MAC units and memory architecture and was therefore not followed.

8.4 Modulation

Modulation can be considered as a much less complex task than a demodulator [1] and the complexity of these operations has therefore not been fully investigated. However, the DSP is fully optimized for tasks such as symbol shaping, shift operations and table look ups which are essential during demodulation.

9 Conclusions and further work

A DSP architecture especially optimized for low-power, low complexity baseband processing has been proposed. To obtain the necessary performance for repetitive signal processing tasks, concepts of both VLIW and SIMD architectures has been included in the DSP architecture. It has especially been focused on energy efficiency, since the radio protocols the processor is aimed at usually have very strict energy requirements. The use of an instruction buffer, storing two instructions at a minimum of added hardware complexity, greatly reduces the required amount of instruction fetches. By the use of shifted or multiplexed input words during convolution of word-sized data, the required number of memory accesses was minimized also for data access. Also, it has been focused on obtaining a minimal logic area, thereby reducing power consumption.

Flexibility has also been an important issue of the DSP architecture. A broad range of logical, arithmetic and shift operations are therefore supported.

The estimated metrics for area, current consumption and clock frequency for the DSP are summarized and compared to given constraint in table 15. The estimates are based on 0.13 micron silicon technology and the implementation of an IEEE 802.15.4 demodulator.

Table 15 - Achieved metrics

	Maximum current consumption	Typical current consumption	Maximum clock frequency	Area
Constraint	10mA	5mA	200MHz	40000 gates
Achieved	7,4mA	4,6mA	200MHz	38500 gates

As seen from the table, software implementation of the demodulator should be achievable for the proposed DSP core. However, the usability of a simpler algorithm for *Start of frame delimiter* detection should be investigated in order to decrease the maximum required clock frequency significantly. If the requirements during this mode of operation can not be simplified, a functional level accelerator for 2 bit correlation should be added to the architecture. Such enhancements will lead to a reduction of both maximum current consumption and clock frequency, but the resulting area may exceed the given constraint.

9.1 Further work

This thesis can be considered as a small step considering a possible commercializing of a software defined baseband processor. A great deal of verification will have to be done before an actual silicon realization of the DSP.

To provide further verification of the proposed DSP architecture, a system level model of the architecture can be developed. More accurate estimates of power consumption and computational requirements should be obtainable by implementing algorithms on a software model of the proposed DSP. Also, an instruction set simulator can be

implemented to verify the functionality of the programming model. An FPGA implementation can provide further verification and simulation before a possible ASIC realization of the DSP.

In this thesis, mainly the IEEE 802.15.4 standard has been considered for implementation on the DSP. Other possible standards should also be throughout investigated to analyze how well the architecture performs when implementing these standards.

10 Bibliography

[1] Roger Martinsen Koteng. Evaluation of SDR-implementation of an IEEE 802.15.4 Physical Layer. Master thesis. 2006.

[2] Software Defined Radio Forum. http://www.sdrforum.org.

[3] Walter Tuttlebee. Software Defined Radio – Enabling Technologies. John Wiley and Sons, LTD. 2002.

[4] Sen M. Kuo, Woon-Seng Gan. Digital Signal Processors. Prentice Hall. 2005.

[5] Dolphin Integration. http://www.dolphin.fr.

[5] Harry F. Jordan, Gita Alaghband. Fundamentals of Parallel Processing. 2003.

[6] Matthias H. Weiss, Gerhard P. Fettweis. Dynamic Codewidth Reduction for VLIW Instruction Set Architectures in Digital Signal Processors. Dresden University. 1998.

[7] William Stallings. Computer Organization and Architecture. Prentice Hall. 2003.

[8] Eric Tell. Design of Programmable Baseband Processors. Lindköping University. 2005.

[9] Ya-Lan Tsao, Wei-Hao Chen, WenSheng Cheng, Maw-Ching Lin, ShyhJye Jou. Hardware Nested Looping of Parameterized and Embedded DSP Core. IEEE. 2003.

[10] Digital ASIC Group, Lund University. Digital ASIC Design - A Tutorial on the Design Flow. 2005.

[11] S. Bourdel, P. Pannier, H. Barthélemy, and N. Dehaese. Low-cost solutions for 802.15.4 rf. Spread Spectrum Techniques and Applications. IEEE Eighth International Symposium. 2004.

[12] Michael Schulte, John Glossner, Suman Mamidi, Mayan Moudgill, and Stamatis Vassiliadis. A Low-Power Multithreaded Processor for Baseband Communication Systems. Sandbridge Technologies. 2004.

[13] Gerard J.M. Smit, Gerard K. Rauwerda. Reconfigurable Architectures for Adaptable Mobile Systems. University of Twente. 2004.

[14] John G. Proakis, Dimitris G. Manolakis. Digital Signal Processing. Prentice Hall. 1996.

[15] Robin Hoel. Feasibility Study. Internal Document. Chipcon AS. 2006.

[16] Ken Turkowski. Fixed-Point Trigonometri with CORDIC Iterations. Apple Computer. 1990.

[17] Synchronization and Channel Estimation in OFDM: Algorithms for Efficient Implementation of WLAN Systems.

[18] IEEE Standard for Information Technology. Part 802.15.4: Wireless medium access control (MAC) and physical layer (PHY) specifications for LR-WPANs. IEEE. 2003.

[19] Data Sheet. Libra Visa, 0.18-micron Chartered Library. Synopsis. 2004.

[20] Texas Instruments. Silicon Technologies, SR40 and GS30. http://www.ti.com

[21] Demetros K. Kostopoulos. An Algorithm for the computation of binary Logarithms. IEEE. 1991.

[22] Graham Petley. VLSI and ASIC Technology Standard Cell Library Design. http://www.vlsitechnology.org/

Appendix A – Estimated computational complexity

Table 16 shows the estimated cycle count for implementation of an IEEE 802.15.4 demodulator.

Table 16 - Estimated cycle count

Mode	Operation	# cycles for initialization and control	# cycles / output sample	# output samples	Total # cycles
Frequency offset correction – 1 symbol period	Channel filter	5	15/4	128	485
	RSSI	6	2/4	16	14
	Frequency offset estimation	3+23	128*9/8	1	170
	Additional control overhead	100			100
	Total #cycles	**769**			
SFD detection – 1 symbol period	Channel filter	5	15/4	128	485
	Frequency offset correction	3	5/8	128	83
	Matched filter	5	7/4	128	229
	Quantization	3	1/4	128	64
	Correlation	10	17+3/4	128	2272
	Additional control overhead	50			50
	Total #cycles	**3183**			
Packet reception - 1 symbol period	Channel filter	5	15/4	128	485
	Frequency offset correction	3	5/8	128	83
	Matched filter	5	7/4	128	229
	Quantization	4	6/4	128	196
	I branch correlation	5	5/4	32	40
	Take absolute value	3	1/4	32	11
	Find max/min	3	32	1	35
	Q branch correlation	5	5/4	4	10
	Additional control overhead	100			100
	Total #cycles	**1189**			

It is assumed a buffering of two symbol periods (64 memory locations), i.e. each of the operations can be performed for one symbol period at a time.

Appendix B – Estimated power consumption

Table 17 and 18 shows the estimated amount of memory accesses for implementation of an IEEE 802.15.4 demodulator. The analysis is based on demodulation of one symbol period during packet reception and SFD detection.

Table 17 - Estimated number of memory accesses during packet reception

Mode	Operation	#instruction accesses	#coefficient memory accesses	#data read accesses	#data write accesses
Packet reception **1 symbol period**	ADC interface	0	0	0	64
	Channel filter	8	112	128	32
	Frequency offset correction	6	0	32	32
	Matched filter	8	32	32	32
	Quantization	3*32+3	0	32	32
	I branch correlation	8	40	10	8
	Take absolute value	7	0	8	8
	Find max/min	7	0	8	1
	Q branch correlation	8	2	2	1
	Additional control overhead	100	0	100	100
Total # memory accesses		251	186	352	310
uA/MHz		33	33	24,7	19,3
Power consumption		0,52mA	0,39mA	0,55mA	0,38mA

Table 18 - Estimated number of memory accesses during SFD detection

Mode	Operation	#instruction accesses	#coefficient memory accesses	#data read accesses	#data write accesses
SFD detection 1 symbol period	ADC interface	0	0	0	64
	Channel filter	8	112	128	32
	Frequency offset correction	6	0	32	32
	Matched filter	8	32	32	32
	Quantization	3	0	32	32
	Correlation	8	512	512	32
	Take absolute value	7	0	32	32
	Find max/min	7	0	32	1
	Additional control overhead	50	0	50	50
Total # memory accesses		97	656	850	306
uA/MHz		33	33	24,7	19,3
Power consumption		0,20mA	1,39mA	1,32mA	0,37mA

It is assumed a buffering of two symbol periods (64 memory locations), i.e. each of the operations can be performed for one symbol period at a time. The power consumption of the memories are based on [5] and [19] for a 0.18um CMOS process.

Appendix C – Estimated area

Table 19 - Area estimates

Part of DSP architecture	Unit	Basic components	Component Area	Total area
Memory architecture	**RAM**	256x32bit asynchronous 2-port RAM	14500	14500
	ROM	512x32bit asynchronous ROM	4600	4600
	Memory organizing unit	2x 32bit 4:1 multiplexers 3x 32bit 2:1 multiplexers 1x 8bit 2:1 multiplexer 4x 8bit registers 2x 32bit registers	400 288 24 184 368	1264
	Write port multiplexer	1x 32bit 4:1 multiplexer	200	200
	Address generation units	7x12bit registers 3x12bit adders 1x12bit 7:1 multiplexer 1x12bit 5:1 multiplexer 1x12bit 4:1 multiplexer 2x10bit comparators	483 360 183 111 75 100	1312
	DMA controller	1x10bit 2:1 multiplexer 1x10bit 3:1 multiplexer Decoder logic	30 60 100	190
Datapath architecture	**MAC unit**	4x 8bit multipliers 4x 4x2bit multipliers 4x 20bit adders/subtractors 4x4 10 bit 2:1 multiplexers 4x2 20 bit 4:1 multiplexers	2176 560 1120 480 1000	5336
	CORDIC unit	24x 12bit adders/ subtractors 1x 8bit adder 1x 8bit register 3x 12bit registers 8x 8bit hardcoded constants 2x 2:1 8bit multiplexers Control logic and state counter	4032 80 46 207 100 48 100	4613
	Accumulators	4x 20bit registers	460	460
	Find max/min unit	4x 8bit subtractors	320	516

		3x 8 bit 2:1 multiplexers 1x 8 bit 5:1 multiplexer Logic	72 74 50	
	Shift and logic unit	4x8bit 8:1 multiplexers 4x 8bit 5:1 multiplexers Logic blocks	436 236 50	722
	ADC/DAC interface	20x8bit registers 4x4x8bit 2:1 multiplexers 32bit 4:1 multiplexer 2x8bit 2:1 multiplexers State logic	920 334 334 18 100	1736
Control path architecture	**Pipeline registers**	2x32 bit register 3x8bit registers 2x8bit 3:1 multiplexers 32bit 3:1 multiplexer	400 106 36 132	674
	Instruction decoder	Decoder logic and multiplexers	1000	1000
	Branch controller	Compare logic	100	100
	Hardware loop	5x 6bit registers 2x 12bit registers 1x 12bit 3:1 multiplexer 3x 5bit 2:1 multiplexers 2x 5bit comparators 2x 5bit decrementors States and logic	173 138 72 45 100 100 500	1128
Total logic area				**19251**

Various standard cell libraries was used to obtain equivalent gate counts of the various basic logic components, see [22] and [19] for further details considering these libraries. The use of ripple carry adders and *Non-Booth-recoded Wallace-tree* multipliers was assumed sufficient considering the slow clock frequencies the DSP should be able to run at. Area of the multipliers was obtained from [10]. Area of the memory modules was found from [5] and [19] and is based on low power memories in a 0.18um technology.